FreeRTOS for Embedded Applications

- References:

  [1] Mastering the FreeRTOS Real Time Kernel, by Richard Barry, On-line free resource, 2016 … main reference for this section!!

  [2] FreeRTOS Reference Manual V10.0.0, On-line free resource, 2017
- FreeRTOS: designed to be small and simple; the kernel has just a few C files.
- Other FreeRTOS versions:
  - a:FreeRTOS – provided by Amazon – it is FreeRTOS with libraries for IOT support.
  - SAFERTOS: a complementary version of FreeRTOS, supporting safety-critical real-time applications.
  - OPENRTOS: a commercially-licensed version of Amazon FreeRTOS.

The Top Directories in the FreeRTOS Distribution ([1] Chapter 1)

- FreeRTOS zip file contains source code for all the FreeRTOS ports, and project files for all the FreeRTOS demo applications
- The core FreeRTOS source code is contained in just two C files that are common to all the FreeRTOS ports: tasks.c, and list.c

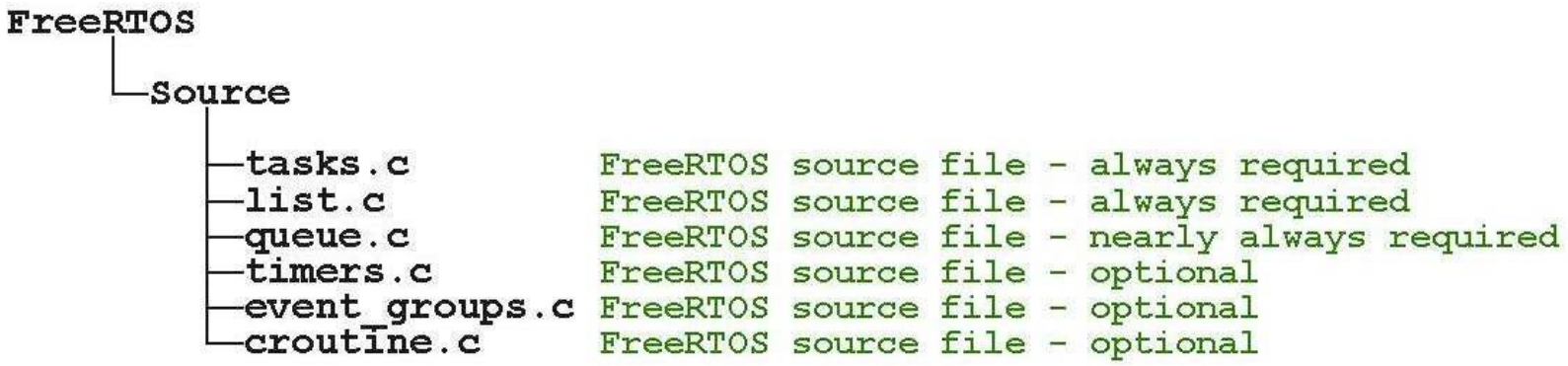


Figure 2. Core FreeRTOS source files within the FreeRTOS directory tree

Creating a FreeRTOS Project

- Every FreeRTOS port comes with at least one pre-configured demo application that should build with no errors or warnings.
- It is recommended that new projects are created by adapting one of these existing projects

To start a new application from an existing demo project:

1. Open the supplied demo project and ensure that it builds and executes as expected.
2. Remove the source files that define the demo tasks. Any file that is located within the Demo/Common directory can be removed from the project.
3. Delete all the function calls within main(), except prvSetupHardware() and vTaskStartScheduler(), as shown in Listing 1.
4. Check the project still builds.

Following these steps will create a project that includes the correct FreeRTOS source files, but does not define any functionality.

```
int main( void )
{
    /* Perform any hardware setup necessary. */
    prvSetupHardware();

    /* --- APPLICATION TASKS CAN BE CREATED HERE --- */

    /* Start the created tasks running. */
    vTaskStartScheduler();

    /* Execution will only reach here if there was insufficient heap to
    start the scheduler. */
    for( ;; );
    return 0;
}
```

Listing 1. The template for a new main() function

Port specific data types used by FreeRTOS

- FreeRTOS configures a periodic interrupt called the tick interrupt
- For each port, the portmacro.h header contains two specific data types:

1. TickType_t: data type used to hold the tick count value, and to specify times.
2. BaseType_t: defines the base data type of the architecture used.
   - 32-bit type for a 32-bit architecture; 16-bit type for a 16-bit architecture and 8-bit type for an 8-bit architecture
   - Generally, it is used for return types that can take only a very limited range of values, and for pdTRUE/pdFALSE type Booleans

Variable Names

- Variables are prefixed with their type:
  - ‘c’ for char, ‘s’ for int16_t (short), ‘l’ int32_t (long), and ‘x’ for BaseType_t and any other non-standard types (structures, task handles, queue handles, etc.).
- If a variable is unsigned, it is also prefixed with a ‘u’.
- If a variable is a pointer, it is also prefixed with a ‘p’.
  - For example, a variable of type uint8_t will be prefixed with ‘uc’, and a variable of type pointer to char will be prefixed with ‘pc’.

Function Names

- Functions are prefixed with both the type they return, and the file they are defined within.
  - For example: vTaskPrioritySet() returns a void and is defined within task.c.
  - xQueueReceive() returns a variable of type BaseType_t and is defined within queue.c.
  - pvTimerGetTimerID() returns a pointer to void and is defined within timers.c.
- File scope (private) functions are prefixed with ‘prv’.

Macros

Most macros are written in upper case, and prefixed with lower case letters that indicate where the macro is defined

Table 3. Macro prefixes

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h or portmacro.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Heap Memory Management ([1] Chapter 2)

- Dynamic memory allocation is a C programming concept, and not a concept that is specific to either FreeRTOS or multitasking.
- FreeRTOS allocates RAM each time a kernel object is created, and frees RAM each time a kernel object is delete
- FreeRTOS provides pvPortMalloc() function to replace malloc() and vPortFree() to replace free()
  - Dynamic memory allocation schemes provided by general purpose compilers are not always suitable for real-time applications.
- This policy reduces design and planning effort, simplifies the API, and minimizes the RAM footprint.

Heap Memory Management

- This topic objective is to give the readers a good understanding of:
  - When FreeRTOS allocates RAM.
  - The five example memory allocation schemes supplied with FreeRTOS.
  - Which memory allocation scheme to select.

Heap Memory Management

- FreeRTOS treats memory allocation as part of the portable layer (as opposed to part of the core code base)
  - Removing dynamic memory allocation from the core code base allow for custom setup in various embedded systems
- When FreeRTOS requires RAM, instead of calling malloc(), it calls pvPortMalloc().
- When RAM is being freed, instead of calling free(), the kernel calls vPortFree().
- FreeRTOS comes with five heap example implementations (See Ch. 2).
  - An applications can use one of the example code, or provide their own.
  - The five examples are defined in the heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap_5.c source files respectively, all of which are located in the FreeRTOS/Source/portable/MemMang directory.

Heap_1 Scheme

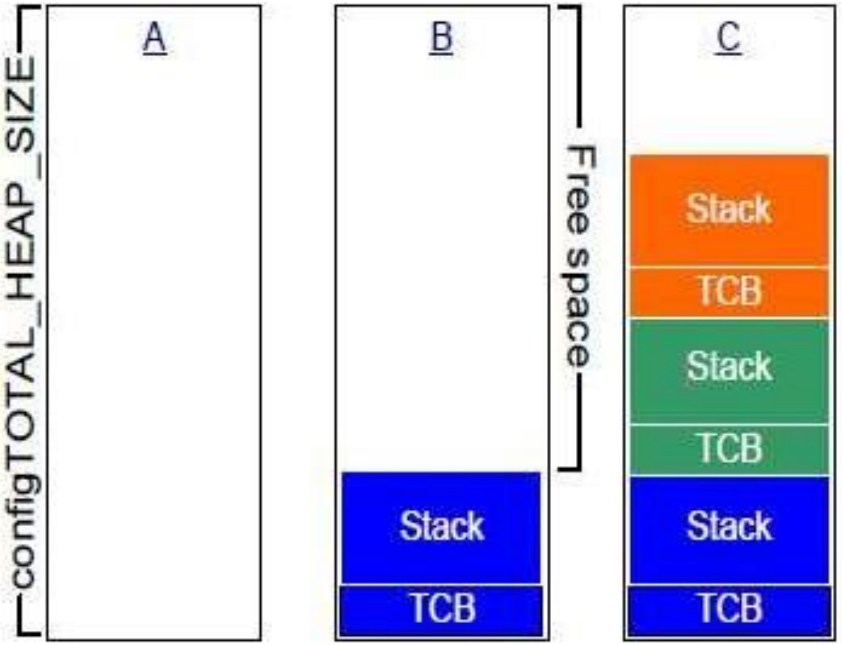
- Heap_1.c implements a very basic version of pvPortMalloc(), and does not implement vPortFree().
- Applications that could use this scheme:
  - Small system applications that never delete a task, or other kernel object
  - Some commercially critical and safety critical systems
    - Critical systems often prohibit dynamic memory allocation due to uncertainties associated with non-determinism, memory fragmentation, and failed allocations—but Heap_1 is always deterministic, and cannot fragment memory.
- The heap_1 allocation scheme subdivides a simple array into smaller blocks as calls to pvPortMalloc() are made
  - the array (called the FreeRTOS heap) is statically declared, so will make the application appear to consume a lot of RAM

Heap_1 Structure

The total size (in bytes) of the array is set by the *definitionconfigTOTAL_HEAP_SIZE* within FreeRTOSConfig.h.

Each created task requires:

1. a task control block (TCB) and
2. a stack



RAM being allocated from the heap_1 array each time a task is created

- A shows the array before any tasks have been created—the entire array is free.
- B shows the array after one task has been created.
- C shows the array after three tasks have been created.

Heap_2 and Heap_3 Structures

- Heap_2 is retained in the FreeRTOS distribution for backward compatibility, but its use is not recommended for new designs
- Heap_2 works by subdividing an array into smaller blocks but allows memory to be freed
  - Uses best fit algorithm to allocate memory to ensure that pvPortMalloc() choses the best block size that is closest in size to the number of bytes requested
  - The scheme is not deterministic
- Heap_3 uses the standard library malloc() and free() functions
  - So, the size of the heap is defined by the linker configuration, and the configTOTAL_HEAP_SIZE setting has no affect

Heap_4 Scheme

Applications: applications that repeatedly allocate and free different sized blocks of RAM.

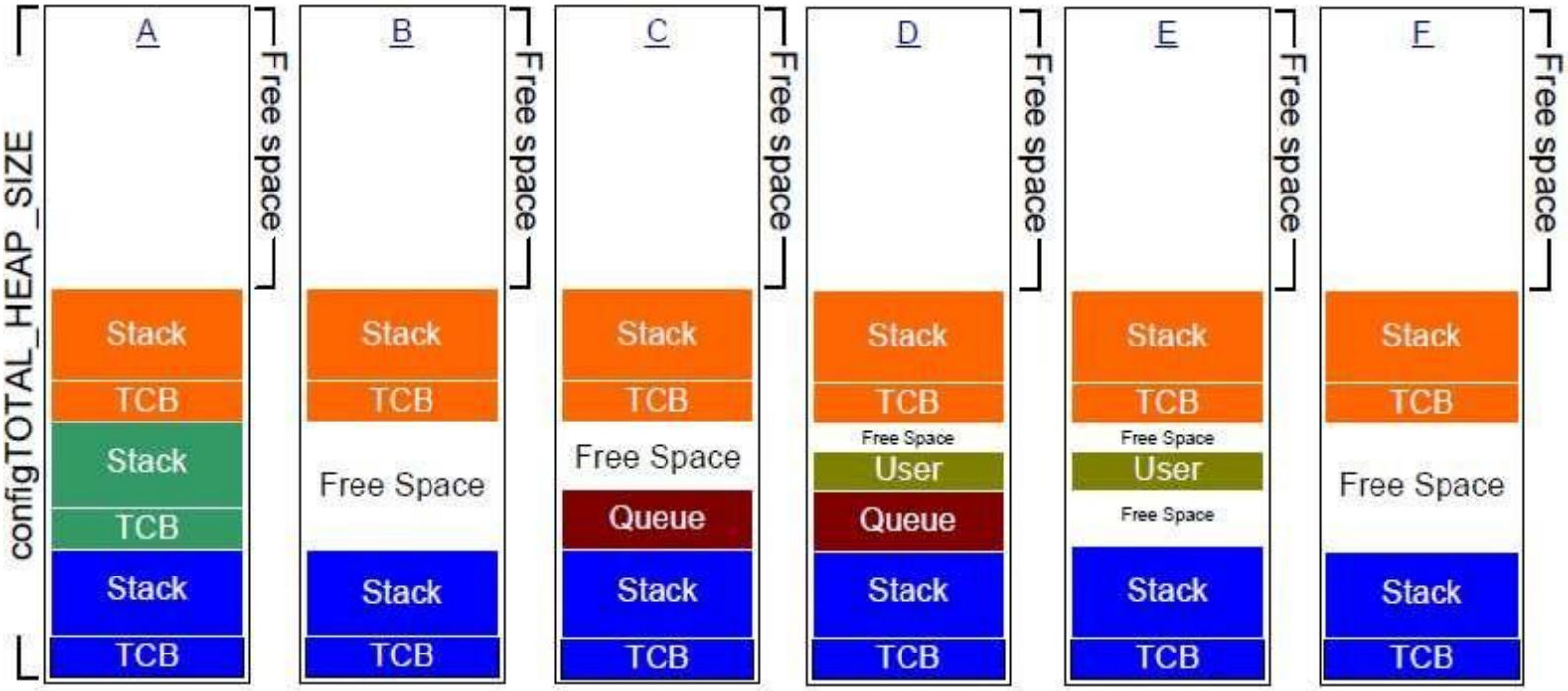
- Heap_4 works by subdividing an array into smaller blocks.
  - The initial array is statically declared, and dimensioned by configTOTAL_HEAP_SIZE,
  - This will make the application appear to consume a lot of RAM
- Heap_4 uses a first fit algorithm to allocate memory but unlike heap_2, heap_4 joins adjacent free blocks of memory into a single larger block, which minimizes the risk of memory fragmentation.
- The first fit algorithm ensures pvPortMalloc() uses the first free block of memory that is large enough to hold the number of bytes requested

Heap_4 Example Scenario

- Scenario:
  - The heap contains three blocks of free memory in the order:
    - 5 bytes, 200 bytes, and 100 bytes, respectively.
  - Then, pvPortMalloc() is called to request 20 bytes of RAM.
- How it works?
  - The first free block of RAM into which the requested number of bytes will fit is the 200-byte block
    - So pvPortMalloc() splits the 200-byte block into one block of 20 bytes, and one block of 180 bytes, before returning a pointer to the 20-byte block.
    - The new 180-byte block remains available to future calls to pvPortMalloc().

Heap_4 Scheme

(A) …;
(B) …;
(C) …;
(D) …;
(E) …;
(F) …



- Heap_4 combines adjacent free blocks into a single larger block, minimizing the risk of fragmentation
- Note: Heap_5 scheme is similar to heap_4 scheme.

Task Management ([1] Chapter 3)

- This section covers:
  - 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.
  - How to implement tasks and other task related concepts
- The concepts in this chapter are fundamental to understanding how to use FreeRTOS in an application

Task Management: Task Functions

- Tasks in FreeRTOS are implemented as C functions
- Important about the task prototype is that it must return void and take a void pointer parameter:

```
void ATaskFunction(void *pvParameters);
```

Task features:

- It is a small program on its own
- It has an entry point
- Will normally run as an infinite loop, and will not exit
- Must not contain a 'return' statement and must not be allowed to execute past the end of the function

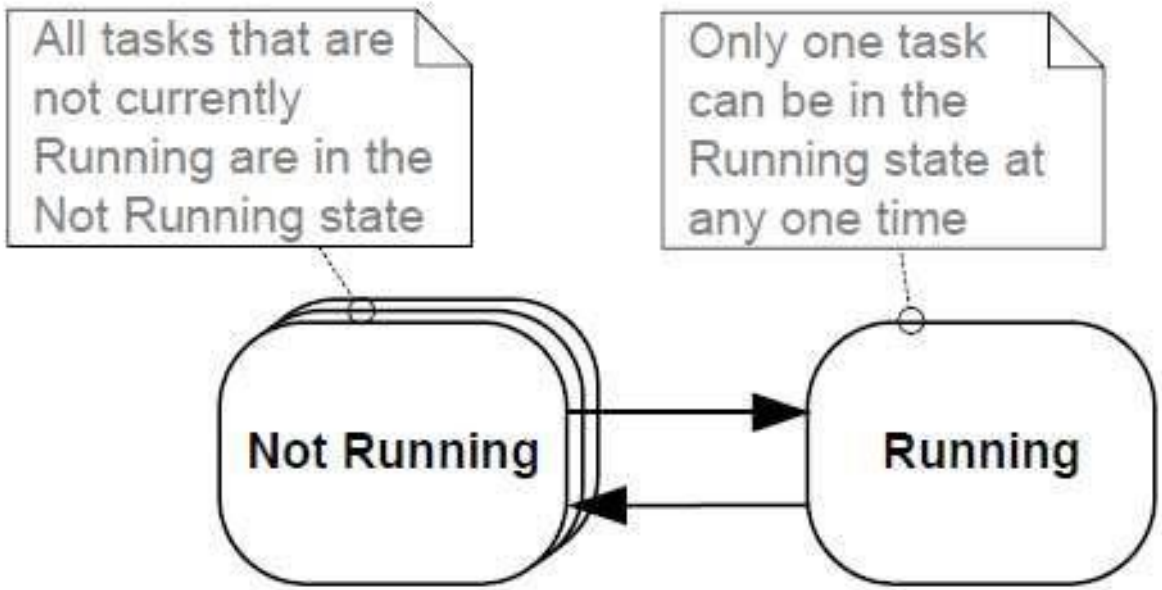
- If a task is no longer required, it should be deleted

Task Structure

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function. Each instance of a task created using this example function will have its own copy of the lVariableExample 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. (The prefixes added to variable names are described in section 1.5, Data Types and Coding Style Guide.) */
int32_t lVariableExample = 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 its implementing function. The NULL parameter passed to the vTaskDelete() API function indicates that the task to be deleted is the calling (this) task. The convention used to name API functions is described in section 0, Projects that use a FreeRTOS version older than V9.0.0 must build one of the heap_n.c files. From FreeRTOS V9.0.0 a heap_n.c file is only required if configSUPPORT_DYNAMIC_ALLOCATION is set to 1 in FreeRTOSConfig.h or if configSUPPORT_DYNAMIC_ALLOCATION is left undefined. Refer to Chapter 2, Heap Memory Management, for more information. Data Types and Coding Style Guide. */
vTaskDelete( NULL );
}
```

Top Level Task States

- An application can consist of many tasks
- If the processor running the real-time application contains a single core, then only one task can execute at any given time
- The FreeRTOS scheduler is the only entity that can switch a task in and out from states.



The xTaskCreate() API Function

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

Parameters passed to the xTaskCreate() function:

- pvTaskCode: a pointer to the function that implements the task
- pcName: a descriptive name for the task; used mainly by user as a debugging aid
- usStackDepth: defines the size in words of the tasks' stack that is allocated at creation of the task
- pvParameters: a parameter of type pointer to void (void*). The value assigned to pvParameters is the value passed into the task.
- uxPriority: defines the priority at which the task will execute; 0 is the lowest priority here
- pxCreatedTask: is used to pass a handle to the task being created. This handle can then be used to reference the task in API calls that; pxCreatedTask can be set to NULL when not needed.

Example 1: Create two simple tasks, then start the tasks executing

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. *
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - small microcontrollers will use much
                             less stack than this. */
                    NULL,    /* This example does not use the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* This example does not use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 2 provides more information on heap memory management. */
    for( ;; );
```

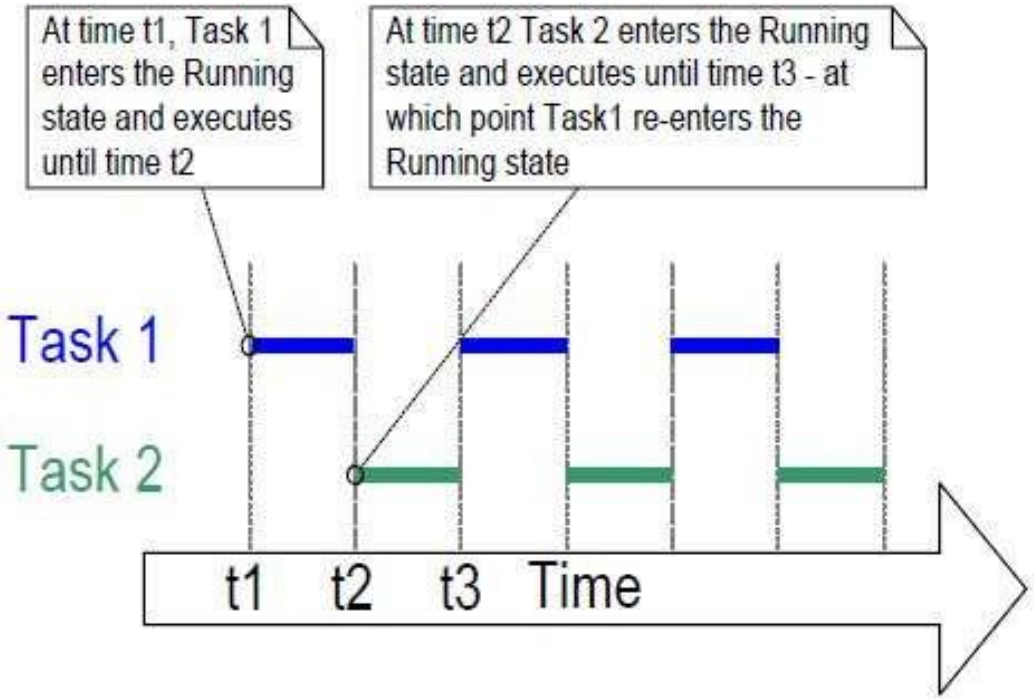
Example 1: Task 1

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* 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. */
        }
```

Example 1: Task 2

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* 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. */
        }
```

Example 1: Execution Results



11. The actual execution pattern of the two Example 1 tasks

Example 2: Using the Task Parameter

- The pvParameters parameter to the xTaskCreate() function is used to pass the text string into the task.

```
/* 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\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(    vTaskFunction,          /* Pointer to the function that
                                               implements the task. */
                    "Task 1",               /* Text name for the task.  This is to
                                               facilitate debugging only. */
                    1000,                   /* Stack depth - small microcontrollers
                                               will use much less stack than this. */
                    (void*)pcTextForTask1,  /* Pass the text to be printed into the
                                               task using the task parameter. */
                    1,                      /* This task will run at priority 1. */
                    NULL );                 /* The task handle is not used in this
                                               example. */

    /* Create the other task in exactly the same way.  Note this time that multiple
    tasks are being created from the SAME task implementation (vTaskFunction).  Only
    the value passed in the parameter is different.  Two instances of the same
    task are being created. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

    /* Start the scheduler so 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 2 provides more information on heap memory management. */
    for( ;; );
```

Example 2: Task

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

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

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

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

Task Priorities

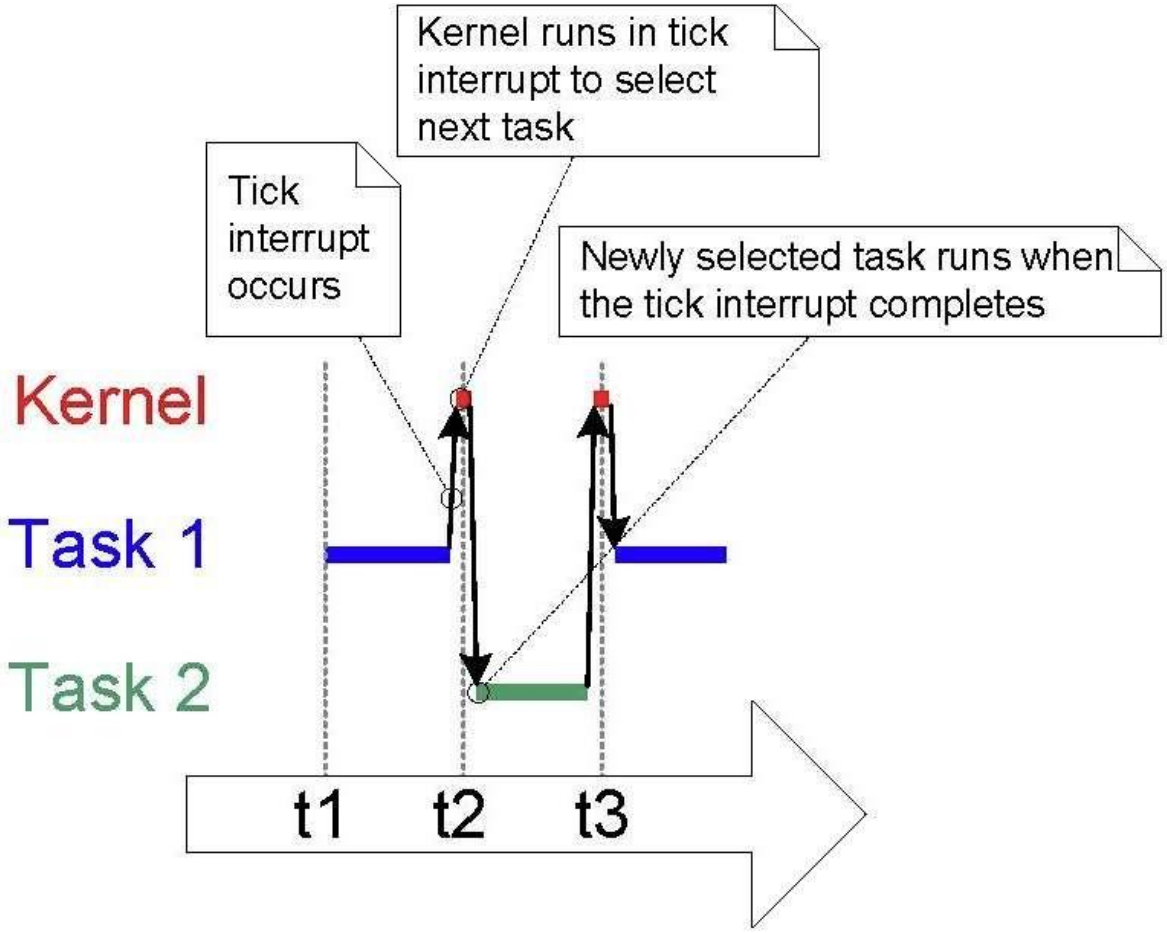
- uxPriority parameter of the xTaskCreate() API function assigns an initial priority to the task being created.
  - There is a maximum nr of priorities, configurable
  - The priority can be changed by using the vTaskPrioritySet()
- The FreeRTOS scheduler will ensure that the highest priority task that is available to run is running
- If tasks have the same priority a round robin scheme is used
- The FreeRTOS scheduler can use one of two methods (configurable) to decide which task will be in the Running state
  - Generic method: implemented in C and can be used with all the FreeRTOS architecture ports.
  - Architecture optimized method: uses a small amount of assembler code, and is faster than the generic method.

Time Measurement and the Tick Interrupt

- A periodic interrupt called the “tick interrupt” is used by an RTOS to measure time delays and time slices
  - An interrupt is issued at the end of a tick interrupt and the scheduler will be invoked
- The length of the time slice is effectively set by the tick interrupt frequency, which is configured by the application-defined configTICK_RATE_HZ compile time configuration constant within FreeRTOSConfig.h.
  - For example, if configTICK_RATE_HZ is set to 100 (Hz), then the time slice will be 10 milliseconds.
  - The time between two tick interrupts is called the ‘tick period’.
  - One time slice (used by Round-Robin scheduling technique) equals one tick period

Tick Interrupt Executing

- FreeRTOS API calls always specify time in multiples of tick periods -> ‘ticks’.
- The pdMS_TO_TICKS() macro converts a time specified in milliseconds into a time specified in ticks.
- The resolution available depends on the defined tick frequency



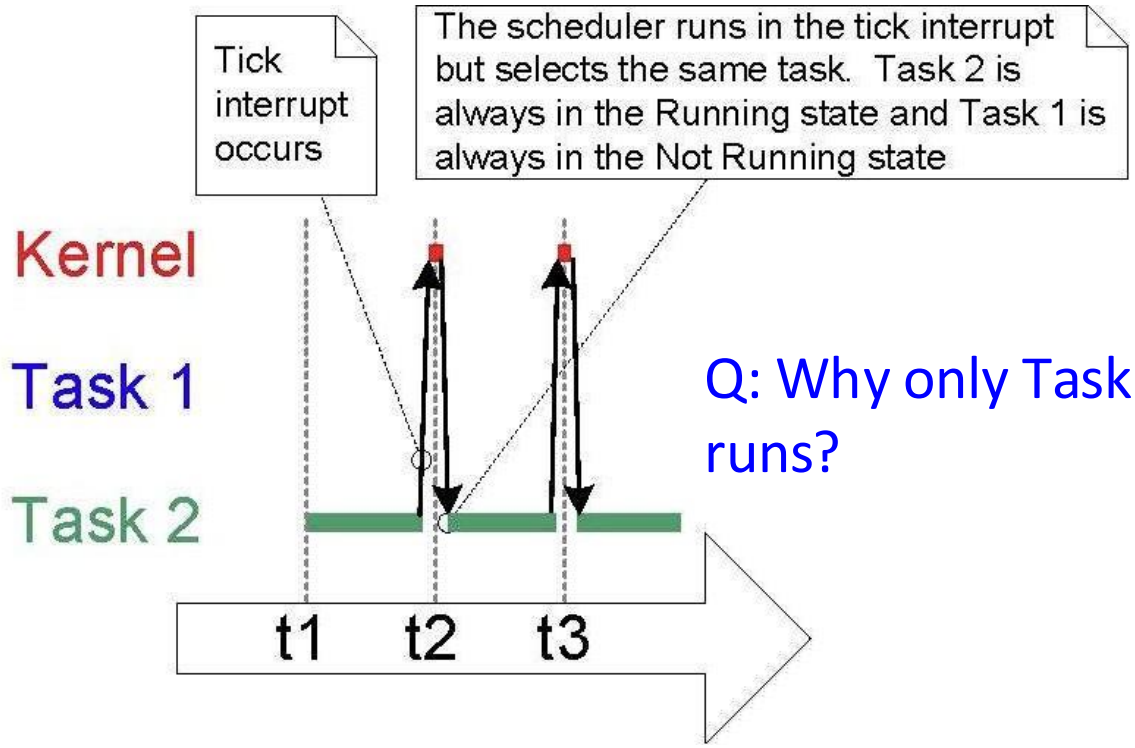
Example 3: Experimenting with Priorities

- The scheduler will always ensure that the highest priority task that is able to run is the scheduled task
- What happens with Example 2 when we have tasks with different priorities?

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

int main( void )
{
    /* Create the first task at priority 1.  The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2, which is higher than a priority of 1.
    The priority is the second to last parameter. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

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

    /* Will not reach here. */
    return 0;
}
```

Example 3: Execution Pattern

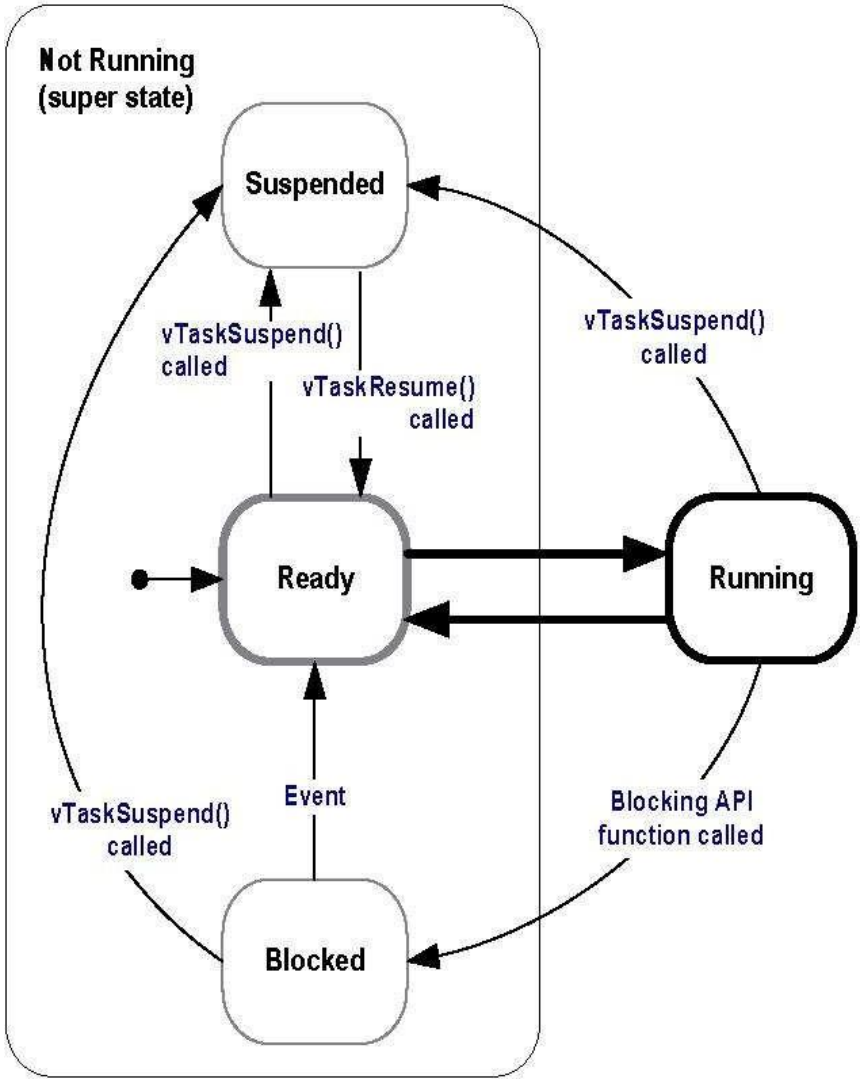
```
C:\Temp>rtosdemo
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
```



Q: Why only Task 2 runs?

Full Task State Machine

- To make tasks useful and run a different priorities they must be re-written to be event-driven
  - An event-driven task has work (processing) to perform only after the occurrence of an event that triggers it … it can enter running state …
- States:
  - Blocked …
  - Suspended …
  - Ready …
  - Running …



The Blocked State

A task that is waiting for an event is said to be in the 'Blocked' state, which is a sub-state of the Not Running state.

Tasks can enter the Blocked state to wait for two different types of event:

1. Temporal (time-related) events—the event being either a delay period expiring, or an absolute time being reached. For example, a task may enter the Blocked state to wait for 10 milliseconds to pass.

2. Synchronization events—where the events originate from another task or interrupt. For example, a task may enter the Blocked state to wait for data to arrive on a queue. Synchronization events cover a broad range of event types.

FreeRTOS queues, binary semaphores, counting semaphores, mutexes, recursive mutexes, event groups and direct to task notifications can all be used to create synchronization events.

Example 4. Blocked state used to create a delay

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );
```

Replace the delay loop from previous examples with a delay event

```
/* 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
    parameter takes a time specified in 'ticks', and the pdMS_TO_TICKS() macro
    is used (where the xDelay250ms constant is declared) to convert 250
    milliseconds into an equivalent time in ticks. */
    vTaskDelay( xDelay250ms );
}
```

The execution sequence

** both tasks run now … why? …

** The execution of the scheduler itself is omitted for simplicity
**The idle task is created automatically when the scheduler is started

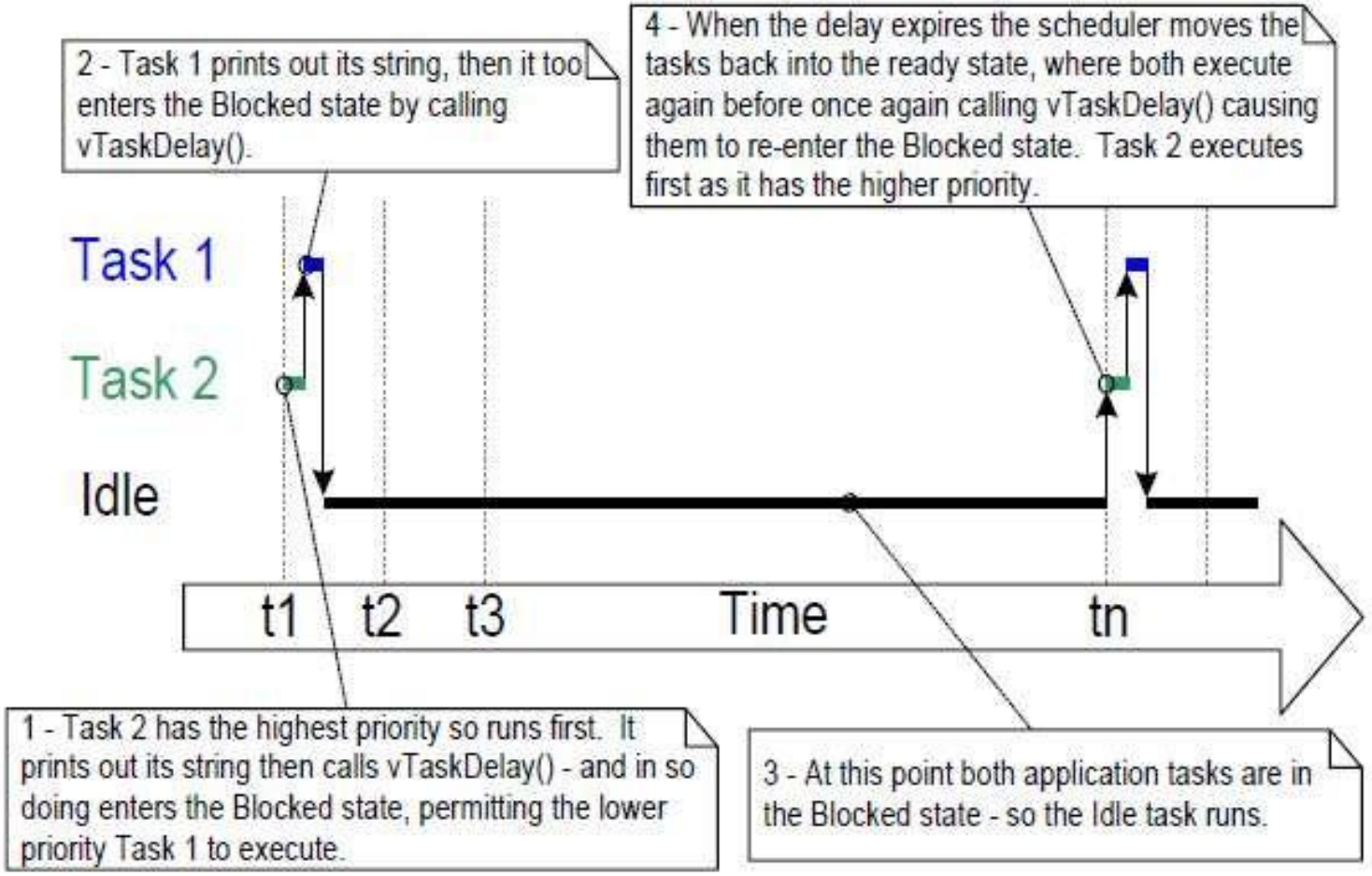


Figure 17. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

Example 5. Converting the example tasks to use vTaskDelayUntil()

- Using an RTOS can significantly increase the spare processing capacity simply by allowing an application to be completely event driven.
- The two tasks created in Example 4 are periodic tasks but using vTaskDelay() does not guarantee 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().
- Converting the tasks to use vTaskDelayUntil() instead of vTaskDelay() solves this potential problem.

Q: How this event delay vTaskDelayUntil (…) is different than the previous event delay vTaskDelay(…)

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

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

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

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

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

Example 6: Combining Blocking and Non-Blocking States

- Two tasks are created at priority 1 and are of continuous processing type tasks; these task are always in Ready or Running state.

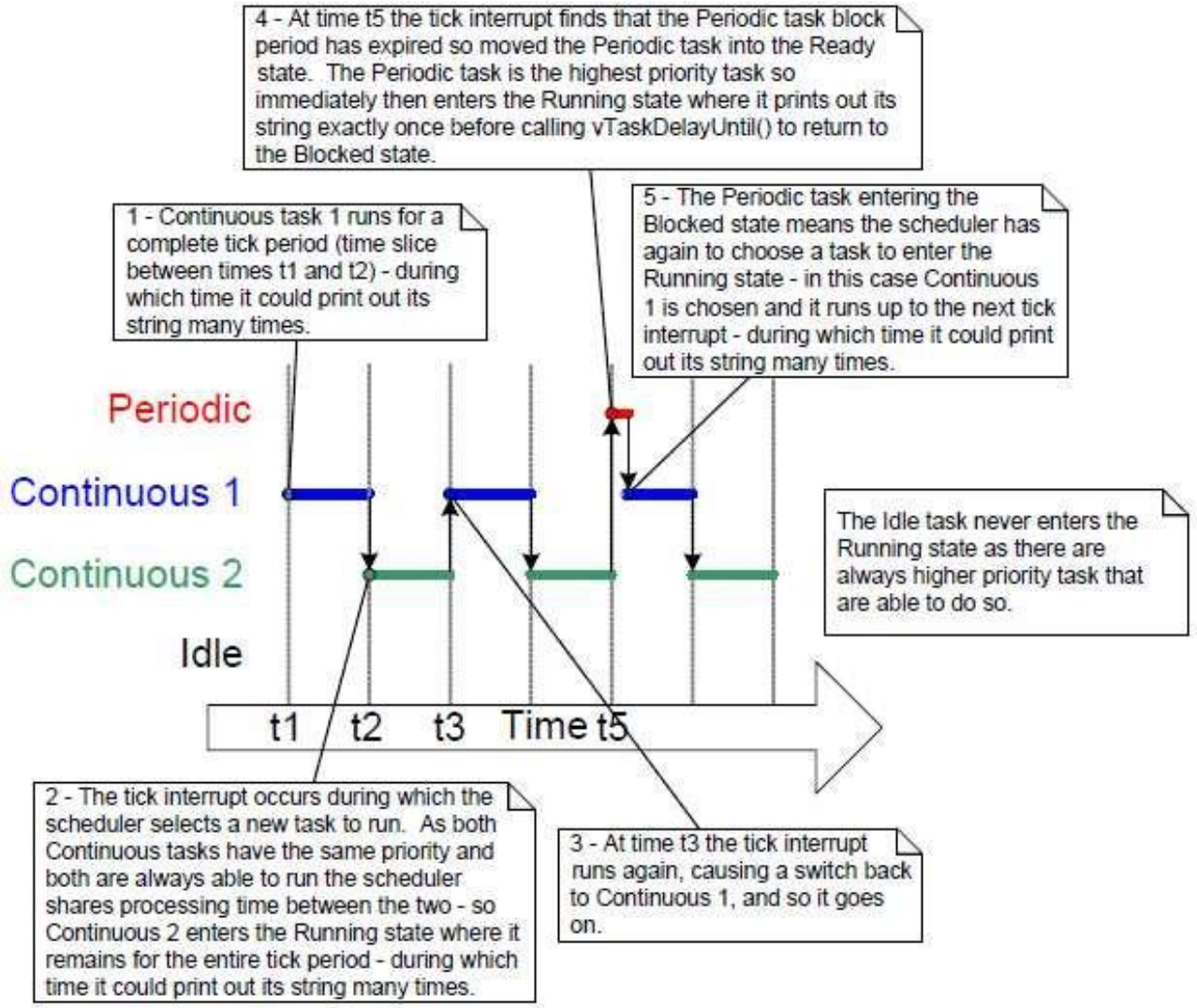
```
void vContinuousProcessingTask( void *pvParameters )
{
char *pcTaskName;

    /* 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 );
    }
```

Example 6: Periodic Task

- A third task is created at priority 2 and is periodic type; so, it uses vTaskDelayUntil() API function to place itself into the Blocked state between each print iteration.

```
void vPeriodicTask( void *pvParameters )
{
TickType_t xLastWakeTime;
const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );

    /* 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\r\n" );

        /* The task should execute every 3 milliseconds exactly - see the
        declaration of xDelay3ms in this function. */
        vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
```

Example 6: Execution pattern



The Idle Task and the Idle Task Hook

- The idle task has the lowest priority (priority zero)
- It is possible to add application specific functionality directly into the idle task through the use of an idle hook (or idle callback) function.
- Common uses for the idle task hook:
  - Executing low priority, background, or continuous processing functionality
  - Measuring the amount of spare processing capacity
  - Placing the processor into a low power mode

Example 7: Use of an Idle Hook Function in Example 4

Example 4 created a lot of idle time—time when the Idle task is executing because both application tasks are in the Blocked state … how can we make use of this time?

```
/* Declare a variable that will be incremented by the hook function. */
volatile uint32_t 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 29. A very simple Idle hook function

configUSE_IDLE_HOOK must be set to 1 in FreeRTOSConfig.h for the idle hook function to get called.

The function that implements the created tasks is modified slightly to print out the ulIdleCycleCount value, as shown in Listing 30.

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* 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( xDelay250ms );
    }
}
```

Scheduling Algorithms, Task States

- The task that is actually running (using processing time) is in the Running state.
- Tasks that are not running and are not in either the Blocked state or the Suspended state, are in the Ready state.
- Tasks in the Ready state are available to be selected by the scheduler to enter the Running state.
- The scheduler will always choose the highest priority Ready state task to enter the Running state.

=> The scheduler needs scheduling algorithms!!!

Scheduling Algorithms

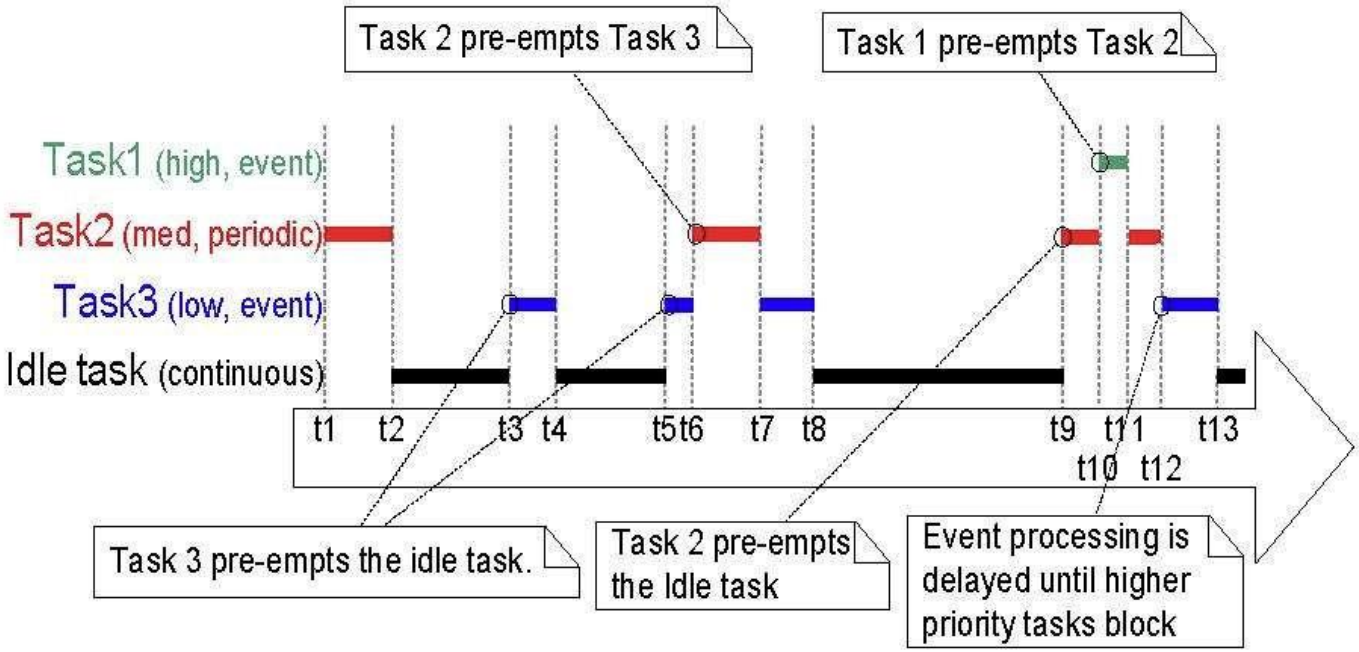
- The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state.
- FreeRTOS provides three configuration constants within FreeRTOSConfig.h that can affect the scheduling algorithm:
  - ConfigUSE_PREEMPTION
  - ConfigUSE_SLICING
  - configUSE_TICKLESS_IDLE
- Scheduler can be set to be preemptive and use time slicing for (Round Robin Scheduling) for tasks at same priority.

Scheduling Algorithm Settings

1. Prioritized Pre-emptive Scheduling with Time Slicing: when configUSE_PREEMPTION set to 1 and configUSE_TIME_SLICING set to 1
2. Prioritized Pre-emptive Scheduling without Time Slicing: when configUSE_PREEMPTION set to 1 and configUSE_TIME_SLICING set to 0
3. Co-operative Scheduling: when configUSE_PREEMPTION set to 0 and configUSE_TIME_SLICING set to any value

1. Fixed Priority Pre-emptive Scheduling with Time Slicing Algorithm Example – Unique Priority Tasks Scenario

All FreeRTOS inter-task communication mechanisms (task notifications, queues, semaphores, event groups, etc.) can be used to signal events and unblock tasks in this way.



26. Execution pattern highlighting task prioritization and pre-emption in a hypothetical application in which each task has been assigned a unique priority

1. Fixed Priority Pre-emptive Scheduling with Time Slicing Algorithm Example – Time Slicing Tasks Scenario

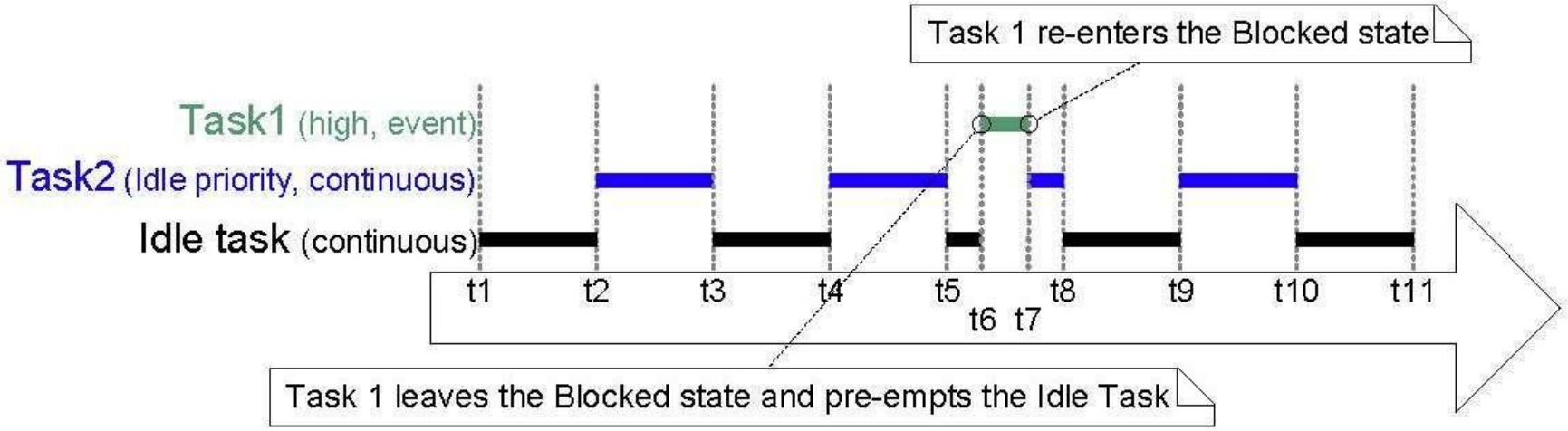
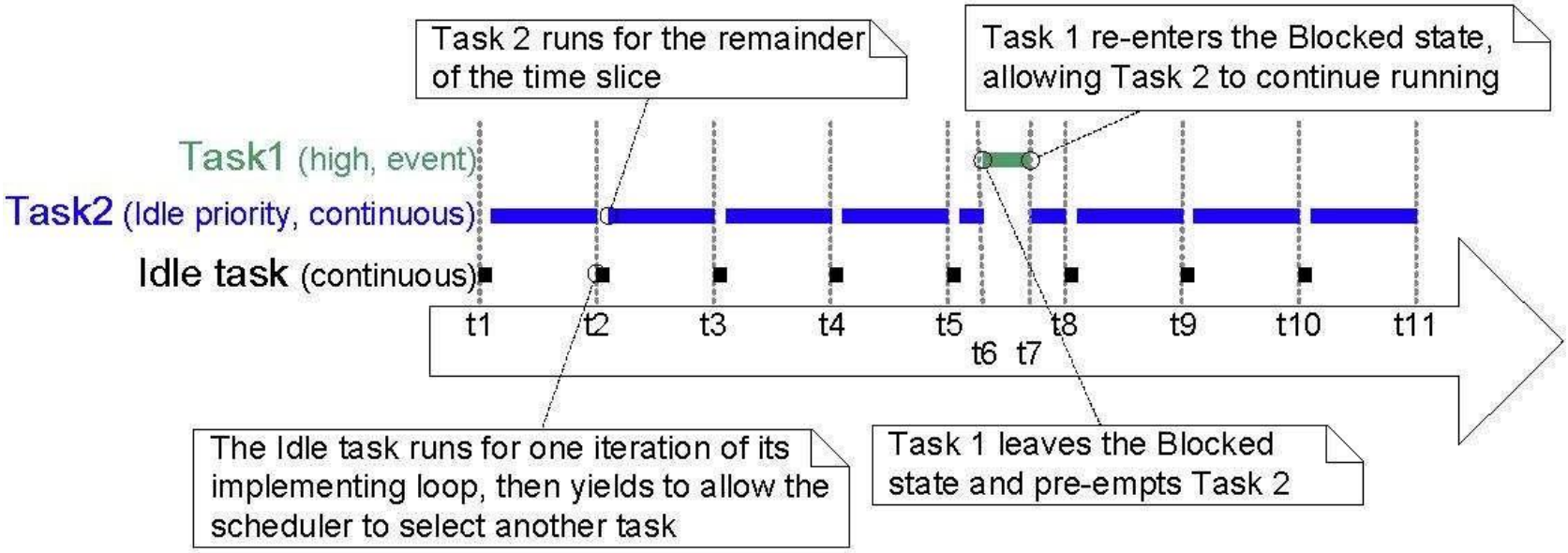


Figure 27 Execution pattern highlighting task prioritization and time slicing in a hypothetical application in which two tasks run at the same priority

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

1. The Effect of configIDLE_SHOULD_YIELD



If configIDLE_SHOULD_YIELD is set to 1 then the Idle task will yield (voluntarily give 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.

2. Prioritized Pre-emptive Scheduling without Time Slicing

With no time slicing scheduling, the scheduler will only select a new task to enter the Running state when either:

1. A higher priority task enters the Ready state.
2. The task in the Running state enters the Blocked or Suspended state

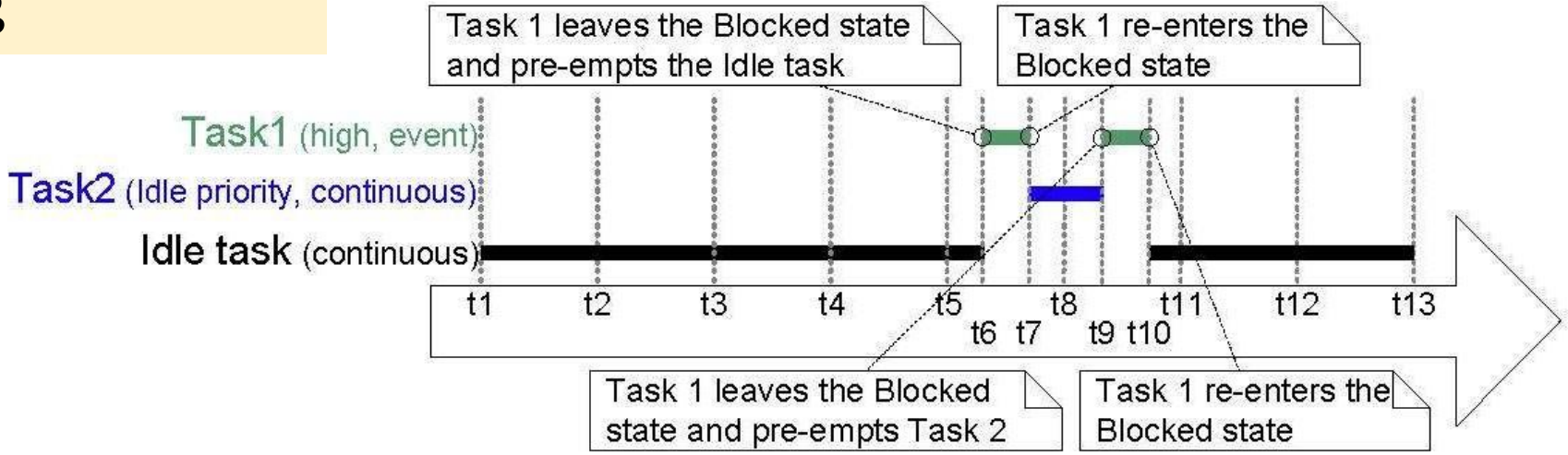


Figure 29 Execution pattern that demonstrates how tasks of equal priority can receive hugely different amounts of processing time when time slicing is not used

3. Co-Operating Scheduling

Co-operative scheduler: a context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a re-schedule) by calling taskYIELD(). Tasks are never pre-empted, so time slicing cannot be used.

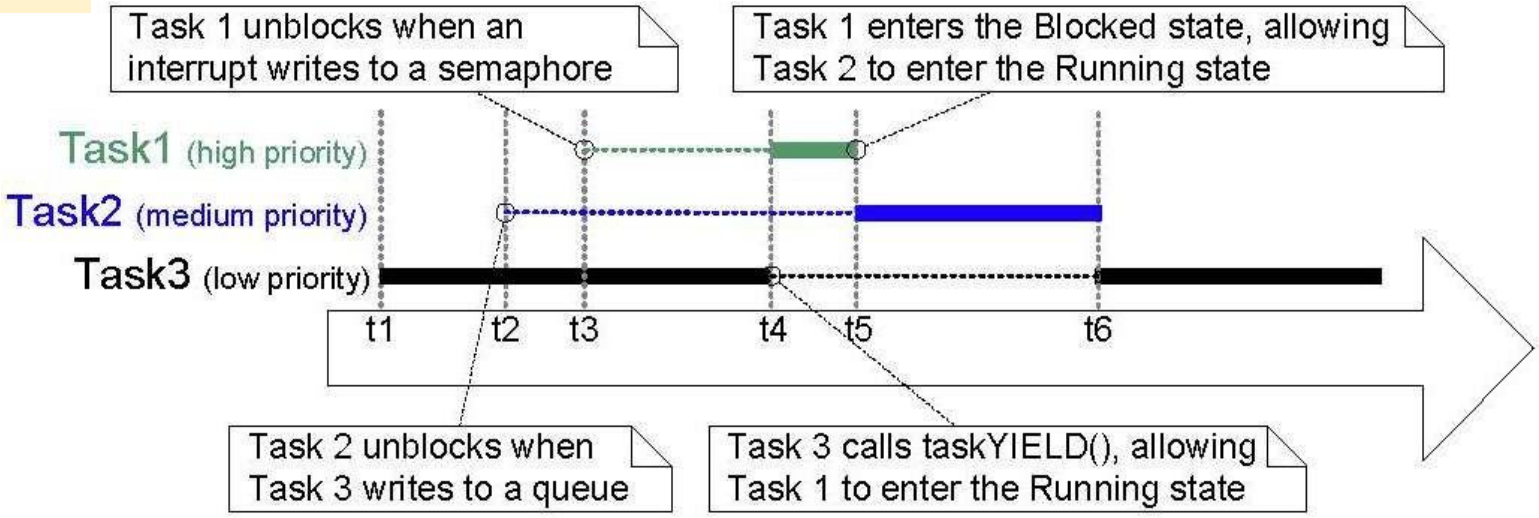


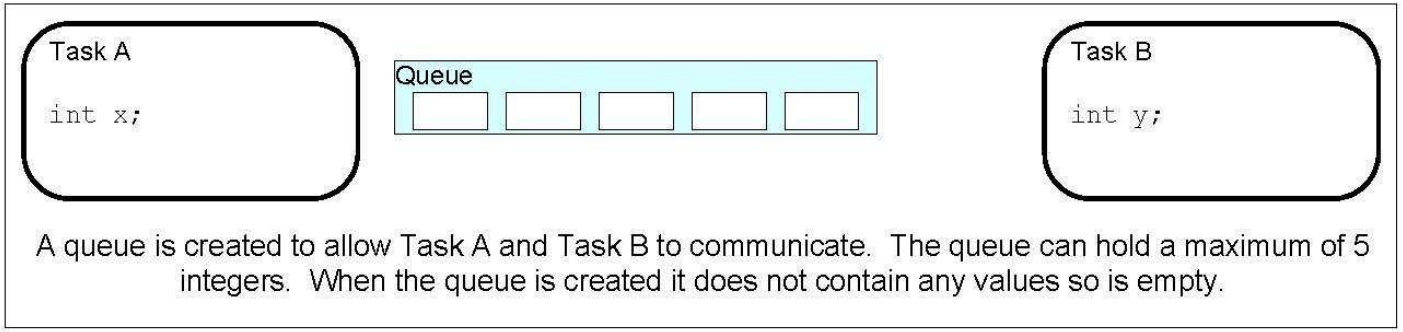
Figure 30 Execution pattern demonstrating the behavior of the co-operative scheduler

Task Communication Support ([1] Chapter 4)

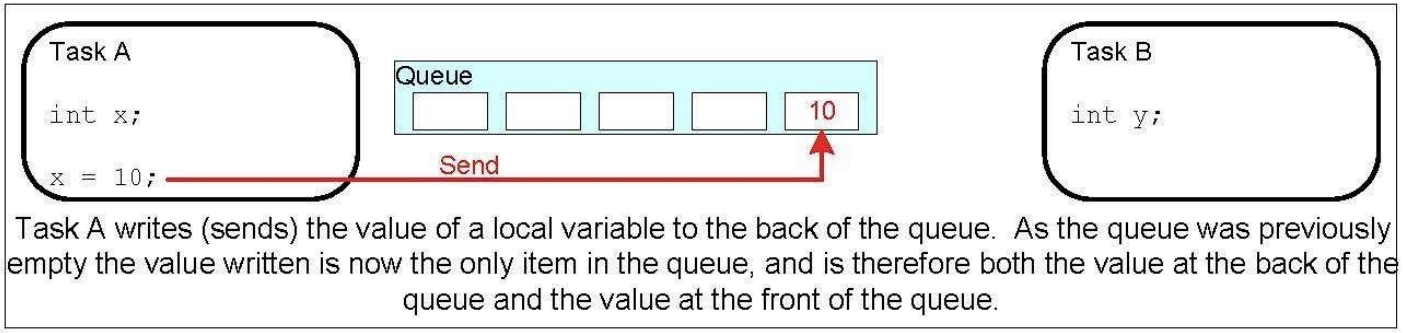
- Queues provide task-to-task, task-to-interrupt and interrupt-to-task communication
- A queue can hold a finite number of fixed size data items
- Both the length and the size of each data item are set when the queue is created
- Queues are normally used as FIFO buffers where data is written to the end (tail) of the queue and removed from the front (head) of the queue.

Task-to-Task Communication

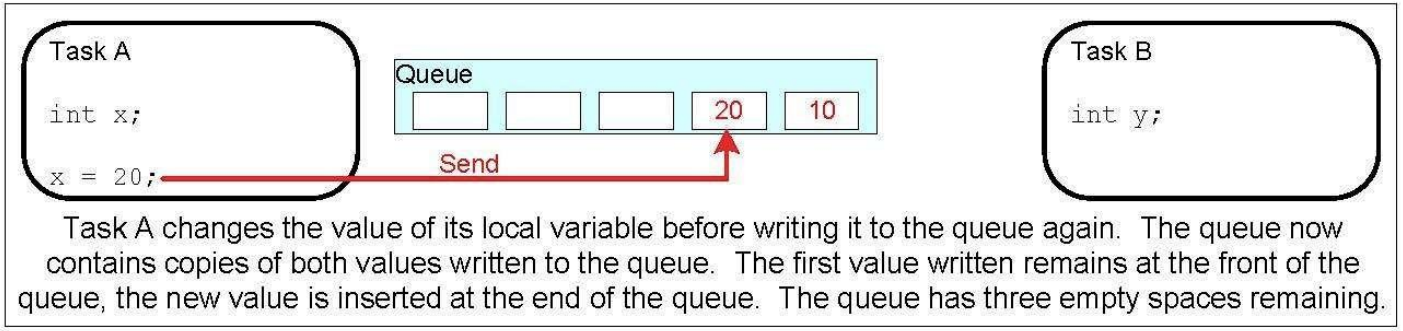
An example sequence of writes to, and reads from a queue, in FIFO mode.



A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

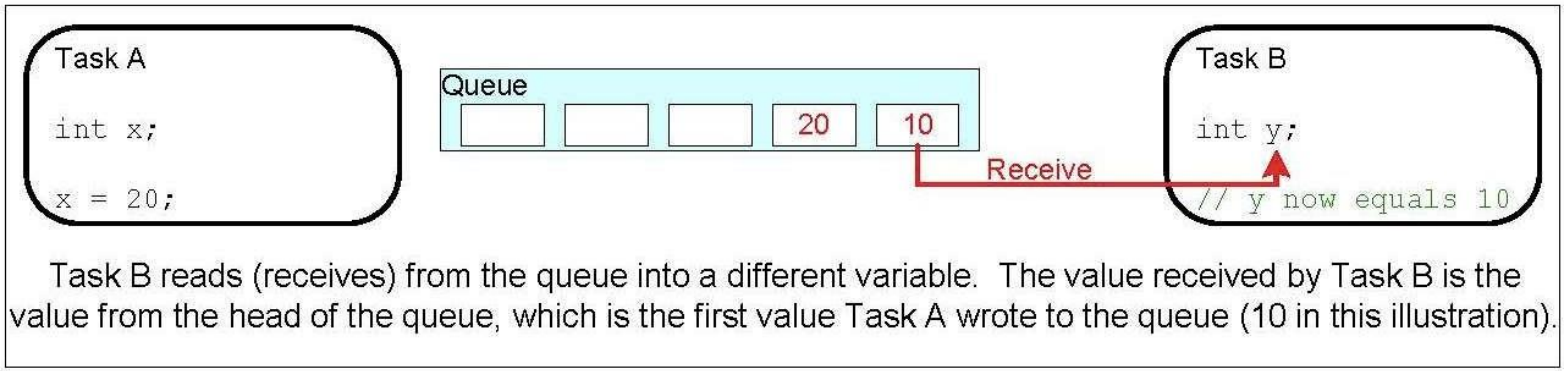


Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

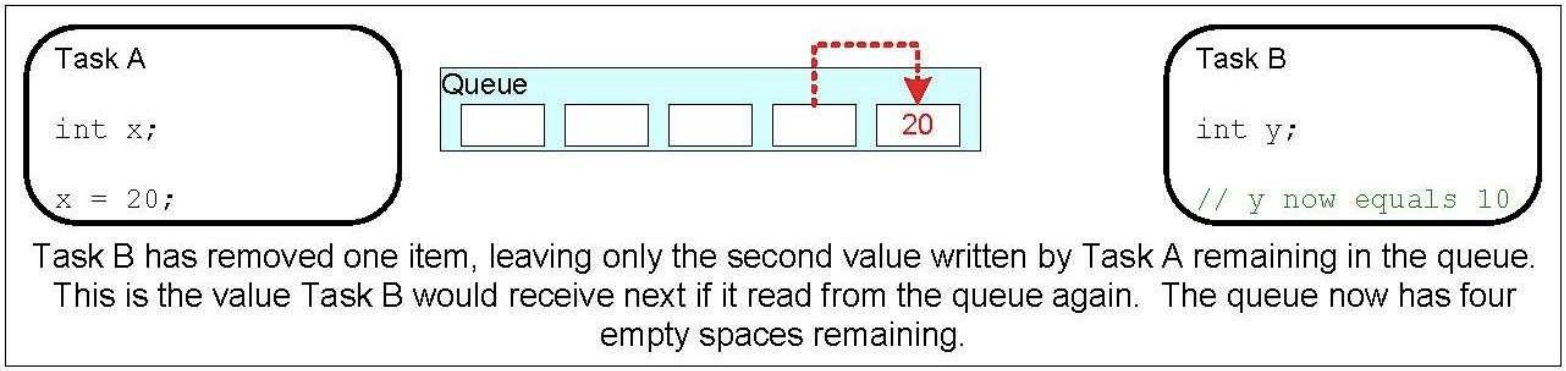


Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.

Cont …



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Queue Behavior Implementation

- There are two implementation techniques:
  - Queue by copy
    - the data sent to the queue is copied byte for byte into the queue.
  - Queue by reference
    - the queue only holds pointers to the data sent to the queue, not the data itself
- FreeRTOS uses the queue by copy method
  - Queuing by copy is considered to be simultaneously more powerful and simpler to use than queueing by reference because:
    - …

Using Queues

- Queues are OS objects that can be accessed by any task or ISR that knows of their existence. Any number of tasks can write to the same queue, and any number of tasks can read from the same queue.
- In practice it is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers.

Blocking on Queues

- Blocking on Queue Reads
  - A task can optionally specify a ‘block’ time on queue reads so the task will be in the Blocked state while waiting for data to be available from the queue, should the queue already be empty
- Blocking on Queue Writes
  - A block time on queue writes is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full.
- Blocking on Multiple Queues
  - Queues can be grouped into sets, allowing a task to enter the Blocked state to wait for data to become available on any of the queues in set

Using a Queue; Common API Functions

- The xQueueCreate() – must create a queue before use
  - Queues are referenced by handles, which are variables of type QueueHandle_t.
- xQueueSendToBack() and xQueueSendToFront() – sends an item to a queue
  - FreeRTOS provides interrupt-safe versions for these functions xQueueSendToFrontFromISR() and xQueueSendToBackFromISR()
- The xQueueReceive() – receives an item from a queue
  - The interrupt-safe version is xQueueReceiveFromISR()
- uxQueueMessagesWaiting() -- query the number of items that are currently in a queue.

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

Listing 40. The xQueueCreate() API function prototype

Table 18. xQueueCreate() parameters and return value

Parameter Name	Description
uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size in bytes of each data item that can be stored in the queue.
Return Value	If NULL is returned, then the queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area. A non-NULL value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.

Queue Usage Example; Blocking when receiving from a queue

- Here we create a queue and send data to the queue from multiple tasks, and data being received from the queue by a task.
  - The queue can hold data items of type int32_t.
  - The tasks that send to the queue do not specify a block time, whereas the task that receives from the queue does.
- The priority of the tasks that send to the queue are lower than the priority of the task that receives from the queue.
  - As a result, the queue should never contain more than one item because, as soon as data is sent to the queue the receiving task will unblock, pre-empt the sending task, and remove the data—leaving the queue empty once again.

Example 10: main()

```
/* Declare a variable of type QueueHandle_t.  This is used to store the handle
to the queue that is accessed by all three tasks. */
QueueHandle_t xQueue;

int main( void )
{
    /* The queue is created to hold a maximum of 5 values, each of which is
    large enough to hold a variable of type int32_t. */
    xQueue = xQueueCreate( 5, sizeof( int32_t ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will send to the queue.  The task
        parameter is used to pass the value that the task will write to the queue,
        so one task will continuously write 100 to the queue while the other task
        will continuously write 200 to the queue.  Both tasks are created at
        priority 1. */
        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

        /* Create the task that will read from the queue.  The task is created with
        priority 2, so above the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
```

Example 10: vSenderTask()

```
static void vSenderTask( void *pvParameters )
{
int32_t lValueToSend;
BaseType_t xStatus;

    /* Two instances of this task are created so the value that is sent to the
    queue is passed in via the task parameter - this way each instance can use
    a different value.  The queue was created to hold values of type int32_t,
    so cast the parameter to the required type. */
    lValueToSend = ( int32_t ) pvParameters;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send the value to the queue.

        The first parameter is the queue to which data is being sent.  The
        queue was created before the scheduler was started, so before this task
        started to execute.

        The second parameter is the address of the data to be sent, in this case
        the address of lValueToSend.

        The third parameter is the Block time - the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        should the queue already be full.  In this case a block time is not
        specified because the queue should never contain more than one item, and
        therefore never be full. */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete because the queue was full -
            this must be an error as the queue should never contain more than
            one item! */
            vPrintString( "Could not send to the queue.\r\n" );
```

Example 10: vReceiverTask()

```
static void vReceiverTask( void *pvParameters )
{
/* Declare the variable that will hold the values received from the queue. */
int32_t lReceivedValue;
BaseType_t xStatus;
const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* This call should always find the queue empty because this task will
        immediately remove any data that is written to the queue. */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }

        /* Receive data from the queue.

        The first parameter is the queue from which data is to be received.  The
        queue is created before the scheduler is started, and therefore before this
        task runs for the first time.

        The second parameter is the buffer into which the received data will be
        placed.  In this case the buffer is simply the address of a variable that
        has the required size to hold the received data.

        The last parameter is the block time - the maximum amount of time that the
        task will remain in the Blocked state to wait for data to be available
        should the queue already be empty. */
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value. */
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
```

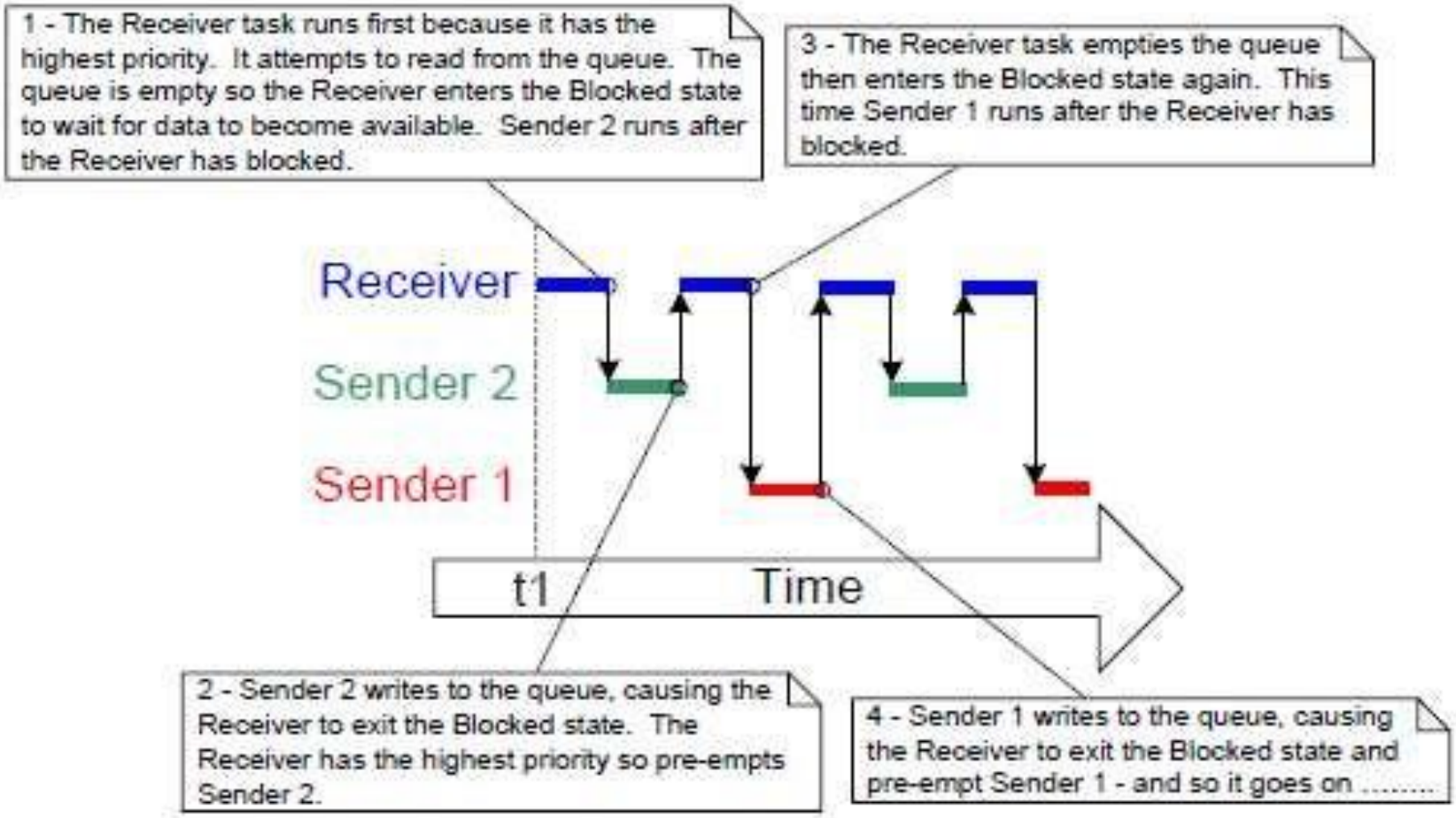


Figure 33. The sequence of execution produced by Example 10

Receiving Data from Multiple Sources: Example 11

Problem: receiving task needs to know where the data comes from to determine how data should be processed.

Solution: use a single queue to transfer structures with both:

- the value of the data
- the source of the data

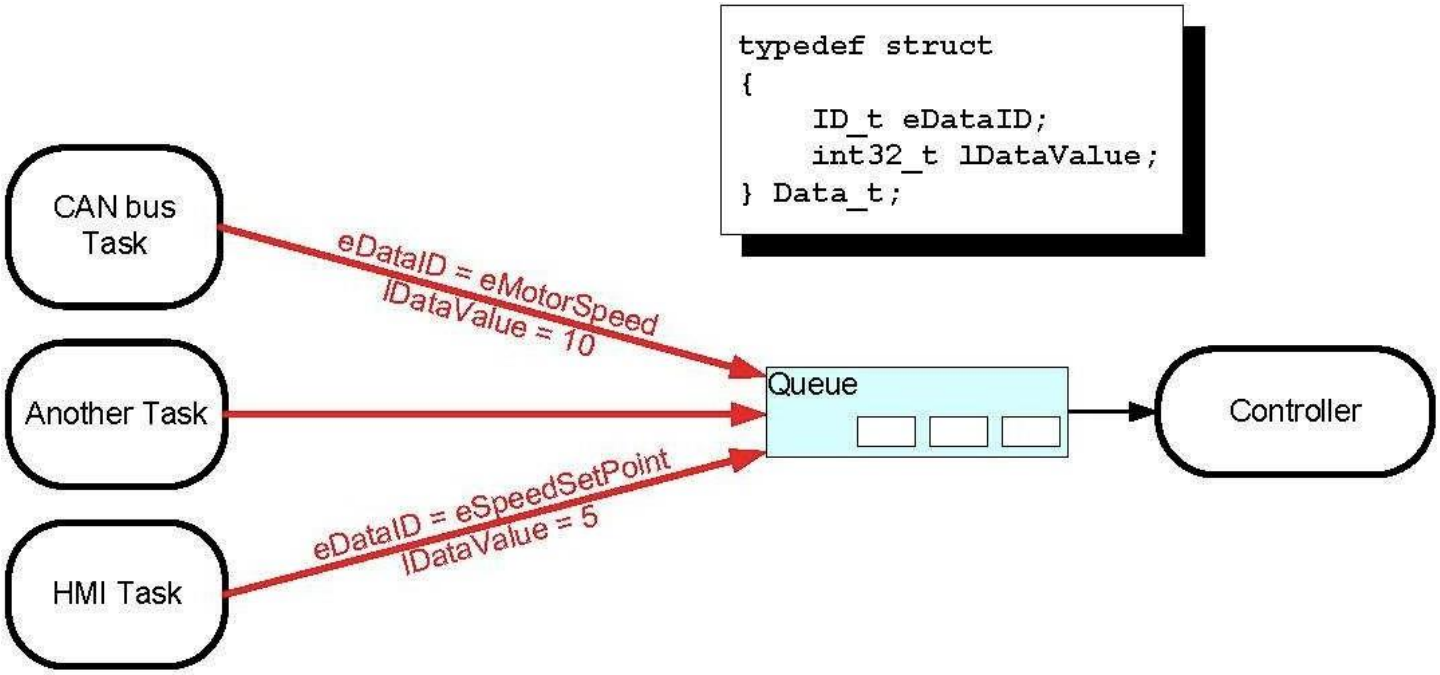


Figure 34. An example scenario where structures are sent on a queue

Referring to Figure 34:

See Example 11; FreeRTOS Book!

- A queue is created that holds structures of type Data_t. The structure members allow both a data value and an enumerated type indicating what the data means to be sent to the queue in one message.

- A central Controller task is used to perform the primary system function. This has to react to inputs and changes to the system state communicated to it on the queue.

- A CAN bus task is used to encapsulate the CAN bus interfacing functionality. When the CAN bus task has received and decoded a message, it sends the already decoded message to the Controller task in a Data_t structure. The eDataID member of the transferred structure is used to let the Controller task know what the data is—in the depicted case it is a motor speed value. The lDataValue member of the transferred structure is used to let the Controller task know the actual motor speed value.

- A Human Machine Interface (HMI) task is used to encapsulate all the HMI functionality.

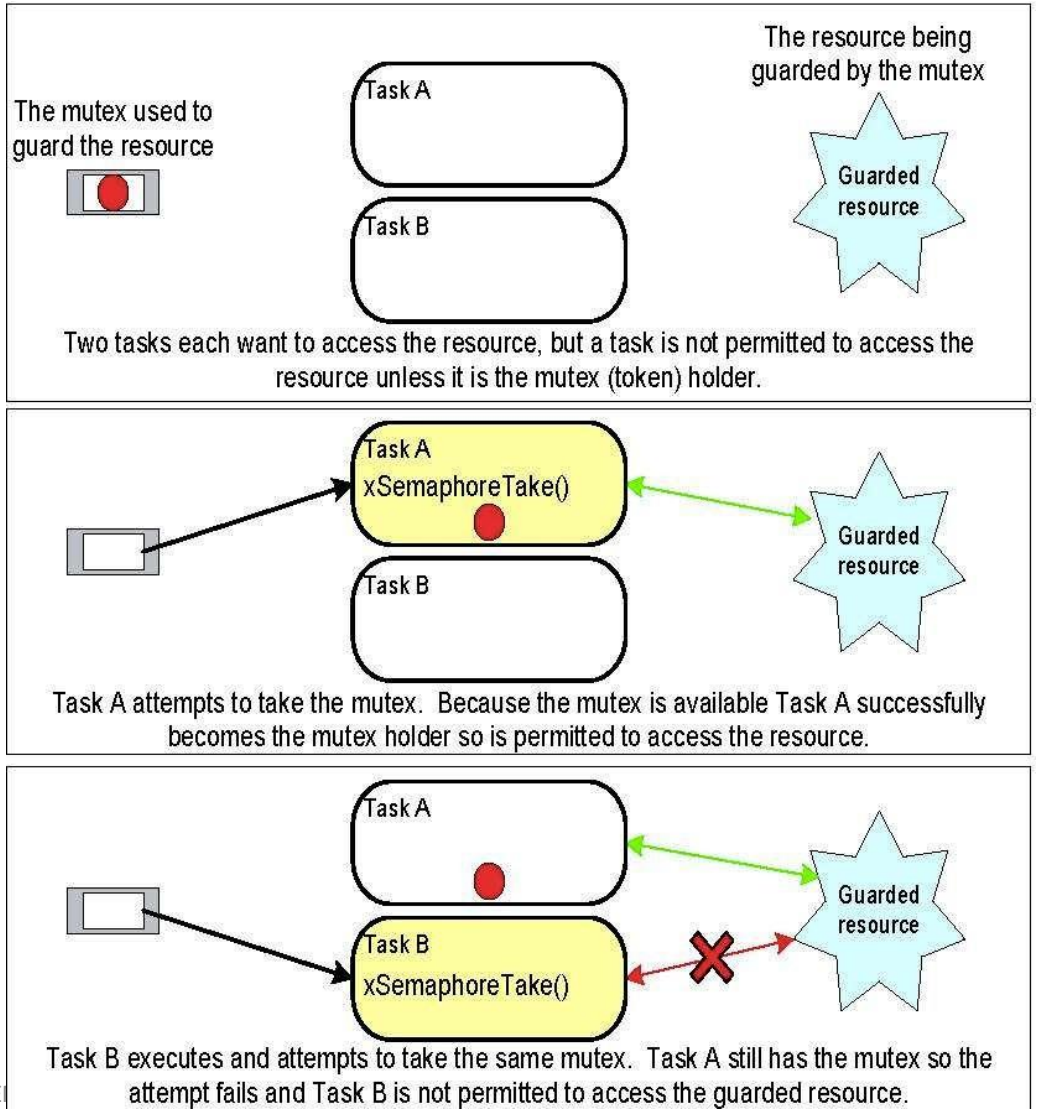
Resource Management (Chapter 7, Section 7.3)

- A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks
  - The mutex in a mutual exclusion scenario can be thought as a token that is associated with the resource being shared
- A mutex can be used for mutual exclusion access and a semaphore can be used for synchronization. What is the difference in this scenarios?
  - A semaphore used for mutual exclusion must always be returned
  - A semaphore used for synchronization is normally discarded and not returned
- See Section 6.4 (Binary Semaphores used for Synchronization)

Mutual Exclusion Implemented using a Mutex

Scenario:

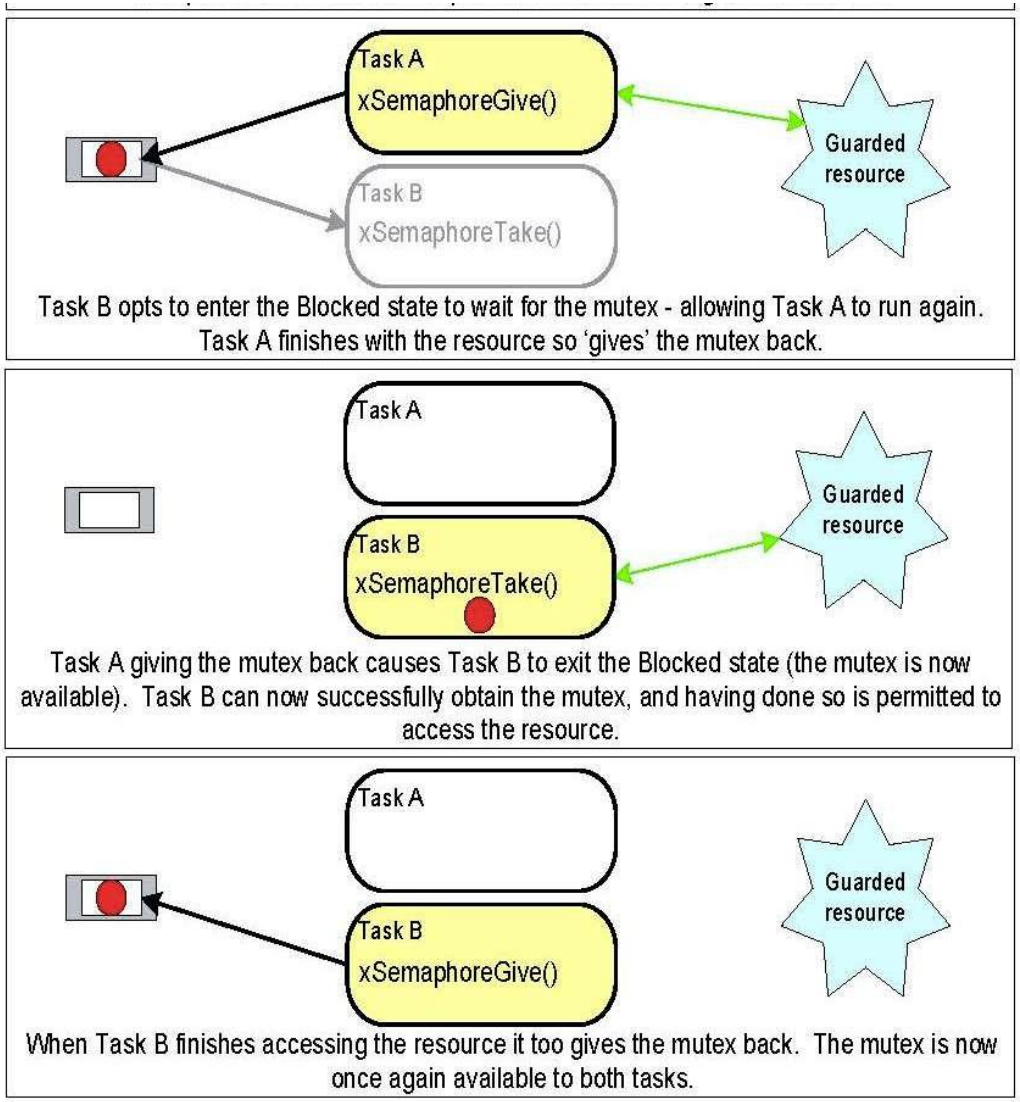
- Task A accesses the resource by becoming first the mutex (token) holder
- Task B is not allowed to access the resource



Mutual Exclusion Implemented using a Mutex

Scenario:

- Task B, opts to block and wait for the mutex
- Task A returns the mutex
- Task B is now allowed to access the resource



ENGG4420: Developed by Radu Muresan, F22

Utilizing a Mutex

Before a mutex can be used, it must be created. To create a mutex type semaphore, use the xSemaphoreCreateMutex() API function.

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Listing 120. The xSemaphoreCreateMutex() API function prototype

Table 41. xSemaphoreCreateMutex() return value

Parameter Name/ Returned Value	Description
Returned value	If NULL is returned then the mutex could not be created because there is insufficient heap memory available for FreeRTOS to allocate the mutex data structures. Chapter 2 provides more information on heap memory management. A non-NULL return value indicates that the mutex has been created successfully. The returned value should be stored as the handle to the created mutex.

Example 20. prvNewPrintString() to use a Mutex

vPrintString() function can be treated as a resource:

- prvNewPrintString(), controls standard out using a mutex rather than locking the scheduler

```
static void prvNewPrintString( const char *pcString )
{
    /* The mutex is created before the scheduler is started, so already exists by the
    time this task executes.

    Attempt to take the mutex, blocking indefinitely to wait for the mutex if it is
    not available straight away.  The call to xSemaphoreTake() will only return when
    the mutex has been successfully obtained, so there is no need to check the
    function return value.  If any other delay period was used then the code must
    check that xSemaphoreTake() returns pdTRUE before accessing the shared resource
    (which in this case is standard out).  As noted earlier in this book, indefinite
    time outs are not recommended for production code. */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute once the mutex has been successfully
        obtained.  Standard out can be accessed freely now as only one task can have
        the mutex at any one time. */
        printf( "%s", pcString );
        fflush( stdout );

        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );
}
```

```
static void prvPrintTask( void *pvParameters )
{
char *pcStringToPrint;
const TickType_t xMaxBlockTimeTicks = 0x20;

    /* Two instances of this task are created.  The string printed by the task is
    passed into the task using the task's parameter.  The parameter is cast to the
    required type. */
    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        /* Print out the string using the newly defined function. */
        prvNewPrintString( pcStringToPrint );

        /* Wait a pseudo random time.  Note that rand() is not necessarily reentrant,
        but in this case it does not really matter as the code does not care what
        value is returned.  In a more secure application a version of rand() that is
        known to be reentrant should be used - or calls to rand() should be protected
        using a critical section. */
        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );
    }
}
```

```
int main( void )
{
    /* Before a semaphore is used it must be explicitly created.  In this example a
    mutex type semaphore is created. */
    xMutex = xSemaphoreCreateMutex();

    /* Check the semaphore was created successfully before creating the tasks. */
    if( xMutex != NULL )
    {
        /* Create two instances of the tasks that write to stdout.  The string they
        write is passed in to the task as the task's parameter.  The tasks are
        created at different priorities so some pre-emption will occur. */
        xTaskCreate( prvPrintTask, "Print1", 1000,
                     "Task 1 ****************************************\r\n", 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 1000,
                     "Task 2 ----------------------------------------\r\n", 2, NULL );

        /* Start the scheduler so the created 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 2
    provides more information on heap memory management. */
    for( ;; );
}
```

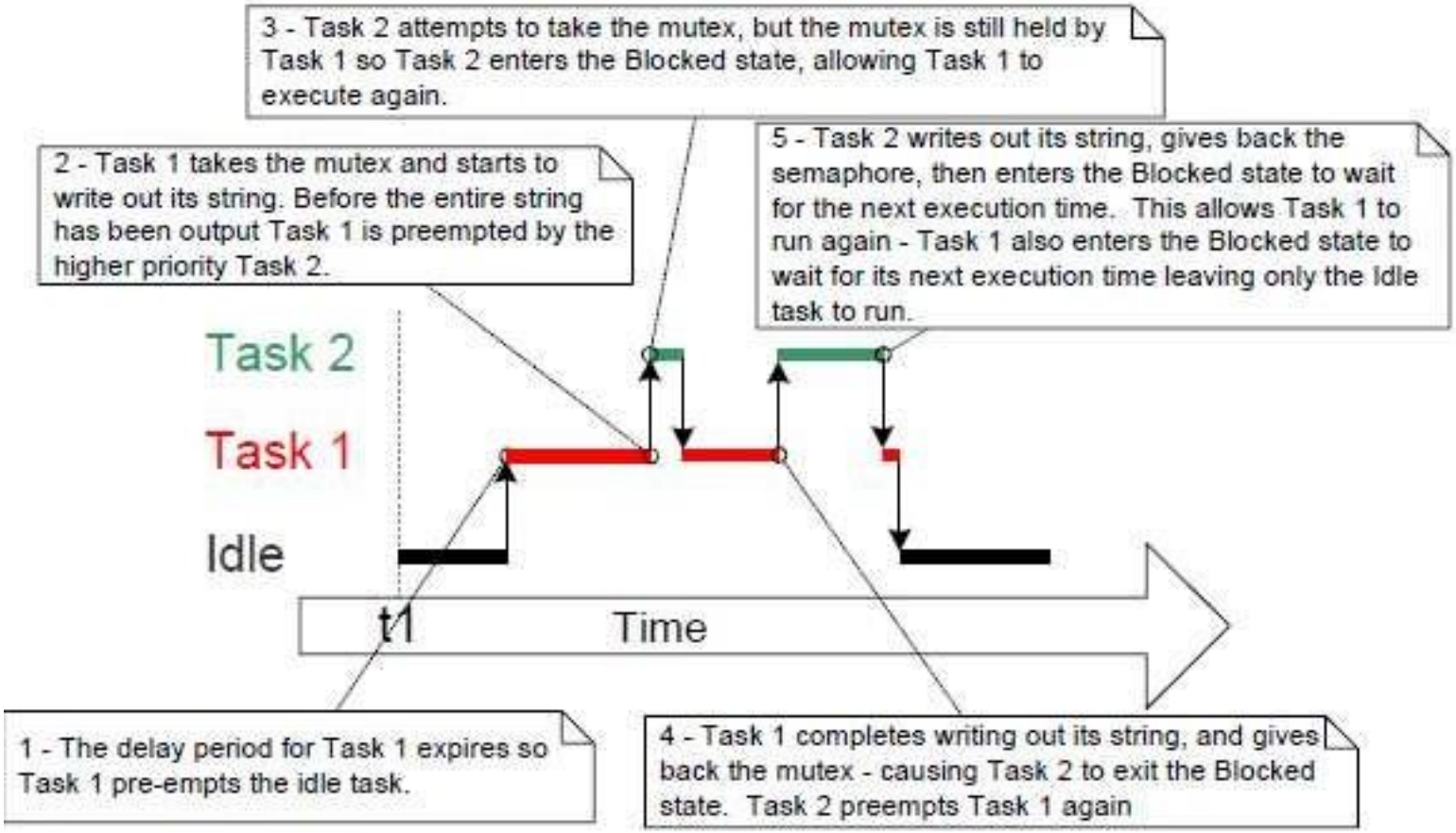


Figure 65. A possible sequence of execution for Example 20

Use of the Mutex for Solving Priority Inversion

- See Examples 21 (Homework)

End of FreeRTOS Basics

- Other topics not covered:
  - Software timers (Chapter 5)
  - Interrupt management (Chapter 6)
  - Event groups (Chapter 8)

- Homework:
  - From FreeRTOS section, review the examples and homework given and understand the task diagrams presented