Операционная система реального времени

Операционные системы реального времени (ОСРВ) – являются операционными системами (ОС) предназначенные для использования в приложениях, запросы которых поступают в реальном времени.

Ключевой характеристикой ОСРВ, является количество времени, необходимое для принятия и завершения задачи приложения, так же значение задержек (variability is jitter). Гораздо важнее насколько быстро и предсказуемо ОСРВ обрабатывает новые задачи, чем количество задач, выполненных за определенный период времени.

Жесткие ОСРВ (hard) имеют меньшую величину задержек, чем ОСРВ с допусками (soft).

Жесткое реальное время требует, чтобы время отклика никогда не превышало срок исполнения, в случае если срок истекает, а отклик не был выработан, происходит **фатальный отказ системы**.

Реальное время с допусками допускает флуктуации времени отклика, при условии, что среднее время отклика равно сроку исполнения. Система работает хуже, но **сохраняет работоспособность** даже если срок исполнения иногда просрочен.

Популярные ОСРВ

VxWorks - (WinRiver)

Integrity - (Green Hills)

FreeRTOS (Freertos.org)

uC/OS (Micrium Technologies Corporation)

CMX-RTX (CMX Systems)

embOS (Segger)

Это не ОСРВ:

Linux

Android

Windows

STM32F4-Discovery + Atollic + FreeRTOS

Используемые ресурсы ядра:

- Системный таймер (SysTick) – генерирует время системы (отсчеты времени)
- Два указателя стека: MSP, PSP

Вектора прерываний (эти три вектора должны быть удалены из файла ***stm32f4xx_it.c***):

- **SVC** – system service call (как SWI в ARM7)
- **PendSV** – pended system call (переключение контекста)
- **SysTick** – System Timer

Файловая система -> пожалуйста обратитесь к следующему слайду. Наиболее важные файлы ***port.x*** и ***portmacro.x,*** *которые* строго зависят от ядра и используемого ПО.

Настройка системы выполнена через ***FreeRTOSConfig.h***

FreeRTOS структура исходных файлов

File / header Directory	Назначение
croutine.c / croutine.h .\Source .\Source\include	Программы определяющих функций. Эффективность в 8 и 16 битных архитектурах. В 32-х битной архитектуре предусматривается использование задач.
heap_x.c .\Source\portable\MemMang	Функции управления памятью (выделение свободного сегмента памяти, три различных подхода в файлах heap_1, heap_2 и heap_3).**Heap_2.c** самый эффективный
list.c / list.h .\Source .\Source\include	Список применений, используемых диспетчером задач (scheduler).
port.c / portmacro.h .\Source\portable\gcc\ARM_CM3	Низкоуровневые функции поддерживающие SysTick таймер, context switch, управления прерываниями на аппаратном уровне – зависят от платформы (ядро и компилятор). В основном написаны на ассемблере. В portmacro.h находятся определения portTickType и portBASE_TYPE
queue.c / queue.h .\Source .\Source\include	Очереди, семафоры, мьютексы
tasks.c / task.h .\Source .\Source\include	Функции задач и определение утилит
FreeRTOS.h .\Source\include	Конфигурационный файл, который содержит все исходники FreeRTOS
FreeRTOSConfig.h	Конфигурация системы FreeRTOS, тактирование и настройка прерываний

FreeRTOS – структура кода

- Файлы:
  - ***FreeRTOS.h*** – главный заголовочный файл основного функционала FreeRTOS
  - ***Task.h*** – если задачи будут использованы в программе (99% случаев)
  - ***Semphr.h*** – если семафоры/очереди будут использованы в программе
- В начале main() все аппаратные настройки должны быть выполнены (тактирование (clock), GPIO настройка). В нашем примере это функция **prvSetupHardware()**.
- Следующий шаг — создание базовых компонентов программы:
  - **tasks** – периодически повторяемые участки кода (LCD_control, GPIO_control)
  - **semaphores** – используется для синхронизации между задачами, а также между задачами и прерываниями
  - **queues** – используется для передачи данных между задачами
- Последнее действие в main() запускает диспетчер задач (scheduler):

  vTaskStartScheduler()
- Если МК пропустит начало диспетчера, значит стек переполнен
- Код задачи (**task**) должен быть заключен в бесконечный цикл, как:

```
for(;;)
{
    // task code
}
```

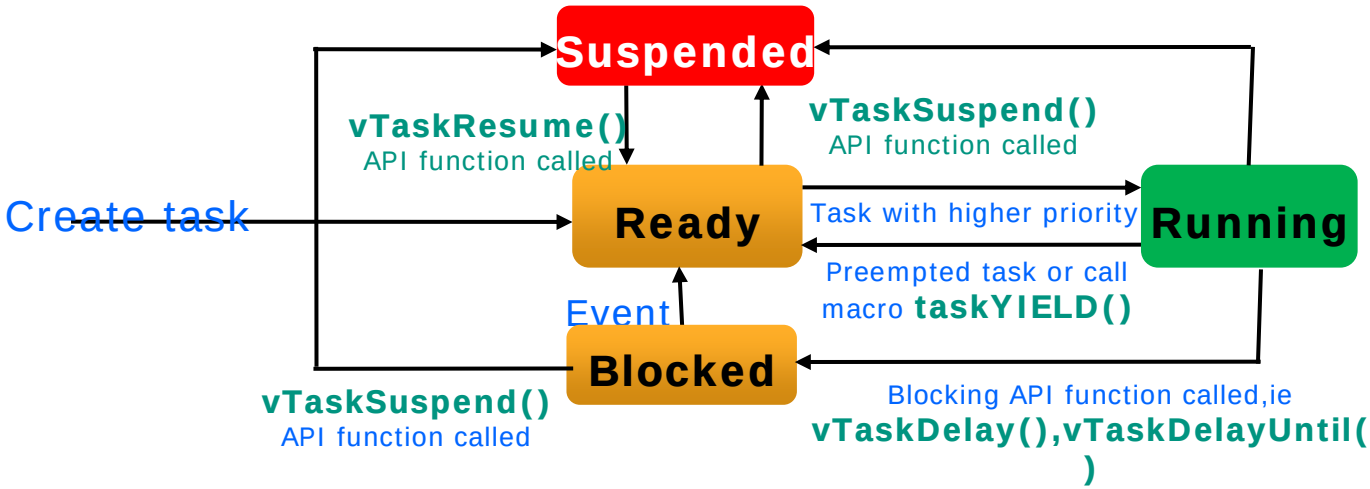
- Между специализированными задачами, запускается задача IDLE (если это возможно в соответствии со схемой приоритета задач)

FreeRTOS – задачи

- Задача - это C функция **void vTask(void *pvParameters)** которая может быть использована для запуска любого количества отдельных случаев функцией **xTaskCreate()**. В нашем примере мы создадим 4 различных задачи для управления светодиодами в зависимости от аргумента номера светодиода и параметров задержки.

- Созданая задача может находиться в одном из четырех состояний:

Running : выполняется в настоящее время. Только **одна задача может быть в режиме Run** одновременно.
Ready : готова к выполнению (она не Blocked или Suspended), пока другая задача равнозначного или высшего приоритета находится в режиме Run.
Blocked : ждет либо временное (вызывается SysTick), либо внешнее событие (обозначенное семафором или очередью).
Suspended : недоступна для диспетчера. Задачи могут **входить** и **выходить** из приостановленного **(suspended)** состояния только через вызовы API ОСРВ (**Task_Suspend()** и **Task_Resume()** соответственно).



Каждая задача имеет свою область стека и приоритет (который может быть изменен)

- Для освобождения оперативной памяти рекомендуется удалять задачи, если их больше не планируется использовать (функции: **xTaskDelete**(обработчик задачи или NULL если мы хотим удалить текущую задачу)

FreeRTOS – семафоры

- При использоваании FreeRTOS в основе семафоров механизм очереди
- Существует три типа семофоров во FreeRTOS:
  - Binary – простой механизм вкл/выкл
  - Counting – многоканальный вызов и завершение(counts multiple give and multiple take)
  - Recursive
- Семафоры служат для синхронизации задач с другими событиями системы (особенно прерываниями)
- Ожидание семафора равнозначно процедуре wait(), задачи в режиме block не тратят время RTOS
- Семафоры должны быть созданы перед использованием: **vSemaphoreCreateBinary()**
- Вкл. семафор = вызов семафора может осуществиться из другой задачи или подпрограммы прерывания

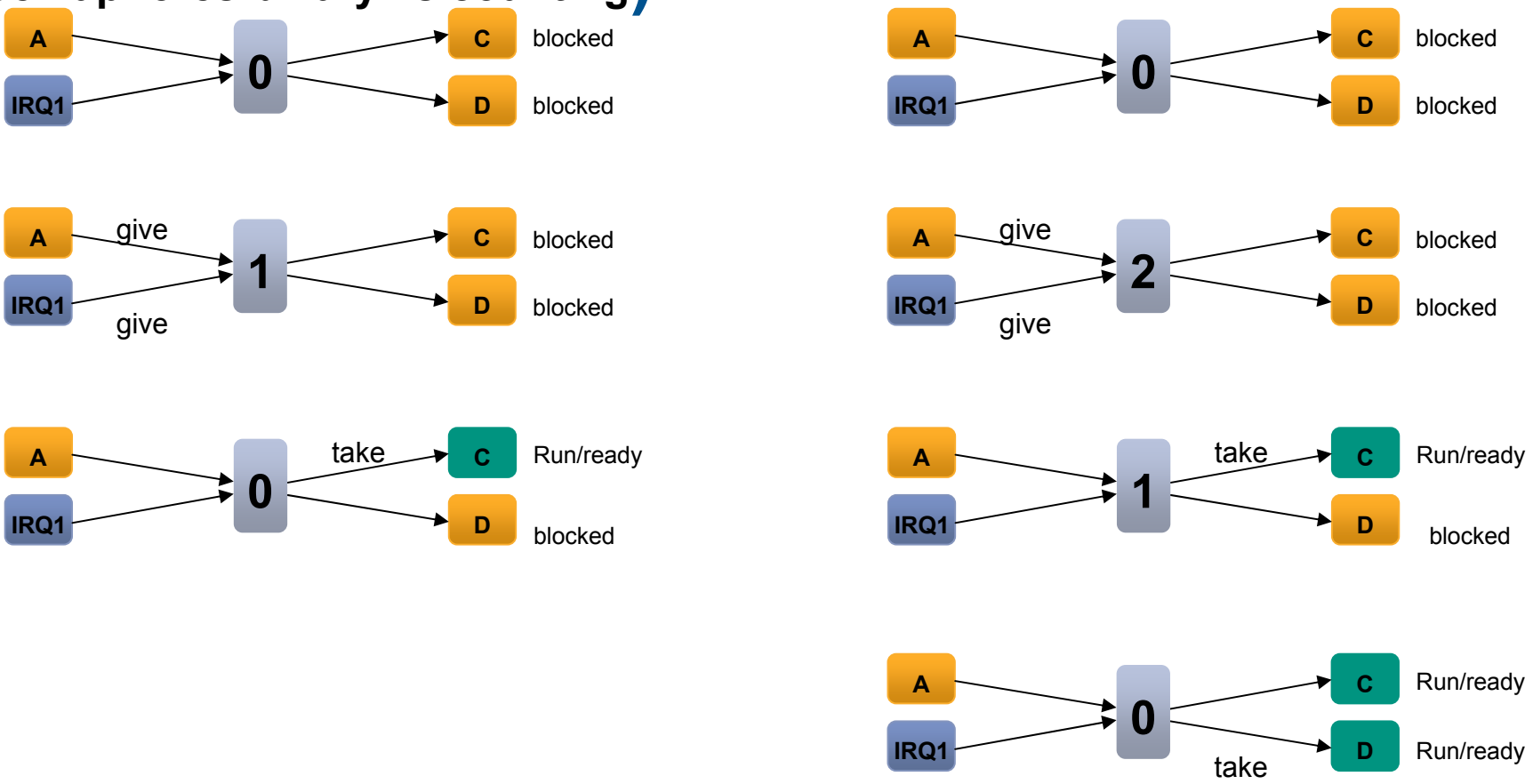
xSemaphoreGive() или **xSemaphoreFiveFromISR()**

- Выкл. семафор = ответить на семафор можно сделать из задачи

xSemaphoreTake()

Двоичный семафор против счетчика

(Semaphores binary vs counting)



Binary **Counting**

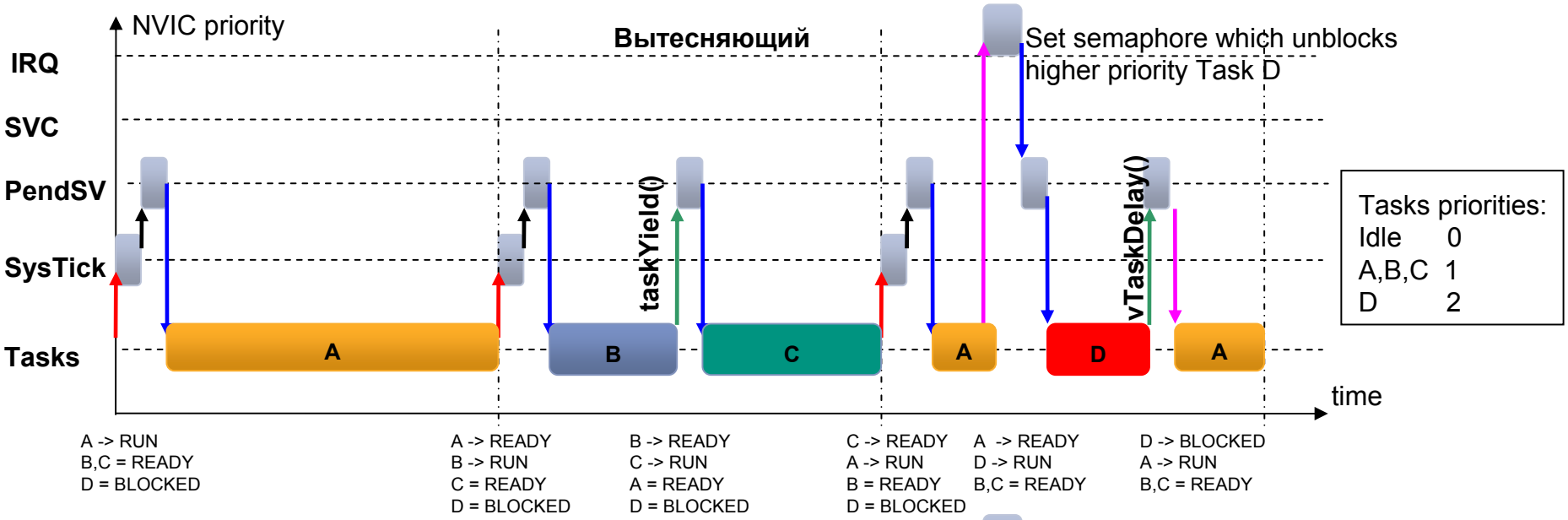
FreeRTOS – очереди

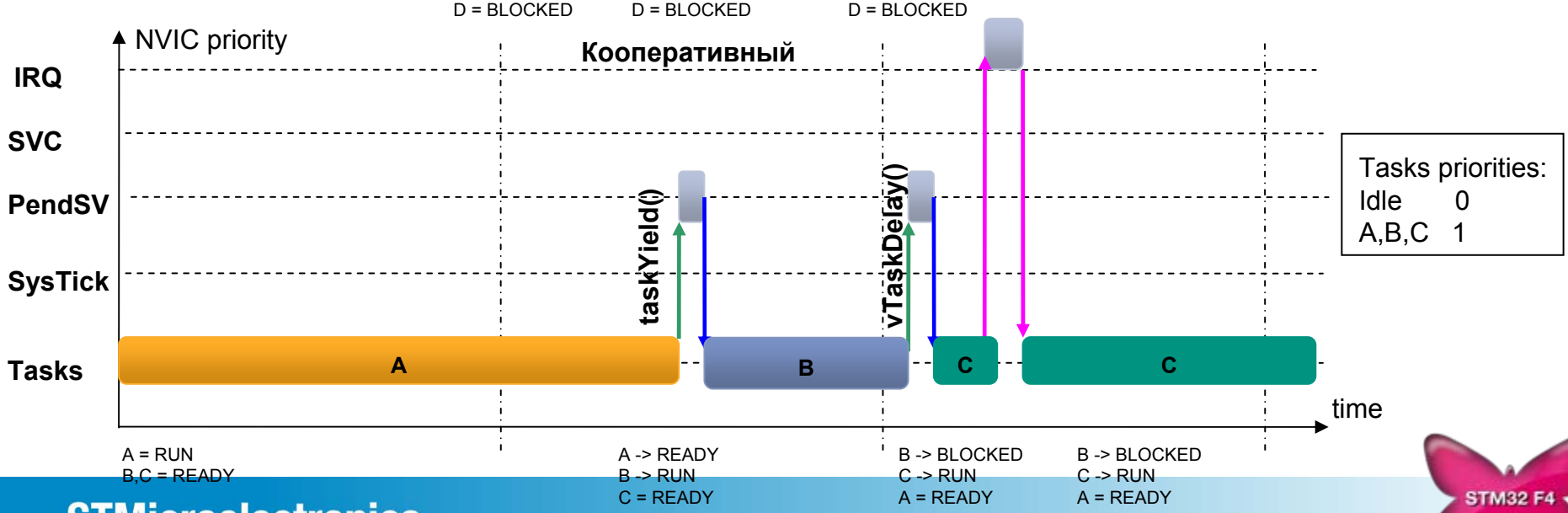
- Очереди - это разновидность конвейера для передачи данных между задачами RTOS
- Очередь должна быть создана перед использованием при помощи функции **xQueueCreate()**
- Для передачи данных в очередь используются функции:
  - **xQueueSendToBack()** -> FIFO (first in – first out) порядок передачи данных

  xQueueSendToFront() ->LIFO (last in – first out) порядок передачи данных
- Функции для получения данных из очереди:
  - **xQueueReceive()** -> выгрузить данные из очереди
  - **xQueuePeek()** -> забрать данные из очереди, не удаляя их
- Все данные переданные в очередь должны быть одного типа, объявленных во время создания очереди. Это может быть обычная переменная или структура.
- Длина очереди так же определяется во время ее создания. Она высчитывается в зависимости от количества данных передаваемых через очередь.
- Пример:

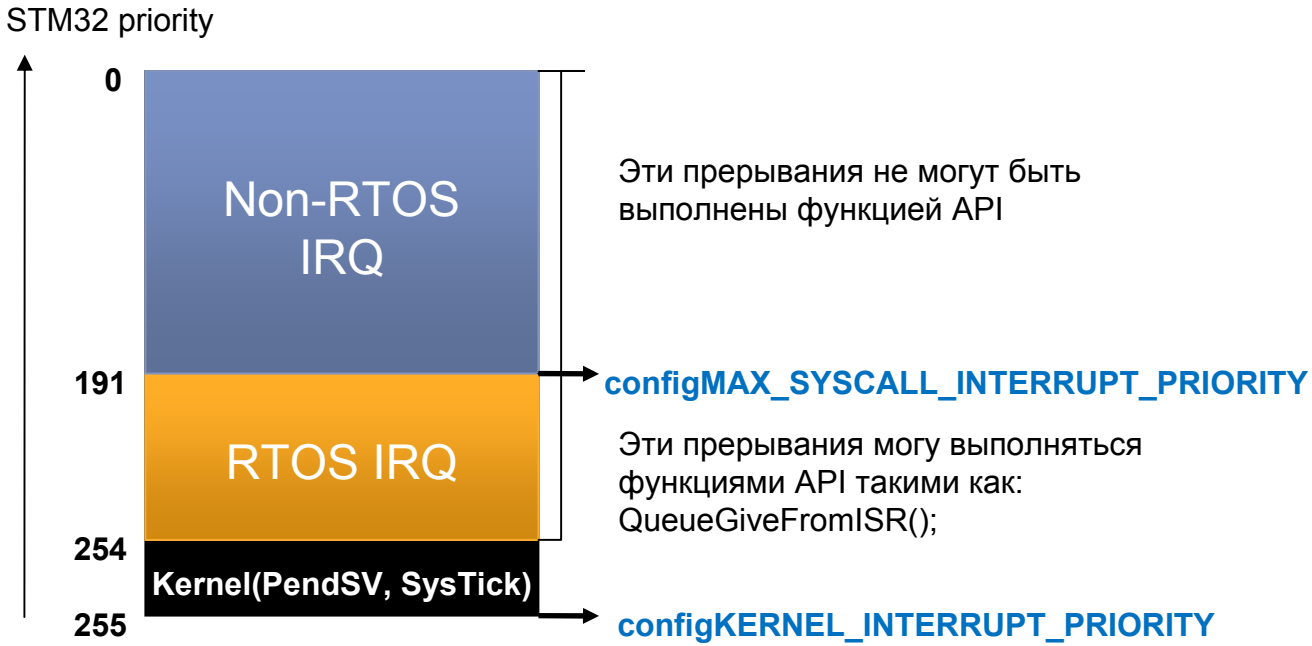
```
/* Data structure to be sent via queue */
typedef struct
{
    int who;            //Sender ID
    char data[10];      //message
}xDataFromTasks;

/* Create queue which will contain maximum 10 objects type XDataFromTasks */
xQueue=xQueueCreate(10,sizeof(xDataFromTasks))
```

Переключение между задачами





Конфигурация контроллера векторов прерываний NVIC

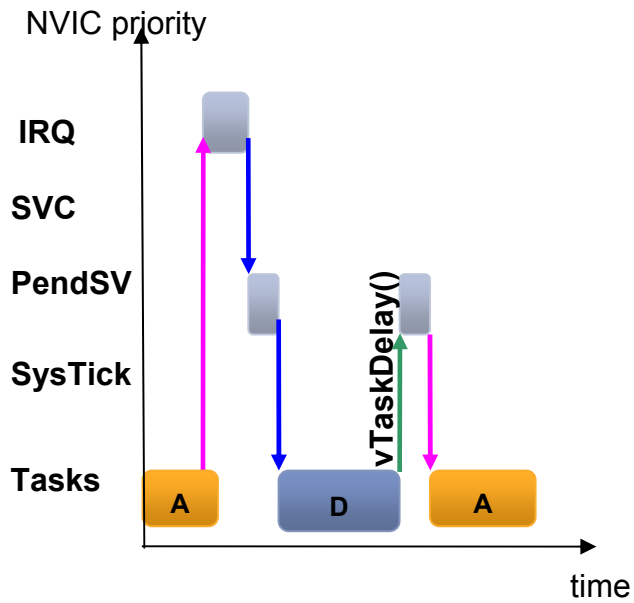


- Ядро FreeRTOS и процедуры прерываний (PendSV, SysTick) имеют наименьший приоритет прерывания (255). Определено в FreeRTOSConfig.h (**configKERNEL_INTERRUPT_PRIORITY**)
- Существует группа прерываний которые могут использоваться совместно с FreeRTOS API через вызов функций. Максимальный уровень которых (основан на позиции в таблице векторов прерываний) основан **configMAX_SYSCALL_INTERRUPT_PRIORITY**
- Возможно использовать настраиваемые(nested) прерывания.

FreeRTOS – обработка прерываний

- Специальные API функции выполняются в процедурах прерываний (IRQ procedures). Все имена этих функций имеют приставку **FromISR**. Ниже представлен список наиболее важных:
  - **xSemaphoreGiveFromISR**(semaphore, *hp_task)
  - **xQueueSendFromISR**(...,*hp_task)
  - **xQueueSendToBackFromISR**(...,*hp_task)
  - **xQueueSendToFrontFromISR**(...,*hp_task)
  - **xQueueReceiveFromISR**(...,*hp_task)

Все эти функции имеют свои эквиваленты без использования прерываний.



- Единственное отличие для программиста в добавлении аргумента *hp_task. Это указатель на переменную которая используется для определения операции с очередью или семафором из прерывания для разблокирования задачи с приоритетом выше, чем у текущей задачи. Если этот параметр pdTRUE, переключение контекста должно быть вызвано ядром, перед тем как прерывание произойдет.

FreeRTOS – защита от переполнения стека

- Проверить сколько стека использует задача –‘ватерлиния’ для стека.

  Специальная функция:

  uxTaskGetStackHighWaterMark(xTaskHandle xTask);

  После вызова с обработчиком задачи (task handler) в качестве аргумента, выводит минимальное количество свободного места стека для задачи xTask.

- Функция захвата переполнения стека – функция вызываемая ядром при переполнении стека. Она должна быть включена пользователем. Определение должно выглядеть так:

  vApplicationStackOverflowHook(xTaskHandle *pxTask, signed char *pcName);

- Дополнительные механизмы проверки стека (определяется в ***FreeRTOSConfig.h***)
  - Метод 1 (**configCHECK_FOR_STACK_OVERFLOW** set to **1**)

  быстрый старт, может пропустить некоторые случаи переполнения стека
  - Метод 2 (**configCHECK_FOR_STACK_OVERFLOW** set to **2**)

  медленный, но более надежный

FreeRTOS упражнения – описание задачи

Создайте новую рабочую область Atollic IDE и импортируйте архив проекта:

Ex2 – FreeRTOS_on_STM32F4-Discovery.zip

Исходный код поврежден в нескольких местах (только в ***main.c***).

Будьте осторожны, если поврежденный участок кода прошел компиляцию без ошибки, то во время выполнения программы может произойти сбой (**HardFault)** или программа не будет подавать признаков жизни.

Обратите внимание на структуру кода (главный цикл и задачи в соответствии с правилами программирования, используя FreeRTOS)

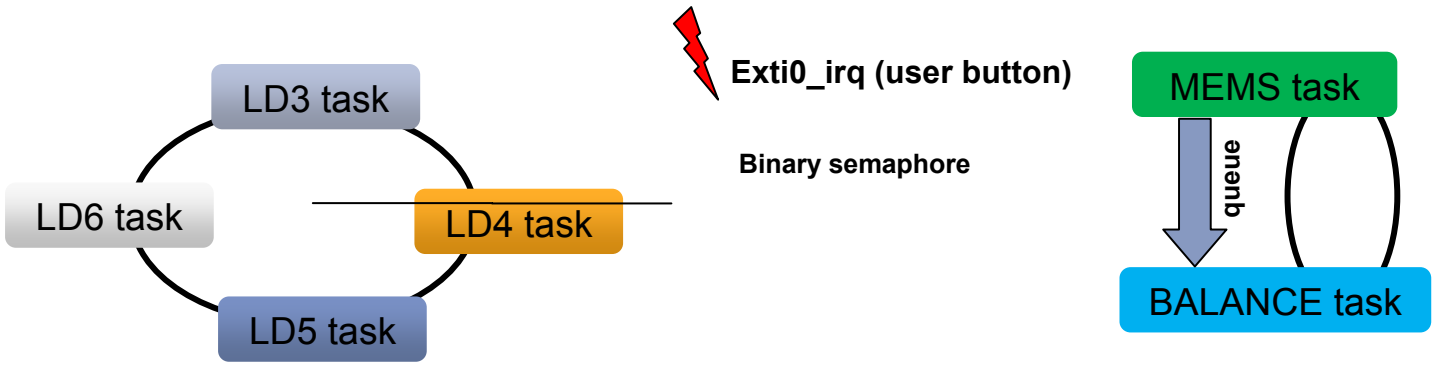
Задача заключается в том, чтобы обнаружить и устранить все проблемы, не позволяющие программе работать на плате STM32F4-Discovery -> обратитесь к следующей странице за пояснением

Задачи можно увидеть во вкладке задач (Task Tag), с приставкой EXERCISE

FreeRTOS упражнение – корректная работа

Приложение работает следующим образом:

- После сброса все 4 LEDs: LD3..6 мигают с различной частотой (MEMS task и BALANCE task приостановлены)
- При нажатии User button, приложение начинает работать как детектор уровня (LEDs мигают если плата расположена не горизонтально –работает от данных MEMS – U5. (LED tasks: LD3, LD4, LD5, LD6 приостановлены)
- Повторное нажатие User button переключит приложение в начальное состояне (4LEDs blinking with different speed).



FreeRTOS упражнение – решение

- Добавьте бесконечный цикл внутри функции **vLEDTask()** чтобы не оказаться в исключении HardFault:

```
for( ;; ) {
   STM_EVAL_LEDToggle((Led_TypeDef)LED[0]);
   vTaskDelay(LED[1]/portTICK_RATE_MS);
}
```

- Унификация приоритетов всех задач. Это выглядит как:

```
xTaskCreate(vSWITCHTask,( signed portCHAR * ) "SWITCH", configMINIMAL_STACK_SIZE, NULL,
                        tskIDLE_PRIORITY, NULL );
```

- Добавьте еще три vLEDTasks с различными аргументами (LED номер и частоту обновления):

```
xTaskCreate( vLEDTask, ( signed portCHAR * ) "LED4", configMINIMAL_STACK_SIZE,
    (void *)LEDS[1],tskIDLE_PRIORITY, &xLED_Tasks[1] );
xTaskCreate( vLEDTask, ( signed portCHAR * ) "LED5", configMINIMAL_STACK_SIZE,
    (void *)LEDS[2],tskIDLE_PRIORITY, &xLED_Tasks[2] );
xTaskCreate( vLEDTask, ( signed portCHAR * ) "LED6", configMINIMAL_STACK_SIZE,
    (void *)LEDS[3],tskIDLE_PRIORITY, &xLED_Tasks[3] );
```

- Управляйте новыми LED tasks в **vSWITCHTask()**. Пример:

i=0 (LED tasks)	i=1 (MEMS+BALANCE tasks)
vTaskSuspend(xBALANCE_Task);	**vTaskSuspend(xLED_Tasks[0]);**
TIM_Cmd(TIM4, *DISABLE);*	**vTaskSuspend(xLED_Tasks[1]);**
vTaskSuspend(xMEMS_Task);	**vTaskSuspend(xLED_Tasks[2]);**
prvLED_Config(GPIO);	**vTaskSuspend(xLED_Tasks[3]);**
vTaskResume(xLED_Tasks[0]);	prvLED_Config(TIMER);
vTaskResume(xLED_Tasks[1]);	TIM_Cmd(TIM4, *ENABLE);*
vTaskResume(xLED_Tasks[2]);	**vTaskResume(xBALANCE_Task);**
vTaskResume(xLED_Tasks[3]);	**vTaskResume(xMEMS_Task);**

FreeRTOS упражнение – настройка

Измените исследуемую программу следующим способом:

- Только одна задача постоянно активна + задача переключения (SWITCH task)
- Нажатие пользовательской кнопки (User button) переключает “только одна задача”, приостанавливая выполнение других задач.

Для блокировки других задач используйте:

  - Приоритеты задач
  - Бинарные семафоры

- Ликвидация (минимализация) эффекта дребезга переключения.