



Real Time Operating System Introduction & “FreeRTOS”

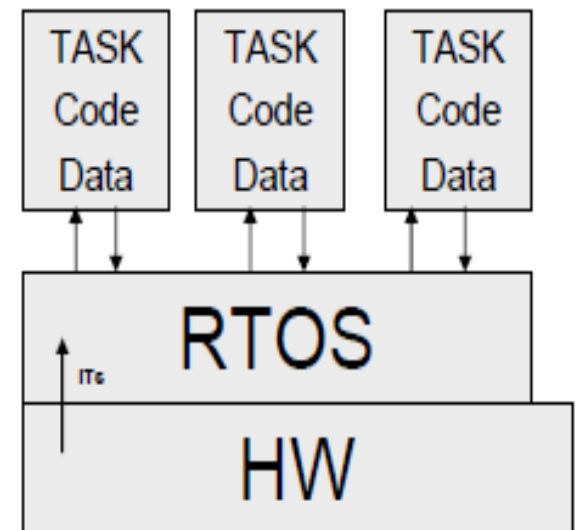
Sherif Hammad



Agenda

- **RTOS Basics**
- **RTOS sample State Machine**
- **RTOS scheduling criteria**
- **RTOS optimization criteria**
- **Soft/Hard Real Time requirements**
- **Tasks scheduling**

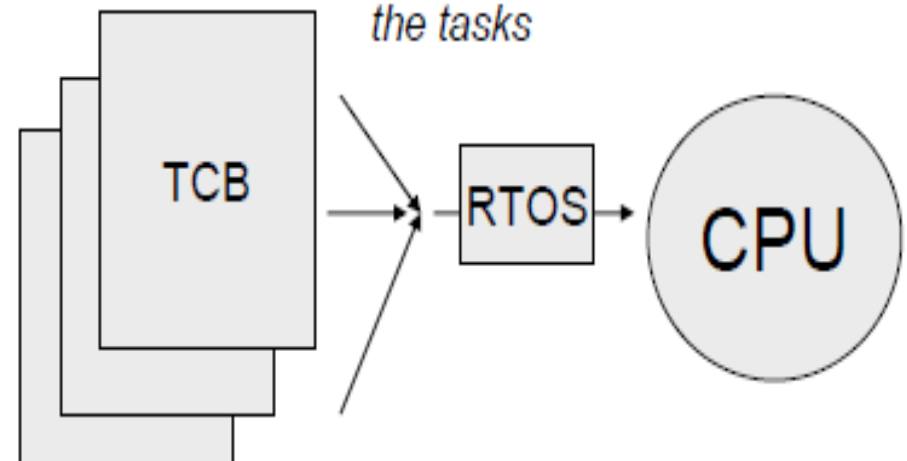
- Kernel: schedules tasks
- Tasks: concurrent activity with its own state (PC, registers, stack, etc.)



Tasks

- Tasks = Code + Data + State (context)
- Task State is stored in a Task Control Block (TCB) when the task is not running on the processor
- Typical TCB:

ID
Priority
Status
Registers
Saved PC
Saved SP

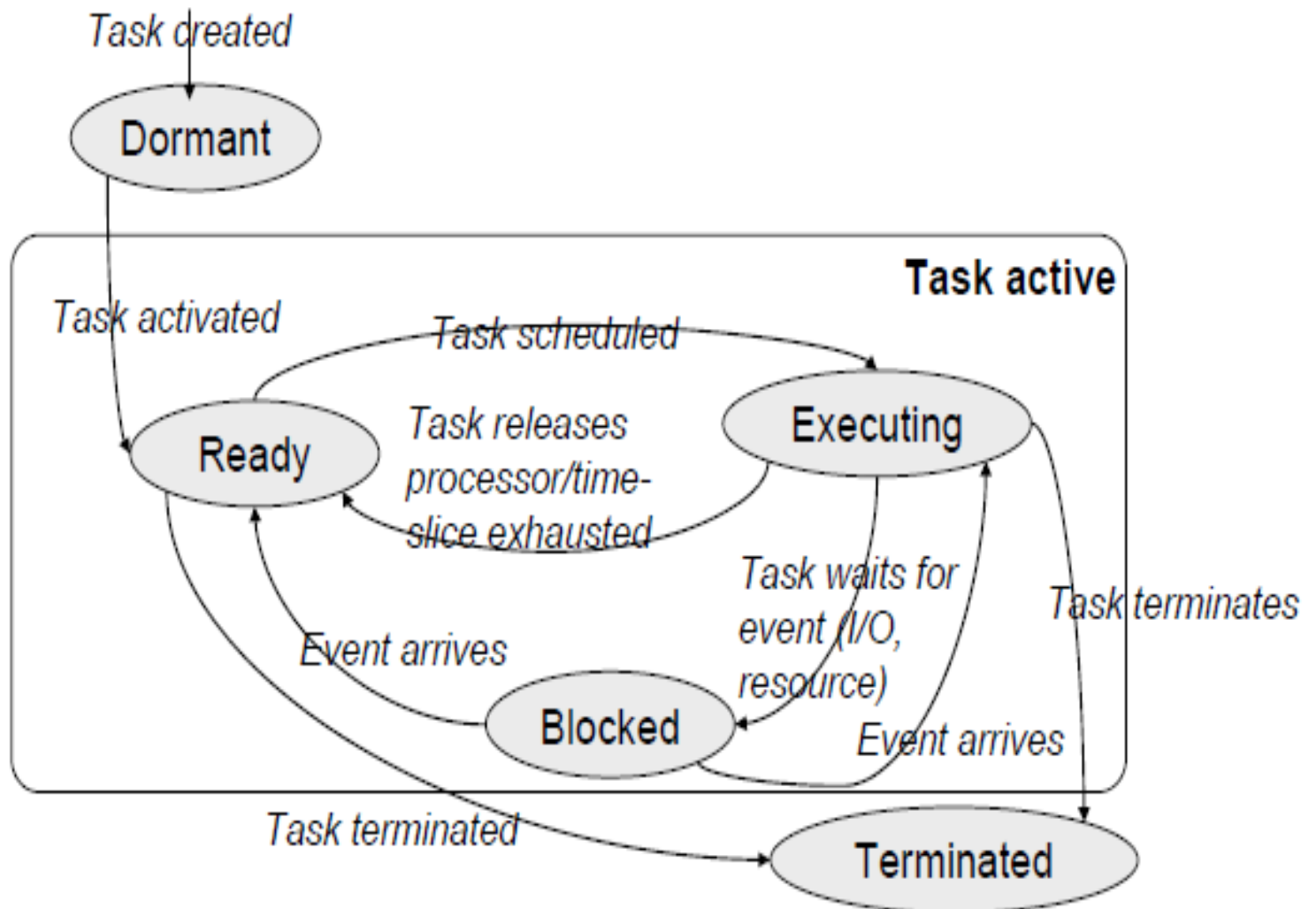


Task states

- **Executing**: running on the CPU
- **Ready**: could run but another one is using the CPU
- **Blocked**: waits for something (I/O, signal, resource, etc.)
- **Dormant**: created but not executing yet
- **Terminated**: no longer active

The RTOS implements a Finite State Machine for each task, and manages its transitions.

Task State Transitions



RTOS Scheduler

- Implements task state machine
- Switches between tasks
- Context switch algorithm:
 1. Save current context into current TCB
 2. Find new TCB
 3. Restore context from new TCB
 4. Continue
- Switch between EXECUTING -> READY:
 1. Task yields processor voluntarily: **NON-PREEMPTIVE**
 2. RTOS switches because of a higher-priority task/event: **PREEMPTIVE**

CPU Scheduling Criteria

- **CPU Utilization:** CPU should be as busy as possible (40% to 90%)
- **Throughput:** No. of processes per unit time
- **Turnaround time:** For a particular process how long it takes to execute. (Interval between time of submission to completion)
- **Waiting time:** Total time process spends in ready queue.
- **Response:** First response of process after submission

Optimization criteria

- It is desirable to
 - Maximize CPU utilization
 - Maximize throughput
 - Minimize turnaround time
 - Minimize start time
 - Minimize waiting time
 - Minimize response time
- In most cases, we strive to optimize the average measure of each metric
- In other cases, it is more important to optimize the minimum or maximum values rather than the average

Soft/Hard Real Time

- **Soft real-time requirements: state a time deadline—but breaching the deadline would not render the system useless.**
- **Hard real-time requirements: state a time deadline—and breaching the deadline would result in absolute failure of the system.**
- **Cortex-M4 has only one core executing a single Thread at a time.**
- **The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer.**
- **Application designer could assign higher priorities to hard-real-time-threads and lower priorities to soft real-time**

Why Use a Real-time Kernel?

- **Abstracting away timing information**
- **Maintainability/Reusability/Extensibility**
- **Modularity**
- **Team development**
- **Improved efficiency (No Polling)**


- **Idle time:**

The Idle task is created automatically when the kernel is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

- **Flexible interrupt handling:**

Interrupt handlers can be kept very short by deferring most of the required processing to handler RTOS tasks.

Task Functions

- **Arbitrary naming: Must return void: Must take a void pointer parameter:**


```
void ATaskFunction( void *pvParameters );
```
- **Normally run forever within an infinite loop, and will not exit.**
- **FreeRTOS tasks must not be allowed to return from their implementing function in any way—they must not contain a 'return' statement and must not be allowed to execute past the end of the function.**
- **A single task function definition can be used to create any number of tasks—each created task being a separate execution instance with its own stack and its own copy of any automatic (stack) variables defined within the task itself.**

Task Functions

```
void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared just as per a normal function.  Each instance
    of a task created using this function will have its own copy of the
    iVariableExample variable.  This would not be true if the variable was
    declared static - in which case only one copy of the variable would exist
    and this copy would be shared by each created instance of the task. */
    int iVariableExample = 0;

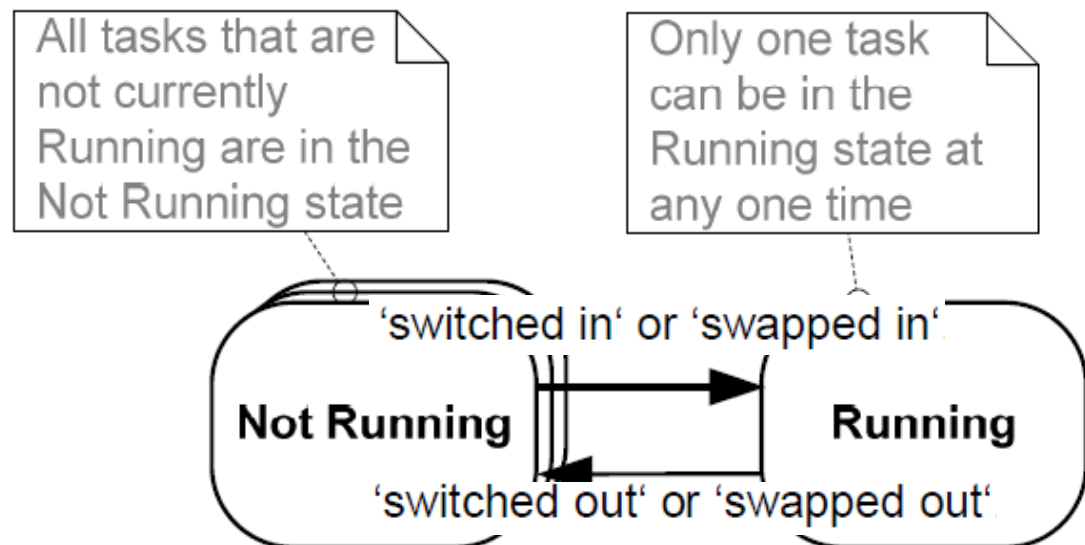
    /* A task will normally be implemented as an infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop
    then the task must be deleted before reaching the end of this function.
    The NULL parameter passed to the vTaskDelete() function indicates that
    the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

Listing 2. The structure of a typical task function

Top Level Task States

- When a task is in the **Running** state, the processor is executing its code.
- When a task is in the **Not Running** state, its status having been saved ready for it to resume execution the next time the scheduler decides it should enter the **Running** state.
- When a task resumes execution, it does so from the instruction it was about to execute before it last left the **Running** state.





Creating Tasks

The xTaskCreate() API Function

```
int main( void )
{
    /* Create one of the two tasks. Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1, /* Pointer to the function that implements the task. */
                  "Task 1", /* Text name for the task. This is to facilitate
                           debugging only. */
                  240,     /* Stack depth in words. */
                  NULL,    /* We are not using the task parameter. */
                  1,       /* This task will run at priority 1. */
                  NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```




Example 1: Task Functions



```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\n";
    volatile unsigned long ul;

    /* 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\n";
    volatile unsigned long ul;

    /* 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. */
        }
    }
}
```


Run Example 1

```

Console Problems Memory Red Trace Preview
Example01 Debug [C/C++ MCU Application] C:\E\Dev\FreeRTOS\DOC\Projects\191-ApplicationNo

Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running

```

Figure 2. The output produced when Example 1 is executed

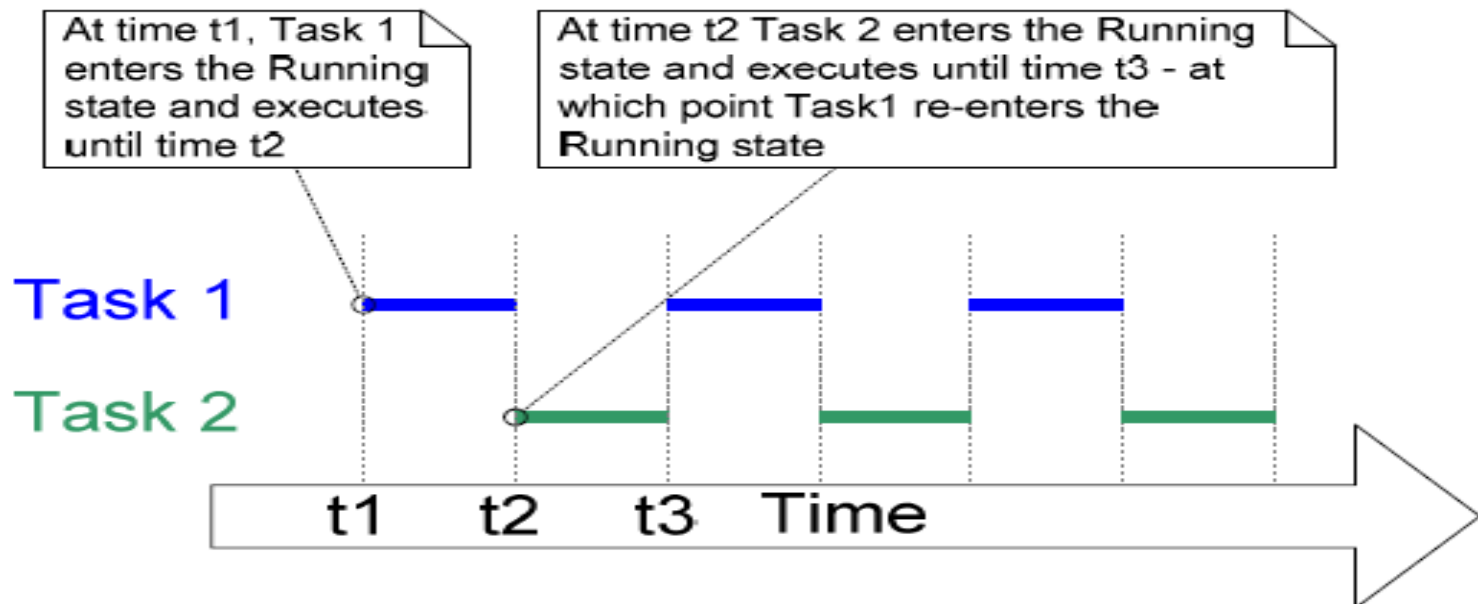


Figure 3. The execution pattern of the two Example 1 tasks

- To be able to select the next task to run, the scheduler itself must execute at the end of each time slice.
- A periodic interrupt, called the tick interrupt, is used for this purpose.
- The length of the time slice is effectively set by the tick interrupt frequency, which is configured by the `configTICK_RATE_HZ` compile time configuration constant in `FreeRTOSConfig.h`.

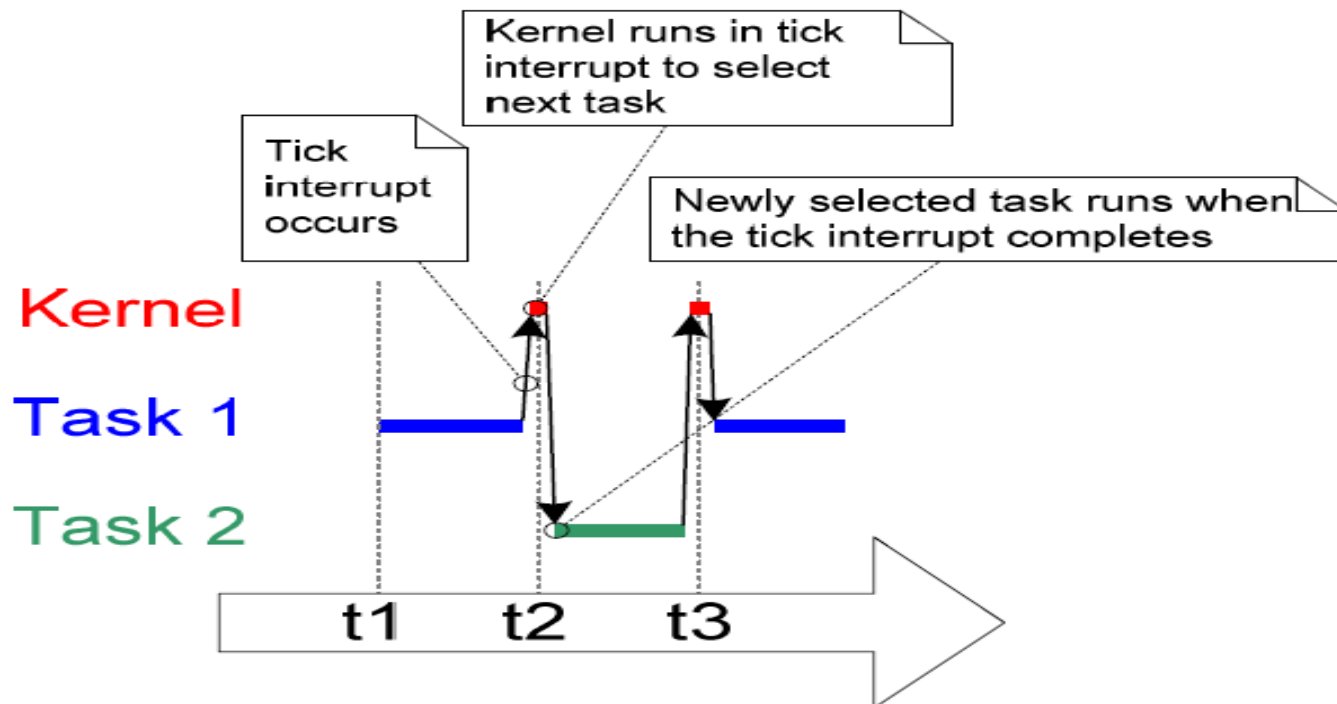


Figure 4. The execution sequence expanded to show the tick interrupt executing