

Real Time Operating System Introduction & “FreeRTOS”

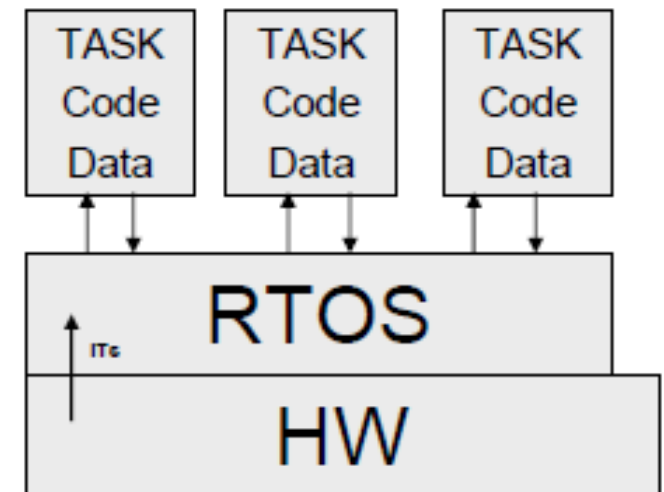
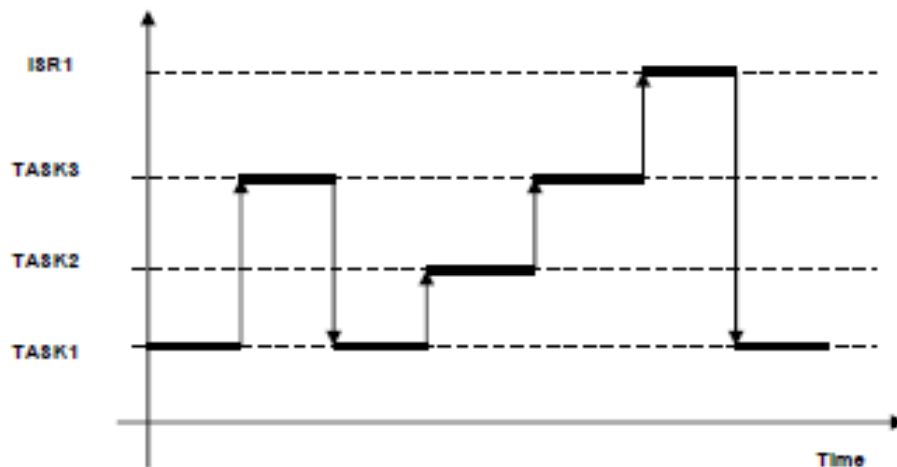


Agenda

- **RTOS Basics**
- **RTOS sample State Machine**
- **RTOS scheduling criteria**
- **RTOS optimization criteria**
- **Soft/Hard Real Time requirements**
- **Tasks scheduling**
- **Tasks priorities**
- **Task State Machine**
- **Periodic tasks**
- **Tasks Blocking**

RTOS Basics

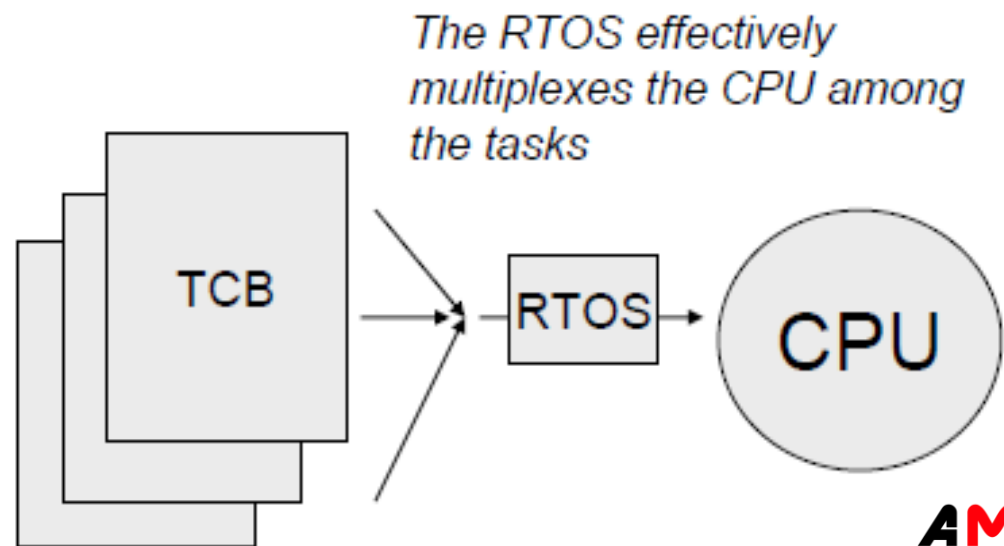
- Kernel: schedules tasks
- Tasks: concurrent activity with its own state (PC, registers, stack, etc.)



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



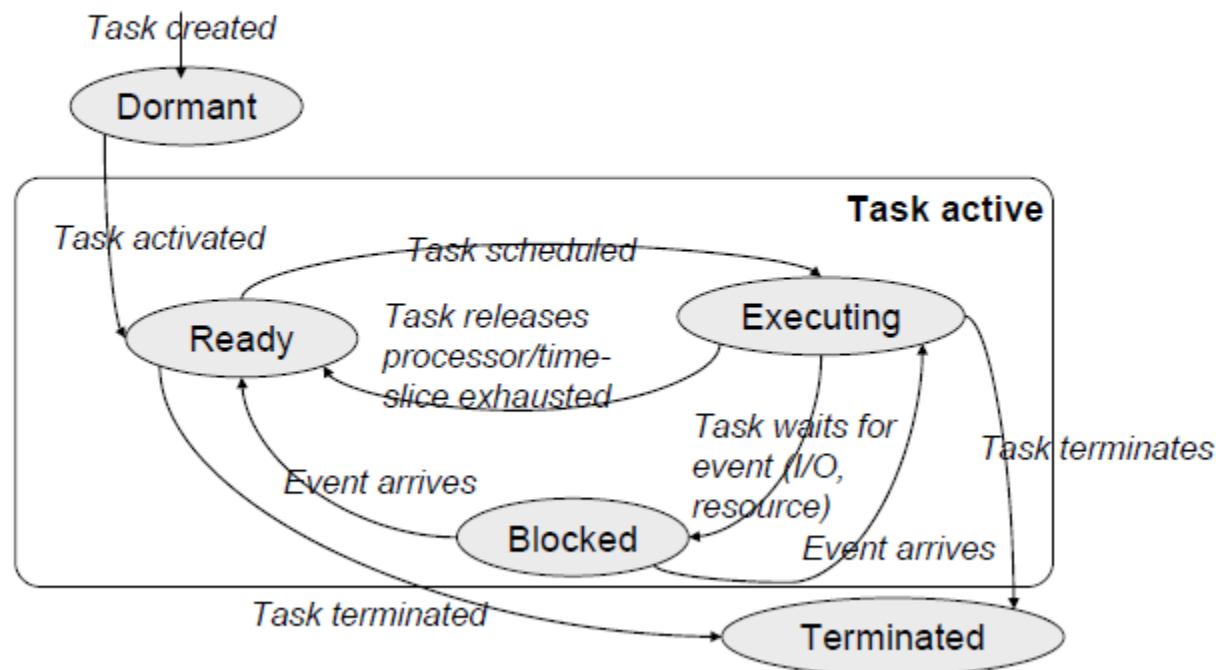
Task States

- **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- **Terminated:** no longer active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

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 kept as busy as possible. (40 to 90 percent)
- Throughput: Work is being done. No. Of processes per unit time
- Turnaround time: For a particular process how long it takes to execute.
(Interval between time of submission to time of 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;

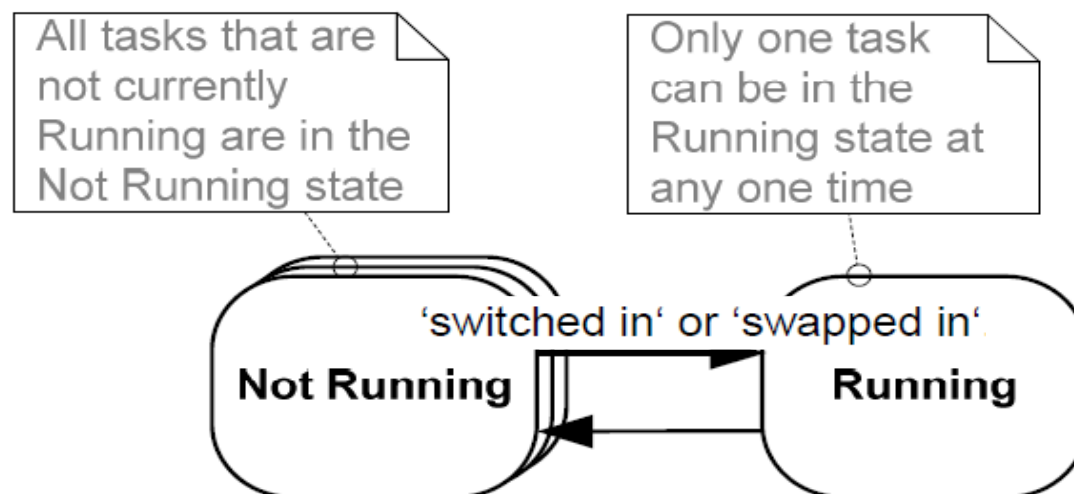
    /* A task will normally be implemented as an infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* 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, the task is dormant, 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 xTask Create() 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



```
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

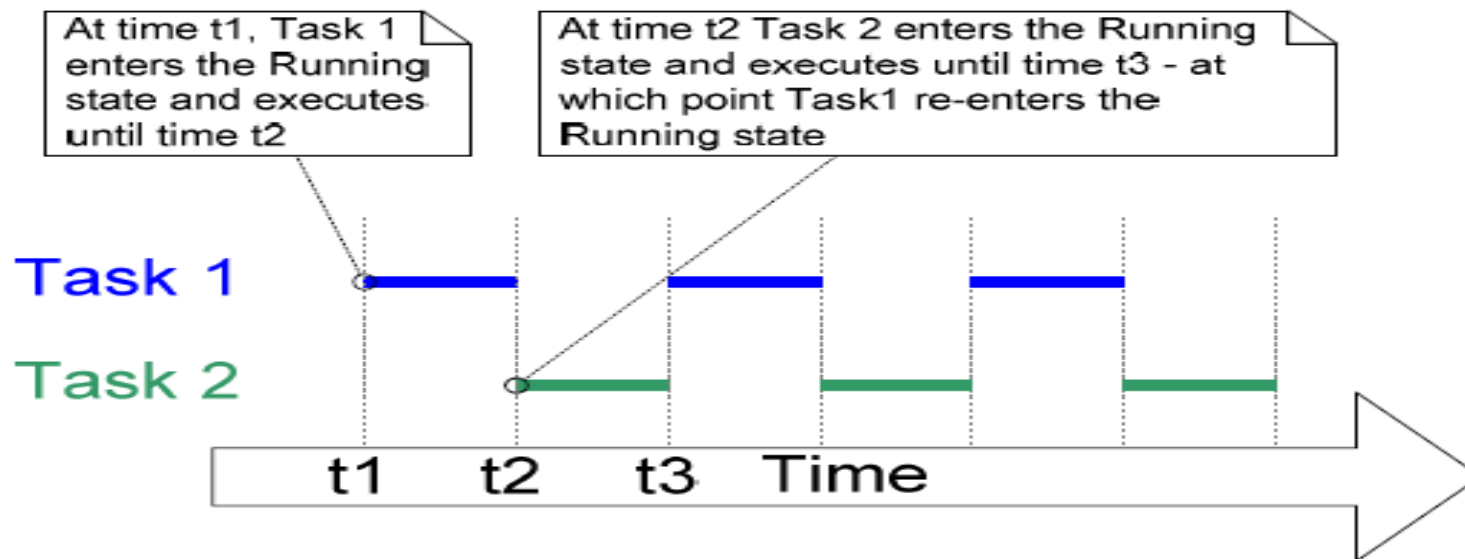


Figure 3. The execution pattern of the two Example 1 tasks

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

- 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`.

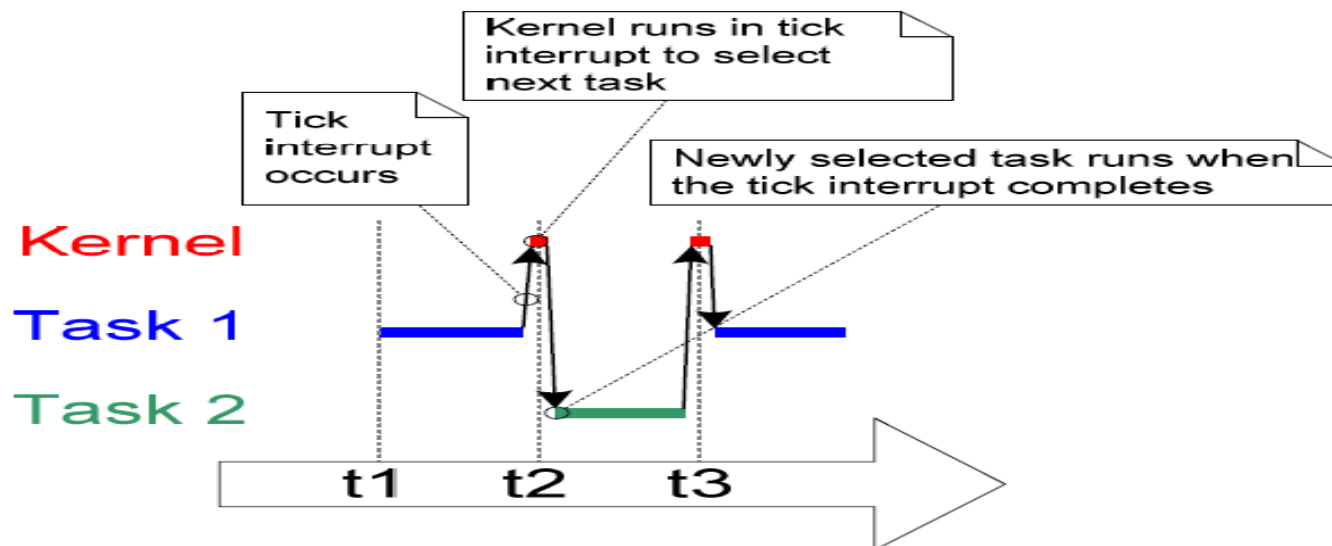


Figure 4. The execution sequence expanded to show the tick interrupt executing

Task Priorities; Example 3: “Starvation”

```
/* 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”



```
Example03 (Debug) [C/C++ MCU Application] C:\E\Dev\FreeRTOS\DOC\Projects\191-ApplicationM
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
```

Figure 5. Running both test tasks at different priorities

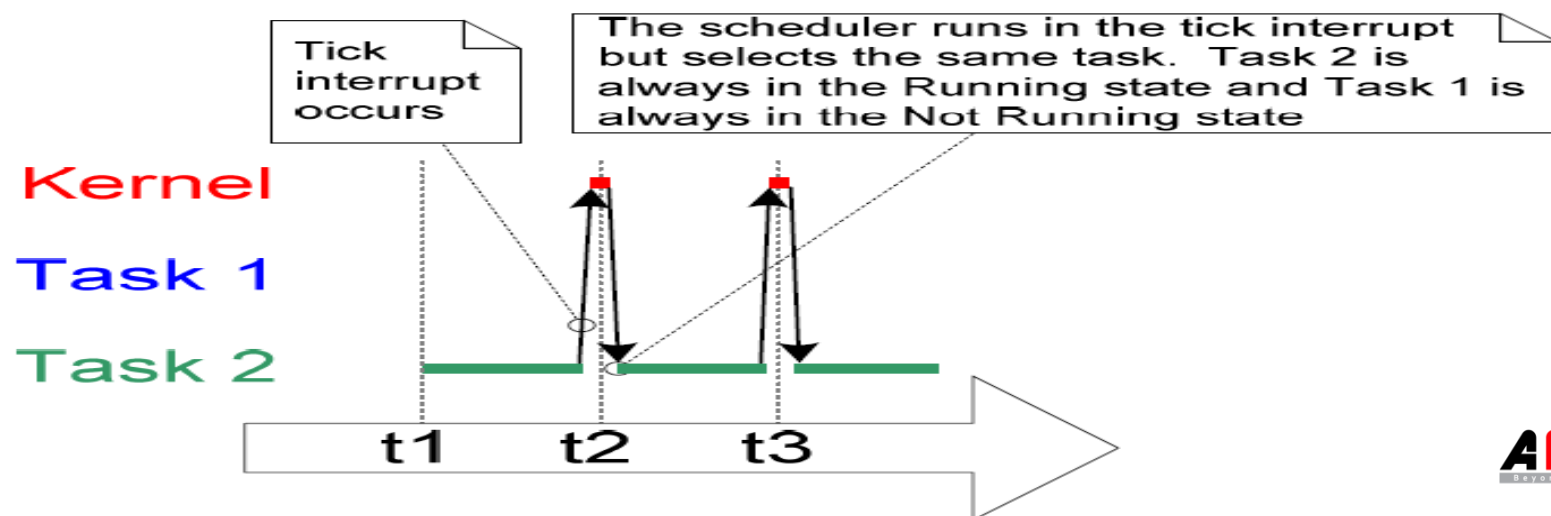


Figure 6. The execution pattern when one task has a higher priority than the other

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";

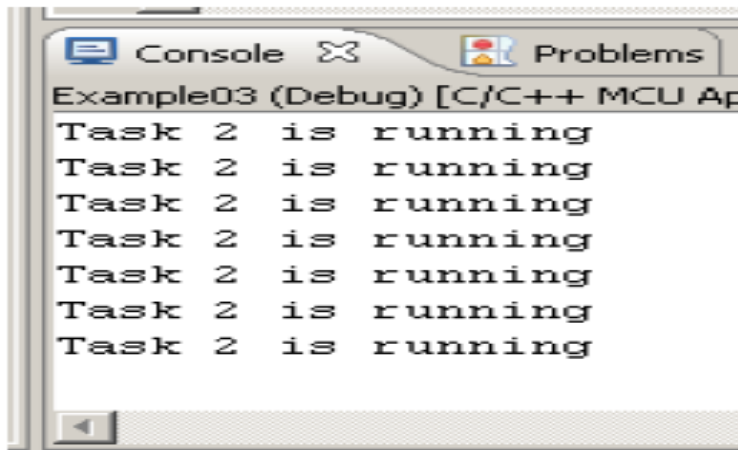
int main( void )
{
    /* Create the first task at priority 1. The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 240, (void*)pcTextForTask1, 1, NULL );

    /* 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



```
Console Problems
Example03 (Debug) [C/C++ MCU Ap
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
```

- 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?

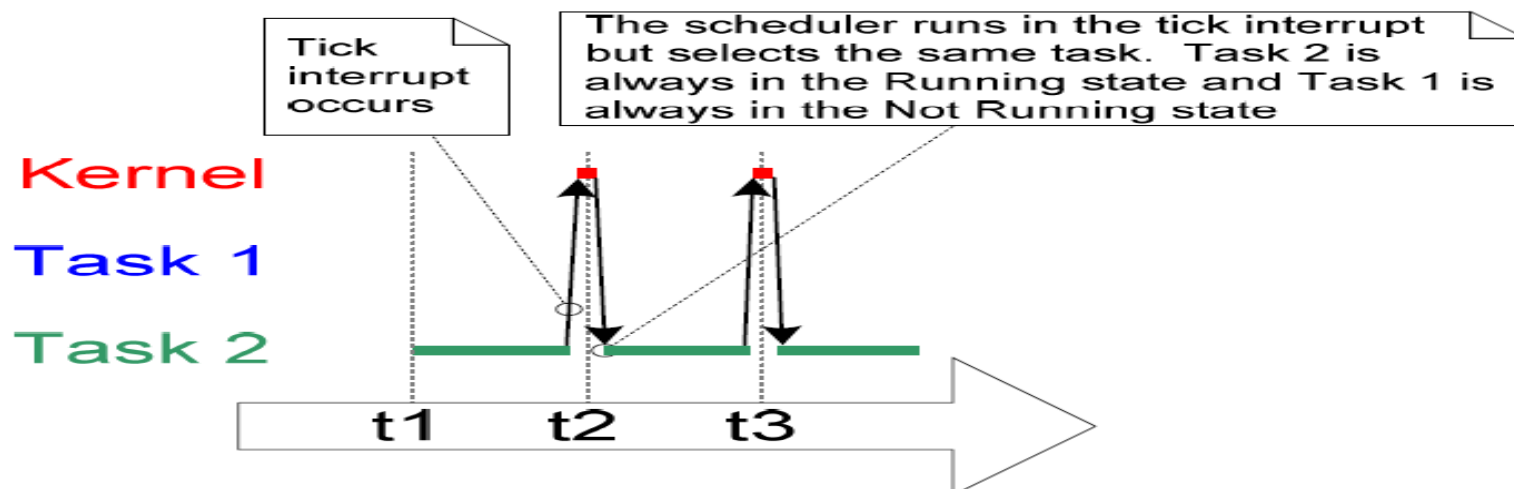
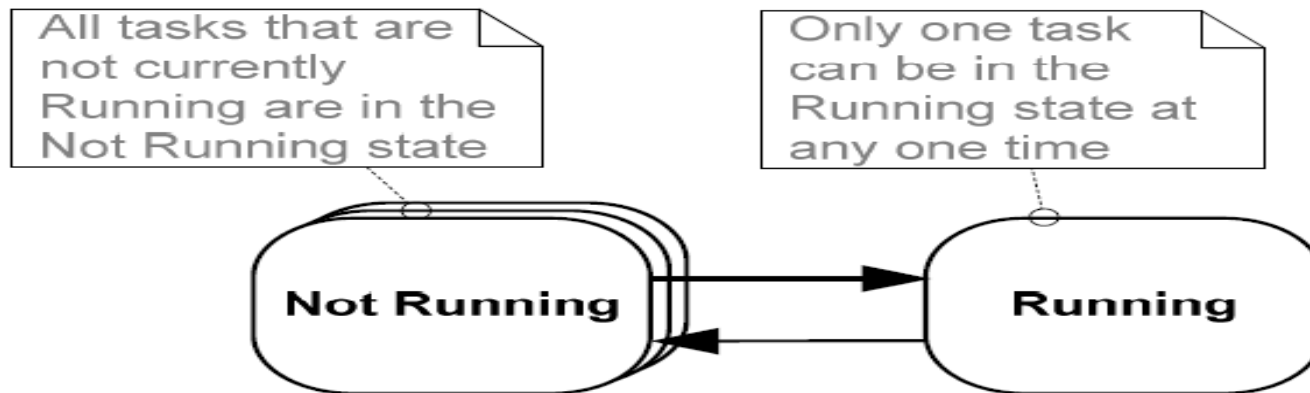


Figure 6. The execution pattern when one task has a higher priority than the other

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.

FreeRTOS Task SM

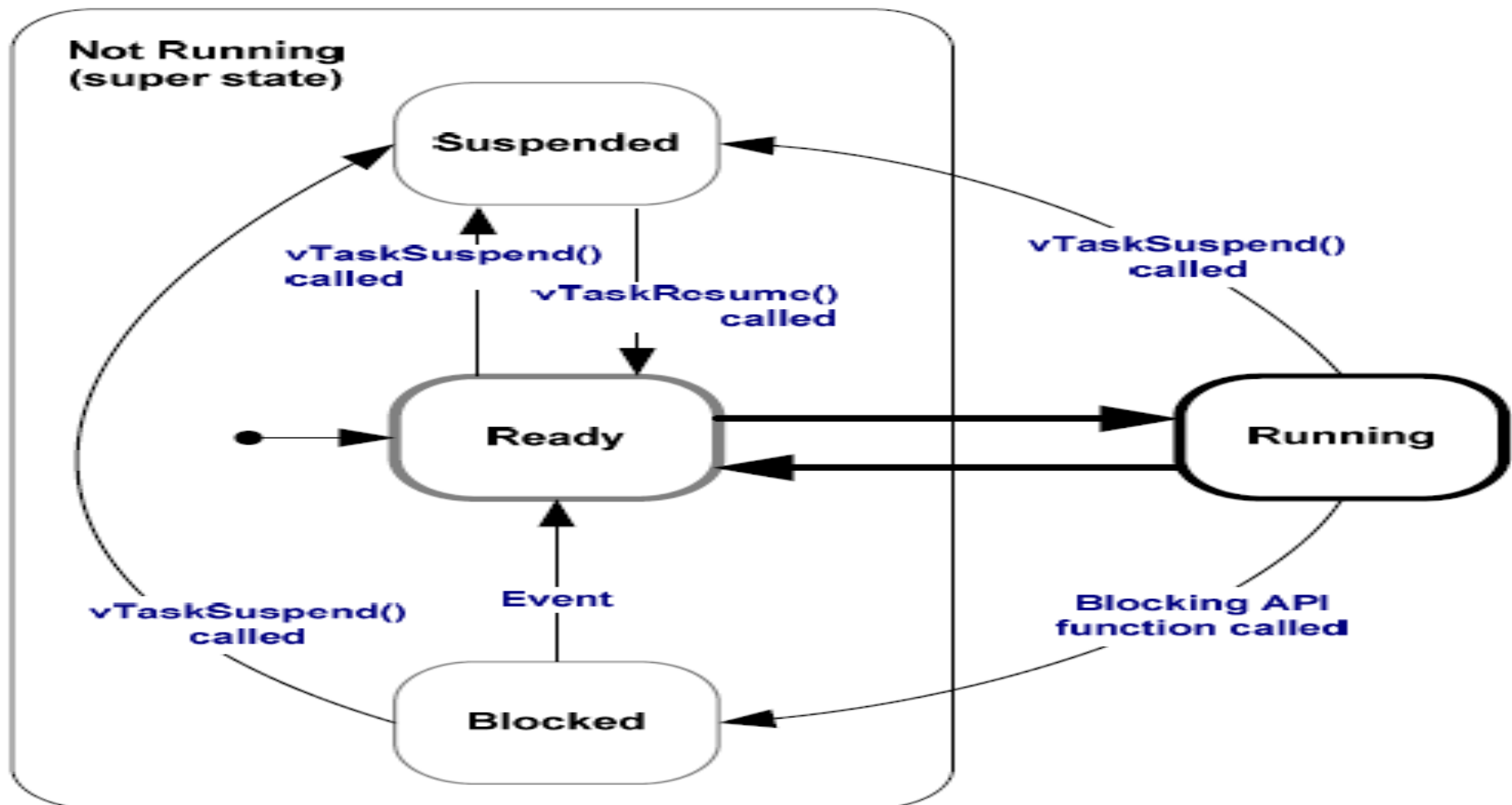


Figure 7. Full task state machine

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.

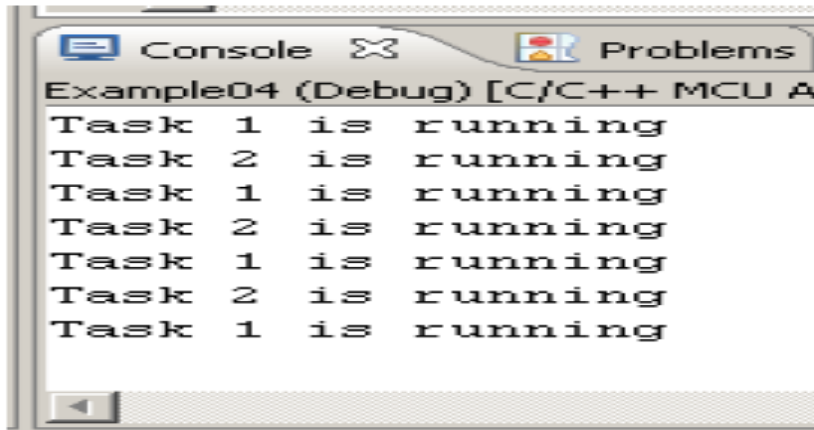
```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;

    /* 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



```
Example04 (Debug) [C/C++ MCU A]
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
```

- 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.

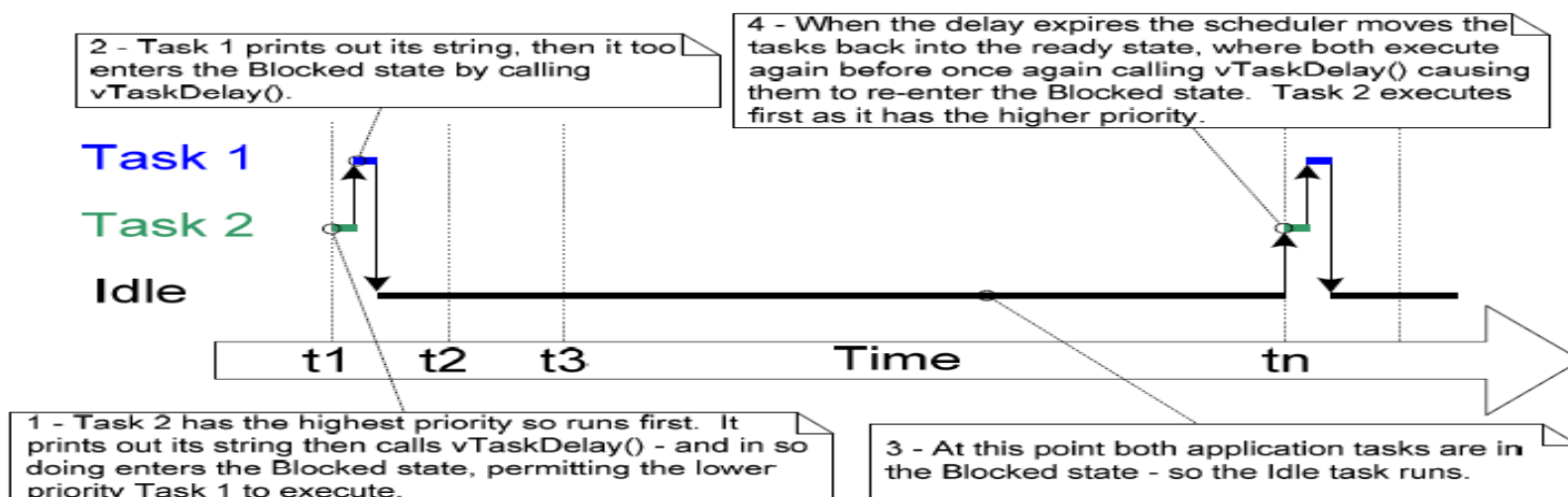


Figure 9. The execution sequence when the tasks use `vTaskDelay()` in place of the NULL loop

Task Blocking: Example 4

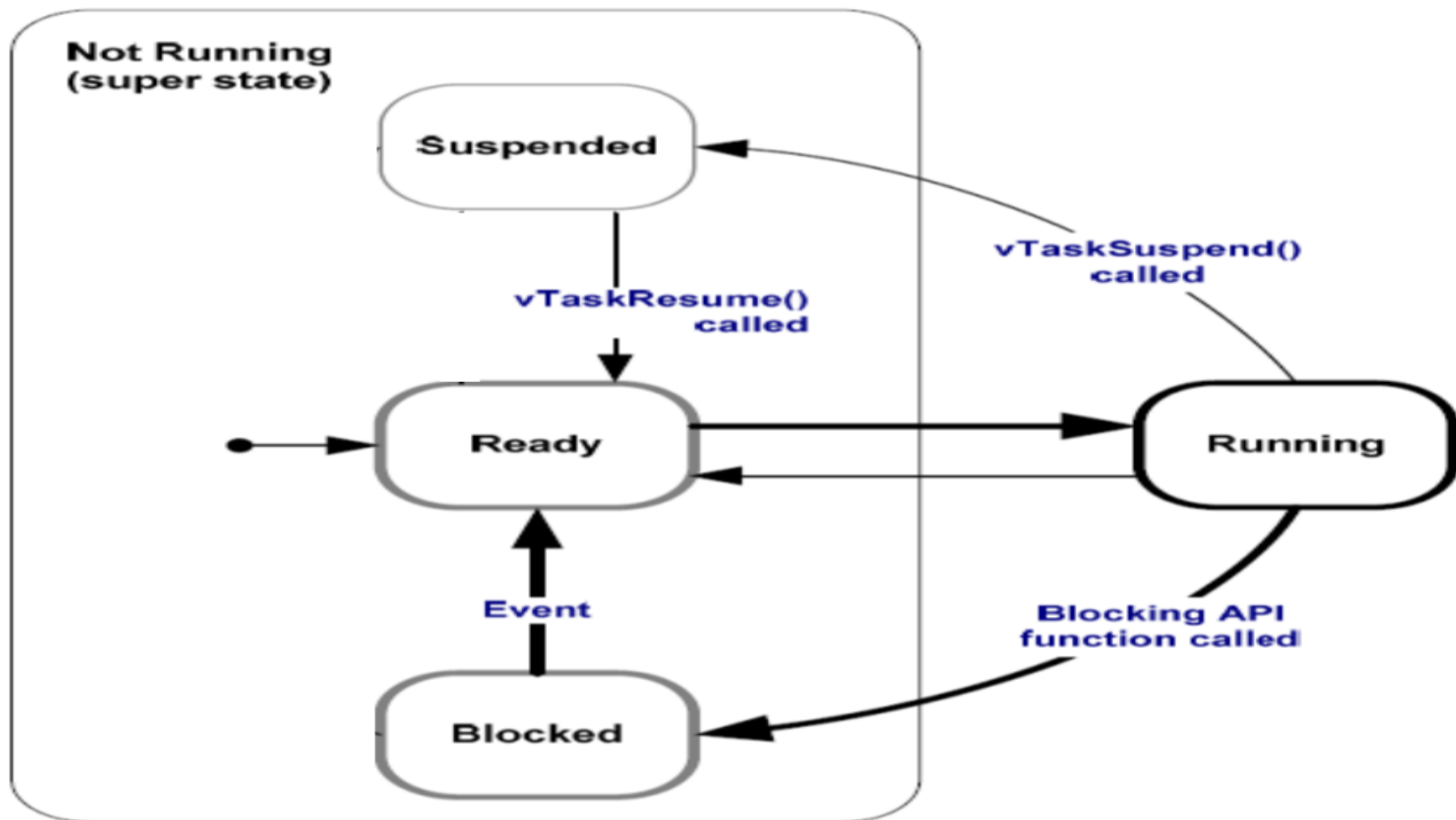


Figure 10. Bold lines indicate the state transitions performed by the tasks in Example 4