Labo n° 2
Real-Time systems [ELEC-H-410]
Sharing resources under µC/OS-II

2014–2015

Purpose

Throughout this second lab, you will learn how to manage the dependencies between tasks and how to share resources.

Useful documents are stored on the network share:

`\\labo\ELEC-H-410\Useful Documents\`

- `dsPIC30F-33F Programmer's Reference.pdf`
- `dsPIC33 Data Sheet.pdf`
- `Introduction to MPLAB.pdf`
- `Explorer 16 User Guide 51589a.pdf`
- `MPLAB C30 C Compiler User's guide.pdf`
- `uCOSII_RefMan.pdf`
- `Enhanced Controller Area Network.pdf`
- `Introduction to C programming`
- `Troubleshooting µC/OS-II`

1 Sharing resources

The simplest way to make two tasks communicate is to use structures of shared data: most of the time they are global variables. It is then necessary to establish a mechanism of protection to control the access to these variables.

Open the project `resource_sharing`.

Exercise 1.

- Analyse the code, knowing that each task is executed only **once** (`OSTaskSuspend()` is called at the end of the *for* loop), what should be the value of the global variable `cntr` at the end of the execution?
- Compile and execute the code.
- Using a watch window and the debugger, check the value of `cntr`. Is this the expected result? Why? (Use the logic analyser to help)

To protect the shared variables, uC/OS-II proposes two main solutions: **masking the interruptions** or using a **mutex**.

The first solution is very efficient, but can only be used for very short lapses of time (shorter than the critical sections created by the RTOS itself), otherwise the latency time of the interruptions is increased and preemption of the task is blocked, which is against real-time.

The **mutex** (see chapter over priority-driven systems) is a convenient way to protect shared data, while still handling interrupts and authorize the preemption of the running task. Note: when a task acquires a resource via `OSMutexPend()`, it should not forget to release it via `OSMutexPost()`. A pseudo code is given in Listing 1.

Listing 1: Mutex declaration/use

```
OS_EVENT *ResourceMutex; //declaration as global variable
ResourceMutex=OSMutexCreate(20,&err);//creation
void function()
{
        INT8U err ;
        ...
        OSMutexPend(ResourceMutex, 0, &err);
                //operations on the shared resource
                ...
        OSMutexPost(ResourceMutex);
        ...
}
```

The first 2 lines of Listing 1 create a mutex by:

- declaring a pointer with the type `OS_EVENT`: it will point to your mutex. This declaration must be global, so that the mutex is visible from anywhere in the program.
- creating the mutex. The first parameter indicates the maximum priority that the task owning the mutex can get.

For more details, see µC/OS-II user's manual pp 58-67.

Exercise 2. Modify the code of the `Ressource_Sharing` project by using a mutex to obtain the expected result for `cntr`.

2 Deadlock problem

Improper use of the mutex (or of any other mechanism of synchronization between tasks) can bring to a situation where no task can execute anymore, this is called a *deadlock* situation.

Open the `Deadlock` project and launch its execution.

Observe using the logic analyzer that no task is executed although the tasks should be periodic.

Exercise 3. Find the reason of this deadlock by drawing a time graph of the different calls to the mutexes

Exercise 4. Fix this problem using the parameter `timeout` of `OSMutexPend()`.

3 Priority inheritance

µC/OS-II uses a particular version of the priority inheritance. During the creation of the mutex, it is possible to define a priority which a task T_1 will inherit if T_1 blocks a higher priority task T_n. To work correctly, it is absolutely necessary to raise the priority above the priority of all the tasks using the mutex. Moreover, this priority shouldn't be used by any other task of the system. We insist however on the fact that this implementation of priority inheritance should perhaps be called "priority raising" to avoid the confusion with the normal priority inheritance protocol (see lecture notes chapter 4).

Open the project `Prio_inheritance`.

Exercise 5. Observe the mechanism with the logic analyzer.

4 Use of a semaphore

The semaphores (in a flag version) allow a synchronization between tasks or between an interrupt service routine (ISR) and a task.

In the Listing 2, `Task1` immediately goes to the "waiting" state until another `Task2` has reached a certain point of its execution, then `Task1` resumes its execution.

Listing 2: Semaphore example

```
OS_EVENT *startTask1;

Task1(){
        while(1){
                INT8U err;
                OSSemPend(startTask1, 0, &err);//Task 1 goes "waiting"
                //start of Task1
                ...
        }
}
Task2(){
        while(1){
                //begin of Task2...
                OSSemPost(startTask1); //Task1 goes to "ready"
                // other instructions of Task2...
        }
}
```

It should be noted that several "flags" can be posted in the same semaphore (counting semaphore). Thus a task can require that an event occurs several times before resuming its execution. In the same way, several tasks can synchronize on the same event if this event posts several flags in the same semaphore (each task gets one of the flags).

For more information, see µC/OS-II user's manual pp 92-97.

Open the project entitled `SpeexRTOS`.

This project contains two tasks:

- `AppTaskSpeex`, which decodes a sound by batches of 64 samples (long execution time).
- `AppTaskDac` which reads the decoded samples and sends them at 1kHz to a DAC connected to a loudspeaker.

A first semaphore was placed in this code so that the first samples are not send to the DAC (by `AppTaskDAC`) before they actually have been decoded by `AppTaskSpeex`.

Exercise 6. Use an additional semaphore so that a new batch of decoded samples is not written in one of the two buffers (`buf0[]` or `buf1[]`) by `AppTaskSpeex` before the previous batch has been completely send to the loudspeaker by `AppTaskDac`.

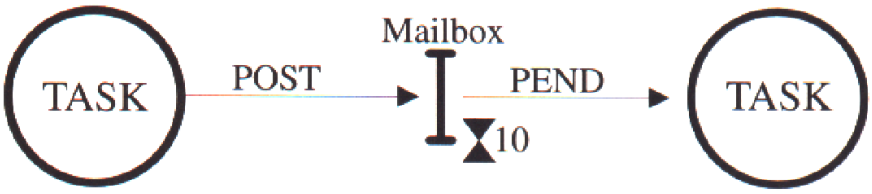
5 Mailboxes

Until now, we saw that the use of global variables made it possible to exchange information between two tasks. This method however presents some disadvantages among which the fact that a task is not automatically informed of a change of the variable. It is thus necessary that the task checks it periodically (polling) or to announce any change via a mutex or semaphore.

To make this exchange easier, µC/OS-II implements another protocol: **mailboxes**.

A task or an ISR can deposit a message in a mailbox. In a similar way, one or more tasks can receive a message in this mailbox.

Note: when we speaks about "message" under µC/OS-II, it actually acts of a pointer to a structure. The type of structure must obviously be known as well by the transmitting task as by the receiving task(s).



If a task wishes to receive a message coming from a void mailbox, it is suspended until the arrival of the message, or during a lapse of time defined by the application. Each mailbox is thus associated to a waiting list containing these suspended tasks. When the message is posted, it's the highest priority waiting task that receives the message.

All information about the functions of µC/OS-II allowing the management of the mailboxes can be found in the user's manual pp 32-43.

Open the project `Mailbox`.

This project contains two tasks :

- `Task_KeyScan` is the task which manages the keyboard: if a new key is pressed, the `KeyBuffer` variable contains the ASCII code of this key. The task is reactivated every 25ms to detect the actions of the user. `KeyboardScan()` is the function in charge of scanning the matrix keyboard, it is defined in the file `Keyboard.c`.
- `Task_Display` displays a character on the LCD screen. The function `DispStr()` is defined in the file `LCD.c` containing all the functions pre-written for the LCD. You will find a short explanation of these fonctions in the `LCD.h` file.

Exercise 7. Create a mailbox allowing to transmit the characters pressed on the keyboard to the task displaying the characters on the screen.