FreeRTOS
Resource Management

Norman McEntire

Textbook Reference

- Mastering the FreeRTOS Real Time Kernel by Richard Barry

  - Chapter 7: Resource Management

Topics

- 7.1 Intro and Scope
- 7.2 Critical Sections and Suspending the Scheduler
- 7.3 Mutexes (and Binary Semaphores)
- 7.4 Gatekeeper Tasks

7.1 Intro and Scope

- When running multiple tasks, there is potential for error if one task starts to access a resource….
  - …then gets pre-emptied….
    - Then another task starts accessing the same resource
- Result can be subtle errors and resources left in inconsistent state

Some “race condition” examples

- Example 1: Writing to LCD screen without sync with multiple threads
- Example 2: Read, Increment, Write, a data structure without sync with multiple threads
- Example 3: Increment a shared counter without sync with multiple threads
- Example 4: Function that is not reentrant (not thread safe)

Example Reentrant Function (Multiple tasks can call at same time)

- uint32_t uAddOneHundred(uint32_t num1) {
    uint32_t num2;

    num2 = num1 + 10;
    return num2
}

Example Non-Reentrant Function

```
• uint32_t uAddOneHundred(uint32_t num1) {
      uint32_t num2;
      static uint32_t num3; //Not reentrant!
      num3++;
      num2 = num1 + num3 + 10;

      return num2
}
```

Mutual Exclusion

- Shared resources must be accessed using Mutual Exclusion

- Mutual Exclusion ensures only a single task at a time can access a shared resource

What You Will Learn

- When and why resource management control is necessary
- What is a critical section
- What is mutual exclusion
- What does it mean to suspend the scheduler
- How to use a mutex
- How to create and use a gatekeeper task
- What priority inversion is
    - And how priority inheritance can reduce (not remove) impact of priority inversion

7.2 Critical Sections and Suspending the Scheduler

- Base Critical Sections are regions of code that are surrounded by calls to:
  - taskENTER_CRITICAL() - nothing returned
  - taskEXIT_CRITICAL()
- The above can be nested - the FreeRTOS kernel keeps a count of the nesting depth
- Can be used in ISRs (add “FromISR” at end)
  - taskENTER_CRITICAL_FROM_ISR()
    - Returns a value that must be passed to taskEXIT_CRITICAL_FROM_ISR()
  - taskEXIT_CRITICAL_FROM_ISR()

Code Demo 1
Changing a shared variable

- taskENTER_CRITICAL();
  PORTA |= 0x01;
  taskEXIT_CRITICAL();

Code Demo 2
Protecting multiple tasks calling print

- void vPrintString(const char *pcString) {
    taskENTER_CRITICAL();
    {
        printf(“%s”, pcString);
        fflush(stdout);
    }
    taskEXIT_CRITICAL();
}

Code Demo 3

- void vMyISR(void) {
  UBaseType_t unSavedInterruptStatus;

  uxSavedInterruptStatus =
taskENTER_CRITICAL_FROM_ISR();
  ...
  ....

taskEXIT_CRITICAL_FROM_ISR(uxSaveInterruptStatus);

}

Suspending the Scheduler

- An alternative way of creating a critical section is to suspend the scheduler
    - void vTaskSuspendAll(void)
    - BaseType_t xTaskResumeAll(void)
        - Returns pdTRUE if a pending context switch; otherwise pdFALSE
- Suspending the scheduler prevents context switch from occurring
    - But leaves interrupts enabled
    - FreeRTOS APIs cannot be called while schedule suspended

7.3 Mutexes (and Binary Semaphores)

- A mutex is a special type of binary semaphore used to control access to two or more tasks
  - Mutex = MUTual EXclusion
- To use a mutex - the task must hold the token to access the resource
  - You “take” the token to access the resource
  - You “give” the token once you are done

xSemaphoreCreateMutex()
xSemaphoreCreateMutexStatic()

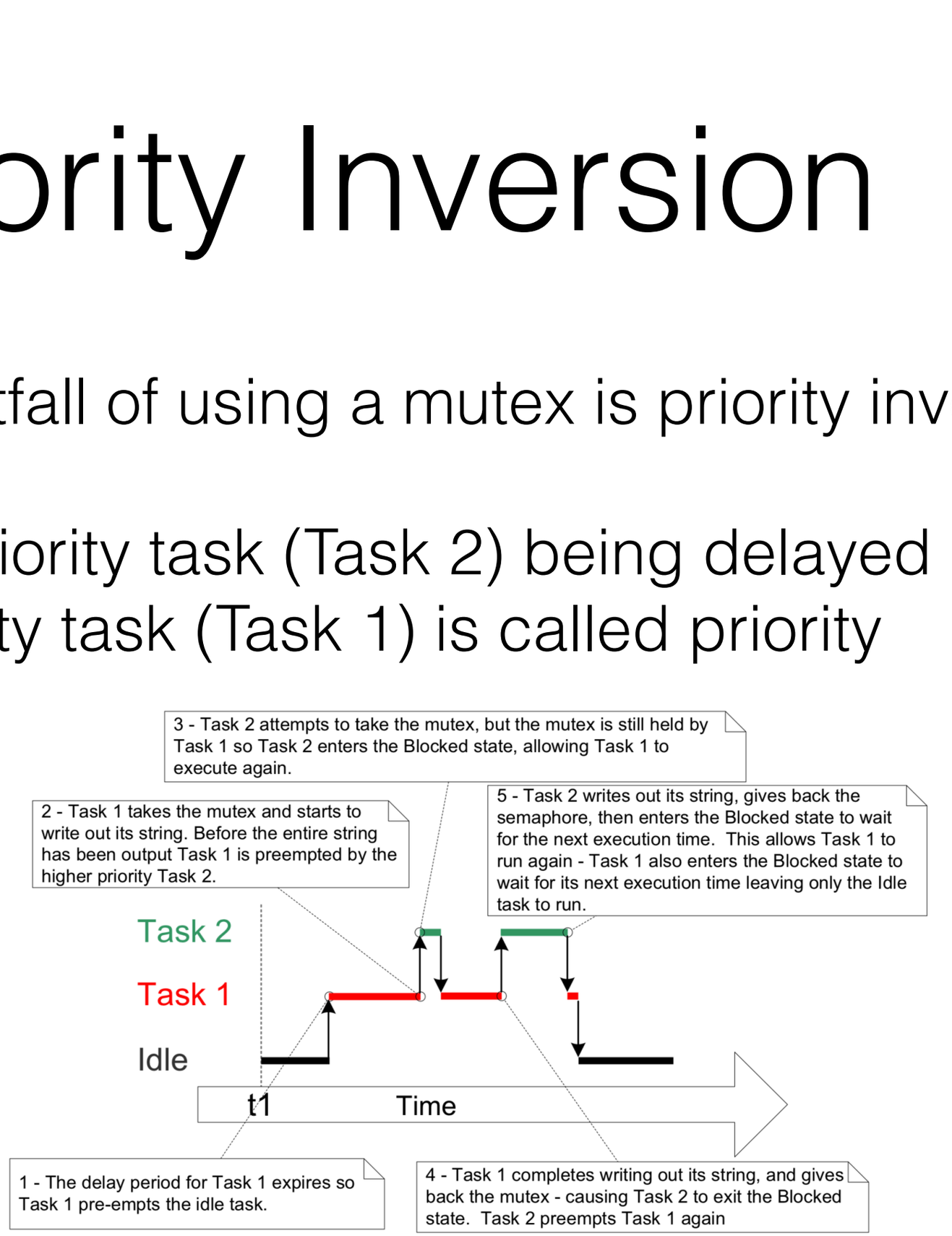
- SemaphoreHandle_t xSemaphoreCreateMutex(void)
- Notes
  - Returns NULL if cannot be created, otherwise the semaphore handle

Code Demo

- SemaphoreHandle_t xMutex;
- int main() {
  …
  xMutex = xSemaphoreCreateMutex();
  …
  }
- static void vPrintString(char *pcString) {
  xSemaphoreTake(xMutex, portMAX_DELAY);
  {
  printf(“%s”, pcString);
  fflush(stdout);
  }
  xSemaphoreGive(xMutex);
  }

Priority Inversion

- A potential pitfall of using a mutex is priority inversion
  - A higher-priority task (Task 2) being delayed by lower-priority task (Task 1) is called priority inversion



Worse Case Priority Inversion

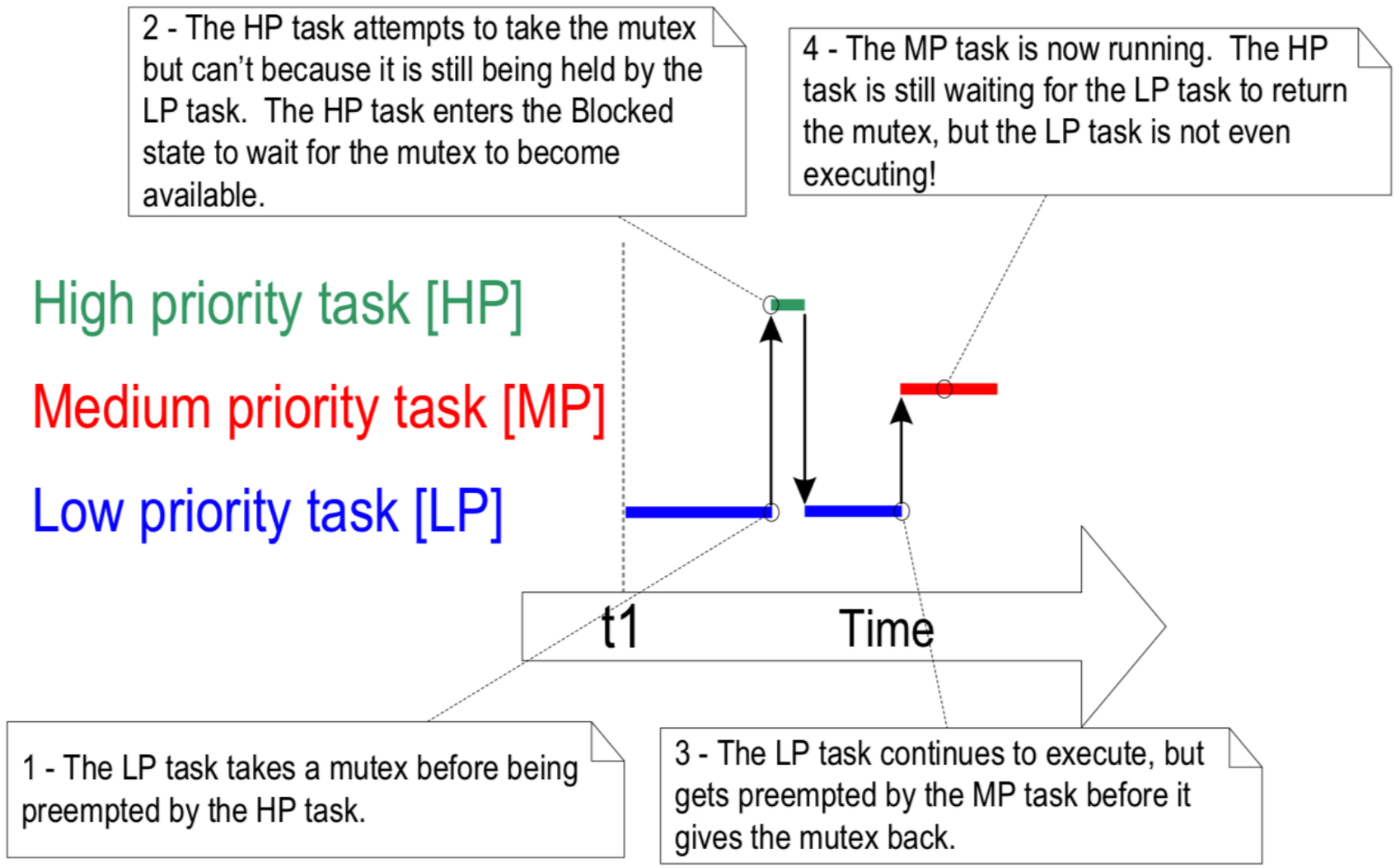


Figure 66. A worst case priority inversion scenario

Priority Inheritance
Part 1

- FreeRTOS mutexes and semaphores are similar
- The difference is that mutexes use a **priority inheritance** mechanism while semaphores do not
    - Priority inheritance minimizes (but does not eliminate) priority inversion

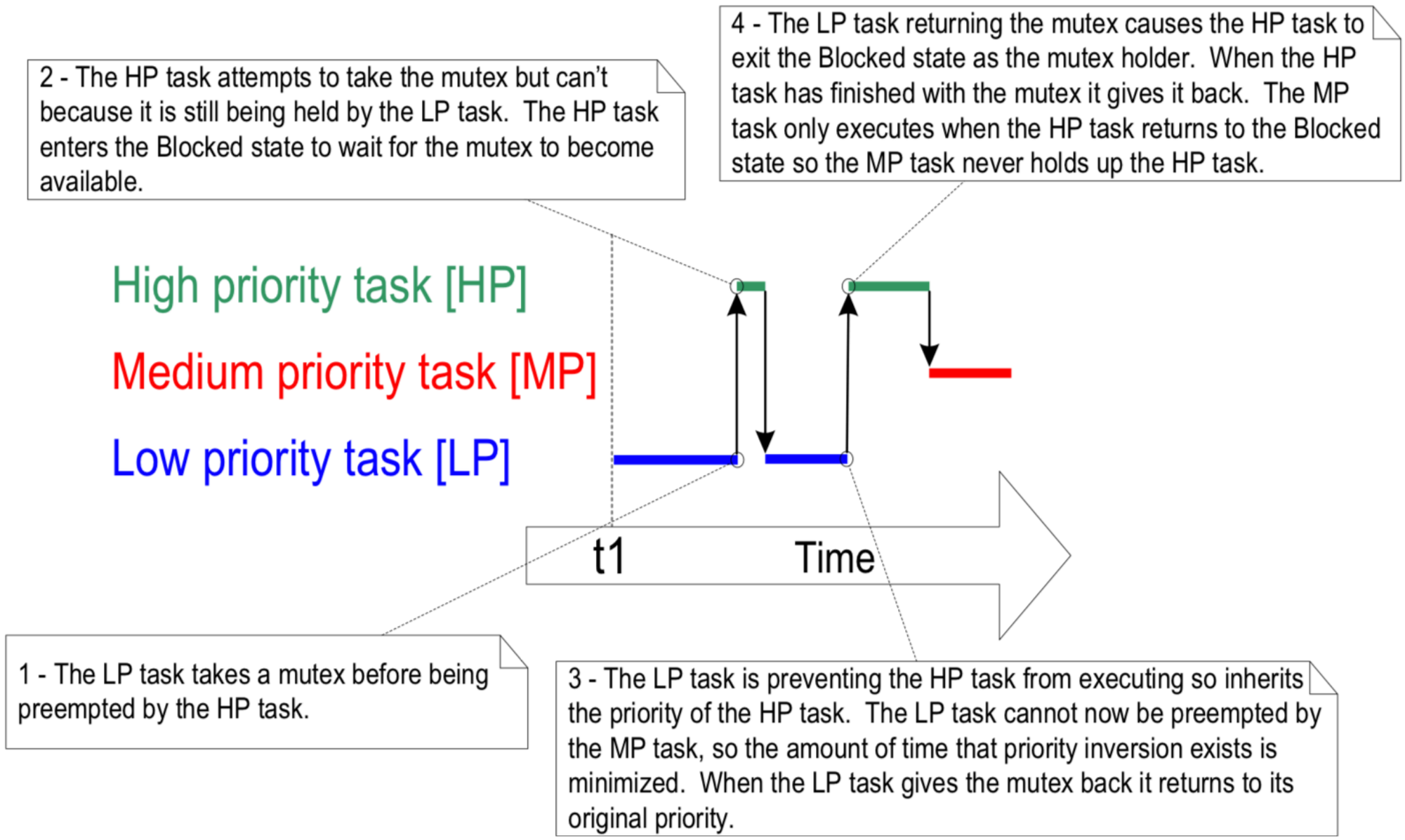
Priority Inheritance Part 2

- Priority inheritance works by temporarily raising the priority of the mutex holder to the priority of the highest priority task that is attempting to obtain the same mutex

- The lower priority task inherits the priority of the higher priority task

- Priority is auto reset once mutex released

Priority Inheritance



Deadlock

- Deadlock (also called Deadly Embrace) occurs when two tasks cannot proceed because each is waiting on a resource held by the other

- Task A holds a resource Task B needs

- Task B holds a resource Task A needs

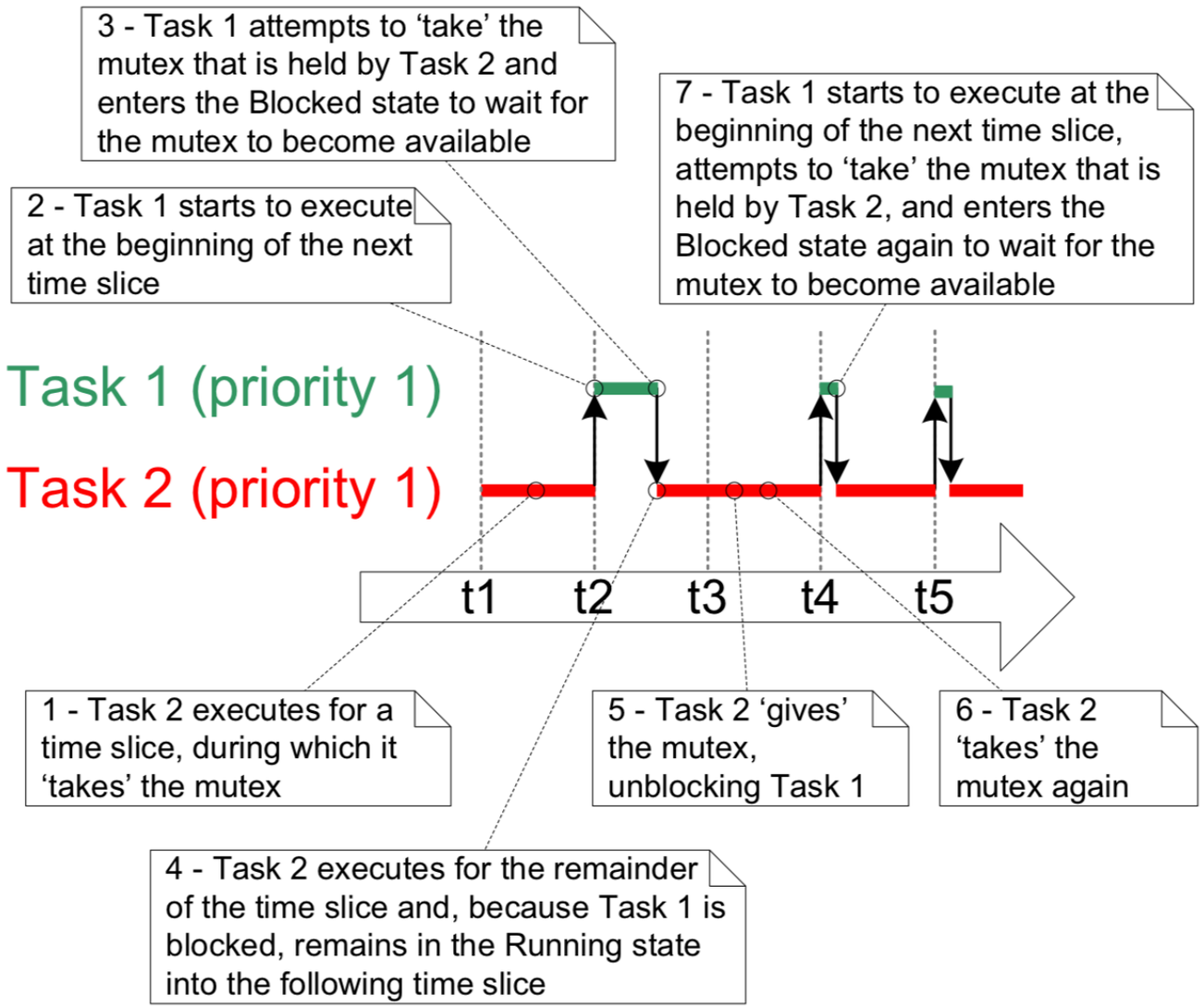
Recursive Mutexes

- It is possible for a task to deadlock with itself
    - This happens if the task attempts to take the same mutex more than once
        - Before returning the mutex
- Example
    - Task A take the mutex
    - While holding mutex, Task A calls library
    - The Library tries to take the same Mutex ==> Deadlock!
        - Solution is to use Recursive Mutex - See next slide

Mutexes - Standard and Recursive

- Create
  - xSemaphoreCreateMutex()
  - xSemaphoreCreateRecursiveMutex()
- Take
  - xSemaphoreTake()
  - xSemaphoreTakeRecursive()
- Give
  - xSemaphoreGive()
  - xSemaphoreGiveRecursive()

Block Diagram - Tasks at Same Priority



7.4 Gatekeeper Tasks

- Gatekeeper tasks provide a clean method of implementing mutual exclusion without the risk of priority inversion or deadlock
- A gatekeeper task is a task that has sole ownership of the resource
    - Any task needing to access resource does so by going through the gatekeeper

Gatekeeper Example

- Re-implement vPrintString() using GateKeeper Task
- The GateKeeper manages access to stdout
- Other tasks that call vPrintString() send output via a queue to the Gatekeeper
  - The Gatekeeper reads from queue and sends output to stdout
- Note: Using this technique, ISRs can send messages to stdout (because they can send to the queue from the ISR)

Code Demo - Part 1

- static void prvStdioGatekeeperTask(void *pv) {
    char *pcMsg;
    for (;;) {
     xQueueReceive(
        xPrintQueue, &pcMsg, portMAX_DELAY);
    printf(“%s”, pcMsg);
    fflush(stdout);
    }
}

Code Demo - Part 2

- QueueHandle_t xPrintQueue;

- int main(void) {
    xPrintQueue = xQueueCreate(5, sizeof(char *));
    xTaskCreate(prvTask1, “Task 1”, 1000, NULL, 1, NULL);
    xTaskCreate(prvTask2, “Task 2”, 1000, NULL, 2, NULL);
    xTaskCreate(prvGatekeeperTask,”Gatekeeper”, 1000, NULL, 0, NULL);

    vTaskStartScheduler();
}

Summary So Far

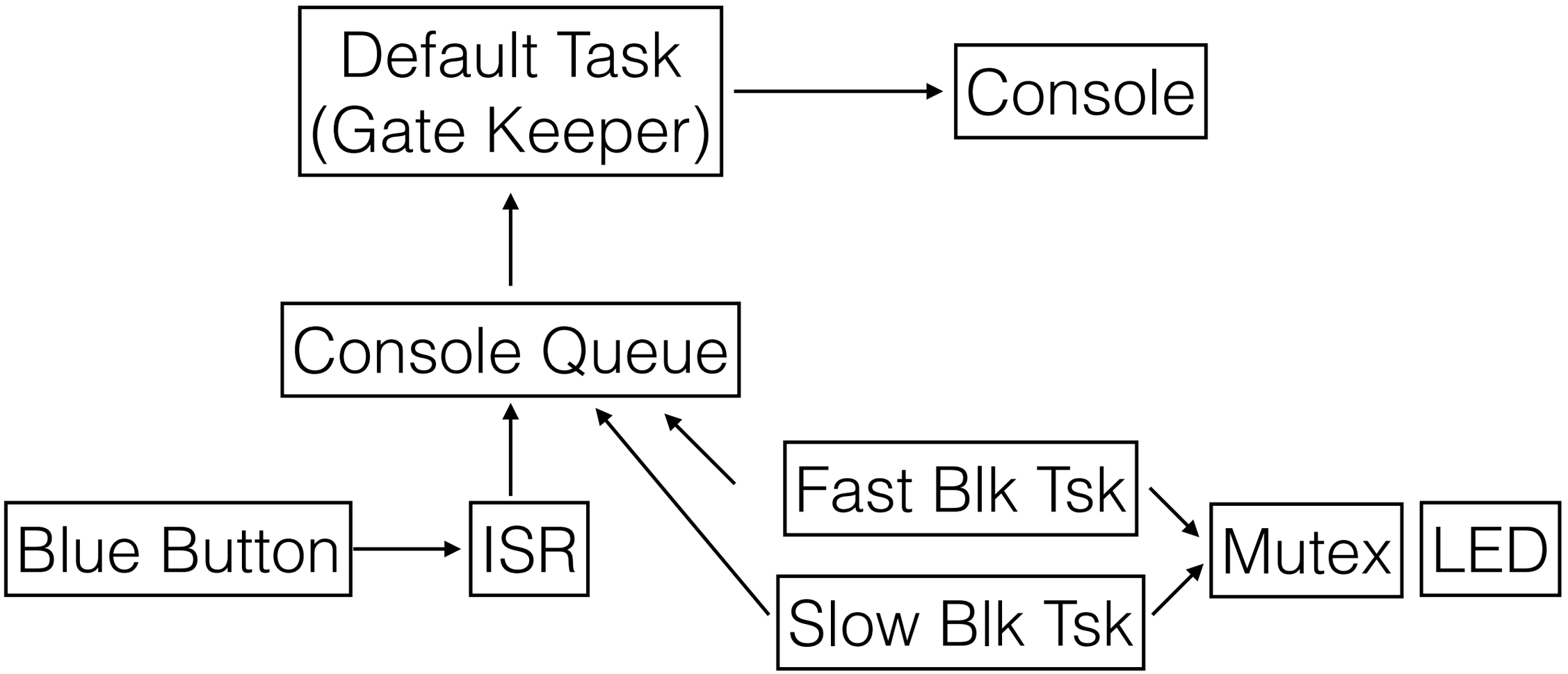
- 7.1 Intro and Scope
- 7.2 Critical Sections and Suspending the Scheduler
- 7.3 Mutexes (and Binary Semaphores)
- 7.4 Gatekeeper Tasks

Coding Demo

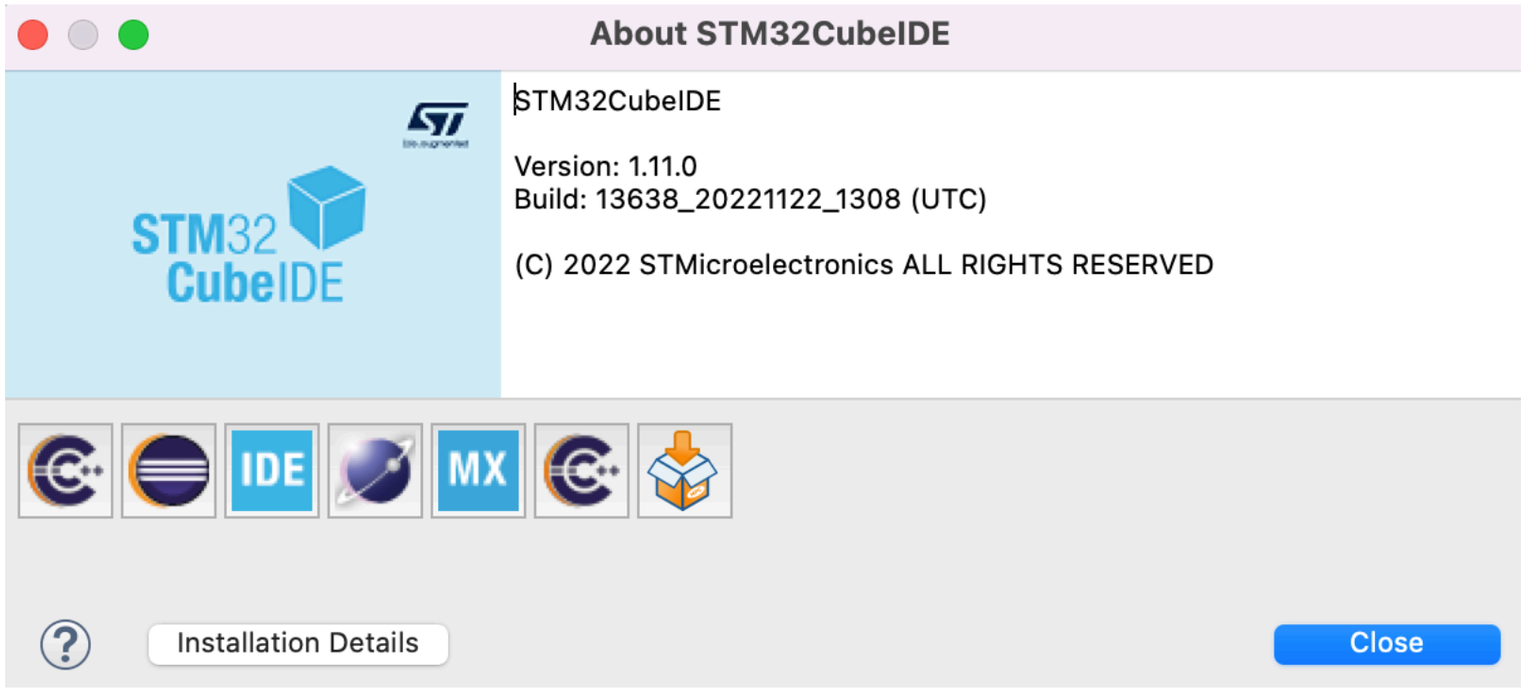
HelloFreeRTOSResMan Project

- Default Task - Gatekeeper for console output
    - Read from queue and send to UART
- Fast Blink LED Task - Blink fast 3x
    - Write to queue
    - Grab LED Mutex and flash LED
- Slow Blink LED Task - Blink slow 3x
    - Write to queue
    - Grab LED Mutex and flash LED
- Button Press - ISR sends message to queue
        - Write to queue

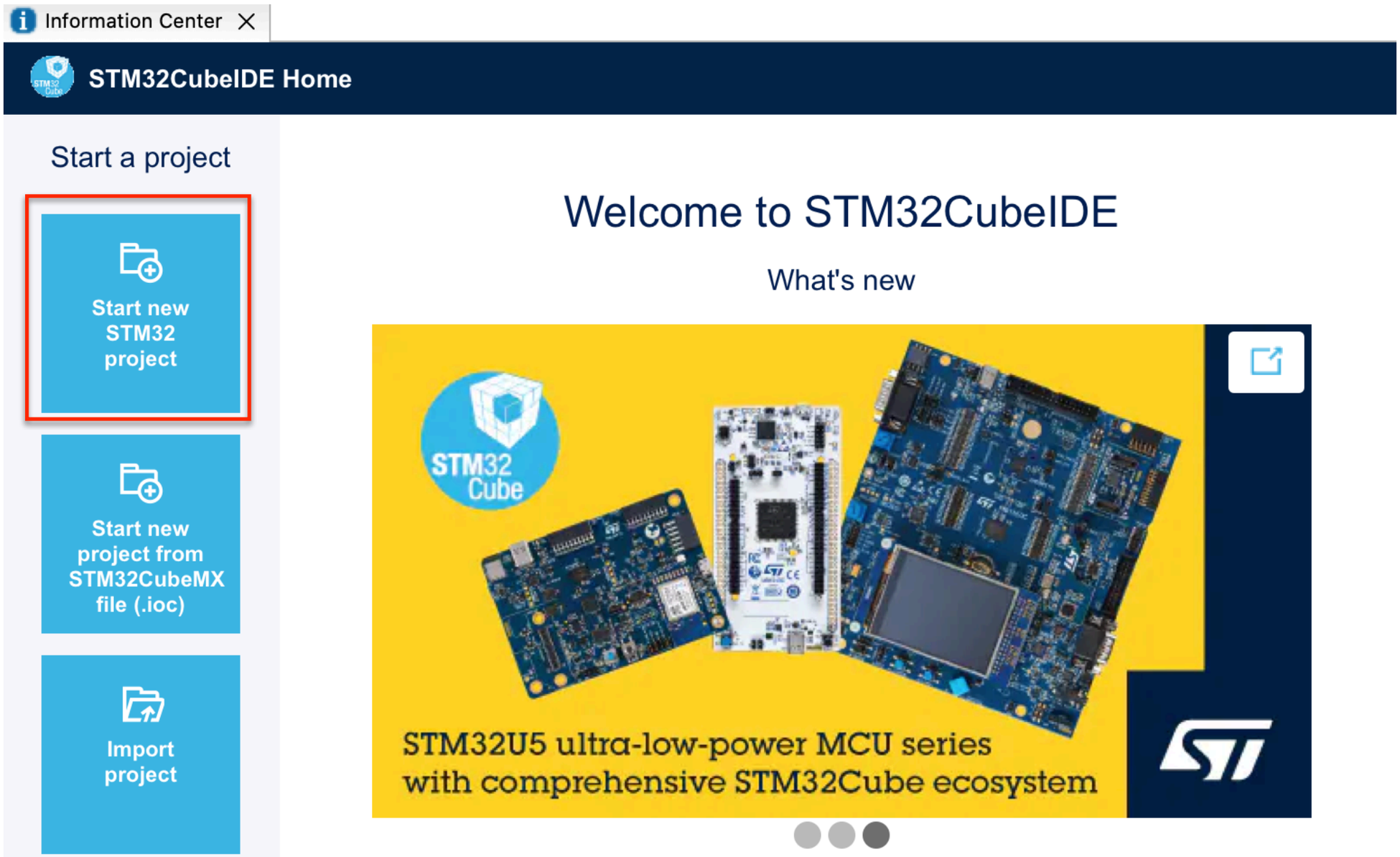
HelloFreeRTOSResMan Block Diagram



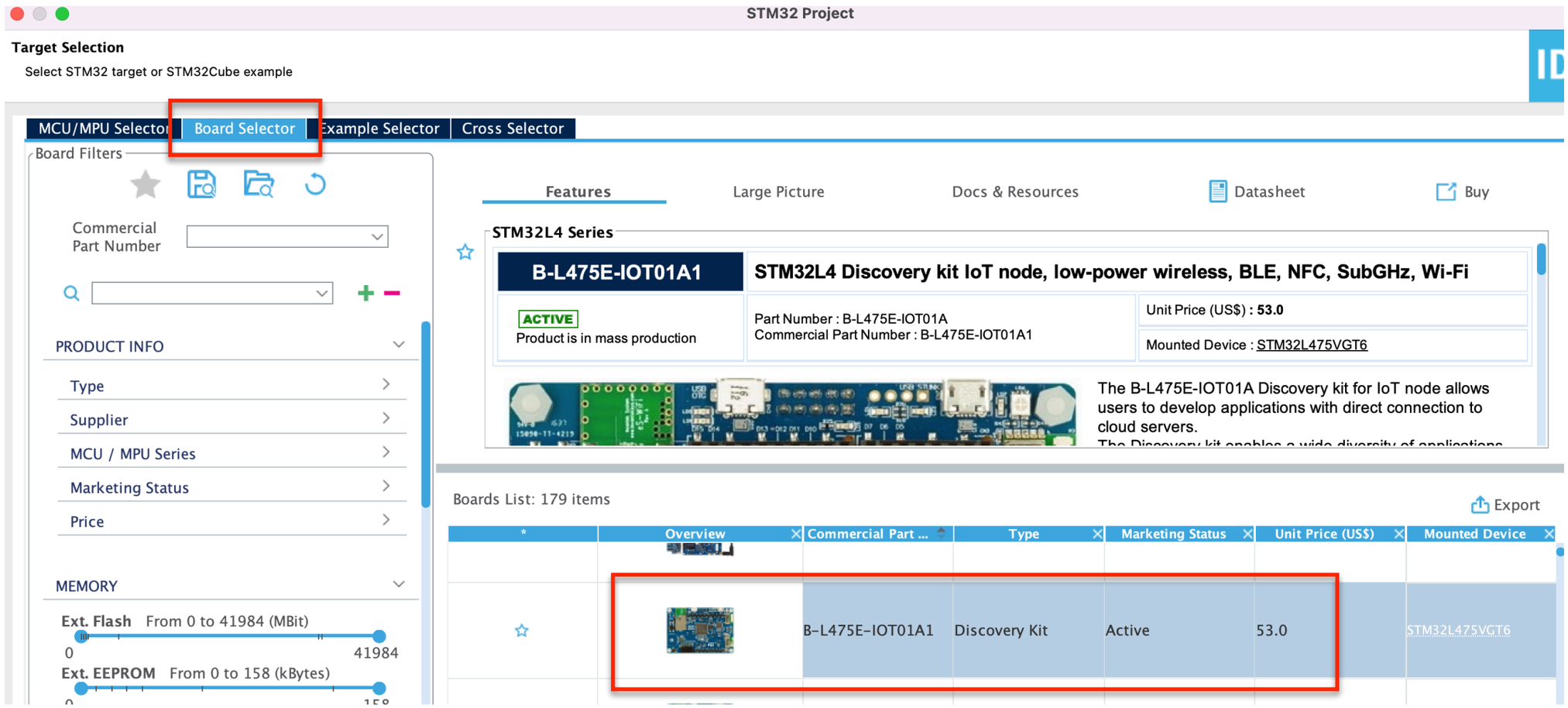
Version of STM32CubeIDE 1.11.0



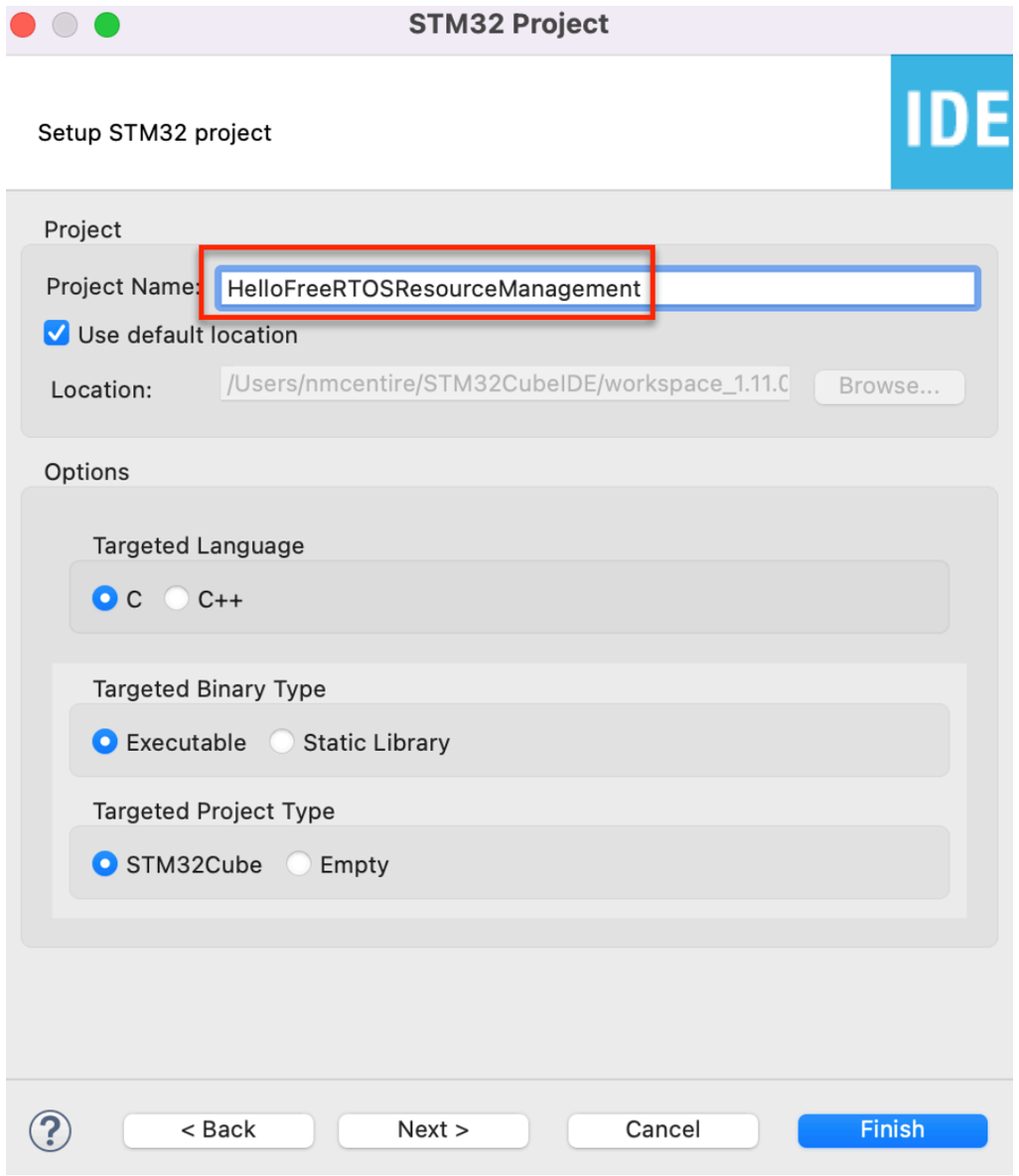
Start new STM32 Project



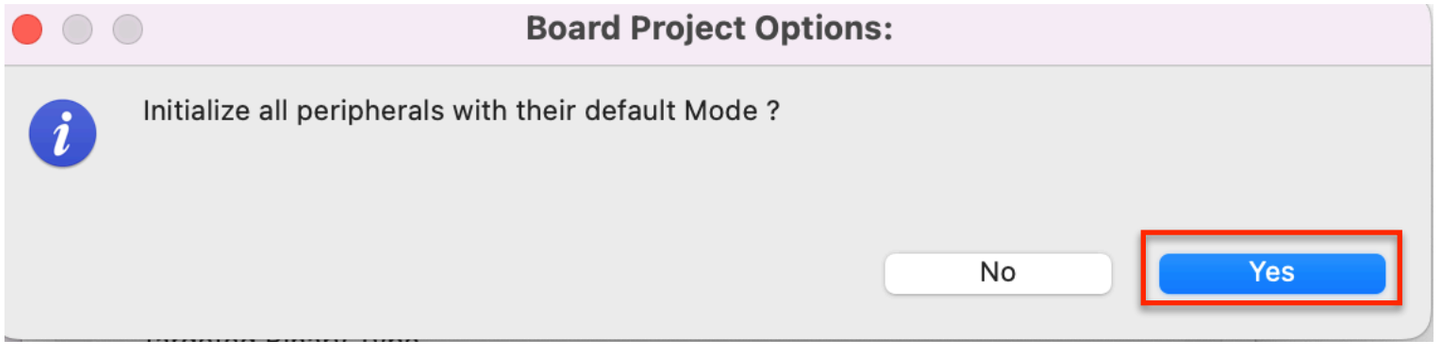
Board Selector



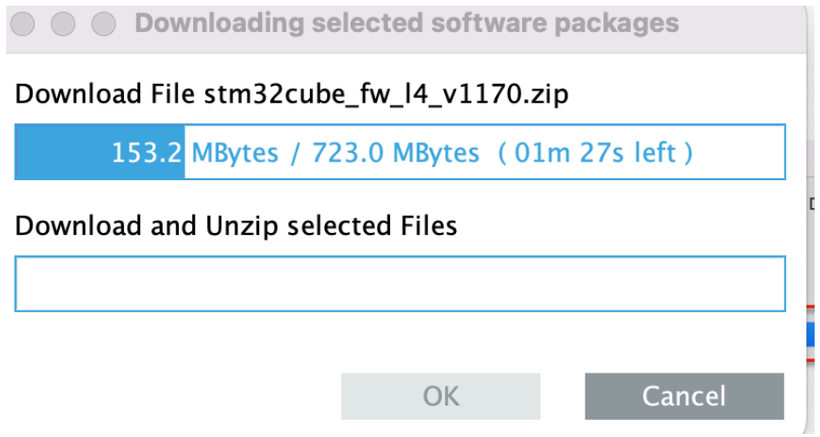
Name Project HelloFreeRTOSResourceManagement

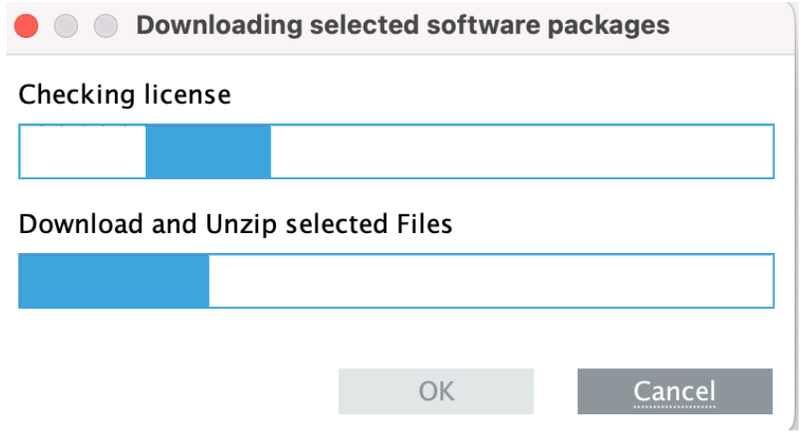


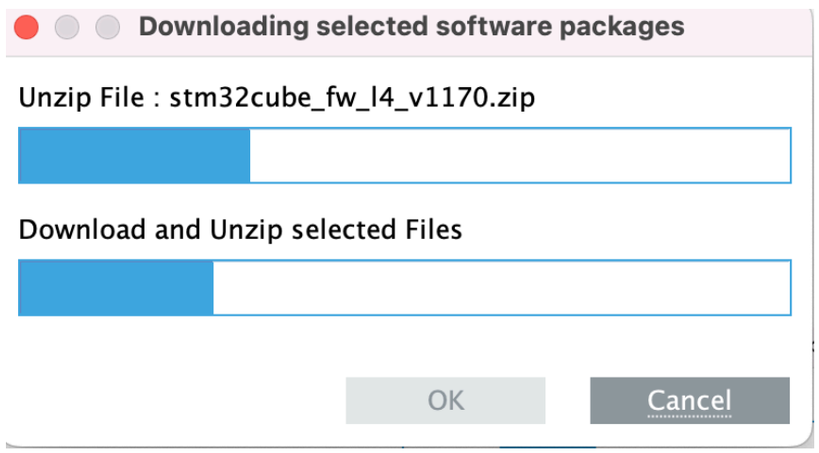
Initialize all peripherals



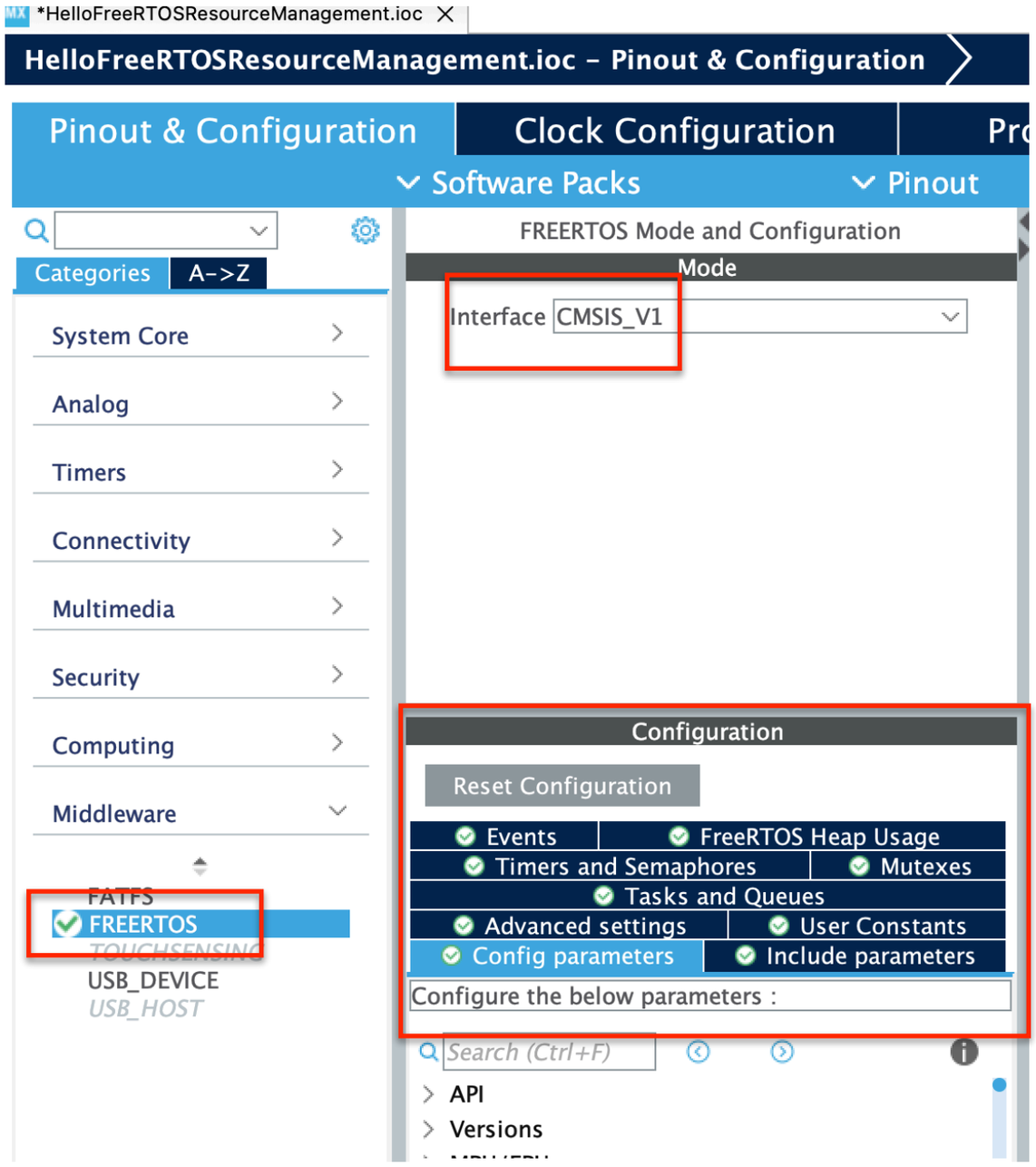
Downloading Software



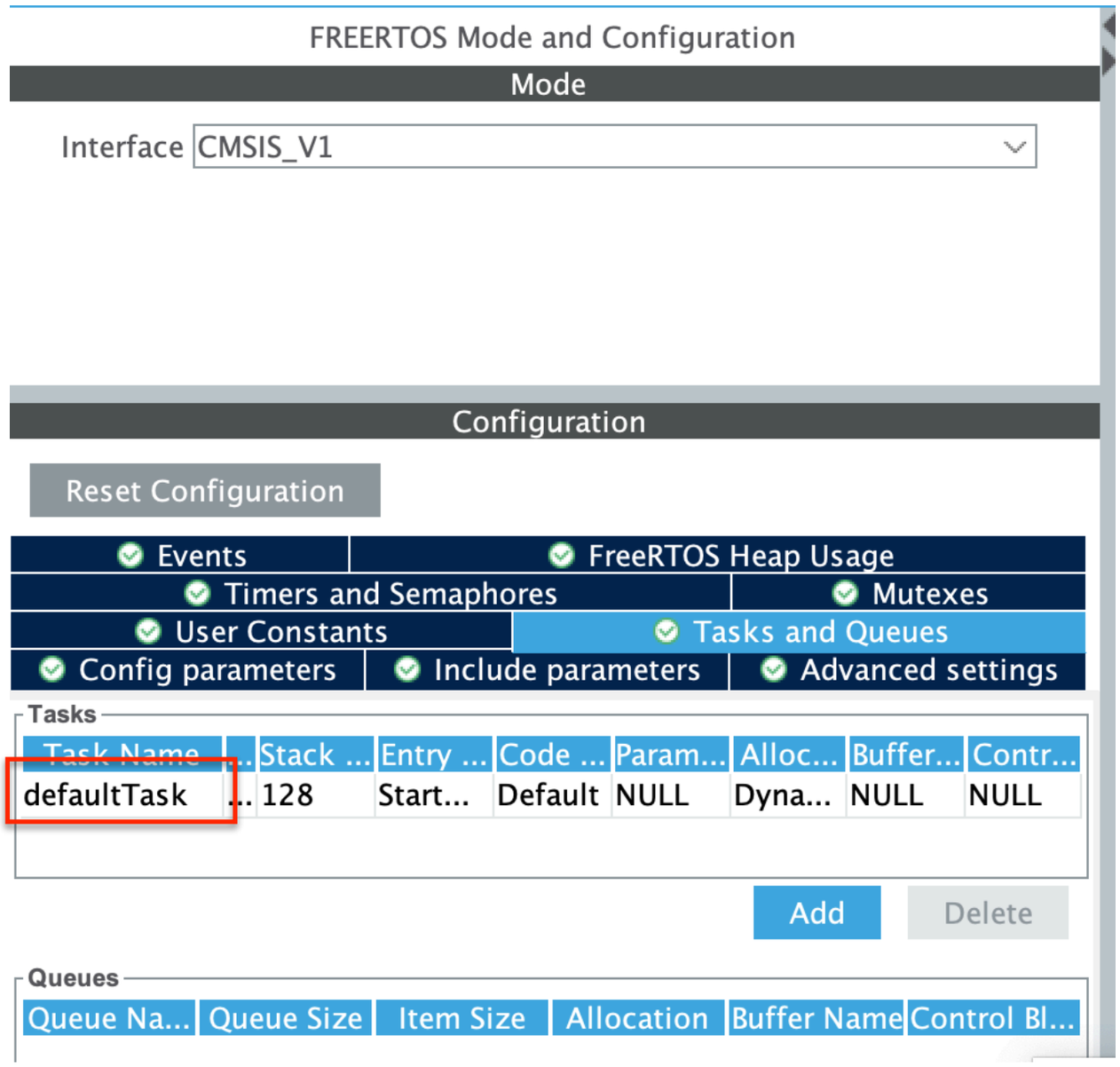




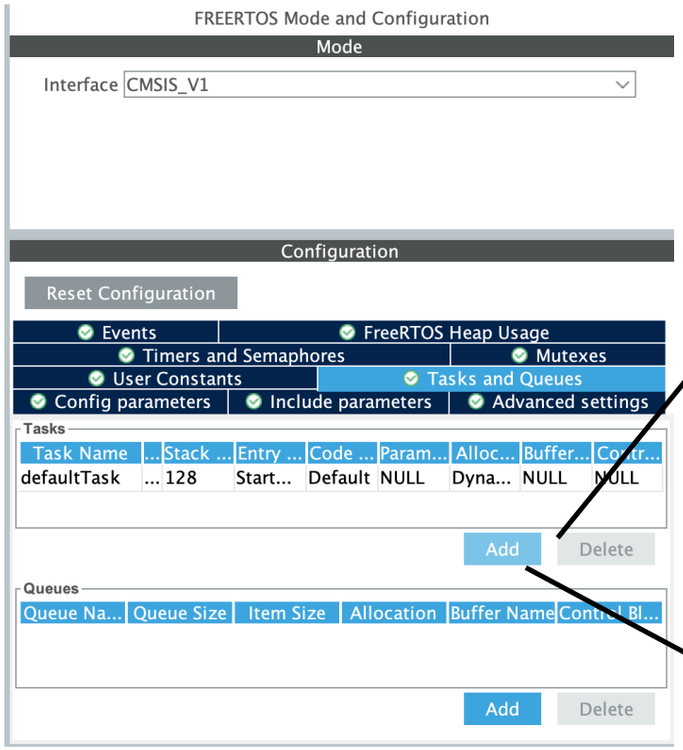
Middleware - FreeRTOS

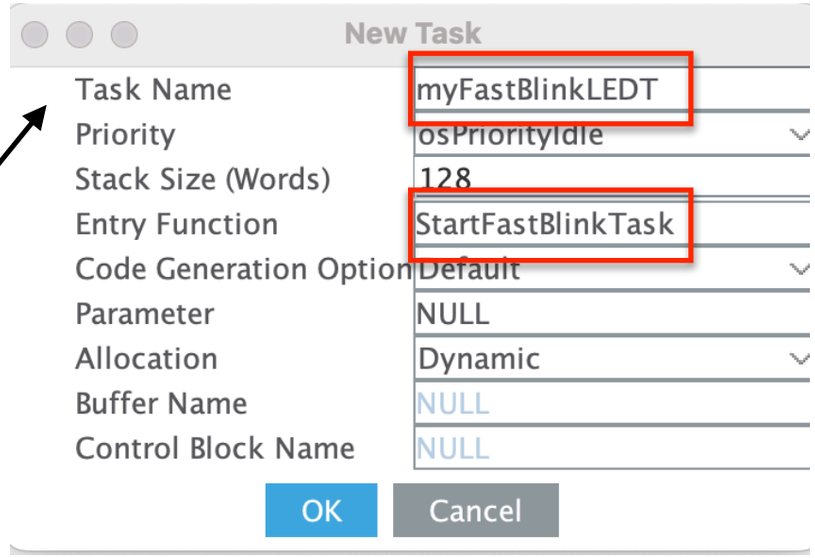


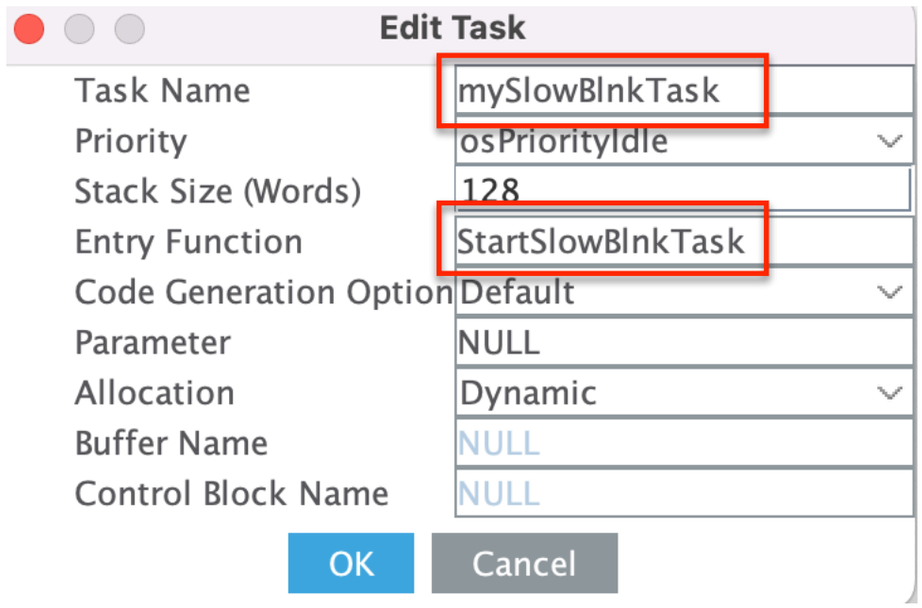
Default Task



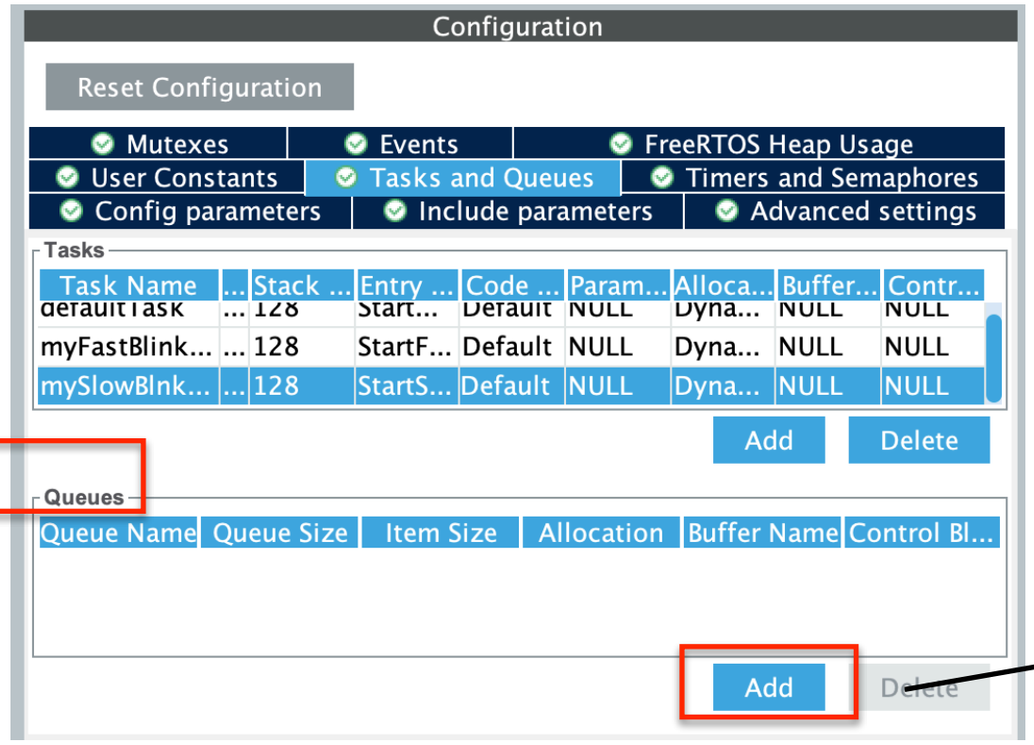
Add Fast Blink and Slow Blink Tasks

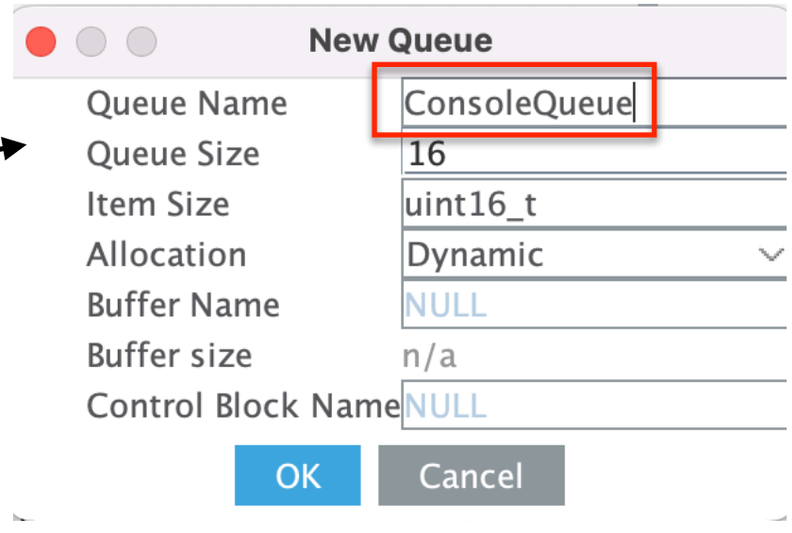




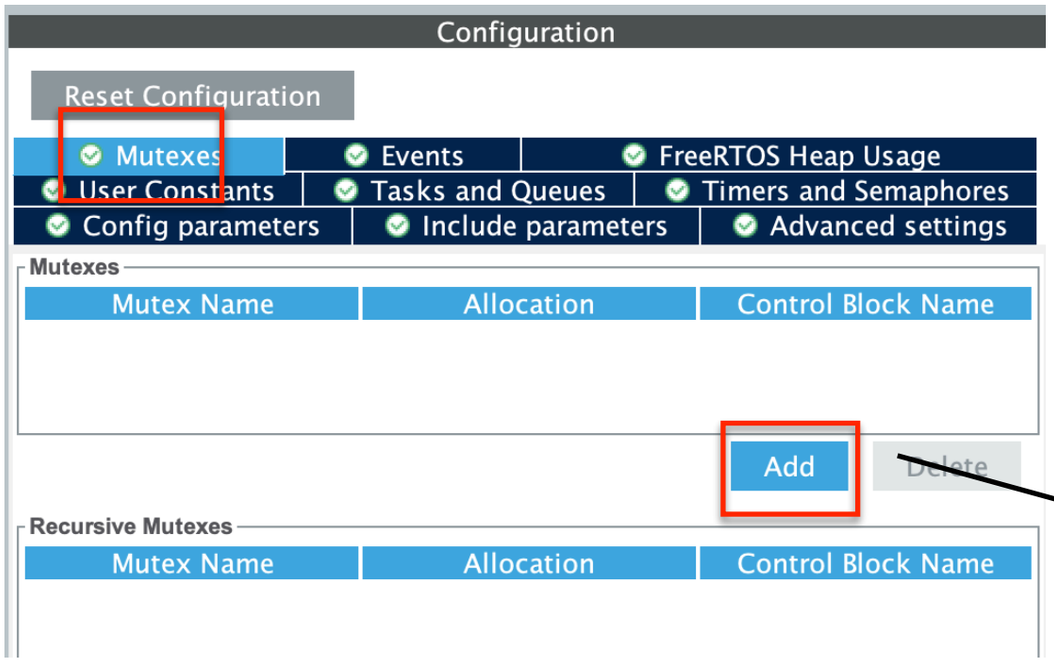


Add Queue

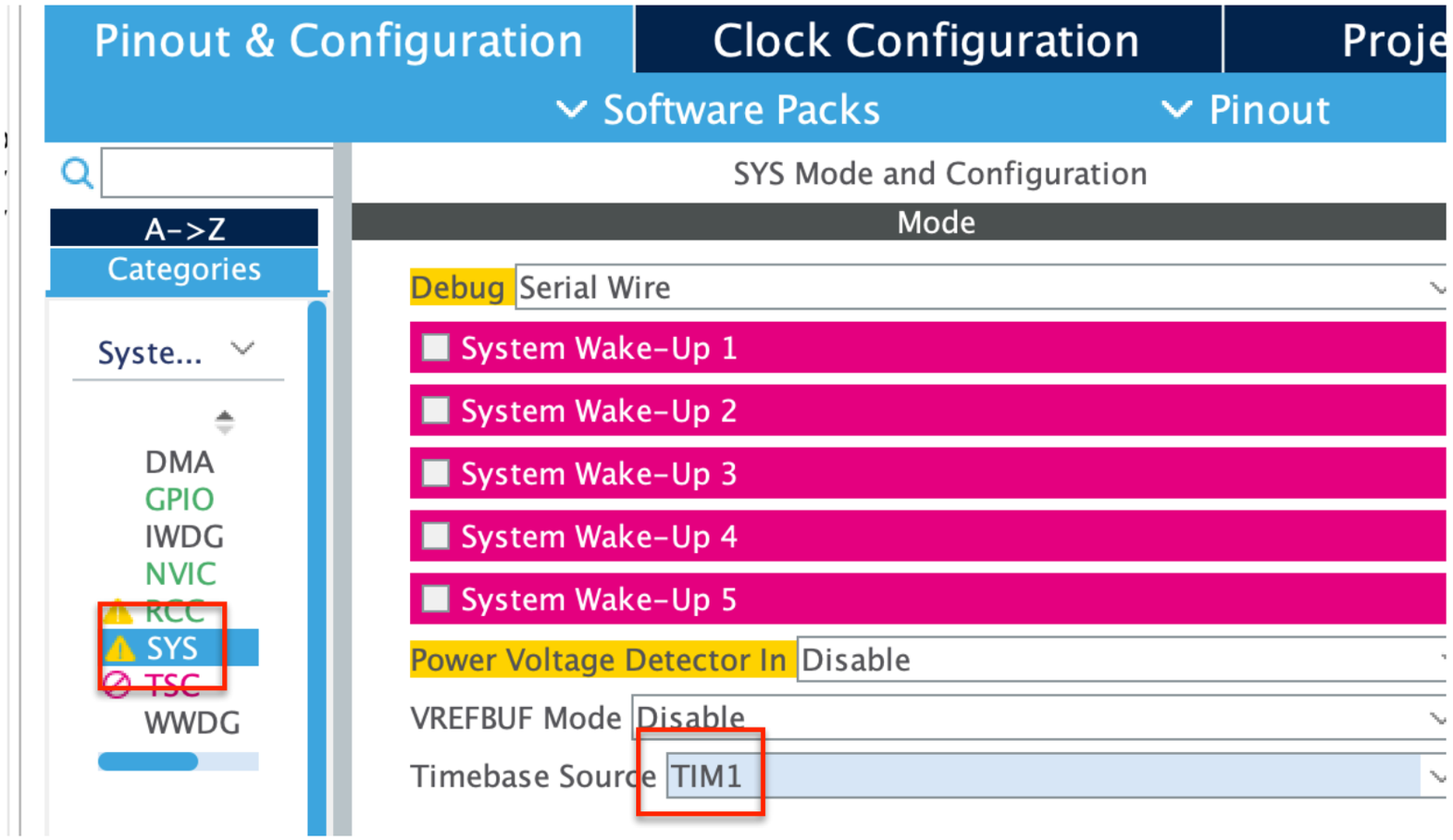




Add Mutex

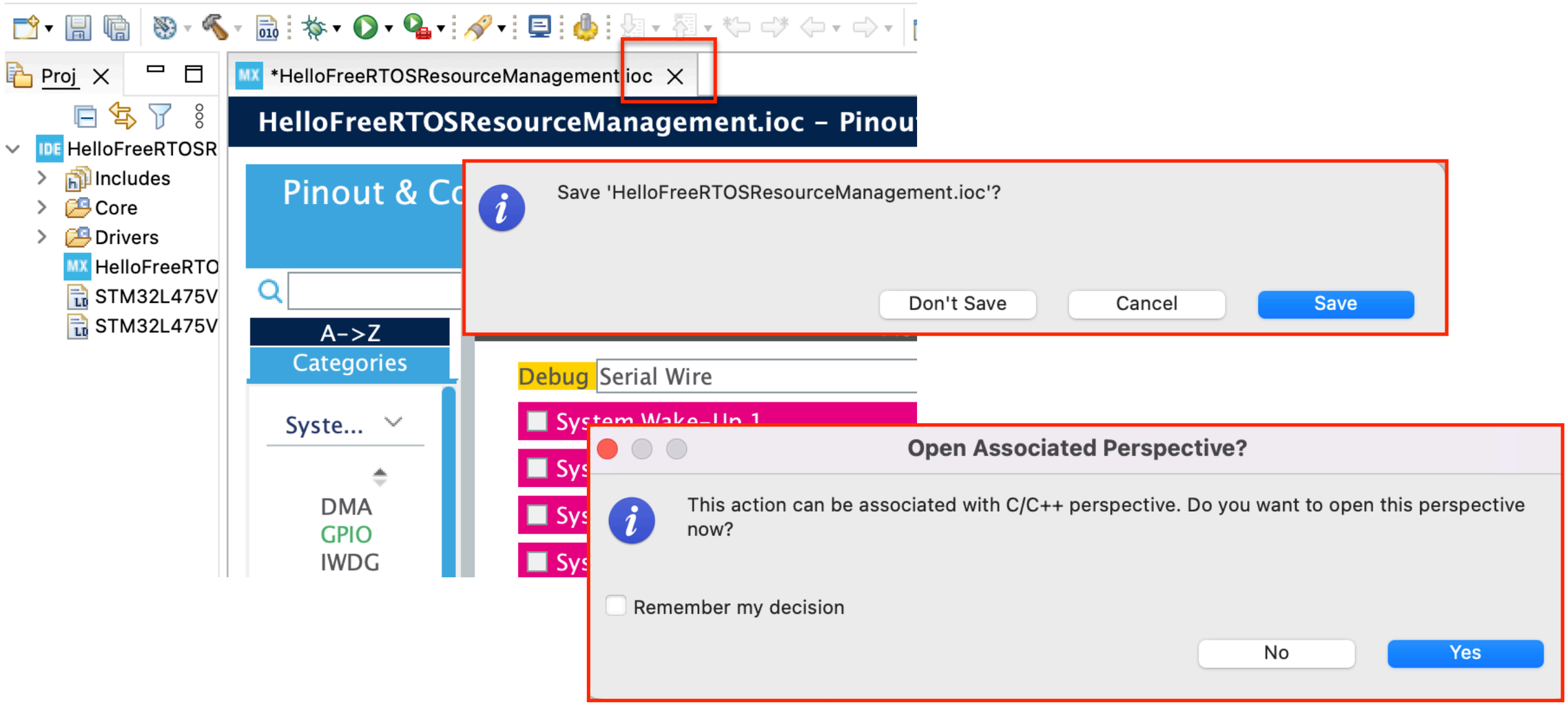


Select Timebase

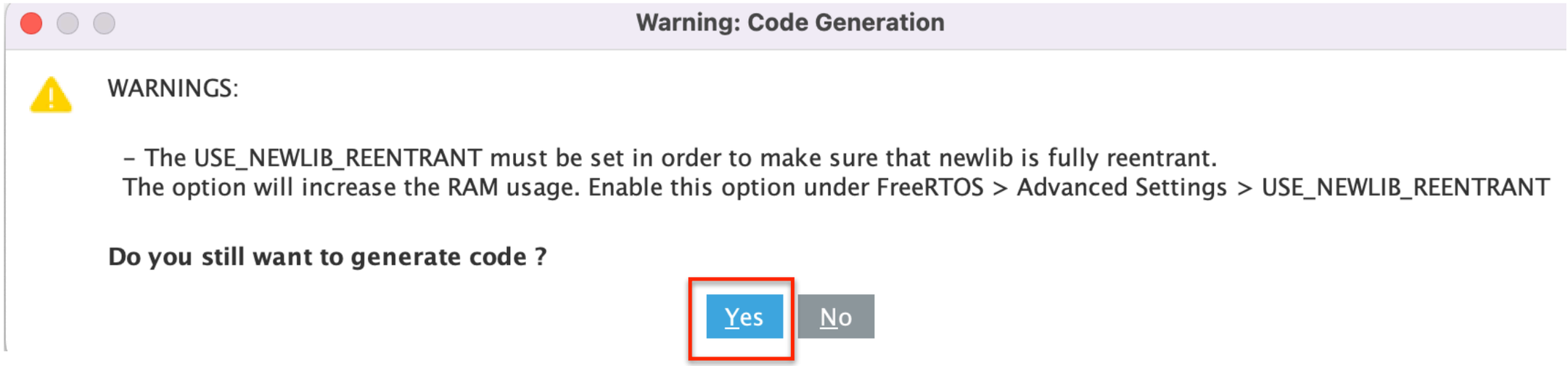


Note: This is timebase for HAL (Hardware Abstraction Layer)

Click on “x” to save and generate code



Warning related to code generation



For this demo we do not need re-entrant code

Looking at Generated Code

main.c
Header Files

```
18 /* USER CODE END Header */
19 /* Includes ------------------------------
20 #include "main.h"
21 #include "cmsis_os.h"
22
23 /* Private includes ----------------------
24 /* USER CODE BEGIN Includes */
25 #include <stdio.h> //For snprintf()
26 #include <string.h> //For strlen()
```

Notice use of cmsis_os.h

We added these

CMSIS: Common Microcontroller Software Interface Standard

main.c
cmsis_os.h

CMSIS: Common Microcontroller Software Interface Standard

```
55
56  #include "FreeRTOS.h"
57  #include "task.h"
58  #include "timers.h"
59  #include "queue.h"
60  #include "semphr.h"
61  #include "event_groups.h"
62

69
70  The file cmsis_os.h contains:
71   - CMSIS-RTOS API function definitions
72   - struct definitions for parameters and return types
73   - status and priority values used by CMSIS-RTOS API functions
74   - macros for defining threads and other kernel objects
75
```

main.c
Our Commands (Sent to the Queue)

```
34  /* Private define ---------------
35    /* USER CODE BEGIN PD */
36
37    #define CMD_NO_OP 0
38    #define CMD_BUTTON_PRESS 1
39    #define CMD_FAST_LED_FLASH 2
40    #define CMD_SLOW_LED_FLASH 3
41
42    /* USER CODE END PD */
43
```

main.c
RTOS Related Handles

```
62
63  osThreadId defaultTaskHandle;
64  osThreadId myFastBlinkLEDTHandle;
65  osThreadId mySlowBlnkTaskHandle;
66  osMessageQId ConsoleQueueHandle;
67  osMutexId LEDFlashMutexHandle;
68  /* USER CODE BEGIN PV */
```

main.c
RTOS Related Function Prototypes

```
81  static void MX_USB_OTG_FS_PCD_Init(void);
82  void StartDefaultTask(void const * argument);
83  void StartFastBlinkTask(void const * argument);
84  void StartSlowBlnkTask(void const * argument);
85
```

main.c
Mutext Create

```
133
134   /* Create the mutex(es) */
135   /* definition and creation of LEDFlashMutex */
136   osMutexDef(LEDFlashMutex);
137   LEDFlashMutexHandle = osMutexCreate(osMutex(LEDFlashMutex));
138
```

main.c
Queue Create

```
151   /* Create the queue(s) */
152   /* definition and creation of ConsoleQueue */
153   osMessageQDef(ConsoleQueue, 16, uint16_t);
154   ConsoleQueueHandle = osMessageCreate(osMessageQ(ConsoleQueue), NULL);
155
```

main.c
Thread (Task) Create

```
159
160   /* Create the thread(s) */
161   /* definition and creation of defaultTask */
162   osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);
163   defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);
164
165   /* definition and creation of myFastBlinkLEDT */
166   osThreadDef(myFastBlinkLEDT, StartFastBlinkTask, osPriorityIdle, 0, 128);
167   myFastBlinkLEDTHandle = osThreadCreate(osThread(myFastBlinkLEDT), NULL);
168
169   /* definition and creation of mySlowBlnkTask */
170   osThreadDef(mySlowBlnkTask, StartSlowBlnkTask, osPriorityIdle, 0, 128);
171   mySlowBlnkTaskHandle = osThreadCreate(osThread(mySlowBlnkTask), NULL);
172
173   /* USER CODE BEGIN RTOS_THREADS */
174   /* add threads, ... */
175   /* USER CODE END RTOS_THREADS */
176
177   /* Start scheduler */
178   osKernelStart();
179
```

main.c
Interrupts are correct priority

```
694    /* EXTI interrupt init*/
695    HAL_NVIC_SetPriority(EXTI9_5_IRQn, 5, 0);
696    HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
697
698    HAL_NVIC_SetPriority(EXTI15_10_IRQn, 5, 0);
699    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
```

NOTE: Notice priority is set correctly

```
106  /* The highest interrupt priority that can be used by any interrupt service
107   routine that makes calls to interrupt safe FreeRTOS API functions.  DO NOT CALL
108   INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
109   PRIORITY THAN THIS! (higher priorities are lower numeric values. */
110   #define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5
```

cmsis_os.h
osMessageGet

```
802 /// Get a Message or Wait for a Message from a Queue.
803 /// \param[in]     queue_id      message queue ID obtained with \ref osMessageCreate.
804 /// \param[in]     millisec      timeout value or 0 in case of no time-out.
805 /// \return event information that includes status code.
806 /// \note MUST REMAIN UNCHANGED: \b osMessageGet shall be consistent in every CMSIS-RTOS.
807 osEvent osMessageGet (osMessageQId queue_id, uint32_t millisec);
```

```
347 /// Event structure contains detailed information about an event.
348 /// \note MUST REMAIN UNCHANGED: \b os_event shall be consistent in every CMSIS-RTOS.
349 ///       However the struct may be extended at the end.
350 typedef struct  {
351   osStatus                 status;     ///< status code: event or error information
352   union  {
353     uint32_t                    v;     ///< message as 32-bit value
354     void                       *p;     ///< message or mail as void pointer
355     int32_t                 signals;   ///< signal flags
356   } value;                             ///< event value
357   union  {
358     osMailQId               mail_id;   ///< mail id obtained by \ref osMailCreate
359     osMessageQId         message_id;   ///< message id obtained by \ref osMessageCreate
360   } def;                               ///< event definition
361 } osEvent;
362
```

main.c
StartDefaultTask

```
715 void StartDefaultTask(void const * argument)
716 {
717   /* USER CODE BEGIN 5 */
718   /* Infinite loop */
719
720     // Send out initial message
721     char buf[100];
722     snprintf(buf, sizeof(buf), "StartDefaultTask\n\r");
723     HAL_UART_Transmit(&huart1, (uint8_t *) buf, strlen(buf), 1000);
724
725   for(;;)
726   {
727     // Wait to receive command
728     osEvent event = osMessageGet(ConsoleQueueHandle, osWaitForever);
729     // Format message for console
730     snprintf(buf, sizeof(buf), "event.value.v: %lu\n\r", event.value.v);
731     HAL_UART_Transmit(&huart1, (uint8_t *) buf, strlen(buf), 1000);
732
733   }
734   /* USER CODE END 5 */
735 }
```

cmsis_os.h
osMutextWait()/osMutextRelease()

```
613
614 /// Wait until a Mutex becomes available.
615 /// \param[in]     mutex_id      mutex ID obtained by \ref osMutexCreate.
616 /// \param[in]     millisec      timeout value or 0 in case of no time-out.
617 /// \return status code that indicates the execution status of the function.
618 /// \note MUST REMAIN UNCHANGED: \b osMutexWait shall be consistent in every CMSIS-RTOS.
619 osStatus osMutexWait (osMutexId mutex_id, uint32_t millisec);
620
621 /// Release a Mutex that was obtained by \ref osMutexWait.
622 /// \param[in]     mutex_id      mutex ID obtained by \ref osMutexCreate.
623 /// \return status code that indicates the execution status of the function.
624 /// \note MUST REMAIN UNCHANGED: \b osMutexRelease shall be consistent in every CMSIS-RTOS
625 osStatus osMutexRelease (osMutexId mutex_id);
626
```

main.c
StartFastBlinkTask

```
745 void StartFastBlinkTask(void const * argument)
746 {
747   /* USER CODE BEGIN StartFastBlinkTask */
748   /* Infinite loop */
749   for(;;)
750   {
751     osDelay(1000);
752
753     // Grab Mutex
754     osMutexWait (LEDFlashMutexHandle, osWaitForever);
755
756     // Write to queue
757     osMessagePut(ConsoleQueueHandle, CMD_FAST_LED_FLASH, osWaitForever);
758
759     // Turn LED On
760     HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, SET);
761     // Sleep briefly
762     osDelay(250);
763     // Turn LED Off
764     HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, RESET);
765
766     // Release Mutex
767     osMutexRelease (LEDFlashMutexHandle);
768
769   }
770   /* USER CODE END StartFastBlinkTask */
771 }
```

main.c
StartSlowBlinkTask

```
780 void StartSlowBlnkTask(void const * argument)
781 {
782   /* USER CODE BEGIN StartSlowBlnkTask */
783   /* Infinite loop */
784   for(;;)
785   {
786         osDelay(1000);
787
788         // Grab Mutex
789         osMutexWait (LEDFlashMutexHandle, osWaitForever);
790
791         // Write to queue
792         osMessagePut(ConsoleQueueHandle, CMD_SLOW_LED_FLASH, osWaitForever);
793
794         // Turn LED On
795         HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, SET);
796         // Sleep briefly
797         osDelay(1000);
798         // Turn LED Off
799         HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, RESET);
800
801         // Release Mutex
802         osMutexRelease (LEDFlashMutexHandle);
803   }
804   /* USER CODE END StartSlowBlnkTask */
805 }
```

stm32l4xx_hal_gpio.c

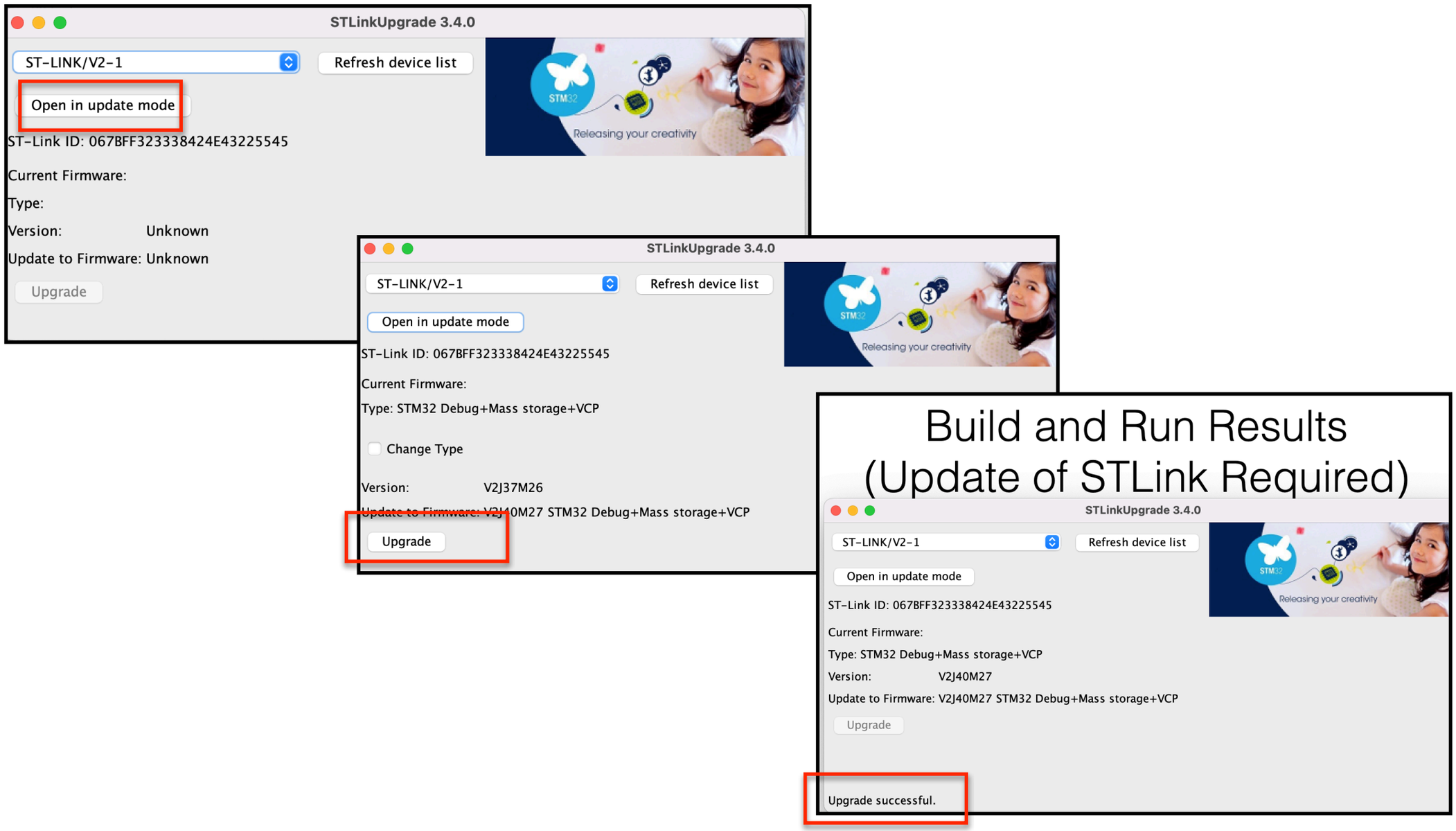
```
504 /**
505   * @brief  Handle EXTI interrupt request.
506   * @param  GPIO_Pin Specifies the port pin connected to corresponding EXTI line.
507   * @retval None
508   */
509 void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
510 {
511   /* EXTI line interrupt detected */
512   if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != 0x00u)
513   {
514     __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
515     HAL_GPIO_EXTI_Callback(GPIO_Pin);
516   }
517 }
518
519 /**
520   * @brief  EXTI line detection callback.
521   * @param  GPIO_Pin Specifies the port pin connected to corresponding EXTI line.
522   * @retval None
523   */
524 __weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
525 {
526   /* Prevent unused argument(s) compilation warning */
527   UNUSED(GPIO_Pin);
528
529   /* NOTE: This function should not be modified, when the callback is needed,
530            the HAL_GPIO_EXTI_Callback could be implemented in the user file
531    */
532 }
533
```

Put this in main.c

main.c
Interrupt Callback

```
806
807  // Our interrupt handling routine
808  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
809      // At this point interrupt is clear
810      // Just send the message
811      osMessagePut(ConsoleQueueHandle, CMD_BUTTON_PRESS, osWaitForever);
812  }
813  /**
```

Build and Run Results (Update of STLink Required)



Results

```
36  /* USER CODE BEGIN PD */
37
38  #define CMD_NO_OP 0
39  #define CMD_BUTTON_PRESS 1
40  #define CMD_FAST_LED_FLASH 2
41  #define CMD_SLOW_LED_FLASH 3
42
43  /* USER CODE END PD */
44
```

```
StartDefaultTask
event.value.v: 2
event.value.v: 3
event.value.v: 2
event.value.v: 3
event.value.v: 2
event.value.v: 3
event.value.v: 2
event.value.v: 3
event.value.v: 1  <-- Button Press
event.value.v: 2
event.value.v: 3
event.value.v: 2
event.value.v: 3
event.value.v: 1  <-- Button Press
event.value.v: 2
event.value.v: 3
event.value.v: 2
event.value.v: 3
event.value.v: 2
event.value.v: 3
event.value.v: 2
event.value.v: 3
```

Questions/Answers