Real Time Operating System Introduction & "FreeRTOS"

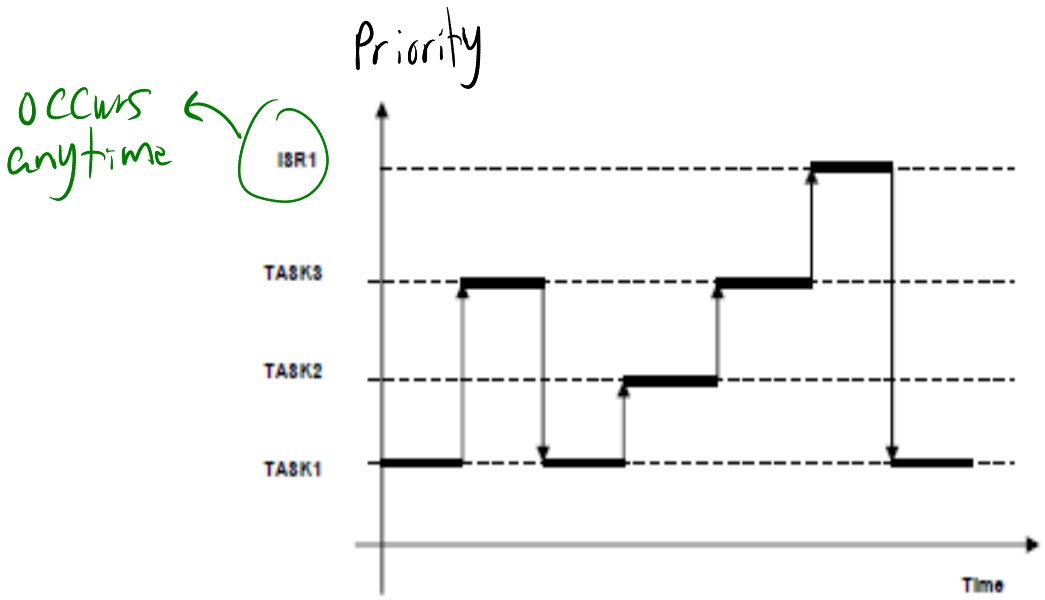
Sherif Hammad

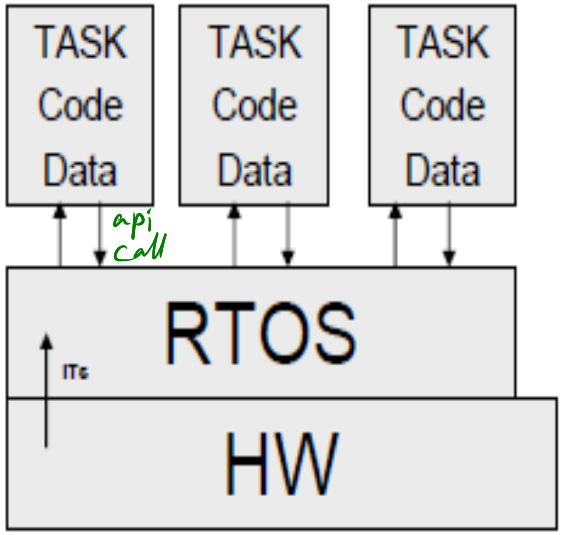
Agenda

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

RTOS Basics

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



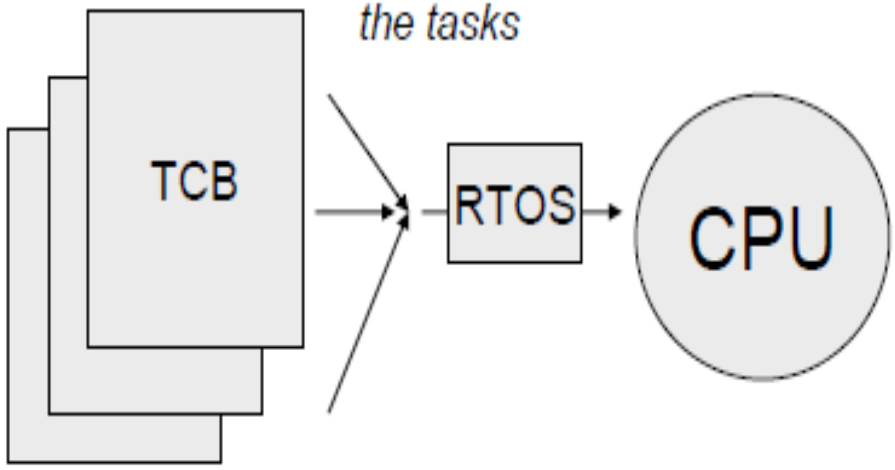


Tasks

- 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

The RTOS effectively multiplexes the CPU among the tasks

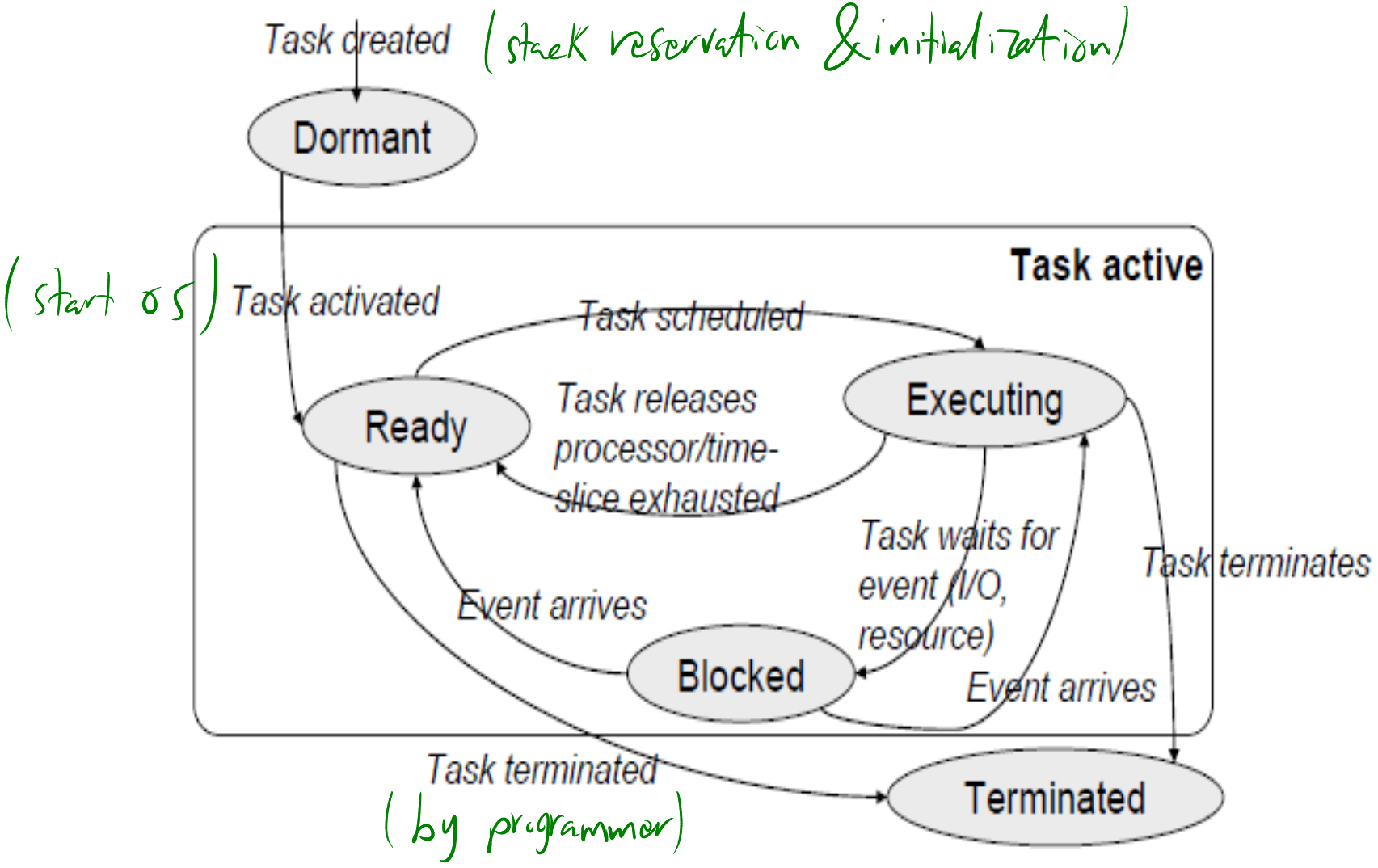


Task states

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- **Terminated**: no longer active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

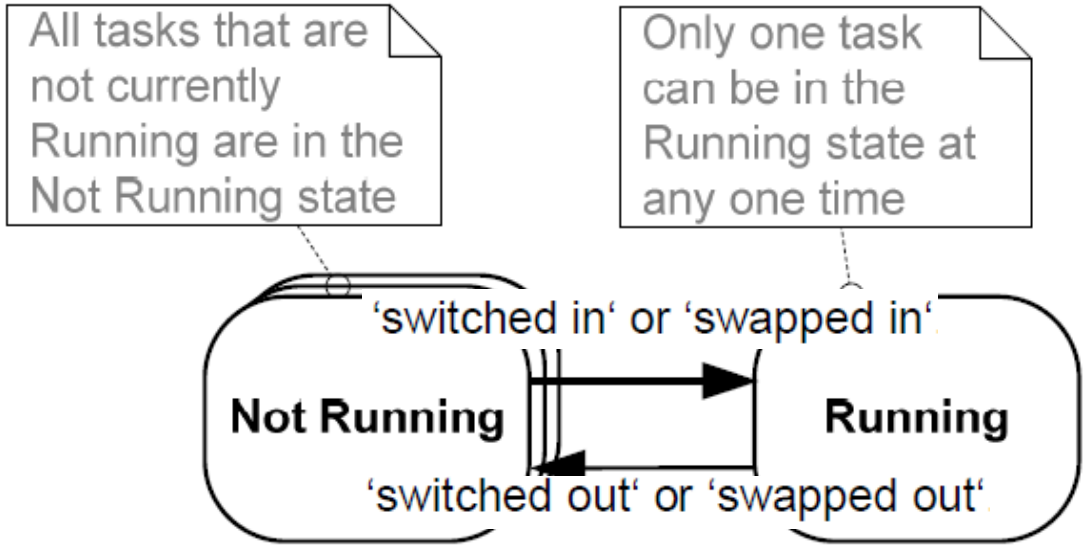
Task Functions

```
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.



xTaskCreate

[Task Creation]

task. h

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

Create a new task and add it to the list of tasks that are ready to run. configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or left undefined (in which case it will default to 1), for this RTOS API function to be available.

Each task requires RAM that is used to hold the task state, and used by the task as its stack. If a task is created using `xTaskCreate()` then the required RAM is automatically allocated from the FreeRTOS heap. If a task is created using xTaskCreateStatic() then the RAM is provided by the application writer, so it can be statically allocated at compile time.

pvTaskCode Pointer to the task entry function (just the name of the function that implements the task, see the example below).

Tasks are normally implemented as an infinite loop; the function which implements the task must never attempt to return or exit. Tasks can, however, delete themselves.

pcName A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used to obtain a task handle.

The maximum length of a task's name is defined by `configMAX_TASK_NAME_LEN` in FreeRTOSConfig.h.

usStackDepth The number of words (not bytes!) to allocate for use as the task's stack. For example, if the stack is 16-bits wide and usStackDepth is 100, then 200 bytes will be allocated for use as the task's stack. As another example, if the stack is 32-bits wide and usStackDepth is 400 then 1600 bytes will be allocated for use as the task's stack.

The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type `size_t`.

pvParameters A value that is passed as the paramater to the created task.

If `pvParameters` is set to the address of a variable then the variable must still exist when the created task executes - so it is not valid to pass the address of a stack variable.

uxPriority The priority at which the created task will execute.

Systems that include MPU support can optionally create a task in a privileged (system) mode by setting the bit `portPRIVILEGE_BIT` in `uxPriority`. For example, to create a privileged task at priority 2 set `uxPriority` to (2 | `portPRIVILEGE_BIT`).

Priorities are asserted to be less than `configMAX_PRIORITIES`. If `configASSERT` is undefined, priorities are silently capped at (`configMAX_PRIORITIES` - 1).

pxCreatedTask Used to pass a handle to the created task out of the `xTaskCreate()` function. `pxCreatedTask` is optional and can be set to NULL.

Creating Tasks
The xTaskCreate() API Function

Same priority tasks →

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

Same priority → round robin

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

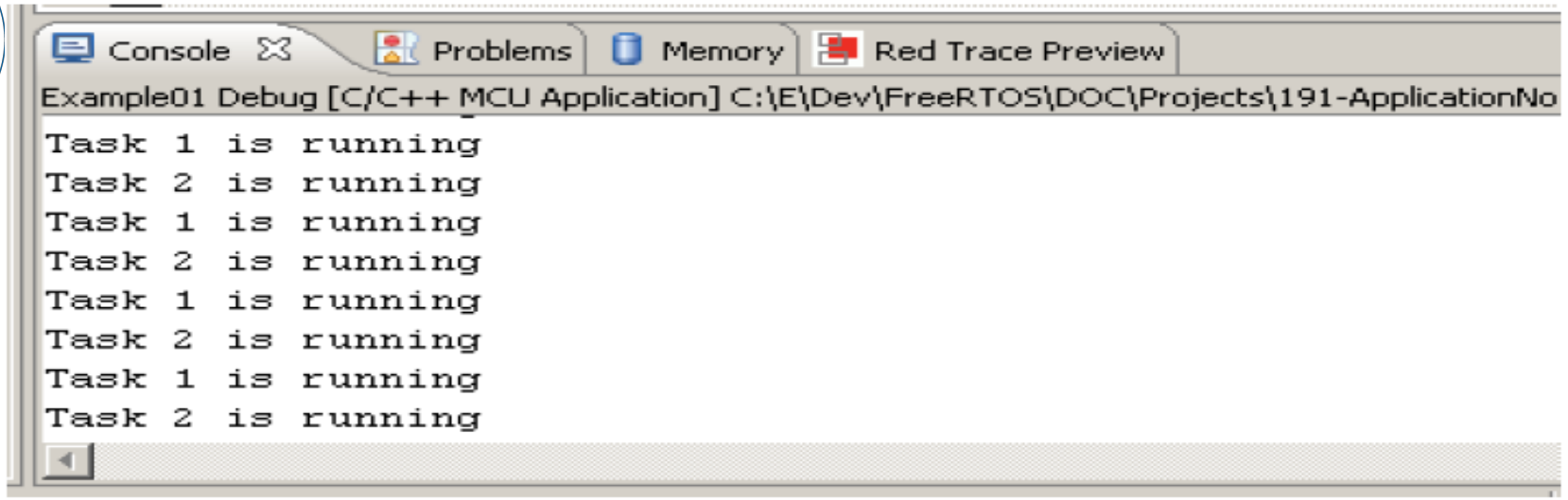


Figure 2. The output produced when Example 1 is executed

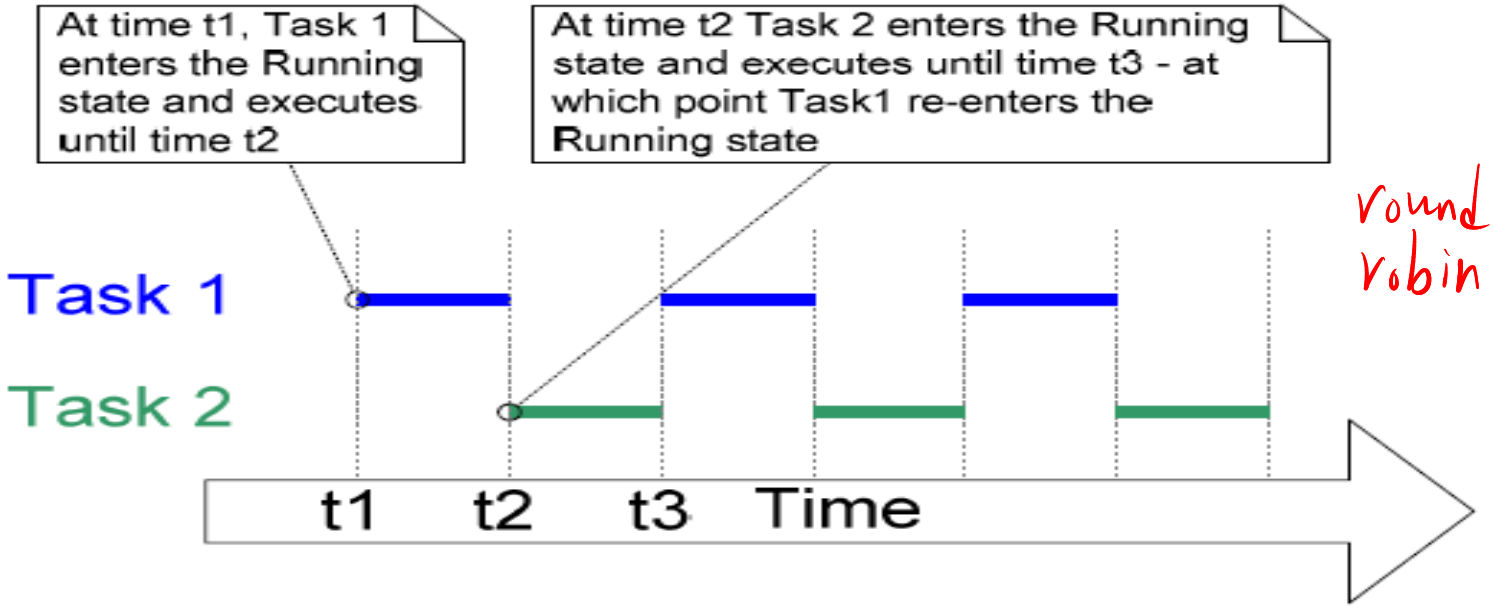


Figure 3. The execution pattern of the two Example 1 tasks

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.

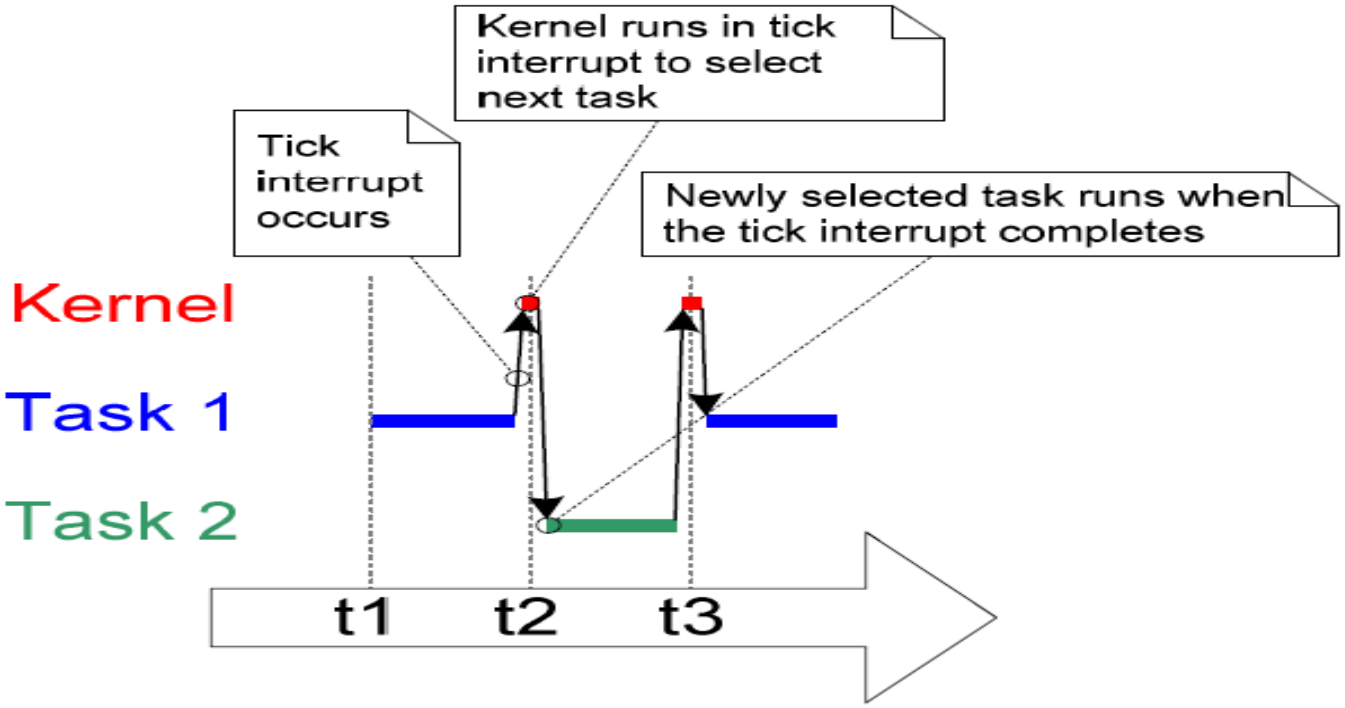
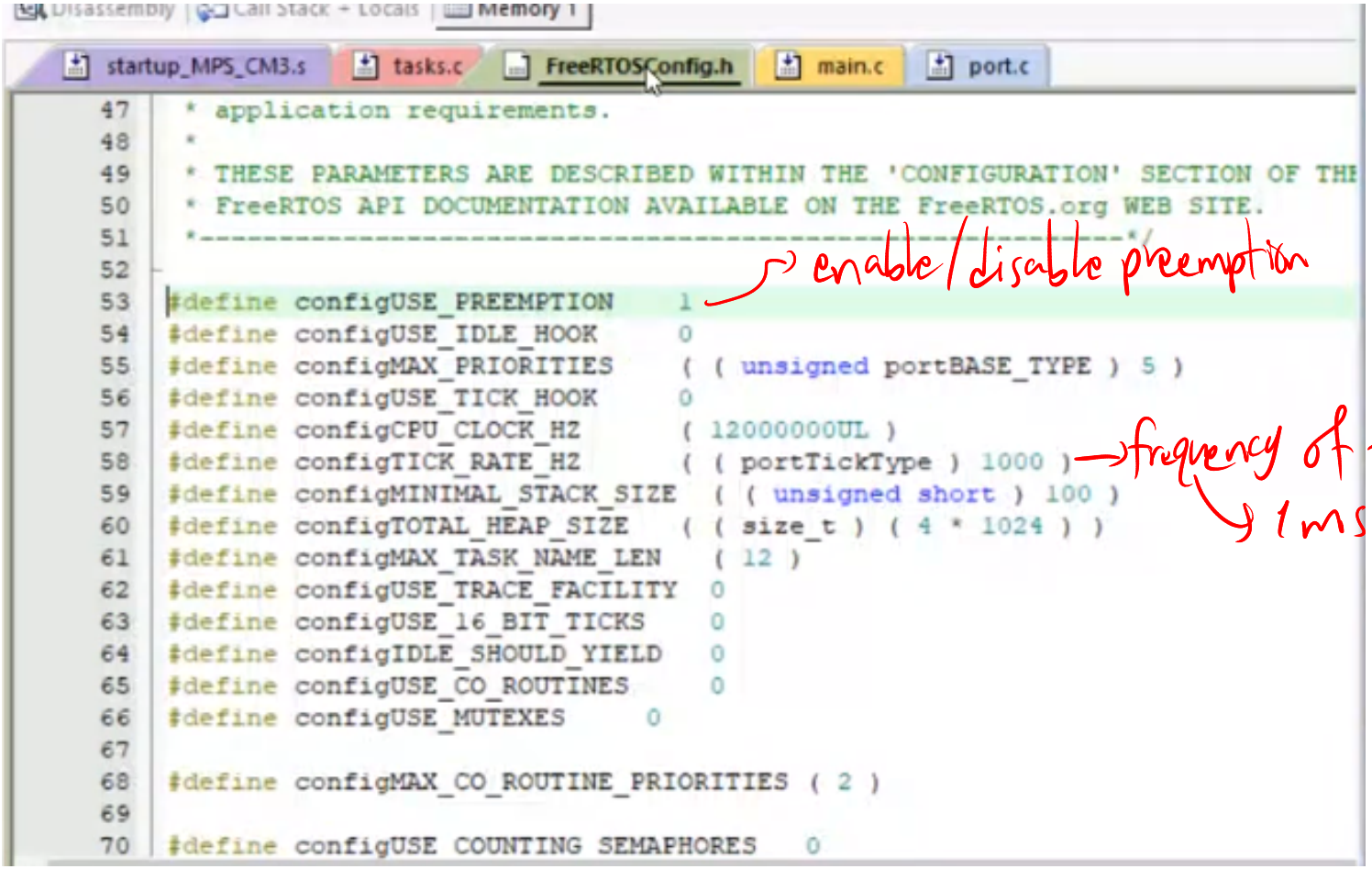


Figure 4. The execution sequence expanded to show the tick interrupt executing

Agenda

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

OS Config


Disassembly
Call Stack + Locals
Memory 1
startup_MPS_CM3.s
tasks.c
FreeRTOSConfig.h
main.c
port.c
47 * application requirements.
48 *
49 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
50 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
51 *----------*/
52
53 #define configUSE_PREEMPTION 1
54 #define configUSE_IDLE_HOOK 0
55 #define configMAX_PRIORITIES ((unsigned portBASE_TYPE) 5)
56 #define configUSE_TICK_HOOK 0
57 #define configCPU_CLOCK_HZ (12000000UL)
58 #define configTICK_RATE_HZ ((portTickType) 1000)
59 #define configMINIMAL_STACK_SIZE ((unsigned short) 100)
60 #define configTOTAL_HEAP_SIZE ((size_t) (4 * 1024))
61 #define configMAX_TASK_NAME_LEN (12)
62 #define configUSE_TRACE_FACILITY 0
63 #define configUSE_16_BIT_TICKS 0
64 #define configIDLE_SHOULD_YIELD 0
65 #define configUSE_CO_ROUTINES 0
66 #define configUSE_MUTEXES 0
67
68 #define configMAX_CO_ROUTINE_PRIORITIES (2)
69
70 #define configUSE_COUNTING_SEMAPHORES 0
enable/disable preemption
frequency of ticks
1ms time slice

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)

(different TCBs & Stacks)

```
/* 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.

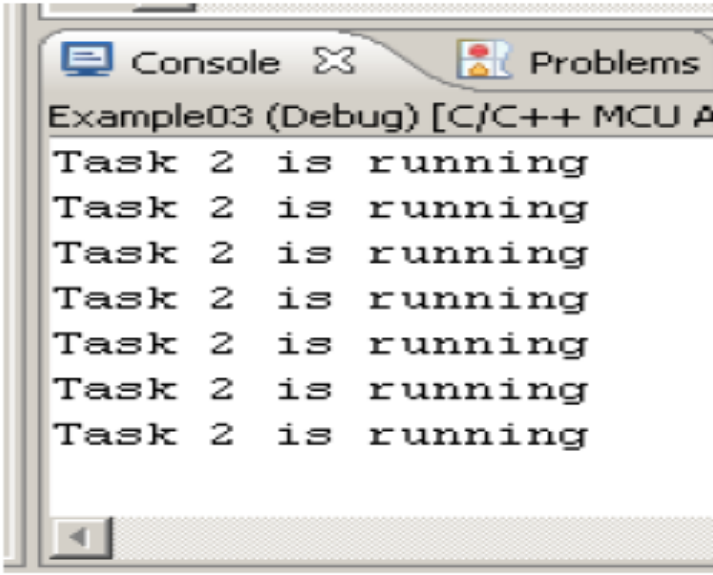
Task Priorities; Example 3

different 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\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?



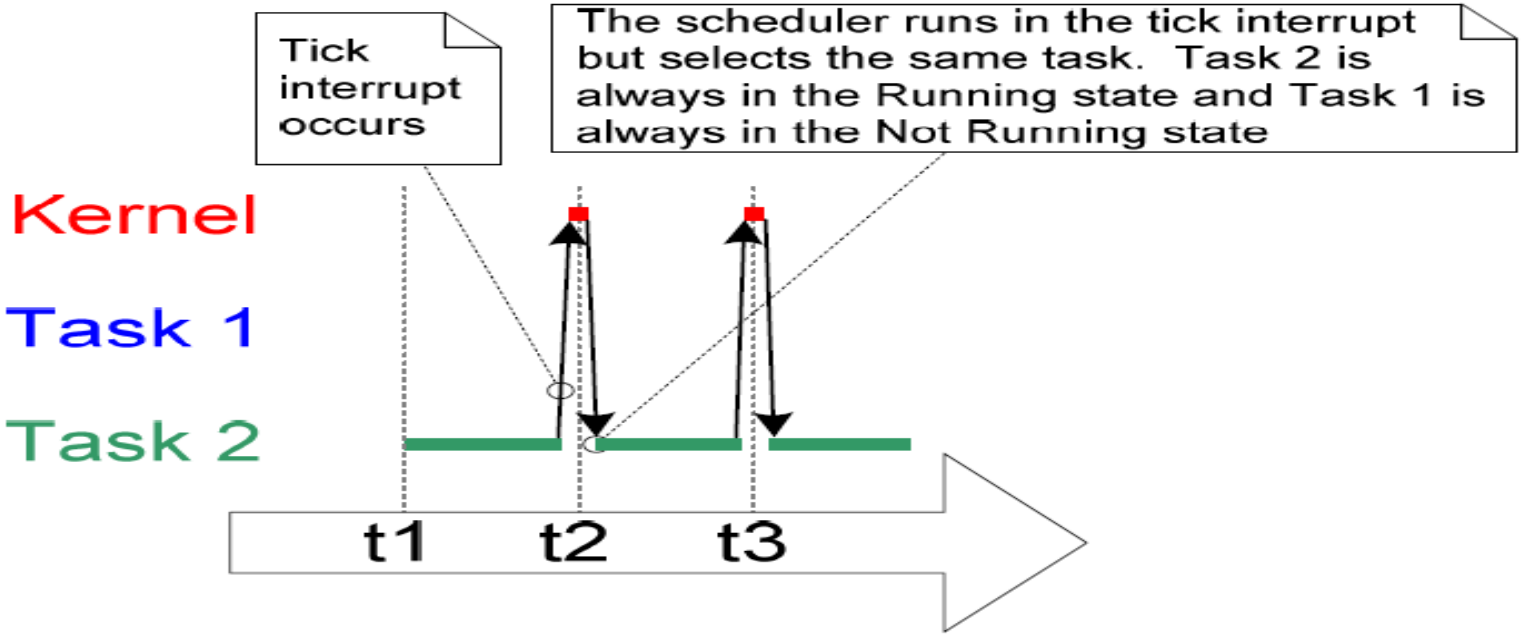


Figure 6. The execution pattern when one task has a higher priority than the other

Using the Blocked state to create a delay

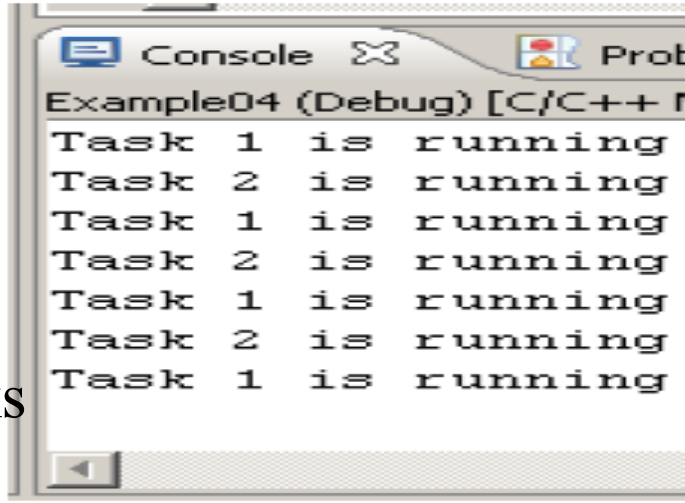
- vTaskDelay() places the calling task into the Blocked state for a fixed number of tick interrupts.
- While in the Blocked state the task does not use any processing time
- vTaskDelay() API function is available only when. INCLUDE_vTaskDelay is set to 1 in FreeRTOSConfig.h
- The constant portTICK_RATE_MS 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)

periodic task (not continuous)

```
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 );
    }
```

Goes to systick

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.

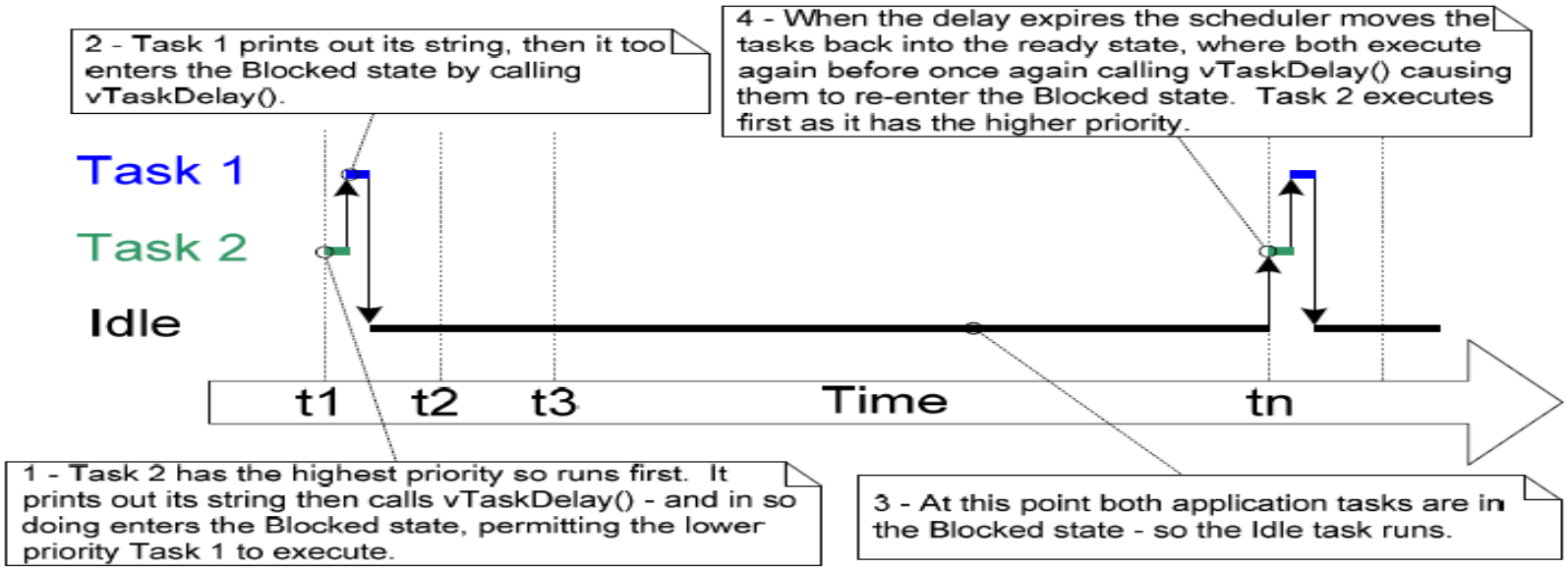
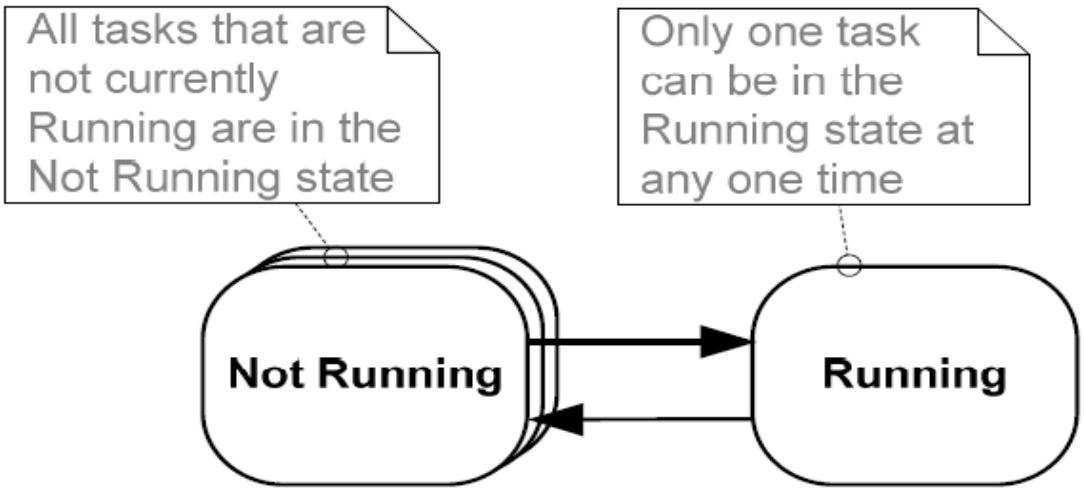


Figure 9. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

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

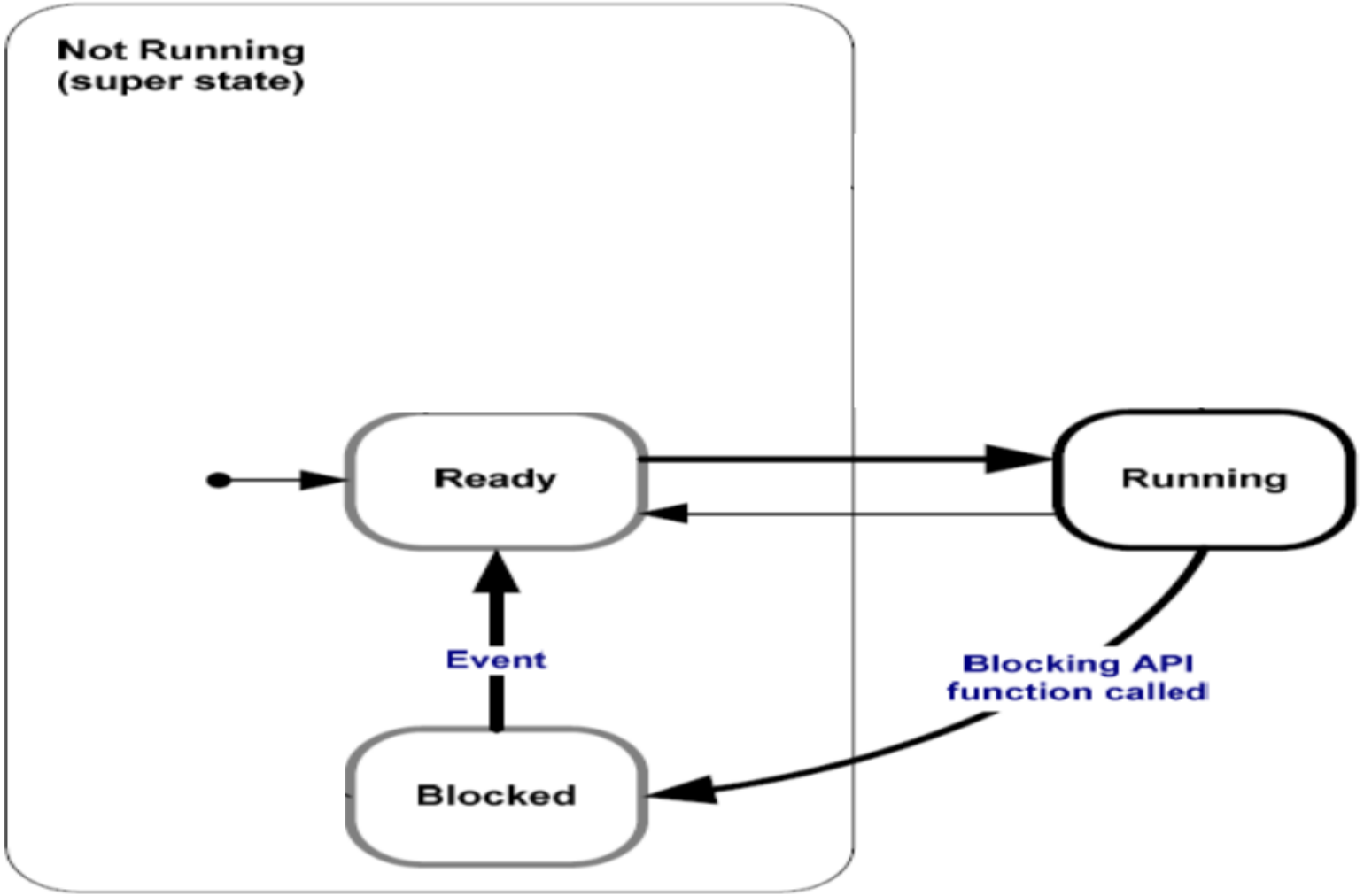


Figure 10. Bold lines indicate the state transitions performed by the tasks in Example 4

FreeRTOS Task SM

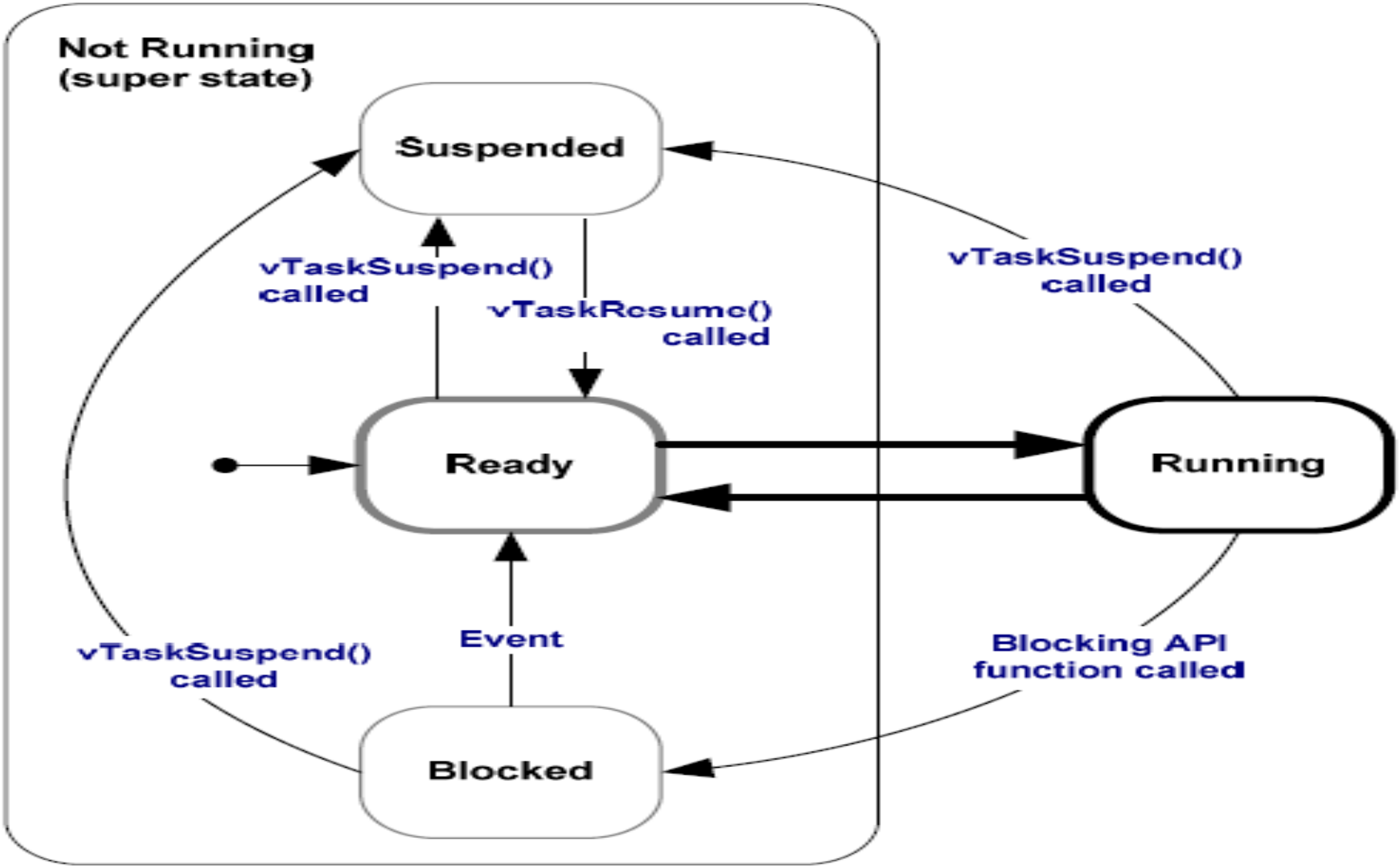


Figure 7. Full task state machine

FreeRTOSConfig.h | main.c | tasks.c | port.c | startup_MPS_CM3.s

```
103    configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.
104    for( ;; );
105  }
106  /*-----------------------------------------------------------
107
108  void vApplicationStackOverflowHook( xTaskHandle *pxTask, sign
109  {
110    /* This function will only be called if a task overflows it
111    that stack overflow checking does slow down the context swi
112    implementation and will only be performed if configCHECK_FO
113    is set to either 1 or 2 in FreeRTOSConfig.h. */
114    for( ;; );
115  }
116  /*-----------------------------------------------------------
117
118  void vApplicationIdleHook( void )
119  {
120    /* This example does not use the idle hook to perform any p
121    idle hook will only be called if configUSE_IDLE_HOOK is set
122    FreeRTOSConfig.h. */
123  }
124  /*-----------------------------------------------------------
125
126  void vApplicationTickHook( void )
127  {
128    /* This example does not use the tick hook to perform any p
```

→ OS call back to tell it what to do on idle task

Agenda

- **Accurate task periods**
- **Continuous & Periodic Tasks**
- **Changing Tasks priorities**
- **Deleting tasks**

Accurate Tasks Periods

vTaskDelayUntil()

- vTaskDelayUntil() is similar to vTaskDelay() "but"
- vTaskDelay() parameter specifies the number of tick interrupts that should occur between a task calling vTaskDelay() and the same task once again transitioning out of the Blocked state.
- The length of time the task remains in the blocked state is specified by the vTaskDelay() parameter, but the actual time at which the task leaves the blocked state is relative to the time at which vTaskDelay() was called.
- The parameters to vTaskDelayUntil() specify, instead, the exact tick count value at which the calling task should be moved from the Blocked state into the Ready state.
- vTaskDelayUntil() is the API function that should be used when a fixed execution period is required (where you want your task to execute periodically with a fixed frequency), as the time at which the calling task is unblocked is absolute, rather than relative to when the function was called (as is the case with vTaskDelay()).
- vTaskDelayUntil() API function is available only when INCLUDE_vTaskDelayUntil is set to 1 in FreeRTOSConfig.h.

vTaskDelayUntil()

- Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution frequency.
- This function differs from vTaskDelay() in one important aspect: vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called, whereas vTaskDelayUntil() specifies an absolute time at which the task wishes to unblock.
- vTaskDelay() will cause a task to block for the specified number of ticks from the time vTaskDelay() is called.
- It is therefore difficult to use vTaskDelay() by itself to generate a fixed execution frequency as the time between a task unblocking following a call to vTaskDelay() and that task next calling vTaskDelay() may not be fixed [the task may take a different path through the code between calls, or may get interrupted or preempted a different number of times each time it executes].
- Whereas vTaskDelay() specifies a wake time relative to the time at which the function is called, vTaskDelayUntil() specifies the absolute (exact) time at which it wishes to unblock.

Accurate Tasks Periods

vTaskDelayUntil()

- It should be noted that vTaskDelayUntil() will return immediately (without blocking) if it is used to specify a wake time that is already in the past.
- Therefore a task using vTaskDelayUntil() to execute periodically will have to re-calculate its required wake time if the periodic execution is halted for any reason (for example, the task is temporarily placed into the Suspended state) causing the task to miss one or more periodic executions.
- This can be detected by checking the variable passed by reference as the pxPreviousWakeTime parameter against the current tick count. This is however not necessary under most usage scenarios.
- The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

xTaskGetTickCount

```
volatile TickType_t xTaskGetTickCount( void );
```

This function cannot be called from an ISR. Use xTaskGetTickCountFromISR() instead.

Returns:

The count of ticks since vTaskStartScheduler was called.

Accurate Tasks Periods (Example 5)

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

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

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count.  Note that this is the only time the variable is written to explicitly.
    After this xLastWakeTime is updated automatically internally within
    vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();

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

        /* This task should execute exactly every 250 milliseconds.  As per
        the vTaskDelay() function, time is measured in ticks, and the
        portTICK_RATE_MS constant is used to convert milliseconds into ticks.
        xLastWakeTime is automatically updated within vTaskDelayUntil() so is not
        explicitly updated by the task. */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

Example 6: Continuous & Periodic Tasks

```
void vContinuousProcessingTask( 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.  This task just does this repeatedly
        without ever blocking or delaying. */
        vPrintString( pcTaskName );
    }
}
```

```
void vPeriodicTask( void *pvParameters )
{
portTickType xLastWakeTime;

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count.  Note that this is the only time the variable is explicitly written to.
    After this xLastWakeTime is managed automatically by the vTaskDelayUntil()
    API function. */
    xLastWakeTime = xTaskGetTickCount();

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

        /* The task should execute every 10 milliseconds exactly. */
        vTaskDelayUntil( &xLastWakeTime, ( 10 / portTICK_RATE_MS ) );
    }
}
```

Listing 16. The periodic task used in Example 6

Example 6: Continuous & Periodic Tasks

```
58  int main( void )
59  {
60    /* Create two instances of the continuous processing task, both at priority 1. */
61    xTaskCreate( vContinuousProcessingTask, "Task 1", 240, (void*)pcTextForTask1, 1, NULL );
62    xTaskCreate( vContinuousProcessingTask, "Task 2", 240, (void*)pcTextForTask2, 1, NULL );
63
64    /* Create one instance of the periodic task at priority 2. */
65  //  xTaskCreate( vPeriodicTask, "Task 3", 240, (void*)pcTextForPeriodicTask, 2, NULL );
66    xTaskCreate( vPeriodicTask, "Task 3", 240, (void*)pcTextForPeriodicTask, 2, NULL );
67
68    /* Start the scheduler so our tasks start executing. */
69    vTaskStartScheduler();
70
71    /* If all is well we will never reach here as the scheduler will now be
72    running.  If we do reach here then it is likely that there was insufficient
73    heap available for the idle task to be created. */
74    for( ;; );
75  }
```

round robin

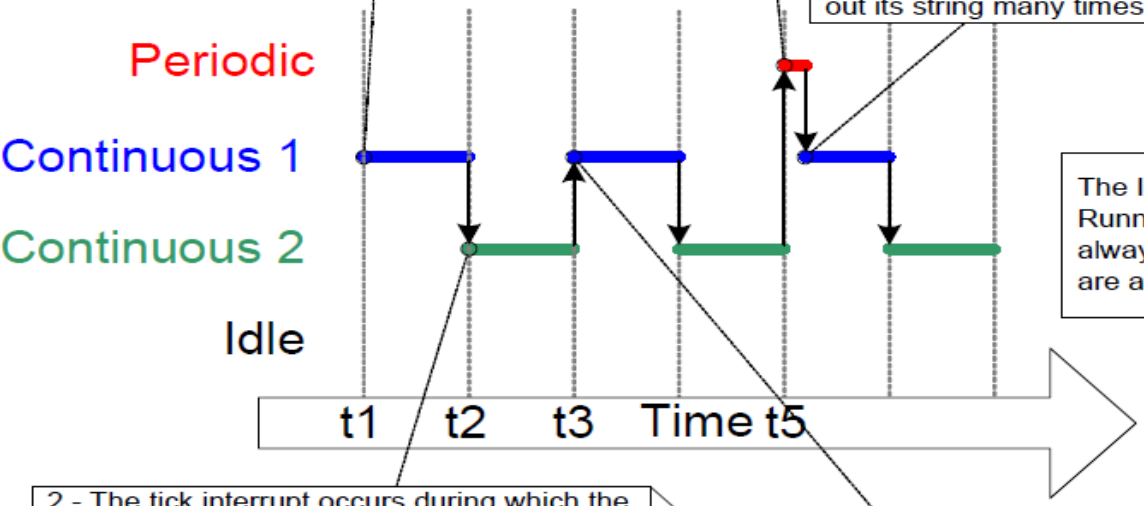
higher priority

Example 6: Continuous & Periodic Tasks

4 - At time t5 the tick interrupt finds that the Periodic task block period has expired so moved the Periodic task into the Ready state. The Periodic task is the highest priority task so immediately then enters the Running state where it prints out its string exactly once before calling vTaskDelayUntil() to return to the Blocked state.

1 - Continuous task 1 runs for a complete tick period (time slice between times t1 and t2) - during which time it could print out its string many times.

5 - The Periodic task entering the Blocked state means the scheduler has again to choose a task to enter the Running state - in this case Continuous 1 is chosen and it runs up to the next tick interrupt - during which time it could print out its string many times.

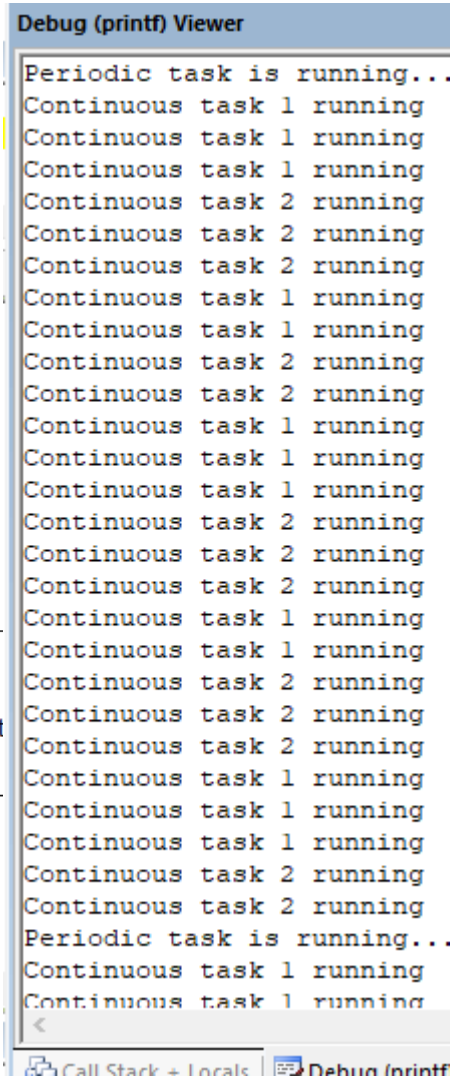


The Idle task never enters the Running state as there are always higher priority task that are able to do so.

2 - The tick interrupt occurs during which the scheduler selects a new task to run. As both Continuous tasks have the same priority and both are always able to run the scheduler shares processing time between the two - so Continuous 2 enters the Running state where it remains for the entire tick period - during which time it could print out its string many times.

3 - At time t3 the tick interrupt runs again, causing a switch back to Continuous 1, and so it goes on.

Figure 12. The execution pattern of Example 6



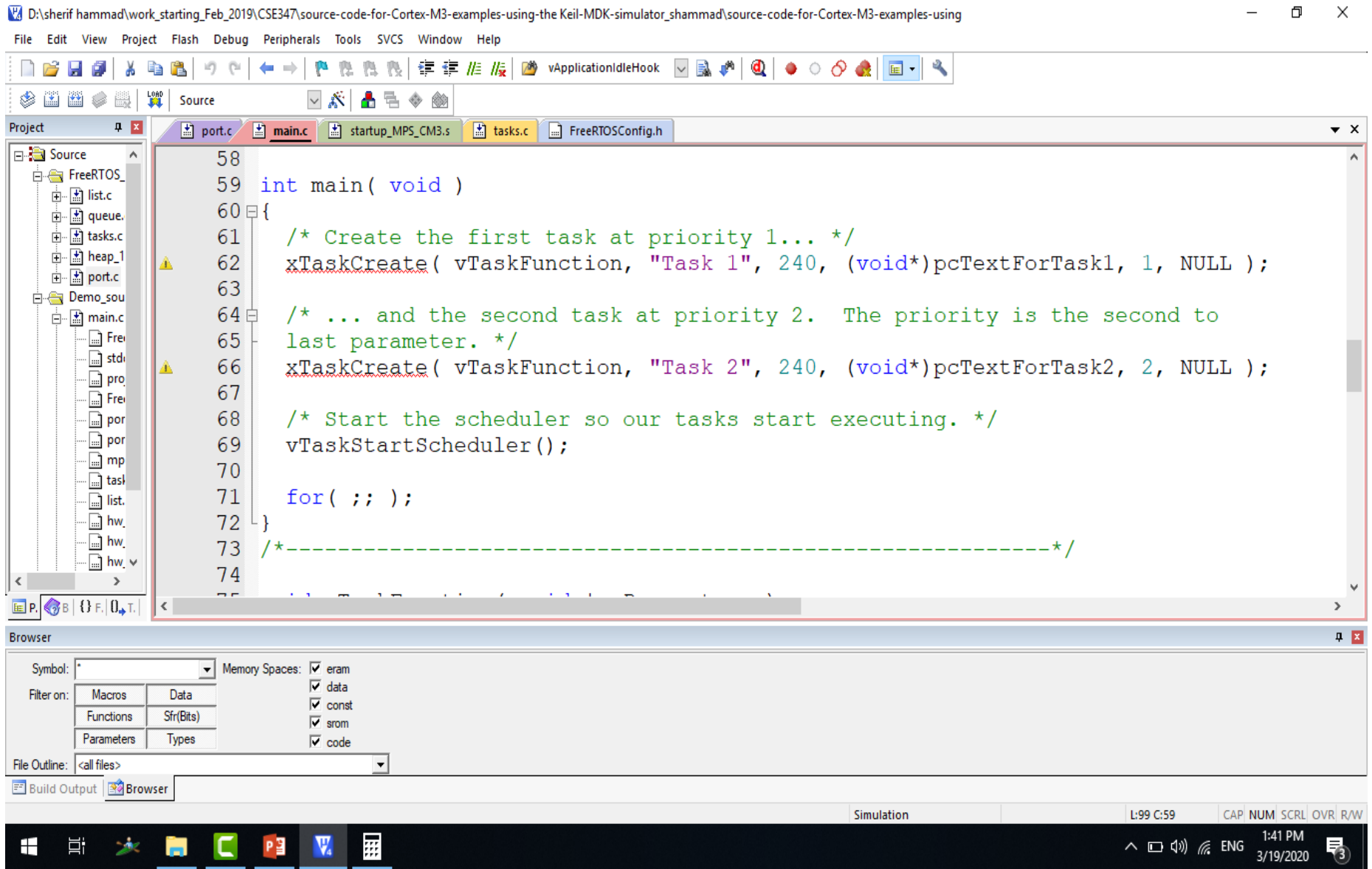
Priorities & Preemption

- **1_three_tasks_same_priority**
- **2_three_tasks_different_priority**
- **3_three_tasks_preemption_delete_task (Example 9)**

The Idle Task and the Idle Task Hook (Idle Task CallBack)

- An Idle task is automatically created by the scheduler when vTaskStartScheduler() is called.
- The idle task does very little more than sit in a loop
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state
- Idle task is always pre-empted by higher priority tasks
- idle hook (or idle callback) function—a function that is called automatically by the idle task once per iteration of the idle task loop
- Common uses for the Idle task hook include:
  - Executing low priority, background, or continuous processing.
  - Measuring the amount of spare processing capacity.
  - Placing the processor into a low power mode
- configUSE_IDLE_HOOK must be set to 1 within FreeRTOSConfig.h for the idle hook function to get called.

Example 7: Idle Task CallBack



Example 7: Idle Task CallBack

```
/* Declare a variable that will be incremented by the hook function. */
unsigned long ulIdleCycleCount = 0UL;

/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters,
and return void. */
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
}
```

Listing 18. A very simple Idle hook function

```
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 AND the number of times ulIdleCycleCount
        has been incremented. */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* Delay for a period of 250 milliseconds. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Listing 19. The source code for the example task prints out the ulIdleCycleCount

How many times IdelTask is called between tasks calls?
For how long the CPU is into Idle Task?

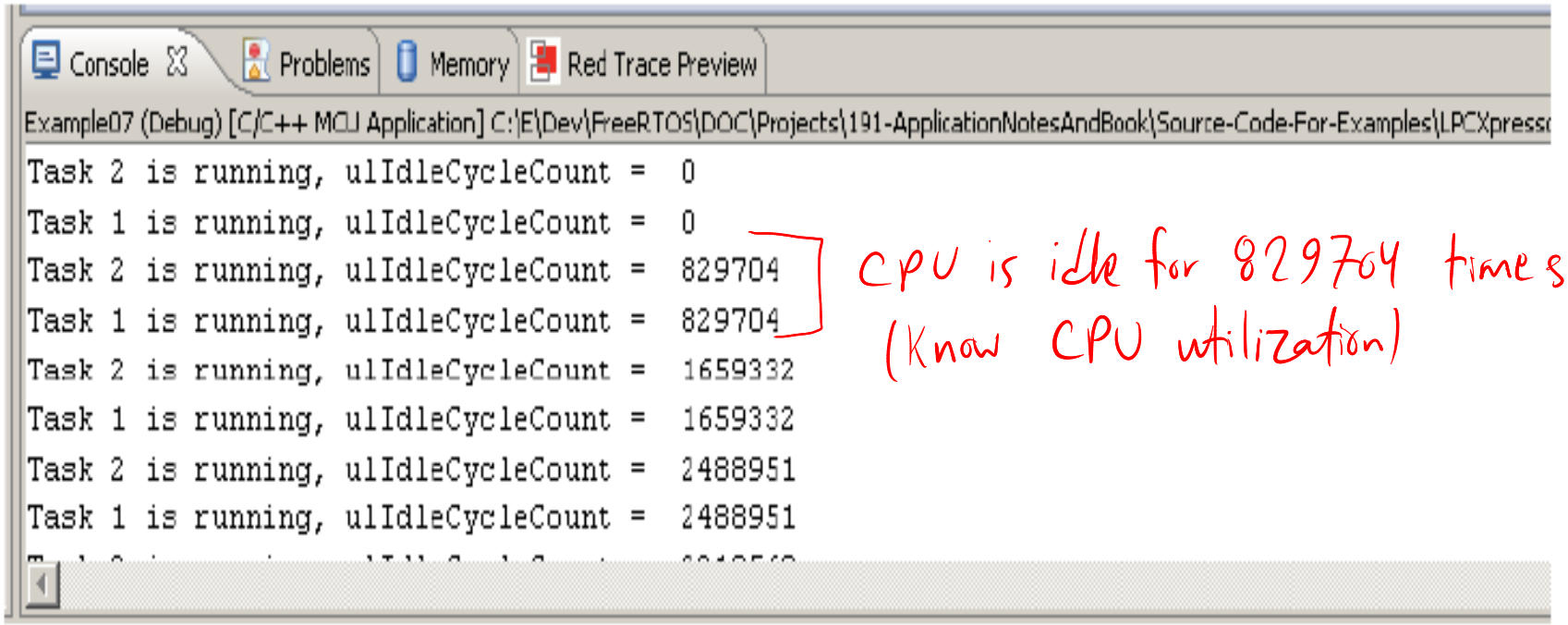


Figure 13. The output produced when Example 7 is executed

TaskHandle_t

task. h

Type by which tasks are referenced. For example, a call to xTaskCreate returns (via a pointer parameter) an TaskHandle_t variable that can then be used as a parameter to vTaskDelete to delete the task.

uxTaskPriorityGet

[Task Control]

task. h

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t xTask );
```

INCLUDE_uxTaskPriorityGet must be defined as 1 for this function to be available. See the RTOS Configuration documentation for more information.

Obtain the priority of any task.

Parameters:

xTask Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

Returns:

The priority of xTask.

vTaskPrioritySet

[Task Control]

task. h

```
void vTaskPrioritySet( TaskHandle_t xTask,
                       UBaseType_t uxNewPriority );
```

`INCLUDE_vTaskPrioritySet` must be defined as 1 for this function to be available. See the RTOS Configuration documentation for more information.

Set the priority of any task.

A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Parameters:

xTask	Handle of the task whose priority is being set. A NULL handle sets the priority of the calling task.
uxNewPriority	The priority to which the task will be set. Priorities are asserted to be less than `configMAX_PRIORITIES`. If `configASSERT` is undefined, priorities are silently capped at (`configMAX_PRIORITIES` - 1).

Changing task priorities; Example 8

```
/* Declare a variable that is used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;

int main( void )
{
    /* Create the first task at priority 2.  The task parameter is not used
    and set to NULL.  The task handle is also not used so is also set to NULL. */
    xTaskCreate( vTask1, "Task 1", 240, NULL, 2, NULL );
    /* The task is created at priority 2 _____^. */

    /* Create the second task at priority 1 - which is lower than the priority
    given to Task 1.  Again the task parameter is not used so is set to NULL -
    BUT this time the task handle is required so the address of xTask2Handle
    is passed in the last parameter. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, &xTask2Handle );
    /* The task handle is the last parameter _____^^^^^^^^^^^^^ */

    /* 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( ;; );
}
```

Listing 24. The implementation of main() for Example 8

Changing task priorities; Example 8

```
void vTask1( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* This task will always run before Task 2 as it is created with the higher
    priority.  Neither Task 1 nor Task 2 ever block so both will always be in either
    the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );
                                          = 2
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\n" );

        /* Setting the Task 2 priority above the Task 1 priority will cause
        Task 2 to immediately start running (as then Task 2 will have the higher
        priority of the two created tasks).  Note the use of the handle to task
        2 (xTask2Handle) in the call to vTaskPrioritySet().  Listing 24 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task 2 priority\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* Task 1 will only run when it has a priority higher than Task 2.
        Therefore, for this task to reach this point Task 2 must already have
        executed and set its priority back down to below the priority of this
        task. */
    }
}
```

Now, Task2 has higher priority and a context switch is done after API call.

Context switching is not started by Systick interrupt

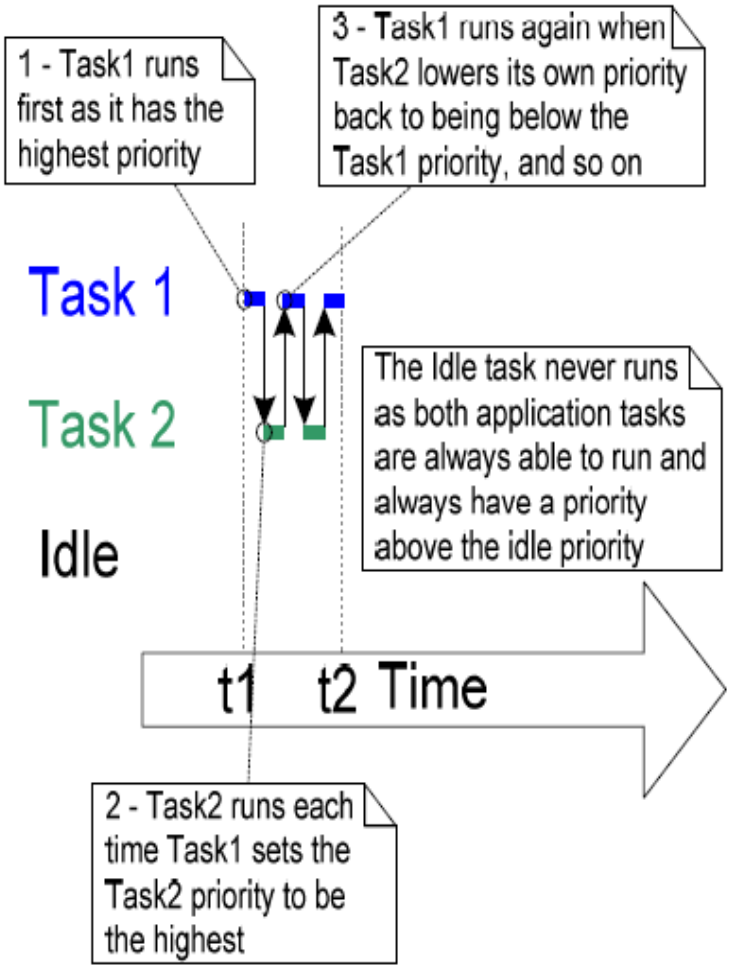
Listing 22. The implementation of Task 1 in Example 8

Changing task priorities; Example 8

```
void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* Task 1 will always run before this task as Task 1 is created with the
    higher priority.  Neither Task 1 nor Task 2 ever block so will always be
    in either the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* For this task to reach this point Task 1 must have already run and
        set the priority of this task higher than its own.

        Print out the name of this task. */
        vPrintString( "Task2 is running\n" );

        /* Set our priority back down to its original value.  Passing in NULL
        as the task handle means "change my priority".  Setting the
        priority below that of Task 1 will cause Task 1 to immediately start
        running again - pre-empting this task. */
        vPrintString( "About to lower the Task 2 priority\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

Listing 23. The implementation of Task 2 in Example 8

Changing task priorities; Example 8



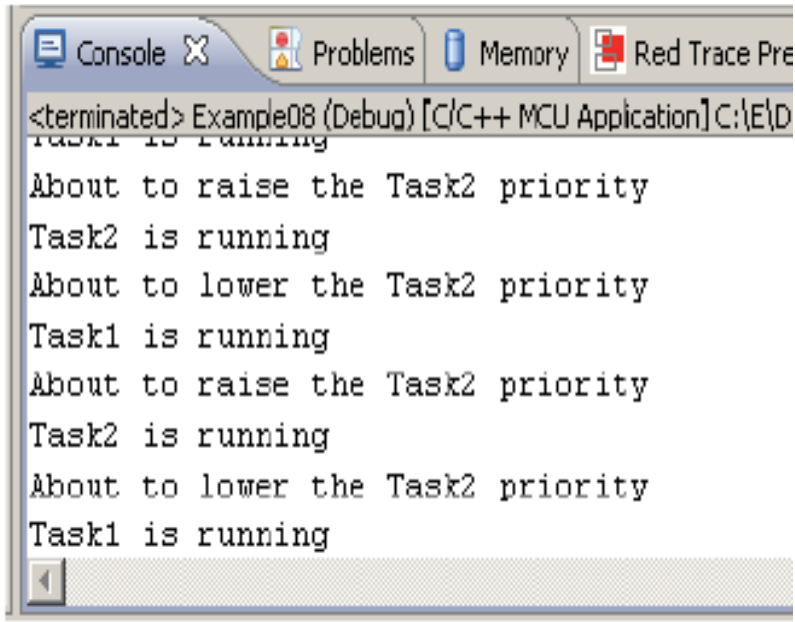


Figure 14. The sequence of task execution when running Example 8

Example X

Three periodic tasks

```
55 int main(void){
56  //  TExaS_Init(SW_PIN_PF40,LED_PIN_PF321);
57    // TExaS_Init initializes the real board grader for lab 4
58    PortF_Init();        // Call initialization of port PF4, PF3, PF2, PF1, PF0
59  //  EnableInterrupts();  // The grader uses interrupts
60    /* Start the standard LED flash tasks as defined in demo/common/minimal. */
61    /*  vStartLEDFlashTasks( mainLED_FLASH_PRIORITY );*/
62    xTaskCreate( vTask1, (const portCHAR *)"Task1", configMINIMAL_STACK_SIZE, NULL, mainLED_FLASH_PRIORITY+2, NULL );
63    xTaskCreate( vTask2, (const portCHAR *)"Task2", configMINIMAL_STACK_SIZE, NULL, mainLED_FLASH_PRIORITY+1, NULL );
64    xTaskCreate( vTask3, (const portCHAR *)"Task3", configMINIMAL_STACK_SIZE, NULL, mainLED_FLASH_PRIORITY, NULL );
65
66    /* Start the scheduler. */
67    vTaskStartScheduler();
68 }
```

Example X

```
71  static void vTask1( void *pvParameters )
72  {
73
74    /* Continuously perform a calculation.  If the calculation result is ever
75    incorrect turn the LED on. */
76    for( ;; )
77    {
78      GPIO_PORTF_DATA_R = 0x04;        // LED is blue
79      vTaskDelay(330);
80    }
81  }
```

```
86  static void vTask2( void *pvParameters )
87  {
88
89    /* Continuously perform a calculation.  If the calculation result is ever
90    incorrect turn the LED on. */
91    for( ;; )
92    {
93      GPIO_PORTF_DATA_R = 0x02;      // LED is red
94      vTaskDelay(660);
95    }
96  }
```

```
99  static void vTask3( void *pvParameters )
100 {
101
102   /* Continuously perform a calculation.  If the calculation result is ever
103   incorrect turn the LED on. */
104   for( ;; )
105   {
106     GPIO_PORTF_DATA_R = 0x08;    // LED is green
107     vTaskDelay(990);
108   }
109 }
```

vTaskDelete

[Task Creation]

task. h

```
void vTaskDelete( TaskHandle_t xTask );
```

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the RTOS Configuration documentation for more information.

Remove a task from the RTOS kernels management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

NOTE: The idle task is responsible for freeing the RTOS kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

See the demo application file death. c for sample code that utilises vTaskDelete ().

Parameters:

xTask The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

Example usage:

```
void vOtherFunction( void )
{
TaskHandle_t xHandle = NULL;

    // Create the task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // Use the handle to delete the task.
    if( xHandle != NULL )
    {
        vTaskDelete( xHandle );
    }
}
```

Deleting a Task; Example 9

```
int main( void )
{
    /* Create the first task at priority 1.  The task parameter is not used
    so is set to NULL.  The task handle is also not used so likewise is set
    to NULL. */
    xTaskCreate( vTask1, "Task 1", 240, NULL, 1, NULL );
    /* The task is created at priority 1 ______^. */

    /* Start the scheduler so the task starts executing. */
    vTaskStartScheduler();

    /* main() should never reach here as the scheduler has been started. */
    for( ;; );
}
```

Listing 26. The implementation of main() for Example 9

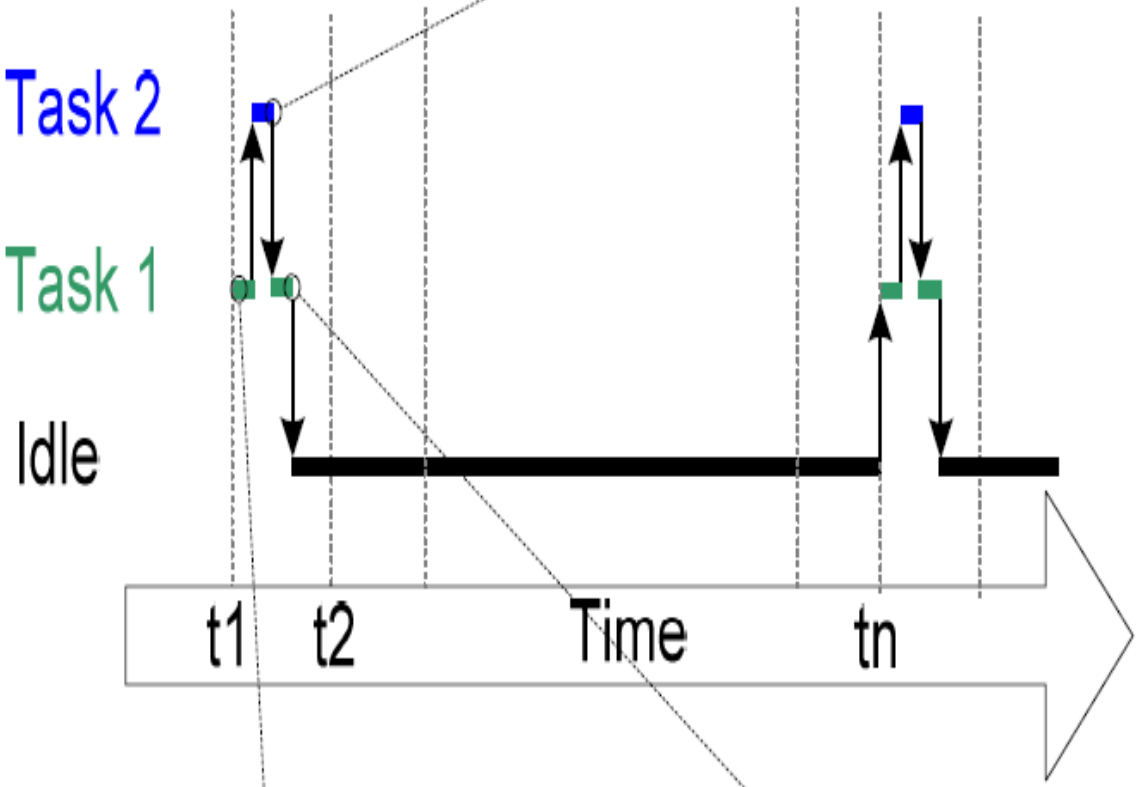
```
void vTask1( void *pvParameters )
{
const portTickType xDelay100ms = 100 / portTICK_RATE_MS;

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\n" );

        /* Create task 2 at a higher priority.  Again the task parameter is not
        used so is set to NULL - BUT this time the task handle is required so
        the address of xTask2Handle is passed as the last parameter. */
        xTaskCreate( vTask2, "Task 2", 240, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter _____^^^^^^^^^^^^^^ */

        /* Task 2 has/had the higher priority, so for Task 1 to reach here Task 2
        must have already executed and deleted itself.  Delay for 100
        milliseconds. */
        vTaskDelay( xDelay100ms );
    }
}
```

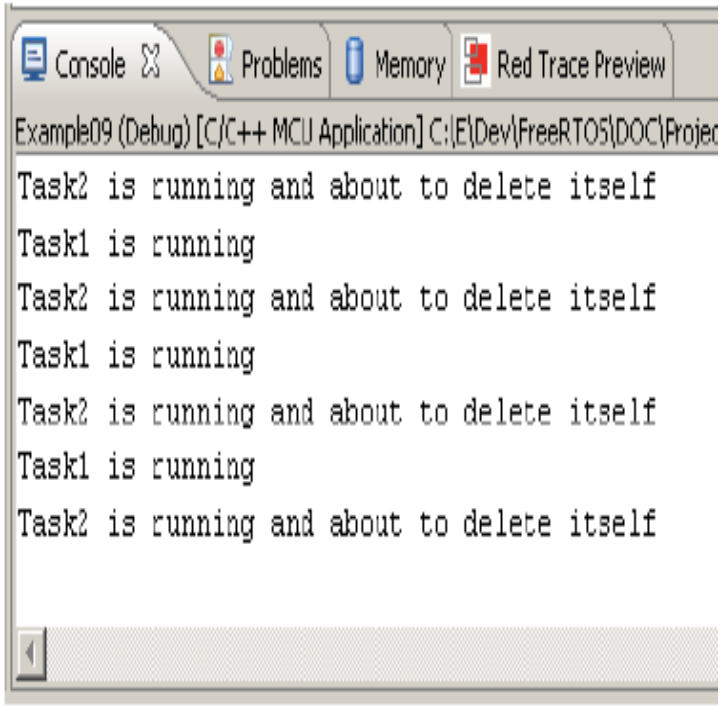
→ context switching

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself.  To do this it could call vTaskDelete()
    using NULL as the parameter, but instead and purely for demonstration purposes it
    instead calls vTaskDelete() passing its own task handle. */
    vPrintString( "Task2 is running and about to delete itself\n" );
    vTaskDelete( xTask2Handle );
}
```

Task 2 Preempts 1

2 - Task 2 does nothing other than delete itself, allowing execution to return to Task 1.





1 - Task 1 runs and creates Task 2. Task 2 starts to run immediately as it has the higher priority.

3 - Task 1 calls vTaskDelay(), allowing the idle task to run until the delay time expires, and the whole sequence repeats.

Task 1 Resumes

Figure 17. The execution sequence for Example 9

Priorities & Preemption

- **1_three_tasks_same_priority**
- **2_three_tasks_different_priority**
- **3_three_tasks_preemption_delete_task (Example 9)**

The Scheduling Algorithm—A Summary

- **Fixed Prioritized Pre-emptive Scheduling**
  - **Each task is assigned a priority.**
  - **Each task can exist in one of several states.**
  - **Only one task can exist in the Running state at any one time.**
  - **The scheduler always selects the highest priority Ready state task to enter the Running state.**

- **Fixed Priority' because each task is assigned a priority that is not altered by the kernel itself (only tasks can change priorities);**

- **'Pre-emptive' because a task entering the Ready state or having its priority altered will always pre-empt the Running state task, if the Running state task has a lower priority.**

The Scheduling Algorithm—A Summary

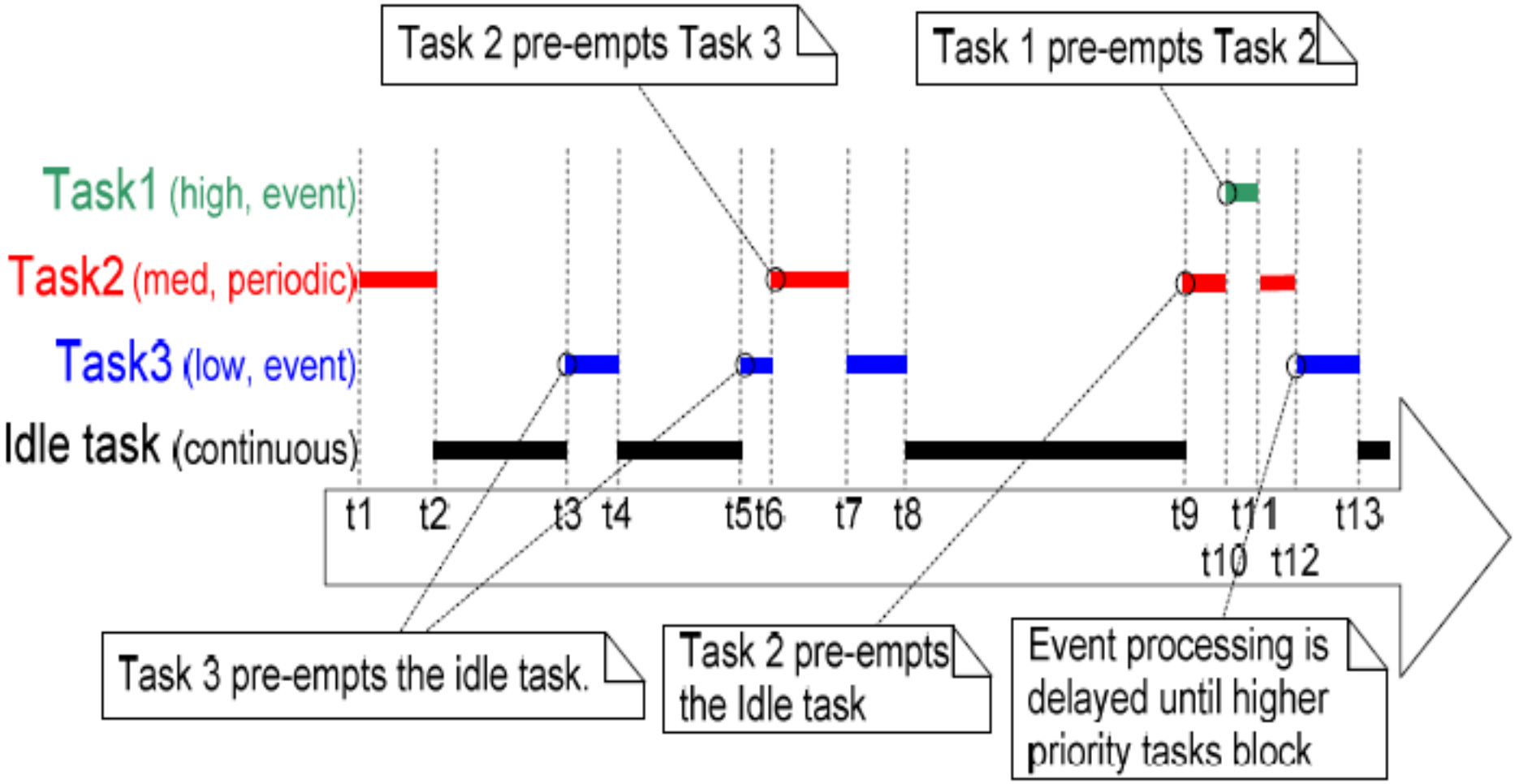


Figure 18. Execution pattern with pre-emption points highlighted