Task management, delays and IPC in uC/OS-II

1 Introduction

This lab is concerned with the use of a simple OS in the development of event-triggered systems. The OS provideds a fixed priority pre-emptive scheduler and some mechanisms for synchronisation and communication. The work here complements the work that you've already done on time-triggered systems, providing you with an opportunity to practice a different, commonly-used approach to developing embedded systems.

2 In the lab

1. Clone the repository `cm0605_lab03.git` and checkout the solution to the week 3 lab.

   ```
   $ git clone https://github.com/DavidKendall/cm0605_lab03.git
   $ cd cm0605_lab03
   $ git checkout P05
   ```

2. Start up EWARM and load the workspace `cm0605_lab03/workspace.eww`.

3. Connect a `LPC-2378-STK` board to a USB port on your computer.

4. Start up EWARM (Version `>=5.50` required) and load the workspace `workspace/workspace.eww`.

5. Download and debug the project. Run it and observe its behaviour. This is a solution to the lab exercise given in week 3. Make sure that you understand the code in `scheduler.c` and how the scheduler functions are used in `main.c`. Ask your tutor if there's anything you're not clear about.

6. Now clone the repository `cm0605_lab05`. This provides a framework for a solution to the same problem as `lab03` but requires you to build an implementation using uC/OS-II. At the moment, the code simply flashes the LEDs. You should add a third task to manage the buttons so that you can control the flashing using button 1 and button 2: button 1 controls

flashing of the LINK led; button 2 controls flashing of the CONNECT led. Use the existing code to remind yourself what is needed to add a task to the system. Remember you need declarations for:

- the task priority
- the stack size and the stack data
- the function prototype for the task
- the function that implements the task

Don't forget to create the task using `OSTaskCreate()` and remember that your task needs to suspend itself at some point to allow other tasks to run.

7. Once you are happy that your basic buttons task is working, modify your program so that you can control the rate of flashing of the LEDs using the joystick: `UP`/`DOWN` to increase/decrease the rate for the link LED and `RIGHT`/`LEFT` to increase/decrease the rate for the connect LED.

8. Now modify your program so that each of the led tasks reports its status using the LCD. The task should report whether or not it is flashing and its current flashing delay, e.g.

```
(LINK) F:ON  D:3300
(CNCT) F:OFF D:4100
```

9. Do you observe any interference between the led tasks now? Remember the LCD is now a shared resource. Use a semaphore to provide mutually exclusive access to the LCD.

10. A better solution to the problem of accessing the LCD is to have only one task write to it. One possible approach would be to have the buttons task write to the LCD, not the led tasks. This would work for the moment since the buttons task has all the information needed for the display. However, this approach is not very maintainable. What if we introduce some other task(s) that also need to report information via the LCD? A better approach is to declare an lcd task and have tasks that need to use the LCD do so via this lcd task. For example, each task that needs to display some information could send a message to the lcd task. The job of the lcd task is to read requests for display and service them. Implement this solution by creating an lcd task that awaits messages arriving into a *message queue* and modifying the buttons task so that it sends messages to the message queue whenever the status of a LED changes.

Remember if there's anything you don't understand in the labs, please ask your tutor for help.