Вступление

STM32Cube – это оригинальная инициатива STMicroelectronics, позволяющая значительно повысить производительность труда разработчиков за счет сокращения усилий, времени и затрат на разработку. STM32Cube покрывает весь портфель STM32.

STM32Cube включает в себя

• STM32CubeMX, графический инструмент конфигурации программного обеспечения, который позволяет генерировать код инициализации C с использованием графических мастеров.

• Комплексная встроенная программная платформа, поставляемая для каждой серии (например, STM32CubeF4 для серии STM32F4)

– STM32Cube HAL, встроенное программное обеспечение уровня абстракции STM32, обеспечивающее максимальную переносимость по всему портфелю STM32,

–Low-layer API (LL), предлагающий быстрый легкий ориентированный на экспертов уровень, который ближе к аппаратному обеспечению, чем HAL. LL API доступны только для набора периферийных устройств.

– согласованный набор компонентов промежуточного программного обеспечения, таких как RTOS, USB, TCP / IP, графика,

– Все встроенные программные утилиты, поставленные с полным набором примеров.

Операционная система реального времени – это операционная система, оптимизированная для использования во встроенных приложениях / приложениях реального времени. Их основная цель – обеспечить своевременный и детерминированный ответ на события. Использование операционной системы реального времени позволяет писать приложения в виде набора независимых потоков, которые взаимодействуют с использованием очередей сообщений и семафоров.

Данное руководство пользователя предназначено для разработчиков, использующих прошивку STM32Cube на микроконтроллерах и микропроцессорах STM32. Он предоставляет полное описание того, как использовать компоненты прошивки STM32Cube с операционной системой реального времени (ОСРВ); это руководство пользователя также содержит описание ряда примеров, основанных на FreeRTOS ™, с использованием общих API-интерфейсов, предоставляемых слоем обертывания CMSIS-OS. В микропрограмме STM32Cube FreeRTOS ™ используется в качестве операционной системы реального времени с помощью универсального слоя упаковки CMSIS-OS, предоставляемого Arm®. Примеры и приложения, использующие FreeRTOS ™, могут быть напрямую перенесены в любую другую ОСРВ без изменения API высокого уровня, в этом случае необходимо изменить только оболочку CMSIS-OS. Обратитесь к примечаниям к выпуску пакета, чтобы узнать версию компонентов прошивки FreeRTOS ™ и CMSIS-RTOS, используемых с STM32Cube.

Этот документ применим ко всем устройствам STM32; однако для простоты устройства STM32F4xx и STM32CubeF4 используются в качестве эталонной платформы. Чтобы узнать больше о реализации примеров на вашем устройстве STM32, обратитесь к файлу readme, предоставленному в соответствующем пакете прошивки STM32Cube.

1 FreeRTOS ™

1.1 Обзор

FreeRTOS ™ – это класс RTOS, который спроектирован так, чтобы быть достаточно маленьким, чтобы работать на микроконтроллере или микропроцессоре, хотя его использование не ограничивается микроконтроллером и микропроцессором.

Микроконтроллер или микропроцессор – это небольшой процессор с ограниченными ресурсами в режиме реального времени, который включает в себя на одном кристалле сам процессор, постоянное запоминающее устройство (ПЗУ или флэш-память) для хранения исполняемой программы и оперативное запоминающее устройство (ОЗУ)) необходимые для запуска программ. Обычно программа выполняется непосредственно из постоянной памяти.

Микроконтроллеры и микропроцессор используются в глубоко встроенных приложениях (тех приложениях, где вы фактически никогда не видите сами процессоры или программное обеспечение, на котором они работают), которые обычно выполняют очень специфическую и специализированную работу. Ограничения по размеру и характер выделенного конечного приложения редко гарантируют использование полной реализации RTOS – или даже делают возможным использование полной реализации RTOS. Таким образом, FreeRTOS ™ предоставляет только основные функции планирования в реальном времени, взаимодействия между задачами, синхронизации и синхронизации. Это означает, что оно более точно описывается как ядро реального времени или исполнительный орган управления в реальном времени. Дополнительные функции, такие как интерфейс командной консоли или сетевые стеки, могут быть включены в дополнительные компоненты.

FreeRTOS ™ – это масштабируемое ядро для демонстрации в реальном времени, разработанное специально для небольших встроенных систем. Основные моменты включают

- FreeRTOS ™ – демонстрационный построитель с преимущественными преимуществами, возможностями совместной и гибридной конфигурации.
- Официальная поддержка 27 архитектур (считая ARM7 и Arm® Cortex®-M3 как одну архитектуру каждая).
- FreeRTOS-MPU поддерживает блок защиты памяти Arm® Cortex®-M3 (MPU).
- Разработанный, чтобы быть маленьким, простым и легким в использовании. Как правило, двоичное изображение ядра демонстрационного компоновщика будет находиться в диапазоне от 4K до 9K байтов.
- Очень переносимая структура кода, преимущественно написанная на C.
- Поддерживает как задачи, так и сопрограммы.
- Очереди, двоичные семафоры, счетные семафоры, рекурсивные семафоры и мьютексы для связи и синхронизации между задачами или между задачами и прерываниями.
- Мьютексы с приоритетным наследованием.
- Поддерживает эффективные программные таймеры.
- Мощное выполнение отслеживает функциональность.
- Опции обнаружения переполнения стека.

• Предварительно сконфигурированные демонстрационные приложения для выбранных одноплатных компьютеров, позволяющие работать «из коробки» и быстро обучаться.

• Бесплатная поддержка форума, или необязательная коммерческая поддержка и лицензирование.

• Нет программного ограничения на количество задач, которые могут быть созданы.

• Нет программного ограничения на количество приоритетов, которые могут быть использованы.

• Никаких ограничений на присвоение приоритета – более чем одной задаче может быть назначен один и тот же приоритет.

• Бесплатные инструменты разработки для многих поддерживаемых архитектур.

• Бесплатный исходный код встроенного программного обеспечения.

• Бесплатно.

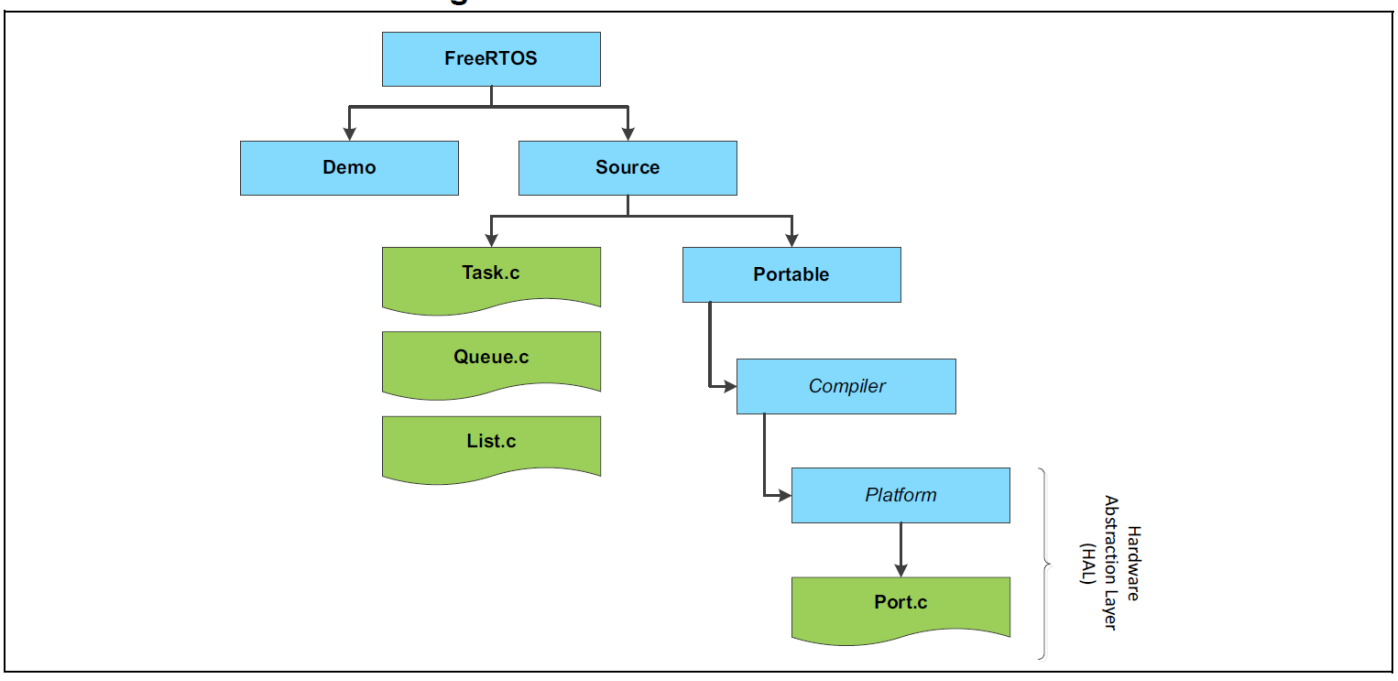
• Перекрестная разработка со стандартного хоста Windows.

Схема heap2 FreeRTOS ™ используется для управления распределением памяти, эта схема использует алгоритм наилучшего соответствия, позволяющий освобождать ранее выделенные блоки. Однако он не объединяет смежные свободные блоки в один большой блок. Общий объем доступной оперативной памяти устанавливается определением configTOTAL_HEAP_SIZE, которое определено в FreeRTOSConfig.h.

1.3 Организация источников FreeRTOS ™

Загрузка FreeRTOS ™ включает исходный код для каждого порта процессора и каждого демонстрационного приложения. Размещение всех портов в одной загрузке значительно упрощает распространение, но количество файлов может показаться пугающим. Структура каталогов, однако, очень проста, и ядро FreeRTOS ™ в реальном времени содержится всего в 4 файлах (дополнительные файлы требуются, если требуется программный таймер или сопрограмма).

Figure 2. FreeRTOS™ architecture



Основной код RTOS содержится в трех файлах, называемых tasks.c, queue.c и list.c., в каталоге FreeRTOS / Source. В одном и том же каталоге содержатся два необязательных файла с именами timers.c и croutine.c, которые реализуют программный таймер и сопрограмму. Каждая поддерживаемая архитектура процессора требует небольшого количества специфичного для архитектуры кода RTOS. Это переносимый уровень RTOS, расположенный в подкаталогах FreeRTOS / Source / Portable / [compiler] / [Architecture], где [компилятор] и [архитектура] – это компилятор, используемый для создания порта, и архитектура, на которой порт работает соответственно.

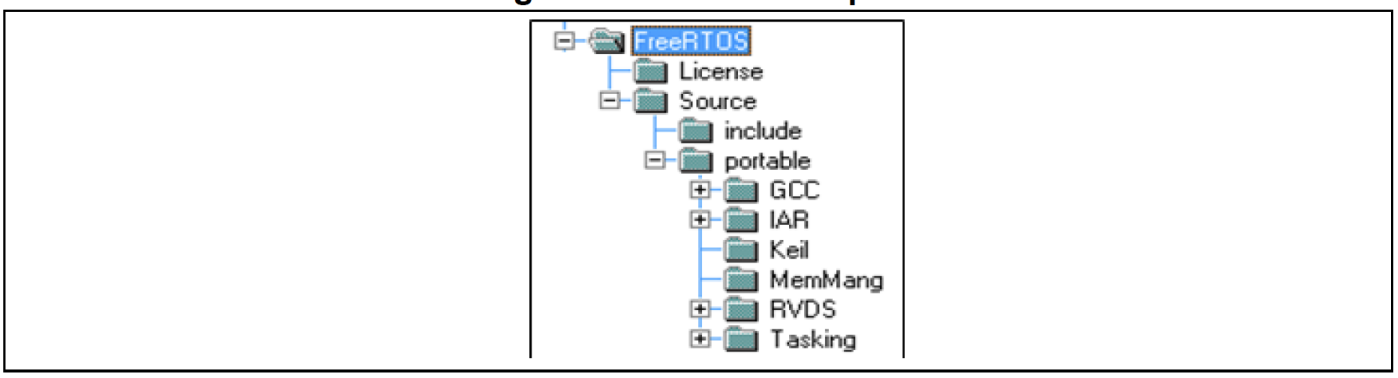
Схемы выборки кучи также расположены на переносном уровне. Различные примеры файлов heap_x.c находятся в каталоге FreeRTOS / Source / portable / MemMang.

1.4 Портирование FreeRTOS ™ на STM32

FreeRTOS ™ поддерживает следующие семейства процессоров ST: STM32 (Arm® (a) Cortex®-M0, Arm® Cortex®-M3 и Arm® Cortex®-M4F), STR7 (ARM7) и STR9 (ARM9) и может использоваться со следующими инструментами: IAR, Atollic® TrueStudio®, GCC, Keil®, Rowley CrossWorks.

Промежуточное программное обеспечение STM32Cube FatFs работает на 32-битных микроконтроллерах STM32 на базе процессора -M.

Figure 3. FreeRTOS™ port



1.5 FreeRTOS™ API

Table 1. FreeRTOS™ API

APIs categories	API
Task creation Создание задачи	– xTaskCreate
	– vTaskDelete
Task control Управление задачами	– vTaskDelay
	– vTaskDelayUntil
	– uxTaskPriorityGet
	– vTaskPrioritySet
	– vTaskSuspend
	– vTaskResume
	– xTaskResumeFromISR
	– vTaskSetApplicationTag
	– xTaskCallApplicationTaskHook

Task utilities Задача утилит	– xTaskGetCurrentTaskHandle
	– xTaskGetSchedulerState
	– uxTaskGetNumberOfTasks
	– vTaskList
	– vTaskStartTrace
	– ulTaskEndTrace
	– vTaskGetRunTimeStats
Kernel control Контроль ядра	– vTaskStartScheduler
	– vTaskEndScheduler
	– vTaskSuspendAll
	– xTaskResumeAll
Queue management Управление очередью	– xQueueCreate
	– xQueueSend
	– xQueueReceive
	– xQueuePeek
	– xQueueSendFromISR
	– xQueueSendToBackFromISR
	– xQueueSendToFrontFromISR
	– xQueueReceiveFromISR
	– vQueueAddToRegistry
	– vQueueUnregisterQueue
Semaphores Семафоры	– vSemaphoreCreateBinary
	– vSemaphoreCreateCounting
	– xSemaphoreCreateMutex
	– xSemaphoreTake
	– xSemaphoreGive
	– xSemaphoreGiveFromISR

1.6 FreeRTOS ™ управление памятью

Четыре примера схемы распределения ОЗУ включены в загрузку исходного кода FreeRTOS ™ (V2.5.0 и далее). Они используются различными демонстрационными приложениями по мере необходимости. В следующих подразделах описаны доступные схемы, когда они должны использоваться, и выделены демонстрационные приложения, демонстрирующие их использование.

Каждая схема содержится в отдельном исходном файле (heap_1.c, heap_2.c, heap_3.c и heap_4.c соответственно), который может находиться в каталоге Source / Portable / MemMang. Другие схемы могут быть добавлены при необходимости.

Схема 1 – heap_1.c

Это самая простая схема из всех. Она не позволяет освобождать память после ее выделения, но, несмотря на это, подходит для удивительно большого числа приложений.

Алгоритм просто подразделяет один массив на более мелкие блоки по мере выполнения запросов к оперативной памяти. Общий размер массива устанавливается определением configTOTAL_HEAP_SIZE, которое определено в FreeRTOSConfig.h. Эта схема:

• может использоваться, если ваше приложение никогда не удаляет задачу или очередь (никаких вызовов vTaskDelete () или vQueueDelete () никогда не производится).

• всегда детерминирован (для возврата блока всегда требуется одинаковое количество времени).

• используется демонстрационными приложениями PIC, AVR и 8051 – поскольку они не динамически создают или удаляют задачи после вызова vTaskStartScheduler ().

heap_1.c подходит для множества небольших систем реального времени, при условии, что все задачи и очереди создаются до запуска ядра.

Схема 2 – heap_2.c

Эта схема использует алгоритм наилучшего соответствия и, в отличие от схемы 1, позволяет освободить ранее выделенные блоки. Однако он не объединяет смежные свободные блоки в один большой блок.

Опять же, общий объем доступной оперативной памяти устанавливается определением configTOTAL_HEAP_SIZE, которое определено в FreeRTOSConfig.h.

Эта схема:

• может использоваться, даже когда приложение многократно вызывает vTaskCreate () / vTaskDelete () или vQueueCreate () / vQueueDelete () (вызывая множественные вызовы pvPortMalloc () и vPortFree ()).

• не следует использовать, если выделяемая и освобождаемая память имеет произвольный размер – это будет иметь место только в том случае, если каждая из удаляемых задач имеет разную глубину стека или удаляемые очереди имеют разную длину.

• может привести к проблемам фрагментации памяти, если ваше приложение создаст блоки очередей и задач в непредсказуемом порядке. Это было бы маловероятным для почти всех приложений, но

• следует иметь в виду.

• не является детерминированным – но также не является особенно неэффективным.

heap_2.c подходит для большинства небольших систем реального времени, которые должны динамически создавать задачи.

Схема 3 – heap_3.c

Это просто оболочка для стандартных функций malloc () и free (). Это делает их безопасными. Эта схема:

• Требуется компоновщик для настройки кучи и библиотека компилятора для обеспечения реализации malloc () и free ().

• Не является детерминированным.

• Вероятно, значительно увеличит размер кода ядра.

• ***Используется демонстрационным приложением для ПК (одноплатный компьютер x86).***

Схема 4 – heap_4.c

Эта схема использует алгоритм первого соответствия и, в отличие от схемы 2, объединяет смежные блоки свободной памяти в один большой блок (он включает алгоритм коалесценции).

Общий объем доступного пространства кучи устанавливается configTOTAL_HEAP_SIZE – который определен в FreeRTOSConfig.h.

Функция API xPortGetFreeHeapSize () возвращает общий объем пространства кучи, который остается нераспределенным (что позволяет оптимизировать настройку configTOTAL_HEAP_SIZE), но не предоставляет информацию о том, как нераспределенная память фрагментируется на более мелкие блоки.

Эта реализация:

- может использоваться, даже когда приложение многократно удаляет задачи, очереди, семафоры и мьютексы.
- гораздо менее вероятно, чем реализация heap_2, в результате чего пространство кучи будет сильно фрагментировано на несколько небольших блоков – даже когда выделенная и освобожденная память имеет произвольный размер.
- Не является детерминированным, но гораздо более эффективен, чем большинство стандартных реализаций библиотеки malloc.

heap_4.c особенно полезена для приложений, которые хотят использовать схемы выделения памяти переносимого уровня непосредственно в коде приложения (а не просто косвенно, вызывая функции API, которые сами вызывают pvPortMalloc () и vPortFree ()).

1.7 FreeRTOS ™ управление низким энергопотреблением

Обычно уменьшается энергопотребление микроконтроллера / микропроцессора, на котором работает FreeRTOS ™, с помощью перехватчика задач в режиме ожидания, чтобы перевести микроконтроллер / микропроцессор в состояние с низким энергопотреблением. Экономия энергии, которая может быть достигнута с помощью этого простого способа, ограничена необходимостью периодически выходить, а затем повторно входить в состояние низкого энергопотребления для обработки прерываний от тиков. Кроме того, если частота прерывания по тику слишком высока, затрачиваемая энергия и время на вход и выход из состояния низкого энергопотребления для каждого тика будут перевешивать любые потенциальные выигрыши в энергосбережении для всех режимов, кроме самых легких.

Режим безключевого режима FreeRTOS ™ останавливает периодическое прерывание тиков в течение периодов простоя (периодов, когда нет потоков приложений, которые могут выполнять), а затем корректирует значение счетчика тиков RTOS при перезапуске прерывания тиков.

Остановка тикового прерывания позволяет микроконтроллеру / микропроцессору оставаться в состоянии энергосбережения до тех пор, пока не произойдет прерывание или пока ядру RTOS не придет время перевести поток в состояние готовности.

1.8 Конфигурация FreeRTOS ™

Существует ряд настраиваемых параметров, позволяющих адаптировать ядро FreeRTOS ™ к вашему конкретному приложению. Эти элементы находятся в фай-

ле с именем FreeRTOSConfig.h. Каждое демонстрационное приложение, включенное в загрузку исходного кода FreeRTOS ™, имеет свой собственный файл FreeRTOSConfig.h. Вот типичный пример

```
/* Ensure stdint is only used by the compiler, and not the assembler. */
/* Убедитесь, что stdint используется только компилятором, а не ассемблером. */
#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.       Cortex-M конкретные определения. */
#ifdef __NVIC_PRIO_BITS
/* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
/* __BVIC_PRIO_BITS будет указано при использовании CMSIS. */
#define configPRIO_BITS __NVIC_PRIO_BITS
#else
#define configPRIO_BITS 4 /* 15 priority levels     15 уровней приоритета */
#endif
/* The lowest interrupt priority that can be used in a call to a «set priority» function. */
/* Самый низкий приоритет прерывания, который можно использовать при вызове
функции «установить приоритет». */
#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 */
/ * Наивысший приоритет прерывания, который может использоваться любой
подпрограммой обработки прерываний, которая выполняет вызовы для прерывания
безопасных функций API FreeRTOS * /
```

```
#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. */
/ * Приоритеты прерываний, используемые самим уровнем порта ядра.
Они являются общими для всех портов Cortex-M и не зависят от каких-либо
конкретных библиотечных функций. * /
#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. */
/ * Определения, которые отображают обработчики прерываний порта FreeRTOS
на их стандартные имена CMSIS. * /
#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 */
/ * ВАЖНО: Это определение ДОЛЖНО быть закомментировано при использовании
с прошивкой STM32Cube, чтобы предотвратить перезапись SysTick_Handler,
определенной в STM32Cube HAL * /
/* #define xPortSysTickHandler SysTick_Handler */
```

Примечание:

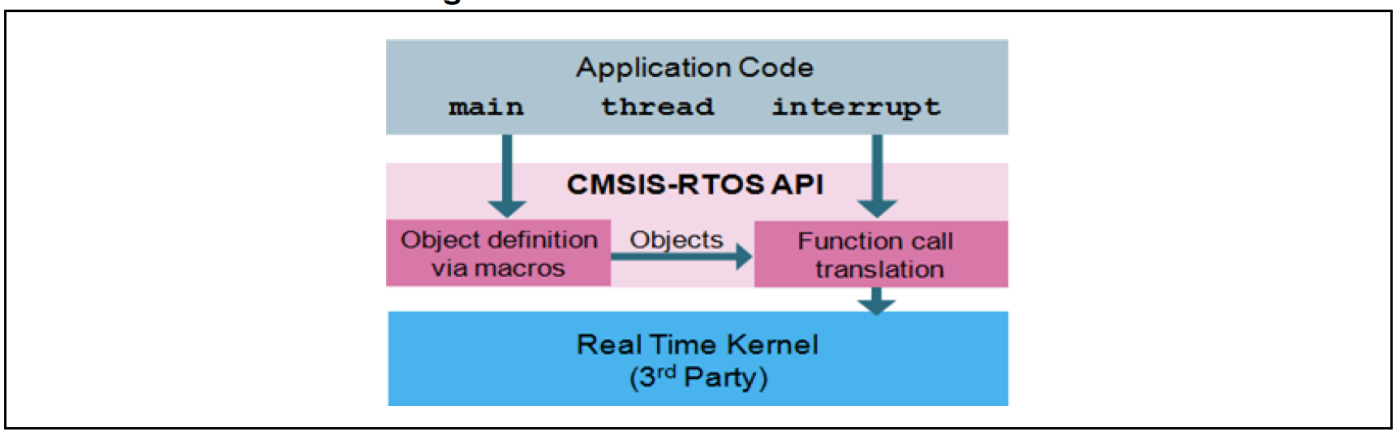
SVC_Handler и PendSV_Handler должны быть удалены из файлов stm32f4xx_it.c / .h при работе с FreeRTOS ™, чтобы избежать дублирования определения

2 CMSIS-RTOS модуль

2.1 Обзор

CMSIS-RTOS – это общий API для операционных систем реального времени. Он обеспечивает стандартизированный интерфейс программирования, который переносим на многие ОСРВ и поэтому позволяет использовать шаблоны программного обеспечения, промежуточное программное обеспечение, библиотеки и другие компоненты, которые могут работать в поддерживаемых системах ОСРВ.

Figure 5. CMSIS-RTOS architecture



Этот модуль представлен файлами cmsis_os.c / h, расположенными в следующем репозитории «Middlewares \ Third_Party \ FreeRTOS \ CMSIS_RTOS».

Типичная реализация API CMSIS-RTOS взаимодействует с существующим ядром реального времени. API-интерфейс CMSIS-RTOS предоставляет следующие атрибуты и функции:

• Имена функций, идентификаторы и параметры являются описательными и простыми для понимания. Функции являются мощными и гибкими, что уменьшает количество функций, предоставляемых пользователю.

• Управление потоками позволяет определять, создавать и контролировать потоки.

• Программы обработки прерываний (ISR) могут вызывать многие функции CMSIS-RTOS. Когда функция CMSIS-RTOS не может быть вызвана из контекста ISR, она отклоняет вызов.

• Три различных типа событий потока поддерживают связь между несколькими потоками и / или ISR:

– Сигналы: флаги, которые могут использоваться для сигнализации о конкретных условиях для потока. Сигналы могут быть изменены в ISR или установлены из других потоков.

– Message: 32-битное значение, которое может быть отправлено потоку или ISR. Сообщения помещаются в очередь. Тип сообщения и размер очереди

– Определены в дескрипторе.

– Mail: это блок памяти фиксированного размера, который можно отправить потоку или ISR. Почта буферизуется в очереди и обеспечивается выделение памяти. Тип почты и размер очереди

– Определены в дескрипторе.

• Управление Mutex и управление семафорами включены.

• Время ЦП может быть запланировано • d со следующими функциями:

– Параметр тайм-аута включен во многие функции CMSIS-RTOS, чтобы избежать блокировки системы. Когда указан тайм-аут, система ждет, пока ресурс не станет доступным или не произойдет событие. Во время ожидания другие темы запланированы.

– Функция osDelay переводит поток в состояние WAITING на указанный период времени.

– Общая функция osWait ожидает событий, которые назначены потоку.

– osThreadYield обеспечивает совместное переключение потоков и передает выполнение другому потоку с таким же приоритетом.

API-интерфейс CMSIS-RTOS разработан для опционального включения многопроцессорных систем и / или защиты доступа через блок защиты памяти (MPU) Arm® Cortex®-M.

В некоторых реализациях RTOS потоки могут выполняться на разных процессорах, поэтому очереди Mail и Message могут находиться в ресурсах общей памяти.

API CMSIS-RTOS поощряет индустрию программного обеспечения развивать существующие реализации RTOS. Объекты ядра определяются и доступны с помощью макросов. Это позволяет дифференцировать. Реализации ОСРВ могут быть разными и оптимизированными в разных аспектах по отношению к процессорам Arm® Cortex®-M. Дополнительные функции могут быть, например:

- Общая функция ожидания; т.е. с поддержкой временных интервалов.
- Поддержка блока защиты памяти Arm® Cortex®-M (MPU).
- Нулевая копия почтовой очереди.
- Поддержка многопроцессорных систем.
- Поддержка контроллера DMA.
- Детерминированное переключение контекста.
- Круглое переключение контекста.
- Устранение тупиковой ситуации, например, с инверсией приоритета.
- Нулевая задержка прерывания с помощью инструкций Arm® Cortex®-M3 / M4 LDEX и STEX.

2.2 CMSIS-RTOS API

В следующем списке приведен краткий обзор всех API-интерфейсов CMSIS-RTOS:

Table 2. CMSIS-RTOS API

Модуль	API	Описание
Kernel information and control. **Ядро информации и контроля.**	**osKernelInitialize**	Initialize the RTOS kernel. **Инициализировать ядро RTOS.**
	osKernelStart	Start the RTOS kernel. **Запустить ядро RTOS.**
	osKernelRunning	Query if the RTOS kernel is running. **Запросить, работает ли ядро RTOS.**
	osKernelSys Tick (*)	Get RTOS kernel system timer counter. **Получить счетчик таймера системы ядра RTOS.**
	osKernelSys TickFrequency (*)	RTOS kernel system timer frequency in Hz. **Частота таймера системы ядра ОСРВ в Гц.**
	osKernelSys TickMicroSec (*)	Convert microseconds value to RTOS kernel system timer value. **Преобразование значения микросекунд в значение системного таймера ядра RTOS.**

1. Модули или API, отмеченные (*), являются необязательными.

Модуль	API	Описание
Thread management: Define, create and control thread functions. **Управление потоками: определение, создание и управление функциями потоков.**	**osThreadCreate**	Start execution of a thread function. **Начать выполнение функции потока.**
	osThreadTerminate	Stop execution of a thread function. **Остановить выполнение функции потока.**
	osThreadYield	Pass execution to next ready thread function. **Передать выполнение следующей функции готового потока.**
	osThreadGetId	Get the thread identifier to reference this thread. **Получить идентификатор потока для ссылки на этот поток.**
	osThreadSetPriority	Change the execution priority of a thread function. **Изменить приоритет выполнения функции потока.**
	osThreadGetPriority	Obtain the current execution priority of a thread function. **Получить текущий приоритет выполнения функции потока.**
Generic wait functions:Wait for a time period or unspecified events. **Общие функции ожидания: ожидание периода времени или неуказанных событий.**	**osDelay**	Wait for a specified time. **Подождать в течение указанного времени.**
	osWait (*)	Wait for any event of the type Signal, Message, or Mail. **Дождаться любого события типа Сигнал, Сообщение или Почта.**
	Mutex, от mutual exclusion — «взаимное исключение») — аналог одноместного семафора, служащий в программировании для синхронизации одновременно выполняющихся потоков.	*Мьютекс отличается от семафора тем, что только владеющий им поток может его освободить, т. е. перевести в отмеченное состояние.*

Модуль	API	Описание
Timer management(1): Create and control timer and timer callback functions. **Управление таймером (1): создание и управление функциями таймера и обратного вызова таймера.**	**osTimerCreate**	Define attributes of the timer callback function. **Определить атрибуты функции обратного вызова таймера.**
	osTimerStart	Start or restart the timer with a time value. **Запустить или перезапустить таймер со значением времени.**
Signal management: Control or wait for signal flags. **Управление сигналами: контроль или ожидание сигнальных флагов.**	**osSignalSet**	Set signal flags of a thread. **Установка сигнальных флагов потока.**
	osSignalClear	Reset signal flags of a thread. **Сброс сигнальных флагов потока.**
	osSignalClear	Suspend execution until specific signal flags are set. **Приостановка выполнения, пока не будут установлены определенные сигнальные флаги.**
Mutex management(1): Synchronize thread execution with a Mutex. **Управление Mutex (1): Синхронизировать выполнение потоков с Mutex.**	**osMutexCreate**	Define and initialize a mutex. **Определить и инициализировать мьютекс.**
	osMutexWait	Obtain a mutex or Wait until it becomes available. **Получить мьютекс или подождать, пока он не станет доступным.**
	osMutexRelease	Release a mutex. **Освободить мьютекс.**
	osMutexDelete	Delete a mutex. **Удалить мьютекс.**

Модуль	API	Описание
Semaphore management(1): Control access to shared resources. **Управление семафорами (1): Контроль доступа к общим ресурсам.**	**osSemaphoreCreate**	Define and initialize a semaphore. **Определить и инициализировать семафор.**
	osSemaphoreWait	Obtain a semaphore token or Wait until it becomes available. **Получить жетон семафора или дождаться его появления.**
	osSemaphoreRelease	Release a semaphore token. **Освободить жетон семафора.**
	osSemaphoreDelete	Delete a semaphore. **Удалить семафор.**
Memory pool management(1): Define and manage fixed-size memory pools. **Управление пулом памяти(1): определение и управление пулами памяти фиксированного размера.**	**osPoolCreate**	Define and initialize a fix-size memory pool. **Определить и инициализировать пул памяти фиксированного размера.**
	osPoolAlloc	Allocate a memory block. **Выделить блок памяти.**
	osPoolCAlloc	Allocate a memory block and zero-set this block. **Выделить блок памяти и заполнить этот блок нулями.**
	osPoolFree	Return a memory block to the memory pool. **Вернуть блок памяти в пул памяти.**
Message queue management(1): Control, send, receive, or wait for messages. **Управление очередью сообщений(1): контроль, отправка, получение или ожидание сообщений.**	**osMessageCreate**	Define and initialize a message queue. **Определить и инициализировать очередь сообщений.**
	osMessagePut	Put a message into a message queue. **Поместите сообщение в очередь сообщений.**
	osMessageGet	Get a message or suspend thread execution until message arrives. **Получить сообщение или приостановить выполнение потока, пока сообщение не прибудет.**

Модуль	API	Описание
Mail queue management(1): Control, send, receive, or wait for mail. **Управление почтовой очередью (1): контроль, отправка, получение или ожидание почты.**	**osMailCreate**	Define and initialize a mail queue with fix-size memory blocks. **Определить и инициализировать почтовую очередь с блоками памяти фиксированного размера.**
	osMailAlloc	Allocate a memory block. **Выделить блок памяти.**
	osMailCAlloc	Allocate a memory block and zero-set this block. **Выделить блок памяти и заполнить этот блок нулями.**
	osMailPut	Put a memory block into a mail queue. **Поместить блок памяти в почтовую очередь.**
	osMailGet	Get a mail or suspend thread until mail arrives. **Получить почту или приостановить поток, пока почта не прибудет.**
	osMailFree	Return a memory block to the mail queue. **Вернуть блок памяти в почтовую очередь.**

3. Приложения FreeRTOS ™ (FreeRTOS™ applications)

Пакет STM32CubeF4 FreeRTOS ™ поставляется с несколькими приложениями, использующими наборы API стека.

Приложения делятся на две категории

Table 3. FreeRTOS™ application categories

Категории	Приложения
Getting started (basic) **Начало работы (базовое)**	Thread creation example **Пример создания потока**
	Semaphore between threads example **Пример семафора между потоками**
	Semaphore from ISR example **Семафор из примера ISR**
	Mutexes example **Пример мьютекса**
	Queues example **Пример очереди**

Категории	Приложения
Features Обособленные	Timer example **Пример таймера**
	Low-power example **Пример с низким энергопотреблением**

3.1 Пример создания потока

Приложение реального времени, использующее ОСРВ, может быть структурировано как набор независимых потоков. Каждый поток выполняется в своем собственном контексте без случайной зависимости от других потоков в системе или самого планировщика RTOS. В любой момент времени может выполняться только один поток в приложении, и планировщик RTOS отвечает за решение, каким потоком это должно быть.

Цель этого примера - объяснить, как создавать потоки с использованием CMSIS-RTOS на основе API FreeRTOS ™.

В этом примере реализованы два потока с одинаковым приоритетом, которые выполняются в периодическом цикле. Ниже подробно о каждом потоке выполнения.

Поток 1: этот поток переключает LED1 каждые 200 мс на 5 секунд, а затем приостанавливает себя, через 5 секунд поток 2 возобновляет выполнение потока 1, который переключает LED1 каждые 400 мс в течение следующих 5 секунд.

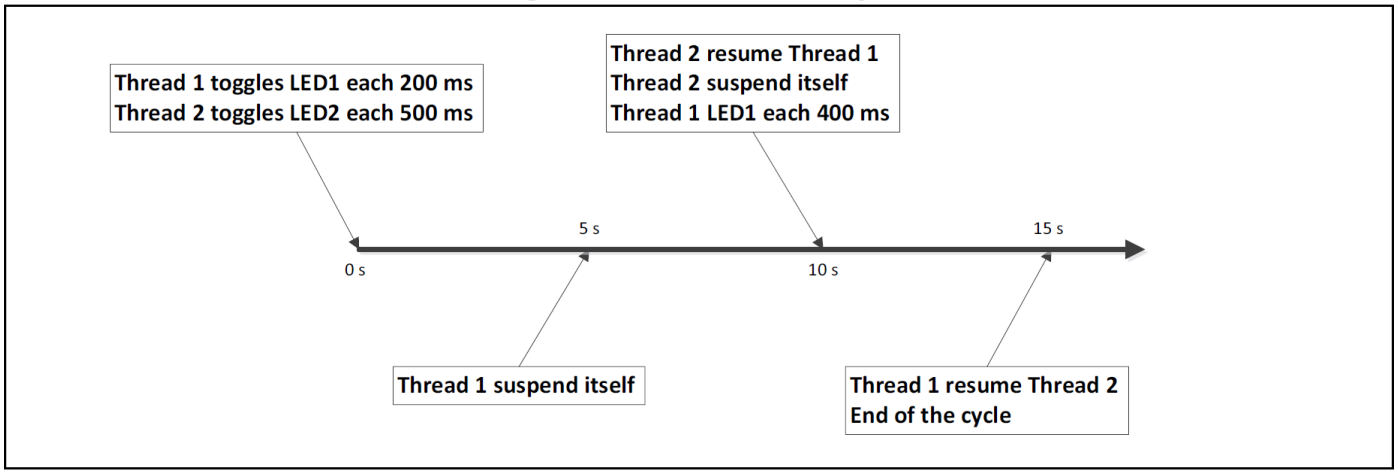
Описание создания потока:

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

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

Поток 2: этот поток переключает LED2 каждые 500 мс на 10 секунд, затем он сам приостанавливается. Поток 1 возобновит выполнение потока 2 через 5 секунд

Figure 6. Thread example



Пример использования:

- Сборка и программирование кода приложения во флэш-памяти STM32.
- Запустите пример и убедитесь, что светодиоды переключаются, как описано на рисунке 6.

3.2 Примеры семафоров

Семафоры используются как для взаимного исключения, так и для синхронизации.

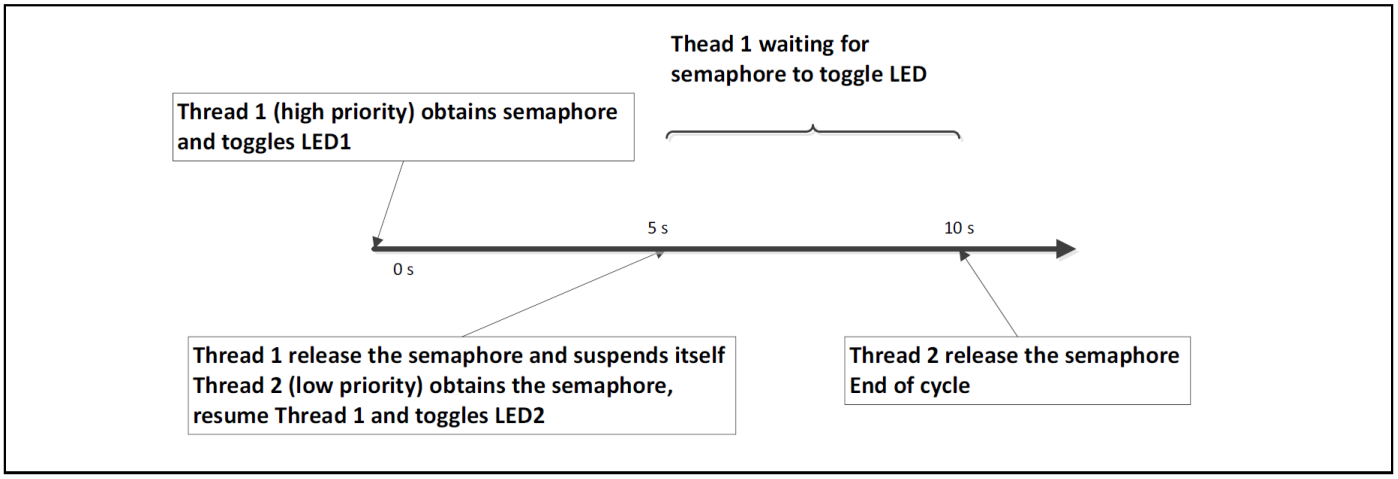
3.2.1 Семафор между потоками

Цель этого примера - объяснить, как использовать семафоры через CMSIS-RTOS на основе API FreeRTOS ™.

В этом примере реализованы два потока с разными приоритетами, которые совместно используют семафор для переключения светодиодов, следуя более подробной информации о выполнении примера.

1. Поток 1 с более высоким приоритетом получает семафор и переключает LED1 на 5 секунд.
2. Поток 1 освобождает семафор и приостанавливает себя.
3. Поток с низким приоритетом теперь может выполняться, он получает семафор и возобновляет выполнение потока 2.
4. Поскольку он имеет более высокий приоритет, поток 1 будет пытаться получить семафор, но он блокируется, потому что семафор уже занят потоком с низким приоритетом.
5. Поток 2 переключит светодиод 2 на 5 секунд, перед тем как выпустить семафор и начать новый цикл.

Figure 7. Semaphore example



Описание создания семафора:

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

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

Пример использования:

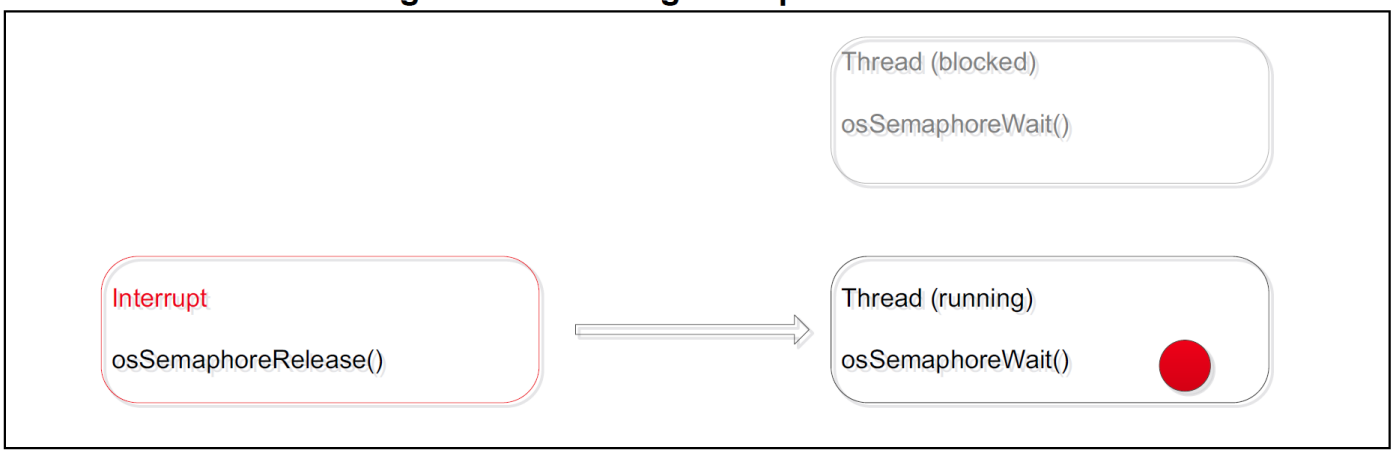
1. Скомпилируйте и запрограммируйте код приложения во флэш-память STM32.
2. Запустите пример и убедитесь, что светодиоды переключаются, как описано на рисунке 7.

3.2.2 Семафор от ISR

Этот пример демонстрирует, как использовать семафоры из прерываний.

Он состоит из основного потока, ожидающего, пока семафор переключит светодиод. Семафор освобождается, когда STM32 генерирует прерывание после нажатия пользователем кнопки KEY оценочной платы

Figure 8. Obtaining semaphore from ISR



Пример использования:

1. Создайте и запрограммируйте код приложения во флэш-память STM32.
2. Запустите пример и убедитесь, что индикатор LED1 переключается при нажатии кнопки KEY оценочной платы.

3.3 Пример мьютексов

Мьютексы - это двоичные семафоры, которые включают механизм наследования приоритетов. В то время как двоичные семафоры являются лучшим выбором для реализации синхронизации (между задачами или между задачами и прерываниями), мьютексы - лучший выбор для реализации простого взаимного исключения.

В этом примере создаются три потока с разными приоритетами, которые обращаются к одному и тому же мьютексу. Следующий

1. Поток с высоким приоритетом выполняется первым, захватывает мьютекс и спит в течение короткого периода времени, чтобы обеспечить выполнение потоков с более низким приоритетом.
2. Поток со средним приоритетом пытается получить доступ к мьютексу, выполняя блокировку «ожидание». Этот поток блокируется, когда мьютекс уже занят потоком с высоким приоритетом. Он не разблокируется до тех пор, пока поток с высоким приоритетом не освободит мьютекс, и фактически не будет работать, пока поток с высоким приоритетом не приостановит себя.
3. Поток с низким приоритетом вращается вокруг узкой петли, пытаясь получить мьютекс с неблокирующим вызовом. Как поток с самым низким приоритетом, он не сможет успешно получить мьютекс, пока не будут приостановлены потоки с высоким и средним приоритетом.
4. Поток с высоким приоритетом возвращает мьютекс перед его приостановкой.
5. Поток со средним приоритетом получает мьютекс, все, что он делает, это возвращает мьютекс до того, как он также приостановит свою работу. На этом этапе оба потока с высоким и средним приоритетом приостановлены.

6. Поток с низким приоритетом получает мьютекс, он сначала возобновляет оба приостановленных потока перед возвратом мьютекса, в результате чего поток с низким приоритетом временно наследует наивысший приоритет потока

Описание создания Mutex:

```
/* Define the mutex      Определить мьютекс */
osMutexDef(osMutex);
/* Create the mutex      Создать мьютекс */
osMutexId osMutex = osMutexCreate(osMutex(osMutex));
```

Пример использования:

1. Создайте и запрограммируйте код приложения во флэш-память STM32.
2. При работе в режиме отладки добавьте следующие переменные в часы рельного времени отладчика:

HighPriorityThreadCycles, MediumPriorityThreadCycles и LowPriorityThreadCycles; эти три переменные должны оставаться равными. LED1, LED2 и LED4 должны бесконечно переключаться, а LED3 включается в случае ошибки.

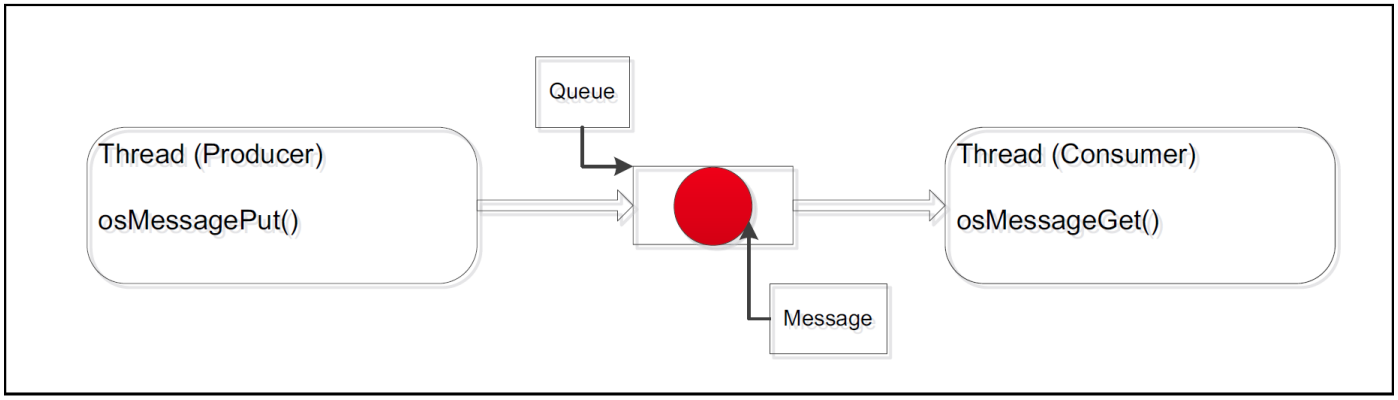
3.4 Пример очередей

Очереди являются основной формой межзадачных коммуникаций. Они могут использоваться для отправки сообщений между задачами, а также между прерываниями и задачами. В большинстве случаев они используются в качестве потоковобезопасных буферов FIFO (First In First Out) с новыми данными, отправляемыми в конец очереди, хотя данные также можно отправлять на фронт.

В этом примере создаются два потока, которые отправляют и получают возрастающее число в / из очереди. Один поток действует как производитель, а другой - как потребитель.

Потребитель имеет более высокий приоритет, чем производитель, и он настроен на блокировку чтения из очереди. В очереди есть место только для одного элемента, как только производитель отправляет сообщение в очередь, потребитель разблокирует, выгрузит производителя и удалит элемент.

Figure 9. Queue process



Описание создания очереди:

```
/* Define a queue with “QUEUE_SIZE” items of 2 bytes */
/* Определить очередь с элементами «QUEUE_SIZE» по 2 байта */
osMessageQDef(osqueue, QUEUE_SIZE, uint16_t);
```

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

Пример использования:

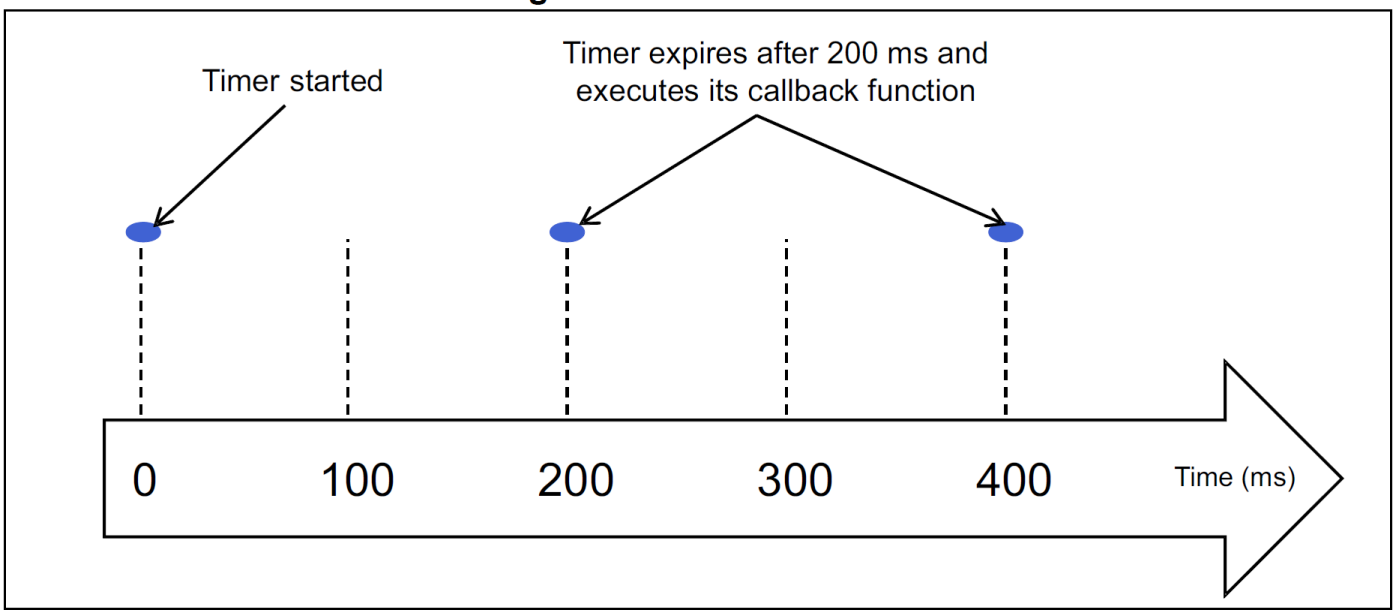
1. Создайте и запрограммируйте код приложения во флэш-память STM32.
2. Запустите пример и убедитесь, что LED1 переключается для каждого правильного полученного сообщения, иначе LED3 будет переключаться.

3.5 Пример таймера

Таймер позволяет в будущем выполнять функцию в установленное время. Функция, выполняемая таймером, называется функцией обратного вызова таймера. Время между запуском таймера и выполнением его функции обратного вызова называется периодом таймера. Проще говоря, функция обратного вызова таймера выполняется, когда истекает период таймера.

В этом примере показано, как использовать таймеры API-интерфейса RTOS CMSIS на основе API FreeRTOS ™, создавая периодический таймер, который каждые 200 мс вызывает функцию обратного вызова для переключения светодиода 1 оценочной платы.

Figure 10. Periodic timer



Описание создания периодического таймера:

```
/* Define a timer with “osTimerCallback” as callback process */
/* Определите таймер с «osTimerCallback» в качестве процесса обратного вызова */
osTimerDef(LEDTimer, osTimerCallback);

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

Пример использования:

1. Создайте и запрограммируйте код приложения во флэш-память STM32.
2. Запустите пример и убедитесь, что LED1 переключается каждые 200 мс (истечение таймера)

Примечание.

Чтобы использовать программные таймеры FreeRTOS ™, добавьте «timers.c» в рабочую область вашего проекта.

3.6 Пример с низким энергопотреблением

В этом примере показано, как запустить FreeRTOS ™ в режиме низкого энергопотребления с использованием устройств STM32 (для получения дополнительной информации о режиме низкого энергопотребления FreeRTOS ™ см. Раздел 1.7).

Встроенная функциональность без галочки в режиме ожидания (низкое энергопотребление) включается путем определения configUSE_TICKLESS_IDLE как 1 в FreeRTOSConfig.h

В этом примере два потока и очередь создаются со следующими функциями:

• Первый поток «RxThread» блокирует очередь, ожидая данных, и включает светодиод каждый раз, когда данные принимаются (включается и снова выключается), прежде чем снова вернуться к блокировке в очереди.

• Второй поток «TxThread» повторно входит в заблокированное состояние на 500 мс. При выходе из заблокированного состояния «TxThread» отправляет сообщение через очередь в «RxThread» (в результате чего «RxThread» выходит из заблокированного состояния и включает светодиод).

Когда два потока заблокированы, ядро прекращает тиковое прерывание и переводит STM32 в режим пониженного энергопотребления (спящий режим), чтобы снизить энергопотребление.

В таблице 4 представлено потребление энергии, измеренное на устройствах STM32F4 в контексте примера, описанного выше.

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. Вывод

В этом руководстве пользователя объясняется, как интегрировать компоненты промежуточного программного обеспечения FreeRTOS ™ в драйверы STM32Cube HAL.

Был описан ряд примеров, чтобы помочь пользователям разрабатывать приложения с CMSIS-RTOS API на основе операционной системы FreeRTOS ™.

5 Часто задаваемых вопросов

Как портировать FreeRTOS ™ на разные ядра Arm® Cortex®-M?

Чтобы перенести FreeRTOS ™ на нужный продукт Arm® Cortex®-M, вам необходимо импортировать «port.c» из правильной папки. Например, если микроконтроллер имеет ядро Arm® Cortex®-M0 с инструментом IAR, вы должны получить файл port.c из репозитория «FreeRTOS \ Source \ portable \ IAR \ ARM_CM0».

Сколько ROM / RAM использует FreeRTOS ™?

Это зависит от вашего компилятора, архитектуры и конфигурации ядра RTOS. Обычно для самого ядра RTOS требуется от 5 до 10 Кбайт пространства ПЗУ.

Использование ОЗУ увеличивается, если увеличивается количество созданных потоков или очередей.

Как установить тактовую частоту процессора?

Тактовая частота процессора определяется configCPU_CLOCK_HZ в FreeRTOSConfig.h, в микропрограмме STM32CubeF4 она предоставляется SystemCoreClock, представляющей тактовую частоту HCLK (шина AHB), это значение устанавливается при настройке тактовой частоты RCC с помощью вызова функции SystemClock_Config ().

Как установить приоритеты прерывания?

Любая подпрограмма обработки прерываний, использующая функцию API-интерфейса RTOS, должна иметь свой приоритет, вручную установленный на значение, которое численно равно или превышает значение configMAX_SYSCALL_INTERRUPT_PRIORITY в файле FreeRTOSConfig.h.

Это гарантирует, что логический приоритет прерывания равен или меньше значения настройки configMAX_SYSCALL_INTERRUPT_PRIORITY.

Как использовать часы, отличные от SysTick, для генерации тикового прерывания?

Пользователь может при желании предоставить свой собственный источник прерывания от тиков, генерируя прерывание из таймера, отличного от SysTick:

- Обеспечить реализацию vPortSetupTimerInterrupt (), которая генерирует прерывание с частотой, указанной константой FreeTTConConfig.h configTICK_RATE_HZ.
- Установите xPortSysTickHandler () в качестве обработчика прерывания по таймеру и убедитесь, что xPortSysTickHandler () не сопоставлен с SysTick_Handler () в FreeRTOSConfig.h или переименован в SysTick_Handler () в port.c.

Как включить режим щекотки?

Режим тикков FreeRTOS ™ (низкое энергопотребление) позволяет снизить энергопотребление микроконтроллера за счет перехода в спящий режим и остановки периодического прерывания по тику. Эта функциональность включена путем определения configUSE_TICKLESS_IDLE как 1 в FreeRTOSConfig.h

Режим ожидания щекотки можно включить, когда для генерации прерывания щекотки используется таймер, отличный от SysTick. Пользователь должен добавить следующие действия к описанным в предыдущем вопросе:

- Установите configUSE_TICKLESS_IDLE равным 2 в FreeRTOSConfig.h.
- Определите portSUPPRESS_TICKS_AND_SLEEP (), как описано на странице документации на сайте FreeRTOS ™.

Revision 3 28-Oct-2019