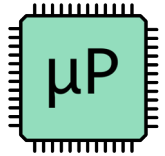
Introduction to Real-time Operating Systems

Lecture 23

Microprocessor-based Systems (E155)

Prof. Josh Brake



Outline

- Motivation for multitasking and real-time operating systems
- Introduction to key concepts
  - Tasks
  - Scheduling
  - Semaphores
  - Queues
  - Interrupts and Events
- Introduction to FreeRTOS

Multitasking Scenarios

- Print information to a display in response to keyboard input
- Car
  - Airbag response
  - Braking system
- Robotics
  - Flight system in a drone
  - Control or signal processing algorithms

Soft vs. hard real-time requirements

- Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless.
- Hard real-time requirements are those that state a time deadline—and breaching the deadline would result in absolute failure of the system.

Why multitasking?

- Bare-metal programming
  - Setup and initialization – runs once
  - Infinite loop – runs continuously and handles main tasks
- Bare-metal + interrupts
  - Can now incorporate additional functionality which quickly responds to inputs and can guarantee that we don’t miss important events.
  - Examples: Receiving and processing UART data, catching button inputs
- But as we build more and more complicated programs, it is hard to guarantee specific timing constraints are met.
  - Many different scenarios
  - Lots of edge cases which makes things very difficult to debug.

Why multitasking?

- Enter the concept of multitasking
  - Multitasking: means that several tasks (or programs) are processed in parallel on the same CPU
- Only have a single core on your microcontroller so this is not true parallelism, just swapping different tasks in and out
- Operations inside your bare-metal infinite loops are tasks.
  - For example, consider that you want to blink 2 LEDs at different frequencies. In bare-metal, you could have your infinite loop use timers to poll a timer and then toggle the LEDs based on the current time.
  - Works, but inefficient as the processor is always running.

Task States

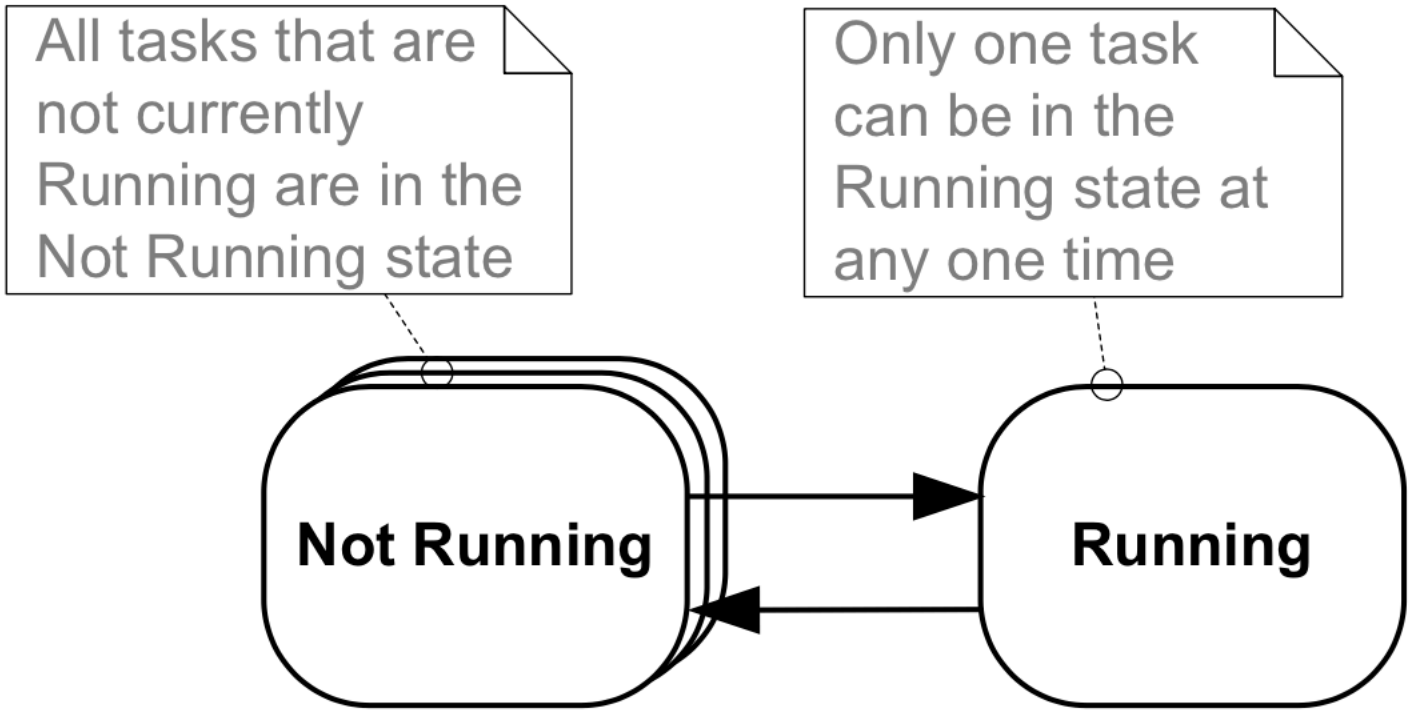


Figure 9. Top level task states and transitions

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

The Mudd Multitasking Kernel

- You are taking E155, Clinic, and an HSA. In addition, you want to sleep 8 hours a night and have time to hang out with your friends playing board games over Zoom.
- Imagine you must manage your clinic mid-year report, MicroPs final project, and must read a book and write a paper for your HSA in addition to chatting with your friends. You only have one brain.
- What are different ways you can manage your tasks?

Scheduling algorithms

- The scheduling algorithm decides which task is running on the core.
- Three main algorithms
  - Co-operative scheduling
  - Round-robin scheduling
  - Preemptive scheduling

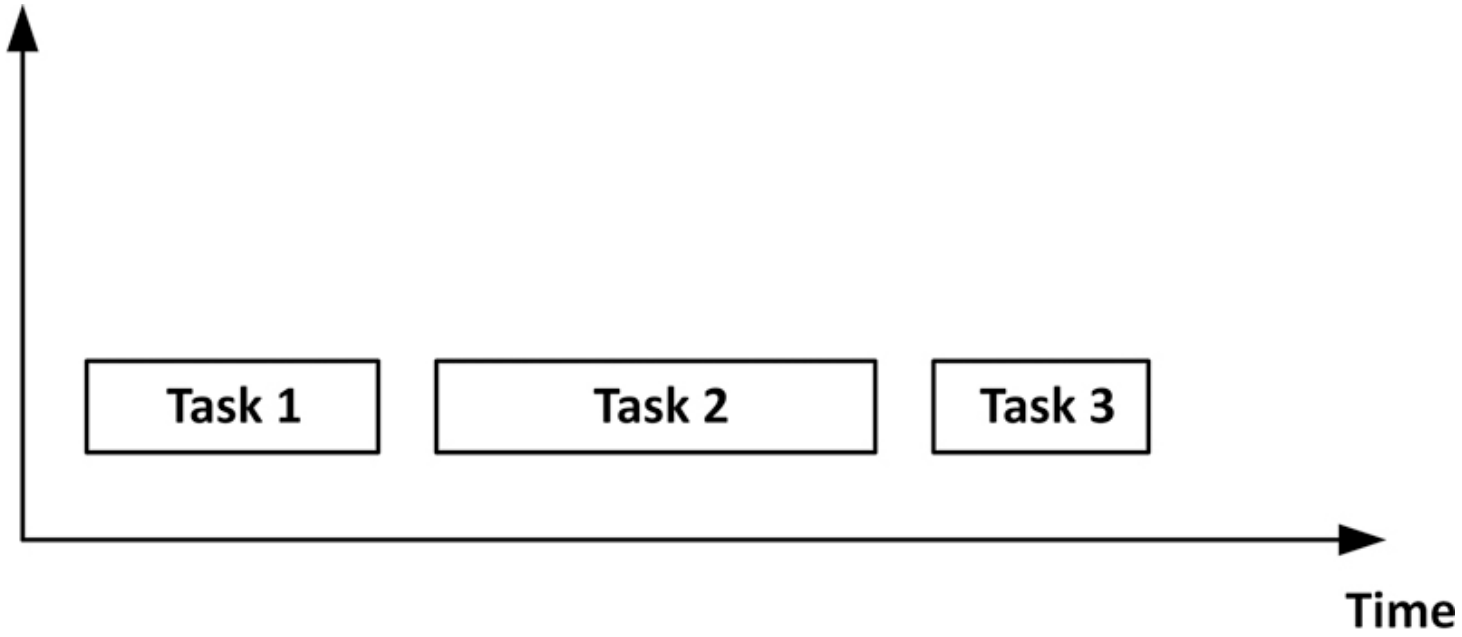
Co-operative scheduling

```
Task1() {
  // Task 1 code
}

Task2() {
  // Task 2 code
}

Task3() {
  // Task 3 code
}

while(1) {
  Task1();
  Task2();
  Task3();
}
```



Each task must yield control or else it can starve all other tasks.

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

Co-operative scheduling

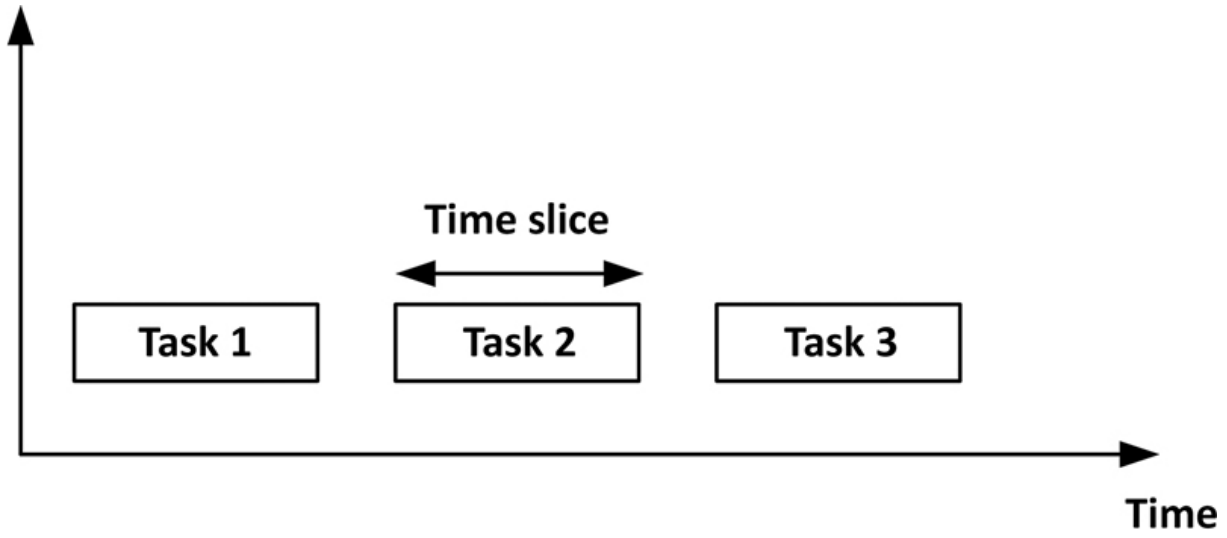
- Tasks must not block the overall execution, for example, by using delays or waiting for some resources and not releasing the CPU.
- The execution time of each tasks should be acceptable to other tasks.
- Tasks should exit as soon as they complete their processing.
- Tasks do not have to run to completion and they can exit for example before waiting for a resource to be available.
- Tasks should resume their operations from the point after they release the CPU.

Round-robin scheduling

- The scheduler creates a periodic time slice and equally divides CPU use between tasks.

Advantages:
- It is easy to implement.
- Every task gets an equal share of the CPU.
- Easy to compute the average response time.

Disadvantages
- It is not generally good to give the same CPU time to each task.
- Some important tasks may not run to completion.
- Not suitable for real-time systems where tasks usually have different processing requirements.

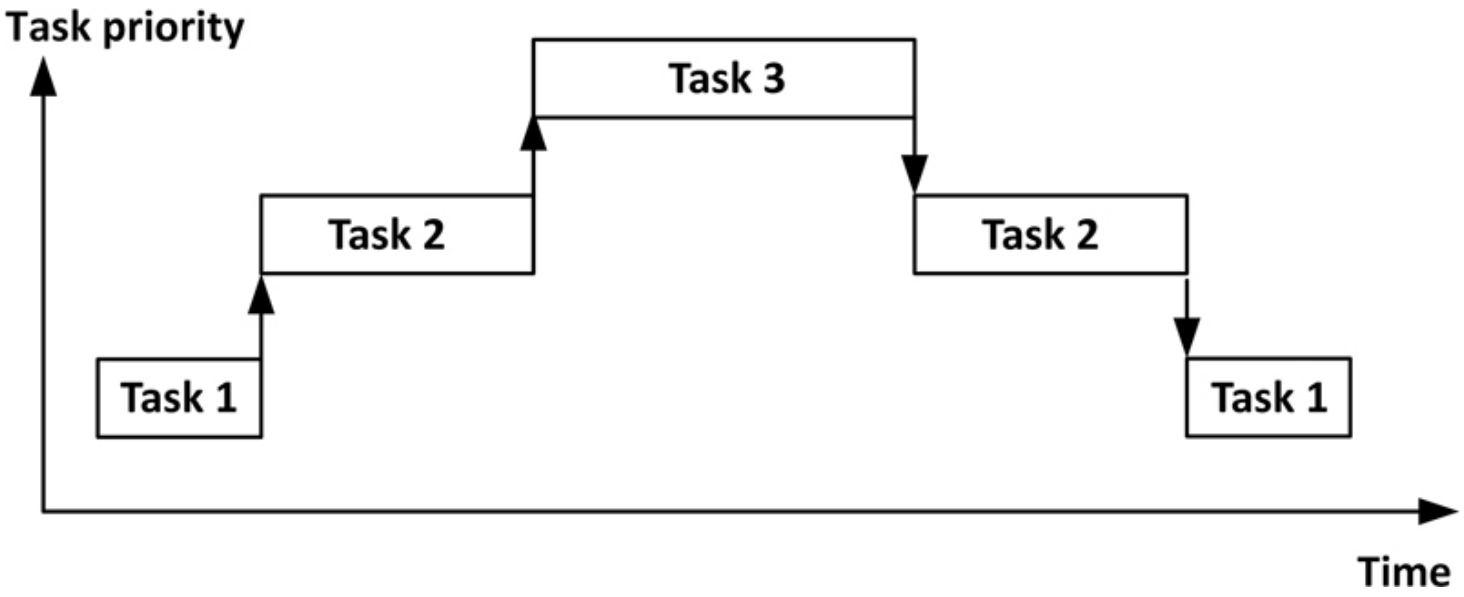


Ibrahim, Dogan. *ARM-Based Microcontroller Multitasking Projects: Using the FreeRTOS Multitasking Kernel*.

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*.

Scheduling Algorithm Goals

- Be fair such that each process gets a fair share of the CPU.
- Be efficient by keeping the CPU busy. The algorithm should not spend too much time to decide what to do.
- Maximize throughput by minimizing the time users must wait.
- Be predictable so that same tasks take the same time when run multiple times.
- Minimize response time.
- Maximize resource use.
- Enforce priorities.
- Avoid starvation.

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

Other scheduling algorithms

- First-come first-served
- Shortest time remaining first
- Longest time remaining first
- Multilevel queue scheduling
- Dynamic priority scheduling


free
RTOS

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

```
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

```
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. */
        }
    }
}
```

```
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. */
        }
    }
}
```

```
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( ;; );
}
```

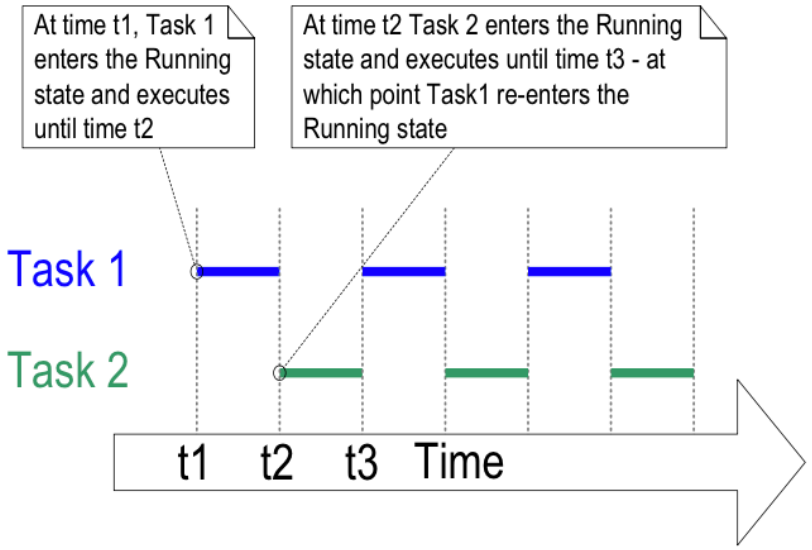


Figure 11. The actual execution pattern of the two Example 1 tasks

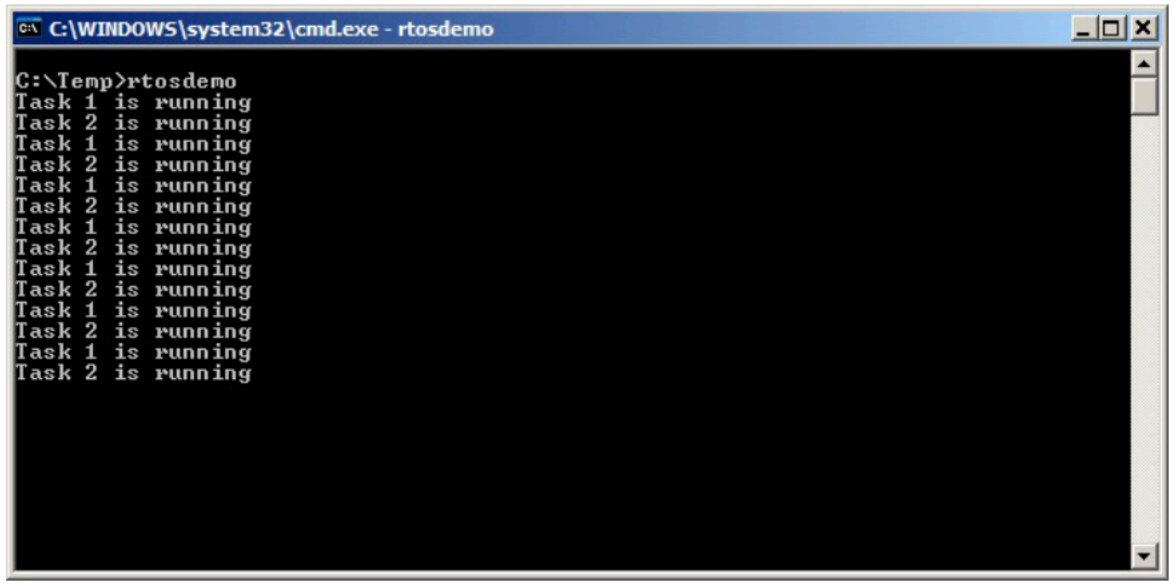


Figure 10. The output produced when Example 1 is executed[1]

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

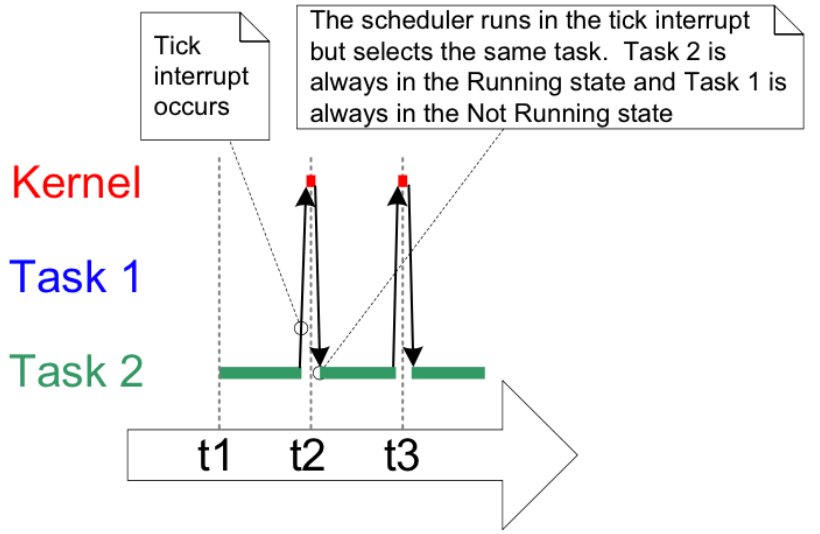


Figure 14. The execution pattern when one task has a higher priority than the other

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

Revisiting Not Running State

- Three Options
  - Suspended
  - Ready
  - Blocked

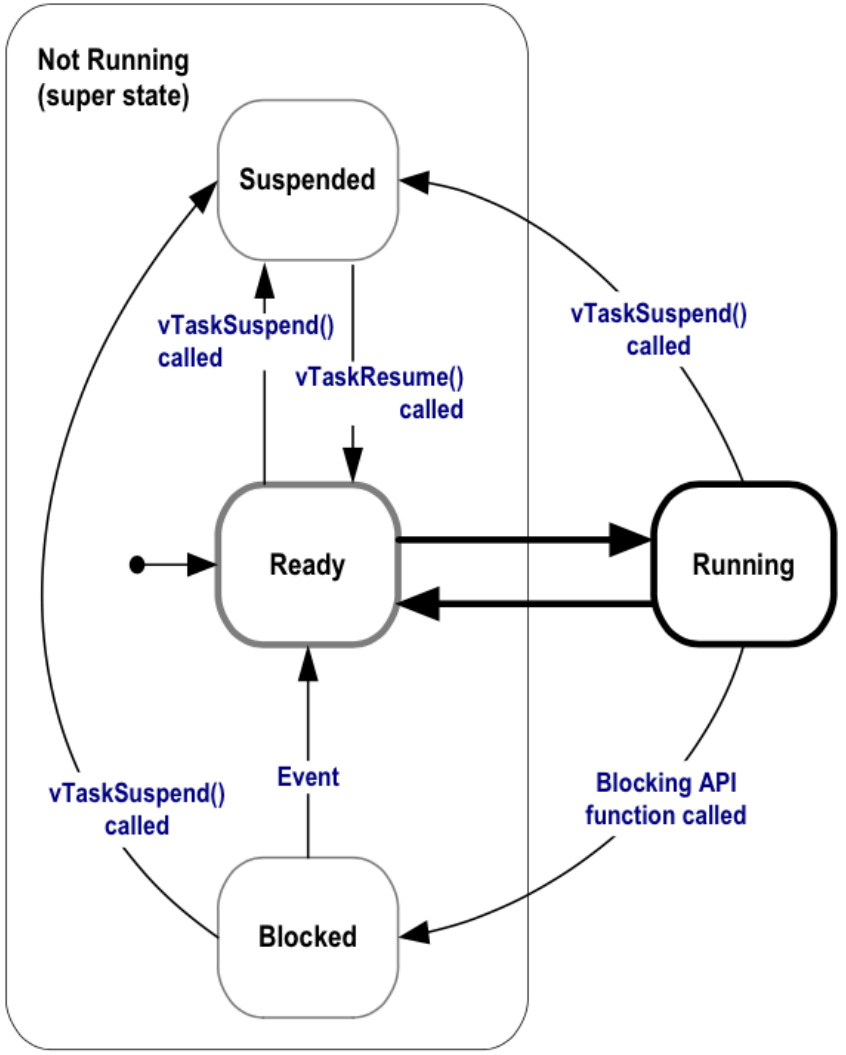


Figure 15. Full task state machine

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

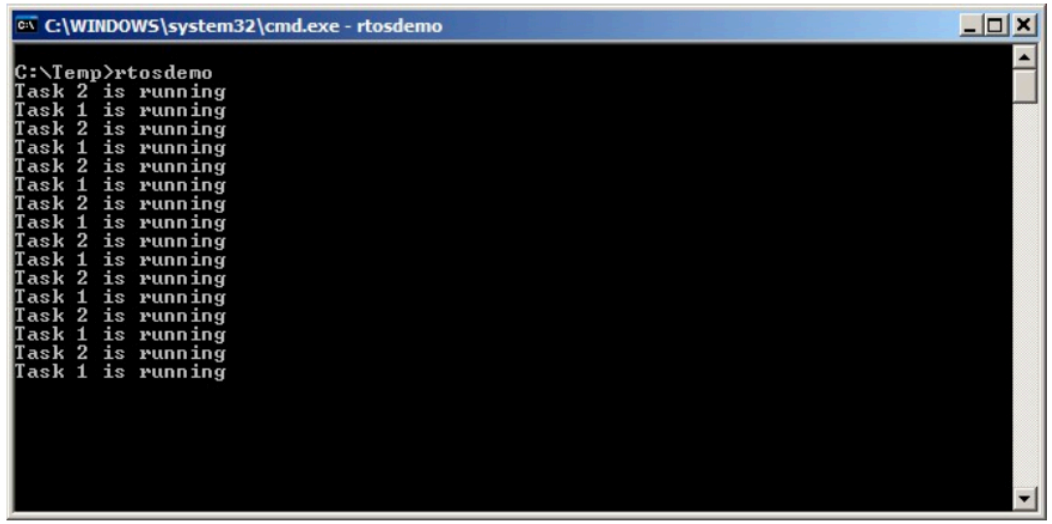


Figure 16. The output produced when Example 4 is executed

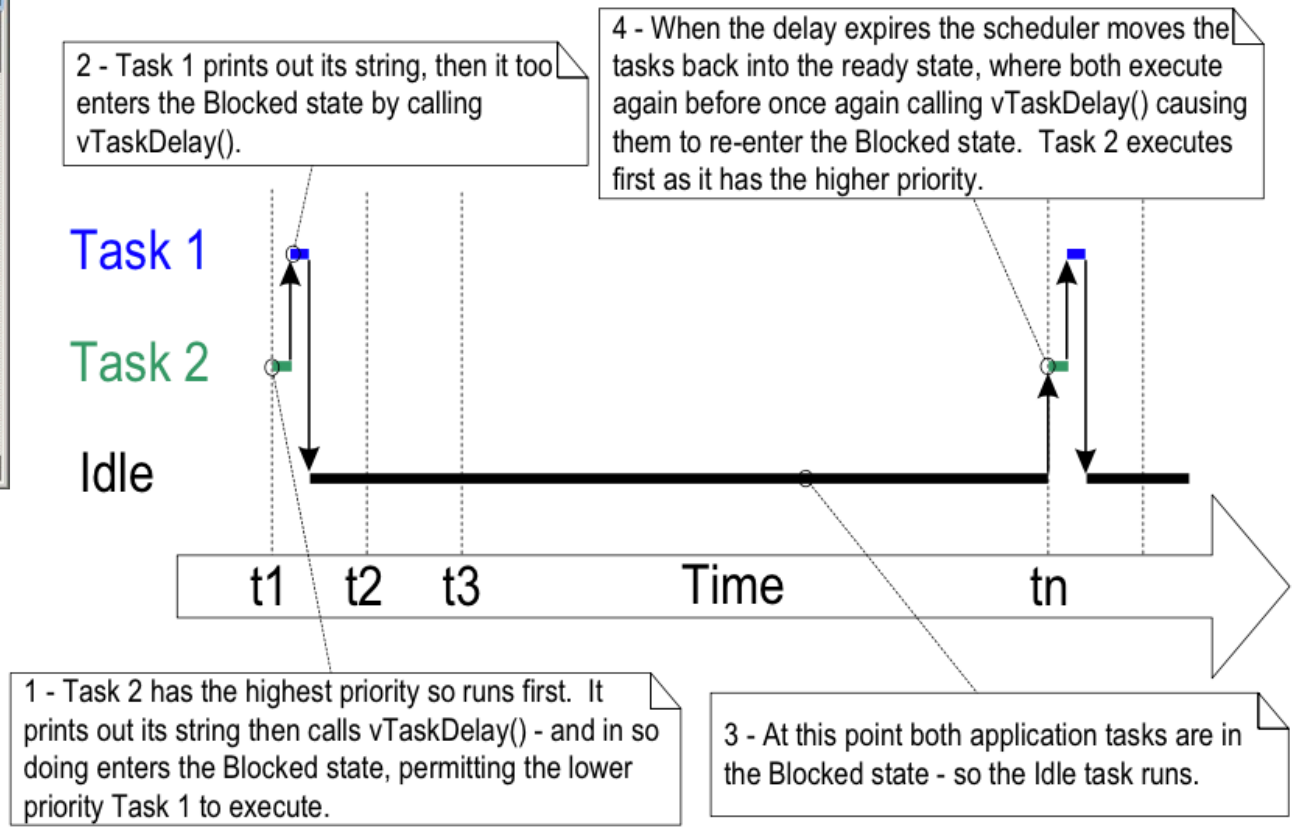


Figure 17. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

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

Key Terms in Real-time Systems

- Tasks – C functions which indicate things to do. Implemented as infinite loops.
- Scheduling – The process of determining what task is currently running
- Semaphores – an abstract data type which controls access to a resource used by multiple tasks
- Queues – A way to communicate between tasks (Chapter 4 of *Mastering FreeRTOS*)
- Interrupts and Events – how to safely integrate interrupts with the RTOS kernel (Chapter 6 of *Mastering FreeRTOS*)

Summary

- Multitasking is an important concept in advanced embedded systems
  - Have timing constraints that must be met (both soft and hard deadlines)
  - Hard to debug and manage systems with increasing complexity while guaranteeing all deadlines are met.
- Real-time operating systems introduce a scheduler which enables the programmer to efficiently use CPU cycles while ensuring deadlines are met.
- FreeRTOS is an open and accessible platform to learn RTOS concepts like tasks, queues, semaphores, and resource management.

Next Up

- For Wednesday:
  - Read Chapter 3 of *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. (available here)
  - Download demo project code from Github (link to be provided)
- Course evaluations

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.

Lecture Feedback

- What is the most important thing you learned in class today?
- What point was most unclear from lecture today?

https://forms.gle/Ay6MkpZ6x3xsW2Eb8