FreeRTOS

A real time operating system for embedded systems

Introduction to Multitasking in Small Embedded Systems

- Most embedded real-time applications include a mix of both hard and soft real-time requirements.
- Soft real-time requirements
  – State a time deadline, but breaching (violating) the deadline would **not render the system useless**.
    - E.g., responding to keystrokes too slowly
- Hard real-time requirements
  – State a time deadline, but breaching the deadline would **result in absolute failure of the system**.
    - E.g., a driver's airbag would be useless if it responded to crash event too slowly.

FreeRTOS

- FreeRTOS is a real-time kernel/scheduler on top of which MCU applications can be built to meet their hard real-time requirements.
  – Allows MCU applications be organized as a collection of independent threads of execution.
  – Decides which thread should be executed by examining the priority assigned to each thread.
    - Assume a single core MCU, where only a single thread can be executing at one time.

The simplest case of task priority assignments

- Assign higher priorities (lower priorities) to threads that implement hard real-time (soft real-time) requirements
  - As a result, hard real-time threads are always executed ahead of soft real-time threads.
- But, priority assignment decision are not always that simple.

- In general, **task prioritization** can help ensure an application meet its processing deadline.

A note about terminology

- In FreeRTOS, each thread of execution is called a ‘task’.

Why use a real-time kernel

- For a simple system, many well-established techniques can provide an appropriate solution without the use of a kernel.
- For a more complex embedded application, a kernel would be preferable.
- But where the crossover point occurs will always be subjective.

- Besides ensuring an application meets its processing deadline, a kernel can bring other less obvious benefits.

Benefits of using real-time kernel 1

- Abstracting away timing information
  - Kernel is responsible for execution timing and provides a time-related API to the application. This allows the application code to be simpler and the overall code size be smaller.
- Maintainability/Extensibility
  - Abstracting away timing details results in fewer interdependencies between modules and allows sw to evolve in a predictable way.
  - Application performance is less susceptible to changes in the underlying hardware.
- Modularity
  - Tasks are independent modules, each of which has a well-defined purpose.
- Team development
  - Tasks have well-defined interfaces, allowing easier development by teams

Benefits of using real-time kernel 2

- Easier testing
  - Tasks are independent modules with clean interfaces, they can be tested in isolation.
- Idle time utilization
  - The idle task is created automatically when the kernel is started. It executes whenever there are no application tasks to run.
  - Be used to measure spare processing capacity, perform background checks, or simply place the process into a low-power mode.
- Flexible interrupt handling
  - Interrupt handlers can be kept very short by deferring most of the required processing to handler tasks.

Standard FreeRTOS features

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Queues
- Binary/Counting / Recursive semaphores
- Mutexes
- Tick/Idle hook functions
- Stack overflow checking
- Trace hook macros
- Interrupt nesting

Outline

- Task Management
- Queue Management
- Interrupt Management
- Resource Management
- Memory Management
- Trouble Shooting

TASK MANAGEMENT

1.1 Introduction and scope

- Main topics to be covered
  - How FreeRTOS allocates processing time to each task within an application
  - How FreeRTOS chooses which task should execute at any given time
  - How the relative priority of each task affects system behavior
  - The states that a task can exist in.

More specific topics

- How to implement tasks
- How to create one or more instances of a task
- How to use the task parameter
- How to change the priority of a task that has already been created.
- How to delete a task.
- How to implement periodic processing.
- When the idle task will execute and how it can be used.

1.2 Task functions

- Tasks are implemented as C functions.
  - Special: Its prototype must return void and take a void pointer parameter as the following

  void ATaskFunction (void *pvParameters);

- Each task is a small program in its own right.
  - Has an entry point
  - Normally runs forever within an infinite loop
  - Does not exit

ATaskFunction

```
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 in 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 );
}
```

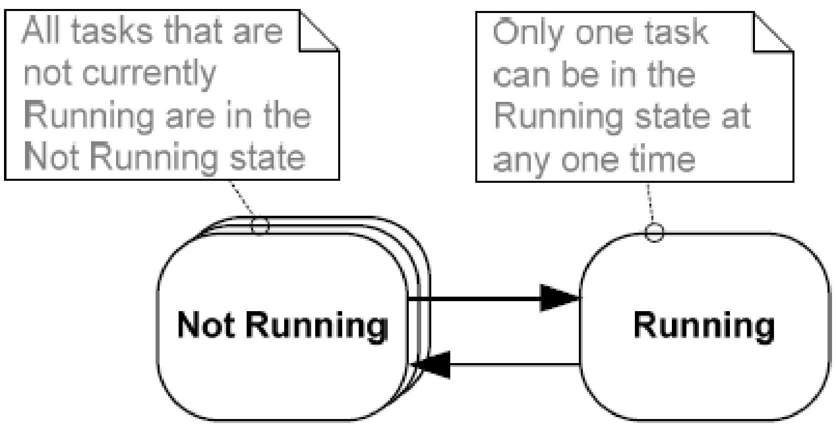
Listing 2 The structure of a typical task function

Special features of task function

- FreeRTOS task
  - Must not contain a '**return**' statement
  - Must not be allowed to execute past the end of the function
  - If a task is no longer required, it should be **explicitly** deleted.
  - Be used to create any number of tasks
    - Each created task is a separate execution instance with its own stack, and its own copy of any automatic variables defined within the task itself.

1.3 Top level task states

- A task can exist in one of two states: Running and Not Running



Running state: the processor is executing its code.

Not Running state: the task is dormant, its status having been saved ready for resuming execution the next time

- Scheduler is the only entity that can switch a task in and out a running state.

1.4 Creating Tasks

- xTaskCreate() API function
  - the most fundamental component in a multitasking system
    - Probably the most complex of all API functions

```
portBASE_TYPE  xTaskCreate(
      pdTASK_CODE  pvTaskCode,
      const signed char          * const pcName,
      unsigned short             usStackDepth,
      void                       *pvParameters,
      unsigned portBASE_TYPE uxPriority,
      xTaskHandle                *pxCreatedTask
);
```

All parameters

- pvTaskCode
  - a pointer to the function (just the function name) that implements the task.
- pcName
  - A descriptive name for the task. It is not used by FreeRTOS, but a debugging aid.
  - configMAX_TASK_NAME_LEN: the application defined constant that defines the maximum length a task name can task including the NULL terminator.

- usStackDepth
  - Each task has its own unique stack that is allocated by the kernel to the task when the task is created.
  - The value specifies the number of **words** the task stack can hold.
    - E.g., Cortex-M3 stack is 32 bits wide, if usStackDepth is passed in as 100, 400 bytes of stack space will be allocated (100*4 bytes)
  - Size of the stack used by the idle task is defined by configMINIMAL_STACK_SIZE.
    - Adjustable w.r.t. applications

- pvParameters
  - The value assigned to *pvParameters* will be the values passed into the task.
- uxPriority
  - defines the priority at which the task will execute.
  - Priorities can be assigned from 0, which is the lowest priority, to *(configMAX_PRIOIRTIES-1),* which is the highest priority.
  - Passing a value above (configMAX_PRIOIRTIES - 1) will result in the priority being capped the maximum legitimate value.

- pxCreatedTask
  - pass out a handle to the created task, then be used to refer the created task in API calls.
    - E.g., change the task priority or delete the task
  - Be set to NULL if no use for the task handle

- Two possible return values
  - pdTRUE : task has been created successfully.
  - errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY
    - Task has not been created as there is insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack.

Example 1 Creating 2 tasks

- To demonstrate the steps of creating two tasks then starting the tasks executing.
  – 2 Tasks simply print out a string periodically, using a crude null loop to create the periodic delay.
  – Both tasks are created at the same priority and are identical except for the string they print out.

Task1

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\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. */
        }
    }
}
```

Listing 4 Implementation of the first task used in Example 1

Task2

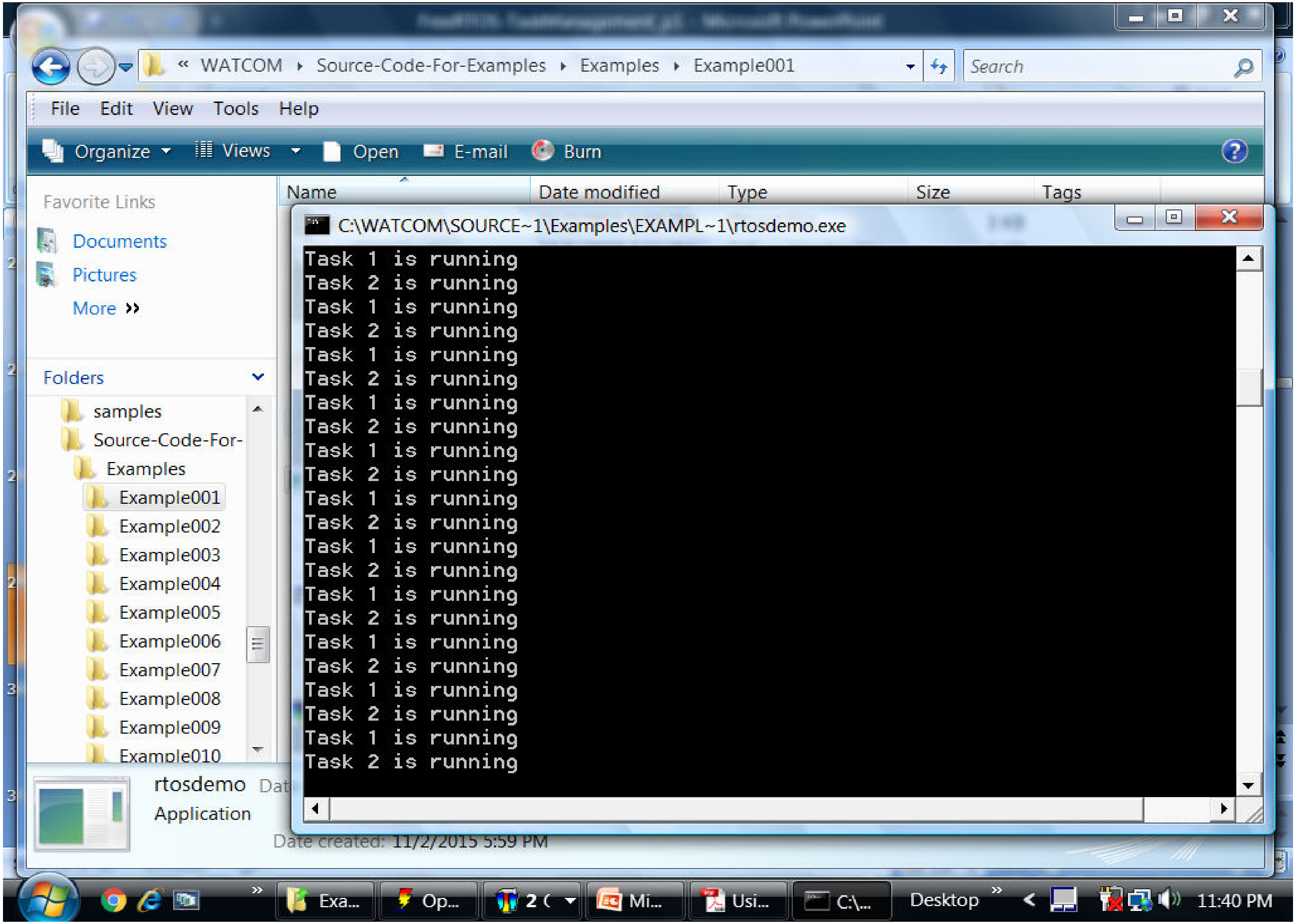
```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile unsigned long ul;

    /* 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. */
        }
    }
}
```

Listing 5 Implementation of the second task used in Example 1

main()

```c
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. */
                    1000,    /* Stack depth - most small microcontrollers will use much
                             less stack than this. */
                    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", 1000, 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( ;; );
}
```

Listing 6 Starting the Example 1 tasks


« WATCOM › Source-Code-For-Examples › Examples › Example001
Search
File Edit View Tools Help
Organize Views Open E-mail Burn
Name Date modified Type Size Tags
Favorite Links
Documents
Pictures
More »
Folders
samples
Source-Code-For-
Examples
Example001
Example002
Example003
Example004
Example005
Example006
Example007
Example008
Example009
Example010
rtosdemo
Application
Date created: 11/2/2015 5:59 PM
C:\WATCOM\SOURCE~1\Examples\EXAMPL~1\rtosdemo.exe
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
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
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Exa...
Op...
Mi...
Usi...
C:\...
Desktop
11:40 PM

Execution pattern of two Example 1 tasks

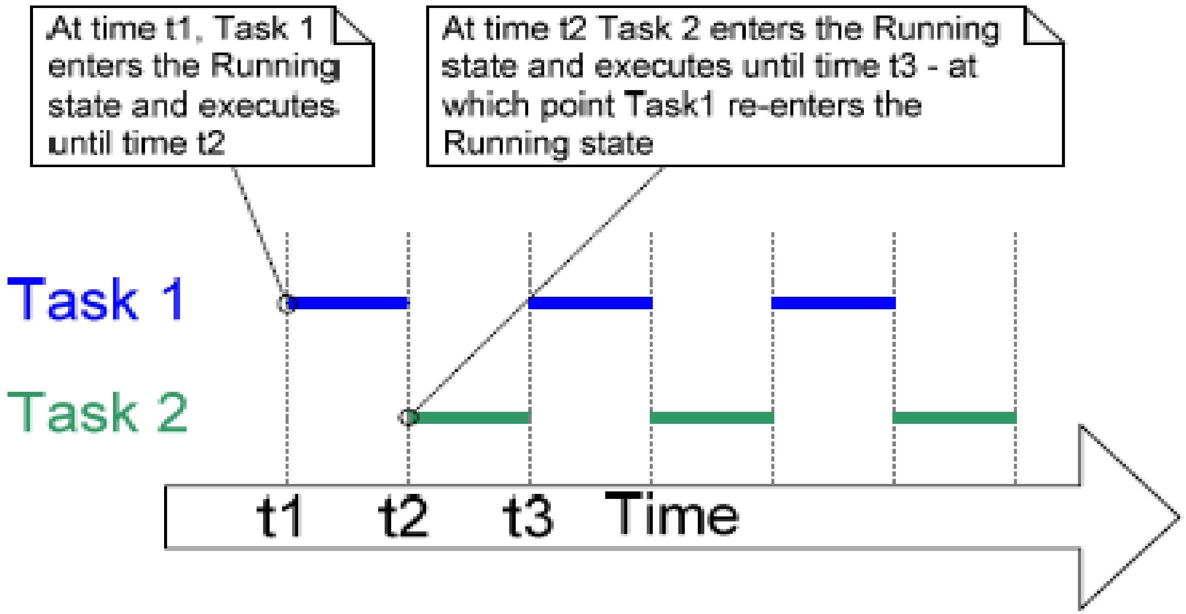


Figure 3 The actual execution pattern of the two Example 1 tasks

- Both tasks are rapidly entering and exiting the Running state.
- Only one task can exist in the Running state at any one time.

Example 2 Using the task parameter

- The two tasks in Example 1 are almost identical, the only difference between them being the text string they print out.
  - Remove such duplication by creating two instances of a single task implementation *vTaskFunction*.
    - Each created instance will execute independently under the control of the FreeRTOS scheduler.
  - The task parameter can then be used to pass into each task the string that it should print out.

Example 2 vTaskFunction

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

Listing 8 The single task function used to create two tasks in Example 2

```c
static const char *pcTextForTask1 = “Task 1 is running\r\n”;
static const char *pcTextForTask2 = “Task 2 is running\t\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. */
                    1000,                   /* Stack depth - most small microcontrollers
                                            will use much less stack than this. */
                    (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", 1000, (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( ;; );
}
```

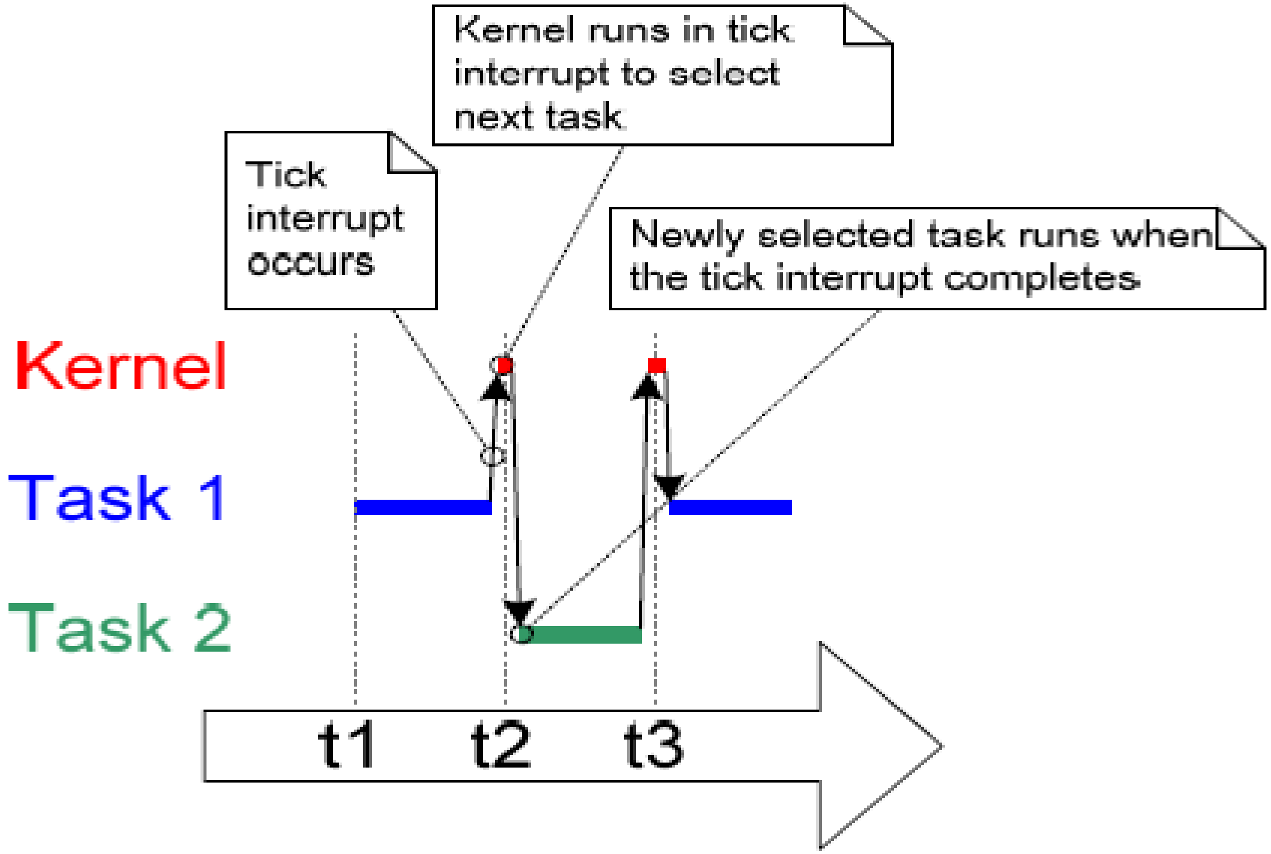
Listing 9 The main() function for Example 2.

1.5 Task priorities

- uxPriority parameter of xTaskCreate() assigns an initial priority to the task being created.
  - It can be changed after the scheduler has been started by using vTaskPrioritySet() API function

- configMAX_RPIORITIES in FreeRTOSConfig.h
  - Maximum number of priorities
  - Higher this value, more RAM consumed
  - Range: [0(low), configMAX-PRIORITIES-1(high)]
  - Any number of tasks can share the same priority

- To select the next task to run, the scheduler itself must execute at the end of each time slice.
  - Use a periodic interrupt called the tick (interrupt).
  - Effectively set the length of time slice by the tick interrupt frequency -- configTICK_RATE_HZ in FreeRTOSConfig.h
- configTICK_RATE_HZ
  - If it is 100(Hz), the time slice will be 10 ms.
  - API always calls specify time in tick interrupts (ticks)
- portTICK_RATE_MS
  - Convert time delays from milliseconds into the number of tick interrupts.



- When kernel itself is running, the arrows in the above figure show the sequence of execution from task interrupt, then from interrupt back to a next task.

Example 3. Experimenting with priorities

- Scheduler always ensures that the highest priority task is able to run is the task selected to enter the Running state.
  - In Example 1 and 2, two tasks have been created at the same priority, so both entered and exited the Running state in turn.
  - In this example, the second task is set at priority 2.

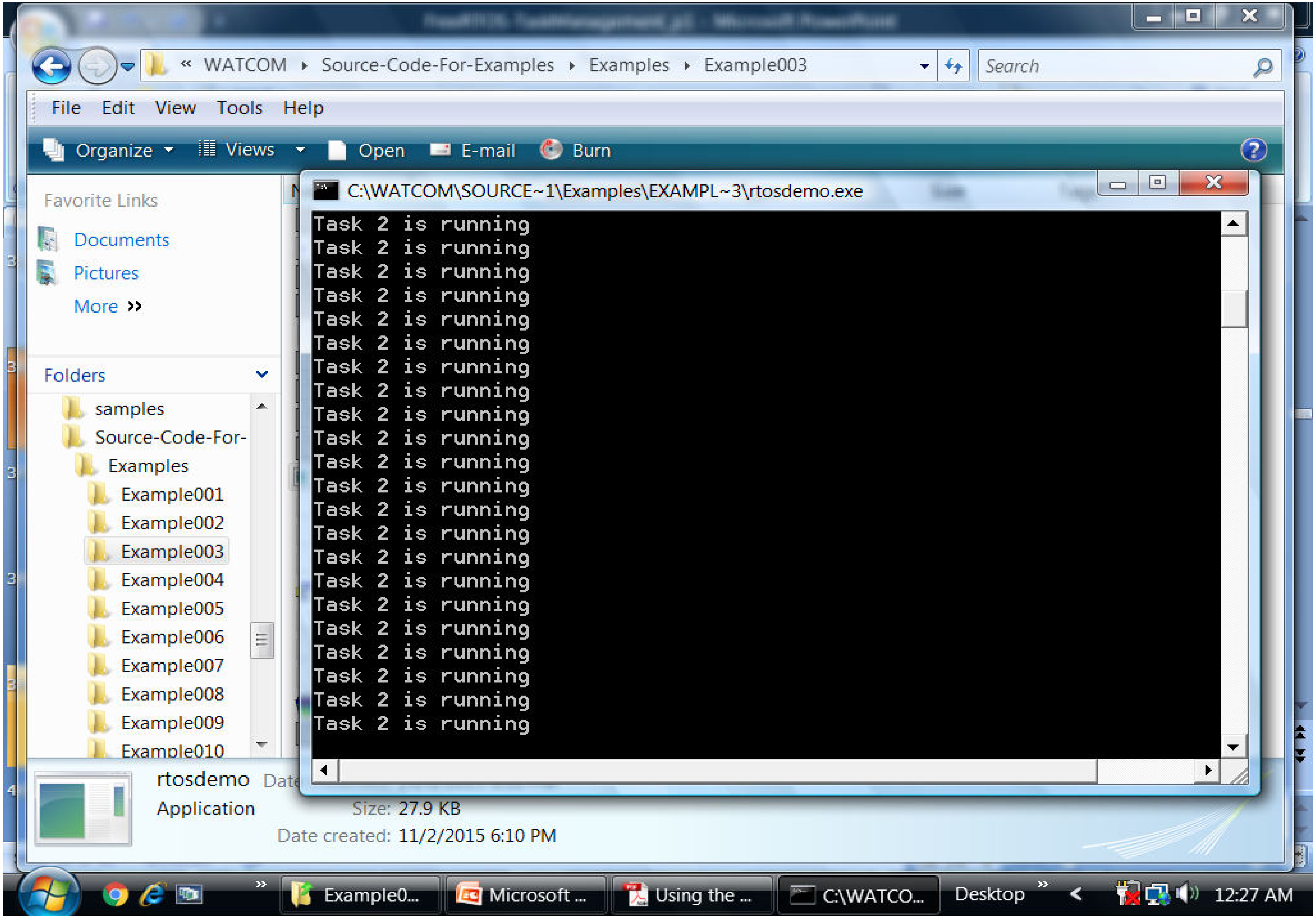
Example 3. Experimenting with priorities

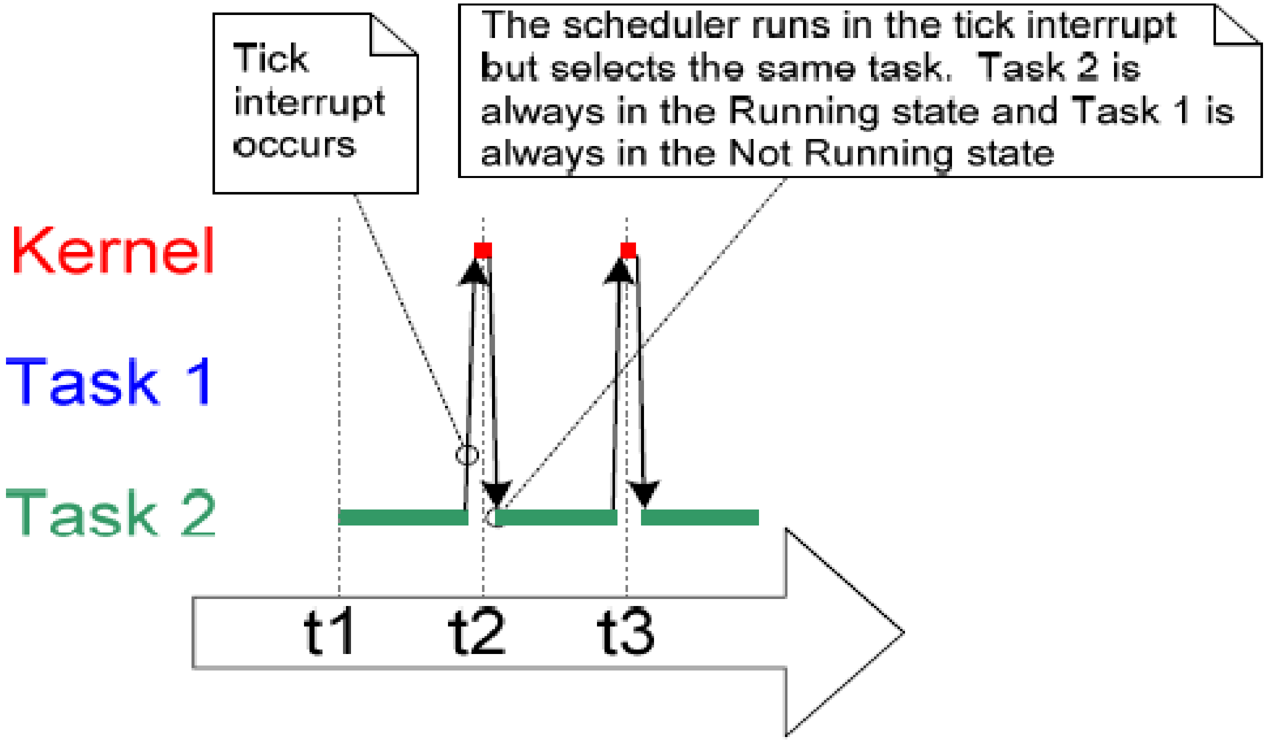
- In main() function, change

  ```
  xTaskCreate( vTaskFunction, "Task 2", 240,
  (void*)pcTextForTask2, 1, NULL );
  ```

  to

  ```
  xTaskCreate( vTaskFunction, "Task 2", 240,
  (void*)pcTextForTask2, 2, NULL );
  ```


WATCOM › Source-Code-For-Examples › Examples › Example003
Search
File Edit View Tools Help
Organize Views Open E-mail Burn
Favorite Links
Documents
Pictures
More »
Folders
samples
Source-Code-For-
Examples
Example001
Example002
Example003
Example004
Example005
Example006
Example007
Example008
Example009
Example010
C:\WATCOM\SOURCE~1\Examples\EXAMPL~3\rtosdemo.exe
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
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
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
Task 2 is running
rtosdemo
Application
Size: 27.9 KB
Date created: 11/2/2015 6:10 PM
Example0...
Microsoft ...
Using the ...
C:\WATCO...
Desktop
12:27 AM



- The scheduler always selects the highest priority task that is able to run.
  - Task 2 has a higher priority than Task 1; so Task 2 is the only task to ever enter the Running state.
  - Task 1 is to be ‘starved’ of processing time of Task 2.

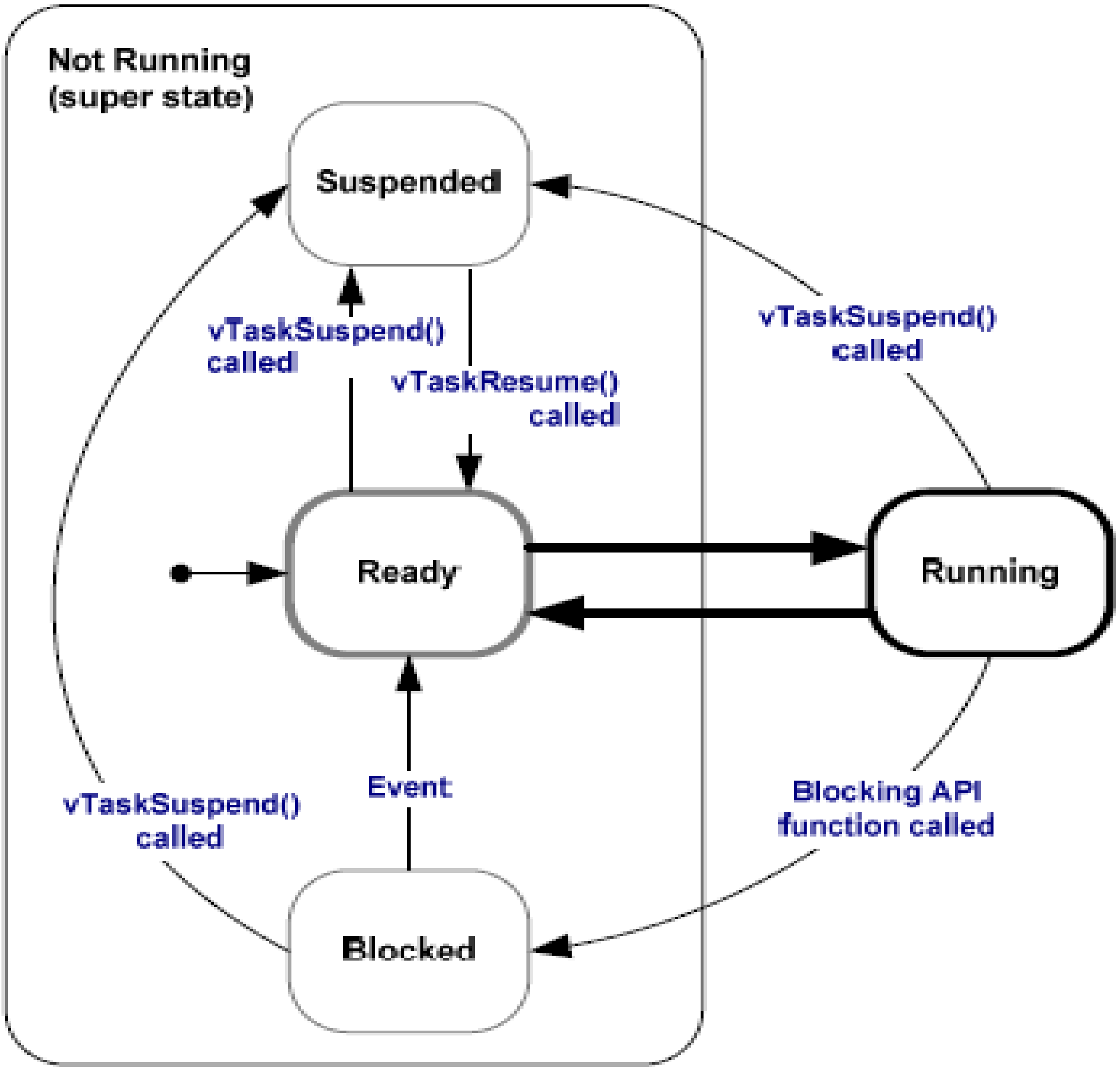
‘Continuous processing’ task

- So far, the created tasks always have work to perform and have never had to wait for anything
  - Always able to enter the Running state.
- This type of task has limited usefulness as they can only be created at the very lowest priority.
  - If they run at any other priority, they will prevent tasks of lower priority ever running at all.
- Solution: Event-driven tasks

1.6 Expanding the ‘Not Running’ state

- An event-driven task
  - has work to perform only after the occurrence of the event that triggers it
  - Is not *able* enter the Running state before that event has occurred.
- The scheduler selects the highest priority task that *is able to* run.
  - High priority tasks not being able to run means that the scheduler cannot select them, and
  - Must select a lower priority task that is able to run.
- Using event-driven tasks means that
  - tasks can be created at different priorities without the highest priority tasks starving all the lower priority tasks.

Full task state machine



Blocked state

- Tasks enter this state to wait for two types of events
  - Temporal (time-related) events: the event being either a delay expiring, or an absolute time being reached.
    - A task enter the Blocked state to wait for 10ms to pass
  - Synchronization events: where the events originate from another task or interrupt
    - A task enter the Blocked state to wait for data to arrive on a queue.
  - Can block on a synchronization event with a timeout, effectively block on both types of event simultaneously.
    - A task waits for a maximum of 10ms for data to arrive on a queue. It leaves the Blocked state if either data arrives within 10ms or 10ms pass with no data arriving.

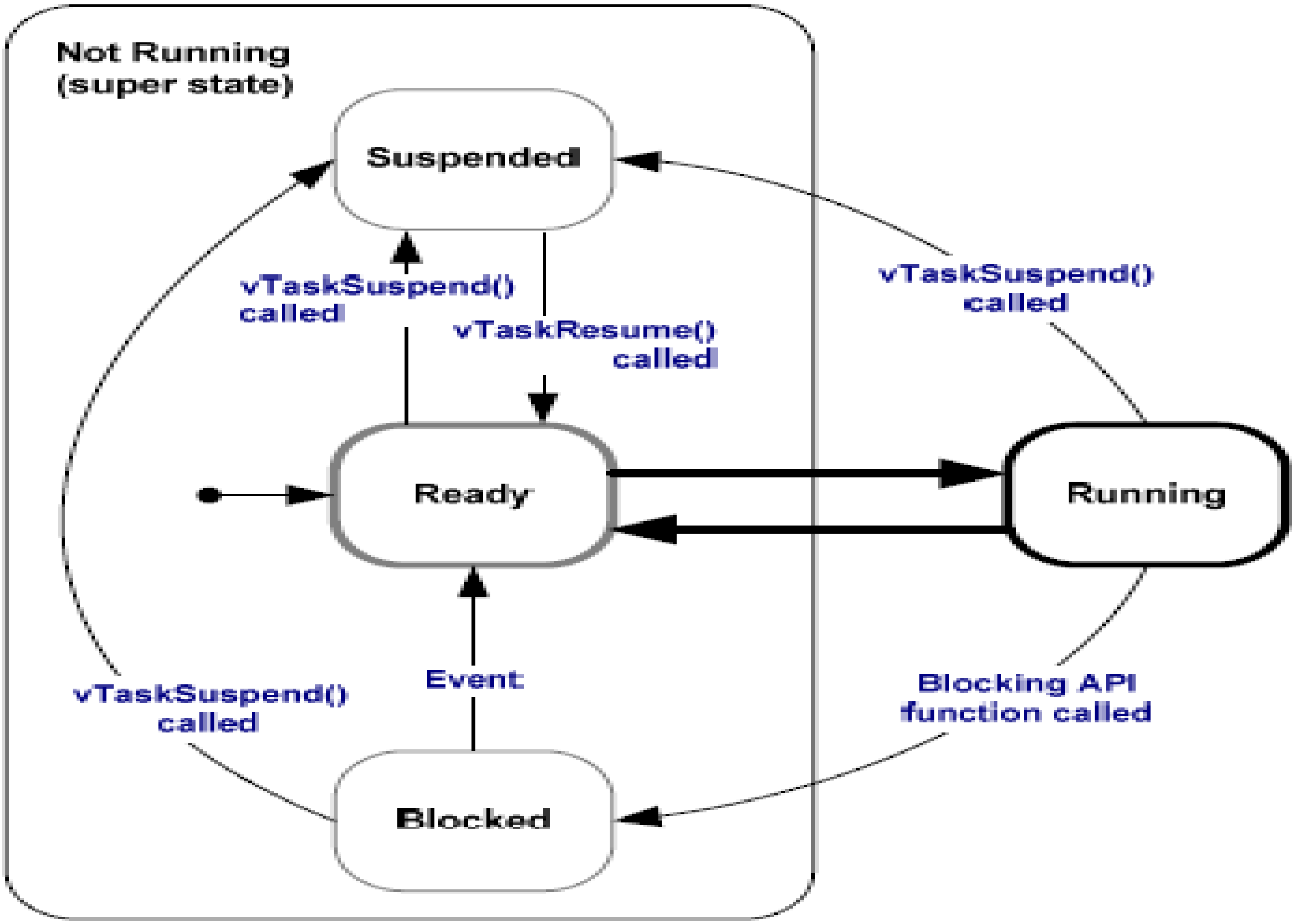
Suspended state

- Tasks in this state are not available to the scheduler.
  - The only way into this state is through a call to the *vTaskSuspend*() API function
  - The only way out this state is through a call to the *vTaskResume*() or *vTaskResumeFromISR*() API functions

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 are not currently in the Running state.

Full Task State machine



Example 4 Using the Block state to create a delay

- All tasks in the previous examples have been periodic
  - They have delayed for a period and printed out their string before delay once more, and so on.
  - Delayed generated using a null loop
    - the task effectively polled an incrementing loop counter until it reached a fixed value.

- Disadvantages to any form of polling
  - While executing the null loop, the task remains in the Ready state, ‘starving’ the other task of any processing time.
  - During polling, the task does not really have any work to do, but it still uses maximum processing time and so wastes processor cycles.
- This example corrects this behavior by
  - replacing the polling null loop with a call to vTaskDelay() API function.
  - setting INCLUDE_vTaskDelay to 1 in FreeRTOSConfig.h

vTaskDelay() API function

- Place the calling task into the Blocked state for a fixed number of tick interrupts.
  - The Blocked state task does not use any processing time, so processing time is consumed only when there is work to be done.

  void vTaskDelay(portTickType xTicksToDelay);

xTicksToDelay: the number of ticks that the calling task should remain in the Blocked state before being transitioned back into the Ready state.

E.g, if a task called vTaskDelay(100) while the tick count was 10,000, it enters the Blocked state immediately and remains there until the tick count is 10,100.

Example 4

- In void vTaskFunction(void *pvParameters)

  Change a NULL loop

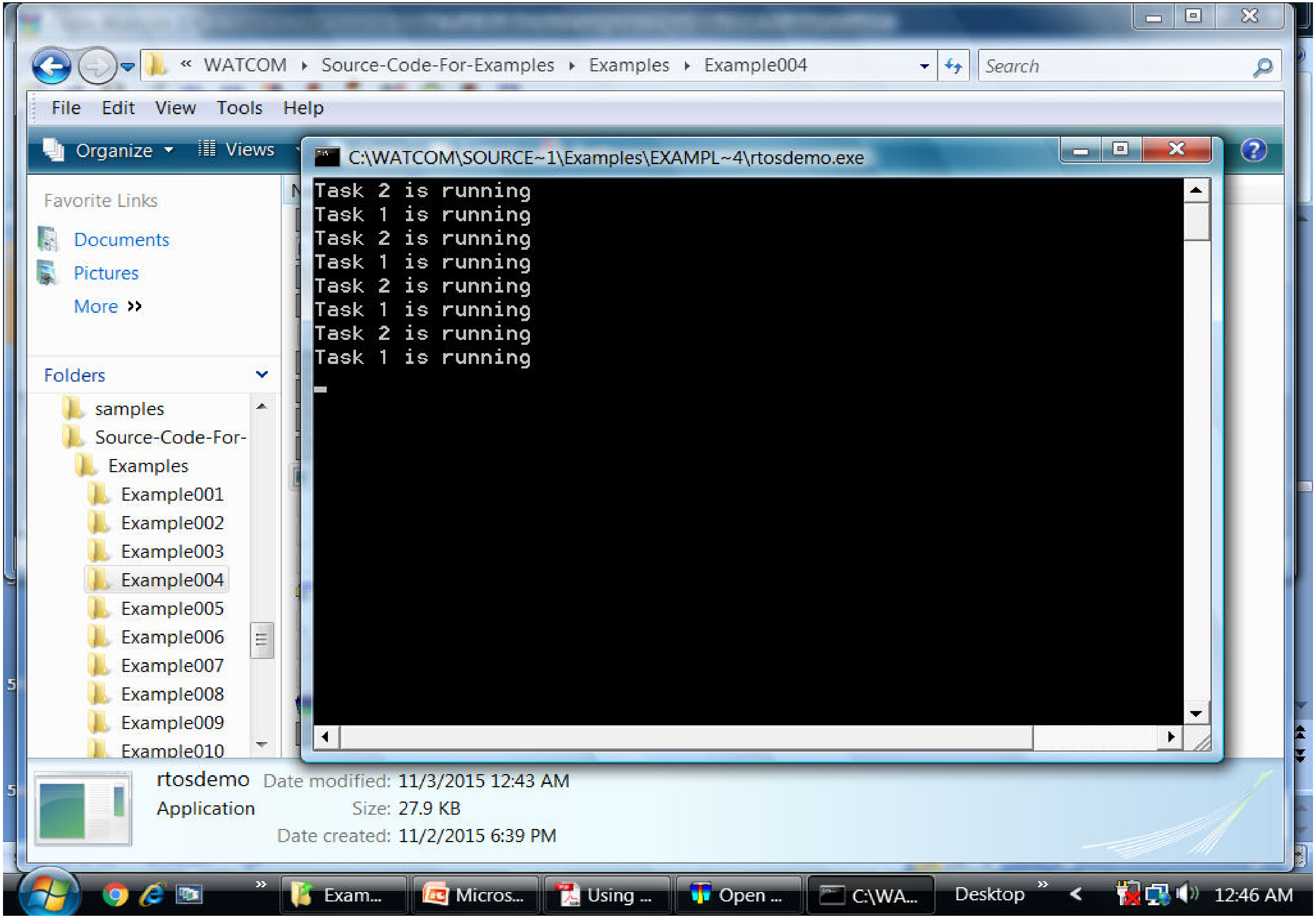
  for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) { }

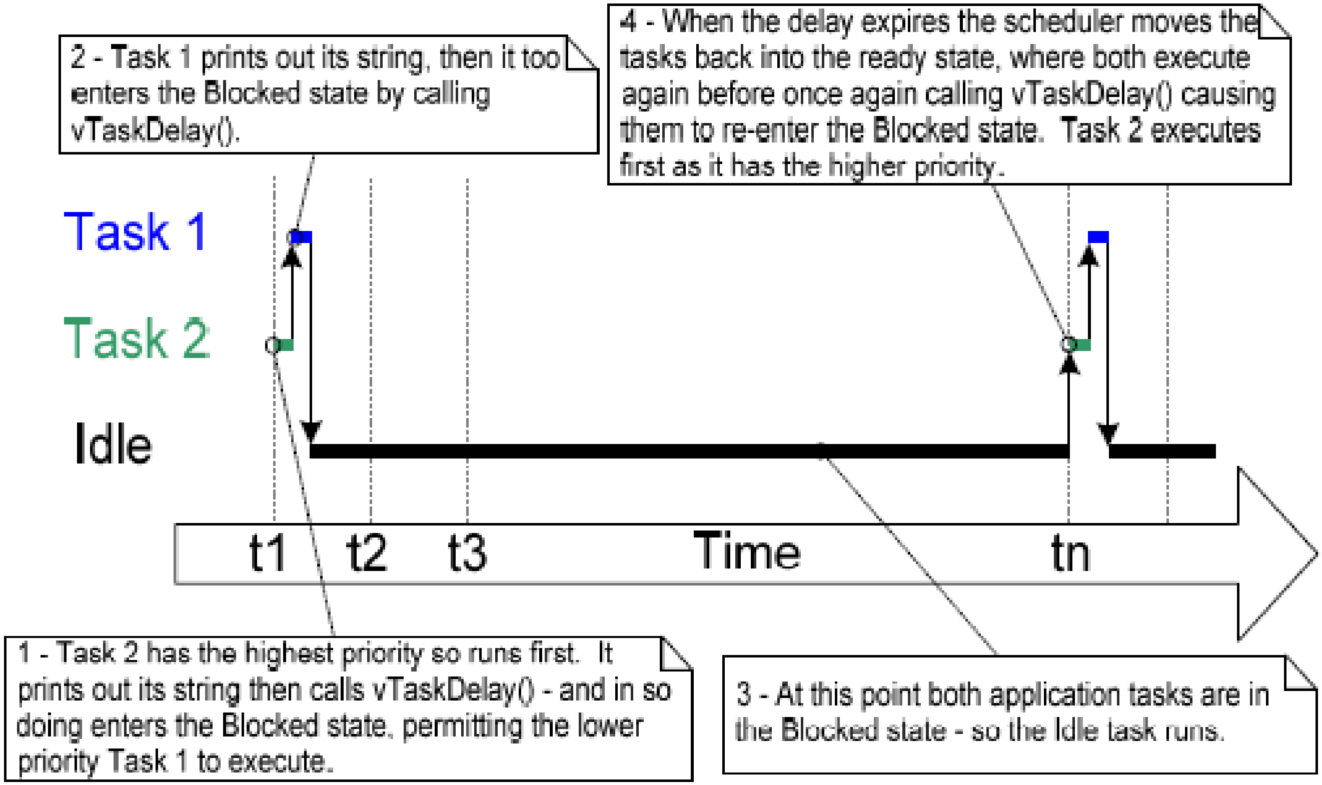
  To

  vTaskDelay(250 / portTICK_RATE_MS);

  // a period of 250ms is being specified.

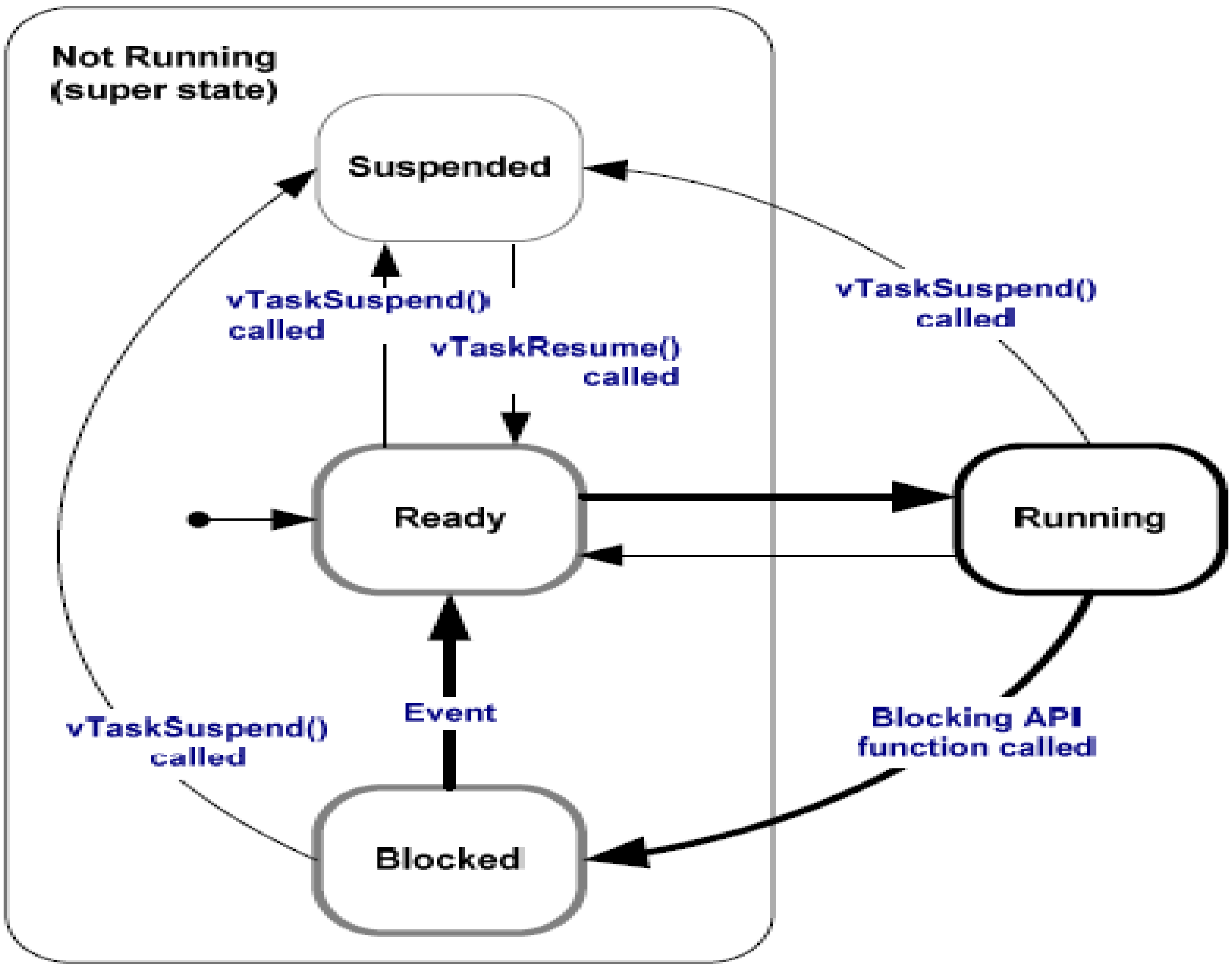
  Although two tasks are being created at different priorities, both will now run.


WATCOM › Source-Code-For-Examples › Examples › Example004
Search
File Edit View Tools Help
Organize Views
C:\WATCOM\SOURCE~1\Examples\EXAMPL~4\rtosdemo.exe
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
Task 1 is running
Favorite Links
Documents
Pictures
More »
Folders
samples
Source-Code-For-
Examples
Example001
Example002
Example003
Example004
Example005
Example006
Example007
Example008
Example009
Example010
rtosdemo Date modified: 11/3/2015 12:43 AM
Application Size: 27.9 KB
Date created: 11/2/2015 6:39 PM
Exam... Micros... Using ... Open ... C:\WA... Desktop 12:46 AM



- 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 no application tasks are able to run and, so, no tasks can be selected to enter the Running state.
  - The idle task will run to ensure there is always at least one task that is able to run.

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



vTaskDelayUntil() API Function

- Parameters to vTaskDelayUntil()
  - specify the exact tick count value at which the calling task should be moved from the Blocked state into the Ready state.
  - Be used when a fixed execution period is required.
    - The time at which the calling task is unblocked is absolute, rather than relative to when the function was called (as vTaskDelay())

```
void vTaskDelayUntil(
        portTickType *pxPreviousWakeTime,
        portTickType xTimeIncrement);
```

vTaskDelayUntil() prototype

- pxPreviousWakeTime
  - Assume that vTaskDelayUtil() is being used to implement a task that executes periodically and with a fixed frequency.
  - Holds the time at which the task left the Blocked state.
  - Be used as a reference point to compute the time at which the task next leaves the Blocked state.
  - The variable pointed by pxPreviousWakeTime is updated automatically, not be modified by application code, other than when the variable is first initialized.

vTaskDelayUntil() prototype

- xTimeIncrement
  - Assume that vTaskDelayUtil() is being used to implement a task that executes periodically and with a fixed frequency – set by xTimeIncrement.
  - Be specified in ‘ticks’. The constant portTICK_RATE_MS can be used to convert ms to ticks.

Example 5 Converting the example tasks to use vTaskDelayUntil()

- Two tasks created in Example 4 are periodic tasks.
- vTaskDelay() does not ensure that the frequency at which they run is fixed,
  - as the time at which the tasks leave the Blocked state is relative to when they call vTaskDelay().

- In void vTaskFunction(void *pvParameters)

  Change

  vTaskDelay(250 / portTICK_RATE_MS);

  // a period of 250ms is being specified.

  To

  vTaskDelayUntil(&xLastWakeTime, (vTaskDelay(250 / portTICK_RATE_MS));

  /*xLastWakeTime is initialized with the current tick count before entering the infinite loop. This is the only time it is written to explicitly. */

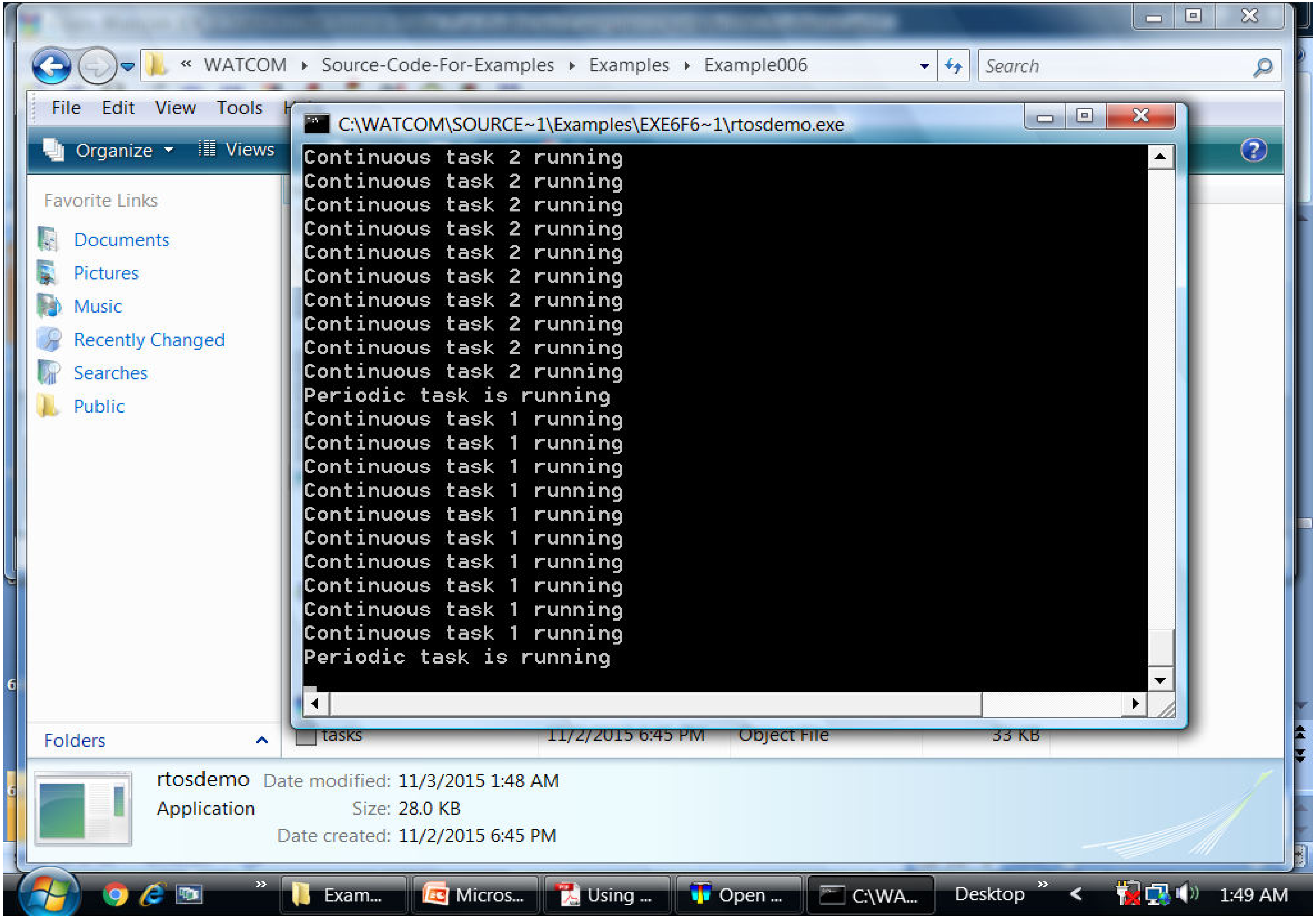
  xLastWakeTime = xTaskGetTickCount();

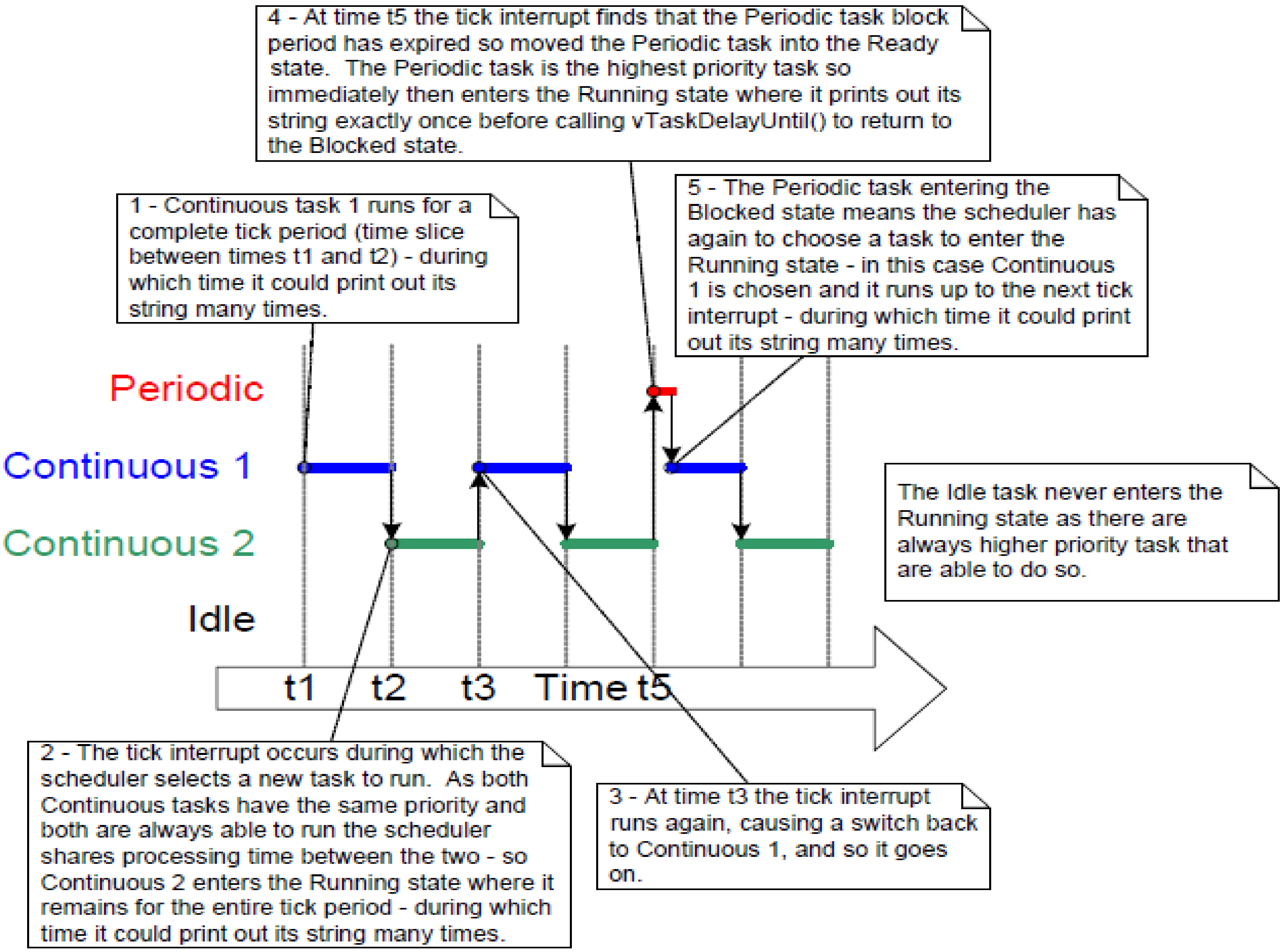
  /*It is then updated within vTaskDelayUntil(); automatically */

Example 6 Combining blocking and non-blocking tasks

- Two tasks are created at priority 1.
  - Always be either the Ready or the Running state as never making any API function calls.
  - Tasks of this nature are called continuous processing tasks they always have work to do.
- A Third task is created at priority 2.
  - Periodically prints out a string by using vTaskDelayUntil() to place itself into the Blocked state between each print iteration.

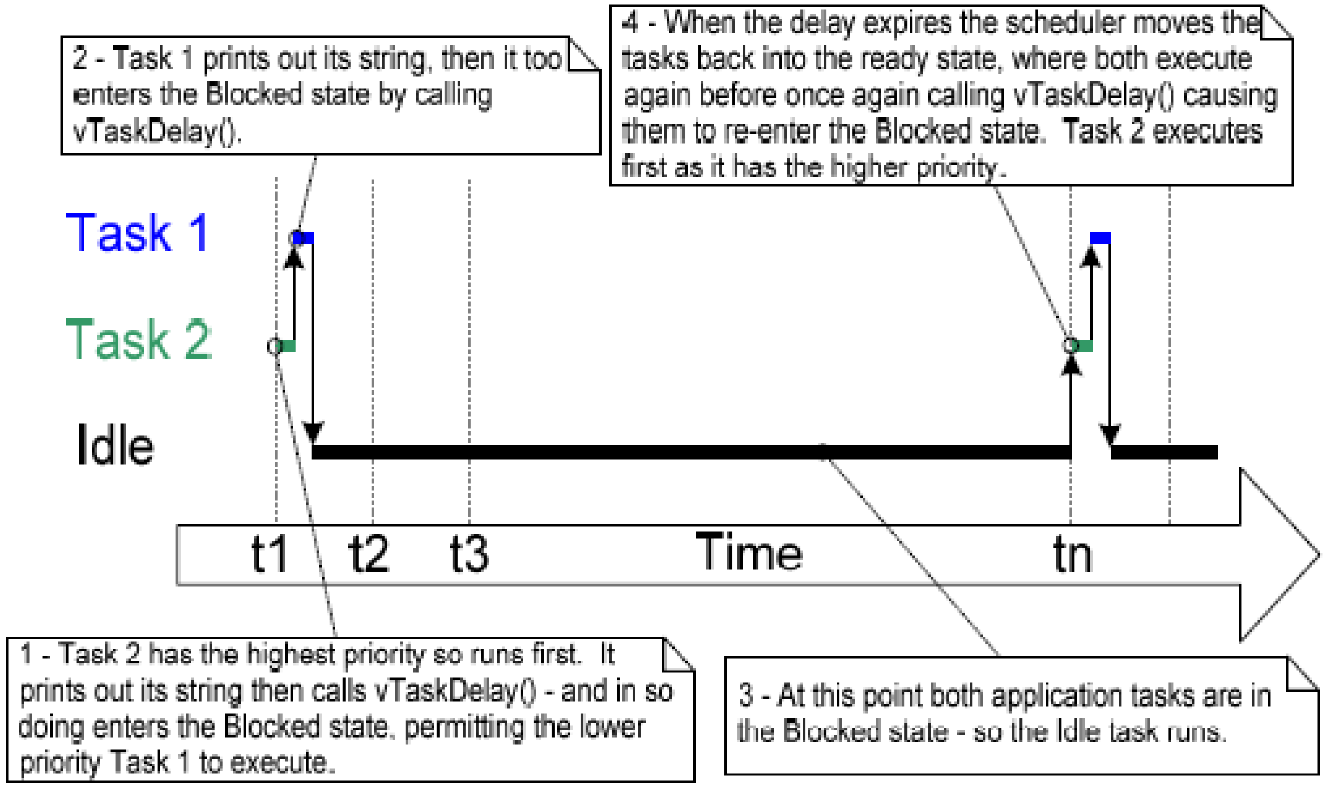
```
void vContinuousProcessingTask(void * pvParameters) {
  char *pcTaskName;
  volatile unsigned long ul;
  pcTaskName = (char *) pvParameters;
  for (;;) { vPrintString(pcTaskName);
        for( ul = 0; ul < 0xffff; ul++ ) { }
  }}

void vPeriodicTask(void * pvParameters) {
  portTickType xLastWakeTime;
  xLastWakeTime = xTaskGetTickCount();
  for (;;){ vPrintString("Periodic task is running\n");
    vTaskDelayUntil(&xLastWakeTime,
                            (10/portTICK_RATE_MS));
  }}
```


WATCOM › Source-Code-For-Examples › Examples › Example006
Search
File Edit View Tools
Organize Views
Favorite Links
Documents
Pictures
Music
Recently Changed
Searches
Public
Folders
C:\WATCOM\SOURCE~1\Examples\EXE6F6~1\rtosdemo.exe
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Periodic task is running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Periodic task is running
tasks 11/2/2015 6:45 PM Object File 33 KB
rtosdemo Date modified: 11/3/2015 1:48 AM
Application Size: 28.0 KB
Date created: 11/2/2015 6:45 PM
Exam... Micros... Using ... Open ... C:\WA... Desktop 1:49 AM


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.
Periodic
Continuous 1
Continuous 2
Idle
The Idle task never enters the Running state as there are always higher priority task that are able to do so.
t1
t2
t3
Time t5
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.

1.7 Idle Task and the Idle task hook

- An idle task is automatically created by the scheduler when *vTaskStartScheduler*() is called.
  - Does very little more than site in a loop
  - Has the lowest possible priority (zero), to ensure it never prevents a higher priority application task from entering the Running state
  - Be transitioned out of the Running state as soon as a higher priority task enters the Ready state



- The Idle task is immediately swapped out to allow Task 2 to execute at the instant Task 2 leaves the Blocked state.
  - Task 2 pre-empts the idle task automatically.

Idle Task Hook Functions

- Add application specific functionality directly into the idle task by the use of an idle hook
  - A function called automatically by the idle task once per iteration of the idle task loop
- Common uses for the Idle task hook
  - Executing low priority, background, or continuous processing
  - Measuring the amount of spare processing capacity
  - Placing the processor into a low power mode, providing an automatic method of saving power whenever no application processing to be performed.

Limitations on the implementation of Idle Task Hook Functions

- Rules idle task hook functions must adhere to
  - Must never attempt to block or suspend.
  - If the application uses vTaskDelete(), the Idle task hook must always return to its caller within a reasonable time period.
    - Idle task is responsible for cleaning up kernel resources after a task has been deleted.
    - If the idle task remains permanently in the Idle hook function, this clean-up cannot occur.
- Idle task hook functions have the name and prototype as

  void vApplicationIdleHook(void);

Example 7 Defining an idle task hook function

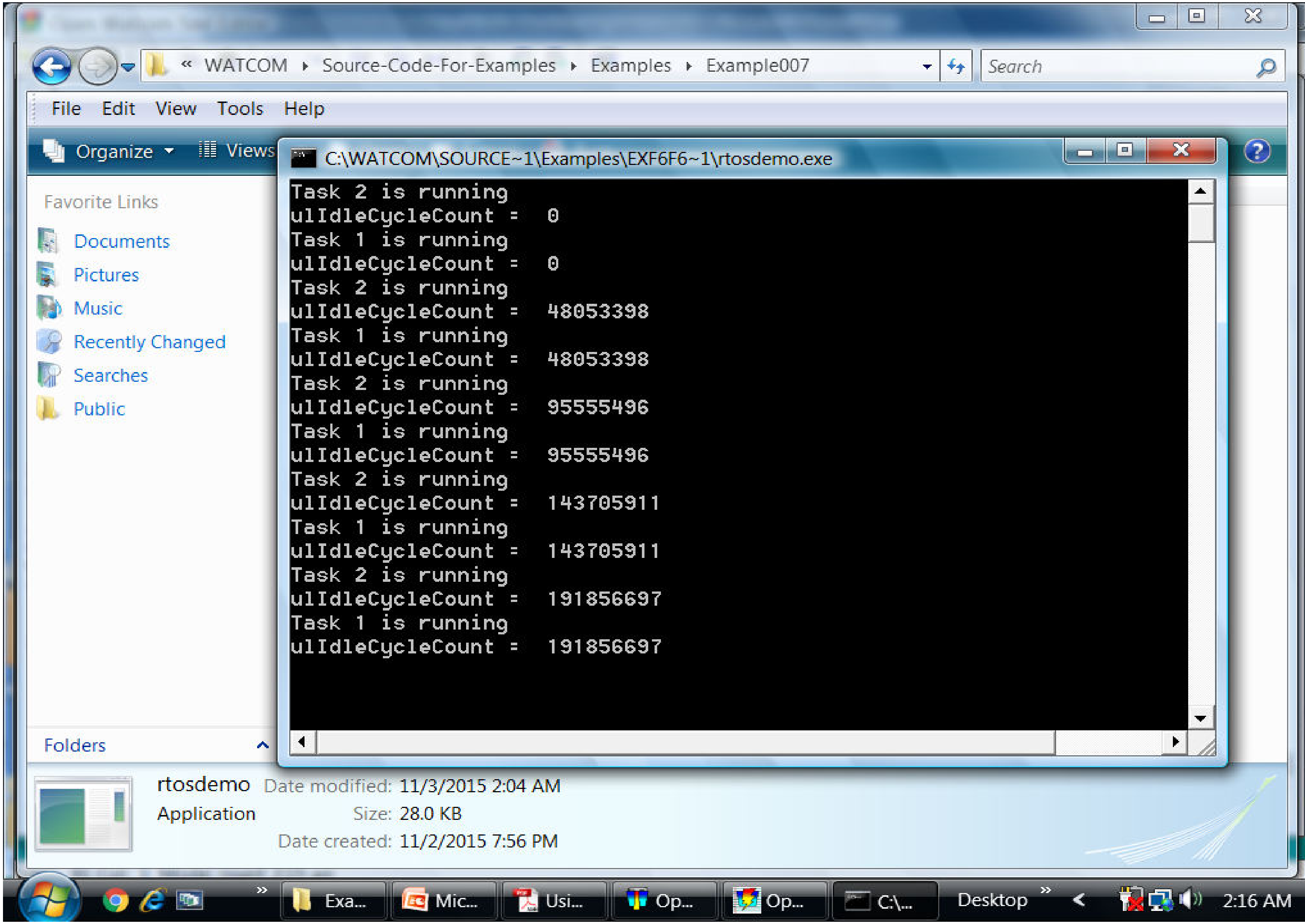
- Set configUSE_IDLE_HOOK to 1
- Add following function

```
unsigned long ulIdleCycleCount = 0UL;
/* must be called this name, take no parameters and return void. */
void vApplicationIdleHook (void) {
  ulIdleCycleCount++;
}
```

- In vTaskFunction(), change

vPrintString(pcTaskName) To

vPrintStringAndNumber(pcTaskName, ulIdleCycleCount);


« WATCOM ▸ Source-Code-For-Examples ▸ Examples ▸ Example007
Search
File Edit View Tools Help
Organize
Views
C:\WATCOM\SOURCE~1\Examples\EXF6F6~1\rtosdemo.exe
Task 2 is running
ulIdleCycleCount = 0
Task 1 is running
ulIdleCycleCount = 0
Task 2 is running
ulIdleCycleCount = 48053398
Task 1 is running
ulIdleCycleCount = 48053398
Task 2 is running
ulIdleCycleCount = 95555496
Task 1 is running
ulIdleCycleCount = 95555496
Task 2 is running
ulIdleCycleCount = 143705911
Task 1 is running
ulIdleCycleCount = 143705911
Task 2 is running
ulIdleCycleCount = 191856697
Task 1 is running
ulIdleCycleCount = 191856697
Favorite Links
Documents
Pictures
Music
Recently Changed
Searches
Public
Folders
rtosdemo Date modified: 11/3/2015 2:04 AM
Application Size: 28.0 KB
Date created: 11/2/2015 7:56 PM
Exa... Mic... Usi... Op... Op... C:\...
Desktop
2:16 AM

1.8 Change the priority of a task

- vTaskPrioritySet(xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority);
  - Be used to change the priority of any task after the scheduler has been started.
  - Available if INCLUDE_vTaskPrioritySet is set 1.
- Two parameters
  - pxTask: Handle of the task whose priority is being modified. A task can change its own priority by passing NULL in place of a valid task handle.
  - uxNewPriority: the priority to be set.

Get the Priority of a Task

- unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);
  - Be used to query the priority of a task
  - Available if INCLUDE_vTaskPriorityGet is set 1
  - pxTask: Handle of the task whose priority is being modified. A task can query its own priority by passing NULL in place of a valid task handle.
  - Returned value: the priority currently assigned to the task being queried

Example 8 Changing task priorities

- Demonstrate the scheduler always selects the highest Ready state task to run
  - by using the vTaskPrioritySet() API function to change the priority of two tasks relative to each other.
- Two tasks are created at two different priorities.
  - Neither task makes any API function calls that cause it to enter the Blocked state,
    - So both are in either Ready or Running state.
    - So the task with highest priority will always be the task selected by the scheduler to be in Running state

Expected Behavior of Example 8

1. Task 1 is created with the highest priority to be guaranteed to run first. Task 1 prints out a couple of strings before raising the priority of Task 2 to above its own priority.
2. Task2 starts to run as it has the highest relative priority.
3. Task 2 prints out a message before setting its own priority back to below that of Task 1.
4. Task 1 is once again the highest priority task, so it starts to run and forcing Task 2 back into the Ready state.

Example 8 code

- Declare a global variable to hold the handle of Task 2.

```
xTaskHandle xTask2Handle;
```

- In main() function, create two tasks

```
xTaskCreate(vTask1, “Task 1”, 240,
                    NULL, 2, NULL);
xTaskCreate(vTask2, “Task 2”, 240,
                    NULL, 1, &xTask2Handle);
```

- Change vTask1 by initialization

```
unsigned portBASE_TYPE uxPriority;
uxPriority = uxTaskPriorityGet(NULL);
```

- And adding to the infinite loop

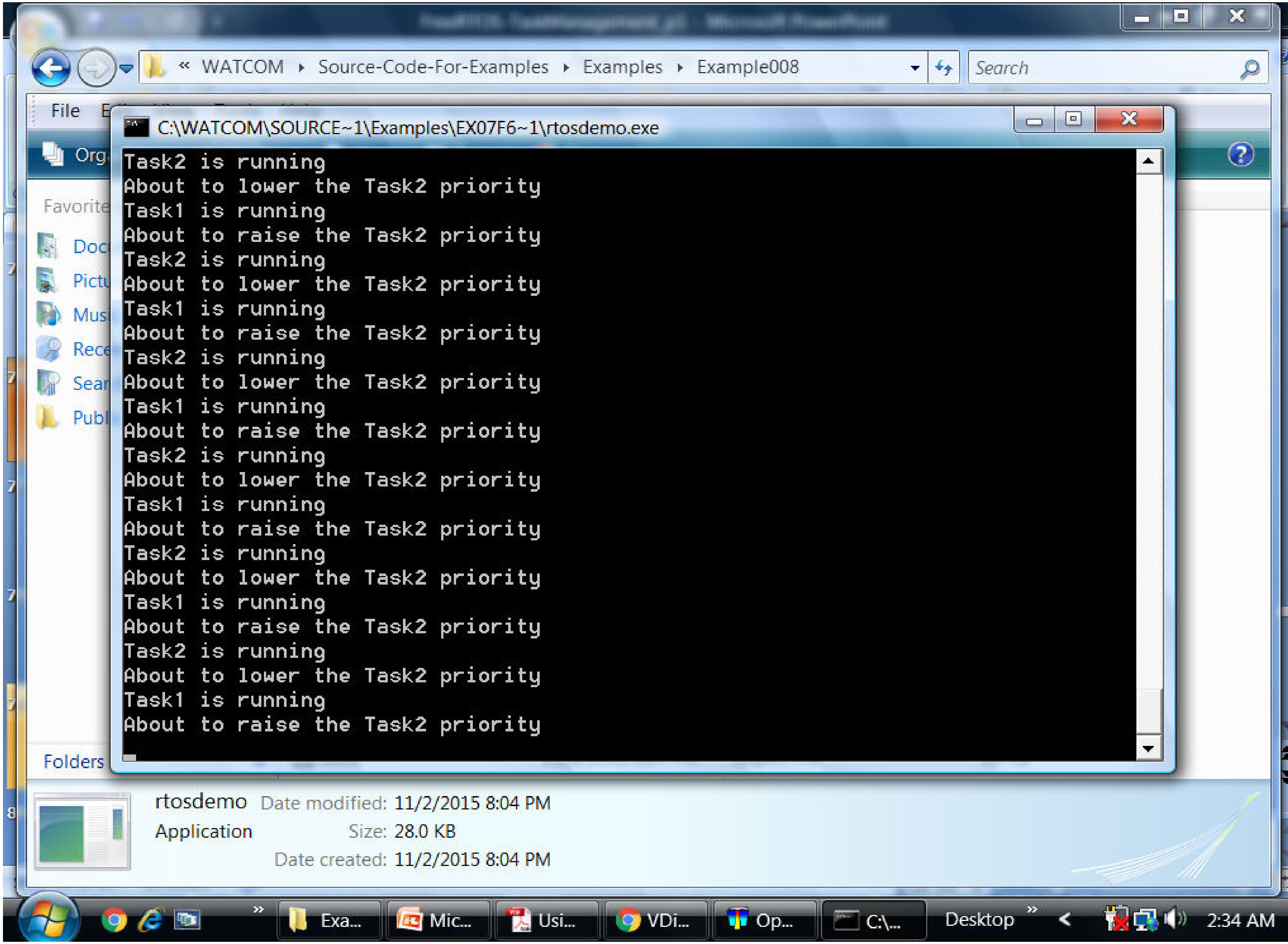
```
vTaskPrioritySet(xTaskHandl1, (uxPriority+1));
```

- Change vTask2 by initialization

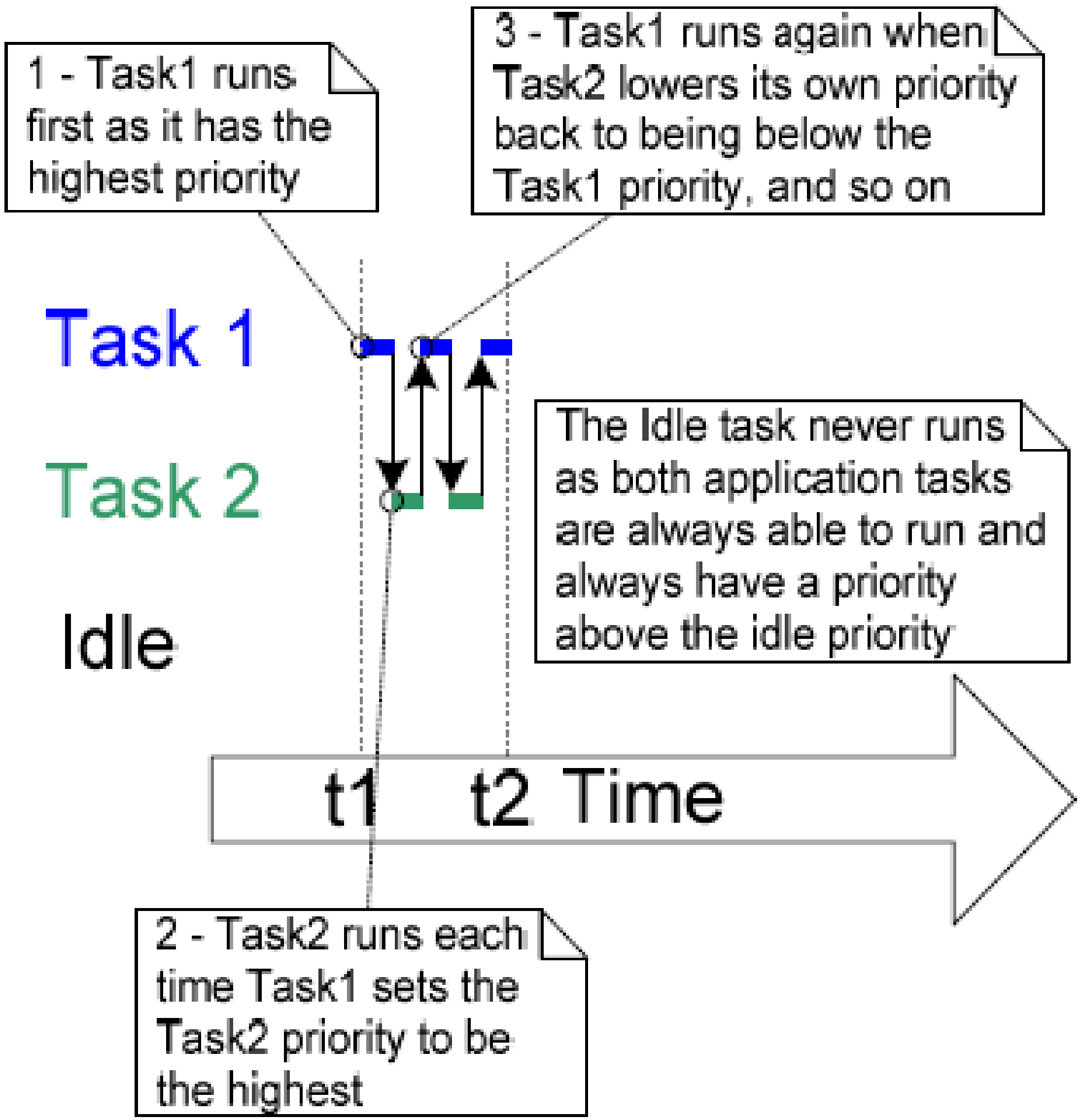
```
unsigned portBASE_TYPE uxPriority;
uxPriority = uxTaskPriorityGet(NULL);
```

- And adding to the infinite loop

```
vTaskPrioritySet(NULL, (uxPriority-2));
```


WATCOM › Source-Code-For-Examples › Examples › Example008
Search
File
C:\WATCOM\SOURCE~1\Examples\EX07F6~1\rtosdemo.exe
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Folders
rtosdemo Date modified: 11/2/2015 8:04 PM
Application Size: 28.0 KB
Date created: 11/2/2015 8:04 PM
Exa... Mic... Usi... VDi... Op... C:\...
Desktop
2:34 AM

Task execution sequence



1.9 Deleting a task

- Deleted tasks no longer exist and cannot enter the Running state again.
- Idle task is responsible to automatically free memory allocated by kernel to tasks that have been deleted.
  - Remember if applications use vTaskDelete(), **do not** completely starve the idle task of all processing time.
- Note: any memory or other resource that the application task allocates itself must by be freed explicitly by the application code.

vTaskDelete() API function

- Function prototype

  void vTaskDelete(xTaskHandle pxTaskToDelete);

  – pxTaskToDelete: Handle of the task that is to be deleted. A task can delete itself by passing NULL in place of valid task handle.
- Available only when INCLUDE_vTaskDelete set 1

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", 1000, NULL, 1, NULL );
    /* The task is created at priority 1 ______^. */

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

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

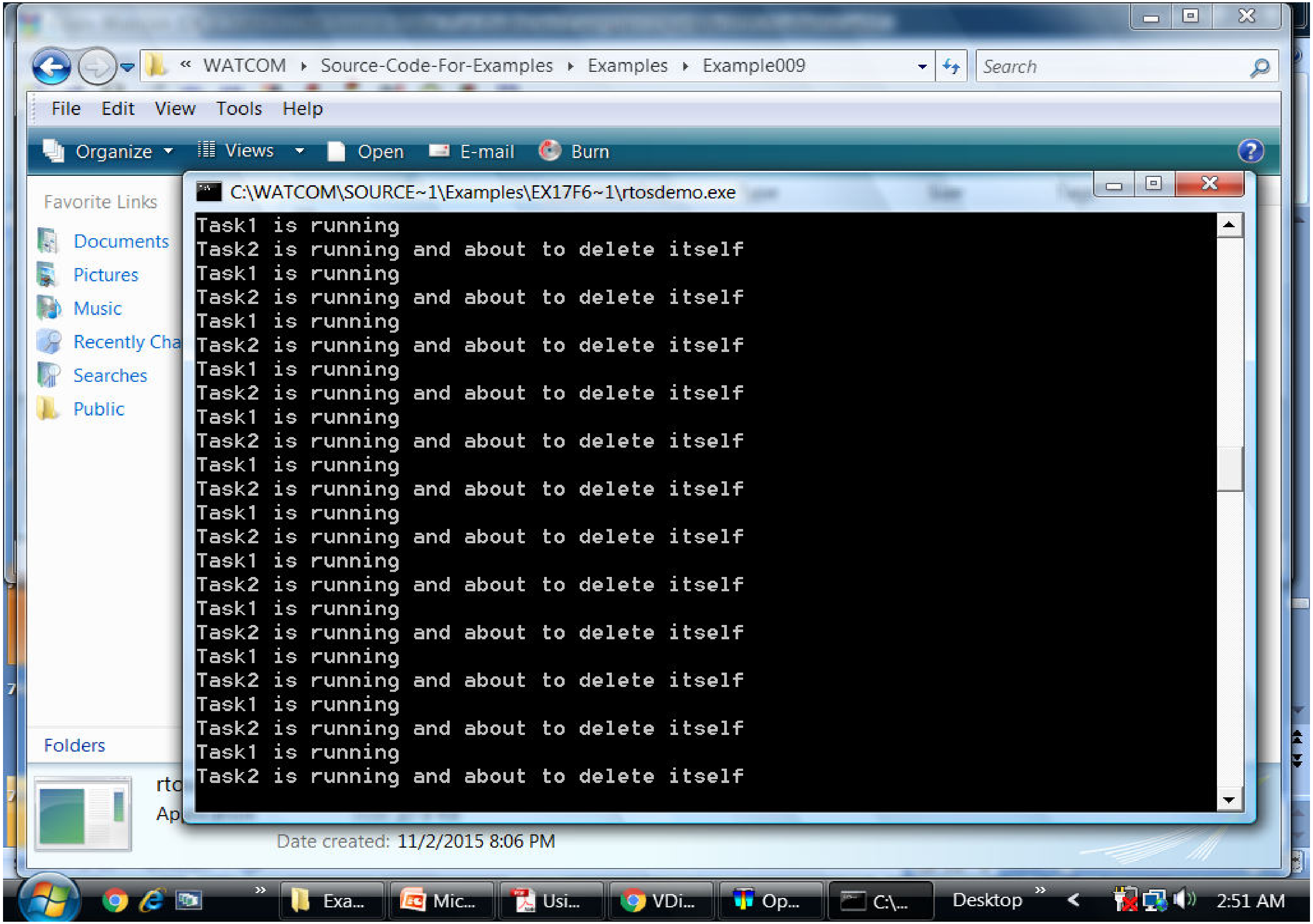
Listing 26 The implementation of main() for Example 9

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

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task1 is running\r\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", 1000, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter _____^^^^^^^^^^^^^ */

        /* Task2 has/had the higher priority, so for Task1 to reach here Task2
        must have already executed and deleted itself.  Delay for 100
        milliseconds. */
        vTaskDelay( xDelay100ms );
    }
}
```

Listing 27 The implementation of Task 1 for Example 9

```c
void vTask2( void *pvParameters )
{
    /* Task2 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\r\n" );
    vTaskDelete( xTask2Handle );
}
```

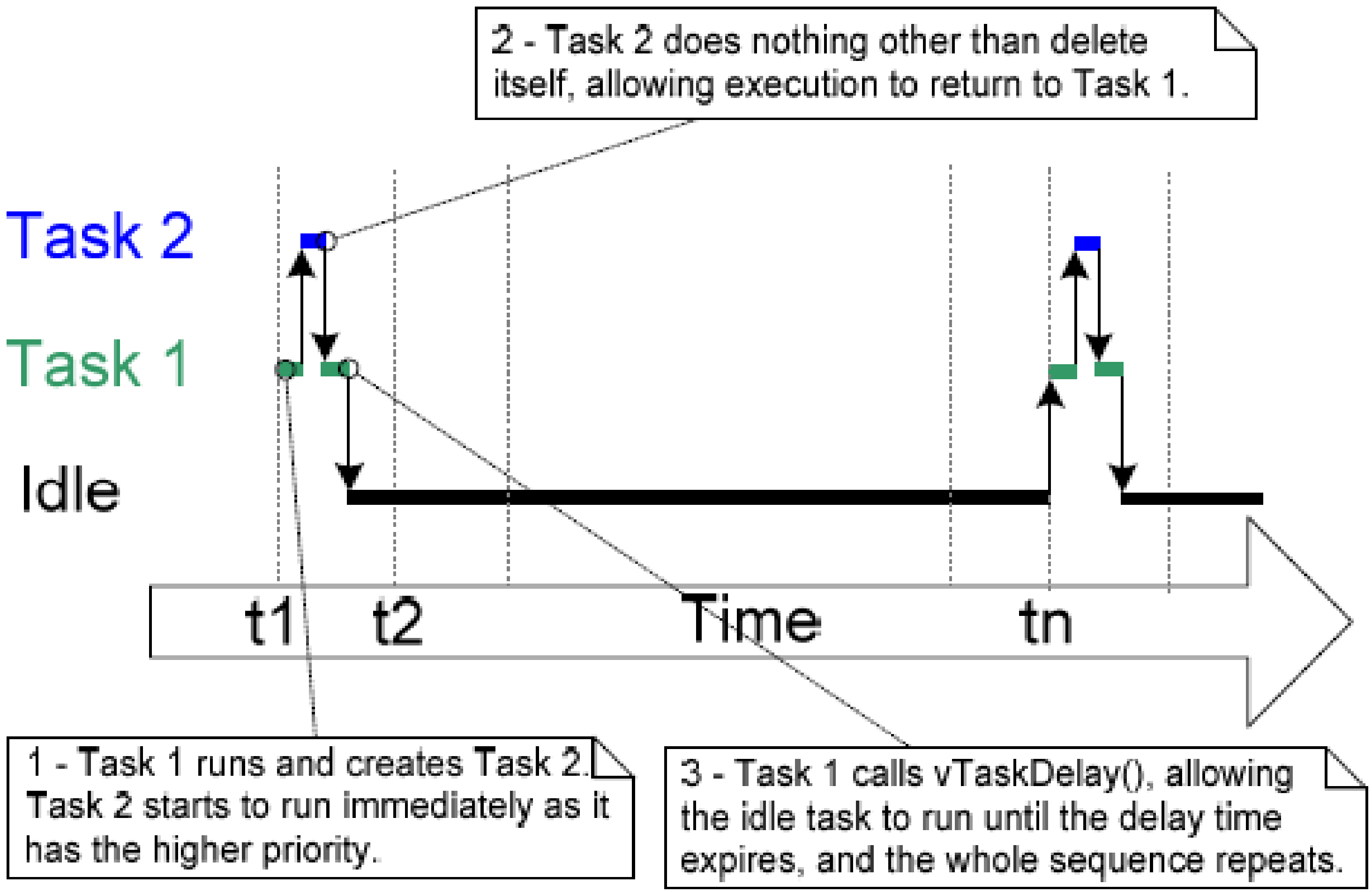
Listing 28 The implementation of Task 2 for Example 9


WATCOM › Source-Code-For-Examples › Examples › Example009
Search
File Edit View Tools Help
Organize Views Open E-mail Burn
Favorite Links
Documents
Pictures
Music
Recently Cha
Searches
Public
Folders
C:\WATCOM\SOURCE~1\Examples\EX17F6~1\rtosdemo.exe
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Date created: 11/2/2015 8:06 PM
Exa... Mic... Usi... VDi... Op... C:\...
Desktop
2:51 AM

Example 9 Deleting tasks (Behavior)

1. Task 1 is created by main() with priority 1. When it runs, it creates Task 2 at priority 2. Task 2 as the highest priority task starts to execute immediately.
2. Task 2 does nothing but delete itself by passing NULL or its own task handle.
3. When Task 2 has been deleted, Task 1 is again the highest priority task, so continues executing – at which point it calls vTaskDelay() to block for a short period.
4. The idle task executes while Task 1 is in the blocked state and frees the memory that was allocated to the now deleted Task 2.
5. When Task 1 leaves the blocked state it again becomes the highest priority Ready state task and preempts the Idle task. Then, start from Step1 again.

Execution sequence



Summary –
1. Prioritized pre-emptive scheduling

- Examples illustrate how and when FreeRTOS selects which task should be in the Running state.
  - 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.

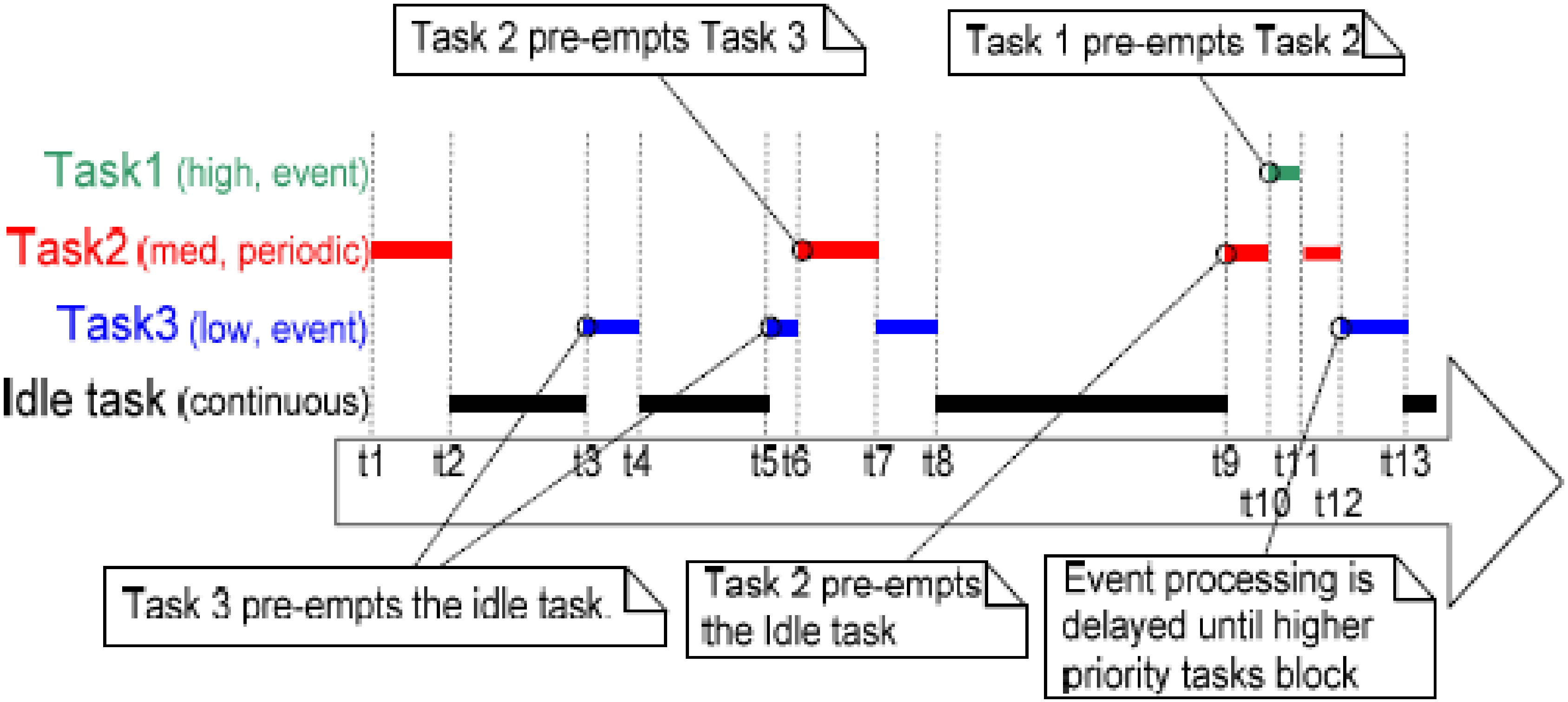
Fixed priority Pre-emptive scheduling

- Fixed priority
  - Each task is assigned a priority that is not altered by the kernel itself (only tasks can change priorities)
- Pre-emptive
  - 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.

Tasks in the Blocked state

- Tasks can wait in the Blocked state for an **event** and are automatically moved back to the Ready state when the **event** occurs.
- Temporal events
  - Occur at a particular time, e.g. a block time expires.
  - Generally be used to implement periodic or timeout behavior.
- Synchronization events
  - Occur when a task or ISR sends info to a queue or to one of the many types of semaphore.
  - Generally be used to signal asynchronous activity, such as data arriving at a peripheral.

Execution pattern with pre-emption points highlighted



Event for Task 1 occur at : t11
Task2 is released at : t1, t6, t9
Event for Task 3 occur at: t3, t5, between t9 and t12

- Idle task
  - The idle task is running at the lowest priority, so get pre-empted every time a higher priority task enters the Ready state
    - E.g., at times t3,t5,t9.

- Task 3
  - An event-driven task
    - Execute with a low priority, but above the Idle task priority.
  - It spends most of its time in the Blocked state waiting for the event of interest, transitioning from Blocked to Ready state each time the event occurs.
    - All FreeRTOS inter-task communication mechanisms (queues, semaphores, etc.) can be used to signal events and unblock tasks in this way.
  - Event occur at t3, t5, and also between t9 and t12.
    - The events occurring at t3 and t5 are processed immediately as it is the highest priority task that is able to run.
    - The event occurring somewhere between t9 and t12 is not processed as until t12 as until then Task 1 and 2 are still running. They enter Blocked state at t12, making Task 3 the highest priority Ready state task.

- Task 2
  - A periodic task that executes at a priority above Task 3, but below Task1. The period interval means Task 2 wants to execute at t1, t6 and t9.
    - At t6, Task 3 is in Running state, but task 2 has the higher relative priority so preempts Task 3 and start to run immediately.
    - At t7, Task 2 completes its processing and reenters the Blocked state, at which point Task 3 can re-enter the Running state to complete its processing.
    - At t8, Task 3 blocks itself.

- Task 1
  - Also an event-driven task.
  - Execute with the highest priority of all, so can preempt any other task in the system.
  - The only Task 1 event shown occurs at t10, at which time Task 1 pre-empts Task 2.
  - Only after Task 1 has re-entered the Blocked at t11, Task 2 can complete its processing.

2. Selecting Task Priorities

- Task that implement hard real-time functions are assigned priorities above those that implement soft real-time functions.
- Must also take **execution times** and **processor utilization** into account to ensure the entire application will never miss a hard real-time deadline.
  - Rate monotonic scheduling

Rate monotonic scheduling (RMS)

- A common priority assignment technique which assigns each task a **unique** priority according to tasks periodic execution rate.
  - Highest priority is assigned to the task that has the highest frequency of periodic execution.
  - Lowest priority is assigned to the task that has the lowest frequency of periodic execution.
  - Can maximize the schedulability of the entire application.
  - But runtime variations, and the fact that not all tasks are in any way periodic, make absolute calculations a complex process.

3. Co-operative scheduling (1)

- In a pure co-operative scheduler, a context switch occur only when
  - the Running state task enters the Blocked state
  - Or, the Running state task explicitly calls taskYIELD().
- Tasks will never be pre-empted and tasks of equal priority will not automatically share processing time.
  - Results in a less responsive system.