AN12881

Motor Control Using FreeRTOS

Rev. 0 — May, 2020 Application Note

by: NXP Semiconductors

Contents

1 Introduction

This application note describes an example use of FreeRTOS in a motor control application. It is primarily targeted as an addendum to AN12235 [1], describing the design of the MCSPTE1AK144 development kit [2], but it can be used as a general guideline for any motor control application.

As a non-OS motor control application is normally interrupt-driven (as described in [1]), application task function execution is usually tied to a periodical interrupt that is synchronous to the PWM signal driving the motor, e.g. ADC interrupt. Application tasks are then executed directly inside the interrupt service routine and in the main function endless loop.

The complexity of an application increases with the number of task functions being used in the application. This may result in an extended number of interrupts and an extended duration of interrupt service routines (ISR) where the application tasks are executed, or in a too complex and time-wasting polling of state variables in the main function endless loop.

FreeRTOS offers powerful tools to manage application tasks without the need to use peripheral interrupts, including priority pre-emption mechanisms.

2 Motor control application tasks

A typical motor control application can be divided into the following tasks:

- Peripheral configuration
- User input detection
- Application fault detection
- MOSFET pre-driver fault detection
- Feedback quantities acquisition
- Motor control state machine
- LED indication
- External communication processing (e.g. CAN, LIN, FreeMASTER)
- FreeMASTER recorder

2.1 Non-OS based application

In a non-OS based motor control application, tasks are normally distributed between the main function and an ADC or PWM interrupt service routine. The distribution is summarized in the following table.

Table 1. Non-OS based application task function distribution

Application task	Executed in
Peripheral configuration	main()

Table continues on the next page...

Table 1. Non-OS based application task function distribution (continued)

Application task	Executed in
MOSFET pre-driver fault detection	main() endless loop
External communication processing	
User input detection	ADC interrupt service routine
Feedback quantities acquisition	
Fault detection	
Motor control state machine	
LED indication	
FreeMASTER recorder	

A typical timing of the main() function and an ADC interrupt in a non-OS based motor control application with a 100 µs fast current-control loop is shown in the following figure. Every n^{th} ADC ISR execution time is longer as the slow speed loop is also processed.

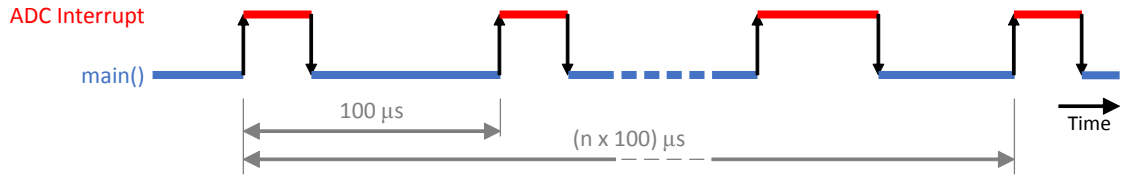

Figure 1. Non-OS application task timing

As a fast reaction to the ADC interrupt is required for precise motor control, timing-critical tasks are executed in the ISR instead of respective peripheral register flag polling in the main() function endless loop.

2.2 FreeRTOS based application

To minimize time spent in an ISR, deferred interrupt handling is supported by FreeRTOS. The following example uses application controlled deferred interrupt handling as it allows control over deferred processing task priority, and offers reduced latency compared to centralized deferred interrupt handling [3].

The following table summarizes distribution of application tasks in respective FreeRTOS tasks.

Table 2. FreeRTOS based application task function distribution

Application task	FreeRTOS task / CPU interrupt	FreeRTOS task priority[1]
Peripheral configuration	Init Task	4
MOSFET pre-driver fault detection	Main Task	2
External communication processing		
Feedback quantities acquisition	ADC Interrupt Service Routine	-

Table continues on the next page...

Table 2. FreeRTOS based application task function distribution (continued)

Application task	FreeRTOS task / CPU interrupt	FreeRTOS task priority[1]
Fault detection		
Motor control state machine	State Machine Task	7 = (configMAX_PRIORITIES - 1)
FreeMASTER recorder		
User input detection	RTOS Daemon Task (software timer callback)	3 = configTIMER_TASK_PRIORITY
LED indication		

1. The maximum FreeRTOS task priority is defined by configMAX_PRIORITIES value in FreeRTOSConfig.h.

The timing of the FreeRTOS based application tasks is shown in the following figure.

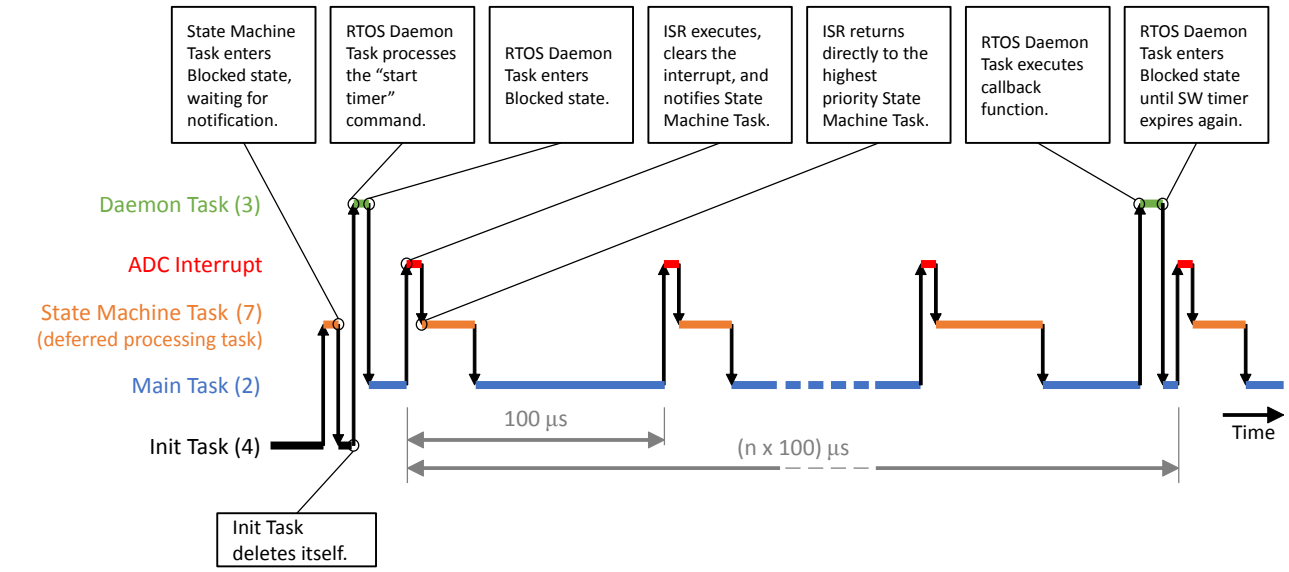

Figure 2. Free-RTOS based application timing

2.2.1 FreeRTOS task creation

A FreeRTOS task can be created by the xTaskCreate() API function. The prototype of the function is the following:

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

The following table lists the input parameters of the xTaskCreate() function.

Table 3. xTaskCreate() input parameters

xTaskCreate() input parameter	Description
pxTaskCode	Pointer to a function that implements the task.
pcName	Name of the task. The length of the task name including NULL terminator is limited by configMAX_TASK_NAME_LEN value defined in FreeRTOSConfig.h
usStackDepth	Task stack size in the number of words the stack can hold (e.g. number of 4 byte words on Cortex-M4).
pvParameters	Task function input parameter of type (void *). The value assigned to the parameter will be passed into the function implementing the task.
uxPriority	Task priority in the range of 0, which is the lowest priority, to (configMAX_PRIORITIES - 1) which is the highest priority.
pxCreatedTask	Pointer for passing out the task handle which can be used to reference the task in API calls.

2.2.2 Init task

The Init Task (initialization task) is created in the main() function before the FreeRTOS scheduler is started by the vTaskStartScheduler() API function call.

- Example 1: Init Task Creation

```
/* Create Init Task */
ret = xTaskCreate(pmsm_init,"pmsm_init_tsk",256u,NULL,4u,NULL);

/* Start FreeRTOS scheduler */
vTaskStartScheduler();
```

The Init Task is used to perform peripheral configuration, create the Main Task and State Machine Task, and initialize the software timer. Once the Main Task is created by the Init Task, it enters the Ready state, but as it has lower priority compared to the Init Task in the Running state, it remains in the Ready state.

- Example 2: Main Task creation

```
/* Main Task function prototype */
void pmsm_main(void* pvParameters);

BaseType_t ret;

/* Create Main Task for fault checking and external communication processing */
ret = xTaskCreate(pmsm_main, "pmsm_main_tsk", 128u, NULL, 2u, NULL);
```

Once the State Machine Task is created by the Init Task, it immediately pre-empts Init Task as it has the higher priority. It starts executing an assigned task function where it enters the Blocked state after calling the ulTaskNotifyTake() function, releasing control back to the Init Task. The State Machine Task is further described in section State Machine Task.

- Example 3: State Machine Task creation

```
/* State Machine Task function prototype */
void pmsm_state(void* pvParameters);

TaskHandle_t  PmsmStateHandle;
```

```
BaseType_t ret;

/* Create State Machine Task (ADC ISR will defer to) */
ret = xTaskCreate(pmsm_state, "pmsm_state_tsk", 128u, NULL, (configMAX_PRIORITIES - 1),
      &PmsmStateHandle);
```

The software timer is created by xTimerCreate() and started by the xTimerStart() function call which sends the “start timer” command to the timer command queue, causing the RTOS Daemon task to transition from the Blocked to Ready state every millisecond. The Daemon Task is further described in section RTOS Daemon Task.

- Example 4: FreeRTOS software timer initialization example

```
/* Software timer callback function prototype */
void PeriodicalTimerCbk(TimerHandle_t xTimer);

TimerHandle_t PeriodicalTimer;
BaseType_t ret;

/* Create 1ms periodical software timer with PeriodicalTimerCbk() callback function */
PeriodicalTimer = xTimerCreate("PeriodicalTimer", pdMS_TO_TICKS(1), pdTRUE, (void *)0,
                  PeriodicalTimerCbk);

/* Start software timer with no block time specified */
ret = xTimerStart(PeriodicalTimer, 0);
```

Once all necessary tasks and the software timer is created, Init Task deletes itself by calling the vTaskDelete(NULL) API function.

2.2.3 RTOS daemon task

The RTOS daemon task is a standard task that is automatically created by the FreeRTOS when the scheduler is started by the vTaskStartScheduler() function call. The daemon task is created with a priority of configTIMER_TASK_PRIORITY and a stack size of configTIMER_TASK_STACK_DEPTH defined in FreeRTOSConfig.h. These are then common for all software timers created.

The daemon task is used to service software timer commands that are being sent from the calling task into the daemon task using a timer command queue. In this application, the software timer periodically executes a callback function responsible for user input detection and LED control.

As shown in Figure 2, a software timer is created and enabled by the Init Task. As the daemon task has lower priority than the Init Task, it processes the “start timer” command in the timer command queue once the Init Task deletes itself by a vTaskDelete(NULL) function call, allowing the daemon task to transition from the Ready to Running state. After processing the “start timer” command, the daemon task enters the Blocked state.

Once the software timer expires, the daemon task enters the Ready state again. If it is currently the highest priority task in the Ready state and there is no higher priority task in the Running state, it enters the Running state and executes the PeriodicalTimerCbk() callback function. Otherwise, it must wait for the higher priority task to enter the Block state first.

NOTE

The software timer is a suitable FreeRTOS feature to trigger periodical events with the minimum of system resources spent. However, in the use of multiple software timers, all configured timers rely on the RTOS daemon task with its own priority. If separate priority levels are desired by the application, the use of separate OS tasks, although with a higher final resource cost, is recommended instead.

2.2.4 ADC interrupt

State variable acquisition is performed in the ADC ISR which also services the ADC interrupt. Motor control application fault detection is also handled in the ISR to minimize any potential fault reaction time. The ADC ISR then defers directly to the highest priority State Machine Task, releasing control to the FreeRTOS scheduler.

To defer the ISR directly in the State machine task, the task notification feature is used in the application. The vTaskNotifyGiveFromISR() API function sends a notification directly to a task, incrementing its notification value, and thus setting its notification state to pending, if it's not in the pending state already. See Example 4 for a task notification code example.

Once vTaskNotifyGiveFromISR() exits, it sets xHigherPriorityTaskWoken to pdTRUE, letting the subsequent portYIELD_FROM_ISR() call know to perform a context switch.

- Example 5: State machine task notification

```
BaseType_t xHigherPriorityTaskWoken;

/* Defer to State Machine Task: */
/* Initialize xHigherPriorityTaskWoken variable */
xHigherPriorityTaskWoken = pdFALSE;
/* Send notification to State Machine Task, update xHigherPriorityTaskWoken variable value */
vTaskNotifyGiveFromISR(PmsmStateHandle, &xHigherPriorityTaskWoken);
/* Perform context switch based on xHigherPriorityTaskWoken variable value, to ensure that the
interrupt returns directly to the highest priority task */
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
```

To successfully notify the State machine task, its handle must be stored in a TaskHandle_t variable type when creating the State machine task by the xTaskCreate() API function (see PmsmStateHandle variable in Example 3).

2.2.5 State machine task

After the State machine task is created by the Init Task and enters the Running state, it enters the Blocked state during the ulTaskNotifyTake() function call, waiting for the notification, since the task notification value is zero. Once the notification is sent to the State machine task by the ADC ISR, the task notification value is incremented, causing the State machine task to enter the Ready state. The ISR returns directly to the State machine task which is currently the highest priority task in the Ready state.

As the task notification value is now not equal to zero, ulTaskNotifyTake() zeroes the notification value and exits. The state machine function including the FreeMASTER recorder is then executed.

After the state machine and FreeMASTER recorder are processed in the task, the State machine task switches back to the Blocked state during the subsequent ulTaskNotifyTake() function call, waiting for the next notification to be received, since the notification value is zero again.

- Example 6: State Machine Task function

```
void pmsm_state(void* pvParameters)
{
     while(1)
     {
         /* Wait for notification from ADC ISR in blocked state indefinitely, zero task
         notification value on exit */
         ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
         /* State machine function */
         StateMachine();
     }
}
```

The first parameter of the ulTaskNotifyTake() function specifies the calling task notification value should be decremented (pdFALSE), by implementing a counting semaphore, or zeroed (pdTRUE), by implementing a binary semaphore [3].

The second parameter of the ulTaskNotifyTake() function specifies how long should the calling task wait for the notification to be received. It is set to wait indefinitely by the portMAX_DELAY parameter value in Example 6.

2.2.6 Main task

The lowest priority Main Task is being run in the background, continuously polling for MOSFET pre-driver faults via interrupt pin level detection. The external communication processing function is continuously called here to ensure the fastest response to the received data.

Where continuous external communication processing is not required, pre-driver faults can be polled not continuously, but periodically, leaving room for potential lower priority task(s). This can be achieved by blocking the Main task until a specific RTOS tick count, using e.g. the vTaskDelayUntil() FreeRTOS function, or by using a software timer (see [3] for more details).

3 Conclusion

To minimize the time spent in the ISR of the ADC or the PWM interrupt source, FreeRTOS offers deferred interrupt handling. Deferring from the ISR into an OS task using task notification, and therefore spending the minimum possible time in the ISR, reduces the latency of other interrupts that might be required in the application.

The utilization of FreeRTOS software features in the motor control application described in this application note are examples only. In most cases, more than one option on how to achieve the desired functionality is offered by FreeRTOS.

4 References

1. 3-phase Sensorless PMSM Motor Control Kit with S32K144, AN12235, Rev. 1, 05/2020
2. MCSPTE1AK144: S32K144 Development Kit for BLDC and PMSM motor control
3. Mastering the FreeRTOS™ Real Time Kernel, 161204, www.FreeRTOS.org

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: May, 2020

Document identifier: AN12881

arm