Mbed OS

Event flags, thread flags and mail

Christian Auby

UiA

Summary of last week

Mbed OS

- Open source lightweight OS for embedded devices
- Helps organize code in larger projects

Thread

- A function that runs independent of other threads
- Each thread has a separate stack for local variables
- Similar to threads on traditional OSes
- The scheduler switches which thread is running continuously

Mutex

- A lock that is used to limit access to a shared resource
- The resource can be a variable, a struct or a hardware component (SPI, Serial …)
- Needed to prevent simultaneous read / write / access (bad)
- Threads lock the mutex to ensure exclusive access to the resource
- Threads unlock the mutex when they are finished using the resource

Semaphores

- A signaling mechanism that can syncrhonize threads.
  Meaning: One thread can wait for another
- One thread calls *.acquire()* which then waits for another thread to call *.release()*
- A semaphore can have 1 or more slots, which then can wake up 1 or more threads

Event flags - What are they?

Scenario

- Imagine a thread that needs to wait on a button interrupt
- How can the thread be told about the interrupt?
- The interrupt is handled in a separate function from the thread
- The thread does not know that the interrupt has happened
- A semaphore (from last week) can be used to solve this waiting problem.
  Semaphores can only say that something has happened, not what

Event flags

- Ideal for signaling *what has happened* to a thread
- The thread can wait or check for one or more events
- Both threads and interrupts can set / get events
- Each *EventFlags* object supports up to 31 flags
- A *flag* is one specific thing that has happened
- Example: you could use 3 flags for 3 different buttons, 1 for movement and 1 for light.
  This EventFlags object would then use 5 out of 31 available flags.

Event flags example

```
// Each flag is one bit. Use bit 0 for the button event
#define EVENT_FLAG_BUTTON_PRESSED (1 << 0)

// Each EventFlags supports 31 flags (1 bit per flag)
EventFlags event_flags;

static void button_interrupt_cb(void)
{
  // Set event flag EVENT_FLAG_BUTTON_PRESSED
  event_flags.set(EVENT_FLAG_BUTTON_PRESSED);
}

// Thread that handles events from event_flags
void main()
{
  // Setup interrupts etc. here

  while (true) {
    printf("Waiting for button flag event...\n");

    // This thread will be blocked until the flag is set elsewhere
    event_flags.wait_all(EVENT_FLAG_BUTTON_PRESSED);

    printf("Got button flag!\n");
  }
}
```


See Gitea for the complete example

A note on number systems and bit flags

About

- Up until now we have handled most numbers as decimal numbers, base 10.
  50, 46, -127 and so on.
- When programming, especially microcontrollers, we also need two other systems:
  - The hexadecimal system, base 16, more on this in the next lecture
  - The binary system, base 2, 010110, where each digit is 0 or 1
- Base 2, 10 and 16 work in the same way:
  - The number to the far right is the least significant number, “worth” base^1 * number
  - The second number from the right is “worth” base^2 * number, and so on

Base 2 and bit shifting

- In the previous example we used *event_flags.set(bits);* to signal one or more events
- So the question then is, how can I set multiple flags at the same time?
- The answer is base 2 and two different operators: left shift: **<<** and bitwise or: **|**
- Left shift moves the the *1* into place, while | combines all the bits to a number
- Example - Set event flag 1, 5 and 7 (usually written from high to low):
  event_flags.set((1 << 7) | (1 << 5) | (1 << 1));
  Which is the same as:
  event_flags.set(0b10100010);

Thread flags - What are they?

Thread flags

- Thread flags are a more specialized version of event flags
- Event flags can be used to globally signal a number of threads, thread flags are only sent to a single specific thread
- Every thread instance can receive thread flags without any additional allocation of a thread flags object
- Both threads and interrupts can set thread flags, but only the thread can get using methods in *ThisThread* class
- The thread can wait or check for one or more flags
- Each thread has 31 such flags

Thread flags example

```
// Each flag is one bit. Use bit 0 for the button 1 event and bit 1 for the button 2 event
#define FLAG_BUTTON_1 (1 << 0)
#define FLAG_BUTTON_2 (1 << 1)

// Thread that handles events from event_flags
void thread1()
{
    while (true) {
        printf("Waiting for button flag event...\n");

        // This thread will be blocked until the flag is set elsewhere
        uint32_t flags = ThisThread::flags_wait_any(FLAG_BUTTON_1 | FLAG_BUTTON_2);

        if (flags & FLAG_BUTTON_1) { // Use the & bitwise and operator to check if a bit is set
            printf("Got button 1 flag!\n");
        }

        if (flags & FLAG_BUTTON_2) { // Use the & bitwise and operator to check if a bit is set
            printf("Got button 2 flag!\n");
        }
    }
}

// Somewhere in main
thread1->flags_set(FLAG_BUTTON_1);
```


See Gitea for the complete example

Avoiding “&” confusion

About

- Remember earlier when we talked about how * can mean different things in C++?
  - int* pint create a pointer called pint to an int
  - *pint dereference operator: follow the pointer pint and access the int stored there
  - 2 * 5 multiplication operator (math)
- The same is true for the &:
  - int& rint create reference to int called rint
  - &num address of operator: return the address of the variable num
  - 6 & 2 bitwise and operator: return the bits that are set for both numbers.
    In binary this will be: 110 & 10 = 10
  - Bitwise and operator is often used to check if a specific bit is set (1), like this:
    - `if(number & (0 << 1)) // Check if the first`
    - `if(number & (3 << 1)) // Check if the third bit is set (counting from 0)`
    - `if(number & (7 << 1)) // Check if the seventh bit is set (counting from 0)`

Mail

About

- Mail can be used to send a data between threads and / or interrupts
- The mail data can be any type, e.g. integer or a struct you made
- The mail queue has a maximum size
- Trying to allocate mail when the mail queue is full can block or fail (your choice)
- Trying to receive from an empty mail queue can block or fail (your choice)

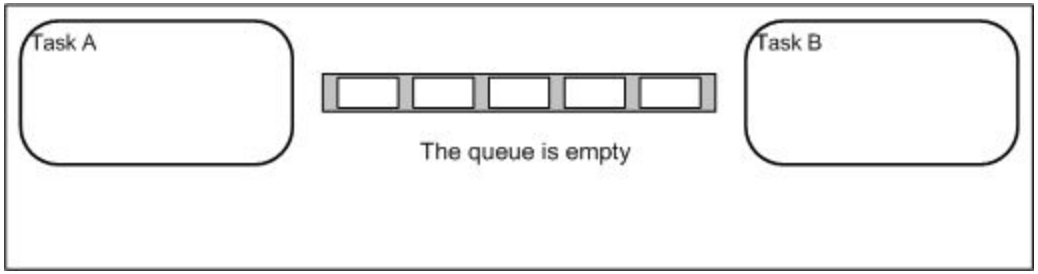

Figure from FreeRTOS.
Mbed OS works the same in principle

Mail example

```
typedef struct {
    float    voltage; /* AD result of measured voltage */
    float    current; /* AD result of measured current */
    uint32_t counter; /* A counter value               */
} mail_t;

Mail<mail_t, 16> mail_box;

void send_thread(void)
{
    uint32_t i = 0;
    while (true) {
        i++; // fake data update
        mail_t *mail = mail_box.alloc();
        mail->voltage = (i * 0.1) * 33;
        mail->current = (i * 0.1) * 11;
        mail->counter = i;
        mail_box.put(mail);
        ThisThread::sleep_for(1000ms);
    }
}

// Somewhere else:
mail_t *mail = mail_box.try_get_for(Kernel::wait_for_u32_forever);
```


See Gitea for an additional example

Questions?

Resources

- Mbed OS event flags
- Mbed OS thread flags
- Queue (general)
- Mbed OS mail