FreeRTOS synchronization methods part 2

Name

Universidad Panamericana

Presentation October 8, 2024

UNIVERSIDAD
PANAMERICANA

Contents

Event group bits

Introduction to Event Bits (Event Flags)

- Event bits, often called event flags, indicate whether an event has occurred.
- **Example Uses:**
  - A bit is set to 1 to indicate "A message is ready for processing"; 0 means no messages.
  - A bit set to 1 might also mean "A message is ready to be sent to a network"; 0 otherwise.
  - A bit could indicate "Time to send a heartbeat message" when set to 1; 0 otherwise.

TIP: *'Task Notifications' can provide a lightweight alternative to event groups in many situations.*

Event Groups

- An event group is a collection of event bits.
- **Example Configuration:**
  - "Message received" might be bit number 0.
  - "Message ready for network" could be bit number 1.
  - "Send heartbeat message" could be bit number 2.

Data Types and Storage

- **EventGroupHandle_t** - Variable type to reference event groups.
- **EventBits_t** - Stores all the event bits in a single unsigned variable.
- Number of bits depends on `configUSE_16_BIT_TICKS`:
  - 8 bits if set to 1.
  - 24 bits if set to 0.

RTOS API Functions

- API functions enable tasks to set, clear, or wait for event bits.
- Useful for task synchronization (*task rendezvous*).

Challenges in Implementing Event Groups

- **Avoiding Race Conditions:**
  - Built-in mechanisms to ensure atomic operations on event bits.
- **Avoiding Non-Determinism:**
  - Adherence to strict FreeRTOS quality standards regarding task and interrupt handling.

Normal API

General Event Group APIs

- **xEventGroupCreate**
  - Creates a new event group.
- **xEventGroupWaitBits**
  - Block to wait for one or more bits in the event group to be set.
- **xEventGroupSetBits**
  - Set one or more bits within an event group.
- **xEventGroupClearBits**
  - Clear one or more bits within an event group.
- **xEventGroupGetBits**
  - Returns the current value of the bits in an event group.
- **xEventGroupSync**
  - Synchronize a task with other tasks through event bits.
- **vEventGroupDelete**
  - Delete an event group and free its resources.

xEventGroupCreate

Purpose: Creates a new event group. **Parameters:** None. **Returns:** EventGroupHandle_t - a handle to the newly created event group. **Usage:**

```
EventGroupHandle_t eventGroup = xEventGroupCreate();
if (eventGroup == NULL) {
    // Handle error: Event group creation failed
}
```

Note: If the event group cannot be created, NULL is returned. Typically used at the initialization phase of the application.

xEventGroupSetBits

Purpose: Set one or more bits within an event group. **Parameters:**

- EventGroupHandle_t xEventGroup - The event group whose bits are being set.
- const EventBits_t uxBitsToSet - The bits to set.

Returns: EventBits_t - The value of the event group at the time each bit was set.
Usage:

```
EventBits_t setBits = xEventGroupSetBits(eventGroup, eBit0
    | eBit2);
```

Note: Useful for signaling to tasks that certain conditions or tasks have been completed.

xEventGroupGetBits

Purpose: Returns the current value of the event bits in an event group.
Parameters:

- `EventGroupHandle_t xEventGroup` - The event group from which to read the bits.

Returns: `EventBits_t` - The current value of all the bits in the event group.
Usage:

```
EventBits_t eventBits = xEventGroupGetBits(eventGroup);
```

Note: Useful for tasks to check the status of flags without changing them, supporting conditional behavior based on multiple flags' states.

xEventGroupWaitBits

Purpose: Wait for a combination of bits to be set within an event group.
Parameters:

- EventGroupHandle_t xEventGroup - The event group to test.
- const EventBits_t uxBitsToWaitFor - The bits within the event group to wait for.
- const BaseType_t xClearOnExit - Whether to clear the bits in the event group before exiting.
- const BaseType_t xWaitForAllBits - If pdTRUE, wait for all bits to be set; if pdFALSE, any bit.
- TickType_t xTicksToWait - Time in tick periods to wait for the event bits to be set.

Usage:

```
EventBits_t waitResult = xEventGroupWaitBits(
    eventGroup, eBit0 | eBit1, pdTRUE, pdFALSE,
        portMAX_DELAY);
```

Note: This function can block and is typically used within task code to synchronize actions.

xEventGroupClearBits

Purpose: Clear one or more bits within an event group. **Parameters:**

- EventGroupHandle_t xEventGroup - The event group whose bits are to be cleared.
- const EventBits_t uxBitsToClear - The bits to clear.

Returns: EventBits_t - The value of the event group at the time the specified bits were cleared. **Usage:**

```
EventBits_t clearedBits = xEventGroupClearBits(eventGroup,
    eBit1);
```

Note: Typically used to reset conditions once they have been handled.

xEventGroupSync

Purpose: Synchronize multiple tasks using an event group, creating a rendezvous point.

Parameters:

- `EventGroupHandle_t xEventGroup` - The event group used for synchronization.
- `EventBits_t uxBitsToSet` - The bits each task sets upon reaching the synchronization point.
- `EventBits_t uxBitsToWaitFor` - The bits each task waits for, ensuring all tasks have reached this point.
- `TickType_t xTicksToWait` - The maximum time to wait for the synchronization.

Usage:

```
EventBits_t syncBits = xEventGroupSync(
    eventGroup, eBit0, eBit1 | eBit2, portMAX_DELAY);
```

Note: Critical for operations where tasks must operate in lockstep, such as multi-stage processing or when tasks depend on each other's results.

ISR API

ISR-Specific Event Group APIs

- **xEventGroupSetBitsFromISR**
  - Set one or more bits within an event group from an ISR.
- **xEventGroupClearBitsFromISR**
  - Clear one or more bits within an event group from an ISR.
- **xEventGroupGetBitsFromISR**
  - Get the current value of the event group bits from within an ISR.

Event group code example

Example - xEventGroup

Listing 1: setup

```
#include "FreeRTOS.h"
#include "event_groups.h"

// Define bit positions using enum
enum EventBits {
    eBit0 = (1 << 0), // Bit 0
    eBit1 = (1 << 1), // Bit 1
    eBit2 = (1 << 2)  // Bit 2
};

// Function to create and return a new Event Group
EventGroupHandle_t createEventGroup() {
    return xEventGroupCreate();
}
```

Continue

Listing 2: Recieve

```
// Function to wait for specific event bits
void waitForEvents(EventGroupHandle_t eventGroup) {
    const TickType_t xTicksToWait = 1000 /
        portTICK_PERIOD_MS;
    EventBits_t uxBits;
    const EventBits_t uxBitsToWaitFor =
        (EventBits_t)(eBit0 | eBit1);

    uxBits = xEventGroupWaitBits(
        eventGroup,         // The event group being
            tested.
        uxBitsToWaitFor,    // The bits to wait for.
        pdTRUE,             // Clear bits on exit.
        pdFALSE,            // Wait for any bit.
        xTicksToWait        // Timeout.
    );
}
```

Continue

Listing 3: Recieve

```
            if ((uxBits & (eBit0 | eBit1)) == (eBit0 | eBit1
                )){
                 printf("Both eBit0 and eBit1 were set.\n");
            }
            else {
            printf("Timeout reached before bits were set.\n"
                );
            }
        }
```

Continue

Listing 4: Sent

```
// Function to set specific event bits
void setEventBits(EventGroupHandle_t eventGroup) {
    // Set eBit0 and eBit1
    xEventGroupSetBits(eventGroup, eBit0 | eBit1);
}
```

Timers

What are Software Timers?

Definition: Software timers are mechanisms that allow tasks to be executed at set intervals. These timers are managed entirely in software by the FreeRTOS scheduler, which ensures that they execute in a time-controlled manner.

How They Work:

- Software timers run in the context of a dedicated FreeRTOS service task, often referred to as the "timer service task".
- When a timer expires, it can trigger a callback function, allowing for periodic or one-time tasks without the need for manual timing control.

Key Features:

- *Configurability:* Timers can be configured for one-shot or periodic execution.
- *Precision:* Although managed by software, precision is typically adequate for many embedded applications.
- *Resource Efficiency:* Uses the system's existing scheduling framework to manage timing, conserving hardware resources.

Note: While software timers are highly useful, they should not be used for time-critical operations due to their dependence on the task scheduler and potential delays in a busy system.

xTimerCreate

Purpose: Creates a new software timer. **Parameters:**

- `const char * const pcTimerName` - A descriptive name for the timer.
- `const TickType_t xTimerPeriodInTicks` - The timer's period in tick counts.
- `const UBaseType_t uxAutoReload` - pdTRUE for automatic reloading, pdFALSE for one-shot timer.
- `void * const pvTimerID` - Identifier for the timer.
- `TimerCallbackFunction_t pxCallbackFunction` - Function to call when the timer expires.

Usage: Typically used to create timers that trigger tasks at fixed intervals or as single shot delays.

xTimerIsTimerActive

Purpose: Check if a timer is currently active. **Parameters:**

- `TimerHandle_t xTimer` - The handle of the timer to check.

Usage: Useful in scenarios where task execution depends on the status of a timer.

vTimerSetReloadMode

Purpose: Set or reset the auto-reload mode of a timer. **Parameters:**

- `TimerHandle_t xTimer` - The handle of the timer.
- `UBaseType_t uxAutoReload` - pdTRUE to set the timer to auto-reload, pdFALSE for one-shot.

Usage: Adjust the reload behavior of a timer during runtime.

xTimerStart, xTimerStop, xTimerChangePeriod, xTimerDelete

Functions: Control and modify timer states.

- **xTimerStart:** Starts a timer.
- **xTimerStop:** Stops a timer.
- **xTimerChangePeriod:** Change the period of a timer.
- **xTimerDelete:** Delete a timer and free its resources.

Common Parameters:

- `TimerHandle_t xTimer` - Timer handle.
- `TickType_t xBlockTime` - Time in ticks to wait for the command to be successful.

Usage: These functions provide basic timer manipulations essential for dynamic timing adjustments in applications.

Timer code example

Example: Using FreeRTOS Timers

```
// Timer callback function
void vTimerCallback(TimerHandle_t xTimer)
{
    // Code to execute when the timer expires
}

// Creating and starting a timer
TimerHandle_t xMyTimer;
xMyTimer = xTimerCreate("Timer", 1000 / portTICK_PERIOD_MS
    , pdTRUE, (void *) 0, vTimerCallback);
if(xMyTimer != NULL)
{
    if(xTimerStart(xMyTimer, 0) != pdPASS)
    {
        // Handle error
    }
}
```

Note: This example demonstrates creating a periodic timer that runs a callback function every second.