Department of Engineering

EE 4710 Lab 8

Title: Mailboxes in Real-Time Systems

Objective: The student should understand how mailboxes can be used to provide task synchronization and how mailboxes can be implemented.

Parts: 1-C8051FX20-TB Evaluation Board
1-USB Debug Adapter
1-DB-9 Serial cable (USB adapter cable is also ok)

Software: Silicon Laboratories IDE version 3.50.00 or greater. Keil compiler.

Preparation: Write the title and a short description of this lab in your lab book. Make sure the page is numbered and make an entry in the table of contents for this lab.

A version of the real-time operating system code we've developed over the last few labs has been "refactored" and stored in a .zip file on the course website (rtos.zip). Unfortunately, sadistic hackers have broken in and removed all the comments from rtos.c, the file that implements all of our operating system functions. To make matters worse, they have gutted all of the mailbox functions. This means you will have to re-write them.

Download rtos.zip from the course website. Study rtos.h and use the comments there to help you figure out what each function does so you can put comments back into the code. There are 3 mailbox functions that no longer have any code. Using your notes from class as a guide, reconstitute these three functions. (They should only be about 3-4 lines of code each not counting comments.) Do a good job. Your comments and mailbox code are worth half the points in this lab exercise (assuming it works).

As you go over the code to write the comments, you will see some differences between this code and your code in previous labs. First, create_task() now takes a fourth parameter called stack_size. This is because under certain circumstances the C compiler puts variables and function parameters on a "software" stack. This is a push-down stack, so create_task locates the software stack at the end of the stack memory supplied to create_task (which is why it is necessary to have the stack size).

On a related subject, you will note that the mailbox functions all have the additional keyword “reentrant”. Assuming the software stack is set up correctly, this keyword helps make the function “multi-thread safe.” In other words, two tasks can be executing the same function (at least one of which is suspended) without interfering with each other. For the Keil compiler, any function that has local variables or calls another function will not be thread safe by default. Setting up the software stack and adding the “reentrant” keyword to functions that may be shared solves this problem.

Once rtos.c has been reconstituted and compiled, it will be necessary to test it. Do this by creating three tasks. The first is released by the A/D complete interrupt. It reads the A/D, averages 60 samples and sends the result (once every 60 samples) to a mailbox.

The second task self-suspends 2.5 seconds then sends a negative 1 (-1) to the same mailbox.

The third task waits for a message in the mailbox and interprets it. If it is -1, the task prints “Scotty! More Power!” to the serial port. If it is -2 the task prints “Core Eject Failed”. For positive numbers it prints “Core Temperature” followed by the number unless the number exceeds 999, in which case it prints “Core Breach Imminent!”.

The aperiodic (background) task continually monitors the RI0 flag and, if the RI0 flag is set, it attempts to write -2 to the mailbox. If it succeeds, the RI0 flag is cleared.

Getting the A/D converter to interrupt is a little tricky. First, you must configure the A/D and voltage reference:

```
AMX0SL = 1;       // A/D channel 1, or whichever channel you choose
REF0CN = 7;       // turn on voltage reference
ADC0CN = 0x8c; // start on timer 2 overflow
EIE2 |= 2;        // enable A/D Complete interrupt
```

Next you must write an interrupt service routine that clears the A/D done interrupt flag and releases the first task (above). This code is quite simple as shown below.

```
void ad_interrupt(void) interrupt 15
{
    ADC0CN &= ~0x20; // clear interrupt flag
    sem_give(&sem_release_task1);
}
```

There are, however, some ramifications to this interrupt service routine. The C compiler tries to be clever about how many registers it saves during an interrupt, but when it sees a function call (like sem_give), it saves them all. Since the interrupt can occur in any task, you need to allocate enough stack space for each task to meet the demands of this interrupt. A stack size of at least 32 bytes is recommended.

After writing this code and compiling it without error, bring it to your scheduled lab session.

Procedure: Connect the serial port of your evaluation board to a computer running a terminal emulator such as puTTY. Connect the A/D input you have chosen to a potentiometer or adjustable power supply (be careful not to exceed 2.4V). Verify that "Scotty! More Power" is output once every 2.5s. Verify that a core temperature or "Core Breach Imminent" message is printed every 600ms. Adjust the input to the A/D to verify that the displayed temperature changes. Press a key and verify that "Core Eject Failed" is output. Demonstrate your code to your lab instructor.

Affix all your source code to your lab book then write a summary or conclusion. Remember to sign or initial then date each page.