Labo n° 1
Real-Time systems [ELEC-H-410]
Realization of an application under μC/OS-II

2013–2014

Purpose

During the 5 following laboratories you will carry out a project having for goal to design a distributed alarm; this project will enable you to study:

- how to program under a real-time OS: μC/OS-II ;
- properties and uses of the network CAN;

The laboratories will be divided into two parts:

- the first three labs will have as goals to familiarize you with the programming under μC/OS-II and the use of the CAN network.
- the two following labs will be used for the realization of the distributed alarm, using the concepts previously acquired.

Useful documents are stored on the lab server:
`\ELEC-H-410\Useful Documents\uCOSII_RefMan.PDF` OK update this

1 First lab

During this first lab, you will learn how to write a task under μC/OS-II , to make it periodic and to assign it a priority, in an intelligent way. The hardware will be composed of a microcontroller board and a logic analyser.

If you are not confident with C programming, read XXX-XXX.

Principles of the logic analyser are explained in the chapter 9; an how to guide for the Asix Sigma2 logic analyser is in AppendixA.

1.1 Creation of a task under μC/OS-II

A task is a succession of instructions doing a specific operation. Contrary to a function, a task cannot return a value. Moreover you do not have any direct influence on the order of execution of the various tasks which you create. Indeed, it is the operating system which is given the responsibility to schedule the tasks and thus to choose which task must be carried out at which time on the processor. μC/OS-II is a premptive RTOS based on fixed priorities that you have assigned to the tasks. The choice of those priorities is thus crucial so that the system behaves as you wish. This is why the second part of this lab will be related to the judicious choice of the priorities.

First, you will learn how to create a single simple task in μC/OS-II and to initiate the execution of the operating system.

Using the Integrated Development Environment MPLab, open the project **"Example1"** in the folder `\ELEC-H-410\uCOS-II\Exercices.` OK update that

In the file main.c you will find the function main (see Listing 1) in which are executed :

- the initialization of μC/OS-II and all its internal variables : `OSInit()`
- the creation of the task `AppTaskStart`: `OSTaskCreateExt()`
- the starting of μC/OS-II : `OSStart()`

This structure cannot vary. The operating system must indeed be initialized before any creation of task and at least one task must have been created before giving control to OS. If no task were present in the system when the `OSStart()` function is called, μC/OS-II would launch a useless task "Idle" and do nothing else would be executed by the CPU.

For more details on the parameters sent during the creation of the task, refer to the μC/OS-II user's manual (page 113).

Listing 1: Function main.c

```
#include <includes.h>
CPU_INT16S main (void)
{
CPU_INT08U err;
OSInit();
 // Initialize "uC/OS-II "
OSTaskCreateExt(
AppTaskStart, // creates AppStartTask
        (void *)0,
        (OS_STK *)&AppTaskStartStk[0],
        APP_TASK_STARTPRIO,
        APP_TASK_START_PRIO,
        (OS_STK *)&AppTaskStartStk[APP_TASK_START_STK_SIZE-1],
        APP_TASK_START_STK_SIZE,
        (void *)0,
        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
OSStart();
 // Start multitasking (i.e. give control to uC/OS-II)
}
return (-1);
// Return an error - This line of code is unreachable
```

OK add caption and refr to listing

1.2 How to write a task

- the task must be written like a function which returns nothing (`void`)
- the task must contain an infinite loop : use one of the 2 structures `for( ; ; )...` or `while(1)...`
- a task must always call at least one of the services of µC/OS-II that will make the task "waiting" like `OSTimeDly`, `OSTaskSuspend()`, `OSSemPend()`, `OSMailboxPend()`, `OSMutexPend()`. Since µC/OS-II is preemptive the currently running task has got the highest priority among all "ready" tasks, hence if no event occurs (like an ISR making a higher priority task ready or the current task giving the control back to the scheduler) no other tasks will ever run.

Listing 2: task1.c

```
void task1 (void *data){
...
}
for(;;){
...
OSTimeDly(10);
}
//ask the RTOS to put task1 in "waiting"
//state for at least 9 ticks
```

1.3 Put a task to sleep for some time

Sometimes, it is necessary to let a task sleep for a while (maybe the job is complete...) reformuler

Let a task sleep in "waiting" state for some time `OSTimeDly(INT16U tick_nbr)`; The parameter `tick_nbr_tick` is an unsigned 16bit integer (ranging from 0 and 65535) which determines the number of ticks during which the task will sleep. The timer creating the periodic interrupts has been configured for a frequency of 1kHz, hence 1 tick = 1 ms. More precisely the task will sleep at least (`tick_nmbr-1`), if you want to be sure to sleep during 1 tick you should specifiy `tick_nmbr=2`. To demonstrate that, draw a chronogram of tick interrupts and imagine where the call `OSTimeDly()` could occur.

Exercise 1. Create one second task in the Example1 project which lights a LED of the uC board at a frequency of 1Hz.

To change the state of the LED, toggle pin `LATAbits.LATA3`. Remember that you have to configure the pin in the output direction by the instruction `TRISAbits.TRISA3 = 0`. (see

`\ELEC-H-410\Useful Documents\La Carte Explorer 16.pdf p3 or \ELEC-H-410\Useful Documents\dsPIC3`

Use the code for `AppTask1` as a model.

1.4 Creation of periodic tasks

In Exercise1, you have created a periodic task, i.e. a task executing forever at regular intervals. In most industrial applications, those tasks are frequent and a the periodicity should be realized with a good precision (see example of PI controller in chapter 3 of the course).

Exercise 2. Open the project entitled (Example_Periodicity). You will find 4 tasks in this example:
- `AppTaskStart` whose only goal is to create the three other tasks
- `AppTask1` who should have a period of 10ms;
- `AppTask2` who should have a period of 50ms;
- `AppTask3` who should have a period of 100ms.
- Switch the logical analyser on and launch the display interface on the PC.
- Open the configuration file elec-h-410.lwc in the folder /MesDocuments/ELEC-H-410. change that
- Start your program Example_Periodicity on the microcontroller and launch a first data acquisition. with the logic analyser.
- Observe the evolution of the value of the bus `RunningTaskId` which shows the identifier of the tasks running on the processor. Observe preemptions of certain tasks when a higher priority task is active (see signals `Task1Active`, `Task2Active` et `Task3Active` whose value is 1 when the tasks `AppTask1`, `AppTask2` and `AppTask3` are respectively active, *i.e.*, between its first instruction until its completion).
- Use the logic analyser to measure the real period of real activation of each task. Are they exactly in conformity with the desired periods? Identify 2 causes of these errors.

1.4.1 Use of `OSTimeGet()`

μC/OS-II provides the `OSTimeGet()` function which returns a 32 bit integer (`INT32U`) representing the number of ticks since the launching of OS.

Exercise 3. Compute after how long this counter will overflow.

Exercise 4. Use `OSTimeGet()` in each task to compensate for the error over the period.

1.4.2 Use of a software timer

It is possible to use software timers in μC/OS-II . Those are used exactly in the same way as hardware timers, except that they are entirely managed by the operating system and that they are synchronized on the ticks of the system. The function `OSTimerCreate()` allows to create a software timer (see manual for the parameters details) and `OSTmrStart()` to start it. When a timer expires, it calls a function whose pointer was given to it in the parameters.

Open the project (Example_Timer). You will find the same 4 tasks as in the previous example except that their period are generated by using three timers softwares.

Functions `OSTaskSuspend()` and `OSTaskResume()` allow to suspend and restart the execution of a specific task.

Exercise 5. Check with the logical analyser that the periods are strictly respected.

This method for creating periodic task gives very precise results. However, it is rather heavy and should therefore be used when this precision is absolutely required.

Exercise 6. Create a new a timer which switches a LED on after 5s.

1.4.3 Choice of the priorities

As explained earlier, the choice of the priorities of the task is the only tool at our disposal to help the operating system to choose which task must be running at which time. To be convinced of the importance of a judicious choice of these priorities, we will look at a simple example.

Exercise 7. Open the project (Example_Priorites).
- The task `AppTask1` should run every 1ms
- The task `AppTask2` should run every 100ms
- Check the behaviour of the tasks with the logic analyser.
- Reverse the priorities of `AppTask1` and `AppTask2` and reverify what occurs.

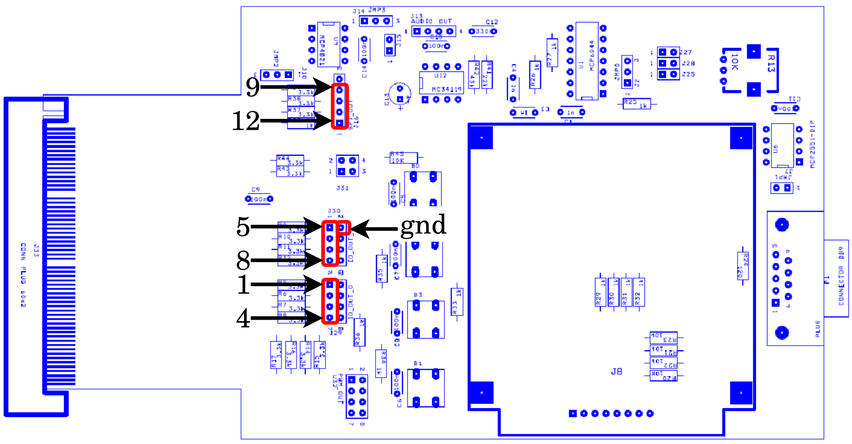
– By comparing the periods of each task and the priorities assigned, which systematic rule of assignment can you deduce?
– How is called this method to assign the priorities?
– What happens when tasks have relative deadlines different from their periods?
– Which scheduling algorithm would you use if you could assign priorities directly to jobs instead of tasks?

A The *Asix Sigma2* logic analyser

The Asix Sigma2 logic analyser is like:

A.1 Electrical connections to the Explorer 16 board

Connect the analyser to the extension board with the numbered ribbon cable following this scheme:



A.2 Software on the computer

Add some screenshots here

A.3 Basic measurements

There is a cursor showing the time and values of signals in the main window. To place a marker, press space. If you move your cursor, the difference between the marker and the cursor will show in a tooltip.