

## Realization of an application under $\mu$ C/OSII (1)

### 1 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:  $\mu$ C/OSII;
- 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  $\mu$ COSII 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 document: see on the lab server \MesDocuments\ELEC-H-410\Useful Documents\uCOS-II-RefMan.pdf

### 2 First lab

During this first lab, you will learn how to write a task under  $\mu$ C/OSII, 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.

Principles of the logic analyser are in the chapter 9; a summary of the user manual of the Agilent Logicwave is in appendix A

### Creation of a task under $\mu$ C/OSII

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.  $\mu$ C/OSII is a preemptive 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, we will learn how to create a single simple task in  $\mu$ C/OSII 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\day\Exercices.

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

- the initialization of  $\mu$ C/OSII and all its internal variables : OSInit()
- the creation of the task AppTaskStart: OSTaskCreateExt()
- the starting of  $\mu$ COSII: 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,  $\mu$ COSII would launch a useless task "Idle" and no nothing else would be executed by the CPU.

For more details on the parameters sent during the creation of the task, refer to the  $\mu$ C/OSII user's manual (page 113).

```
#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_START_PRIO,
        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
}
```

Figure 1

### 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  $\mu$ C/OS-II that will make the task "waiting" like OSTimeDly, OSTaskSuspend(), OSSemPend(), OSMailboxPend(), OSMutexPend(). Since  $\mu$ C/OSII 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.

### Example :

```
void task1 (void *data){

    ...

    for( ; ; ){

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

}
```

Figure 2

### 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\_nمبر-1***), if you want to be sure to sleep during 1 tick you should specify ***tick\_nمبر=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  $\mu$ C 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.

Take the code for the AppTask1 as a model.

## 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 the periodicity should be realized with a good precision (see example of PI controller in chapter 3 of the course).

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.
  - Start your program **Example\_Periodicity** on the microcontroller and launch a first data acquisition with the logic analyzer
  - 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 analyzer to observe the real period of real activation of each task. Are they **exactly** in conformity with the desired periods ? **Identify 2 causes of these errors**

## Use of OSTimeGet()

$\mu$ C/OS-II provides the OSTimeGet() function which returns a 32 bit integer (INT32U) representing the number of ticks since the launching of OS. **Compute after how much time this counter will overflow.**

### Exercise 2:

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

## Use of a software timer

It is possible to use software timers in  $\mu$ C/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.

- Check with the logical analyzer 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 3:**

Create a new a timer which swithes a LED on after 5s.

## **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.

Open the project **Example\_Priorites**.

- The task AppTask1 should run every 1ms
- The task AppTask2 should run every 100ms
- Check the behavior of the tasks with the logic analyzer.
- Reverse the priorities of App Task1 and App Task2 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?

## Appendix A : User's manual of the Agilent Logicwave

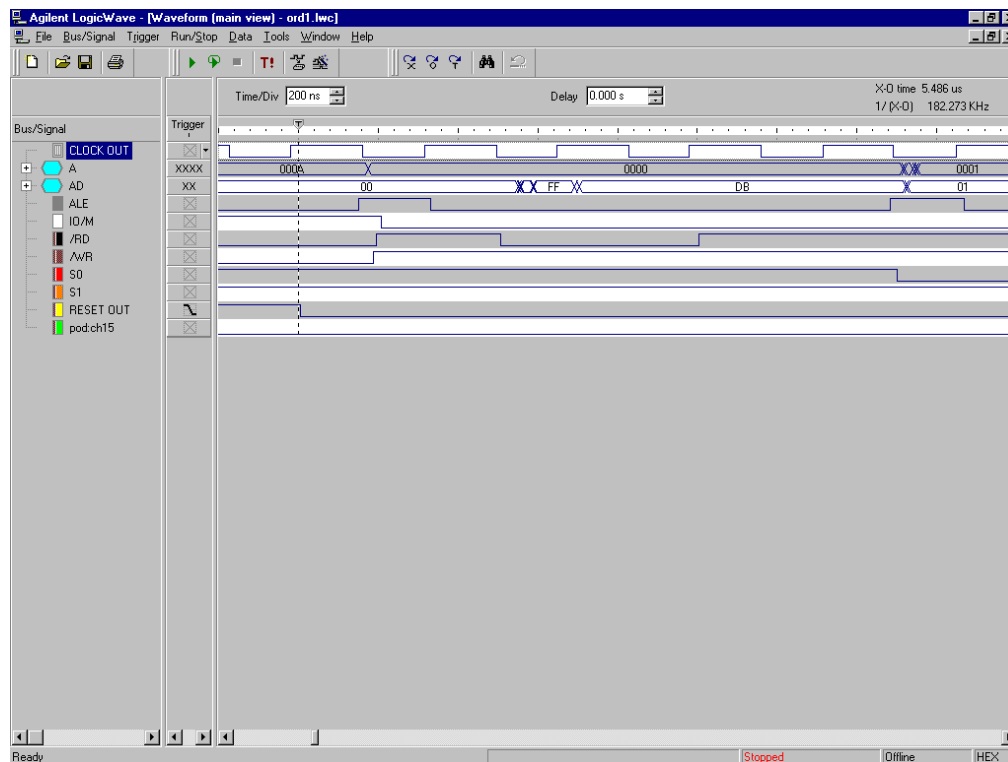
### Principles: see lecture notes

The logic analyzer is connected to a PC. An interface program allows to control the analyzer and display the signals. The parameters and the results of acquisitions can be saved in .lwc (LogicWave Configuration) files.

### Starting

After starting, the program initializes the logical analyzer, then displays a first page; this one enables the user to create a new configuration, to load the last configuration used or an other existing configuration.

### Main screen



The main screen is divided into 3 zones (from left to right):

- the list of measured signals: to improve the legibility, we can group them in busses and rename them
- trigger conditions: for each signal, you can define a condition *don't care*, *=1*, *=0*, *rising edge*, *falling edge* or *both*. The trigger condition is true when the conditions on all the signals are checked simultaneously. You can also define two conditions which must be checked sequentially with a condition on the time separating them;
- the chronogram displaying the signals. You can use 3 time-related markers :
  - the trigger marker (T) indicate the moment when the trigger condition occurred
  - markers 0 and X can be moved allow to measure the lapse of time between two events.