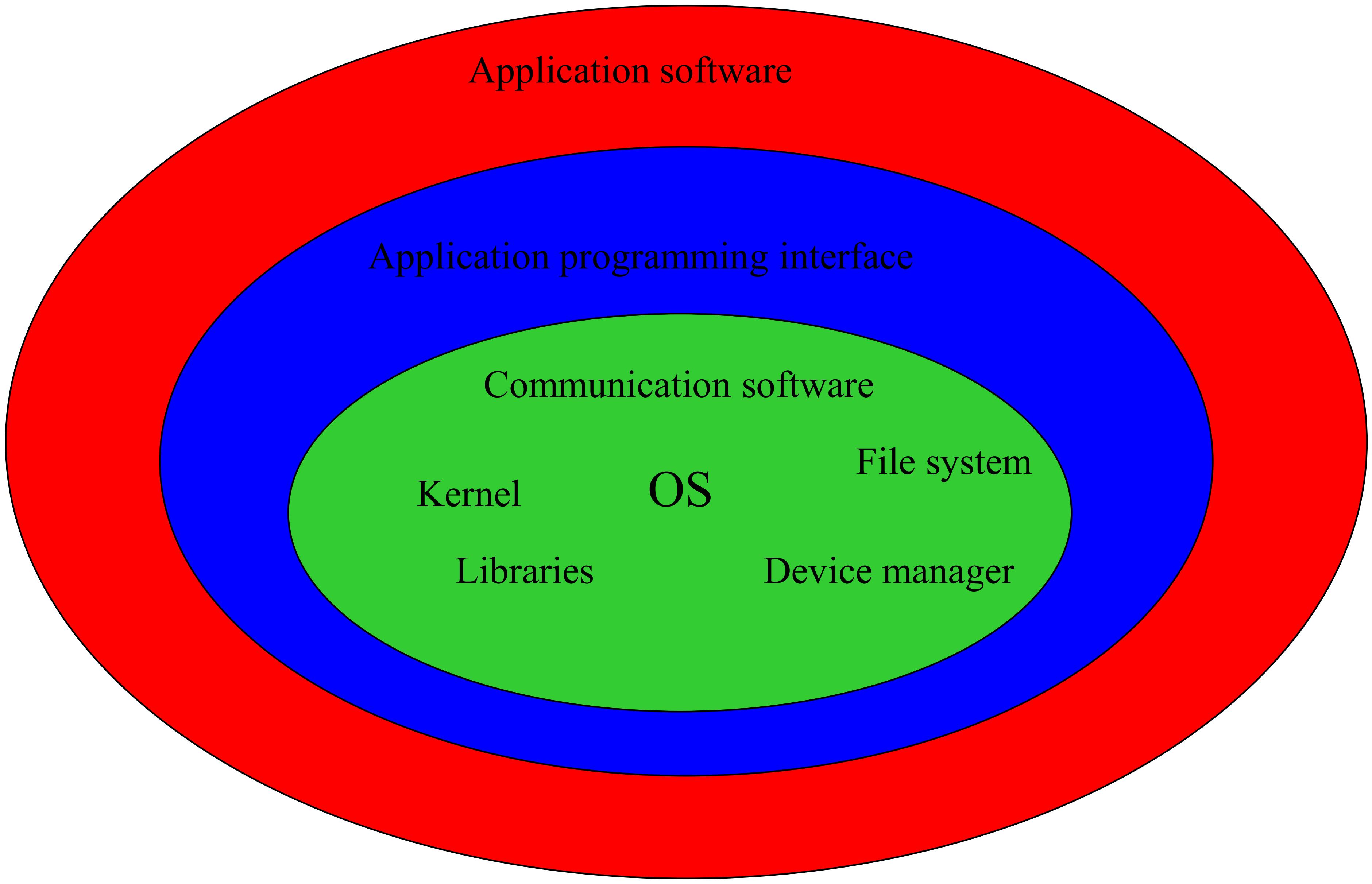
RTOS

Task & Task Management

Real-Time Operating System (RTOS)

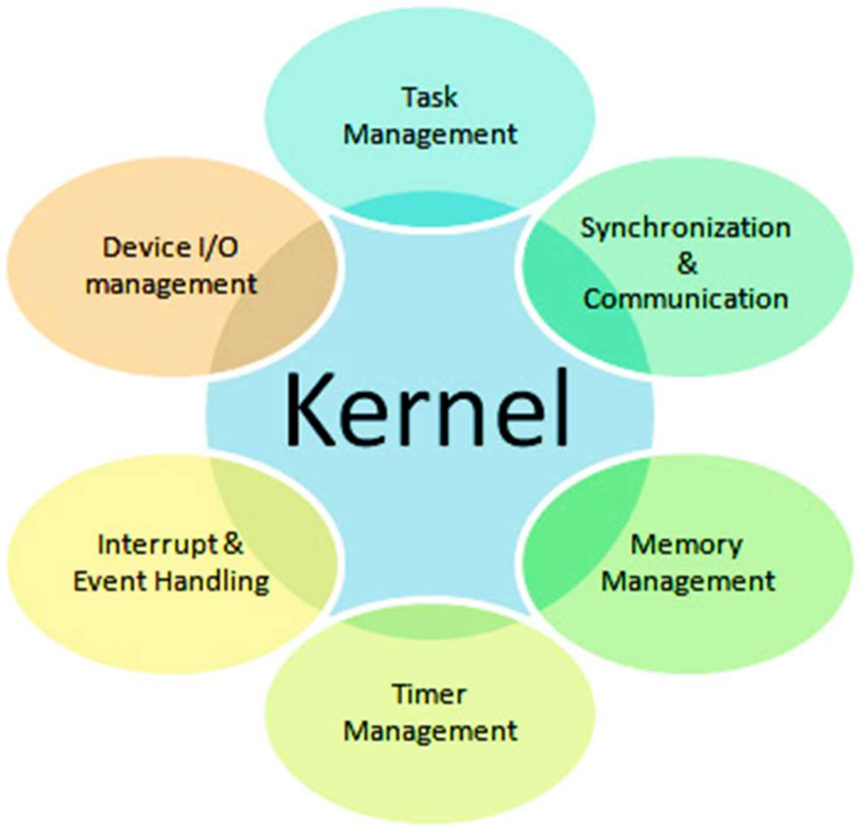
- An operating system intended to server real-time application requests
- Specified time constrains
- Applications
  - Automotive systems
  - Avionics
  - Pacemaker

Embedded software architecture



RTOS

- Kernel is the heart of the OS.
- Manages tasks through scheduling to achieve desired performance.
- Provides inter task communication.
- Kernel consists of objects such as tasks, mutexes, ISRs, events, message box, mail boxes, pipes and timers.
- Kernel provides memory management services, interrupt handling services and device management services.
- Device manager manages the IO devices through interrupts and device drivers.



Real Time Operating Systems

- Operating systems - Solving problems using organized tasks that work together
- Coordination requires
  - Sharing data
  - Synchronization
  - Scheduling
  - Sharing resources
- An operating system that meets specified time constraints is called a Real-Time Operating System (RTOS)

Tasks

- Individual jobs that must be done (in coordination) to complete a large job
- Partition design:
    - Based on things that could/should be done together
    - In a way to make the problem easier
    - Based on knowing the most efficient partitioning for execution
- Example tasks/design partitions for a digital thermometer with flashing temperature indicator
    - Detect & Signal button press
    - Read Temperature & update flash rate
    - Update LCD
    - Flash LED

Defining a task

- Independent thread of execution that can compete with concurrent tasks for processor execution time.
    - Thus, schedulable & allocated a priority level according to the scheduling algorithm
- Elements of a task
    - Unique ID
    - Task control block (TCB)
    - Stack
    - Priority (if part of a preemptive scheduling plan)
    - Task routine

RTOS

- Concept of RTOS:
- What is Task?
- Task is basic building block of software. Each task compete for the CPU time independently. Each task will have its own stack area.
- Task can be subroutine.
- Each task is implemented as an infinite loop
- for (; ;)   while (1)

```
{                                              {
 statements                        or                statements
}                                              }
```

Tasks/Processes

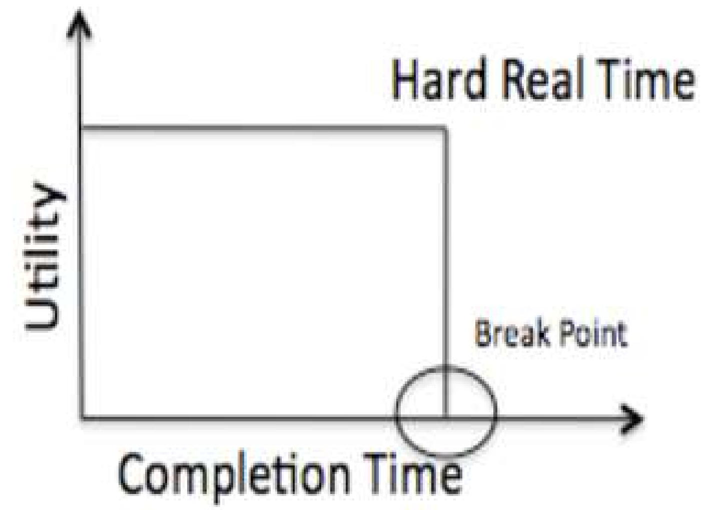
- Tasks require resources or access to resources
  - Processor, stack memory registers, P.C. I/O Ports, network connections, file, etc…
- These should be allocated to a processes when chosen to be executed by the operating system
- Contents of PC & other pieces of data associated with the task determine its process state

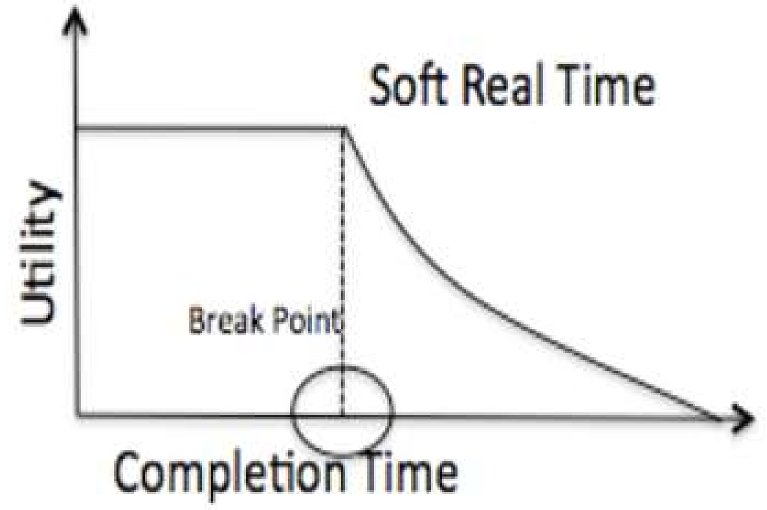
Task Terminology

- Execution Time – Amount of time each process requires to complete
- Persistence – Amount of time between start and termination of a task
- Several tasks time-share the CPU and other resources, execution time may not equal persistence
  - Ex. Task execution time = 10ms, is interrupted for 6ms during the middle, persistence = 16ms
- OS manages resources, including CPU time, in slices to create the effect of several tasks executing concurrently
  - Cannot operate truly concurrently unless there is a multi-core processor

RTOS

- All the problems of multitasking are solved by RTOS (Real Time Operating System).
- Two types – Soft real and Hard real
- Soft Real time system computes the response as fast as possible but does not know dead line.
- Hard real time system imposes dead line on each task.





RTOS

- A thread is a — “lightweight” process, in the sense that different threads share the same address space, with all code, data, process status in the main memory, which gives Shorter creation and context switch times.
- Each thread is treated as an infinite loop.
  - Code thread 1

```
{

                    }
   Code thread 2
{

                    }
```

RTOS

- A thread is a — “lightweight” process, in the sense that different threads share the same address space, with all code, data, process status in the main memory, which gives Shorter creation and context switch times.
- Each thread is treated as an infinite loop.
  - Code thread 1

```
{

                }
Code thread 2
{

                }
```

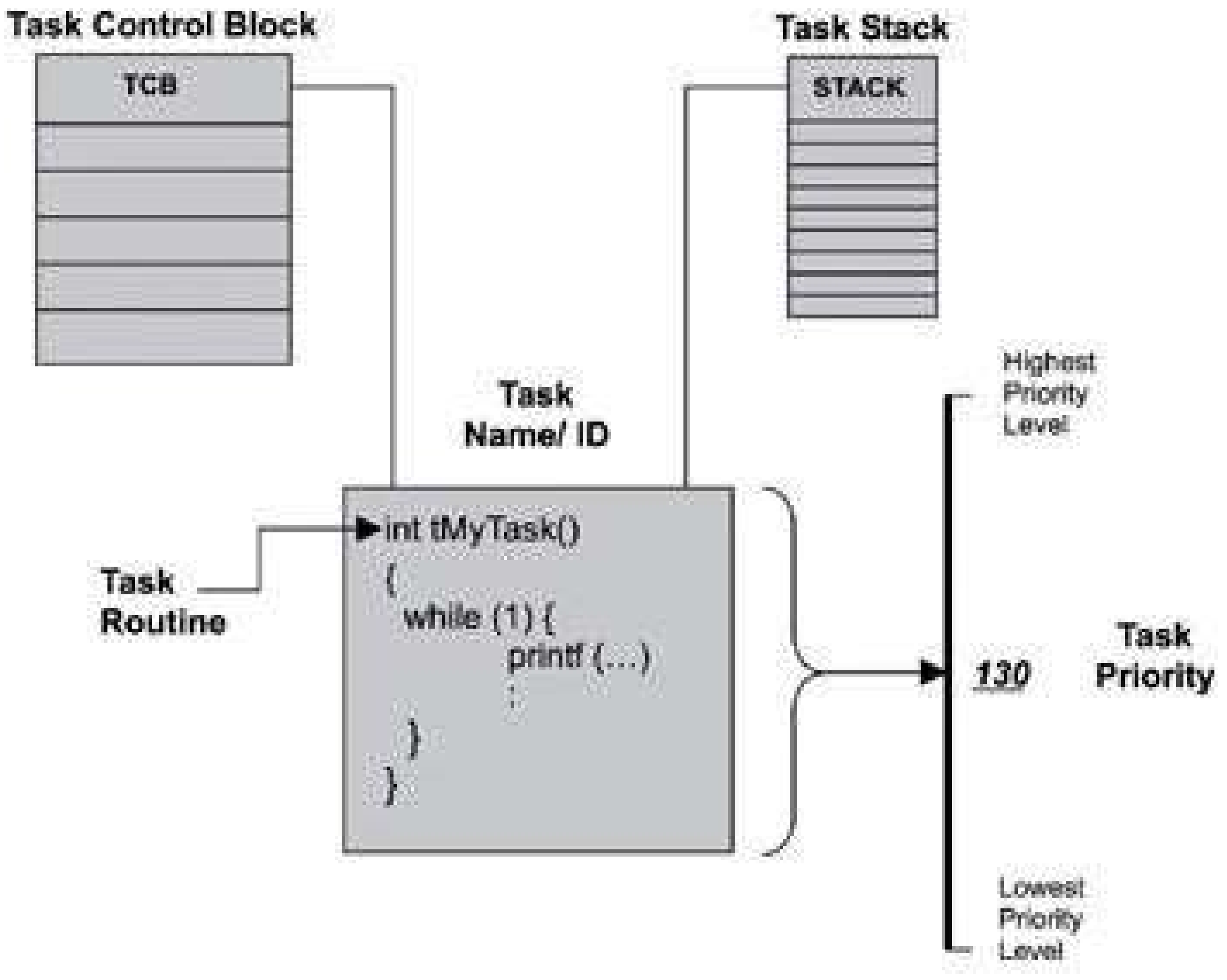
RTOS

- RTOS identifies time critical tasks.
- RTOS schedules the tasks in such a way that all the tasks are completed and meets the deadline of time critical task.
- RTOS decides when to send interrupt to the processor.
- RTOS identifies the shared resources and solves shared resources problems.
- RTOS manages inter task communication.

RTOS

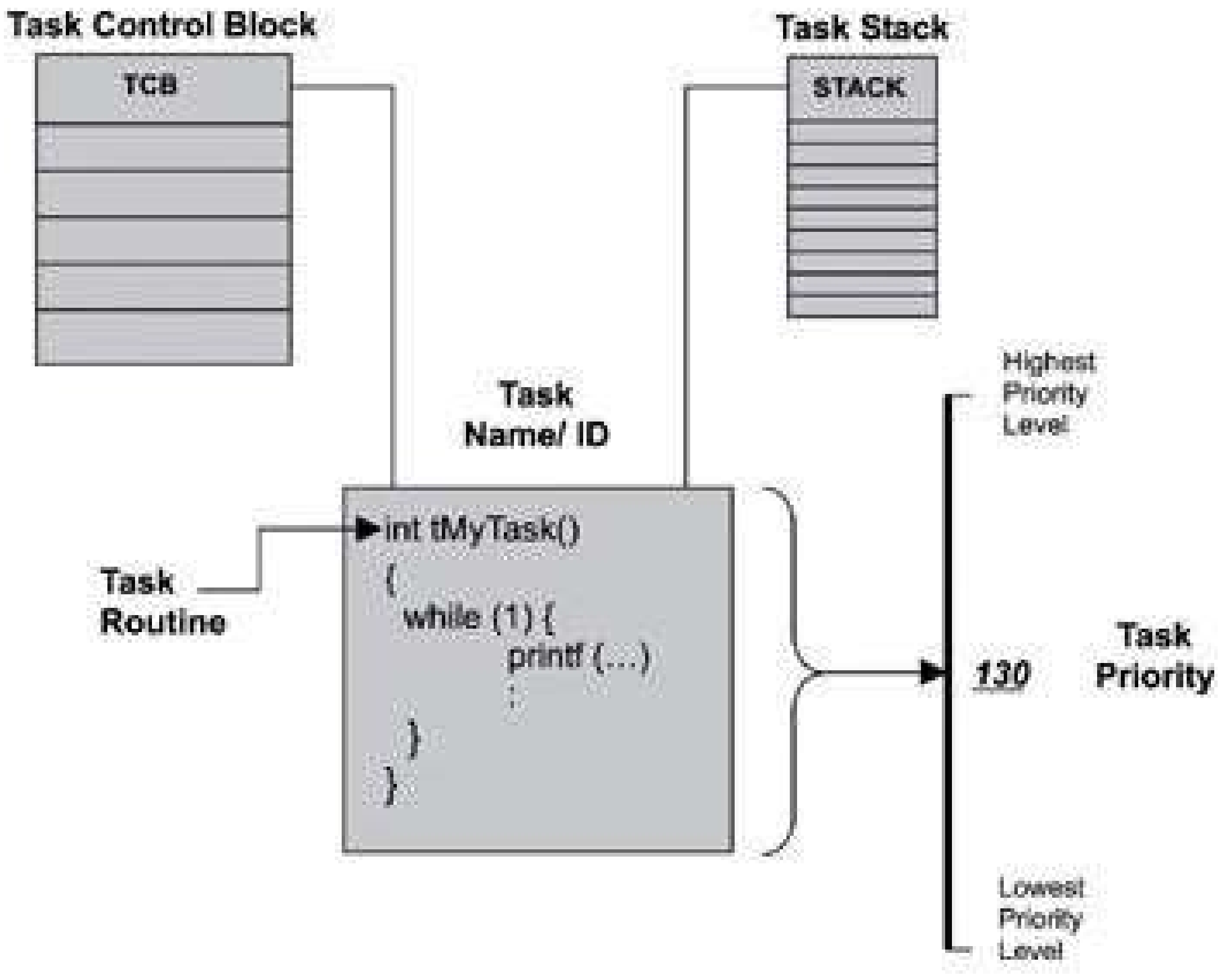
- Each task has priority
- Each task is divided into threads. Each thread runs independently but multiple threads in a task can share variables.
- Task has its own name, a unique ID, a priority, a stack, memory block or register set and a task control block (TCB) or processor control block.
- A task can generate signals, semaphores or mutex.
- Embedded software consists of OS task and application task.
- A task cannot call another task but it calls the functions to implement certain action.

Elements of a task



```
int tMyTask()
{
  while (1) {
          printf (...)
          ;
  }
}
```

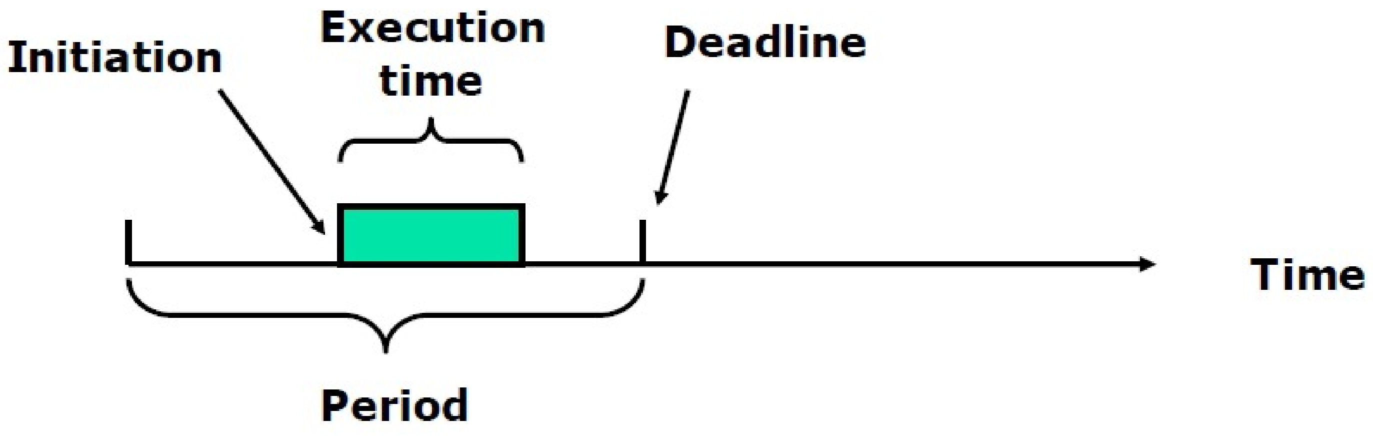
Elements of a task



```
int tMyTask()
{
  while (1) {
            printf (...)
            :
  }
}
```

Task Management

- Tasks are implemented as threads in RTOS
- Have timing constraints for tasks
- Each task a triplet: (execution time, period, deadline)
- Can be initiated any time during the period



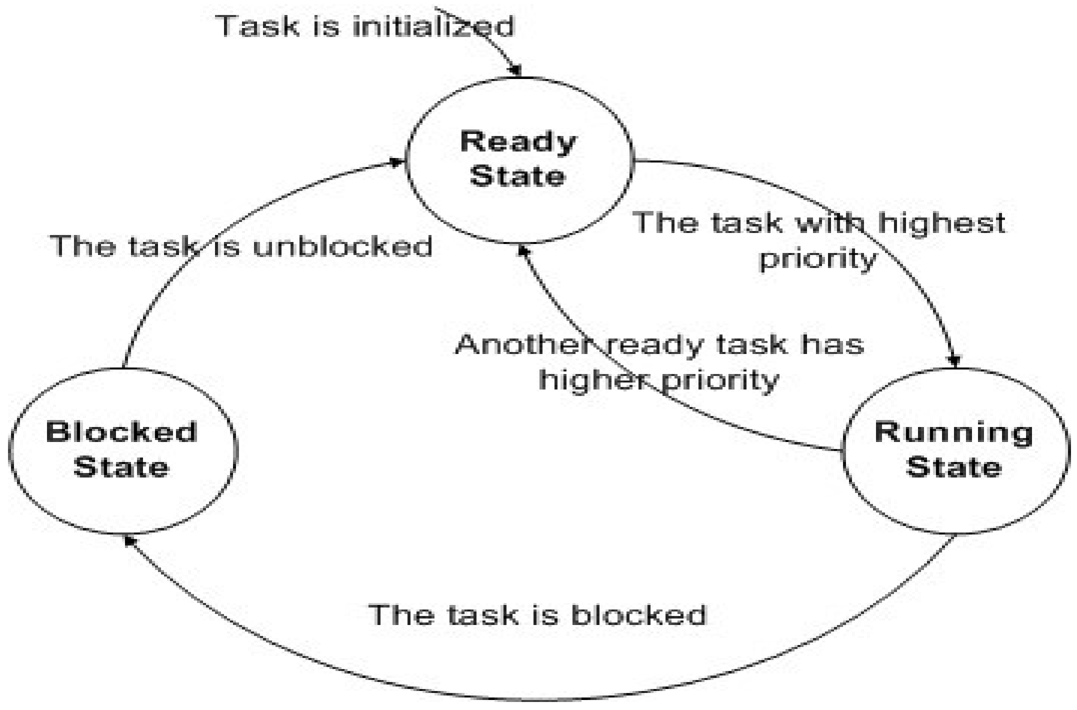
Task States

- **Idle** : task has no need for computer time
- **Ready** : task is ready to go active, but waiting for processor time
- **Running** : task is executing associated activities
- **Waiting** : task put on temporary hold to allow lower priority task chance to execute
- **suspended**: task is waiting for resource

Task States

A **task** is an independent thread of execution that can compete with other concurrent tasks for processor execution time. And applications are decomposed into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints.

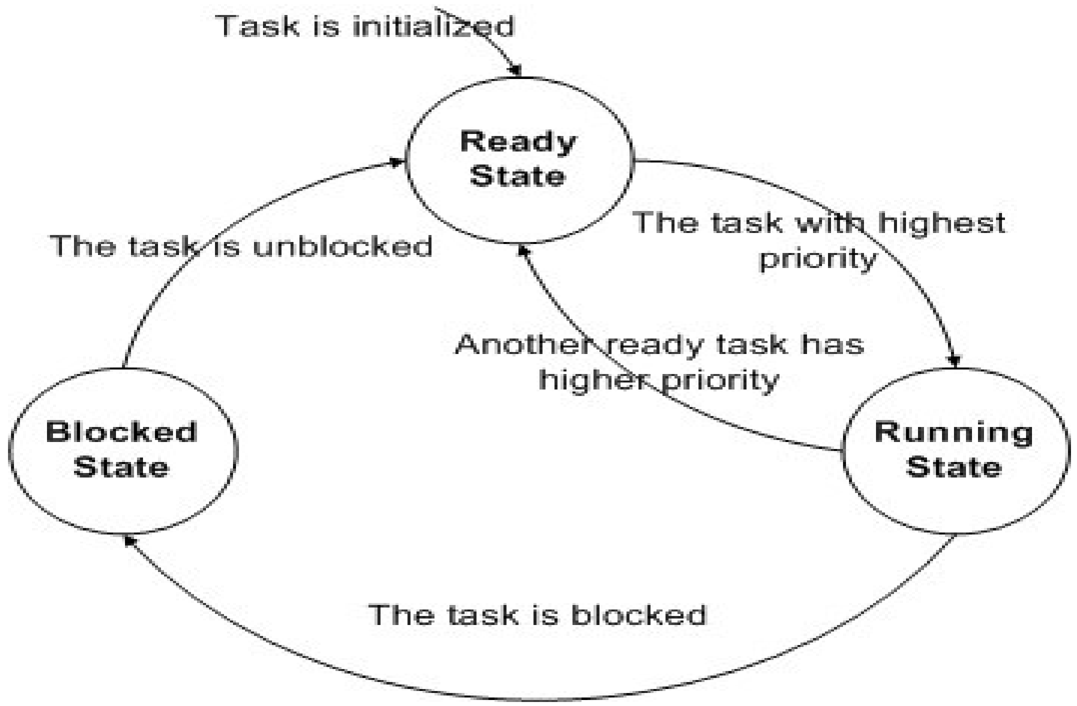
A typical finite state machine for task execution states



Task States

A **task** is an independent thread of execution that can compete with other concurrent tasks for processor execution time. And applications are decomposed into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints.

A typical finite state machine for task execution states



Task states

- **ready state**-the task is ready to run but cannot because a higher priority task is executing.
- **blocked state**-the task has requested a resource that is not available, has requested to wait until some event occurs, or has delayed itself for some duration.
- **running state**-the task is the highest priority task and is running.

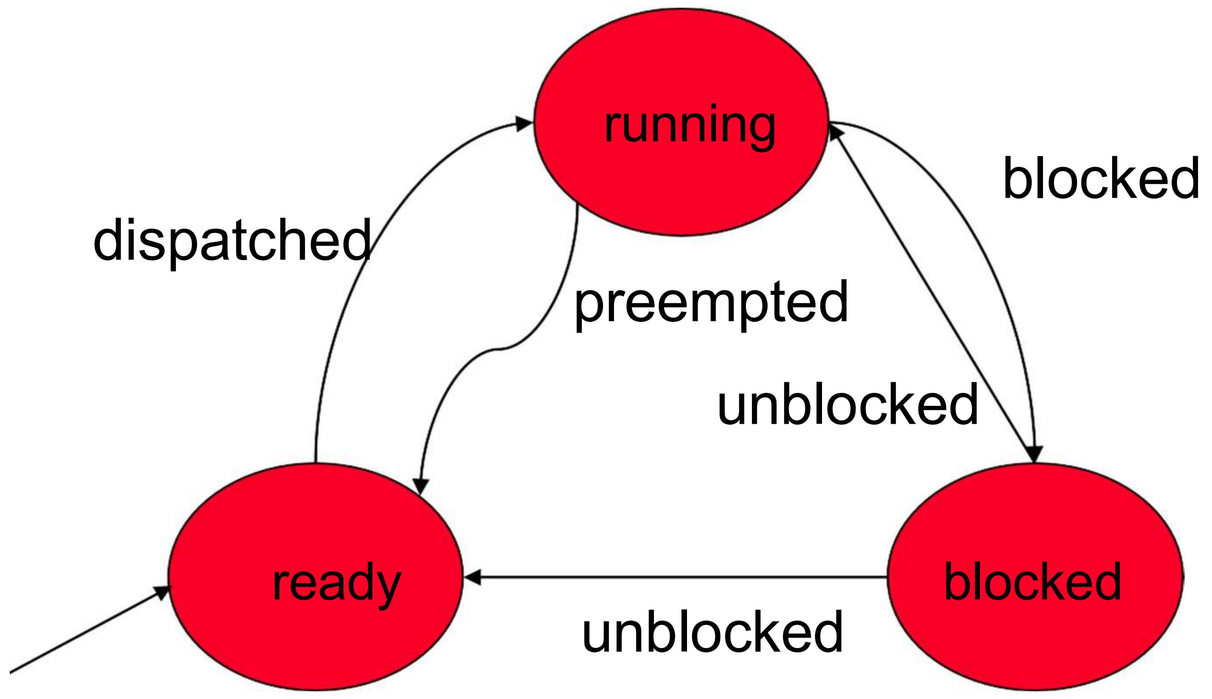
Tasks and Task States

- A task – a simple subroutine
- ES application makes calls to the RTOS functions to start tasks, passing to the OS, start address, stack pointers, etc. of the tasks
- Task States:
  - Running
  - Ready (possibly: suspended, pended)
  - Blocked (possibly: waiting, dormant, delayed)
  - [Exit]

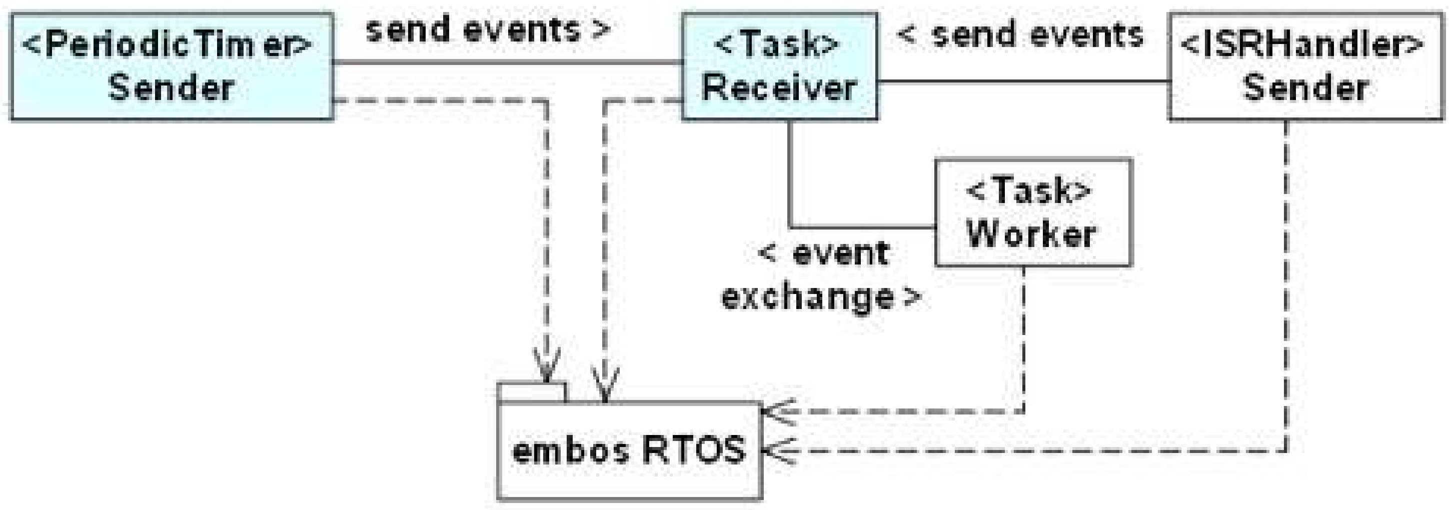
  - Scheduler – schedules/shuffles tasks between Running and Ready states
  - Blocking is self-blocking by tasks, and moved to Running state via other tasks' interrupt signaling (when block-factor is removed/satisfied)
  - When a task is unblocked with a higher priority over the 'running' task, the scheduler 'switches' context immediately (for all pre-emptive RTOSs)
- (See Fig)

Task states & scheduling

- Task states:
  - Ready state
  - Running state
  - Blocked state
- Scheduler determines each task's state.
- Task scheduling in a real-time operating system (RTOS) is the process of determining which task should be executed next, based on the system's resources and the tasks' priorities and deadlines



State Machine Example



Task scheduling

- RTOS task scheduling is critical for ensuring that real-time tasks meet their deadlines, as even a small delay in executing a task can have serious consequences.
- There are two main types of RTOS task scheduling algorithms: **preemptive** and **non-preemptive**.
  - **Preemptive scheduling** algorithms allow the RTOS to interrupt a running task and switch to a higher-priority task. This is important for ensuring that critical tasks meet their deadlines, even if lower-priority tasks are already running.
  - **Non-preemptive scheduling** algorithms do not allow the RTOS to interrupt a running task. This can be useful for tasks that need to run to completion without being interrupted, but it can also lead to missed deadlines for higher-priority tasks if a lower-priority task takes a long time to complete

Task scheduling

- In addition to preemption, RTOS task scheduling algorithms also consider the following factors:
  - **Task priority:** Each task in an RTOS is assigned a priority. Higher-priority tasks are always scheduled to run before lower-priority tasks.
  - **Task deadline:** Some RTOS task scheduling algorithms also consider the deadlines of tasks when making scheduling decisions. For example, a deadline-based scheduling algorithm might give priority to tasks that are closer to their deadlines.
  - **Task execution time:** Some RTOS task scheduling algorithms also consider the estimated execution time of tasks when making scheduling decisions. This can help to ensure that all tasks are scheduled to complete within their deadlines.
  - **Resource availability:** Some tasks may require resources, such as memory or peripherals. The RTOS scheduler will only schedule tasks if the required resources are available.

Task scheduling

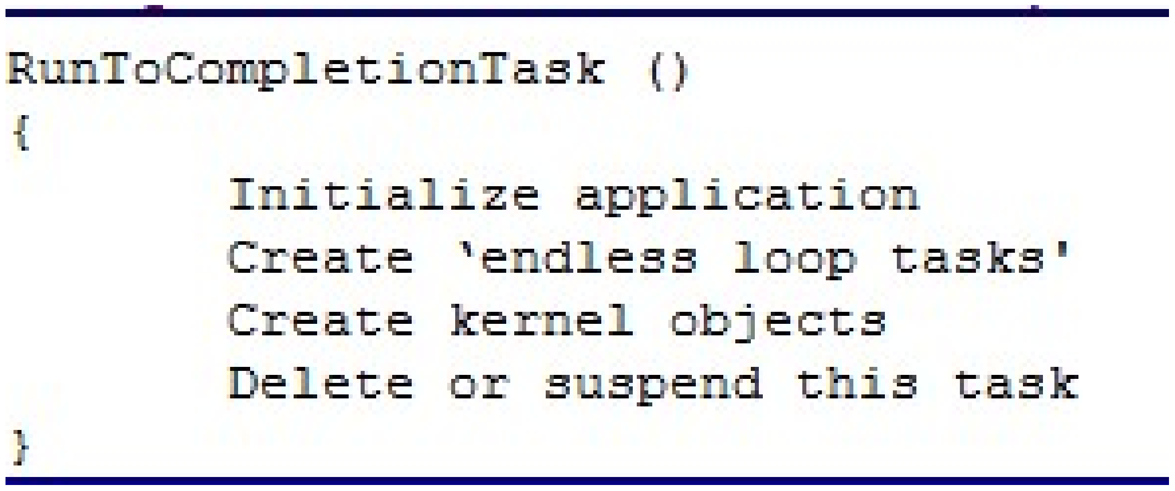
- RTOS task scheduling algorithms are complex and must be carefully designed to meet the specific needs of the real-time system. Some common RTOS task scheduling algorithms include:
  - **Priority-based scheduling:** This is the most common type of RTOS task scheduling algorithm. It simply assigns a priority to each task and schedules the task with the highest priority to run first.
  - **Round-robin scheduling:** This algorithm gives each task an equal amount of time to run, regardless of its priority. This can be useful for tasks that need to be executed periodically, but it can also lead to missed deadlines for higher-priority tasks.
  - **Deadline-based scheduling:** This algorithm schedules tasks based on their deadlines. Tasks with earlier deadlines are scheduled to run first. This algorithm can help to ensure that all tasks meet their deadlines, but it can be complex to implement.

Task scheduling

- The following is a more detailed explanation of the task scheduling process in an RTOS:
    1. **Task creation:** When an RTOS application starts up, it creates a number of tasks. Each task is assigned a priority and a stack space.
    2. **Task state:** Each task can be in one of three states: ready, running, or blocked. A task is in the ready state if it is waiting to be executed. A task is in the running state if it is currently being executed. A task is in the blocked state if it is waiting for an event to occur before it can continue executing.
    3. **Scheduling decision:** When a task becomes ready, the scheduler decides which task to run next. The scheduler typically uses the task scheduling algorithm to make this decision.
    4. **Context switching:** If the scheduler decides to run a different task, it must perform a context switch. This involves saving the state of the current task and restoring the state of the new task.
    5. **Task execution:** The scheduler starts executing the selected task. The task runs until it finishes executing, or until it is preempted by a higher priority task.
    6. **Task deletion:** When a task finishes executing, it is deleted.

Typical task structure(1)

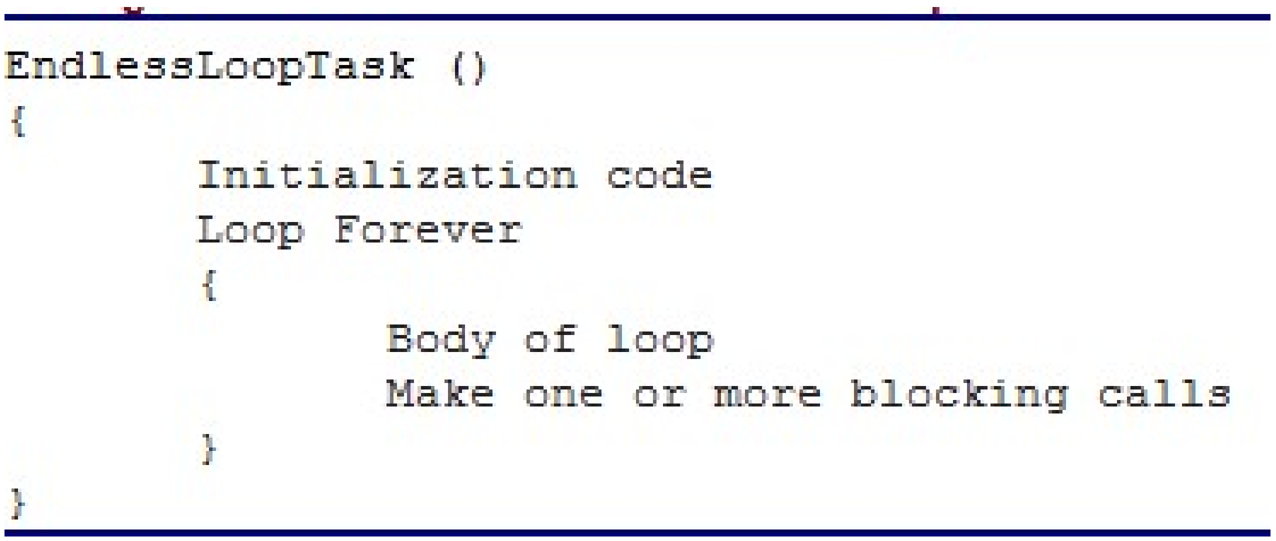
- Typical task structures:
  - Run-to-completion task: Useful for initialization & startup tasks



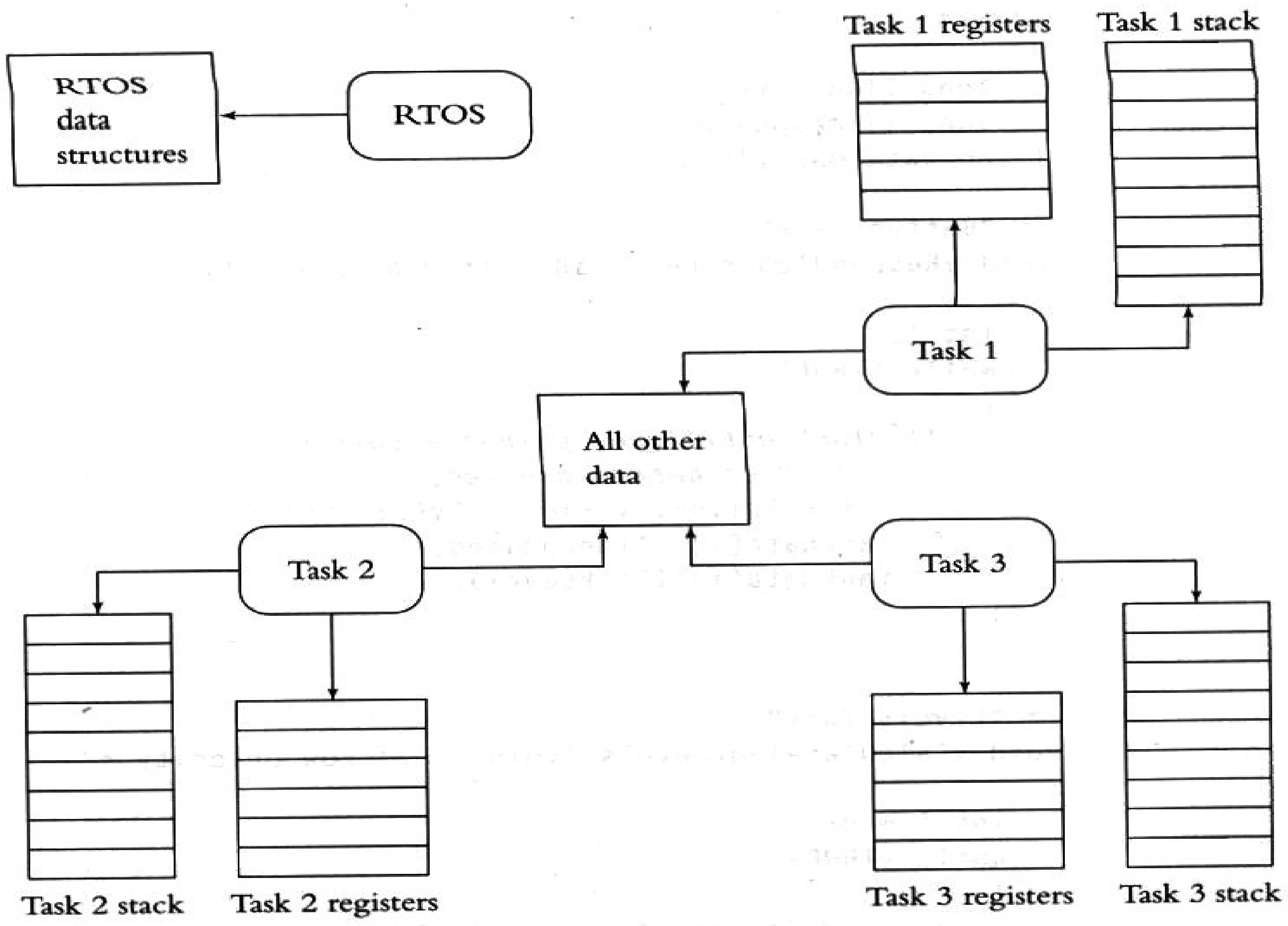
```
RunToCompletionTask ()
{
        Initialize application
        Create 'endless loop tasks'
        Create kernel objects
        Delete or suspend this task
}
```

Typical task structure(2)

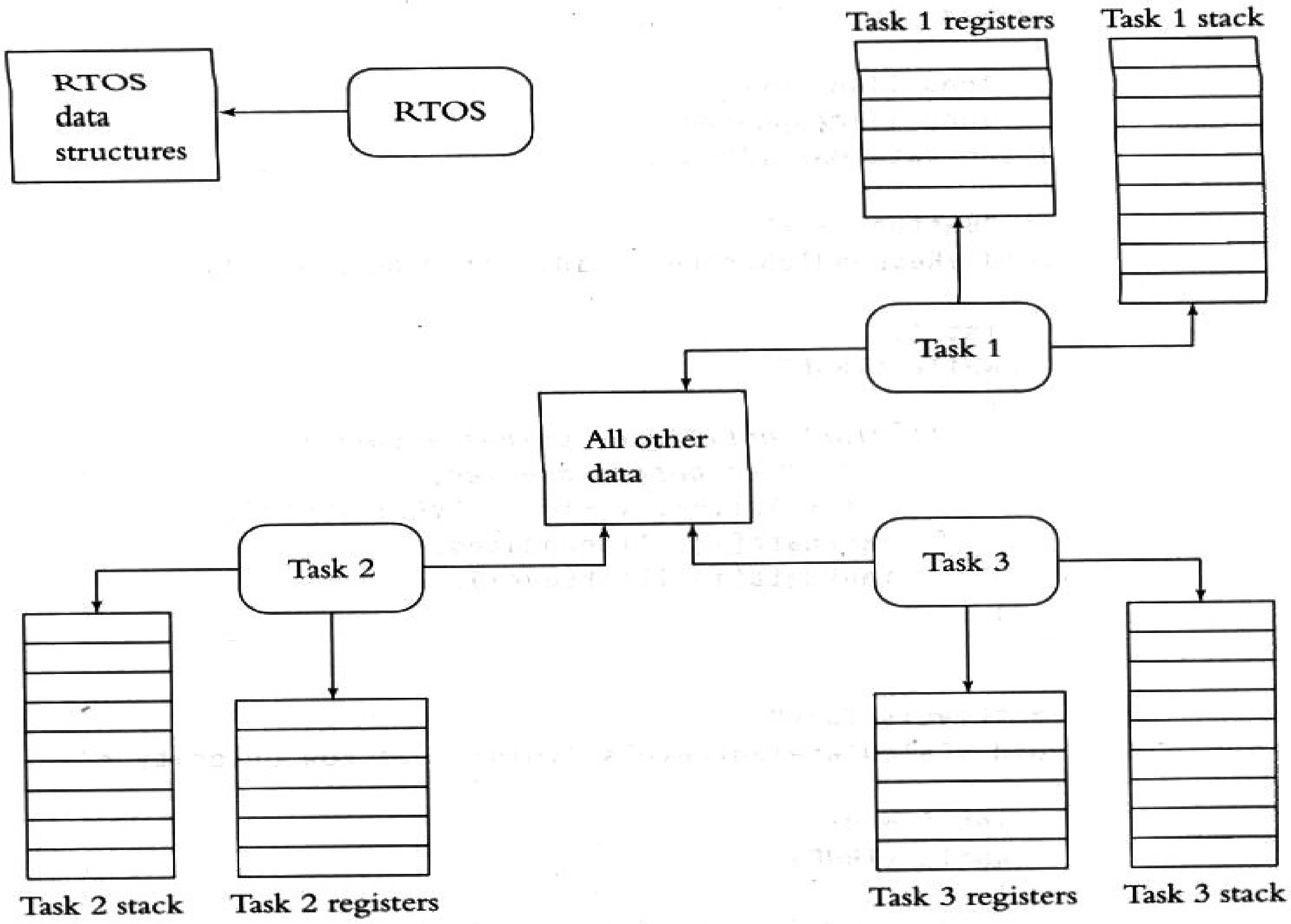
- Typical task structures:
  - Endless-loop task: Work in an application by handling inputs & outputs



```
EndlessLoopTask ()
{
        Initialization code
        Loop Forever
        {
                Body of loop
                Make one or more blocking calls
        }
}
```



Data in an RTOS-Based Real-time System



Data in an RTOS-Based Real-time System

Tasks in FREETOS

Task (Implementing) Function

```c
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance
of a task created using this function will have its own copy of the
iVariableExample variable.  This would not be true if the variable was
declared static - in which case only one copy of the variable would exist
and this copy would be shared by each created instance of the task. */
int iVariableExample = 0;

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

    /* Should the task implementation ever break out of the above loop
    then the task must be deleted before reaching the end of this function.
    The NULL parameter passed to the vTaskDelete() function indicates that
    the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

Task States

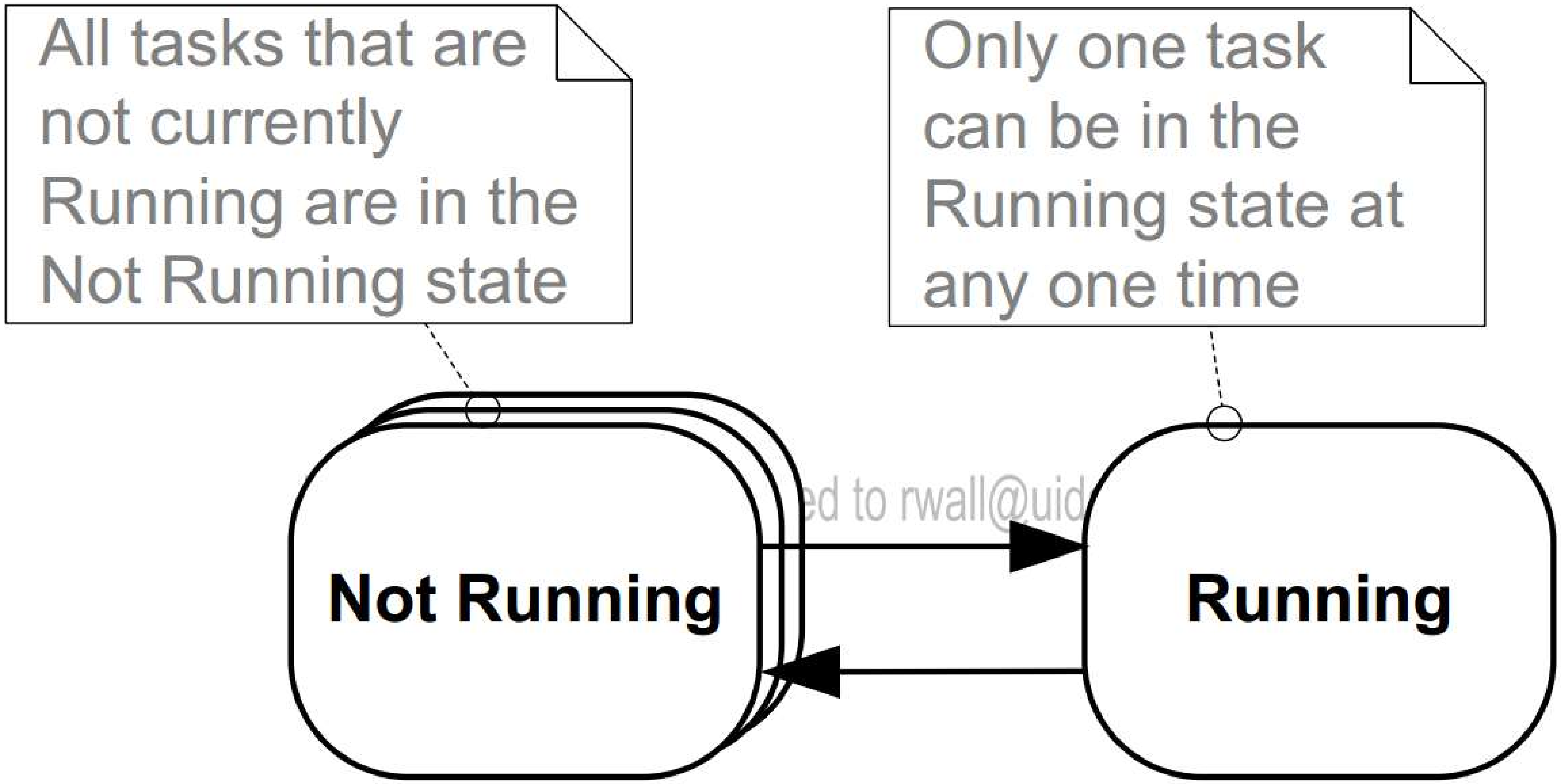

All tasks that are not currently Running are in the Not Running state
Only one task can be in the Running state at any one time
Not Running
Running
ed to rwall@uid

Figure 5. Top level task states and transitions

"Not Running" is a "Super State", encompassing multiple states

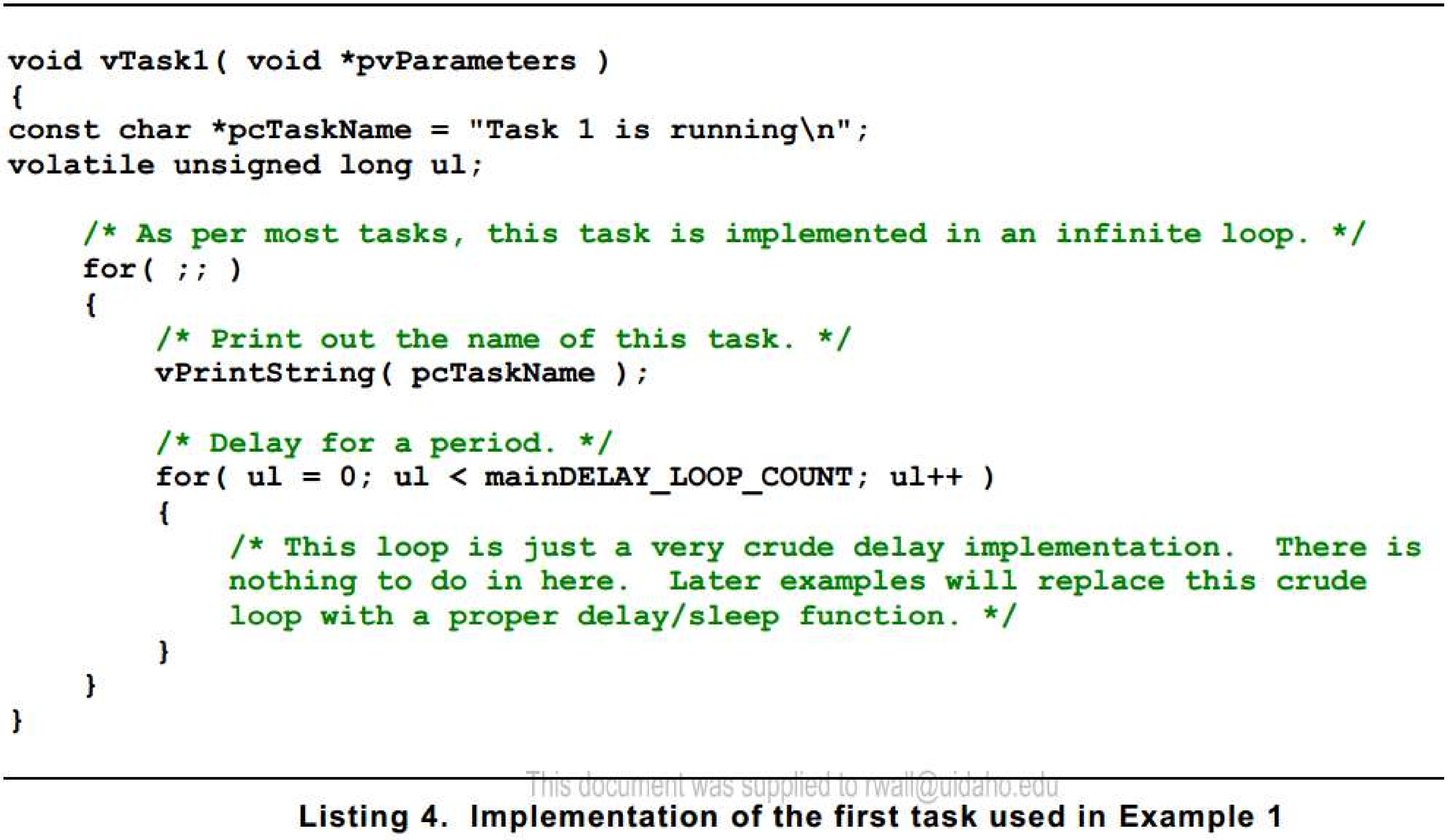
Creating a Task

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

- pvTaskCode – A pointer to the task function
- pcName – Descriptive task name used during profiling
- usStackDepth – Stack depth/size
- pvParameters – Pointer to task parameters
- uxPriority – Task priority
- pxCreatedTask – Returns a task “handle”

Returns success or failure

Example 1 – Task 1



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

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

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

This document was supplied to rwall@uidaho.edu

Listing 4. Implementation of the first task used in Example 1

Task 2

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

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

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 5. Implementation of the second task used in Example 1

Creating the Tasks

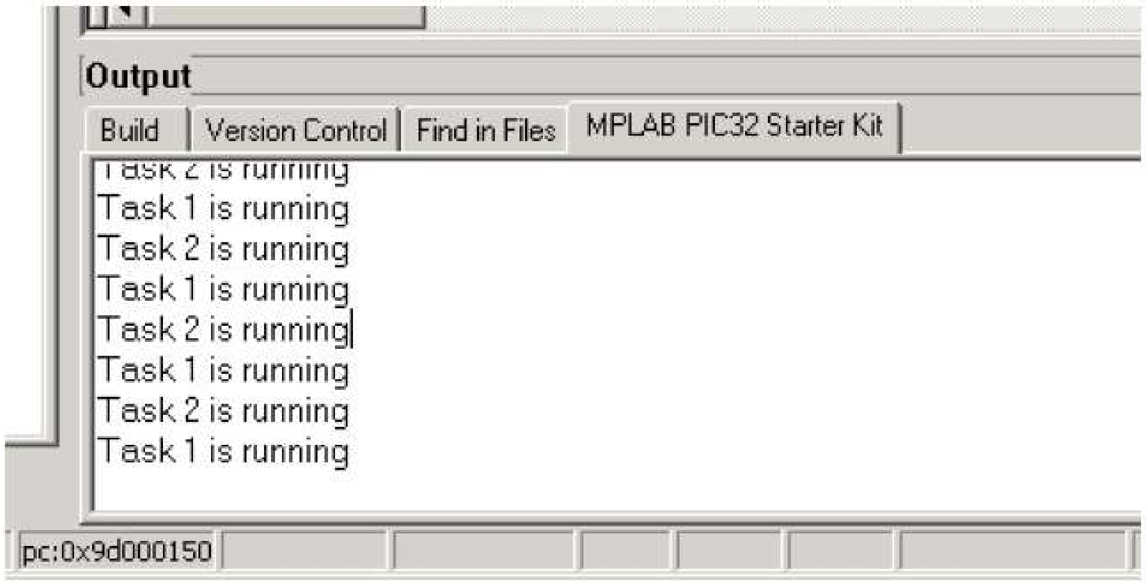
```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    240,     /* Stack depth in words. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the Idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 6. Starting the Example 1 tasks

Execution Behavior

The output generated by vPrintString() is displayed in the MPLAB IDE (assuming a PIC32 starter kit is being used as the target). Executing this example produces the output shown in Figure 6.

Single user license for ifrenzel@uidaho.edu



```
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
```

Figure 6. The output produced when Example 1 is executed

Task Execution

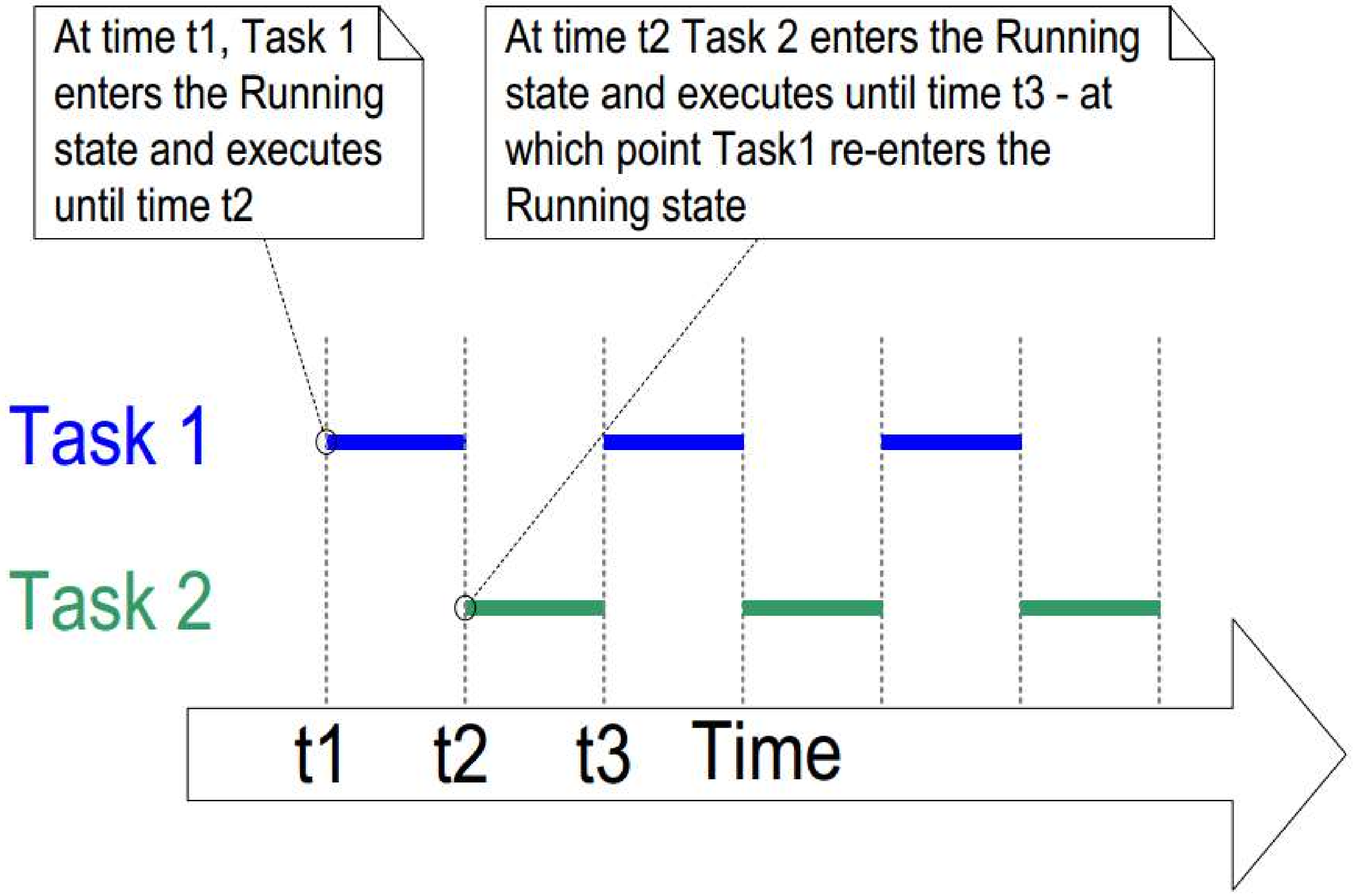

At time t1, Task 1 enters the Running state and executes until time t2
At time t2 Task 2 enters the Running state and executes until time t3 - at which point Task1 re-enters the Running state
Task 1
Task 2
t1
t2
t3
Time

Figure 7. The execution pattern of the two Example 1 tasks

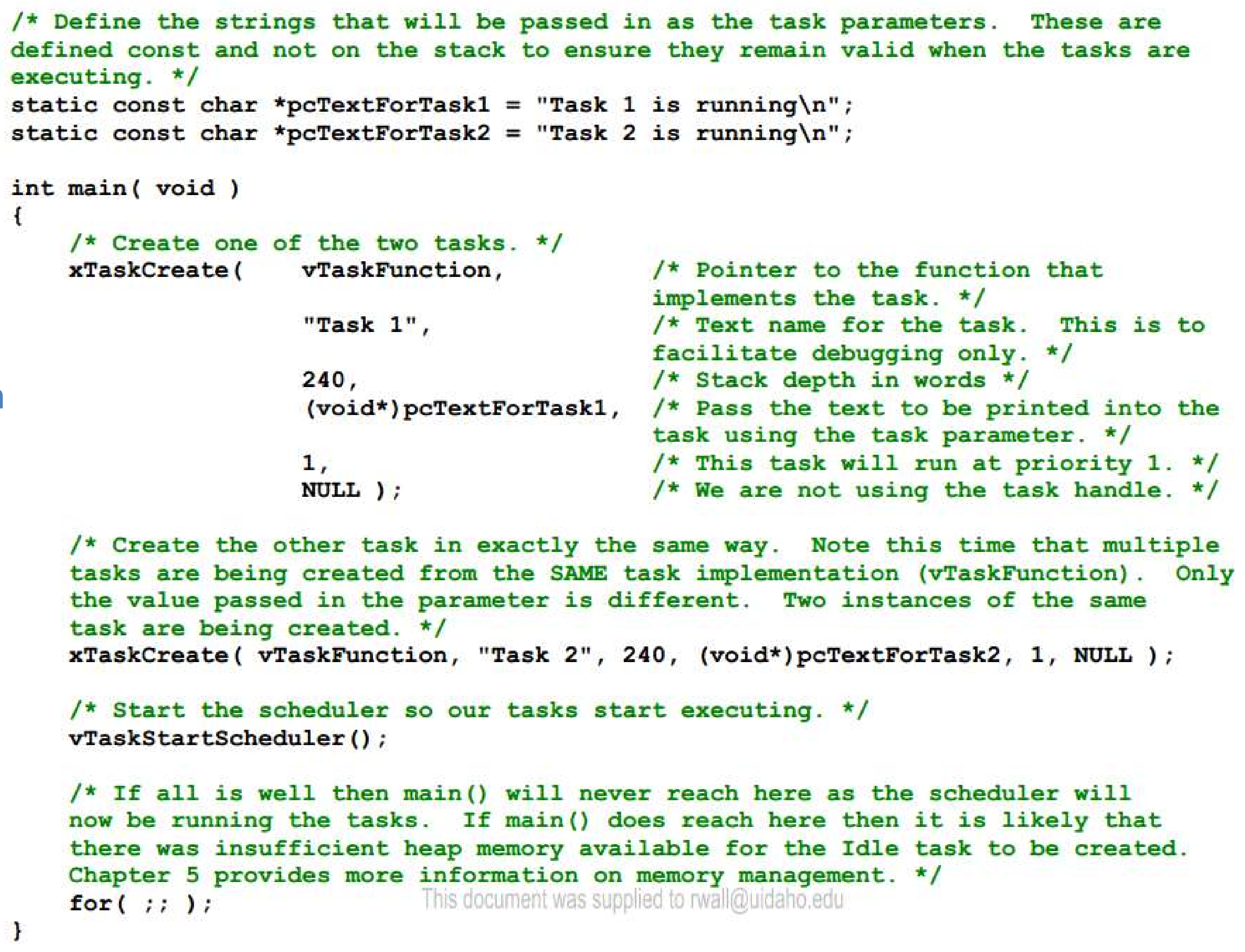
Using the Task Parameters

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

Rather than duplicate the task function, use the parameters to differentiate two <u>instances</u>

Example 2 – Task Parameters

Same behavior, but less program memory space



```
/* Define the strings that will be passed in as the task parameters.  These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\n";
static const char *pcTextForTask2 = "Task 2 is running\n";

int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(    vTaskFunction,          /* Pointer to the function that
                                            implements the task. */
                    "Task 1",               /* Text name for the task.  This is to
                                            facilitate debugging only. */
                    240,                    /* Stack depth in words */
                    (void*)pcTextForTask1,  /* Pass the text to be printed into the
                                            task using the task parameter. */
                    1,                      /* This task will run at priority 1. */
                    NULL );                 /* We are not using the task handle. */

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

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the Idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

This document was supplied to rwall@uidaho.edu

Task Priorities

- When one or more tasks of equal priority are all “Ready”, the scheduler will cycle through each one.

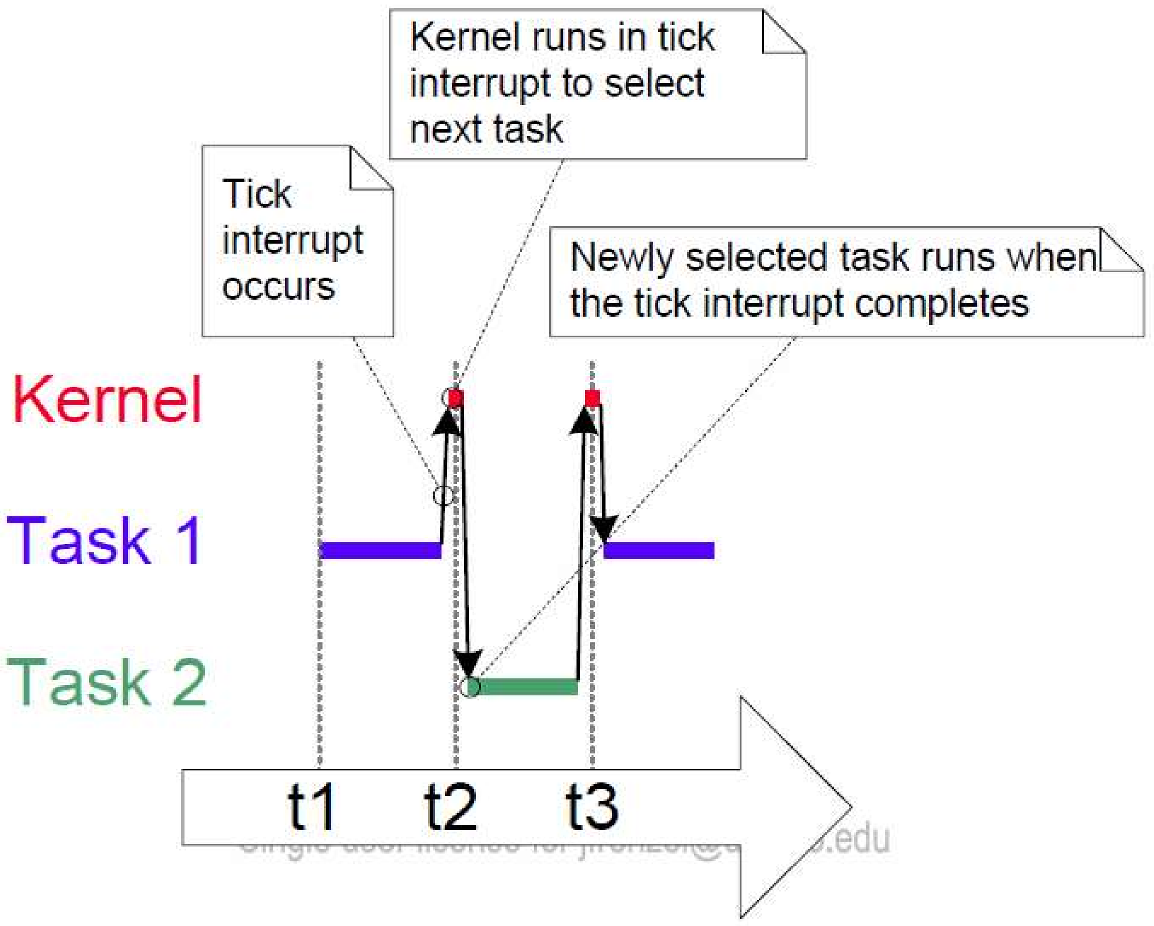

Kernel runs in tick interrupt to select next task
Tick interrupt occurs
Newly selected task runs when the tick interrupt completes
Kernel
Task 1
Task 2
t1
t2
t3

Figure 8. The execution sequence expanded to show the tick interrupt executing

Task Priorities

- On the other hand, if one of the tasks is a higher priority, it will <u>always</u> be selected …

Example 3 – Task Priority

```
/* Define the strings that will be passed in as the task parameters.  These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\n";
static const char *pcTextForTask2 = "Task 2 is running\n";

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

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

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

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

Listing 10. Creating two tasks at different priorities

Example 3 – Task Execution

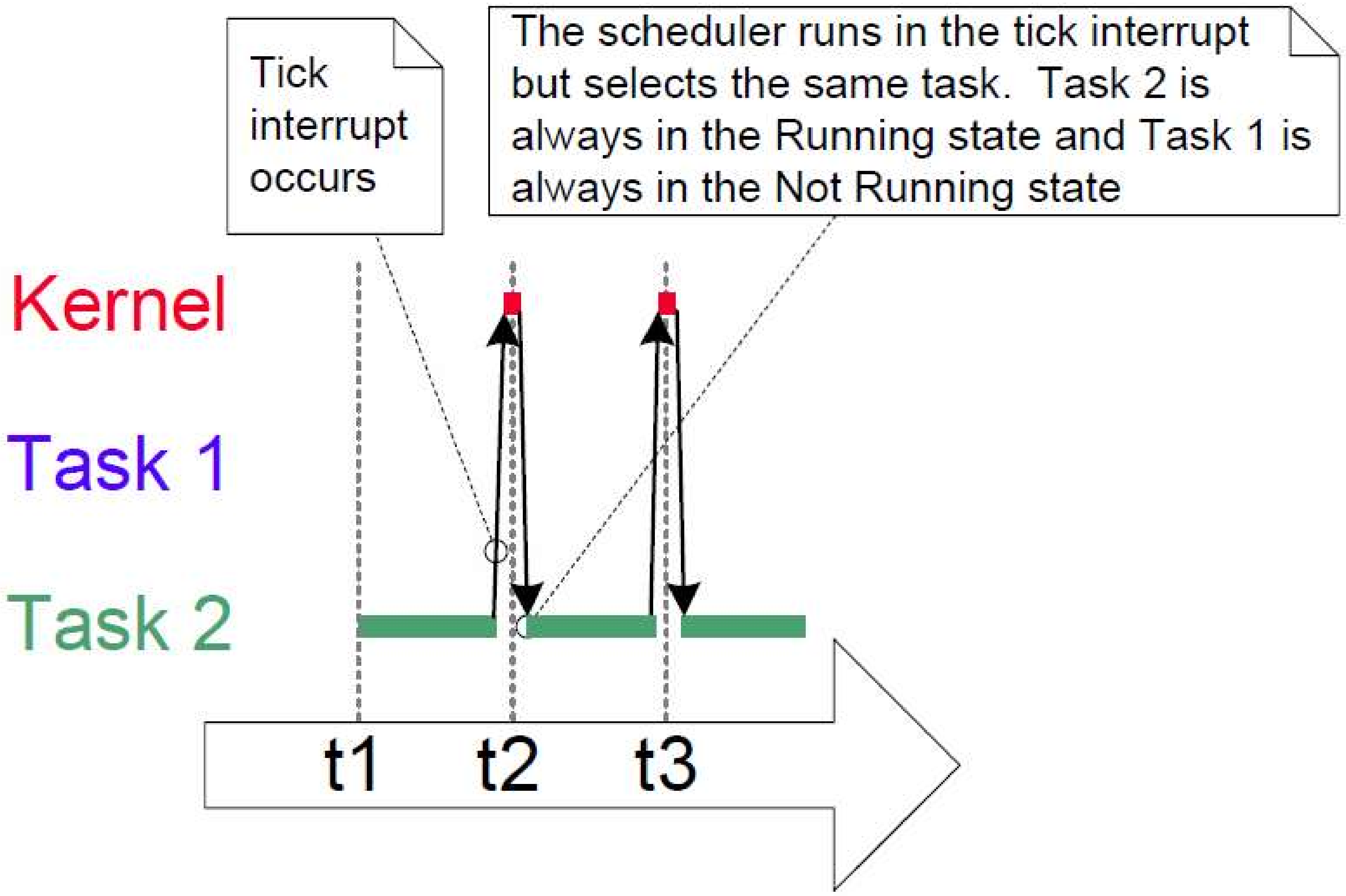

Tick interrupt occurs
The scheduler runs in the tick interrupt but selects the same task. Task 2 is always in the Running state and Task 1 is always in the Not Running state
Kernel
Task 1
Task 2
t1
t2
t3

Figure 10. The execution pattern when one task has a higher priority than the other

Task Starvation

- Even though each task has a SW delay loop, the task continues to “run” if it is in the “Running” state, polling the SW variable.
- Thus, Task 1 is “starved” because Task 2 is a higher priority and is always “Ready.”
- Need a way for Task 2 to “wait”, but not run …

“Not Running” Super State

- When “Not Running,” a task may be in one of three sub-states:
- “Ready” – The task is ready to run if selected
- “Blocked” – The task is waiting for an event
- “Suspended” – The task enters and exits this state only through API functions (rarely used)

Full Task State Machine

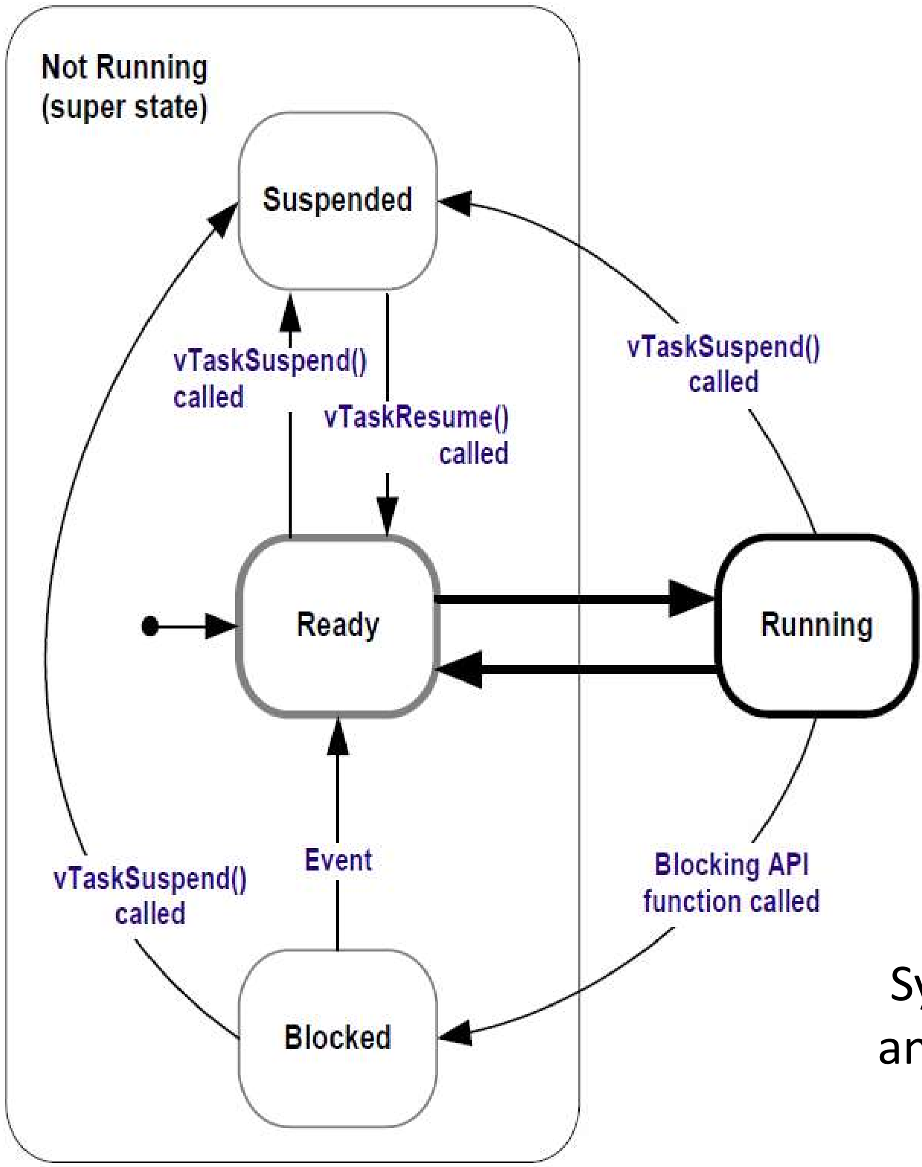


Figure 11. Full task state machine

Tasks can enter the “Blocked” state to wait for two types of events:

-Temporal Events, e.g. delaying for a fixed amount or until an absolute time

-Synchronization Events, e.g., waiting for data

Synchronization events can come in many forms and can be combined with delay events, e.g. wait for data until a time period has elapsed.

Task Delay using vTaskDelay()

```
void vTaskDelay( portTickType xTicksToDelay );
```

Listing 11. The vTaskDelay() API function prototype

Table 3. vTaskDelay() parameters

Parameter Name	Description
xTicksToDelay	The number of tick interrupts that the calling task should remain in the Blocked state before being transitioned back into the Ready state. For example, if a task called vTaskDelay(100) while the tick count was 10,000, then it would immediately enter the Blocked state and remain there until the tick count reached 10,100. The constant portTICK_RATE_MS can be used to convert milliseconds into ticks.

Task Delay using vTaskDelay()

```
void vTaskDelay( portTickType xTicksToDelay );
```

Listing 11. The vTaskDelay() API function prototype

Table 3. vTaskDelay() parameters

Parameter Name	Description
xTicksToDelay	The number of tick interrupts that the calling task should remain in the Blocked state before being transitioned back into the Ready state. For example, if a task called vTaskDelay(100) while the tick count was 10,000, then it would immediately enter the Blocked state and remain there until the tick count reached 10,100. The constant portTICK_RATE_MS can be used to convert milliseconds into ticks.

Example 4

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

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

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

        /* Delay for a period.  This time a call to vTaskDelay() is used which
        places the task into the Blocked state until the delay period has expired.
        The delay period is specified in 'ticks', but the constant
        portTICK_RATE_MS can be used to convert this to a more user friendly value
        in milliseconds.  In this case a period of 250 milliseconds is being
        specified. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Listing 12. The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()

Example 4 – Task Execution

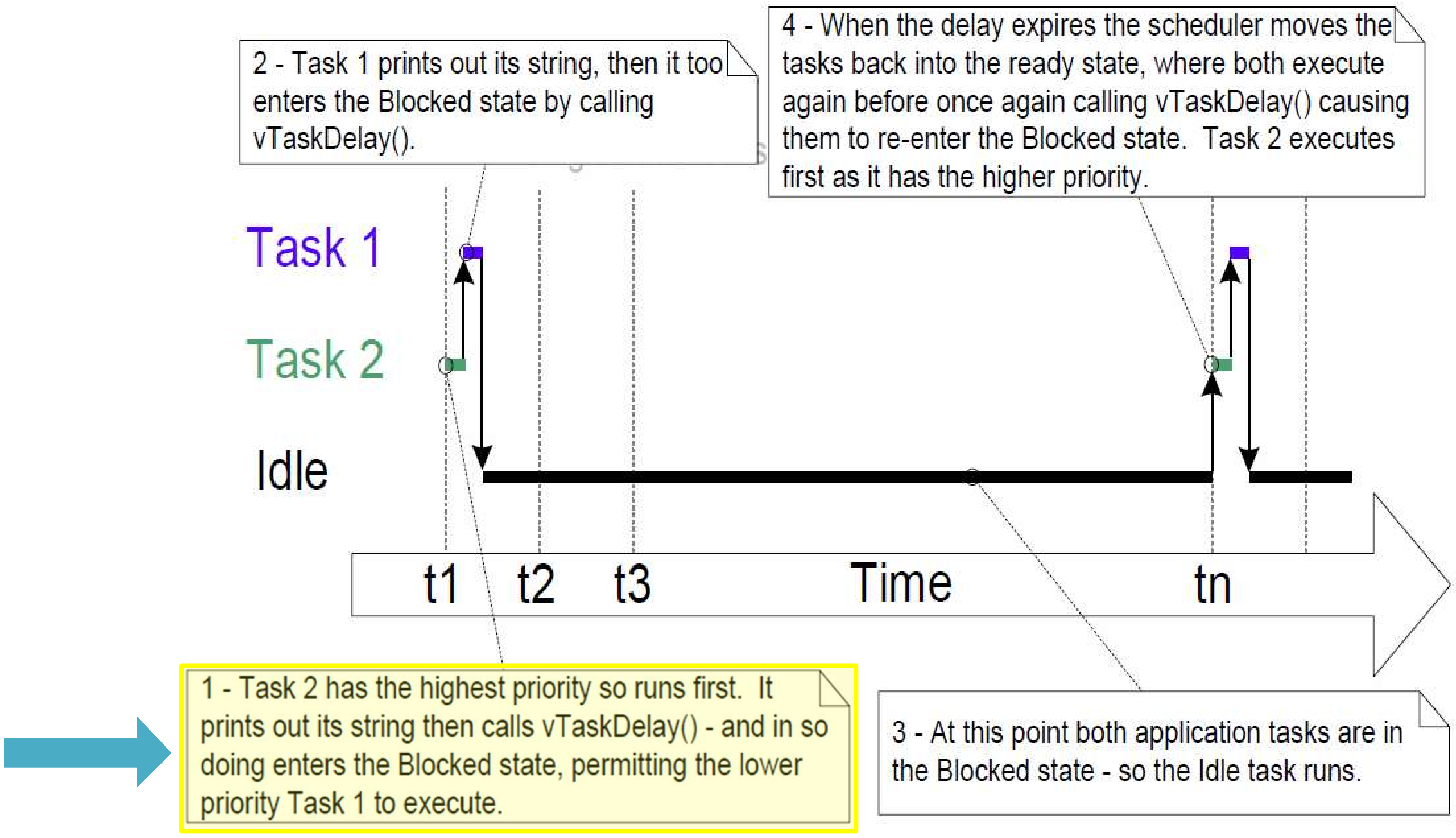

2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().
4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes 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.

Figure 13. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

The Idle Task

- Created when the scheduler starts
- Ensures “something” can always run
- Runs at the lowest priority (0)
- Later, we will add background processing to the Idle Task

Mixing Blocking/Non-Blocking Tasks

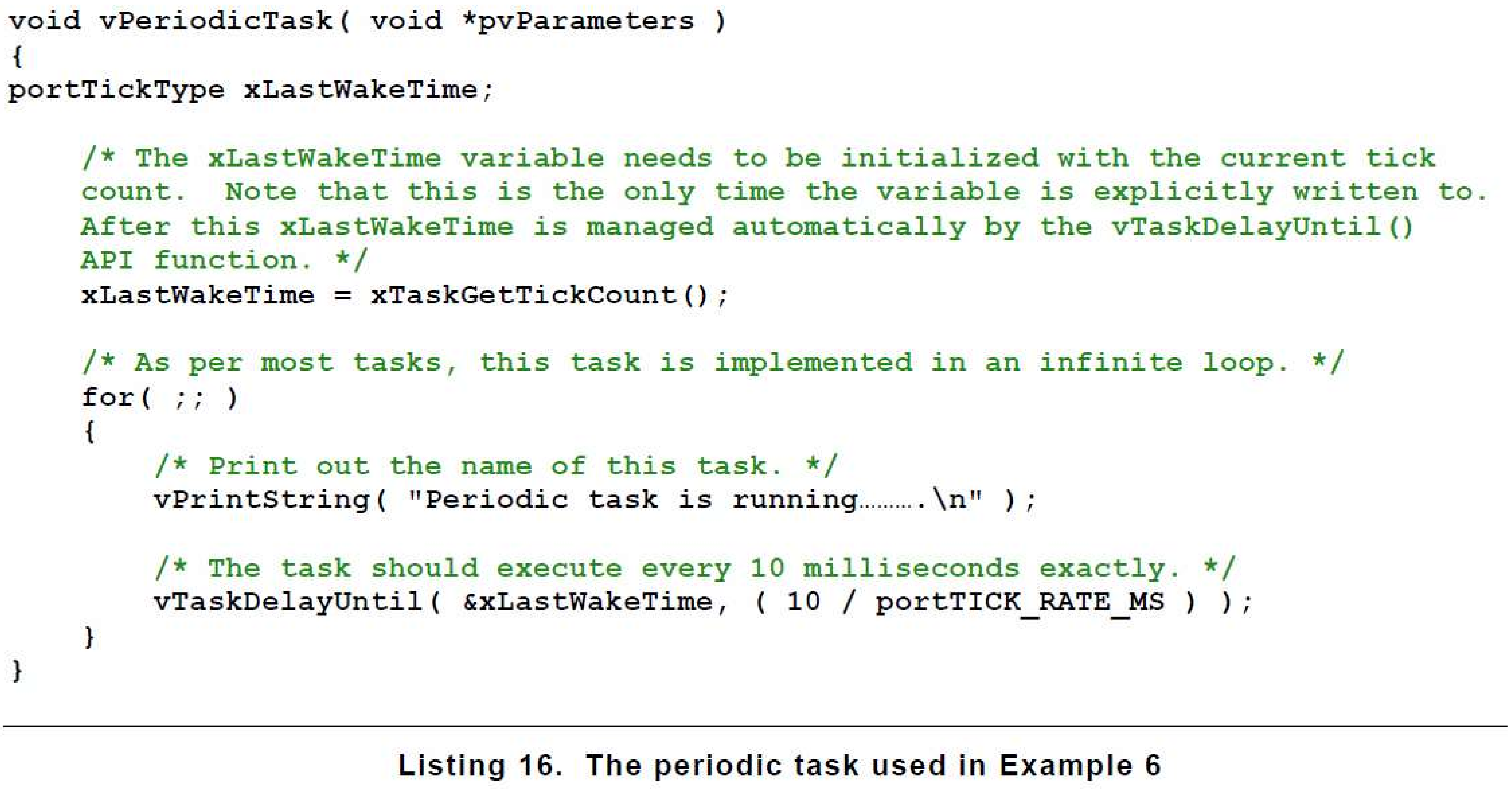
- Previous example had polling of a software variable or “blocking” for a fixed tick count
- Next example mixes “continuous” processing (always “ready” or “running”) with blocking
- Two non-blocking tasks at priority 1 and a blocking task at priority 2

Continuous Task - Example 6

```c
void vContinuousProcessingTask( void *pvParameters )
{
char *pcTaskName;

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

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task.  This task just does this repeatedly
        without ever blocking or delaying. */
        vPrintString( pcTaskName );
    }
}
```

Listing 15. The continuous processing task used in Example 6

Blocking Task – Example 6

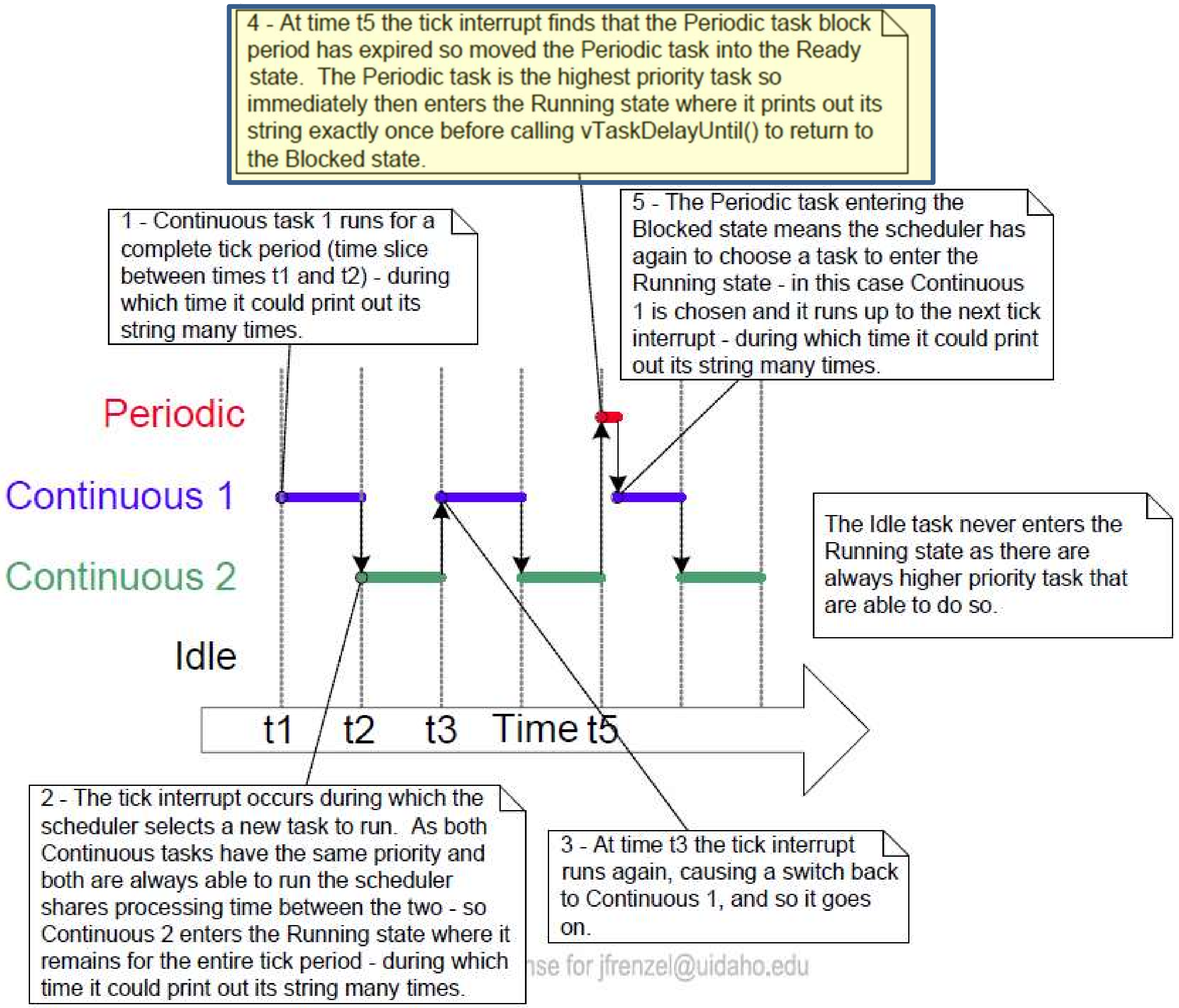


```
void vPeriodicTask( void *pvParameters )
{
portTickType xLastWakeTime;

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

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

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

Listing 16. The periodic task used in Example 6

vTaskDelayUntil() is better for accurate scheduling of periodic tasks. Why? ☺

Execution – Example 6



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.

nse for jfrenzel@uidaho.edu

Figure 16. The execution pattern of Example 6

The Idle "Hook/Callback" Function

- Must be enabled in FreeRTOSConfig.h
- Must be named vApplicationIdleHook()
- No arguments, no return value
- Must allow Idle Task to run
  - Must never block or suspend
  - If a task is deleted (vTaskDelete) then the Idle hook must return within "reasonable" time
  - This allows the Idle Task to perform "cleanup", i.e., release unneeded resources

Simple Idle Hook Function

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

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

Listing 18. A very simple Idle hook function

Task Functions

- Each task is a small program in its own right.
- It has an entry point, will normally run forever within an infinite loop, and will not exit.
- The structure of a typical task is shown in Listing 12.
- FreeRTOS tasks must not be allowed to return from their implementing function in any way—
- they must not contain a ‘return’ statement and must not be allowed to execute past the end of the function.
- If a task is no longer required, it should instead be explicitly deleted.
- This is also demonstrated in Listing 12.

Task Function

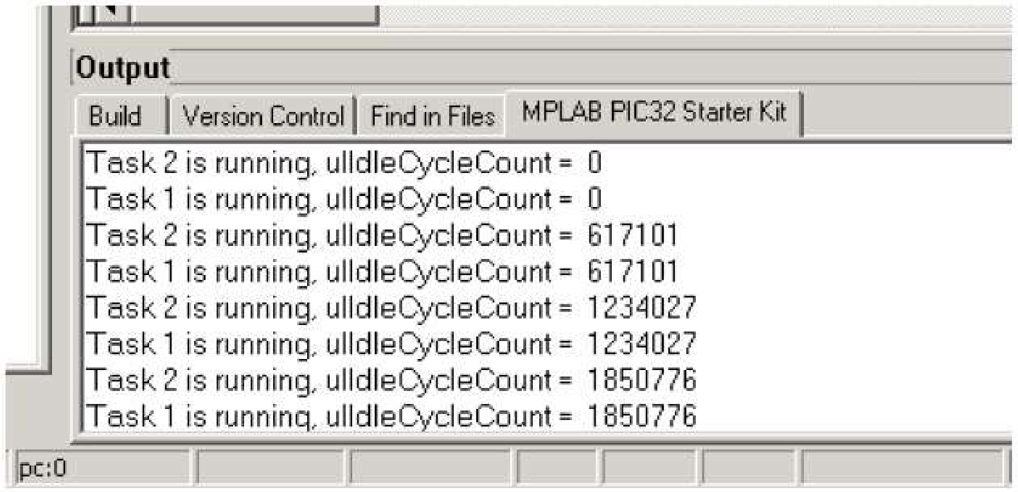
```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;

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

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task AND the number of times ulIdleCycleCount
        has been incremented. */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

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

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

Single user license for jfrenzel@uidaho.edu

Output Result – Example 7

The output produced by Example 7 is shown in Figure 17 and shows that the Idle task hook function is called approximately 617000 times between each iteration of the application tasks.



```
Task 2 is running, ulIdleCycleCount =  0
Task 1 is running, ulIdleCycleCount =  0
Task 2 is running, ulIdleCycleCount =  617101
Task 1 is running, ulIdleCycleCount =  617101
Task 2 is running, ulIdleCycleCount =  1234027
Task 1 is running, ulIdleCycleCount =  1234027
Task 2 is running, ulIdleCycleCount =  1850776
Task 1 is running, ulIdleCycleCount =  1850776
```

Figure 17. The output produced when Example 7 is executed

The Idle Hook function is called one per iteration of the Idle Task loop. The Idle Task yields at the end of every loop iteration, changing its state to “Ready” and allowing other tasks to run, even if they are the same priority. If all other tasks are blocked or suspended, then the Idle Hook function can run multiple times in one tick interval.

Miscellaneous

- A task can be created from within another task implementation function, after the scheduler has run
- Task priority can be changed through the task handle by another task, or by the task using a NULL handle
- Similarly, can query to get a task priority, or delete a task through API functions