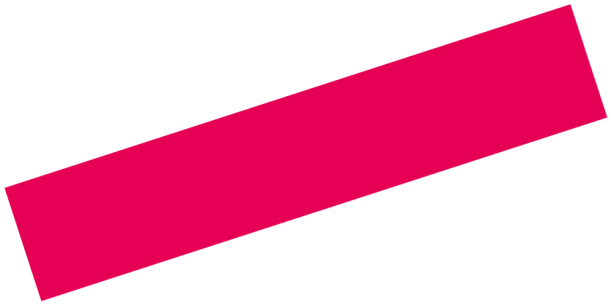
Contents

Preface



Barry, R. (2016). *Mastering the freertos real time kernel. Pre-release 161204 Edition*. Real Time Engineers Ltd.

Preface

Multitasking in Small Embedded Systems

FreeRTOS is ideally suited to deeply embedded real-time applications that use microcontrollers or small microprocessors. This type of application normally includes a mix of both hard and soft real-time requirements.

Soft real-time

Responding to keystrokes too slowly might make a system seem annoyingly unresponsive without actually making it unusable.

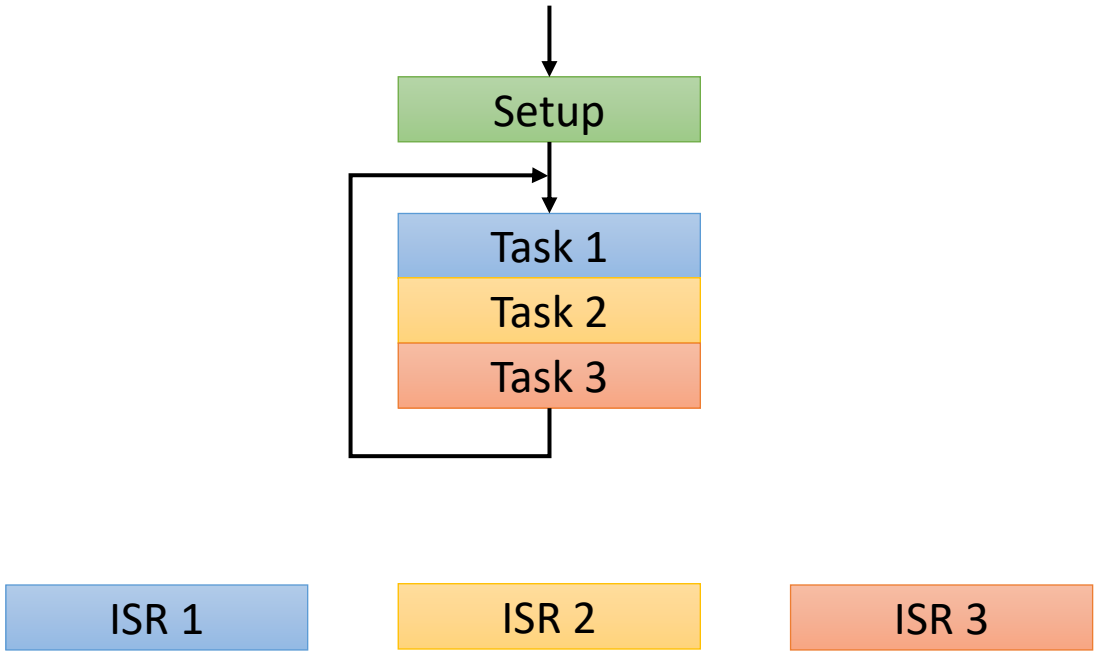
Hard real-time

A driver's airbag has the potential to do more harm than good if it responded to crash sensor inputs too slowly.

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which embedded applications can be built to meet their hard real-time requirements.

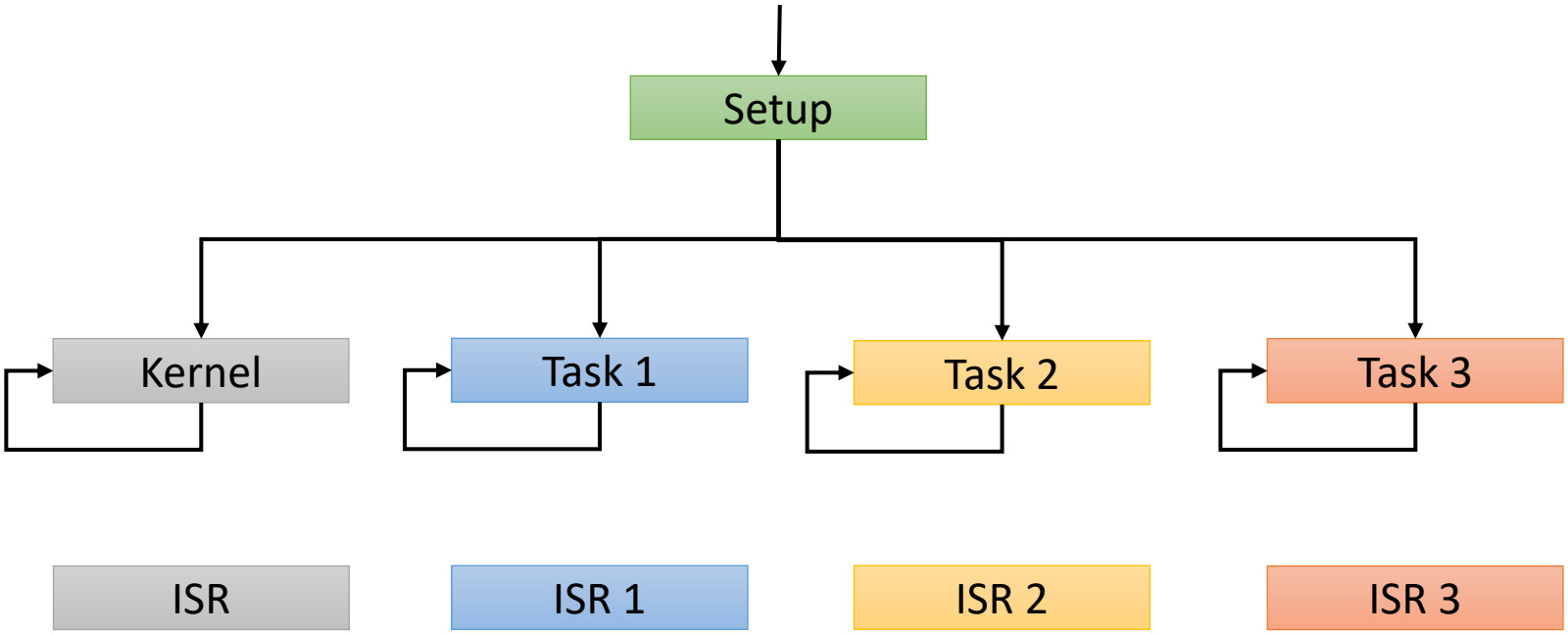
Multitasking in Small Embedded Systems

Simple scheduler, such as cyclic executive with interrupts



Multitasking in Small Embedded Systems

FreeRTOS allows applications to be organized as a collection of independent threads (referred to as a task in FreeRTOS) of execution.



Value Proposition

- Who is the creator of FreeRTOS?
- How can we use FreeRTOS in our (commercial) products?

Why Use a Real-time Kernel?

- Task prioritization
- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
- Code reuse
- Improved efficiency
- Flexible interrupt handling
- Mixed processing requirements

FreeRTOS Features

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Flexible, fast and light weight task notification mechanism
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes
- Recursive Mutexes
- Software timers
- Event groups
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace recording
- Task run-time statistics gathering
- Optional commercial licensing and support
- Full interrupt nesting model (for some architectures)
- A tick-less capability for extreme low power applications
- Software managed interrupt stack when appropriate (this can help save RAM)

Understanding the FreeRTOS distribution

Barry, R. (2016). *Mastering the freertos real time kernel. Pre-release 161204 Edition*. Real Time Engineers Ltd.
Chapter 1

Download

FreeRTOS is distributed as a single zip file archive that contains all the official FreeRTOS ports, and a large number of pre-configured demo applications.

https://www.freertos.org/index.html

Each supported combination of **compiler** and **processor** is considered to be a separate FreeRTOS port.

Files

- FreeRTOS can be thought of as a library that provides multi-tasking capabilities.
- It is supplied as a set of source and header files written in C
  - Some files are common to all ports
  - Other files are specific to a port
  - Each official FreeRTOS port comes with a demo application, which *should* build out of the box…
- FreeRTOS is configured by a single header file called **FreeRTOSConfig.h**
  - Configuration constants, such as configCPU_CLOCK_HZ
  - Minimal stack size
  - Mapping of interrupt handlers
  - Should be located in a directory which is part of the application, not in the FreeRTOS source code files
  - There is no need to create such a file from scratch, use a demo application as an example

Top level directories

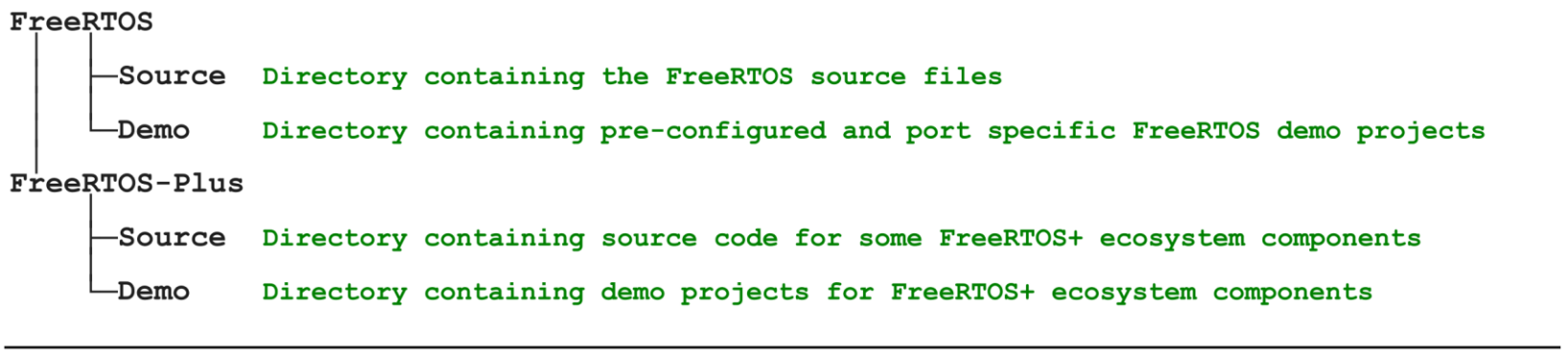


Figure 1. Top level directories within the FreeRTOS distribution

Core source files common to all ports

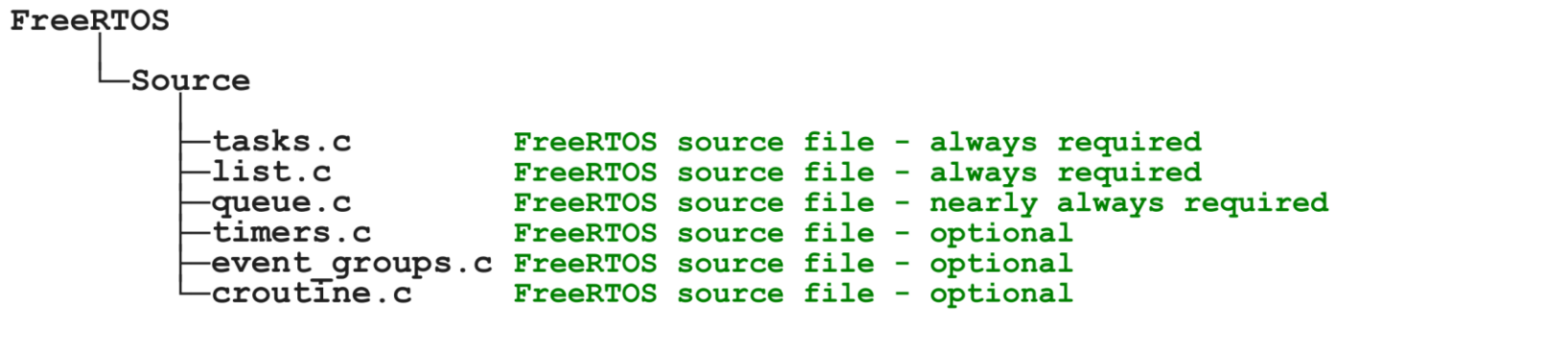


Figure 2. Core FreeRTOS source files within the FreeRTOS directory tree

Core source files specific to a port

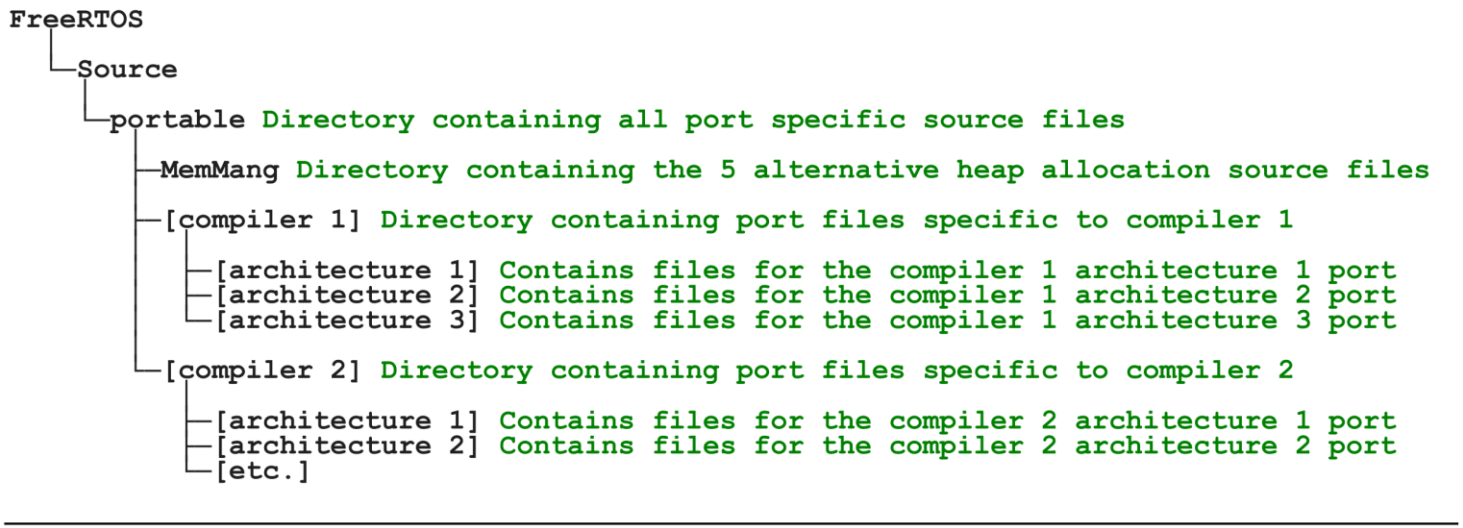


Figure 3. Port specific source files within the FreeRTOS directory tree

Core source files specific to a port

Conclusion

Running FreeRTOS on a processor with architecture ‘architecture’ using compiler ‘compiler’ then, in addition to the core FreeRTOS source files, we must also add the following files to our project:

```
FreeRTOS/Source/portable/compiler/architecture
```

All other downloaded files do not need to be part of your project!

Include paths

Three directories must be included in the compiler's include path

- `FreeRTOS/Source/include`
- `FreeRTOS/Source/portable/compiler/architecture`
- The path to FreeRTOSConfig.h header file

Using the FreeRTOS API in a source file (e.g. main.c)

Include the file FreeRTOS.h

Followed by an include for API specific functions, such as

- task.h
- queue.h
- semphr.h
- timers.h
- event_groups.h

Demo FreeRTOS for FRDM-KL25Z

- Demo Week1/Example01
- Files in the downloaded ZIP file
- Files on disk
- Files in the MCUXpresso IDE project

Summary

- Providing a top level view of the FreeRTOS directory structure.
- Describing which files are actually required by any particular FreeRTOS project.
- Introducing the demo applications.

Task management

Barry, R. (2016). *Mastering the freertos real time kernel. Pre-release 161204 Edition*. Real Time Engineers Ltd.

Chapter 3

Task functions

- Tasks are implemented as C functions
- Is a small program: has an entry point and will normally run forever.

Task functions

```
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. */
	vTaskDelete( NULL );
}
```

Simplified task states and transitions

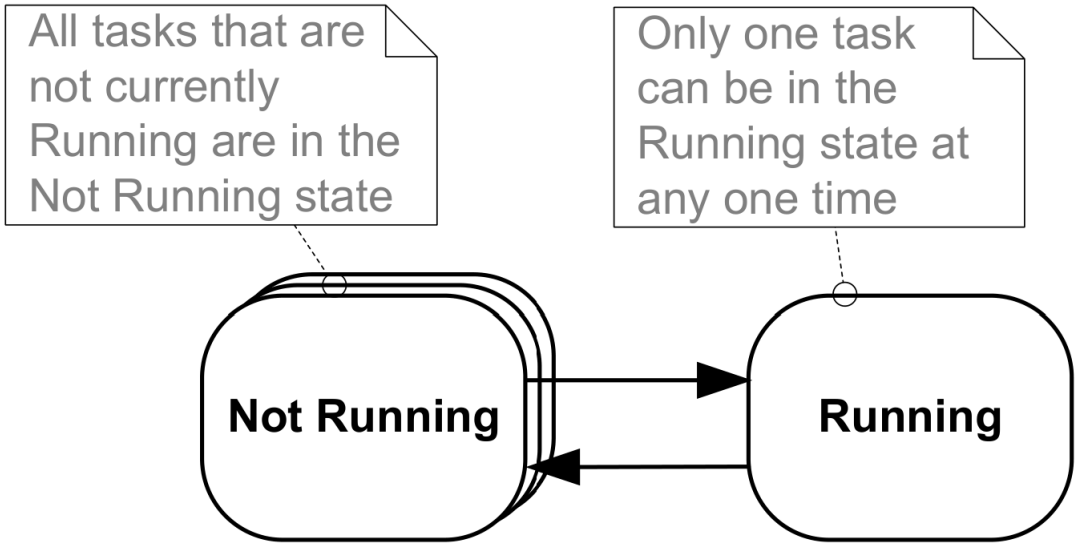


Figure 9. Top level task states and transitions

Creating tasks

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

Listing 13. The xTaskCreate() API function prototype

The pvTaskCode parameter is a pointer to the function that implements the task (in effect, just the name of the function).

Creating tasks

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

Listing 13. The xTaskCreate() API function prototype

A descriptive name for the task. This is not used by FreeRTOS in any way. It is included purely as a debugging aid for identifying a task by a human readable name.

Creating tasks

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

Listing 13. The xTaskCreate() API function prototype

Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack.

The value specifies the number of words the stack can hold, not the number of bytes.

Creating tasks

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

Listing 13. The xTaskCreate() API function prototype

Task functions accept a parameter of type pointer to void (void*). The value assigned to pvParameters is the value passed into the task.

Creating tasks

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

Listing 13. The xTaskCreate() API function prototype

Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority.

Creating tasks

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

Listing 13. The xTaskCreate() API function prototype

pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task.

If your application has no use for the task handle, then pxCreatedTask can be set to NULL.

Creating tasks

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

Listing 13. The xTaskCreate() API function prototype

1. pdPASS: task has been created successfully
2. pdFAIL: task creation failed due to insufficient memory

Creating tasks - example

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

Listing 14. Implementation of the first task used in Example 1

Creating tasks - example

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

Listing 15. Implementation of the second task used in Example 1

Creating tasks - example

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

Listing 16. Starting the Example 1 tasks

Creating tasks - example

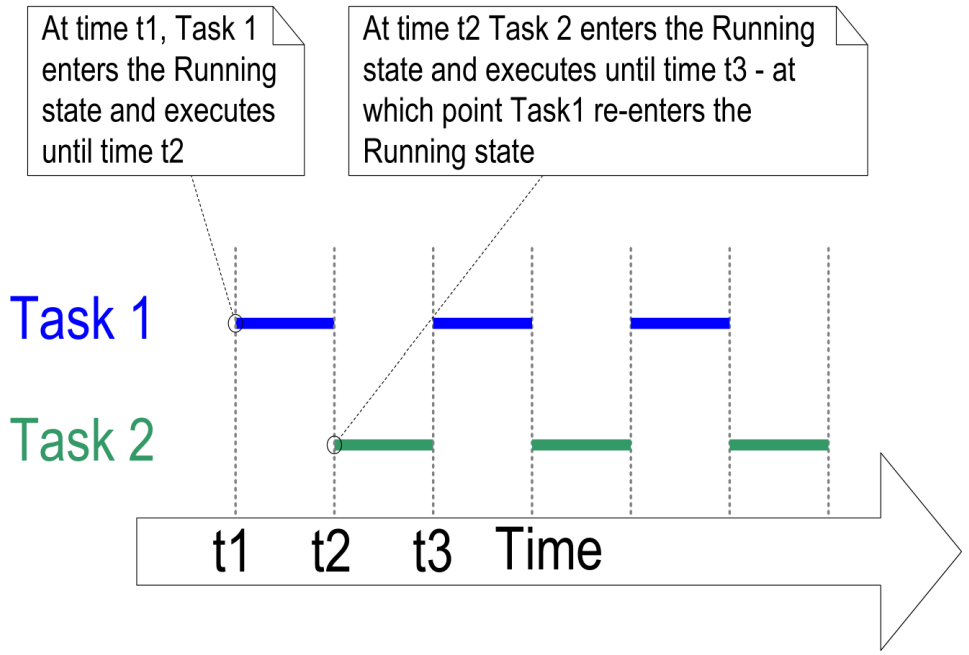


Figure 11. The actual execution pattern of the two Example 1 tasks

Task priorities

- A task priority is initially set by the uxPriority parameter of xTaskCreate() API function
- It can be changed later by the vTaskPrioritySet() API function
- Priority values
  - **0**: lowest priority
  - **configMAX_PRIORITIES - 1**: highest priority (configMAX_PRIORITIES is set by the user in FreeRTOSConfig.h)

- The scheduler ensures that highest priority task that is able to run will enter the Running state.
- Where more than one task of the same priority is able to run, the scheduler will transition each task into and out of the Running state, in turn.

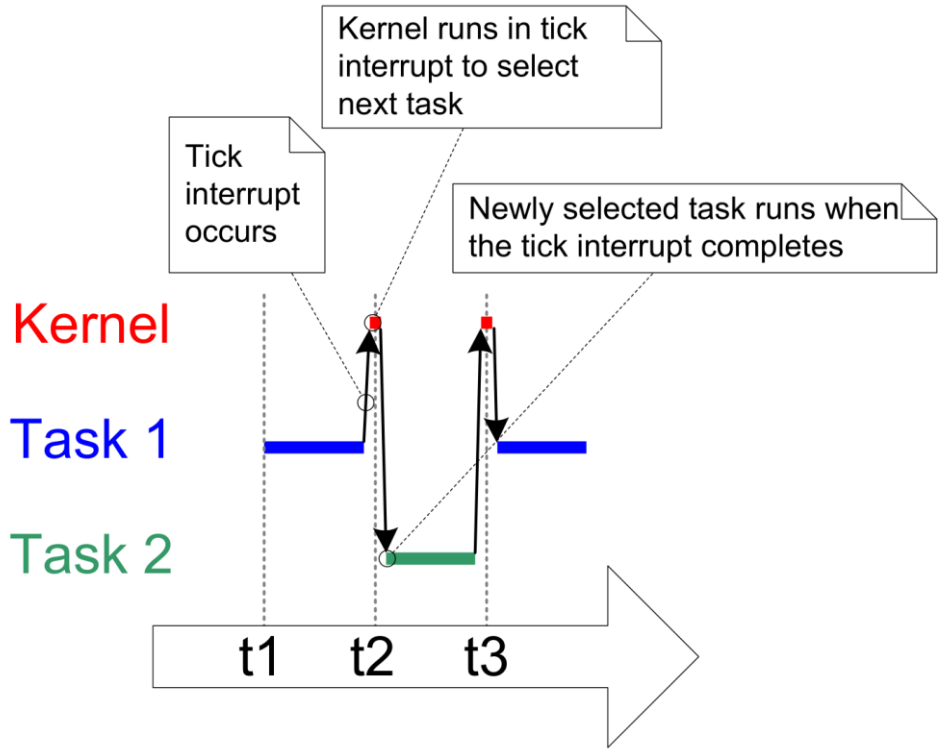
Task priorities

The FreeRTOS scheduler can use one of two methods to decide which task will be in the Running state. The maximum value to which configMAX_PRIORITIES can be set depends on the method used

- Generic Method
  - When configUSE_PORT_OPTIMISED_TASK_SELECTION is not defined or set to 0
  - Is implemented in C and can be used with all the FreeRTOS architecture ports.
  - There is no maximum to which configMAX_PRIORITIES can be set
- Architecture Optimized Method
  - When configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 1
  - Uses a small amount of assembler code
  - Is faster than the generic method
  - configMAX_PRIORITIES cannot be greater than 32

Time Measurement and the Tick Interrupt

A tick interrupt is used to allow the scheduler to select the next task



Time Measurement and the Tick Interrupt

- Set configTICK_RATE_HZ to define the tick interrupt frequency
- The time between two tick interrupts is called the *tick period*
- One time slice equals one tick period
- Although the tick frequency is application dependent, a typical value is 100 Hz.

- FreeRTOS API calls always specify time in multiples of tick periods a.k.a. *ticks*

```
/* pdMS_TO_TICKS() takes a time in milliseconds as its only parameter, and evaluates
to the equivalent time in tick periods. This example shows xTimeInTicks being set to
the number of tick periods that are equivalent to 200 milliseconds. */
TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

Listing 20. Using the pdMS_TO_TICKS() macro to convert 200 milliseconds into an equivalent time in tick periods

Priorities example

```
/* 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;
}
```

Listing 21. Creating two tasks at different priorities

Priorities example

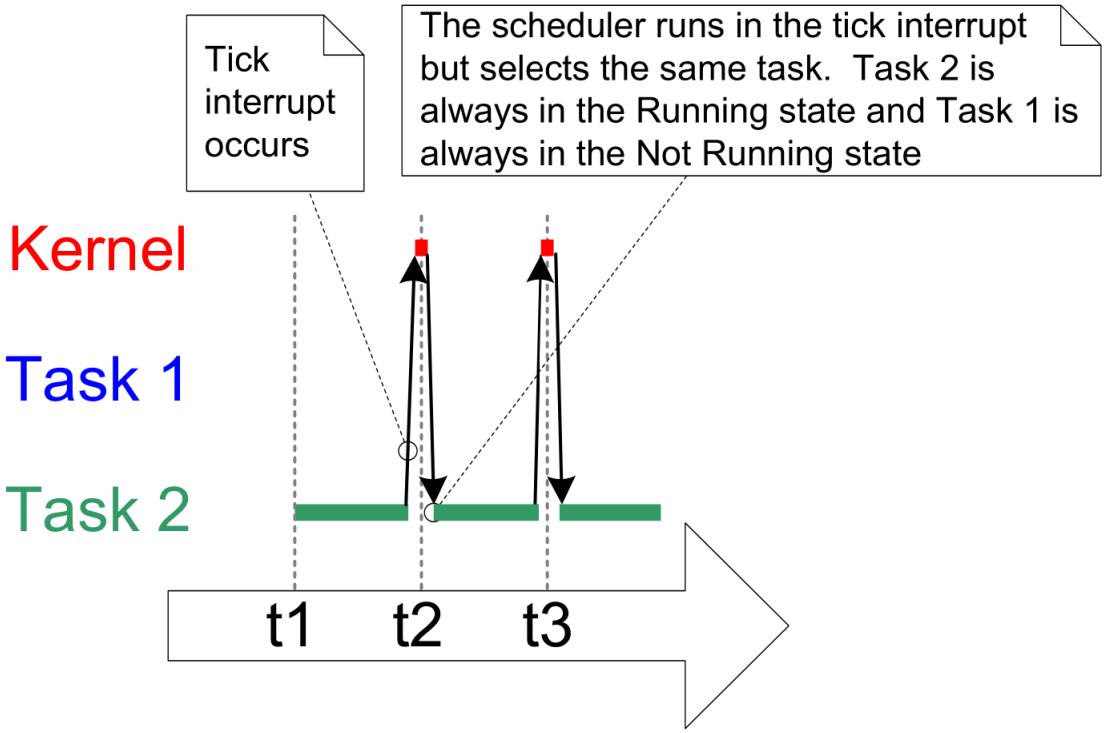


Figure 14. The execution pattern when one task has a higher priority than the other

Full task state machine

- As seen in the previous example: the scheduler always selects the highest priority task that **is able to run**
- But we don't want the highest priority tasks starving all the lower priority tasks of processing time
- Solution: tasks should be event-driven
  - If a high-priority task event trigger occurs, the task is handled immediately.
  - If a low-priority task event trigger occurs, it is no problem if handling the task is delayed.

- A task that is waiting for an event is said to be in the Blocked state

Full task state machine

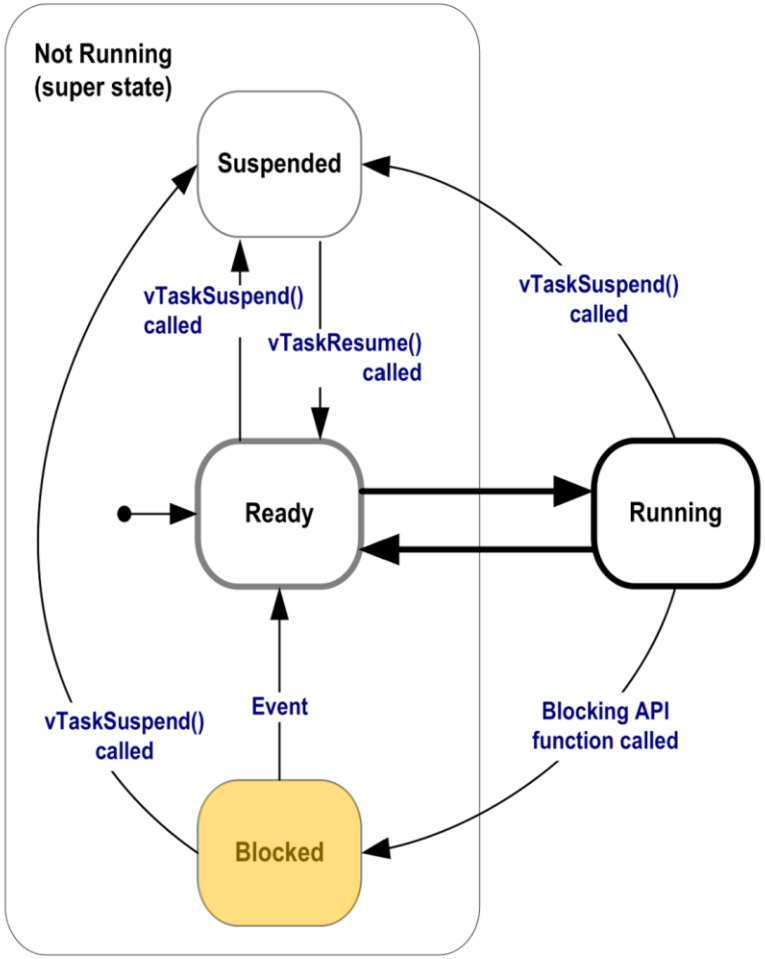


Figure 15. Full task state machine

Blocked state

- A task that is waiting for an event is said to be in the Blocked state
- Two types of events can make a task enter the Blocked state:
  - Temporal (time-related) events
  - Synchronization events
  - (Or a combination of both, e.g., when waiting for a maximum of 10 milliseconds for data to arrive.)

Full task state machine

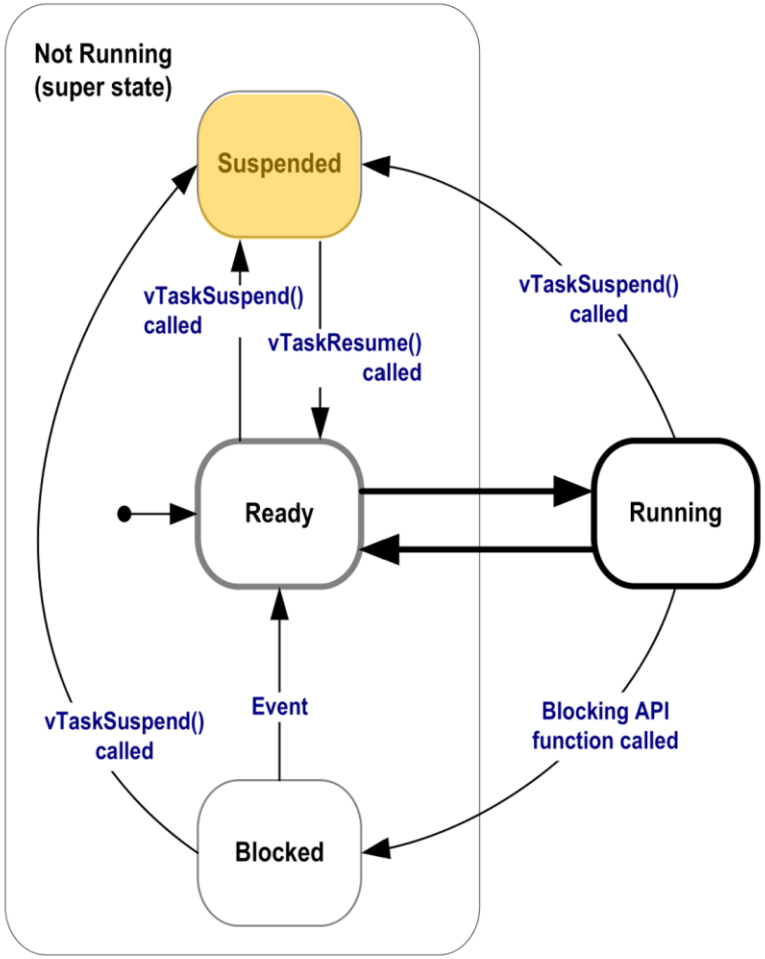


Figure 15. Full task state machine

Suspend state

- Tasks in the Suspended state are not available to the scheduler
- Enter suspend state: vTaskSuspend()
- Leave suspend state: vTaskResume() or vTaskResumeFromISR()

Full task state machine

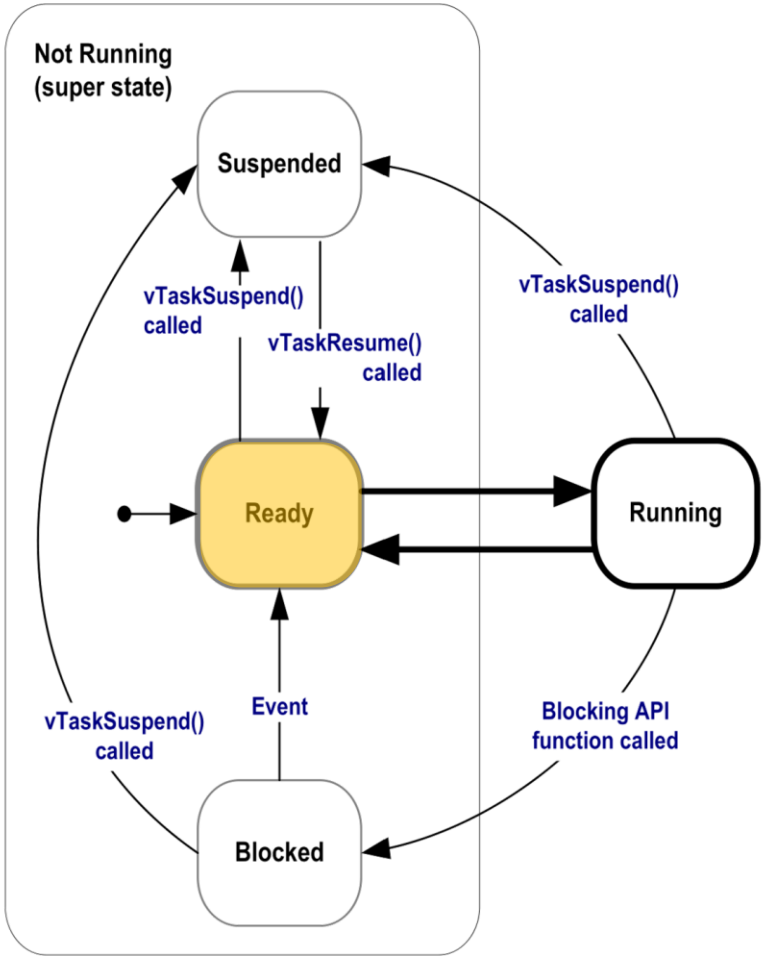


Figure 15. Full task state machine

Ready state

- Tasks that are in the Not Running state but are not Blocked or Suspended are said to be in the Ready state

Blocked state example

```
void vTaskDelay( TickType_t xTicksToDelay );
```

Listing 22. The vTaskDelay() API function prototype

The number of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into the Ready state.

The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks.

Blocked state example

```
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. */
        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 );
    }
}
```

Listing 23. The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()

Blocked state example

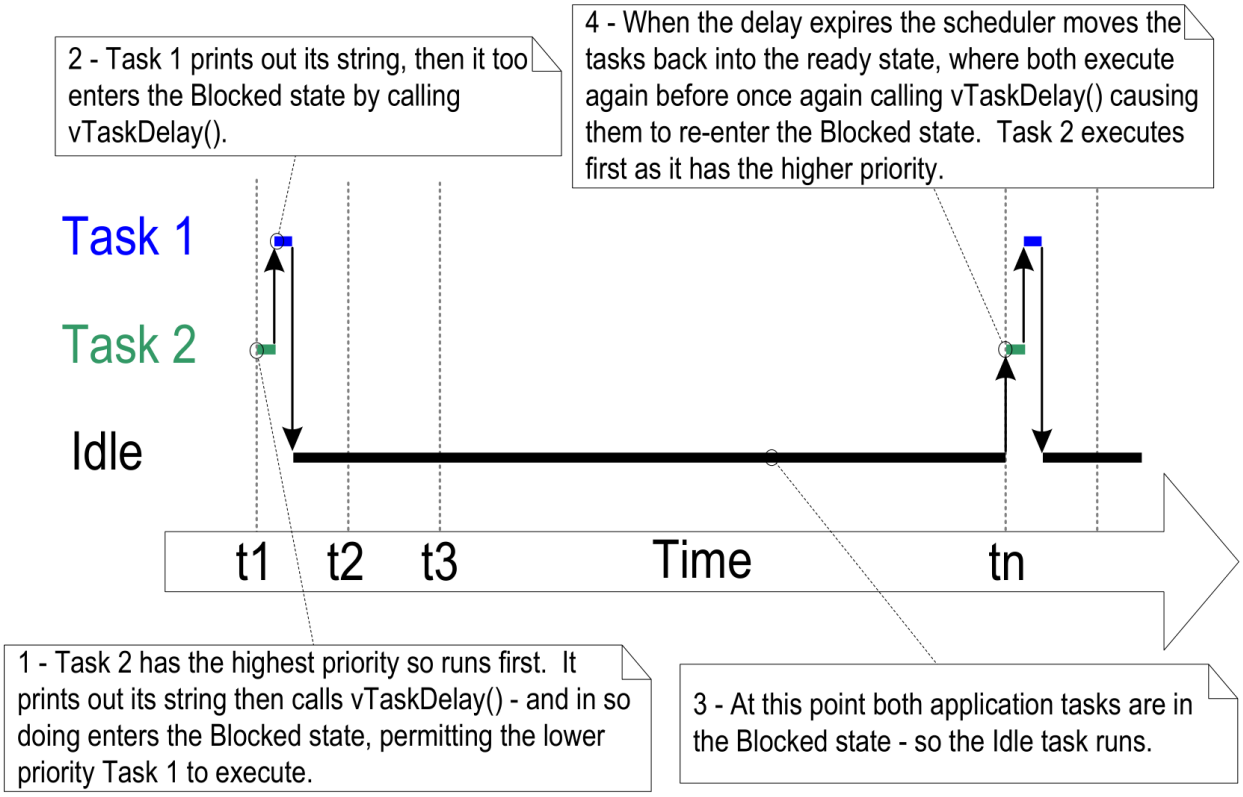
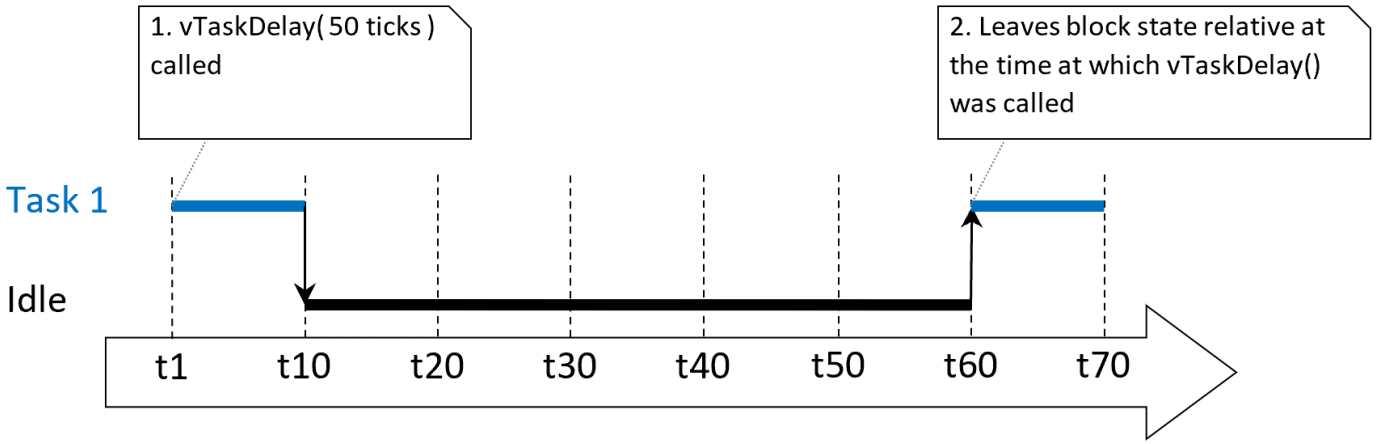


Figure 17. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

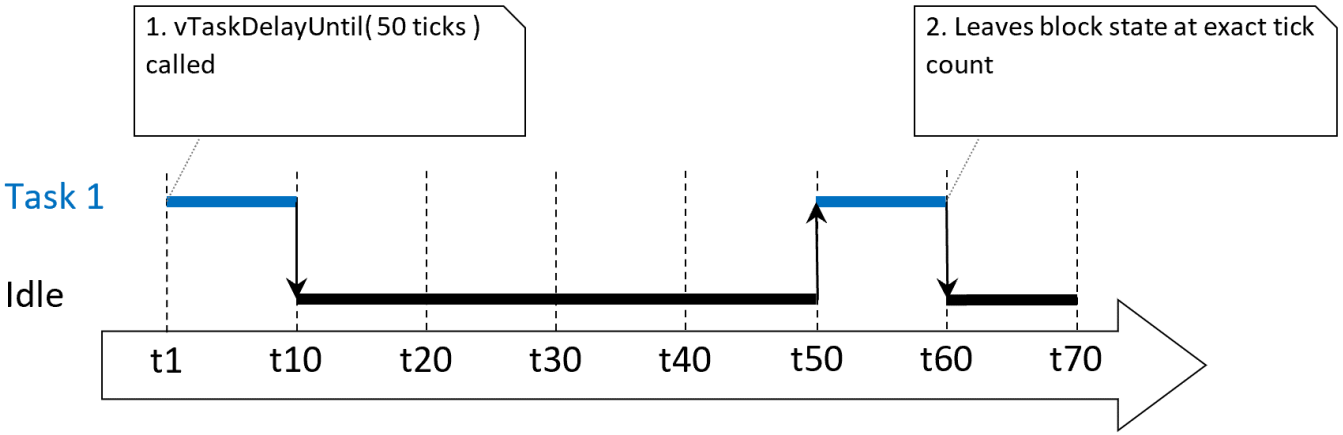
The vTaskDelayUntil() API Function

- The length of time the task remains in the blocked state is specified by the vTaskDelay() parameter
- The time at which the task leaves the blocked state is relative to the time at which vTaskDelay() was called



The vTaskDelayUntil() API Function

- The parameters to vTaskDelayUntil() specify, instead, the exact tick count value at which the calling task should be moved from the Blocked state into the Ready state



The vTaskDelayUntil() API Function

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

Listing 24. vTaskDelayUntil() API function prototype

pxPreviousWakeTime holds the time at which the task last left the Blocked state (was 'woken' up). This time is used as a reference point to calculate the time at which the task should next leave the Blocked state.

The variable pointed to by pxPreviousWakeTime is updated automatically within the vTaskDelayUntil() function!

The vTaskDelayUntil() API Function

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

Listing 24. vTaskDelayUntil() API function prototype

Sets the frequency at which a periodic task should be executed. A value of 100 ticks, for example, means that the task will leave the blocking state at every 100th tick.

The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks.

The vTaskDelayUntil() example

```
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 ) );
    }
}
```

Listing 25. The implementation of the example task using vTaskDelayUntil()

The Idle Task and the Idle Task Hook

- There must always be at least one task that can enter the Running state
- This is called the **Idle task**
- Priority of the Idle task is 0
- If there are other tasks with priority 0, configIDLE_SHOULD_YIELD can be used to prevent the Idle task from consuming processing time

The Idle Task and the Idle Task Hook

- Application specific functionality can be added into the Idle task by using the so-called Idle hook (a.k.a. a callback) function
- Common uses are:
  - Executing low priority, background, or continuous processing functionality
  - Measuring the amount of spare processing capacity
  - Placing the processor into a low power mode
- Must never attempt to block or suspend

```
void vApplicationIdleHook( void );
```

Listing 28. The idle task hook function name and prototype

Demo FreeRTOS for FRDM-KL25Z

- Demo Week1/Example02

Scheduling algorithms

- The scheduling algorithm is the software routine that decides which Ready state task can be transitioned into the Running state
- FreeRTOS scheduler ensures **Round Robin Scheduling**: Ready state tasks of equal priority will enter the Running state in turn
- Two options configure the scheduling algorithm:

configUSE_PREEMPTION	configUSE_TIME_SLICING	Description
0	X	Cooperative scheduling
1	0	Prioritized Pre-emptive Scheduling without Time Slicing
1	1	Fixed Priority Pre-emptive Scheduling with Time Slicing [1)]

1) Used by most small RTOS applications

Definitions

Fixed Priority

The scheduling software does not change the priority assigned to the tasks being scheduled.

Definitions

Pre-emptive

Pre-emptive scheduling algorithms will immediately ‘pre-empt’ the Running state task if a task that has a priority higher than the Running state task enters the Ready state.

Being pre-empted means being involuntarily moved out of the Running state and into the Ready state to allow a different task to enter the Running state.

Definitions

Time Slicing

Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state.

A new task is selected to enter the Running state at the *end of each* time slice if there are other Ready state tasks that have the same priority as the Running task.

A time slice is equal to the time between two RTOS tick interrupts.

Fixed Priority Pre-emptive Scheduling with Time Slicing

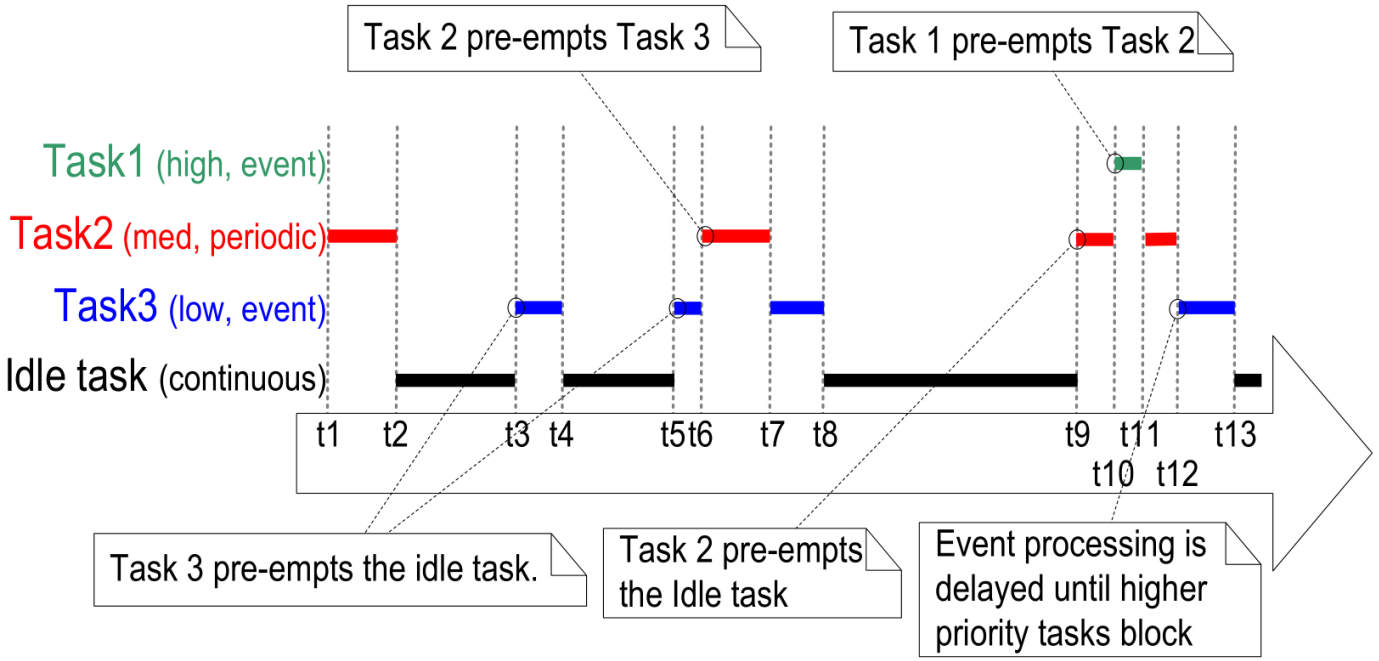


Figure 26. Execution pattern highlighting task prioritization and pre-emption in a hypothetical application in which each task has been assigned a unique priority

Prioritized Pre-emptive Scheduling without Time Slicing

Does not use time slicing to share processing time between tasks of equal priority.

New task to enter the Running state when either:

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

When compared to using time slicing:

- Advantage: less scheduling overhead
- Disadvantage: can result in tasks of equal priority receiving greatly different amounts of processing time

Co-operative Scheduling

A context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields by calling taskYIELD().

Tasks are never pre-empted, so time slicing cannot be used

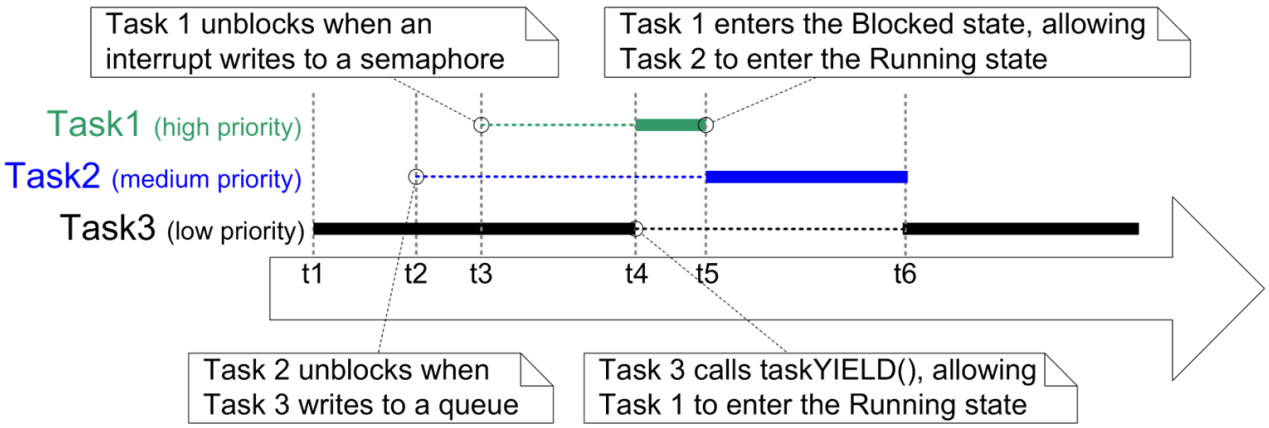


Figure 30 Execution pattern demonstrating the behavior of the co-operative scheduler

Co-operative Scheduling

When compared to a pre-emptive scheduler:

- Advantage: application designer has full control over context switches, which generally helps to avoid problems caused by simultaneously accessing resources

- Disadvantage: less responsive system

Scheduling algorithms

Summary

configUSE_PREEMPTION	configUSE_TIME_SLICING	Description
0	X	Cooperative scheduling
1	0	Prioritized Pre-emptive Scheduling without Time Slicing
1	1	Fixed Priority Pre-emptive Scheduling with Time Slicing [1)]

1) Used by most small RTOS applications

Summary

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