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

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.

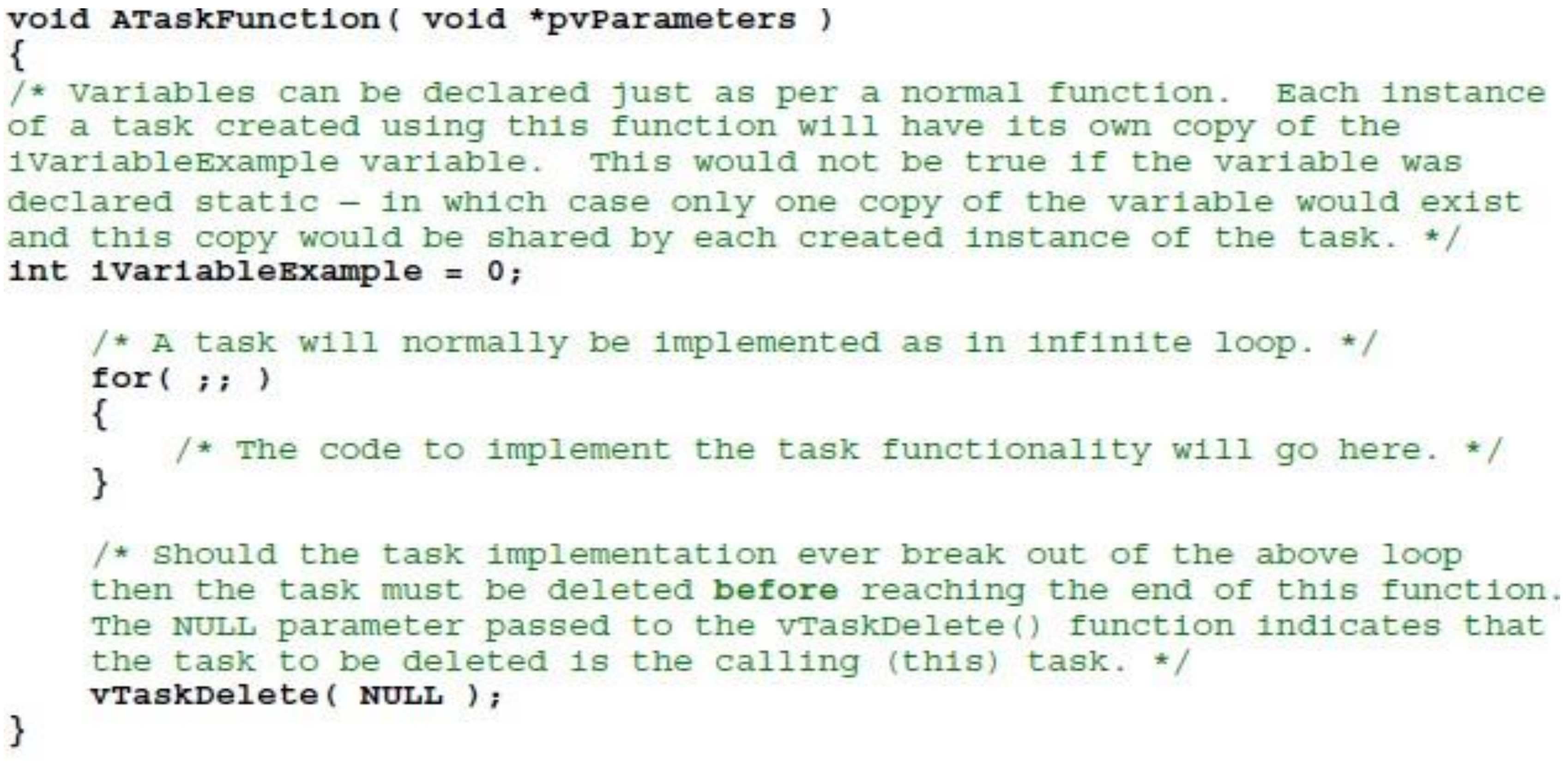
Task functions

- 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 );
}
```

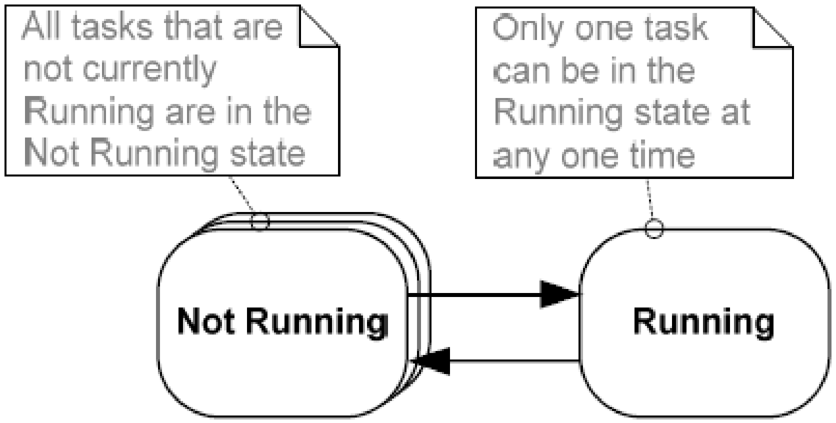
Listing 2 The structure of a typical task function

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.

Top level task states

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


All tasks that are not currently Running are in the Not Running state
Only one task can be in the Running state at any one time
Not Running
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.

Creating Tasks

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

```
BaseType_t  xTaskCreate(TaskFunction_t pvTaskCode,
                        const char * const pcName,
      const configSTACK_DEPTH_TYPE uxStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *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.

- uxStackDepth
  - 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
  - pdPASS : 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.

vTask1

```
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. */
  Serial.print( 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. */
  }
 }
}
```

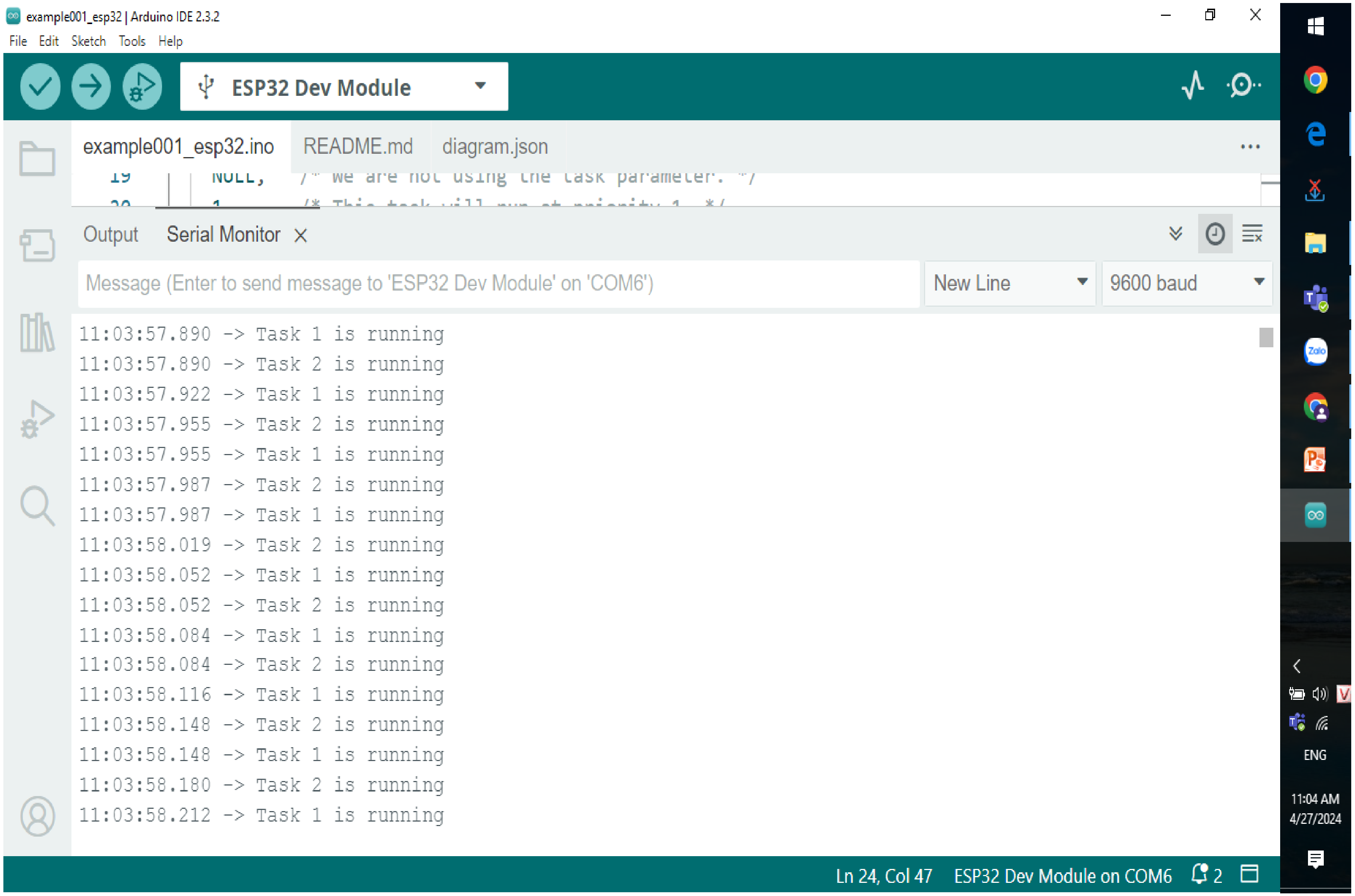
vTask2

```
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. */
    Serial.print( 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. */
    }
  }
}
```

setup()

```
void setup( void )
{
  Serial.begin(9600);
  /* Create one of the two tasks. */
   xTaskCreate(  vTask1,   /* Pointer to the function that implements the task. */
    "Task 1", /* Text name for the task.  This is to facilitate debugging only. */
    2000,    /* 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 using the task handle. */

  /* Create the other task in exactly the same way. */
  xTaskCreate( vTask2, "Task 2", 2000, NULL, 1, NULL );

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

}
```

example001 - Serial Monitor



```
11:03:57.890 -> Task 1 is running
11:03:57.890 -> Task 2 is running
11:03:57.922 -> Task 1 is running
11:03:57.955 -> Task 2 is running
11:03:57.955 -> Task 1 is running
11:03:57.987 -> Task 2 is running
11:03:57.987 -> Task 1 is running
11:03:58.019 -> Task 2 is running
11:03:58.052 -> Task 1 is running
11:03:58.052 -> Task 2 is running
11:03:58.084 -> Task 1 is running
11:03:58.084 -> Task 2 is running
11:03:58.116 -> Task 1 is running
11:03:58.148 -> Task 2 is running
11:03:58.148 -> Task 1 is running
11:03:58.180 -> Task 2 is running
11:03:58.212 -> Task 1 is running
```

Execution pattern of two Example 1 tasks

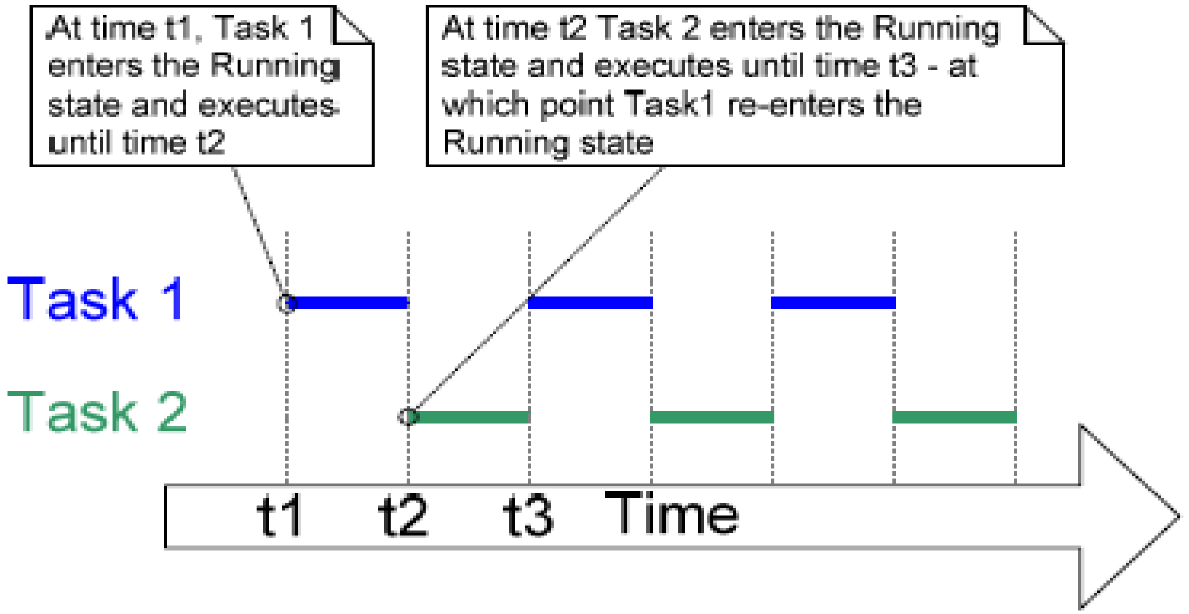

At time t1, Task 1 enters the Running state and executes until time t2
At time t2 Task 2 enters the Running state and executes until time t3 - at which point Task1 re-enters the Running state
Task 1
Task 2
t1
t2
t3
Time

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

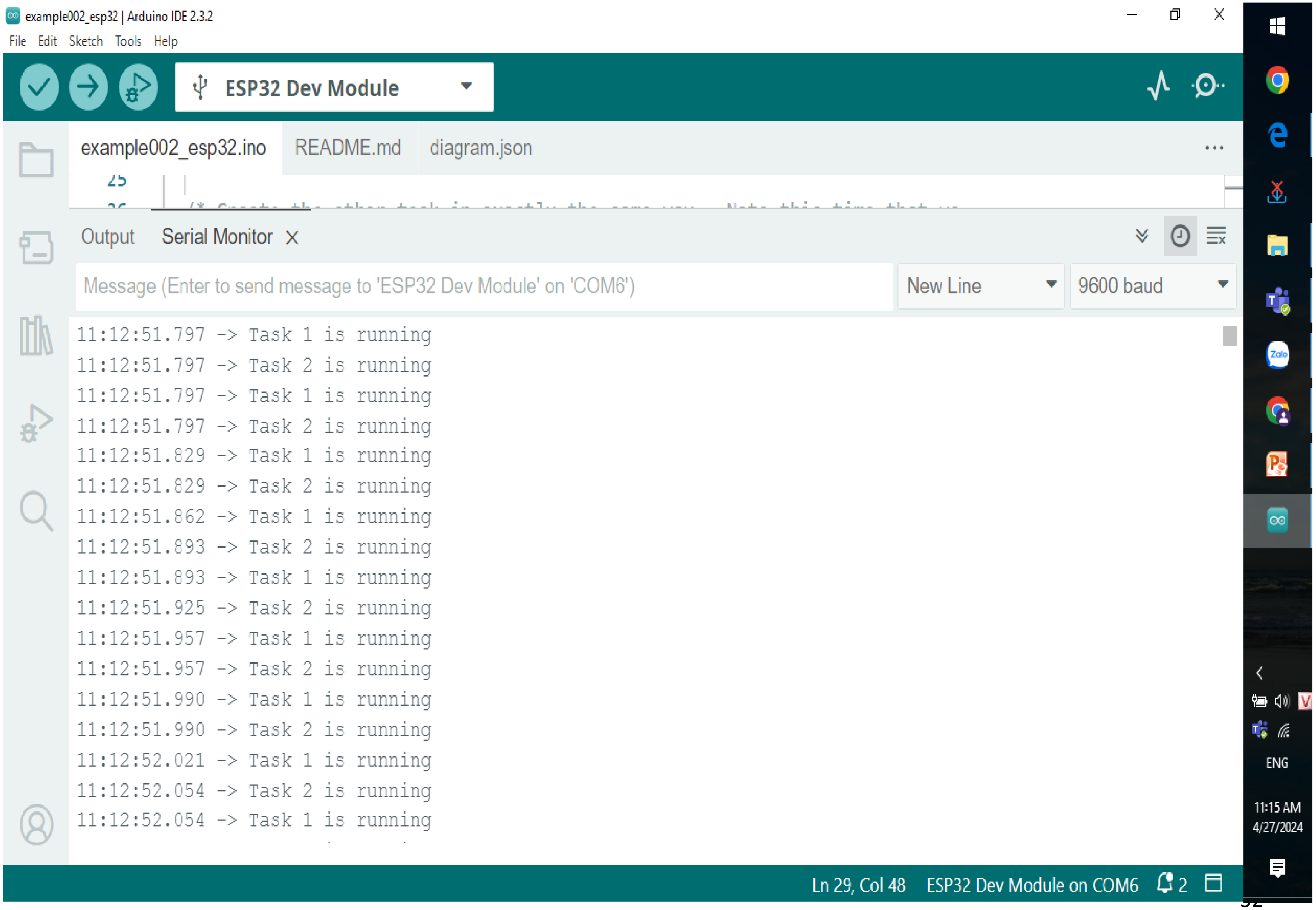
- The two tasks in Example 1 are almost identical, the only difference between them being the text string they print out.
  - Remove such duplication by creating two instances of a single task implementation *vTaskFunction*.
    - Each created instance will execute independently under the control of 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. */
    Serial.print( pcTaskName );

    /* Delay for a period. */
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ ) {   }
  }
}
```

```
const char *pcTextForTask1 = "Task 1 is running\r\n";
const char *pcTextForTask2 = "Task 2 is running\r\n";

void setup( void )
{
  Serial.begin(9600);
  xTaskCreate(  vTaskFunction,      /* Pointer to the function that implements the task. */
    "Task 1",       /* Text name for the task.  This is to facilitate debugging only. */
    2000,          /* Stack depth - most small microcontrollers will use much less stack than this. */
    (void*)pcTextForTask1,  /* Pass the text to be printed in as the task parameter. */
    1,            /* This task will run at priority 1. */
    NULL );        /* We are not using the task handle. */

xTaskCreate( vTaskFunction, "Task 2", 2000, (void*)pcTextForTask2, 1, NULL );

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

for( ;; );

}
```

example002 - Serial Monitor



```
11:12:51.797 -> Task 1 is running
11:12:51.797 -> Task 2 is running
11:12:51.797 -> Task 1 is running
11:12:51.797 -> Task 2 is running
11:12:51.829 -> Task 1 is running
11:12:51.829 -> Task 2 is running
11:12:51.862 -> Task 1 is running
11:12:51.893 -> Task 2 is running
11:12:51.893 -> Task 1 is running
11:12:51.925 -> Task 2 is running
11:12:51.957 -> Task 1 is running
11:12:51.957 -> Task 2 is running
11:12:51.990 -> Task 1 is running
11:12:51.990 -> Task 2 is running
11:12:52.021 -> Task 1 is running
11:12:52.054 -> Task 2 is running
11:12:52.054 -> Task 1 is running
```

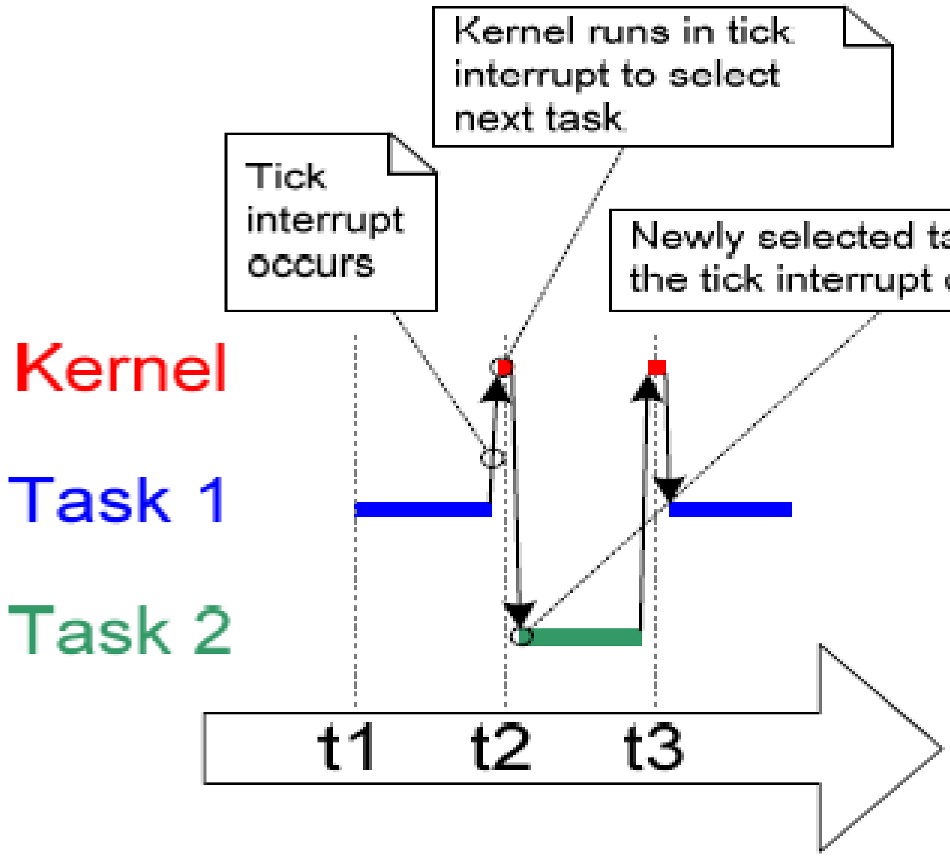
1.5 Task priorities

- 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_PRIORITIES 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_PERIOD_MS
  - Convert time delays from milliseconds into the number of tick interrupts.


Kernel runs in tick interrupt to select next task
Tick interrupt occurs
Newly selected task runs when the tick interrupt completes
Kernel
Task 1
Task 2
t1
t2
t3

- 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

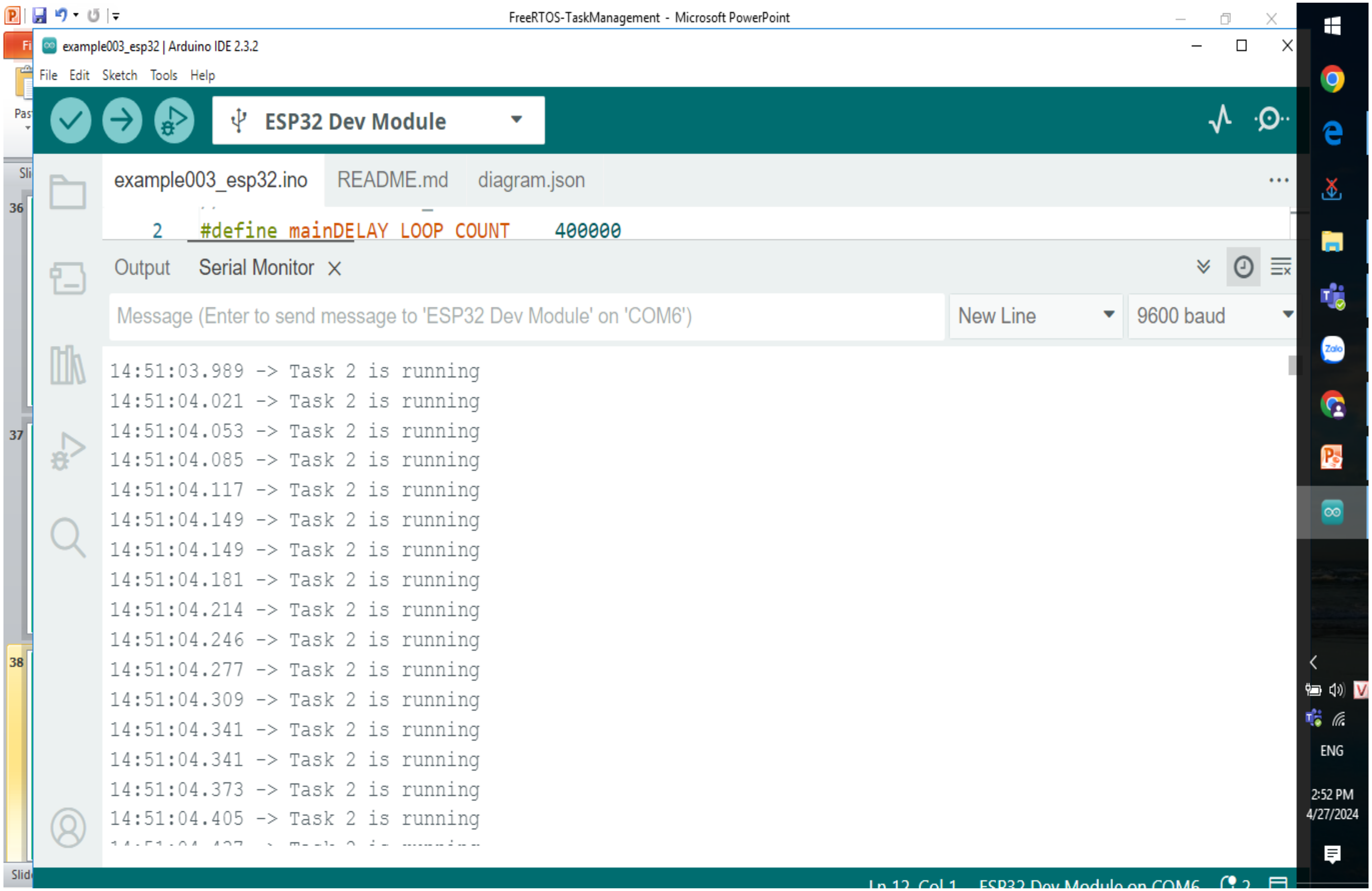
- In setup() function, change

  xTaskCreate(vTaskFunction, "Task 2", 200, (void*)pcTextForTask2, 1, NULL);

  to

  xTaskCreate(vTaskFunction, "Task 2", 200, (void*)pcTextForTask2, 2, NULL);

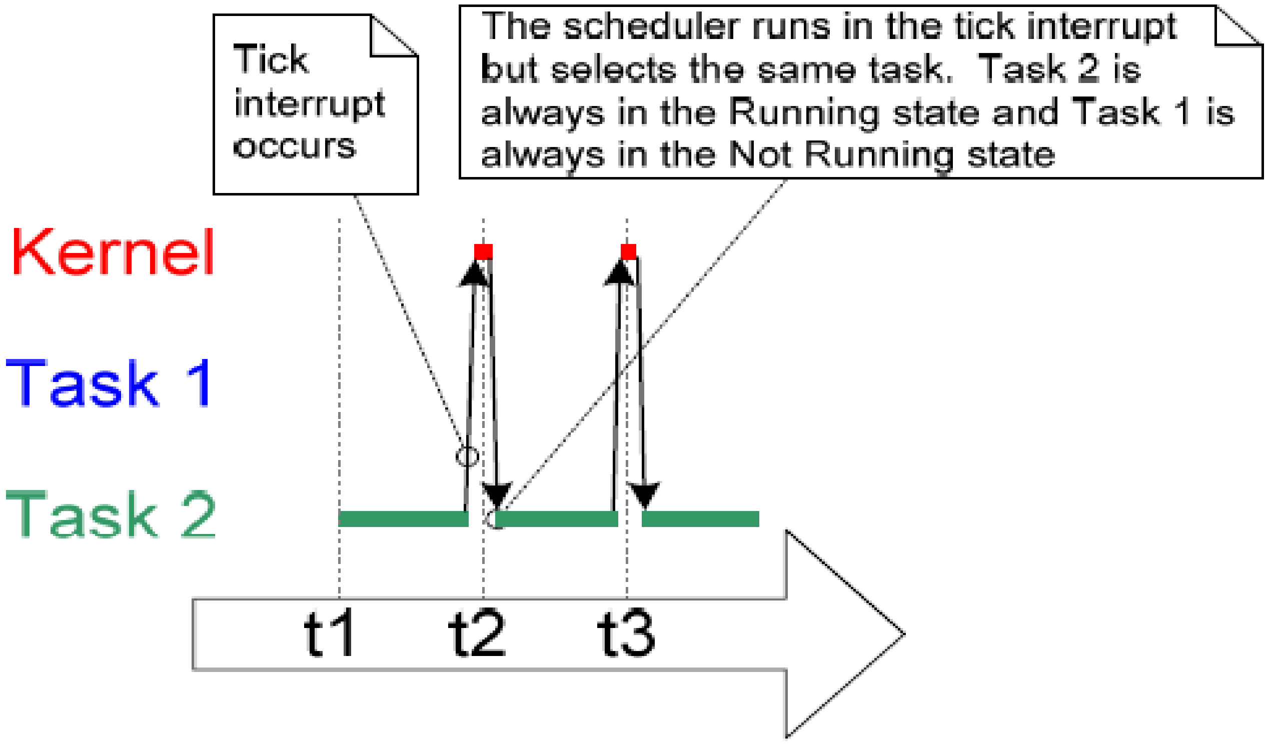
example003 - Serial Monitor



FreeRTOS-TaskManagement - Microsoft PowerPoint
example003_esp32 | Arduino IDE 2.3.2
File Edit Sketch Tools Help
ESP32 Dev Module
example003_esp32.ino
README.md
diagram.json
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM6')
New Line
9600 baud
Ln 12, Col 1 ESP32 Dev Module on COM6
2:52 PM
4/27/2024

```
#define mainDELAY_LOOP_COUNT    400000
```

```
14:51:03.989 -> Task 2 is running
14:51:04.021 -> Task 2 is running
14:51:04.053 -> Task 2 is running
14:51:04.085 -> Task 2 is running
14:51:04.117 -> Task 2 is running
14:51:04.149 -> Task 2 is running
14:51:04.149 -> Task 2 is running
14:51:04.181 -> Task 2 is running
14:51:04.214 -> Task 2 is running
14:51:04.246 -> Task 2 is running
14:51:04.277 -> Task 2 is running
14:51:04.309 -> Task 2 is running
14:51:04.341 -> Task 2 is running
14:51:04.341 -> Task 2 is running
14:51:04.373 -> Task 2 is running
14:51:04.405 -> Task 2 is running
```


Tick interrupt occurs
The scheduler runs in the tick interrupt but selects the same task. Task 2 is always in the Running state and Task 1 is always in the Not Running state
Kernel
Task 1
Task 2
t1
t2
t3

- 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.

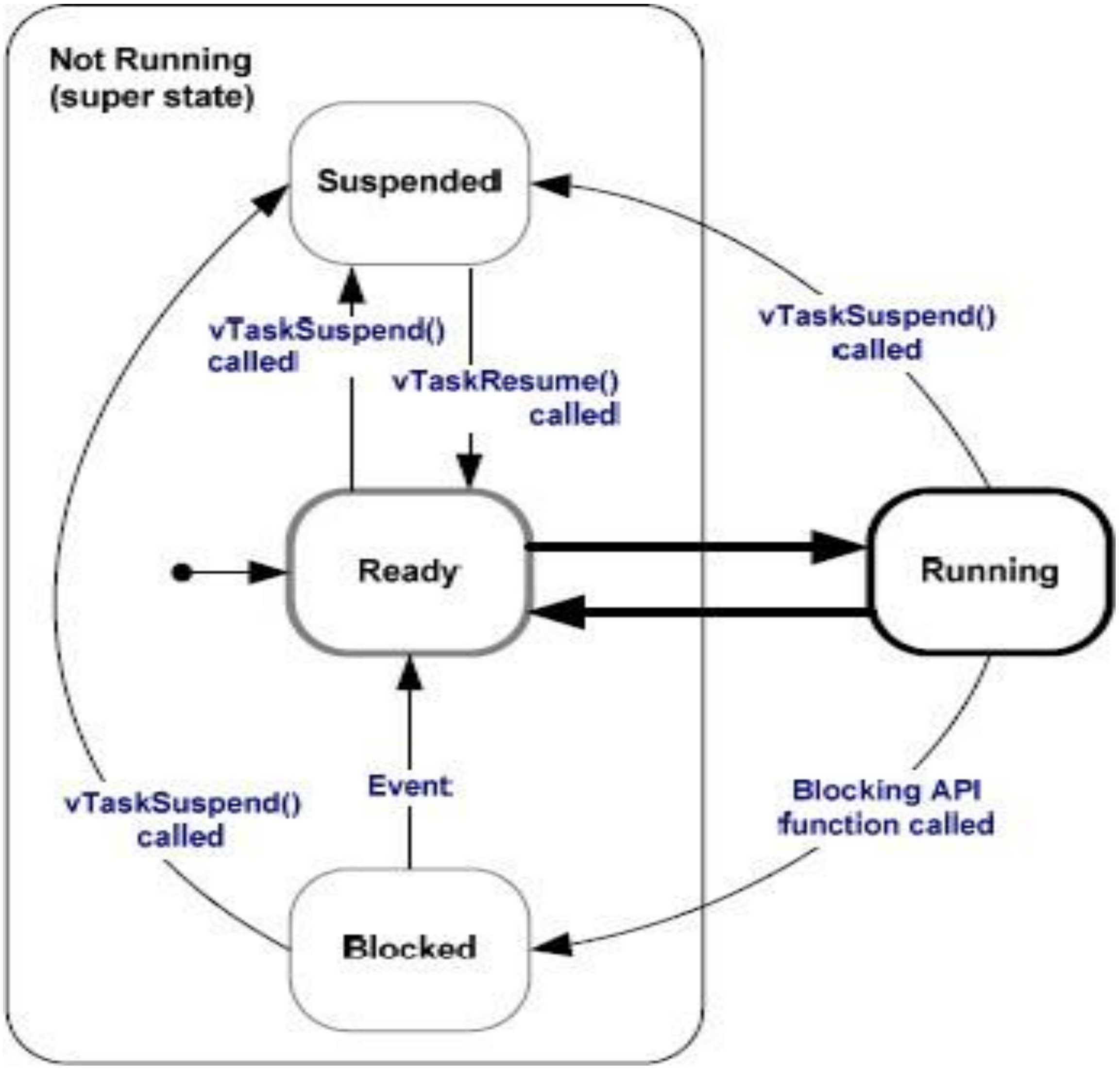
‘Continuous processing’ task

- 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

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.

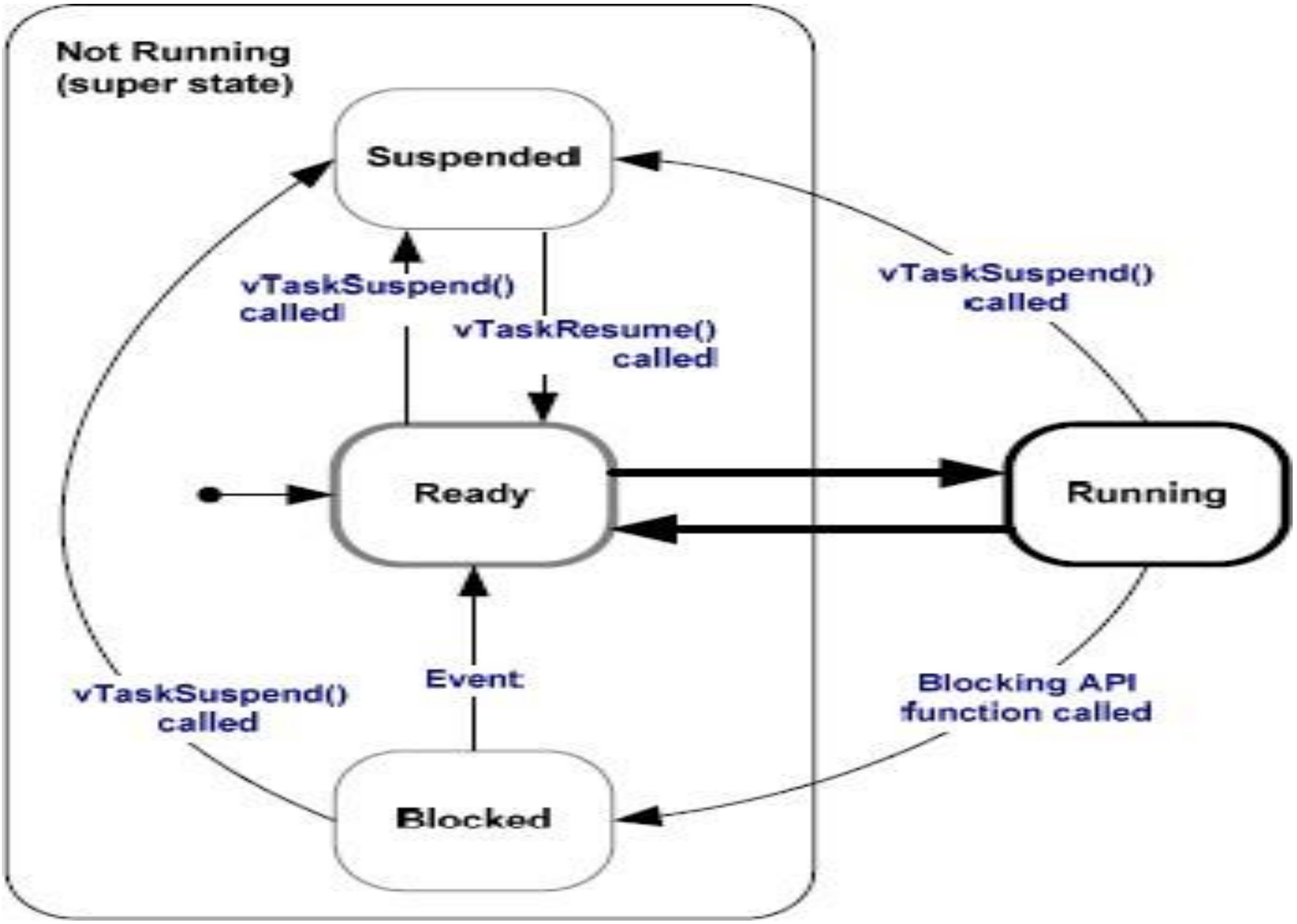
Suspended state

- 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



Example004 - 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( const TickType_t 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 004

- From example 003, in

  void vTaskFunction(void *pvParameters)

  Change a NULL loop

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

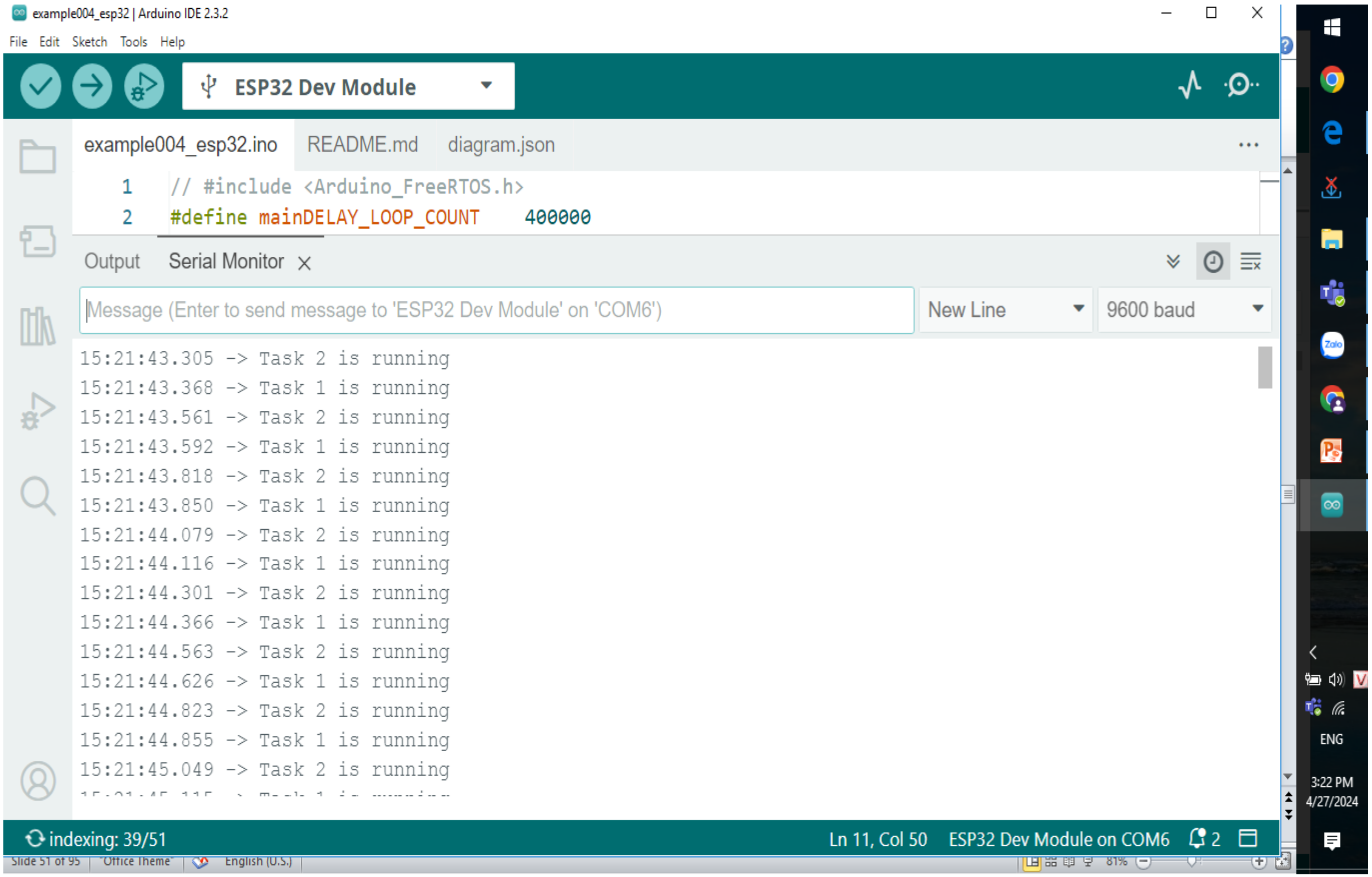
  To

  vTaskDelay(250 / portTICK_PERIOD_MS);

  // a period of 250ms is being specified.

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

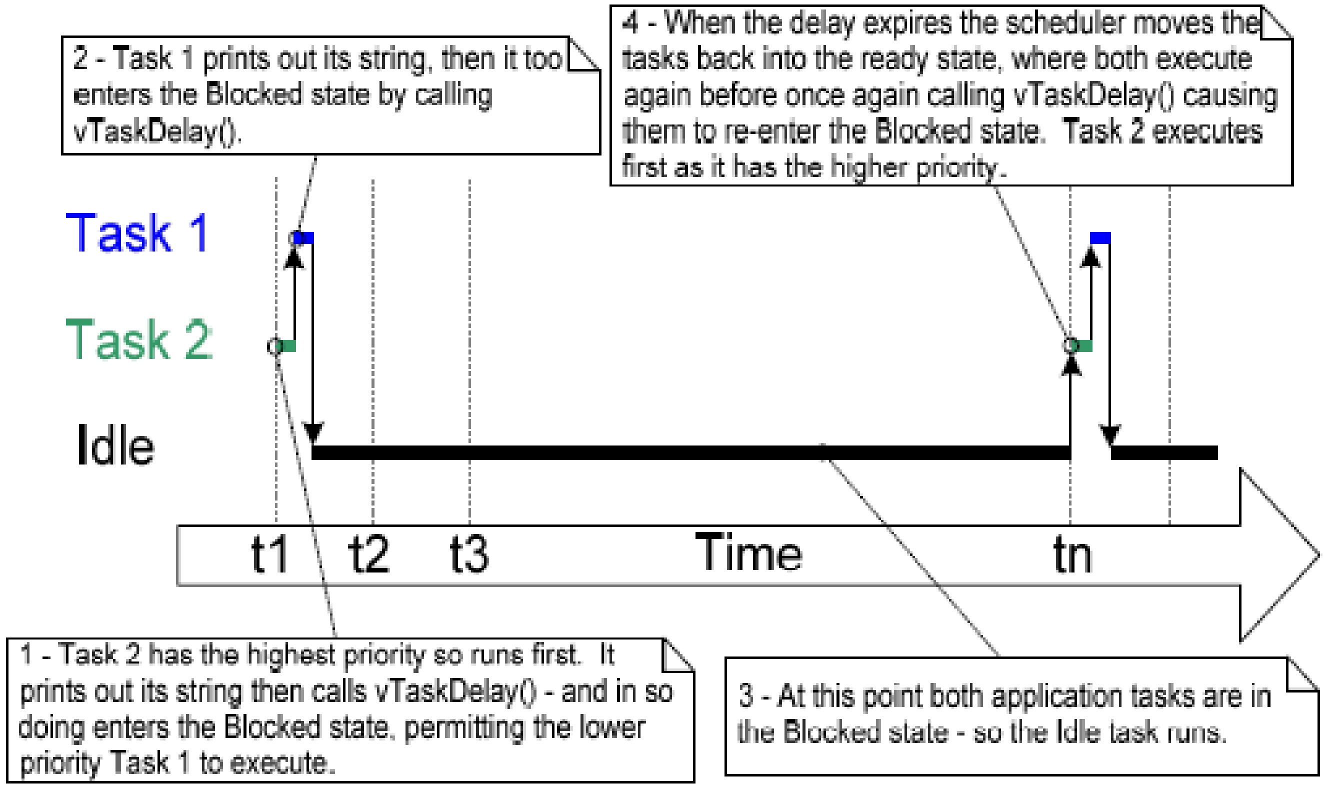
Example 4 - Serial Monitor



```
// #include <Arduino_FreeRTOS.h>
#define mainDELAY_LOOP_COUNT     400000
```

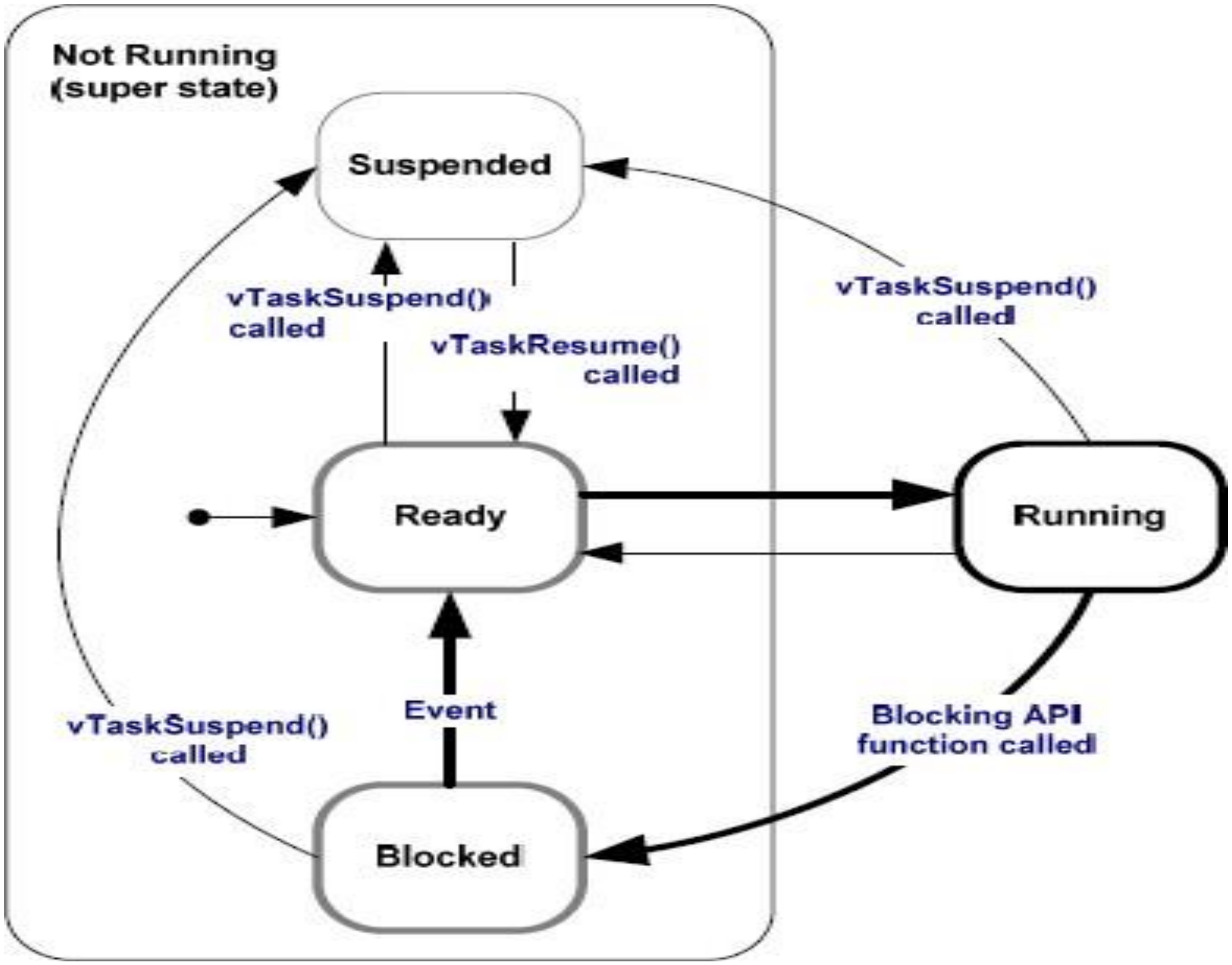
Message (Enter to send message to 'ESP32 Dev Module' on 'COM6')

```
15:21:43.305 -> Task 2 is running
15:21:43.368 -> Task 1 is running
15:21:43.561 -> Task 2 is running
15:21:43.592 -> Task 1 is running
15:21:43.818 -> Task 2 is running
15:21:43.850 -> Task 1 is running
15:21:44.079 -> Task 2 is running
15:21:44.116 -> Task 1 is running
15:21:44.301 -> Task 2 is running
15:21:44.366 -> Task 1 is running
15:21:44.563 -> Task 2 is running
15:21:44.626 -> Task 1 is running
15:21:44.823 -> Task 2 is running
15:21:44.855 -> Task 1 is running
15:21:45.049 -> Task 2 is running
```


2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().
4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.
Task 1
Task 2
Idle
t1
t2
t3
Time
tn
1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.
3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

- 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( TickType_t
*pxPreviousWakeTime, const TickType_t
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_PERIOD_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

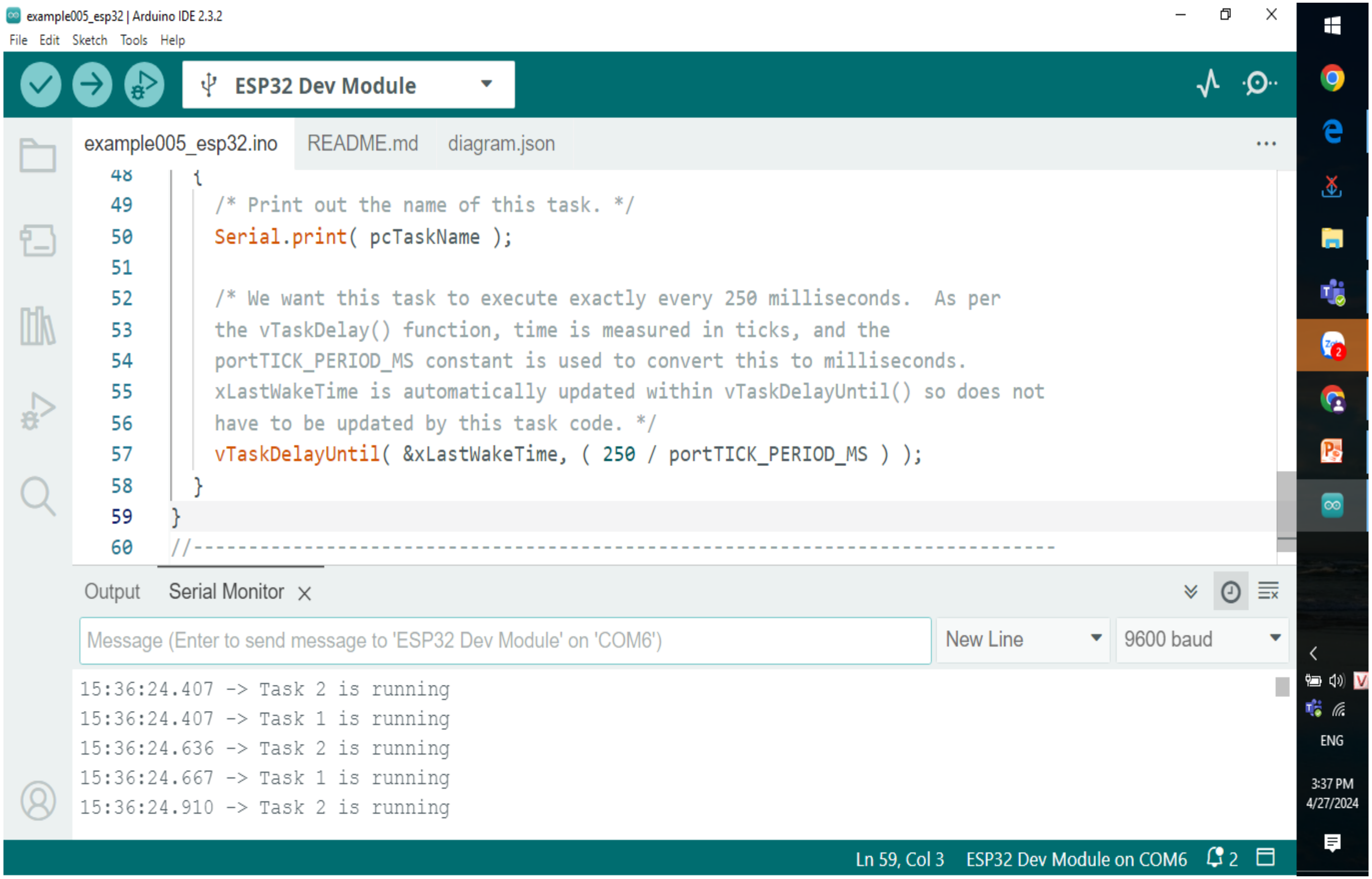
  vTaskDelayUntil(&xLastWakeTime, (250 / portTICK_PERIOD_MS));

  /*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 5 - Serial Monitor



```
  {
    /* Print out the name of this task. */
    Serial.print( pcTaskName );

    /* We want this task to execute exactly every 250 milliseconds.  As per
    the vTaskDelay() function, time is measured in ticks, and the
    portTICK_PERIOD_MS constant is used to convert this to milliseconds.
    xLastWakeTime is automatically updated within vTaskDelayUntil() so does not
    have to be updated by this task code. */
    vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_PERIOD_MS ) );
  }
}
//----------------------------------------------------------------------
```

Message (Enter to send message to 'ESP32 Dev Module' on 'COM6')

New Line | 9600 baud

```
15:36:24.407 -> Task 2 is running
15:36:24.407 -> Task 1 is running
15:36:24.636 -> Task 2 is running
15:36:24.667 -> Task 1 is running
15:36:24.910 -> Task 2 is running
```

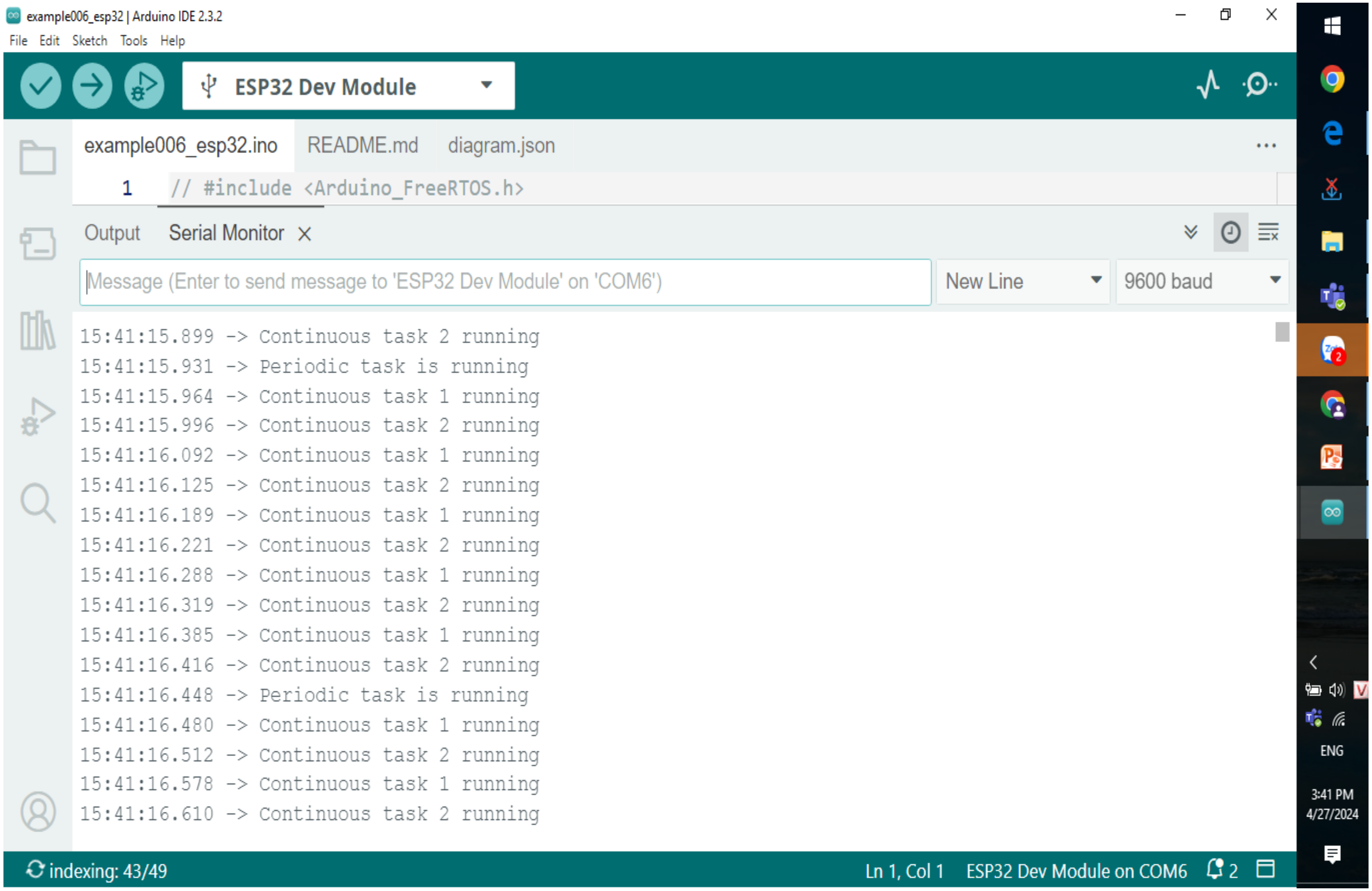
Ln 59, Col 3   ESP32 Dev Module on COM6   2

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;
  pcTaskName = (char *) pvParameters;
  for (;;) { Serial.print(pcTaskName);
        delay(100);
  }}

void vPeriodicTask(void * pvParameters) {
TickType_t xLastWakeTime;
char *pcTaskName;
 pcTaskName = (char *) pvParameters;
 xLastWakeTime = xTaskGetTickCount();
  for (;;){ Serial.print(pcTaskName);
     vTaskDelayUntil(&xLastWakeTime,
                              (500/portTICK_PERIOD_MS));
  }}
```

Example 6 - Serial Monitor



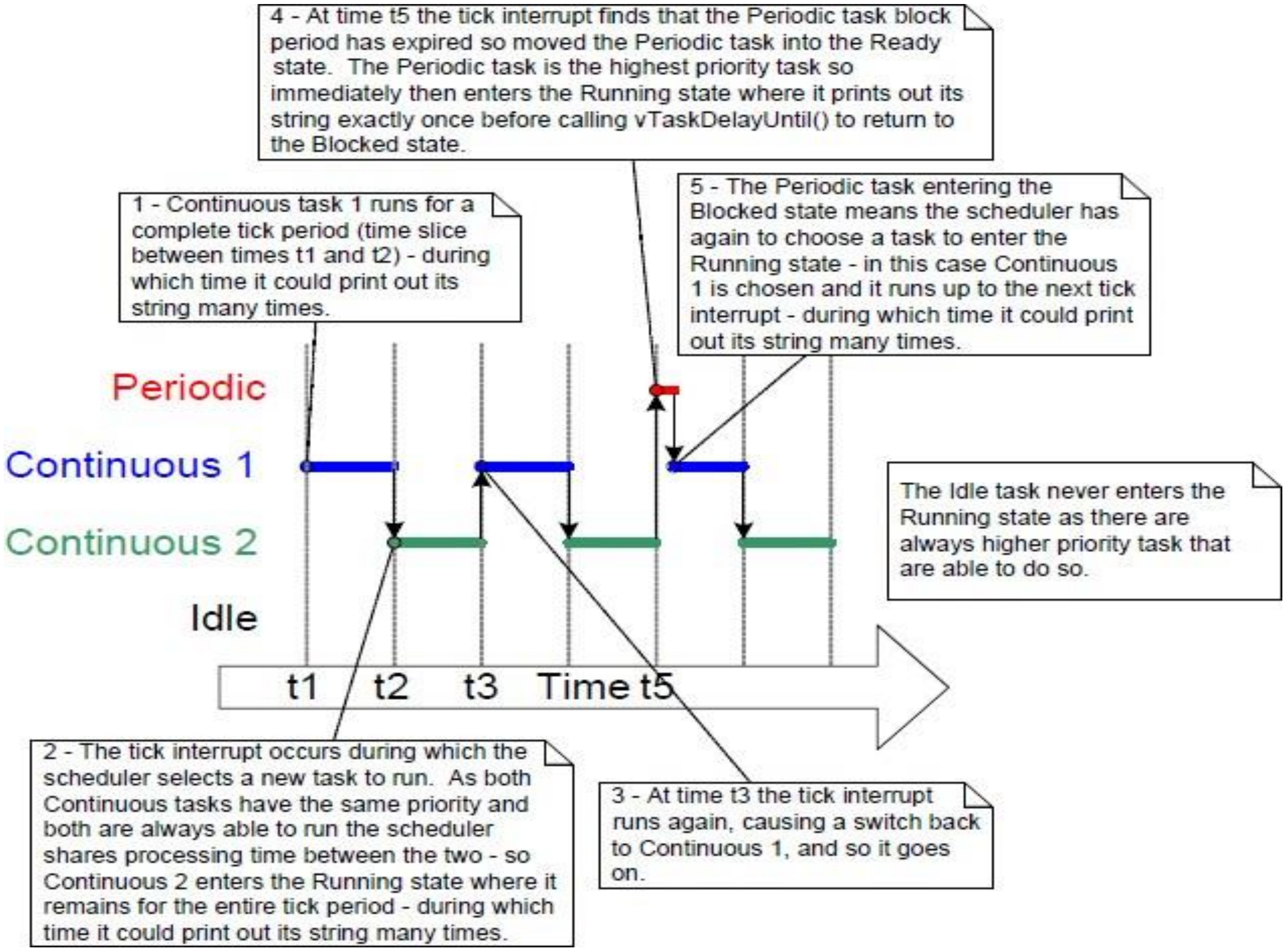
```
// #include <Arduino_FreeRTOS.h>
```

Message (Enter to send message to 'ESP32 Dev Module' on 'COM6')

New Line | 9600 baud

```
15:41:15.899 -> Continuous task 2 running
15:41:15.931 -> Periodic task is running
15:41:15.964 -> Continuous task 1 running
15:41:15.996 -> Continuous task 2 running
15:41:16.092 -> Continuous task 1 running
15:41:16.125 -> Continuous task 2 running
15:41:16.189 -> Continuous task 1 running
15:41:16.221 -> Continuous task 2 running
15:41:16.288 -> Continuous task 1 running
15:41:16.319 -> Continuous task 2 running
15:41:16.385 -> Continuous task 1 running
15:41:16.416 -> Continuous task 2 running
15:41:16.448 -> Periodic task is running
15:41:16.480 -> Continuous task 1 running
15:41:16.512 -> Continuous task 2 running
15:41:16.578 -> Continuous task 1 running
15:41:16.610 -> Continuous task 2 running
```

indexing: 43/49 Ln 1, Col 1 ESP32 Dev Module on COM6 2



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.

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.

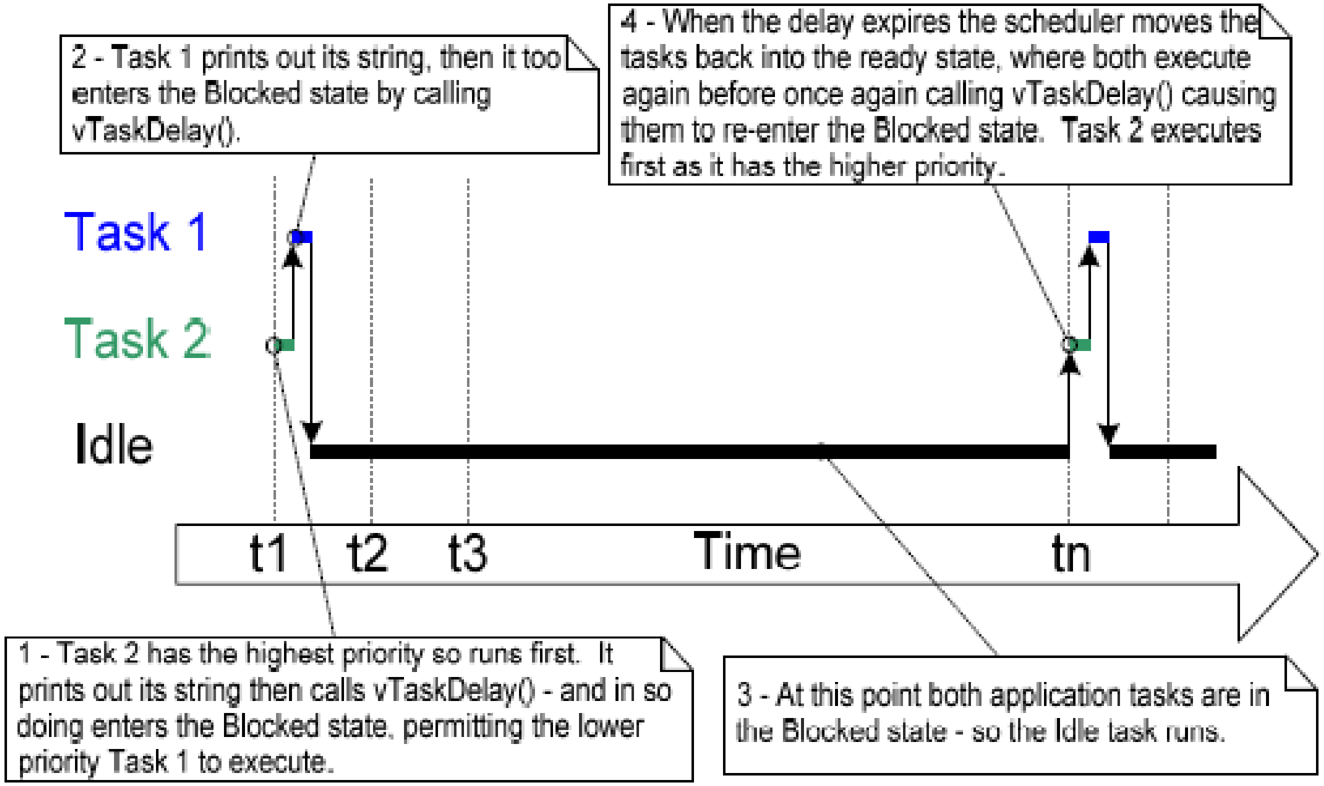
4 - At time t5 the tick interrupt finds that the Periodic task block period has expired so moved the Periodic task into the Ready state. The Periodic task is the highest priority task so immediately then enters the Running state where it prints out its string exactly once before calling vTaskDelayUntil()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

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.

Idle Task and the Idle task hook

- The idle task is created automatically when the RTOS scheduler is started to ensure there is always at least one task that is able to run.
- It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority application tasks in the ready state.
- The idle task is responsible for freeing memory allocated by the RTOS to tasks that have since been deleted. It is therefore important in applications that make use of the vTaskDelete() function to ensure the idle task is not starved of processing time.


2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().
4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.
Task 1
Task 2
Idle
t1
t2
t3
Time
tn
1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.
3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

- 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

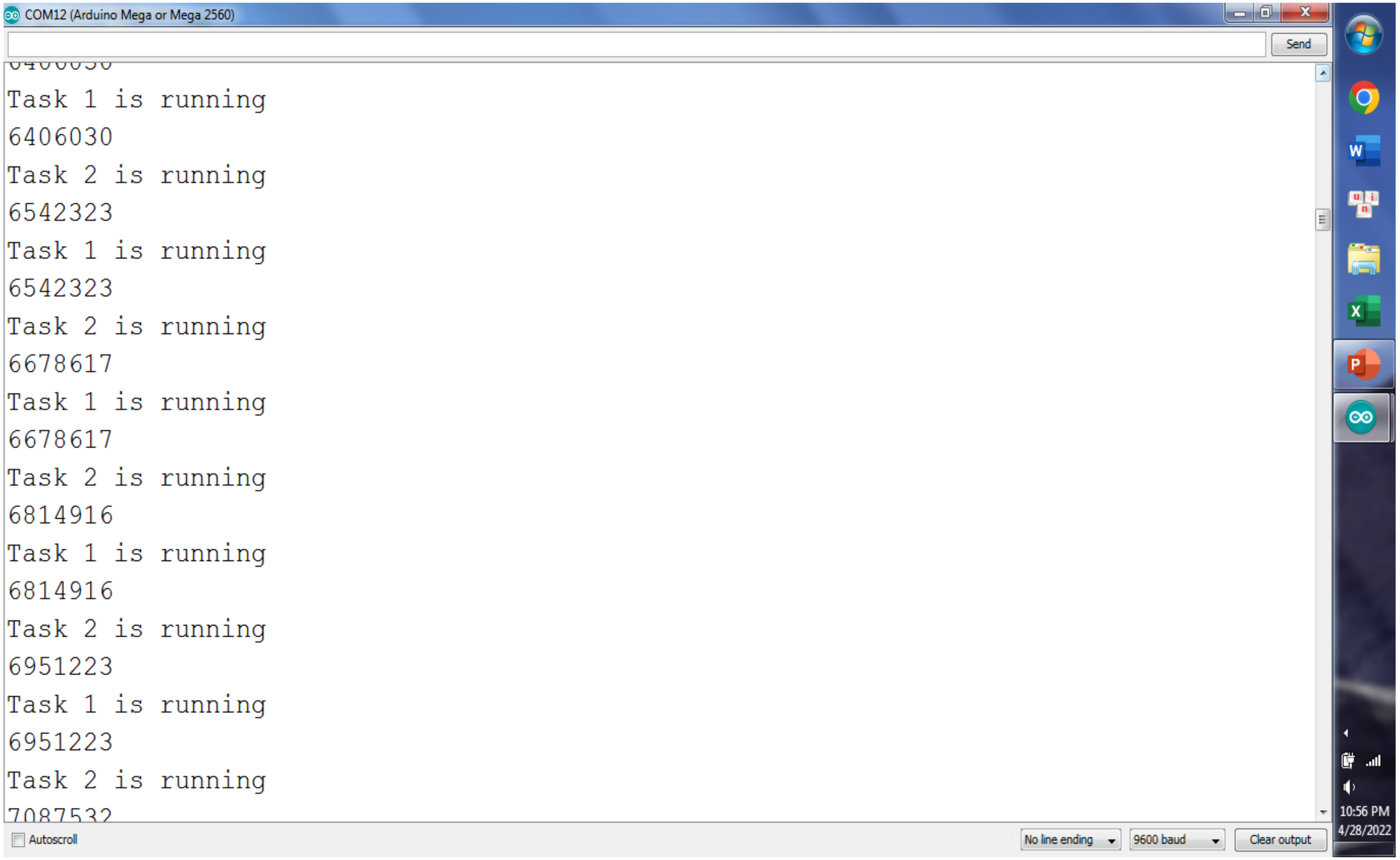
- Set configUSE_IDLE_HOOK to 1
- Add following function

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

- In vTaskFunction(), change

```
        Serial.print(pcTaskName) To
    Serial.print(pcTaskName); Serial.println(ulIdleCycleCount);
```

Example 7 - Serial Monitor



```
Task 1 is running
6406030
Task 2 is running
6542323
Task 1 is running
6542323
Task 2 is running
6678617
Task 1 is running
6678617
Task 2 is running
6814916
Task 1 is running
6814916
Task 2 is running
6951223
Task 1 is running
6951223
Task 2 is running
```

Change the priority of a task

- **void vTaskPrioritySet(TaskHandle_t xTask, UBaseType_t 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
  – xTask: 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

- **UBaseType_t uxTaskPriorityGet(const TaskHandle_t xTask);**

  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 setup() function, create two tasks

```
xTaskCreate(vTask1, “Task 1”, 200,
                    NULL, 2, NULL);
xTaskCreate(vTask2, “Task 2”, 200,
                    NULL, 1, &xTask2Handle);
```

- Change vTask1 by initialization

  unsigned portBASE_TYPE uxPriority;

  uxPriority = uxTaskPriorityGet(NULL);

- And adding to the infinite loop

  vTaskPrioritySet(xTask2Handle, (uxPriority+1));

- Change vTask2 by initialization

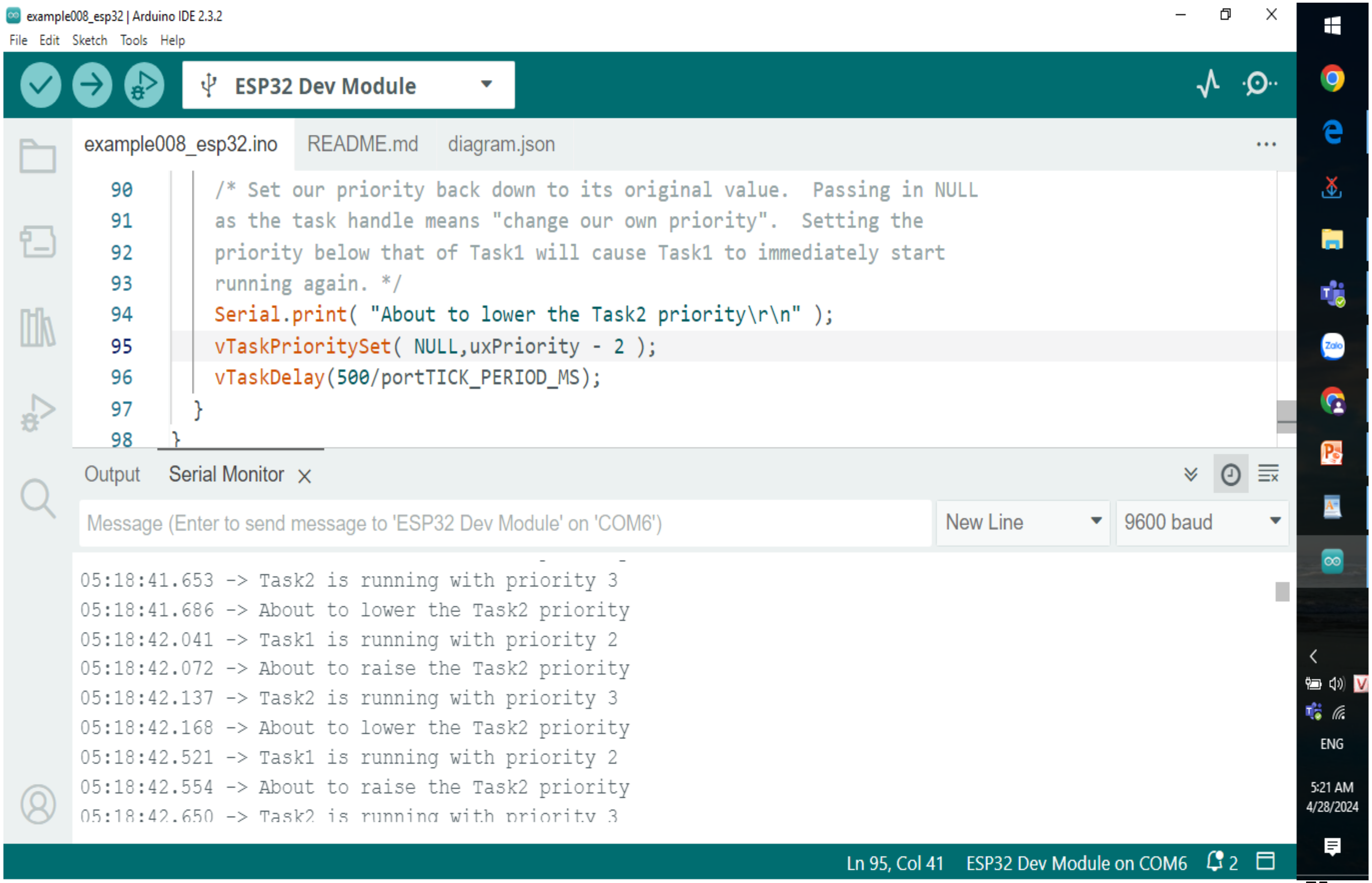
  unsigned portBASE_TYPE uxPriority;

  uxPriority = uxTaskPriorityGet(NULL);

- And adding to the infinite loop

  vTaskPrioritySet(NULL, (uxPriority-2));

Example 8 - Serial Monitor



example008_esp32 | Arduino IDE 2.3.2

File Edit Sketch Tools Help

ESP32 Dev Module

example008_esp32.ino README.md diagram.json

```
    /* Set our priority back down to its original value.  Passing in NULL
    as the task handle means "change our own priority".  Setting the
    priority below that of Task1 will cause Task1 to immediately start
    running again. */
    Serial.print( "About to lower the Task2 priority\r\n" );
    vTaskPrioritySet( NULL,uxPriority - 2 );
    vTaskDelay(500/portTICK_PERIOD_MS);
  }
}
```

Output Serial Monitor

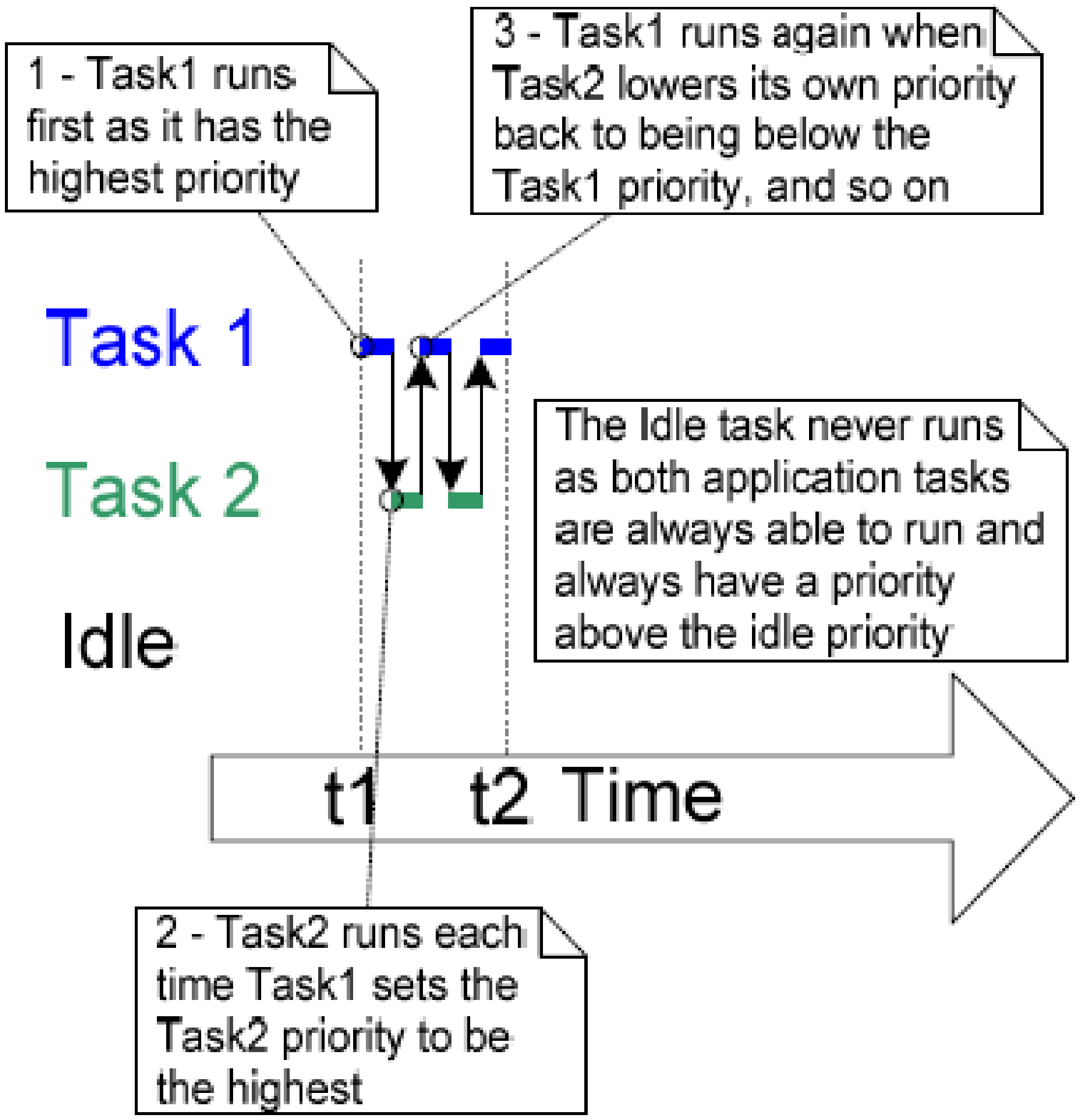
Message (Enter to send message to 'ESP32 Dev Module' on 'COM6')

New Line 9600 baud

```
05:18:41.653 -> Task2 is running with priority 3
05:18:41.686 -> About to lower the Task2 priority
05:18:42.041 -> Task1 is running with priority 2
05:18:42.072 -> About to raise the Task2 priority
05:18:42.137 -> Task2 is running with priority 3
05:18:42.168 -> About to lower the Task2 priority
05:18:42.521 -> Task1 is running with priority 2
05:18:42.554 -> About to raise the Task2 priority
05:18:42.650 -> Task2 is running with priority 3
```

Ln 95, Col 41 ESP32 Dev Module on COM6

Task execution sequence


1 - Task1 runs first as it has the highest priority
3 - Task1 runs again when Task2 lowers its own priority back to being below the Task1 priority, and so on
Task 1
Task 2
Idle
The Idle task never runs as both application tasks are always able to run and always have a priority above the idle priority
t1
t2
Time
2 - Task2 runs each time Task1 sets the Task2 priority to be the highest

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( TaskHandle_t xTask );
```

  - xTask: 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

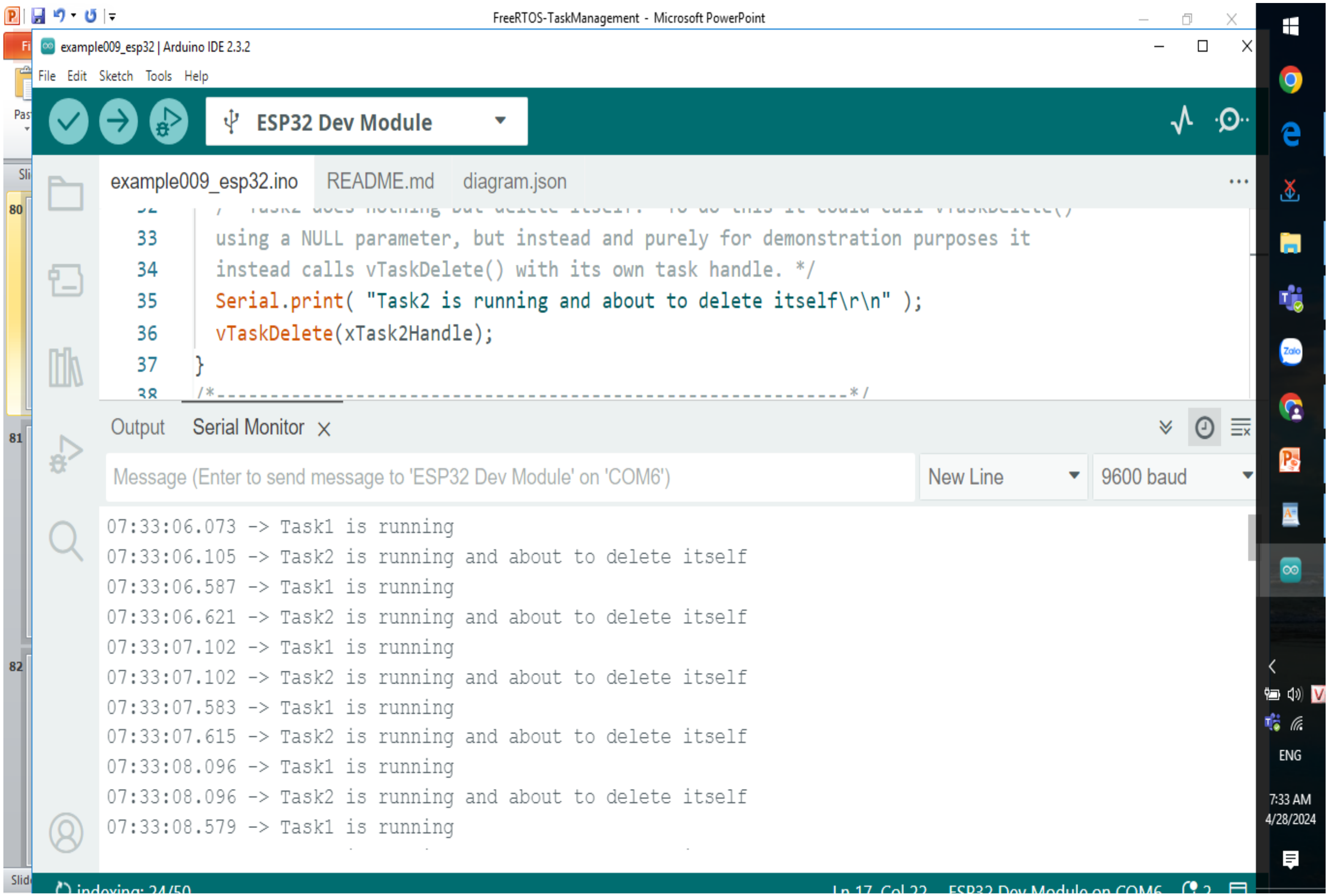
Example 9

```
void setup( void )
{
   Serial.begin(9600);
  /* Create the first task at priority 1.  This time the task parameter is
  not used and is set to NULL.  The task handle is also not used so likewise
  is also set to NULL. */
  xTaskCreate( vTask1, "Task 1", 200, NULL, 1, NULL );
          /* The task is created at priority 1 ^. */

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

  for( ;; );

}
```

```
void vTask1( void *pvParameters )
{
const TickType_t xDelay100ms = 100 / portTICK_PERIOD_MS;

  for( ;; )
  {
    /* Print out the name of this task. */
    Serial.print( "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 we want to obtain a handle to the
    task so pass in the address of the xTask2Handle variable. */
    xTaskCreate( vTask2, "Task 2", 200, 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 );
  }
}
```

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

Example 9 - Serial Monitor



```
using a NULL parameter, but instead and purely for demonstration purposes it
instead calls vTaskDelete() with its own task handle. */
Serial.print( "Task2 is running and about to delete itself\r\n" );
vTaskDelete(xTask2Handle);
}
```

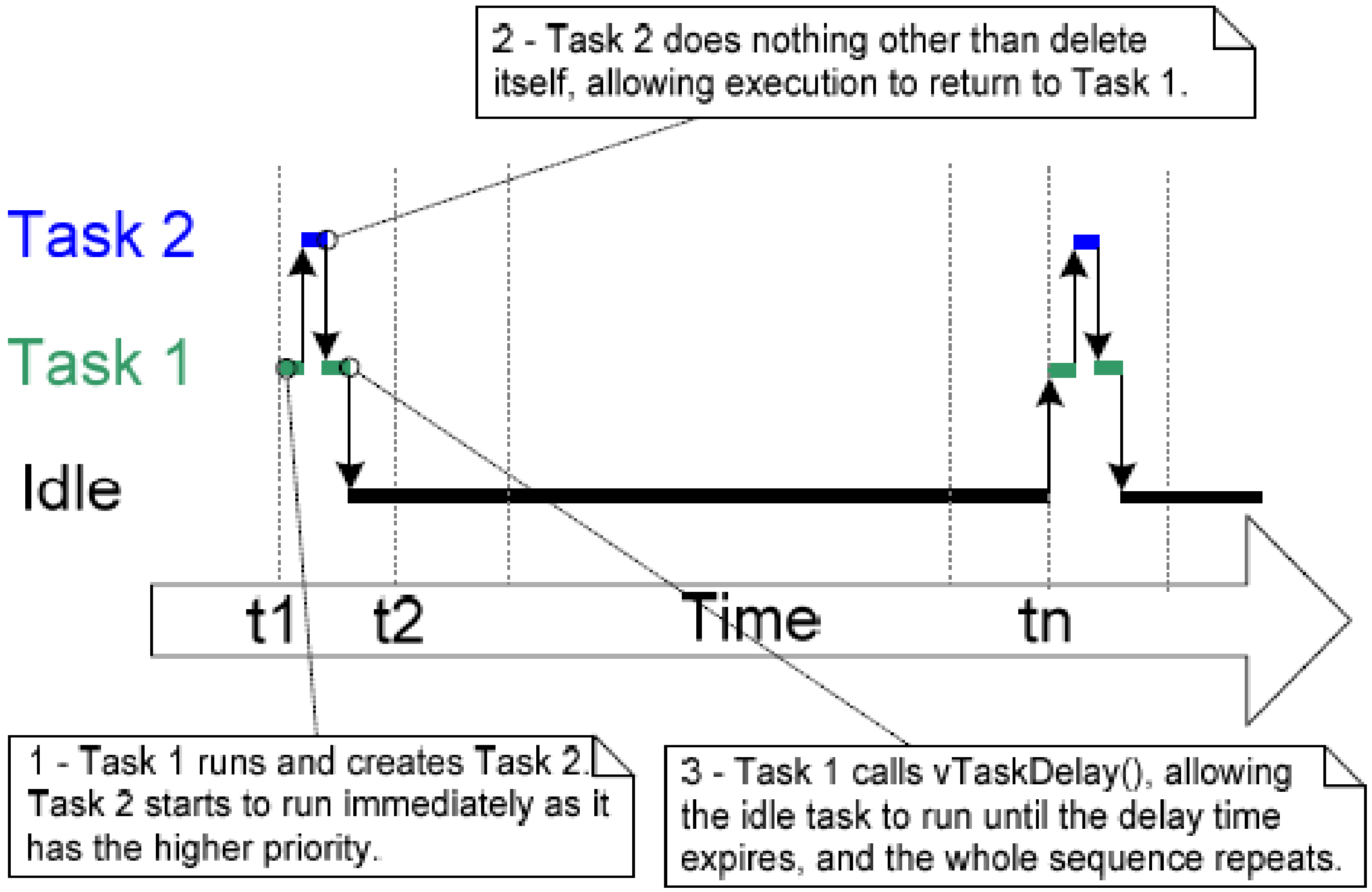
```
07:33:06.073 -> Task1 is running
07:33:06.105 -> Task2 is running and about to delete itself
07:33:06.587 -> Task1 is running
07:33:06.621 -> Task2 is running and about to delete itself
07:33:07.102 -> Task1 is running
07:33:07.102 -> Task2 is running and about to delete itself
07:33:07.583 -> Task1 is running
07:33:07.615 -> Task2 is running and about to delete itself
07:33:08.096 -> Task1 is running
07:33:08.096 -> Task2 is running and about to delete itself
07:33:08.579 -> Task1 is running
```

FreeRTOS-TaskManagement - Microsoft PowerPoint
example009_esp32 | Arduino IDE 2.3.2
File Edit Sketch Tools Help
ESP32 Dev Module
example009_esp32.ino
README.md
diagram.json
Output
Serial Monitor
Message (Enter to send message to 'ESP32 Dev Module' on 'COM6')
New Line
9600 baud
Ln 17, Col 22
ESP32 Dev Module on COM6
7:33 AM
4/28/2024

Example 9 Deleting tasks (Behavior)

1. Task 1 is created by setup() 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


2 - Task 2 does nothing other than delete itself, allowing execution to return to Task 1.
Task 2
Task 1
Idle
t1
t2
Time
tn
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.

Scheduling Algorithms - Summary

- So far, 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.

Selecting the Scheduling Algorithm

The scheduling algorithm is the software routine that decides which *Ready* state task to transition into the *Running* state. All the examples so far have used the same scheduling algorithm, but the algorithm can be changed using the configUSE_PREEMPTION and configUSE_TIME_SLICING configuration constants. Both constants are defined in FreeRTOSConfig.h.

Scheduling Algorithm	Prioritized	configUSE_PREEMPTION	configUSE_TIME_SLICING
Preemptive With Time Slicing	Yes	1	1
Preemptive Without Time Slicing	Yes	1	0
Co-Operative	No	0	Any

Fixed priority Pre-emptive scheduling

- Fixed priority: Scheduling algorithms described as 'Fixed Priority' do not change the priority assigned to the tasks being scheduled, but also do not prevent the tasks themselves from changing their own priority or that of other tasks.

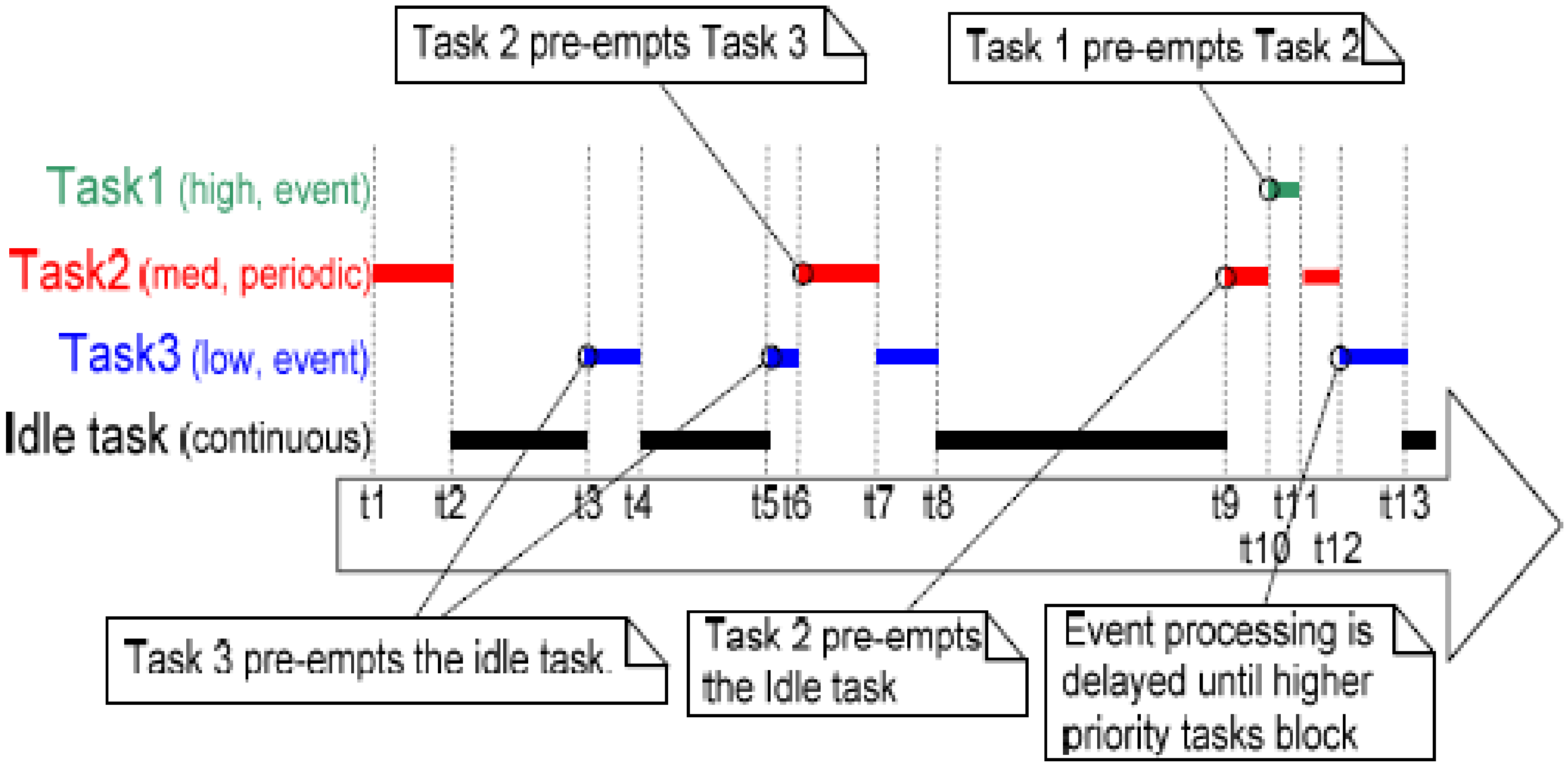
- Pre-emptive: Preemptive scheduling algorithms will immediately 'preempt' the *Running* state task if a task that has a priority higher than the *Running* state task enters the *Ready* state.

- Time Slicing: Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the *Blocked* state. Scheduling algorithms described as using *Time Slicing* select a new task to enter the *Running* state at the end of each time slice if there are other *Ready* state tasks that have the same priority as the Running task. A time slice is equal to the time between two RTOS tick interrupts.

Tasks in the Blocked state

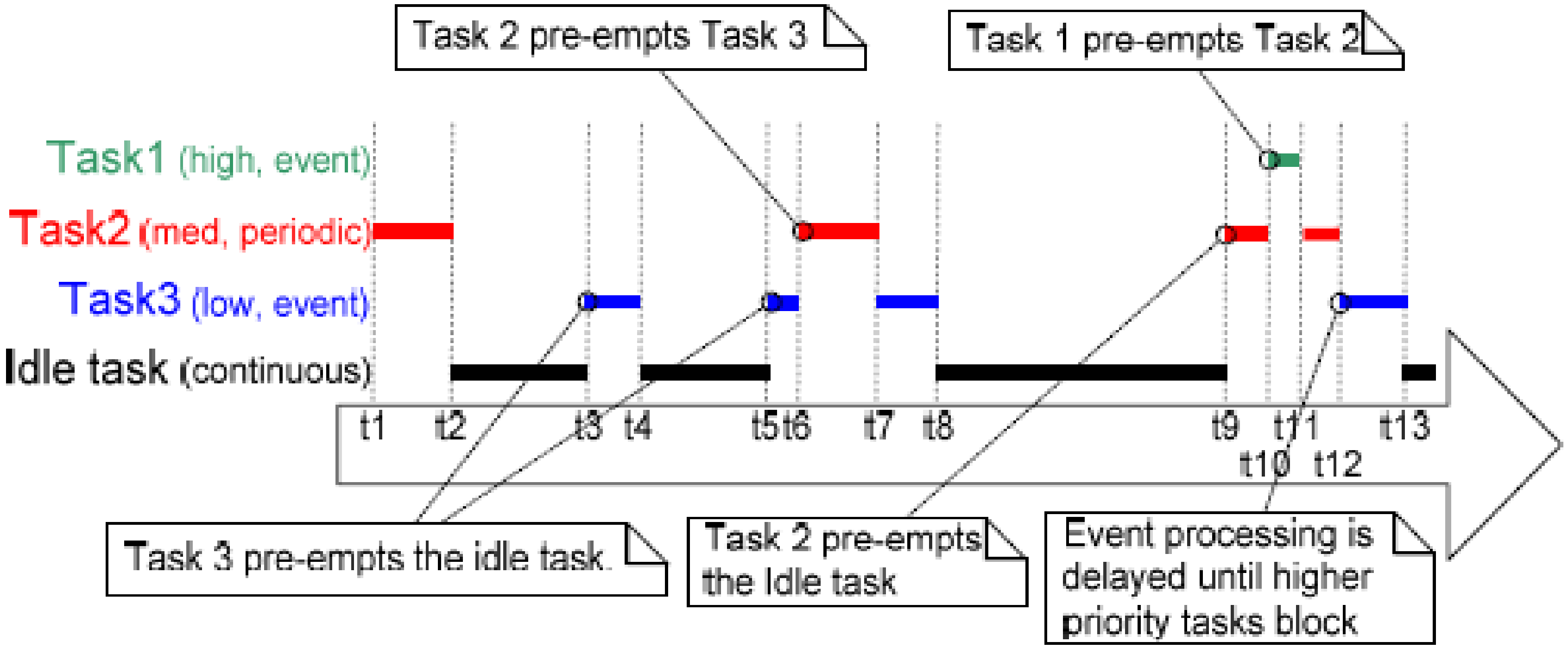
- Tasks can wait in the Blocked state for an **event** and are automatically moved back to the Ready state when the **event** occurs.
- Temporal events
  – Occur at a particular time, e.g. a block time expires.
  – Generally be used to implement periodic or timeout behavior.
- Synchronization events
  – 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 highlighting task prioritization and preemption in a hypothetical application in which each task has been assigned a unique priority



- *Event for Task 1 occur at : t10*
- *Task2 is released at : t1, t6, t9*
- *Events for Task 3 occur at: t3, t5, between t9 and t12*

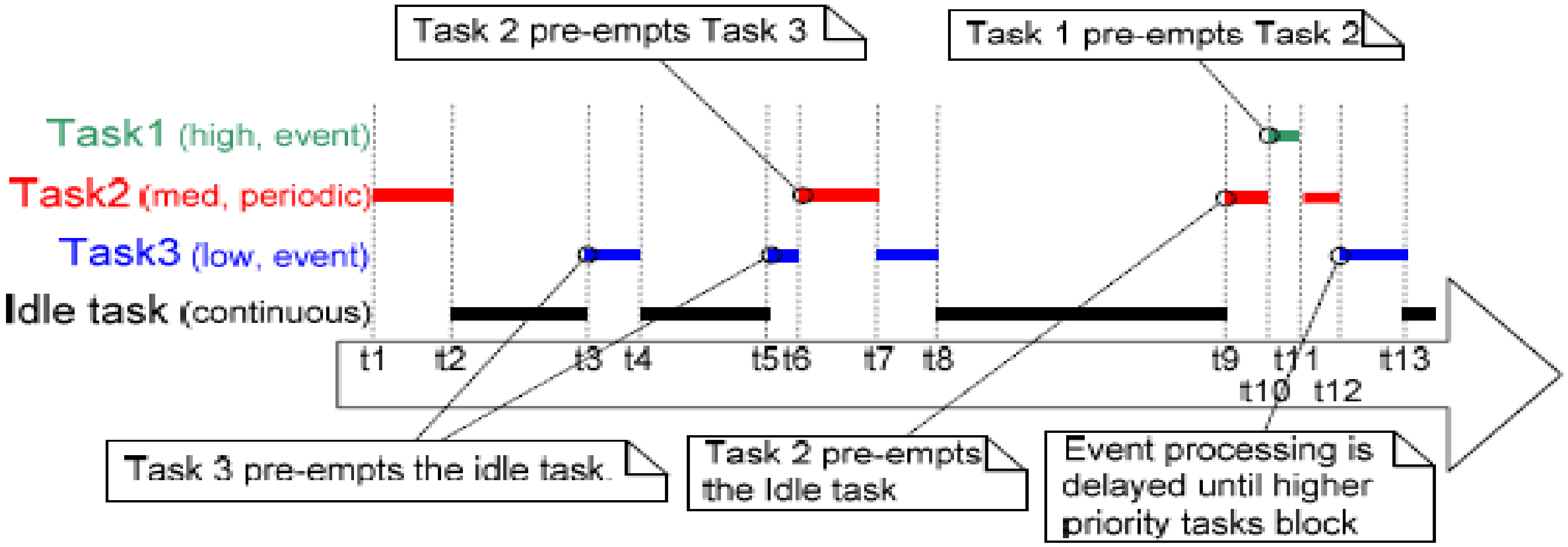
- Idle task
  - The idle task is running at the lowest priority, so gets pre-empted every time a higher priority task enters the Ready state
    - at times t3, t5, and t9.


Task 2 pre-empts Task 3
Task 1 pre-empts Task 2
Task1 (high, event)
Task2 (med, periodic)
Task3 (low, event)
Idle task (continuous)
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13
Task 3 pre-empts the idle task.
Task 2 pre-empts the Idle task
Event processing is delayed until higher priority tasks block

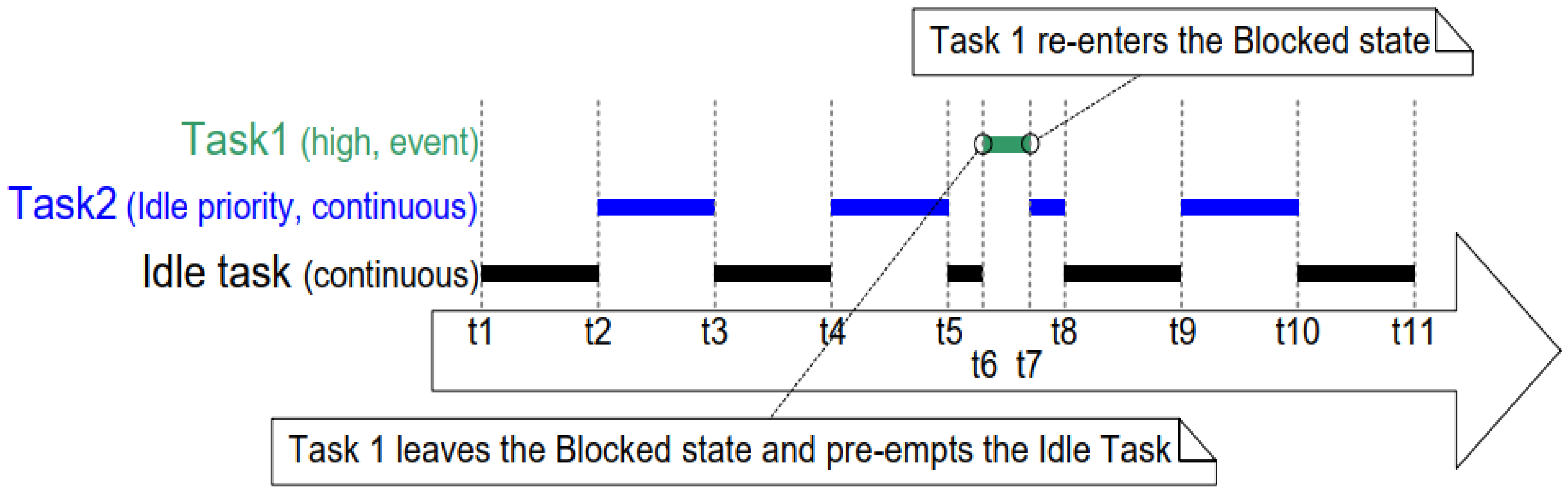
- Task 3
  - An event-driven task
    - Execute with a low priority, but above the Idle task priority.
  - It spends most of its time in the Blocked state waiting for the event of interest, transitioning from Blocked to Ready state each time the event occurs.
    - All FreeRTOS inter-task communication mechanisms (task notifications, 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 because, until then, the higher priority tasks Task 1 and Task 2 are still executing. It is only at time t12 that both Task 1 and Task 2 are in the *Blocked* state, 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.

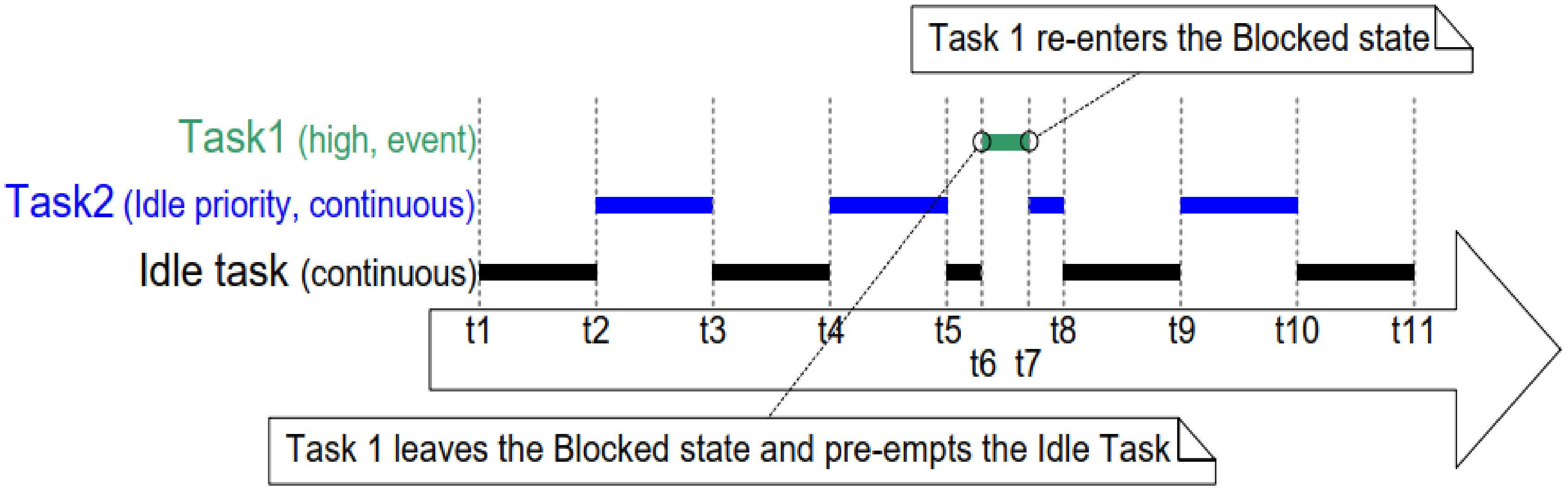


Execution pattern highlighting task prioritization and time slicing in a hypothetical application in which two tasks run at the same priority


Task 1 re-enters the Blocked state
Task1 (high, event)
Task2 (Idle priority, continuous)
Idle task (continuous)
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
Task 1 leaves the Blocked state and pre-empts the Idle Task

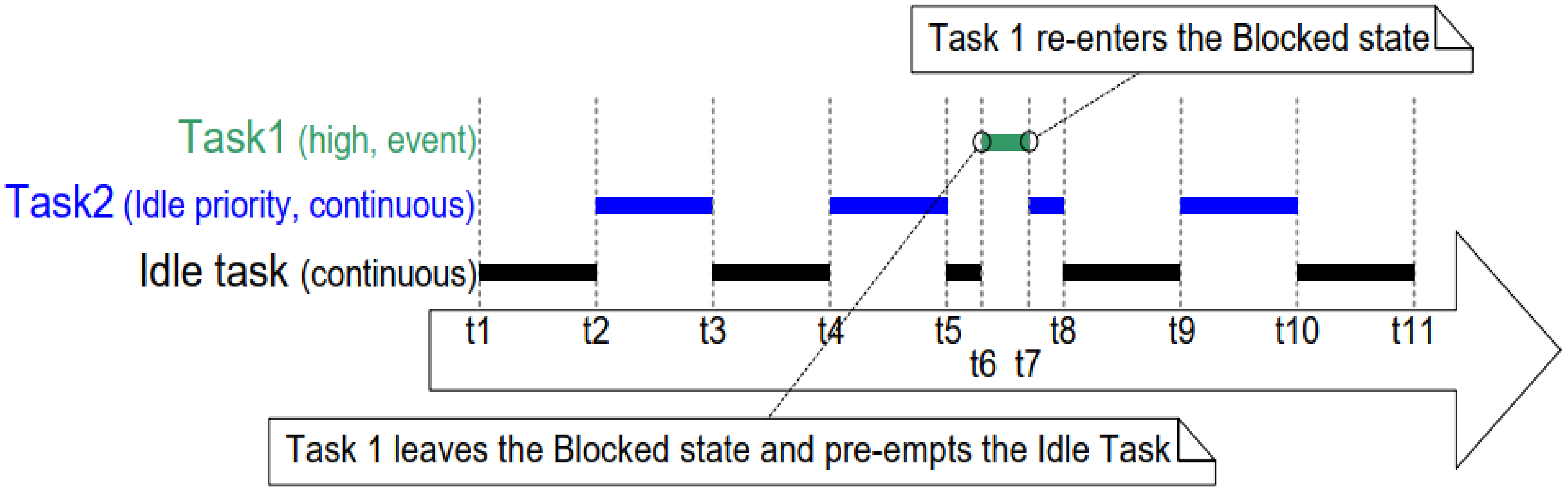
- **The Idle Task and Task 2**

The Idle task and Task 2 are both continuous processing tasks, and both have a priority of 0 (the lowest possible priority). The scheduler only allocates processing time to the priority 0 tasks when there are no higher priority tasks that are able to run, and shares the time that is allocated to the priority 0 tasks by time slicing. A new time slice starts on each tick interrupt, which occurs at times t1, t2, t3, t4, t5, t8, t9, t10 and t11.
The Idle task and Task 2 enter the *Running* state **in turn**, which can result in both tasks being in the *Running* state for part of the same time slice, as happens between time t5 and time t8.


Task 1 re-enters the Blocked state
Task1 (high, event)
Task2 (Idle priority, continuous)
Idle task (continuous)
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
Task 1 leaves the Blocked state and pre-empts the Idle Task

- **Task 1**

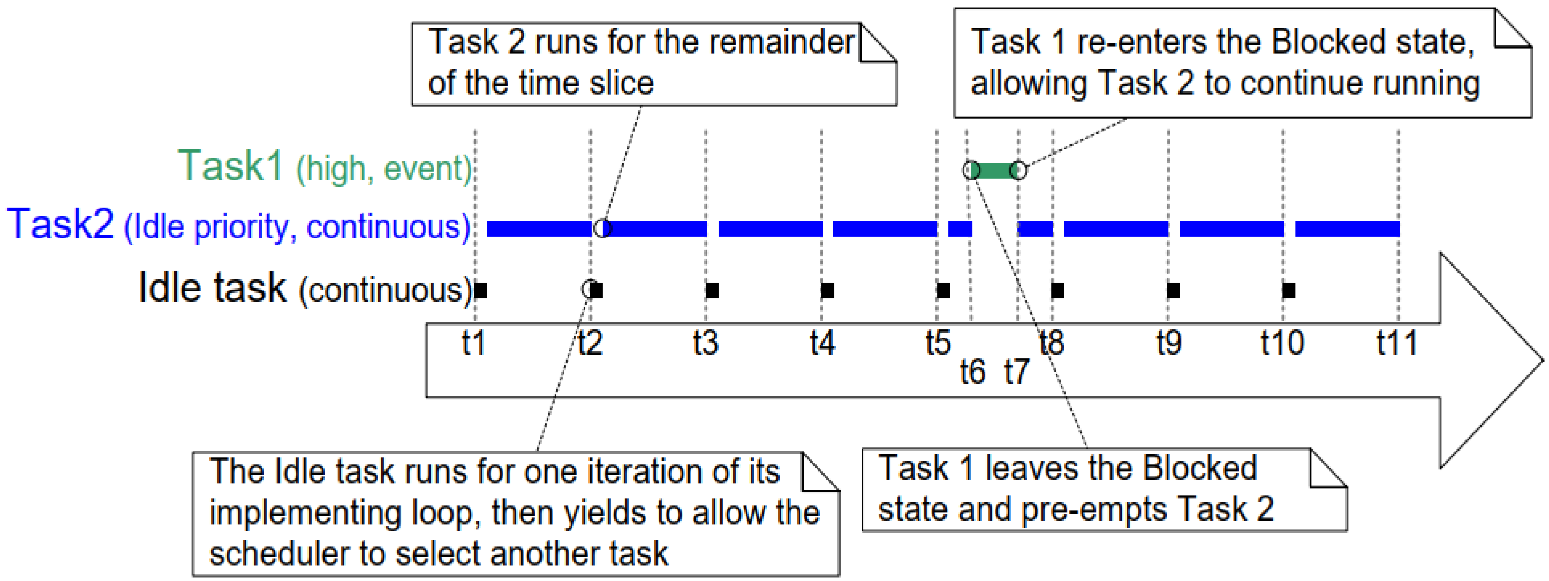
The priority of Task 1 is higher than the Idle priority. Task 1 is an event driven task that spends most of its time in the *Blocked* state waiting for its event of interest, transitioning from the *Blocked* state to the *Ready* state each time the event occurs. The event of interest occurs at time t6. At t6 Task 1 becomes the highest priority task that is able to run, and therefore Task 1 preempts the Idle task part way through a time slice. Processing of the event completes at time t7, at which point Task 1 re-enters the Blocked state.


Task 1 re-enters the Blocked state
Task1 (high, event)
Task2 (Idle priority, continuous)
Idle task (continuous)
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
Task 1 leaves the Blocked state and pre-empts the Idle Task

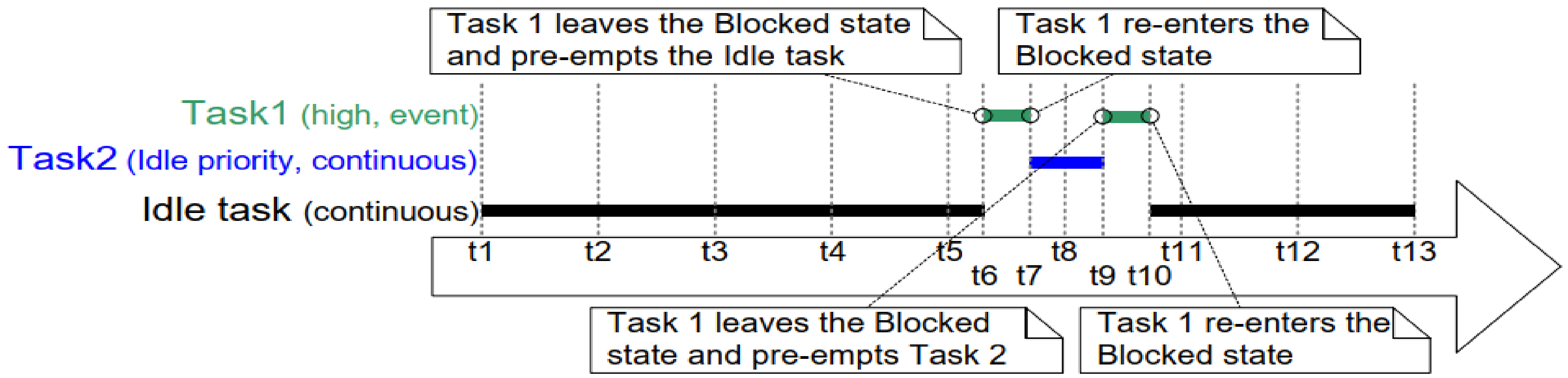
The configIDLE_SHOULD_YIELD compile time configuration constant can be used to change ***how the Idle task is scheduled****:*

•If configIDLE_SHOULD_YIELD is set to 0 then the Idle task remains in the Running state for the entirety of its time slice, unless it is preempted by a higher priority task.

•If ***configIDLE_SHOULD_YIELD is set to 1*** *then the Idle task yields (voluntarily gives up whatever remains of its allocated time slice) on each iteration of its loop if there are other Idle priority tasks in the Ready state.*


Task 2 runs for the remainder of the time slice
Task 1 re-enters the Blocked state, allowing Task 2 to continue running
Task1 (high, event)
Task2 (Idle priority, continuous)
Idle task (continuous)
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
The Idle task runs for one iteration of its implementing loop, then yields to allow the scheduler to select another task
Task 1 leaves the Blocked state and pre-empts Task 2

Prioritized Preemptive Scheduling without Time Slicing


Task 1 leaves the Blocked state and pre-empts the Idle task
Task 1 re-enters the Blocked state
Task1 (high, event)
Task2 (Idle priority, continuous)
Idle task (continuous)
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
t13
Task 1 leaves the Blocked state and pre-empts Task 2
Task 1 re-enters the Blocked state

If time slicing is not used, then the scheduler only selects a new task to enter the *Running* state when either:

- A higher priority task enters the *Ready* state.
- The task in the *Running* state enters the *Blocked* or *Suspended* state.

There are fewer task context switches when time slicing is not used than when time slicing is used. Therefore, turning time slicing off results in a reduction in the scheduler's processing overhead. However, **turning time slicing off can also result in tasks of equal priority receiving greatly different amounts of processing time, a scenario demonstrated by Figure above**. For this reason, running the scheduler without time slicing is considered an advanced technique that should only be used by experienced users.

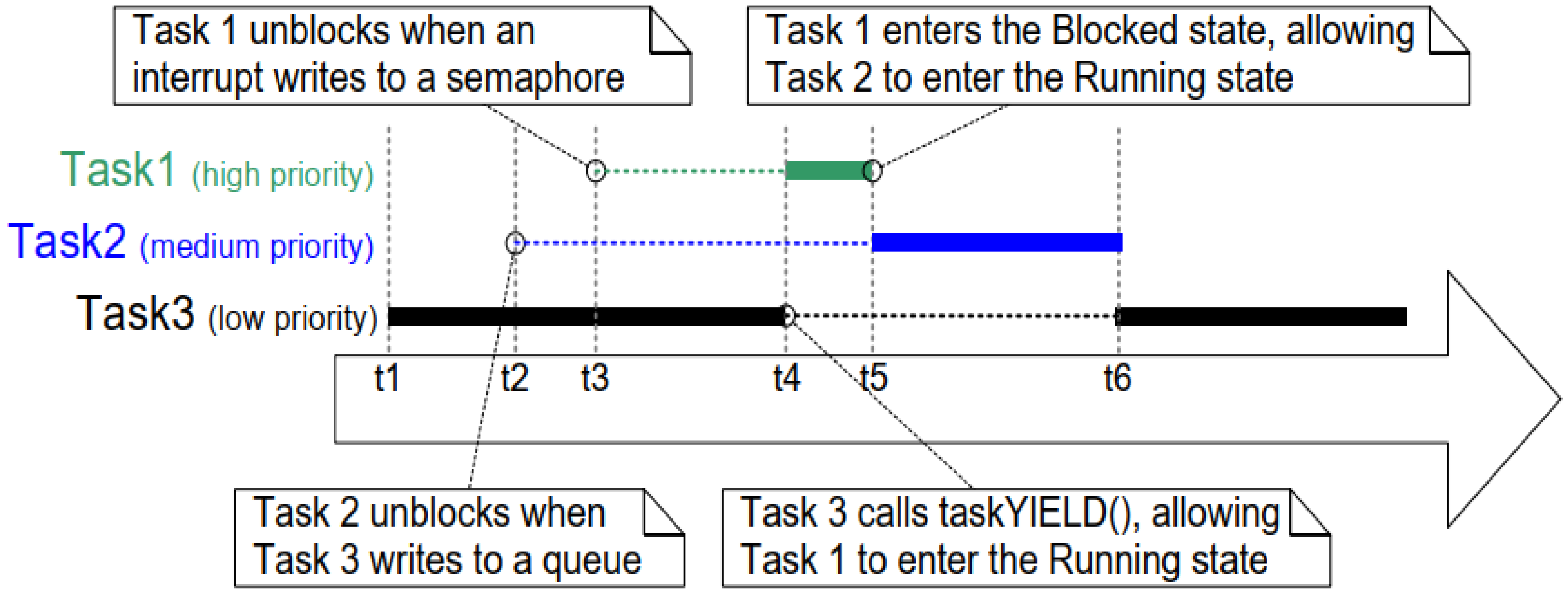
Co-operative scheduling

- When using the cooperative scheduler, a context switch occur only when
  - the Running state task enters the Blocked state

    or
  - the Running state task explicitly calls taskYIELD() – a Kernel Control API function.
- Tasks will never be pre-empted and tasks of equal priority will not automatically share processing time (time slicing cannot be used).
  - Results in a less responsive system.

Co-operative scheduling

Figure below demonstrates the behavior of the cooperative scheduler.
The horizontal dashed lines in the figure show when a task is in the Ready state.


Task 1 unblocks when an interrupt writes to a semaphore
Task 1 enters the Blocked state, allowing Task 2 to enter the Running state
Task1 (high priority)
Task2 (medium priority)
Task3 (low priority)
t1
t2
t3
t4
t5
t6
Task 2 unblocks when Task 3 writes to a queue
Task 3 calls taskYIELD(), allowing Task 1 to enter the Running state