Real Time Operating System Introduction & "FreeRTOS"

Sherif Hammad

Agenda

- **Tasks priorities**
- **Task State Machine**
- **Periodic tasks**
- **Tasks Blocking**

Example 1: Task Functions (Cont.)

```
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

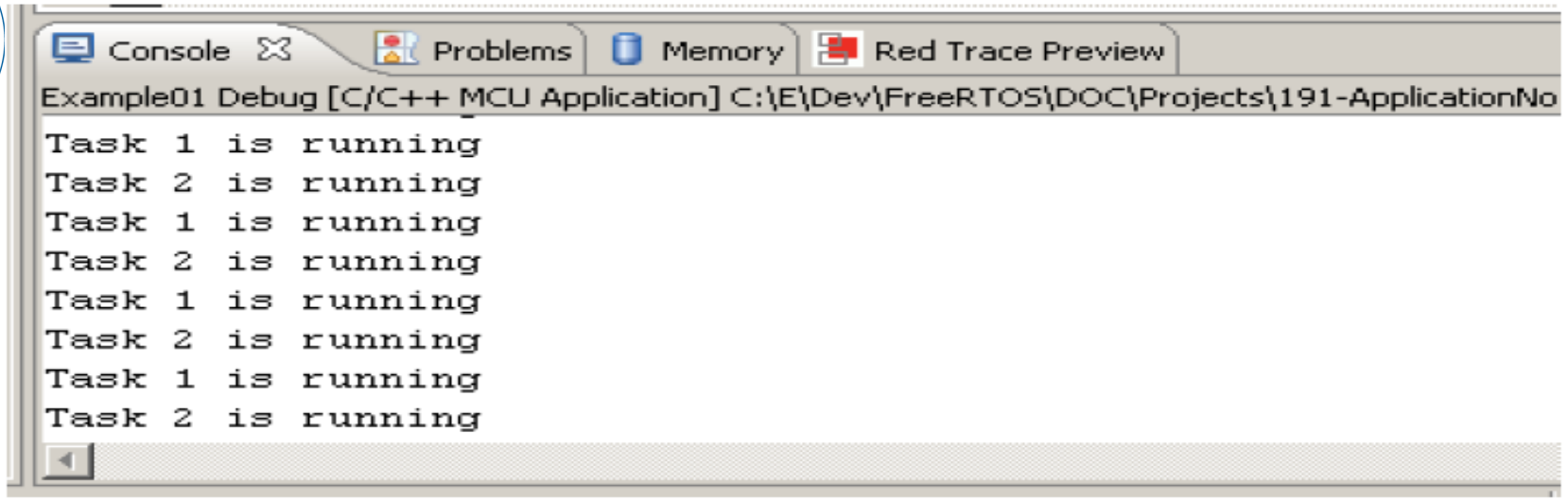


Figure 2. The output produced when Example 1 is executed

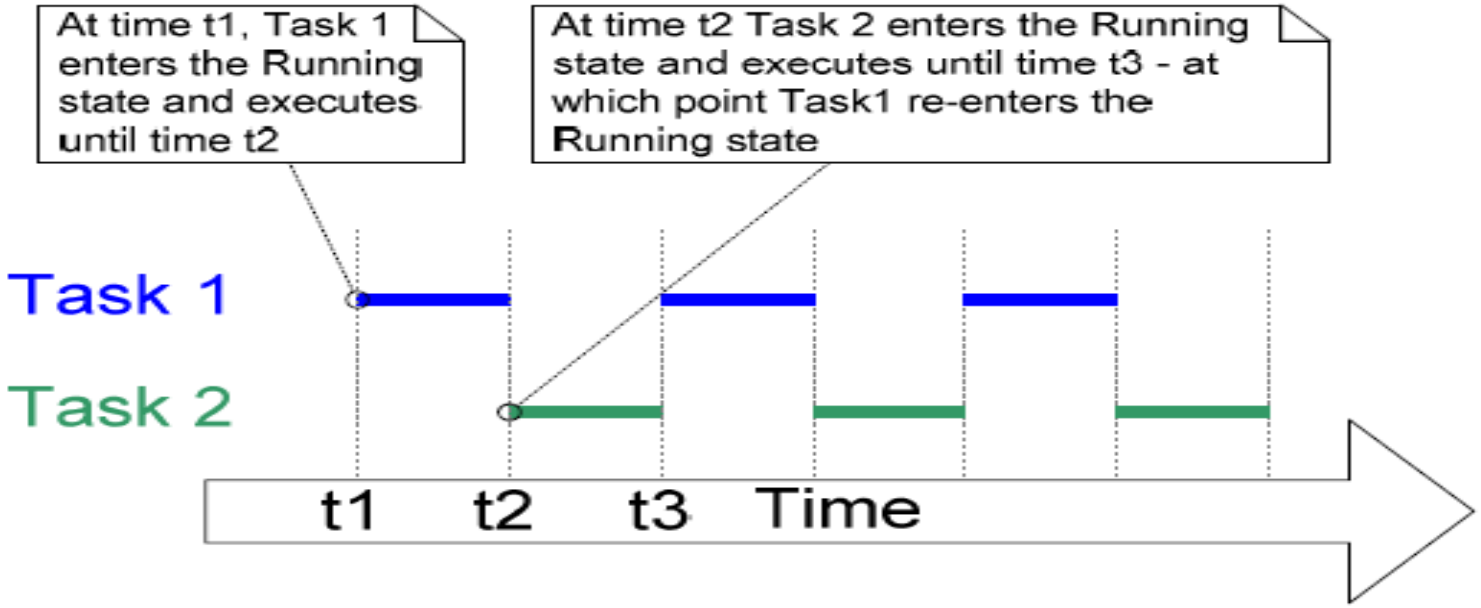


Figure 3. The execution pattern of the two Example 1 tasks

Tick Interrupt

- 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.

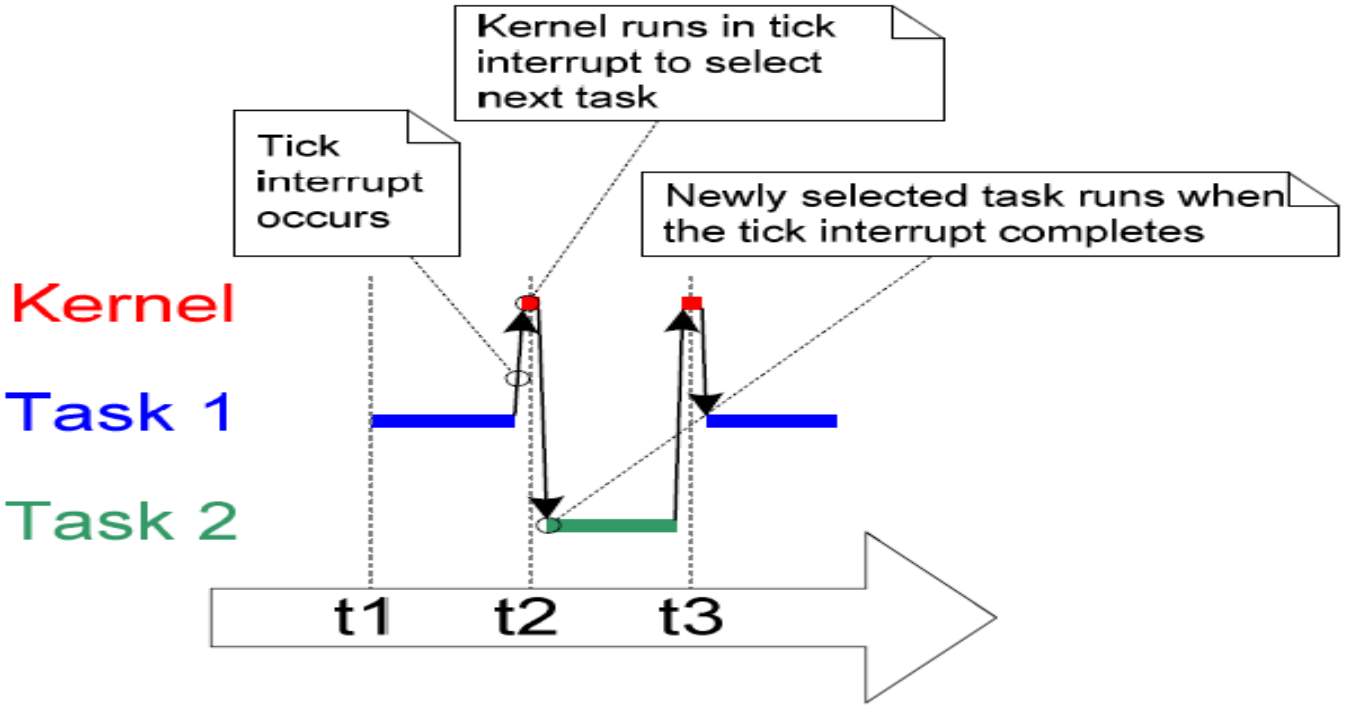


Figure 4. The execution sequence expanded to show the tick interrupt executing

Task Creation After Schedule Started
(From Within Another Task)

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

    /* If this task code is executing then the scheduler must already have
    been started.  Create the other task before we enter the infinite loop. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, NULL );

    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. */
        }
    }
}
```

Example 2: Single Task Function
"Instantiated Twice" (Two Task Instants)

```
/* 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\n";
static const char *pcTextForTask2 = "Task 2 is running\n";

int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(    vTaskFunction,          /* 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 */
                    (void*)pcTextForTask1,  /* Pass the text to be printed into the
                                               task using the task parameter. */
                    1,                      /* This task will run at priority 1. */
                    NULL );                 /* We are not using the task handle. */

    /* Create the other task in exactly the same way.  Note this time that multiple
    tasks are being created from the SAME task implementation (vTaskFunction).  Only
    the value passed in the parameter is different.  Two instances of the same
    task are being created. */
    xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 1, NULL );

    /* Start the scheduler so our 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 2: Single Task Function

"Instantiated Twice" (Two Task Instants)

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile unsigned long ul;

    /* 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. */
        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 exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Task Priorities

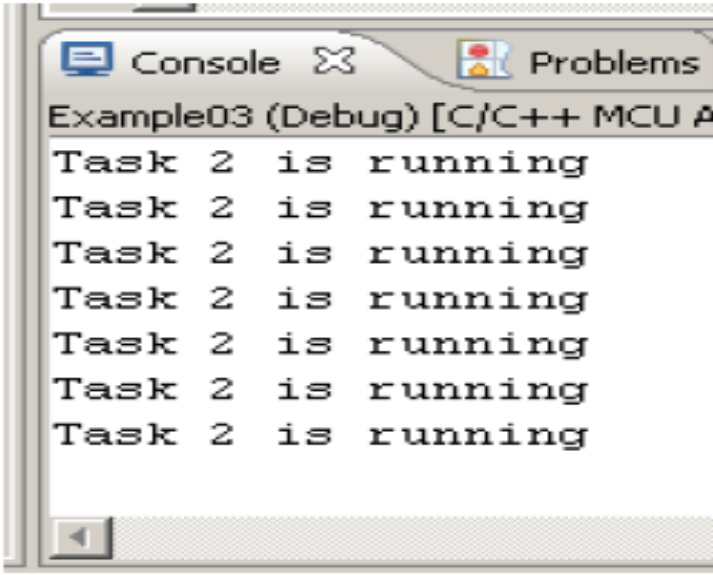
- The priority can be changed after the scheduler has been started by using the vTaskPrioritySet() API function.
- The maximum number of priorities available is set by the application-defined configMAX_PRIORITIES compile time configuration constant within FreeRTOSConfig.h.
- The higher the configMAX_PRIORITIES value the more RAM the kernel will consume, “keep it minimum”.
- Low numeric priority values denote low-priority tasks, with priority 0 being the lowest priority possible.
- Where more than one task of the same priority is able to run, the scheduler will transition each task into and out of the Running state, in turn.
- Each such task executes for a ‘time slice‘; it enters the Running state at the start of the time slice and exits the Running state at the end of the time slice.

Task Priorities; Example 3

```
/* 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\n";
static const char *pcTextForTask2 = "Task 2 is running\n";

int main( void )
{
    /* Create the first task at priority 1.  The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 240, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2. */
    xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 2, NULL );

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

    /* If all is well we will never reach here as the scheduler will now be
    running.  If we do reach here then it is likely that there was insufficient
    heap available for the idle task to be created. */
    for( ;; );
}
```

Task Priorities: Example 3; Starvation

- Both Tasks are made periodic by the “dummy” loop
- Both Tasks only needs CPU for short execution time!
- Task 2 (High Priority) takes CPU all the time
- Task 1 suffers starvation
- Wastes power and cycles!
- Is there another smarter way?



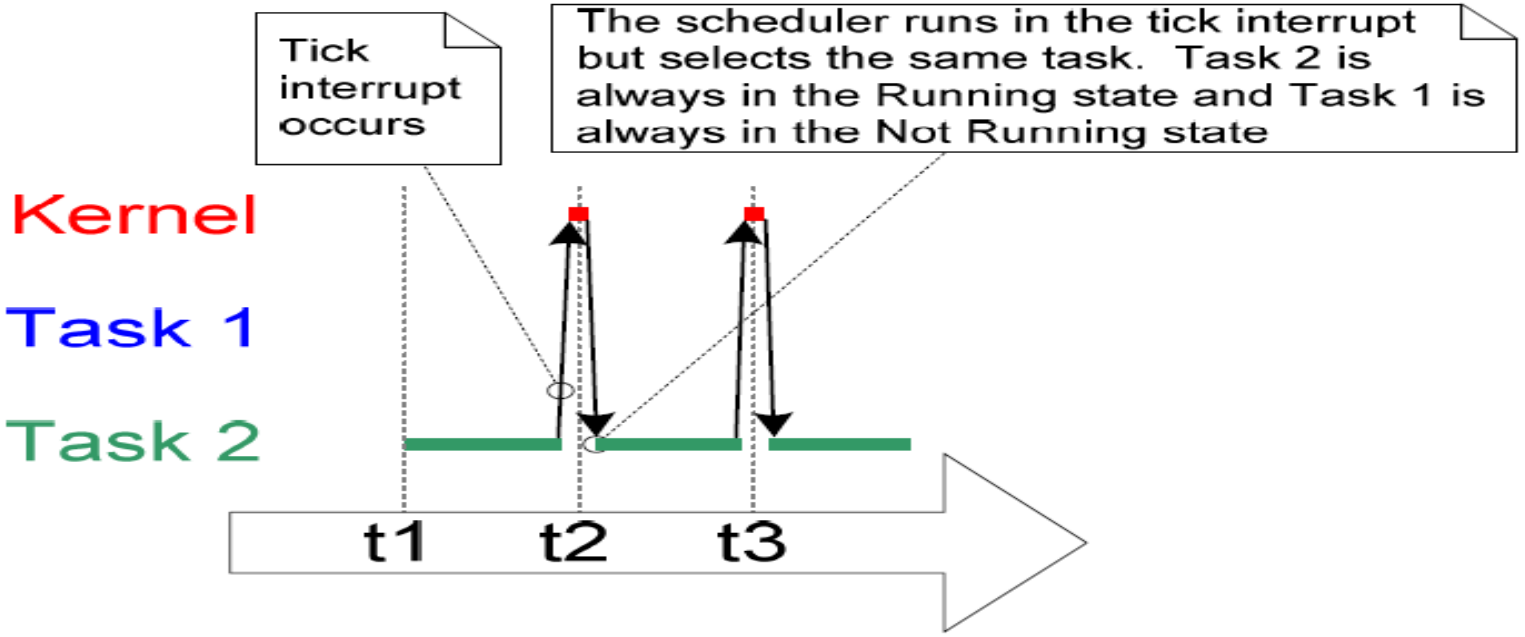


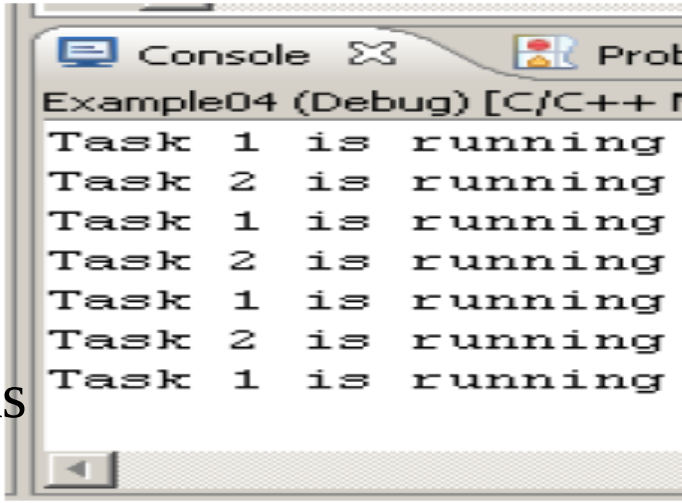
Figure 6. The execution pattern when one task has a higher priority than the other

Using the Blocked state to create a delay

- vTaskDelay() places the calling task into the Blocked state for a fixed number of tick interrupts.
- While in the Blocked state the task does not use any processing time
- vTaskDelay() API function is available only when. INCLUDE_vTaskDelay is set to 1 in FreeRTOSConfig.h
- The constant portTICK_RATE_MS can be used to convert milliseconds into ticks.
- Portmacro.h: #define portTICK_RATE_MS ((portTickType) 1000 / configTICK_RATE_HZ)
- FreeRTOSConfig.h: #define configTICK_RATE_HZ ((portTickType) 1000)

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;

    /* 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 delay period is specified in 'ticks', but the constant
        portTICK_RATE_MS can be used to convert this to a more user friendly value
        in milliseconds.  In this case a period of 250 milliseconds is being
        specified. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
```

Task Blocking: Example 4



- Each time the tasks leave the Blocked state they execute for a fraction of a tick period before re-entering the Blocked state.
- Most of the time there are no application tasks that are able to run; The idle task will run.
- Idle task time is a measure of the spare processing capacity in the system.

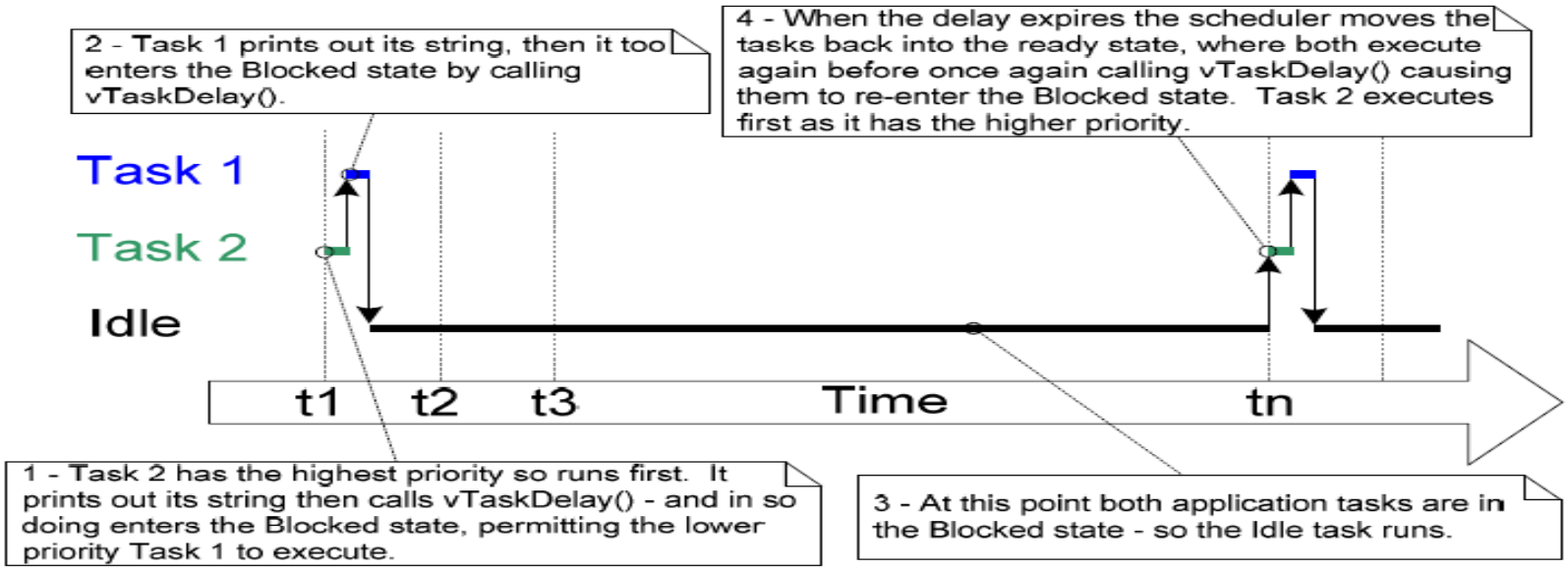
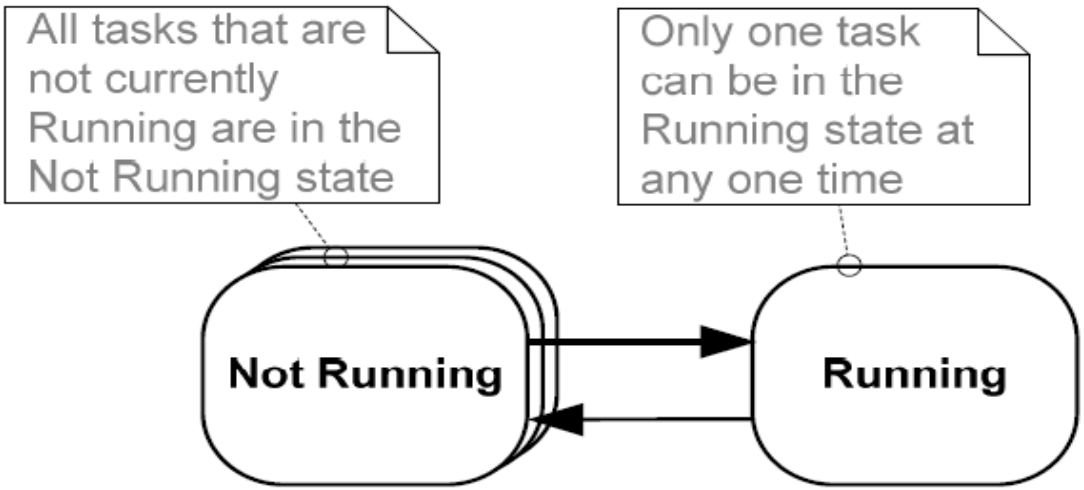


Figure 9. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

Expanding the 'Not Running' State



The Blocked State

- A task that is waiting for an event is said to be in the 'Blocked' state, which is a sub-state of the Not Running state.
- Tasks can enter the Blocked state to wait for two different types of event:
  - Temporal (time-related) events—the event being either a delay period expiring, or an absolute time being reached. For example, a task may enter the Blocked state to wait for 10 milliseconds to pass.
  - Synchronization events—where the events originate from another task or interrupt. For example, a task may enter the Blocked state to wait for data to arrive on a queue. Synchronization events cover a broad range of event types.

Expanding the 'Not Running' State

The Suspended State

- 'Suspended' is also a sub-state of Not Running.
- Tasks in the Suspended state are not available to the scheduler.
- The only way into the Suspended state is through a call to the vTaskSuspend() API function
- the only way out being through a call to the vTaskResume() or xTaskResumeFromISR() API functions.
- Most applications do not use the Suspended state.

The Ready State

- Tasks that are in the Not Running state but are not Blocked or Suspended are said to be in the Ready state.
- They are able to run, and therefore 'ready' to run, but their priorities are not qualifying to be in the Running state.

Task Blocking: Example 4

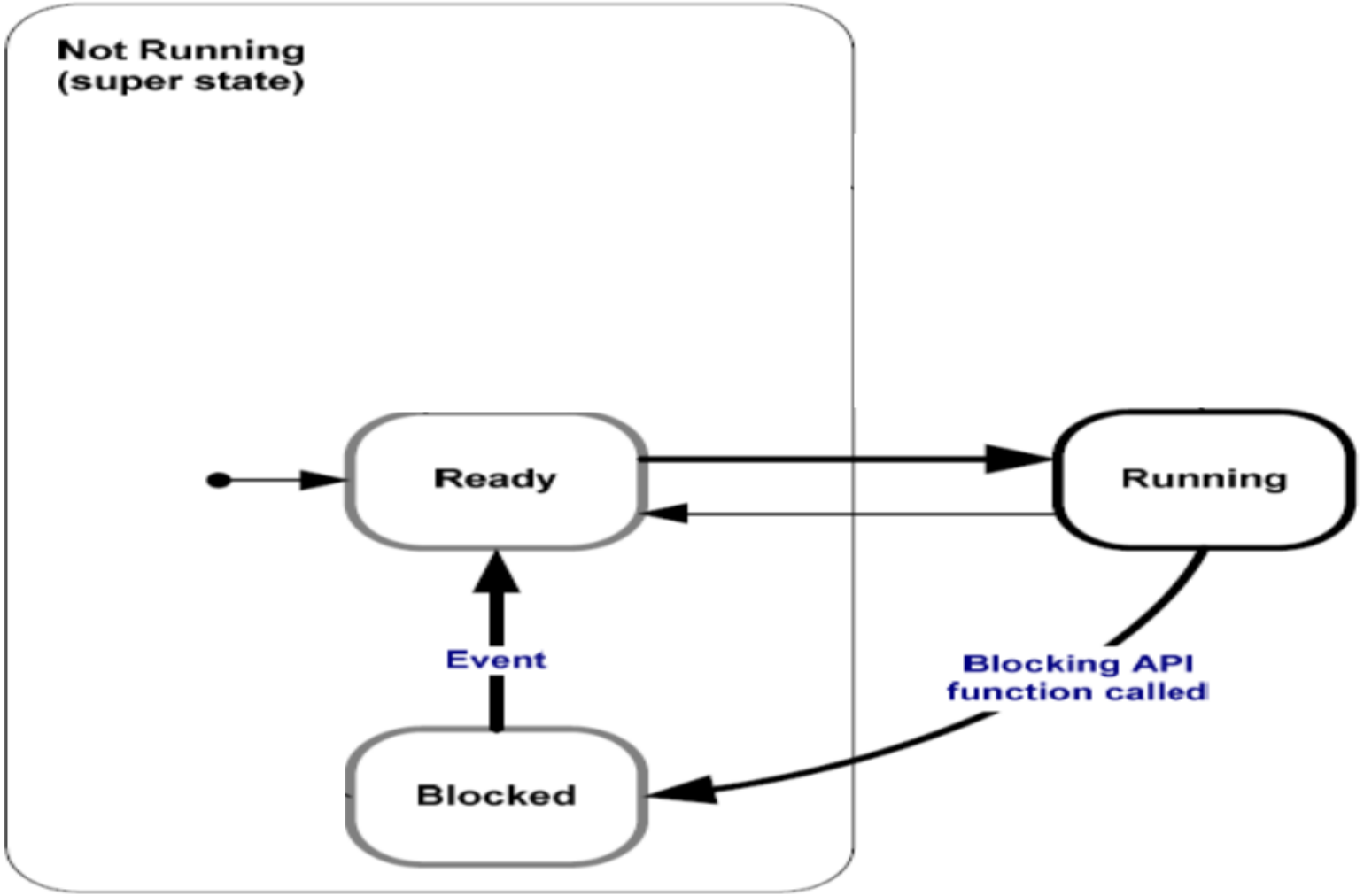


Figure 10. Bold lines indicate the state transitions performed by the tasks in Example 4

FreeRTOS Task SM

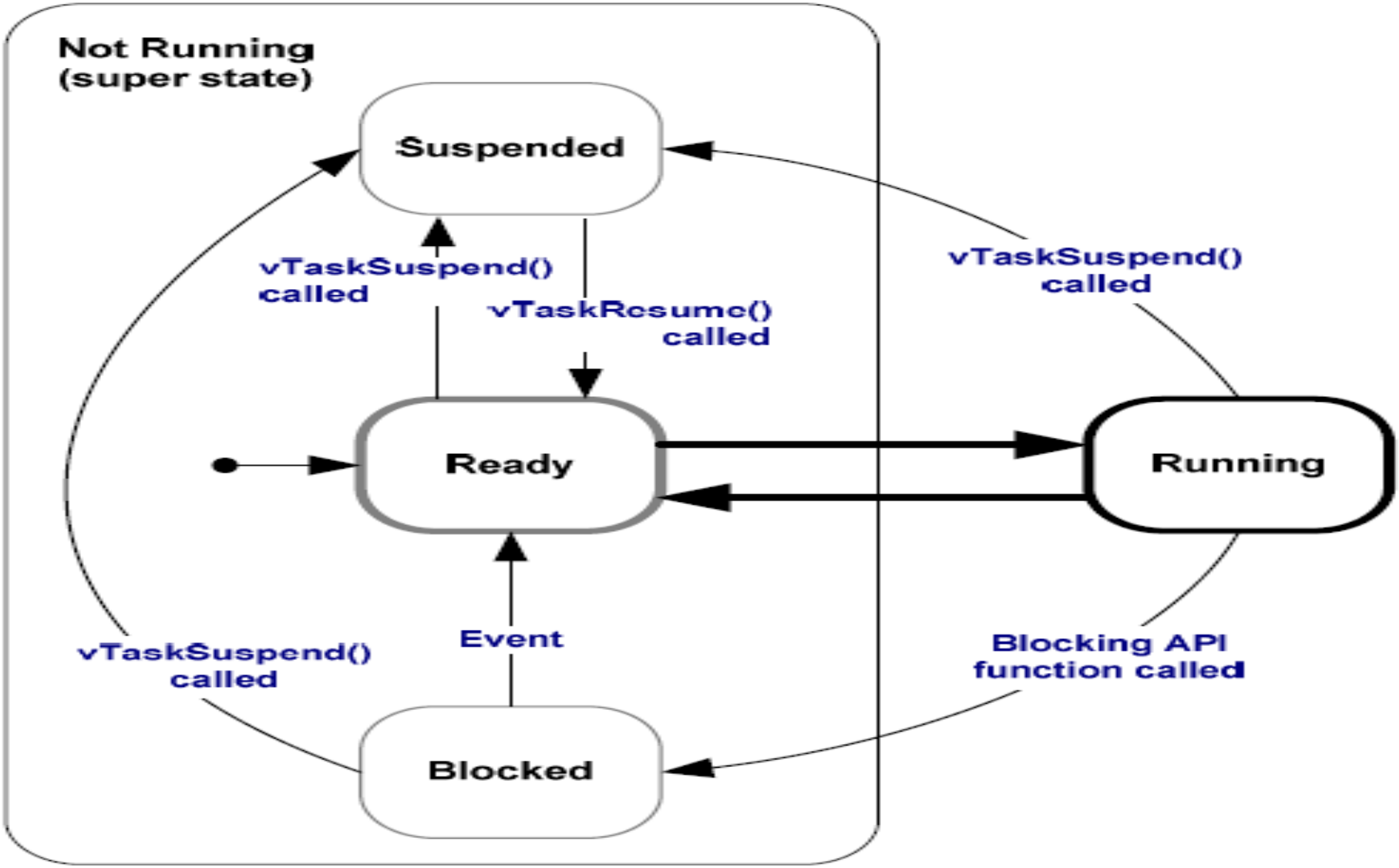


Figure 7. Full task state machine