Task Management in FreeRTOS

Mojtaba Bagherzadeh, Adrien Lapointe

Royal Military College (RMC)

mojtaba@cs.queensu.ca,adrien.lapointe@rmc.ca

February 16, 2018

Overview

Recall

Task

A task is implemented as a C function and must return void and take a void pointer parameter. It has an entry point, will normally run forever within an infinite loop, and will not exit.

Recall

Task

A task is implemented as a C function and must return void and take a void pointer parameter. It has an entry point, will normally run forever within an infinite loop, and will not exit.

Task Instance

A single task function definition can be used to create any number of tasks. Each created task being a separate execution instance, with its own stack defined within the task itself.

Recall

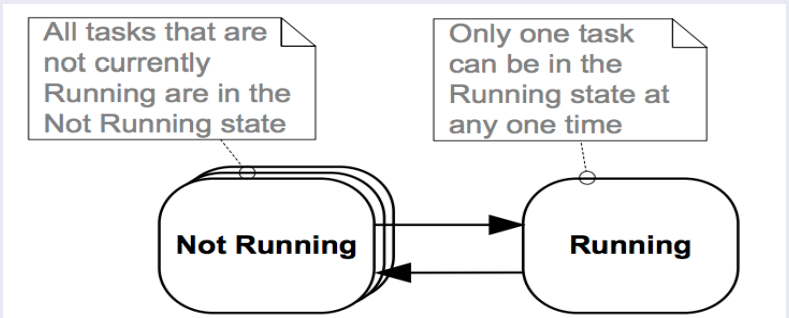
Task

A task is implemented as a C function and must return void and take a void pointer parameter. It has an entry point, will normally run forever within an infinite loop, and will not exit.

Task Instance

A single task function definition can be used to create any number of tasks. Each created task being a separate execution instance, with its own stack defined within the task itself.

Task Status



A Typical Task Function

```
void ATaskFunction( void *pvParameters ){
        /* Variables can be declared just as per a normal
            function. */
        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 );
}
```

Creating Tasks

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

Argument	Description
pvTaskCode	name of the task function
pcName	task name
usStackDepth	size of stack for the task
pvParameters	a value which is passed into the task function.
uxPriority	priority of the task that can be assigned from 0 (the lowest priority) to `configMAX_PRIORITIES - 1`.
pxCreatedTask	Can be used to pass out a handle to the task being created.

Creating Tasks

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

Argument	Description
pvTaskCode	name of the task function
pcName	task name
usStackDepth	size of stack for the task
pvParameters	a value which is passed into the task function.
uxPriority	priority of the task that can be assigned from 0 (the lowest priority) to `configMAX_PRIORITIES - 1`.
pxCreatedTask	Can be used to pass out a handle to the task being created.

Creating Tasks

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

Argument	Description
pvTaskCode	name of the task function
pcName	task name
usStackDepth	size of stack for the task
pvParameters	a value which is passed into the task function.
uxPriority	priority of the task that can be assigned from 0 (the lowest priority) to `configMAX_PRIORITIES - 1`.
pxCreatedTask	Can be used to pass out a handle to the task being created.

Return values of the function:

- `pdPASS`: the task has been created successfully.
- `pdFAIL`: the task has **not** been created .

Task Parameter

- Use task parameter to differentiate between different instances of a task.

Task Parameter

- Use task parameter to differentiate between different instances of a task.
- You can pass every data type using `void *`, and use type casting inside the implementation function.

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

Task Priorities

- The maximum number of priorities available is set by the application-defined `configMAX_PRIORITIES` within `FreeRTOSConfig.h`. (The range of available priorities is 0 to `configMAX_PRIORITIES-1`)

Task Priorities

- The maximum number of priorities available is set by the application-defined `configMAX_PRIORITIES` within `FreeRTOSConfig.h`. (The range of available priorities is 0 to `configMAX_PRIORITIES-1`)
- Low numeric priority values denote low-priority tasks.

Task Priorities

- The maximum number of priorities available is set by the application-defined `configMAX_PRIORITIES` within `FreeRTOSConfig.h`. (The range of available priorities is 0 to `configMAX_PRIORITIES-1`)
- Low numeric priority values denote low-priority tasks.
- The FreeRTOS scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state.

Time Measurement

- ***Tick interrupt*** is a periodic interrupt which is configured by the application-defined `configTICK_RATE_HZ`. The time between two tick interrupts is called the ***tick period***. Tick period is used to measure the time.

Time Measurement

- ***Tick interrupt*** is a periodic interrupt which is configured by the application-defined `configTICK_RATE_HZ`. The time between two tick interrupts is called the ***tick period***. Tick period is used to measure the time.
- FreeRTOS API calls always specify time in multiples of tick periods, which are often referred to simply as 'ticks'.

Time Measurement

- ***Tick interrupt*** is a periodic interrupt which is configured by the application-defined `configTICK_RATE_HZ`. The time between two tick interrupts is called the ***tick period***. Tick period is used to measure the time.
- FreeRTOS API calls always specify time in multiples of tick periods, which are often referred to simply as 'ticks'.
- The `pdMS_TO_TICKS()` macro converts a time specified in milliseconds into a time specified in ticks.

Create a Periodic Task

null loop

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

- The task always will be in running state and will consume the processing resource.
- The higher priority task remained in the Running state while it executed the null loop, starving the lower priority task of any processing time.

Create a Periodic Task

Use FreeRTOS timing API

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

- The task goes to non-running (blocking) state and is triggered when the specified time is elapsed. The task does not use any processing time while it is in the Blocked state.

vTaskDelay Function

vTaskDelay places the calling task into the Blocked state for a fixed number of tick interrupt.

Example

If a task call `vTaskDelay( 100 )` when the tick count was 10,000, then it would immediately enter the Blocked state, and remain in the Blocked state until the tick count reached 10,100.

Hint

- Set `INCLUDE_vTaskDelay` is set to 1 in FreeRTOSConfig.h.
- The macro `pdMS_TO_TICKS()` can be used to convert a time specified in milliseconds into a time specified in ticks. For example, calling `vTaskDelay( pdMS_TO_TICKS( 100 ))` will result in the calling task remaining in the Blocked state for 100 milliseconds.

vTaskDelayUntil Function

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

`vTaskDelayUntil()` can be used when a fixed execution period is required, as the time at which the calling task is blocked is absolute, rather than relative to when the function was called (as is the case with `vTaskDelay()`).

Argument	**Description**
pxPreviousWakeTime	holds the time at which the task last left the Blocked state (was 'woken' up).
xTimeIncrement	set next waken up time value as tick count.

vTaskDelay Function Example

```
void vPeriodicTask( void *pvParameters ){
 const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );
 for( ;; ){
         // some operations
    // The task execute every 3 milliseconds exactly
         vTaskDelay( xDelay3ms );
 }
}
```

vTaskDelay Function Example

```
void vPeriodicTask( void *pvParameters ){
 const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );
 for( ;; ){
         // some operations
    // The task execute every 3 milliseconds exactly
         vTaskDelay( xDelay3ms );
 }
}
```

Note

The task period is exactly 3 ms + the execution time of the task.

vTaskDelayUntil Function Example

```
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();

 for( ;; ){
         // some operations
         vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
 }
}
```

vTaskDelayUntil Function Example

```
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();

 for( ;; ){
         // some operations
         vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
 }
}
```

Note

The task period is exactly 3ms.

Idle Task

- An Idle task is automatically created by the scheduler when `vTaskStartScheduler()` is called and executed when there is no other task in running state.

Idle Task

- An Idle task is automatically created by the scheduler when `vTaskStartScheduler()` is called and executed when there is no other task in running state.
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state.

Idle Task

- An Idle task is automatically created by the scheduler when `vTaskStartScheduler()` is called and executed when there is no other task in running state.
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state.
- Idle task is responsible for cleaning up kernel resources after a task has been deleted.

Idle Task

- An Idle task is automatically created by the scheduler when `vTaskStartScheduler()` is called and executed when there is no other task in running state.
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state.
- Idle task is responsible for cleaning up kernel resources after a task has been deleted.
- It is possible to add application-specific functionality directly into the idle task through the use of an idle hook function.

Idle Task

- An Idle task is automatically created by the scheduler when `vTaskStartScheduler()` is called and executed when there is no other task in running state.
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state.
- Idle task is responsible for cleaning up kernel resources after a task has been deleted.
- It is possible to add application-specific functionality directly into the idle task through the use of an idle hook function.
- Idle hook function is called automatically by the idle task once per iteration of the idle task loop.

Idle Task

- An Idle task is automatically created by the scheduler when `vTaskStartScheduler()` is called and executed when there is no other task in running state.
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state.
- Idle task is responsible for cleaning up kernel resources after a task has been deleted.
- It is possible to add application-specific functionality directly into the idle task through the use of an idle hook function.
- Idle hook function is called automatically by the idle task once per iteration of the idle task loop.
- Example uses for the Idle task hook include execution of background functionalities, measuring the amount of spare processing capacity, placing the processor into a low power mode.

Idle Hook Function

Prototype

```
void vApplicationIdleHook( void )
```

Note

- Set `configUSE_IDLE_HOOK` to 1.
- An Idle task hook function must never attempt to block or suspend.
- If the application makes use of the `vTaskDelete()` API function, then the Idle task hook must always return to its caller within a reasonable time period.

Other Useful APIs

Chnage a Task Priority

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority)
```

Delete a Task

```
void vTaskDelete( TaskHandle_t pxTaskToDelete )
```

Suspend a Task

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend )
```

Resume a Task

```
void vTaskResume( TaskHandle_t xTaskToResume )
```

References

Richard Barry. Mastering the FreeRTOS Real Time Kernel. FreeRTOS.org, 2016

Question?