Introduction to FreeRTOS

- We will use FreeRTOS as our example
  - Other popular RTOSes include Zephyr, NuttX, VxWorks. Varying licensing argreements.
  - Like a programming language: once you learn one RTOS, concepts transfer to others.
- FreeRTOS licensed under MIT license – very permissive.
  - Can be used in commercial applications and users retain all ownership of their IP.

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Code Structure of FreeRTOS

```
FreeRTOS
   │
   └─Source
        │
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

Figure 2. Core FreeRTOS source files within the FreeRTOS directory tree

```
FreeRTOS
  │
  └─Source
     │
     └─portable Directory containing all port specific source files
        │
        ├─MemMang Directory containing the 5 alternative heap allocation source files
        │
        ├─[compiler 1] Directory containing port files specific to compiler 1
        │   │
        │   ├─[architecture 1] Contains files for the compiler 1 architecture 1 port
        │   ├─[architecture 2] Contains files for the compiler 1 architecture 2 port
        │   └─[architecture 3] Contains files for the compiler 1 architecture 3 port
        │
        └─[compiler 2] Directory containing port files specific to compiler 2
            │
            ├─[architecture 1] Contains files for the compiler 2 architecture 1 port
            ├─[architecture 2] Contains files for the compiler 2 architecture 2 port
            └─[etc.]
```

Figure 3. Port specific source files within the FreeRTOS directory tree

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Data Types and Naming Conventions

- Port specific data types
  - TickType_t – holds tick count value
  - BaseType_t – Base type value which is most efficient data type on a given system. Typically the word size.

Naming Conventions: Variable Names

- Variable Names - prefixes tell their type

Prefix	Type
c	char
s	int16_t (short)
i	int32_t (long)
x	BaseType_t and other non-standard types (structs, task handles, queue handles, etc.)

Naming Conventions: Function Names

- Function Names – prefixed with return type and file they are defined in.

Function	Description
v**Task**PrioritySet()	returns a void and is defined within **task**.c.
x**Queue**Receive()	returns a variable of type BaseType_t and is defined within **queue**.c.
pv**Timer**GetTimerID()	returns a pointer to void and is defined within **timers**.c.

Template Project

```c
int main( void )
{
    /* Perform any hardware setup necessary. */
    prvSetupHardware();

    /* --- APPLICATION TASKS CAN BE CREATED HERE --- */

    /* Start the created tasks running. */
    vTaskStartScheduler();

    /* Execution will only reach here if there was insufficient heap to
    start the scheduler. */
    for( ;; );
    return 0;
}
```

Listing 1. The template for a new main() function

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Creating Tasks

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

Listing 13. The xTaskCreate() API function prototype

Parameters

- pvTaskCode - pointer to C function that implements that task
- pcName – Descriptive name for the task
- usStackDepth – size of stack to be allocated by the kernel when creating the stack (in words)
- pvParameters – pointer to void to pass in parameters. Need to cast void pointer to correct type inside the function to use it.
- uxPriority – Defines the priority of the task
- pxCreatedTask – handle to created task

Return

- pdPass or pdFail - indicates if task was successfully created.

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Printing to Terminal

```c
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

```c
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

```c
int main(void) {
  xTaskCreate(vTask1, "Task 1", 1000, NULL, 1, NULL);
  xTaskCreate(vTask2, "Task 2", 1000, NULL, 1, NULL);
  vTaskStartScheduler();

  for( ;; );
}
```

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Printing to Terminal Example Output

```
int main(void) {
  xTaskCreate(vTask1, "Task 1", 1000, NULL, 1, NULL);
  xTaskCreate(vTask2, "Task 2", 1000, NULL, 1, NULL);
  vTaskStartScheduler();

  for( ;; );
}
```

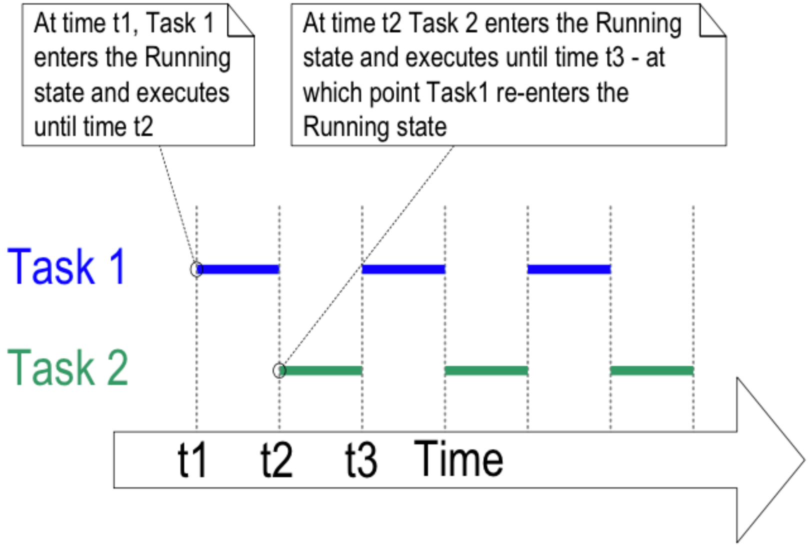


Figure 11. The actual execution pattern of the two Example 1 tasks

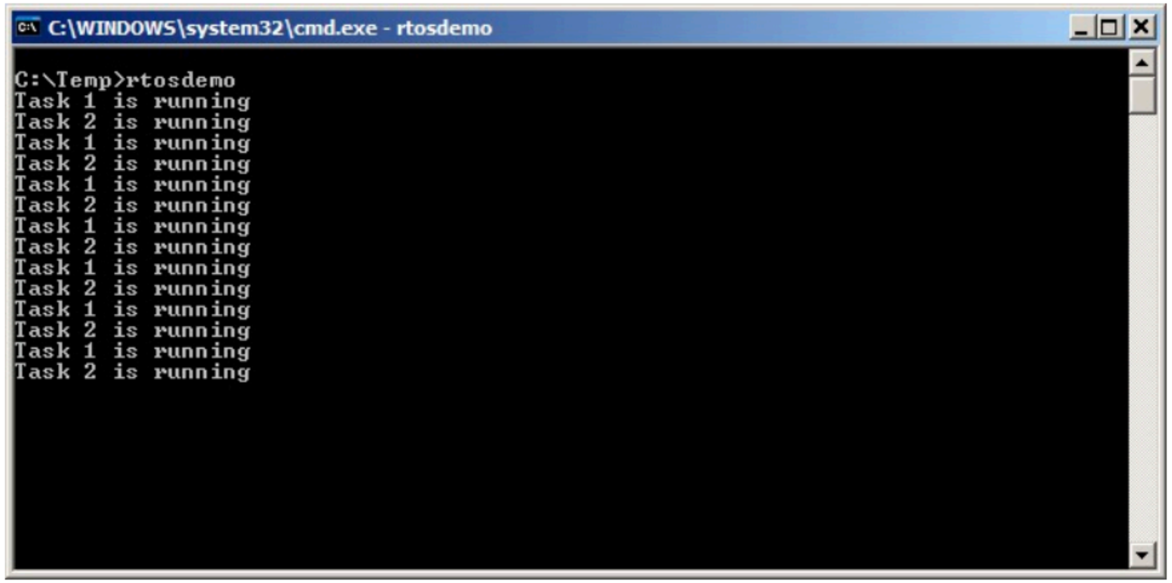


Figure 10. The output produced when Example 1 is executed[1]

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Example: Priorities

```
/* Define the strings that will be passed in as the task parameters.  These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create the first task at priority 1.  The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2, which is higher than a priority of 1.
    The priority is the second to last parameter. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* Will not reach here. */
    return 0;
}
```

Listing 21. Creating two tasks at different priorities

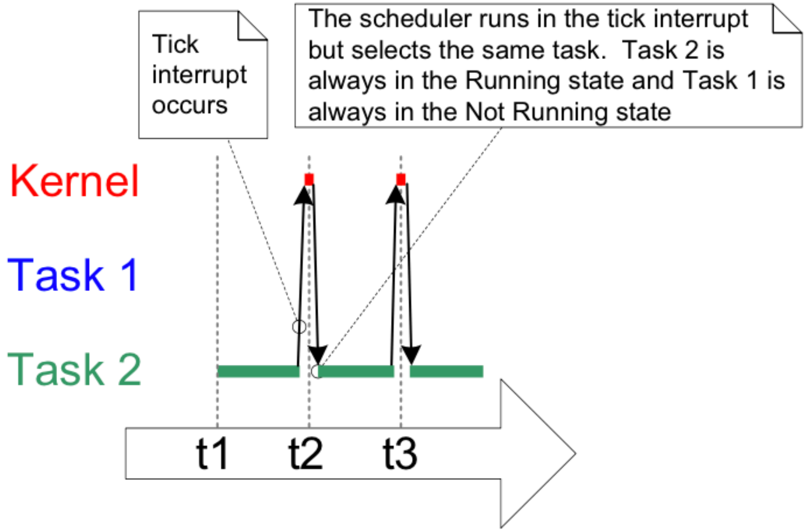


Figure 14. The execution pattern when one task has a higher priority than the other

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Revisiting Not Running State

- Three Options
  - Suspended
  - Ready
  - Blocked

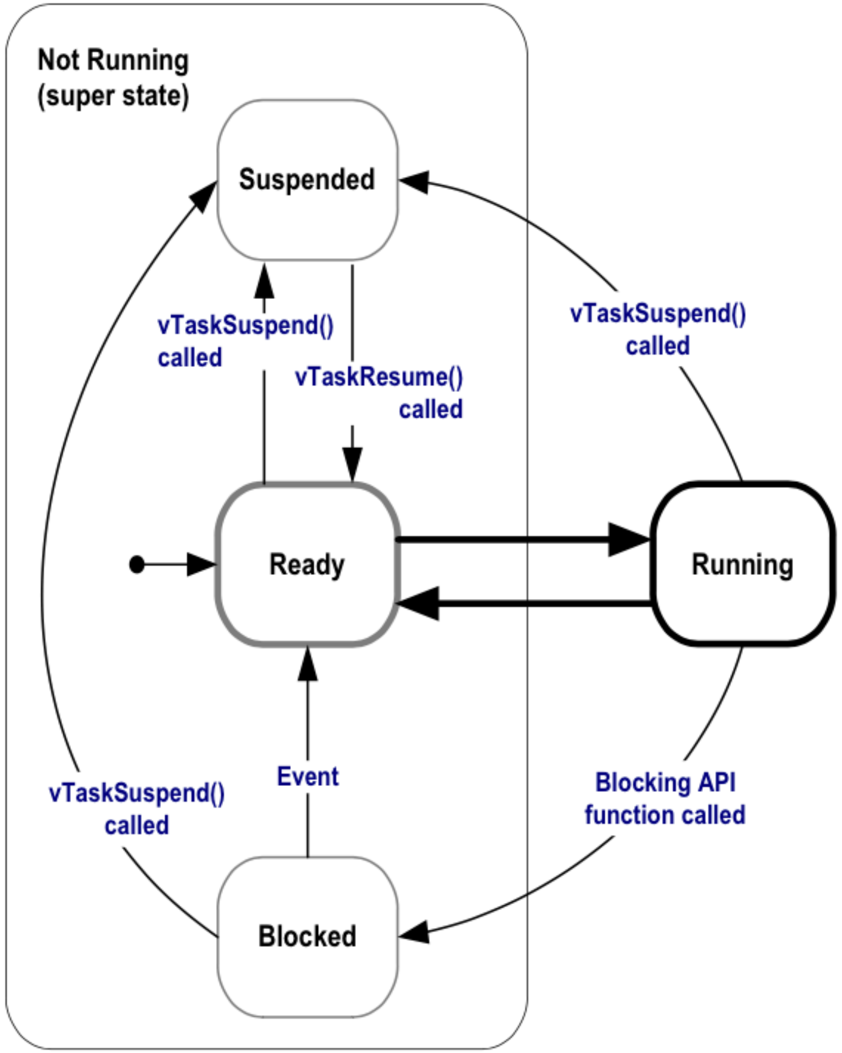


Figure 15. Full task state machine

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Example: Printing with better delay using blocked state

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period.  This time a call to vTaskDelay() is used which places
        the task into the Blocked state until the delay period has expired.  The
        parameter takes a time specified in 'ticks', and the pdMS_TO_TICKS() macro
        is used (where the xDelay250ms constant is declared) to convert 250
        milliseconds into an equivalent time in ticks. */
        vTaskDelay( xDelay250ms );
    }
}
```

Listing 23. The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

Example: Sending tasks to blocked state

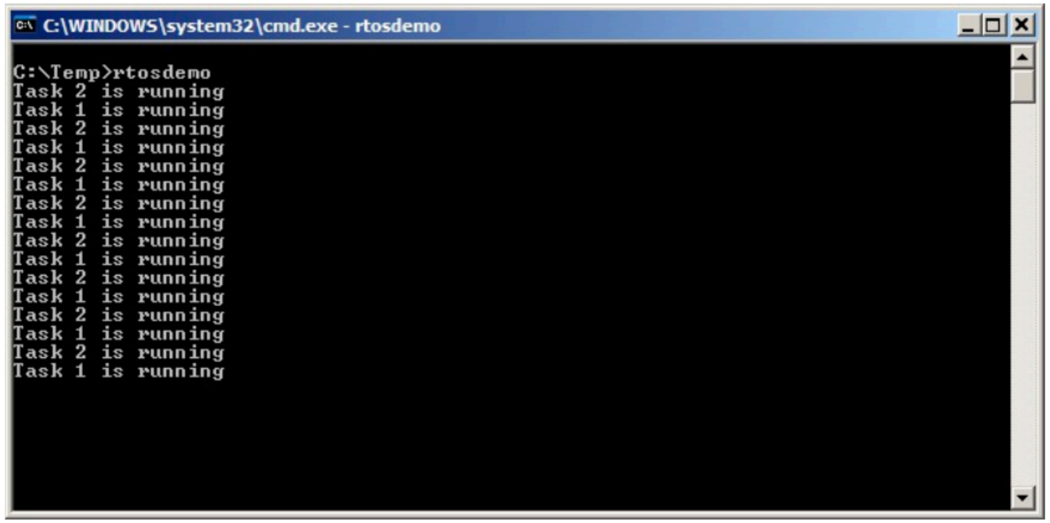


Figure 16. The output produced when Example 4 is executed

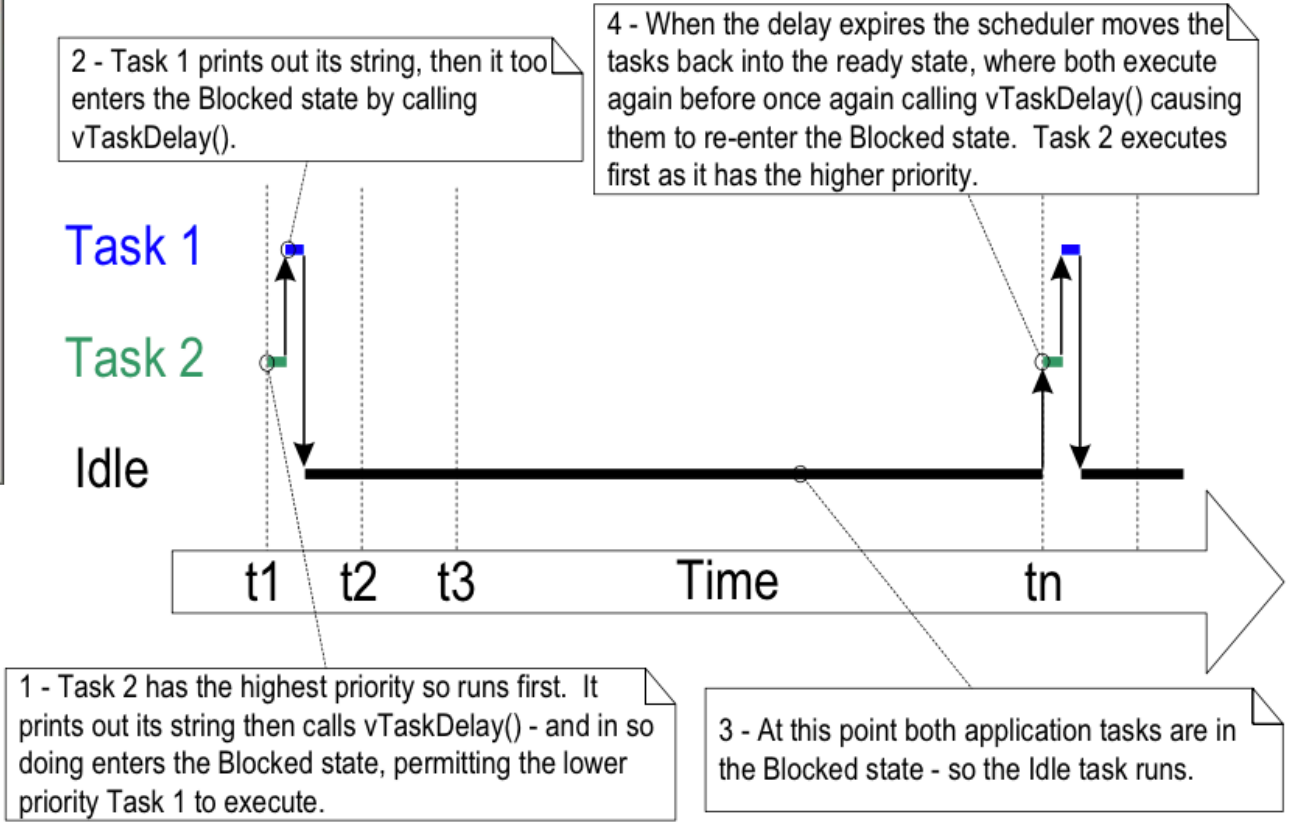


Figure 17. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.

FreeRTOSConfig.h

- Configuration file used to set some of the common options for the kernel

```
// <o>Minimal stack size [words] <0-65535>
// <i> Stack for idle task and default task stack in words.
// <i> Default: 128
#define configMINIMAL_STACK_SIZE ((uint16_t)(128))

// <o>Total heap size [bytes] <0-0xFFFFFFFF>
// <i> Heap memory size in bytes.
// <i> Default: 8192
#define configTOTAL_HEAP_SIZE ((size_t)8192)

// <o>Kernel tick frequency [Hz] <0-0xFFFFFFFF>
// <i> Kernel tick rate in Hz.
// <i> Default: 1000
#define configTICK_RATE_HZ ((TickType_t)1000)
```

```
/* Defines needed by FreeRTOS to implement CMSIS RTOS2 API. Do not change! */
#define configCPU_CLOCK_HZ (SystemCoreClock)
#define configSUPPORT_STATIC_ALLOCATION 1
#define configSUPPORT_DYNAMIC_ALLOCATION 1
#define configUSE_PREEMPTION 1
#define configUSE_TIMERS 1
#define configUSE_MUTEXES 1
#define configUSE_RECURSIVE_MUTEXES 1
#define configUSE_COUNTING_SEMAPHORES 1
#define configUSE_TASK_NOTIFICATIONS 1
#define configUSE_TRACE_FACILITY 1
#define configUSE_16_BIT_TICKS 0
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configMAX_PRIORITIES 56
#define configKERNEL_INTERRUPT_PRIORITY 255
```

Ex. 1: Task creation with LED blink

- Basic task creation workflow
  - Set up and initialization as usual
  - Write tasks according to specified prototype
  - In main
    - Call initialization functions
    - Create tasks
    - Start scheduler

Ex. 1: Task creation with LED blink

```
// Task to toggle LED
static void toggleLedTask(void *pvParameters)
{
    const TickType_t xDelay = pdMS_TO_TICKS(500);

    while (1)
    {
        /* Simply toggle the LED every xDelay ms,
blocking between each toggle. */
        toggleLED(LED_PIN);
        vTaskDelay(xDelay);
    }
}
```

Ex. 1: Task creation with LED blink

```
// Main function where initialization is performed and tasks are created.
int main()
{
    // Call initialization functions
    init_flash();   // Set up flash
    init_clock();   // Configure 84 MHz clock rate
    init_gpio();    // Initialize GPIO for LED

    // Create tasks
    const size_t xRegTestStackSize = 250U; // Set value for stack for each task.
    xTaskCreate(toggleLedTask,      // Task function
                "Blink_1",          // Optional name for task
                xRegTestStackSize,  // Task stack size
                (void*)&led_1,      // void pointer to optional parameters
                1,                  // Task priority
                NULL);              // Handle to created task

    // Start the scheduler
    vTaskStartScheduler();

    // Infinite while loop. Should never get here unless the scheduler fails to start.
    while (1);
}
```

Ex. 2: Passing parameters into task using pvParameters

02_passing_parameters_blink_led.c

```
// New type to hold information about LED
typedef struct param_led {
   uint32_t delay_ms;
   uint8_t led_pin;
} param_led;


// Create param_led struct to hold delay and pin number for LED.
param_led led_1 = {200, 5}; // delay_ms = 200, led_pin = 5


// Task to toggle LED
static void toggleLedTask(void *pvParameters)
{
    const param_led * led_info = (param_led *) pvParameters;
    const TickType_t xDelay = pdMS_TO_TICKS(led_info->delay_ms);

    while (1)
    {
        /* Simply toggle the LED every xDelay ms, blocking between each toggle. */
        toggleLED(led_info->led_pin);
        vTaskDelay(xDelay);
    }
}
```

Ex. 3: Multiple tasks with two serial prints

```
// Initialize and configure USART
void init_uart() {
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
  RCC->APB1ENR |= RCC_APB1ENR_USART2EN;

  // Configure PA2 and PA3 as alternate functions USART2
  GPIOA->MODER  &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3);
  GPIOA->MODER  |= (0b10 << GPIO_MODER_MODER2_Pos | 0b10 << GPIO_MODER_MODER3_Pos);
  GPIOA->AFR[0] |= (0b0111 << GPIO_AFRL_AFSEL2_Pos | 0b0111 << GPIO_AFRL_AFSEL3_Pos);

  USART->CR1 |= (USART_CR1_UE);
  USART->CR1 &= ~(USART_CR1_M | USART_CR1_OVER8);
  USART->CR2 &= ~(USART_CR2_STOP);

  // Set baud rate to 115200
  USART->BRR |= (22 << USART_BRR_DIV_Mantissa_Pos | 13 << USART_BRR_DIV_Fraction_Pos);
  USART->CR1 |= (USART_CR1_TE | USART_CR1_RE);
}

// Simple function to send characters over USART.
void sendChar(uint8_t data){
    USART->DR = (data & USART_DR_DR);
    while(!((USART->SR >> USART_SR_TC_Pos) & 1));
}
```

Ex. 3: Multiple tasks with two serial prints

```
#define USART USART2
#define UART_DELAY_MS 2000


// Strings to print from tasks.
const uint8_t str1[64] = "Hello from Task 1.\n";
const uint8_t str2[64] = "Hello from Task 2.\n";
```

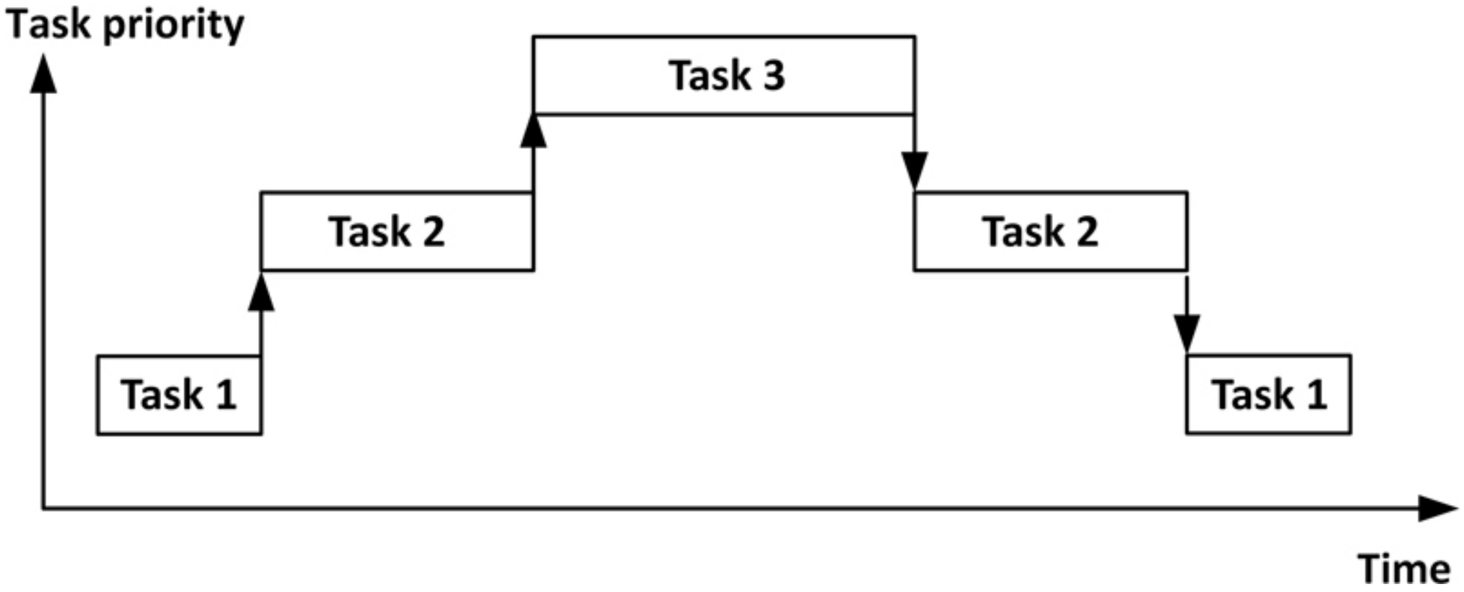
```
// Task to print string over USART
static void printStringTask(void *pvParameters) {
  uint8_t * str = (uint8_t *) pvParameters;
  const TickType_t xDelay =
pdMS_TO_TICKS(UART_DELAY_MS);
  int i = 0;

  while(1) {
    do {
      sendChar(str[i]);
      i++;
    }
    while(str[i] != 0);

    i = 0;

    vTaskDelay(xDelay);
  }
}
```

Ex. 3: Multiple tasks with two serial prints

- Which task prints first? How could you change this?
- Change duty cycle so that Task 1 prints once a second, Task 2 prints every other second.
  - How do you expect the tasks to execute now?

Preemptive scheduling

- Most common scheduling algorithm in real-time systems
- Tasks are assigned priorities
- Higher priority tasks can preempt lower priority tasks to take the CPU
- Need to be careful to assign priorities appropriately or you can starve lower priority tasks



Q: How are tasks and their priorities different than interrupts?

Ibrahim, Dogan. *ARM-Based Microcontroller Multitasking Projects: Using the FreeRTOS Multitasking Kernel*.

Ex. 4: Simple preemption example: poll button and blink LED and single print

```
// Task to poll button
static void pollButtonTask(void *pvParameters) {
  const TickType_t xDelay = pdMS_TO_TICKS(100); // Schedule every 100 ms
  volatile int i;

  while(1) {
    volatile int pin_val = 1;

    // Loop to check if the button is pressed (button is pulled low when pressed)
    // and blink LED rapidly while the button is pressed using a dummy loop.
    while(pin_val){
      pin_val = !((GPIOC->IDR >> BUTTON_PIN) & 1);
      if(pin_val) {
        toggleLED(LED_PIN);
        for(i=0; i < 400000; i++); // Dummy loop to do a delay.
      }
    }
    vTaskDelay(xDelay);
  }
}
```

Ex. 4: Simple preemption example: poll button and blink LED and single print

```
// Main function where initialization is performed and tasks are created.
int main()
{
    // Call initialization functions
    init_flash();   // Set up flash
    init_clock();   // Configure 84 MHz clock rate
    init_gpio();    // Initialize GPIO for LED
    init_uart();    // Initialize UART

    // Create tasks
    const size_t xRegTestStackSize = 250U; // Set value for stack for each task.
    xTaskCreate(toggleLedTask,      // Task function
                "Blink_1",          // Optional name for task
                xRegTestStackSize,  // Task stack size
                (void*)&led_1,      // void pointer to optional parameters
                1,                  // Task priority
                NULL);              // Handle to created task

    xTaskCreate(printStringTask, "Print_Test1", xRegTestStackSize, (void*)&str1, 2, NULL);
    xTaskCreate(pollButtonTask, "Poll_Button", xRegTestStackSize, NULL, 3, NULL);

    // Start the scheduler
    vTaskStartScheduler();

    // Infinite while loop. Should never get here unless the scheduler fails to start.
    while (1);
}
```

Ex. 4: Questions

- What will execution look like?
- Will the other tasks get any CPU time?
- What behavior do you expect to see if we change the priority of the print task such that it is higher than that of the poll button task?
- Task states
  - What happens if you press and release the button quickly? When does the print occur?
  - What happens if you hold the button for several seconds and then release it? When does the print occur then?
  - Why are these two cases different?
- How can we make this example more efficient?

Summary

- To understand the following key concepts of Real-time Operating Systems through examples in FreeRTOS
  - Task creation – tasks are like wrappers for C functions. They should never return and yield to the scheduler once they are done doing their work.
  - Basic scheduling – The scheduler decides what task should be running at any given time.
  - Task priorities and preemption – Task priorities help the scheduler decide between the importance of different tasks. Can be useful to distinguish between hard and soft deadlines and make sure they are met appropriately.

References

- Ibrahim, Dogan. *ARM-Based Microcontroller Multitasking Projects: Using the FreeRTOS Multitasking Kernel*. Netherlands, Elsevier Science, 2020.
- Barry, Richard. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. 2016.