

**Exercises and assignments for the course DIT632 Development of Embedded systems \_ Part 3**

**Work package nr 6 : Processes, Threads and an event driven door automat. (13 p in total)**

**Exercise 6\_1 Test of threads (exerc 6\_1.c) (1p)**

On the homepage in Canvas you can find a file *sortandfind\_18.c* . The code performs the following operations:

1. It creates an array of integer numbers with the given number of elements.
2. Fills such array with random integer numbers whose max value is given by the user [STEP 1]. There is a dummy delay in this random generator part just to view the problem in question 6.
3. Sorts the array [STEP 2].
4. Looks for a specific value given by the user using binary search [STEP 3].

**Answer the following questions:**

1. How many threads (in total) are created during the execution of the program?
2. What are lines 16, 17, 20, 22 and 23 intended for?
3. How many parameters are passed to function runner?
4. How can function runner know the values for parameters that, such as max value, not are passed to it?
5. Test how the behavior of the program change if line 26 is omitted. Explain why.

**To do :** Study the program and write your answer in a simple text document *exerc6\_1.txt* and view it for the TA and they will give you a pass code if ok. Hand it in together with the other files.

**Exercise 6\_2 More about threads (exerc 6\_2.c) (2p)**

Rewrite the application in *sortandfind\_18.c* so that each of the three main steps (filling the array / sorting the array and finally finding the value given by the user) are executed by 3 different threads and work as expected. Clarify the function of each thread with a `printf( )` stating which of the 3 steps the thread is running. Do it work as intended?

**The following four exercises 6\_3 to 6\_7 are for those of you who had access to XCC12 environment.**

**(Note : !!** If not access to XCC12 there is alternative exercises 6\_3alt to 6\_5alt further in this paper. Do also note if you don't do the exercises intended for XCC12 you are recommended to read and try to understand exercise 6\_3 to 6\_6.)

### Regarding this exercises 6\_3 to 6\_6

In the following exercise 6\_3 – 6\_6 you can write control programs for the MCC12 system with a connected door unit ML13. The unit has the **base address of 0xB00**. The idea is to design programs, taking care of opening and closing of a door simulated by unit ML13 in XCC12. You will during the exercises develop three different programs with different principles regarding the way of reading events such as sensor (or push button), door opened, and door closed event and finally a time out event for the end of a delay function. The programs should be developed in the XCC12 and only be tested and verified in its simulator.

#### Exercise nr 6\_3. Time based control program (exerc\_6\_3.c) (2p)

First you are going to develop a time-based implementation of the automatic door. The door will operate in the following manner:

- ✓ If someone approaches the door space (one of two button pushed) the door will open.
- ✓ After the door had been open (in several seconds, controlled by a simple loop) the door should close automatically.
- ✓ If someone approaches the door space while the door is closing, it shall immediately reopen.
- ✓ The main structure of the program is described roughly by the following flowchart:
- ✓

The implementation will be done entirely in the programming language C and in one single module (file).

#### Control of the door unit ML13.

The door is controlled via four different 8 bits registers which is possible to read from or write to depending on the register. The addresses for the registers is the unit's *base address* (Address 0xB00) or *base address +1*.

You will find details about the register and what each bit in the different registers are controlling for the door in : XCC12 Help>>Help on XCC12 and IO simulator.

#### In general:

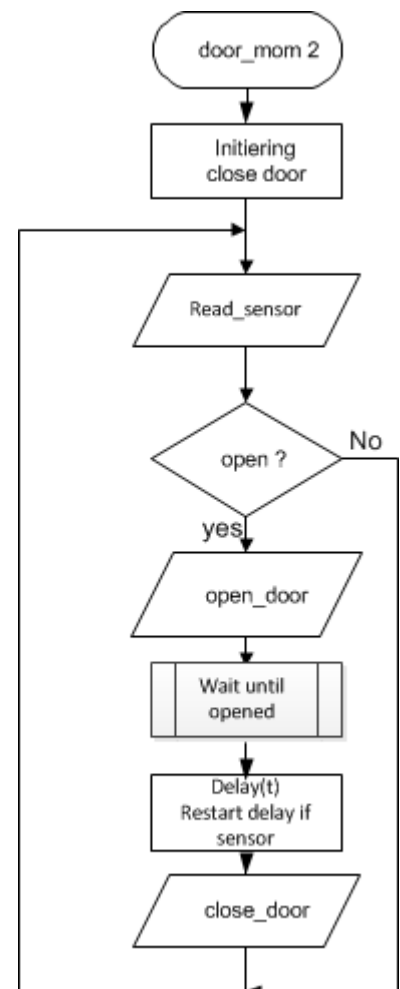
**Control register:** Address 0xB00 (write operation), same as base address for the ML13 unit.

By setting **bit 0** you start open the door, it will open slowly and you can follow the opening process by watching the LED:s going on. Setting **bit 1** starts closing the door. Closing and opening of the door could be done anytime even if the door is in its opening or closing phase.

Bit 7-2, Not used.

Bit 1, CLOSE 1 = Actuate "Close Door"

Bit 0, OPEN 1 = Actuate "Open Door"



**Status register** : Address 0xB00 ( read operation). Each bit describes a certain state of the door. For example door is closed, door is opened ...You can by this register check the actual state of the door.

Bit 7,	CLOSING.	1 = The door is closing.
Bit 6,	OPENING.	1 = Door is opening
Bit 5,	S2: Sense 2.	1 = Door is closed
Bit 4,	Not of interest	
Bit 3,	S1: Sense 1.	1 = Door is wide open.
Bit 2,	Not of interest	
Bit 1,	RIGHT: Sensor B.	1 = Sensor B is activated (Button S2 is pushed)
Bit 0,	LEFT: Sensor A.	1 = Sensor A is activated (Button S1 is pushed)

**To do:**

Your task is to write an ordinary control program in C with a structure as the flowchart above views and verify its function in the simulator. For verifying the program, you must connect the door unit ML13 to the simulator. For controlling the door opened time in this first exercise you can use a simple for loop.

---

**Exercise 6\_4 Event (IRQ) driven control program (exerc\_6\_4)**

**(3p)**

In this task you shall develop an event driven implementation of the automatic door. The door will operate in the same manner as before but every event at the door ( push of button, door opened, and so on ...) will be indicated by a interrupt request signal from the ML13 unit. Instead of repeatedly read the control register to determinate if any event had occurred the system can run a interrupt service routine at any event in the door unit and check what type of event it is. The IRQ signal is in simulator connected to the IRQ in pin on the CPU card **if you on the door unit enable IRQ in menu "Interrupt" on the ML13 unit.**

Every event in the ML13 module will generate a hardware IRQ signal which is connected to the CPU IRQ in port. The IRQ will, if the system is initiated for allowing interrupts, call for the defined IRQ service routine. The IRQ service routines start address must be stored in the vector table at the position for the hardware interrupt IRQ (Address **0x3FF2**).

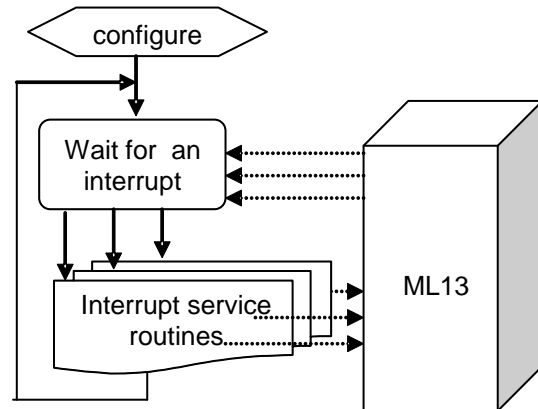
The IRQ service routine can then check the source for the IRQ-request (Sensor, door closed,...) in the IRQ status register (**base address+1**) and by use of a **global variable** send information of which event in ML13 unit that caused the interrupt to the main control function. (Note: An interrupt function can't return a value) due to the way it is called.

**Exercise 6\_4\_a** (filename exercise\_6\_4\_a.c)

**(2 p)**

In this first part you shall design a C-program that controls the door as in the previous exercise but instead of polling the register to check for events from the system you shall use a interrupt service routine that will be called only when an event has occurred.

The figure besides views the structure of the program in this case. You have a main part of the program running in normal mode. When an event happens in the door unit it will generate an interrupt request and the system stop executing the main part directly and start executing an interrupt service routine manage something in the system, for example starts open the door. After this interrupt routine ends and system continue running the main part.



Below you can see a very simple skeleton for a program as above described.

// A global variable affected by the interrupt service routine handler.

```
int          interruptType=0;
// ----- Main program -----
void         main(){
    IRQ_init(); // Set up the system for interrupt
    while(1) {
        interruptType = 0;
        while(interruptType==0); // Wait for IRQ. This should be the normal main part of the program.
        // Do something depending on the value for interruptType
        ....

        interruptType=0; // Clear interruptType to wait for next IRQ and back to main part.
    }
}
// -----
// Interrupt service function for hardware irq ( vector address 0x3FF2 )
// This routine will be called for any IRQ from the ML13 module (Sensor, door opened, door closed..)

__interrupt void ML13_interrupt( void ){
    // This function will be activated for every IRQ request from the Door unit.
    // Check IRQ source (IRQ status..) and set InterruptType to a value (not 0)
    //depending on IRQ source.

    REG8(ML13_IRQ_Control)=0x01; // Ack of the hardware irq.
    //Continue in normal mode at the instruction that was interrupted.
```

```
}

// Function for setting up the system ( IRQ ....)
void IRQ_init(void){
    // Save start address for the interrupt function in Vector table.
    // Allow CPU affect on IRQ signals ( Needs inline assembler for this)
    // Enable IRQ from ML13 unit

}
```

#### To do for 6\_4\_a:

In documents on the course homepage ( Document/Programs examples and skeletons/ Week #6..) you find a skeleton for a solution of this exercise, ***exercise\_6\_4\_a\_template.c***. Download it and continue develop the program so it will work as earlier exercise but this time with help of a IRQ service routine. You shall verify the function in XCC12 debugger. Do not forget to activate interrupt mode for the ML13 unit in the debugger.

In the solution earlier (6\_3) there was a time delay during which the door is fully open before closing. In your first solution for this part of the exercise **you can skip a time delay** during which the door is fully open before it will start closing.

When everything is ok by this insert a time delay still just a simple for(..) loop. Some solutions of this can cause a problem since a IRQ signal can be generated during this delay time and the IRQ service routine will set the **interruptType** flag to a new value and then restart executing of the delay function. For most solutions this will not cause any problem in this case but if there were a lot more possible IRQ-events it could generate problems. This problem will be solved in part b of this exercise.

#### Exercise 6\_4\_b (Use of a timer unit) (exercise\_6\_4\_b.c) (1 p)

As mentioned above there can be some problems by inserting time delay functions in solutions including IRQ routines. Even in solutions working with polling I/O units time delays are problematic since it will slow up continuous execution.

A way of solving this problem is to use hardware timer units. It is possible to start a hardware timer from the program and after that it will tick time independent of and in parallel with the program execution. After a certain time, the timer can generate a Timer interrupt to the system and the system can respond on this IRQ in the same way as any other IRQ. Usually there is a separate position in the vector table for an TimerIRQ.

To test this, we shall further develop the solution from 6\_4\_a and besides the interrupt from events in ML13 (IRQ) we will also use the microcontrollers Real Timer unit to make a delay function with support from this hardware.

For doing this you shall use an assembly – file (**ML13\_irq\_asm\_dit165\_18.s12**) with a number of functions that can be called from C-program module in the same way as other C-functions if they are declared as external functions in program head. The file could be downloaded from the course homepage.

You can configure the RTC module, through some registers, to generate a timer interrupt every x millisecond. The configuration is in our case done by the function **timerSetup()** in the assembly file **ML13\_irq\_asm\_dit165.s12**.

Before calling the assembly function from your C-program module (file) you must :

- Included the **.s12** file into the project in the XCC12 IDE.
- In your C-program module declare it in the program head as : **extern void timerSetup()**

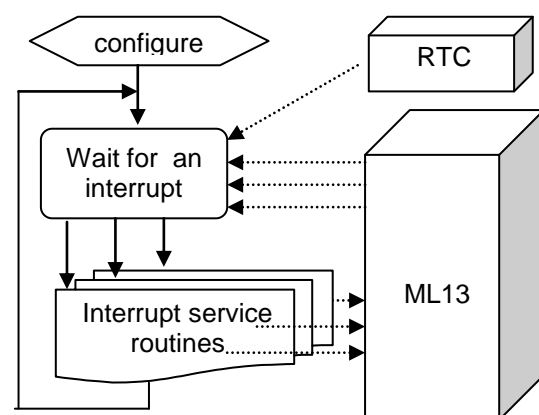
Then you can call the function from the C-program module as a normal call by : **timerSetup();**

By calling another function in the assembler module **setTimeout (x)**, where x is the number of second's delay we wish to create, you start a time delay that will generate a TIME\_OUT interrupt to the system after desired time defined by x. The IRQ routine (in assembler) will set the value of the global integer **interruptType** to TIME\_OUT indicating that the door should be closed.

[ **Just for information** if you look through the assembly code: After the call of **setTimeout (x)** the RTC will generate RTI interrupts with a periodicity depending on the configuration in **timerSetup()**. At every RTI interrupt the RTI routine (**ptimirq**) will decrease a counter but not affect the value of the global **int interruptType**. After a number of RTI:s the counter will be zero and the function set the value TIME\_OUT to the global variable (**interruptType**). By this the main program gets the information that the delay time is ended and in this case the door shall be closed.]

**Below is a roughly description of the main structure of the program.**

The program shall consist of at least two modules (files) one entirely written in C for the main control program including the ML13 interrupt service routines and one for all regarding the timer module written in assembly. The assembly module you can download from the course home page (**ML13\_irq\_asm\_dit165\_18.s12**). For the C-module you just copy your 6\_4\_a solution file and continue develop it for this case. Both files should be included in the project when compiled.



**You should do the following:**

Create a new project in the same workspace for this exercise. Include both files above. Continue to develop the C-program to get the system working as the first program but now added with a TimeDelay for the door open time controlled by the RTI timer module in CPU. In the simulator we

just use setTimeout(2) otherwise it will be too long delay. The timer will then generate a TIME\_OUT interruptType (assigning the value of TIME\_OUT to the variable) after the delayed time has fulfilled.

Note: Do not forget to activate interrupt mode in the ML13 unit in the simulator.

---

### Exercise 6\_5 Introduction to exercises with the Real-Time kernel RTK12

(Note : Exercise 6\_5 shall not be handed in. You do them just to be able to solve the exercises 6\_6 and 6\_7)

During this part of exercises, you will be introduced to a very simple real-time kernel, **RTK12**. The kernel could be included to programs developed in the XCC12 environment. The program using RTK12 kernel could also be simulated in XCC12 debugger as normal programs.

You can find more information about the RTK12 kernel in the XCC12 help system about libraries.

#### A process

A RTK12 process has the same appearance as a C function without parameters. However, there are some important differences that you must consider when programming with processes:

- ✓ A process is not a function in the sense that it can be called from another process (or function).
- ✓ The execution of a process can, at any time, be interrupted and later continue its executing.
- ✓ Several processes can share the same features, i.e. calling the same functions read and write to the same variables (global). Please note that the global variables that are shared by multiple processes (or used by functions that are shared by multiple processes) generally must be protected from inconsistent update with **mutual exclusion**.

**A typical program for executing several processes is build up as follows:**

```
//          RTK12 - application
//          Non preemptive scheduling
#include     <_rtk.h>
#define TIMESLICE      5           // 5 for simulator mode and 100 for hardware
void        AtInterrupt(void);     // Declaration of the function that is called at the end of
                                   // every TIMESLICE
PROCESS     P1(void);              // PROCESS is defined in rtk.h as void.
PROCESS     P2(void);              // This is a declaration of the actual processes.
                                   // P2 is the process name
```

```
int main(void){
```

```
/* This main function will be executed just once to set up the processes and the kernel.
```

```
It must contain the following:
```

- ✓ Initiate RTK12
- ✓ Create all processes in memory.
- ✓ Start the kernel RTK12 // for running in simulator or in hardware

```
*/
```

```
        InitKernel(TIMESLICE, AtInterrupt);
        if( CreateProcess("P1", P1) == -1){
            printf("\nCan't create process");
            exit();
        }
        if( CreateProcess("P2", P2) == -1){
            printf("\nCan't create process");
            exit();
        }
        StartKernelForSim();          // This ends the execution of main()
        return(0);
} // End main

// ----- First process -----
PROCESS    P1(void)
{
    DO_FOREVER          // DO_FOREVER defined as while(1)
                        // This is the process loop that normally never should end.
    {
        _outchar('1');  // A special version of putchar() function in startup.s12 module
    }
}

// ----- Next process -----

PROCESS    P2(void){
    DO_FOREVER
    {
        _outchar('2');
    }
}

//-----          Interrupt routine called at the end of each TIMESLICE .
void        AtInterrupt(void){
    // This function calls by a IRQ –routine in RTK after each full TIMESLICE.
    // Decides if the IRQ should suspend the running process and start next process
    // in QUE.
    // Doing nothing in this function will make the previous process to continue executing
}
```

### Exercise 6\_5\_a

First you should use RTK12 to perform three processes during a "non-preemptive" scheduling strategy. The application of this exercise, consists of a main program, two processes (P1 and P2) and an interrupt handler. The main program follows exactly the structure described above. Note in particular that an interrupt handler must be provided even if, as in this case, no process switch



should be done at interrupt. We will illustrate the **non-preemptive scheduling** and write therefore two processes to be performed sequentially. Note that in general one can't make any assumption about the order in which processes will be started by the real-time kernel. For RTK12 however the rule is that processes are started in the order they are created.

**Processes P1 and P2** are similar, and they look as follows:

```
PROCESS P1(void)
{
    int i;
    for(i=0;i<100;i++)
        _outchar('1');
    TerminateProcess(0);
}
```

```
PROCESS P2(void)
{
    int i;
    for(i=0;i<100;i++)
        _outchar('2');
    TerminateProcess(0);
}
```

The interrupt handler should in this case do nothing, but must be included:

```
void AtInterrupt(void)
{
}
}
```

**You shall do :**

Write a main program as describe earlier and the above processes. Compile it and test to run it in the simulator. For seeing anything you must connect the console to the simulator. Try to understand what happens by the program and why.

**Don't forget !!!** To be able to compile a program including RTK12 you must give a directive to the compiler. You do this in the XCC12 IDE in menu : **Project / Settings** : C-Compiler , in text box DEFINES should you add , RTK12 after \_DEBUG.

### Exercise 6\_5\_b Study of Preemptive Scheduling , Timesharing

Write a main program setting up three processes and write the processes. Each process should be designed to run forever as below:

```
PROCESS P1(void){  
  
    DO_FOREVER {  
  
        // program code;  
  
    }  
}
```

Each process should only print out the number of the process in the console using the special function:

```
_outchar('nr')
```

The kernel will then through its Real Time Clock generate interrupts periodical and count the system time. For each time the timer reach the end of the time period specified by TIMESLICE the kernel will change execution to the next process if we modify the AtInterrupt() function as shown below.

```
void      AtInterrupt(void){  
    // Call of two kernel functions.  
    insert_last(Running, &ReadyQ); // On going proc. To last in Que.  
    Running= remque(&ReadyQ);    // Pointer Runner points to first process in Ready  
}
```

**To do:** Write the program and test in the simulator and see what happens. You can also change the value of TIMESLICE and study what happens.

### Exercise 6\_6 : Use of semaphores ( Hand in as exerc\_6\_6.c)

(1p)

In the previous chapter we learned how to implement a parallel programming model in a time-sharing system. By considering each program as a process and let a real-time kernel manage these processes, we can, in principle build a multi-tasking system. If the system comprises of many independent processes it is relatively simple to implement but this is never the case in reality. In fact, there is usually very close dependencies between the different processes in a real-time system. Often, these dependencies consist of shared global data and special time dependencies. To cope with these dependencies the processes must in a way be synchronize with each other.

Process synchronization in RTK12 is done by using semaphores. In RTK12 the Semaphores are implemented as blocking / Queue type. The maximum number of semaphores is given by the constant MAX\_SEM\_ID defined in the header file rtk.h. The following operations can be performed on a semaphore:

***void initsem (int id, int count);***

The first parameter is an identification number between 1 and MAX\_SEM\_ID. The second parameter assigns an initial value to the semaphore. Each semaphore is initialized once by the main program, this occurs between the execution of InitKernel () and Start Kernel () or StartKernelForSim().

***void waitsem (int);***

The parameter is an identification number as above defined by initsem() . The process performs the wait semaphore specified by the parameter. The routine assumes that the semaphore is initiated by initsem ().

***void signalsem (int);***

The parameter is an identification number 1-MAX\_SEM\_ID. The process signals on the semaphore specified by the parameter. The routine assumes that the semaphore is initiated by initsem ().

**Time synchronization of processes**

The task in this exercise is to construct a chain of events of synchronized processes. Processes "P1", "P2", "P3" and "P4" from the previous exercise shall be executed in this order and be repeated in an unfinished chain.

The processes must be modified so that they only write one character at a time to the screen.

**Tip:** You can call the semaphore (*waitsem (Sx)*) in one process while releasing the semaphore by (*signalsem (Sx)*) in another process. In this way, you can synchronize the execution order.

```
PROCESS Pnr(void){
    DO_FOREVER {
        waitsem(S1) ; // wait until S1 free. Couls be done by any other process.
        _outchar('nr');
        Signalsem(S2); // Freeing S2 that some other process is waiting for.
    }
}
```

Modify the processes in the previous step so that the printout from the processes changed to '1234123412341234 ... ' etc.. Synchronization is done by using semaphores.

### Exercise 6\_7 controlling the door system with process synchronization

(4p)

After a series of operations aimed to make you familiar with a real-time kernel and get you an idea of how it can be used, it is time for you to solve a minor task namely the controlling of the door as you done previously but now we will split the program in several processes.

#### Overall description.

The control of the door ML13 will now be implemented in a C program structured in several processes with the support of the real-time kernel RTK12. We will use semaphores for synchronize between the processes and by that ensure that only processes that for the moment is useful for the system due to the system status. For example, if the door is closed its only of interest to check if any pushes the sensor for open the door and no meaning of checking the sensor for opened door. In addition to earlier demand, open and close the door, we have the following demands:

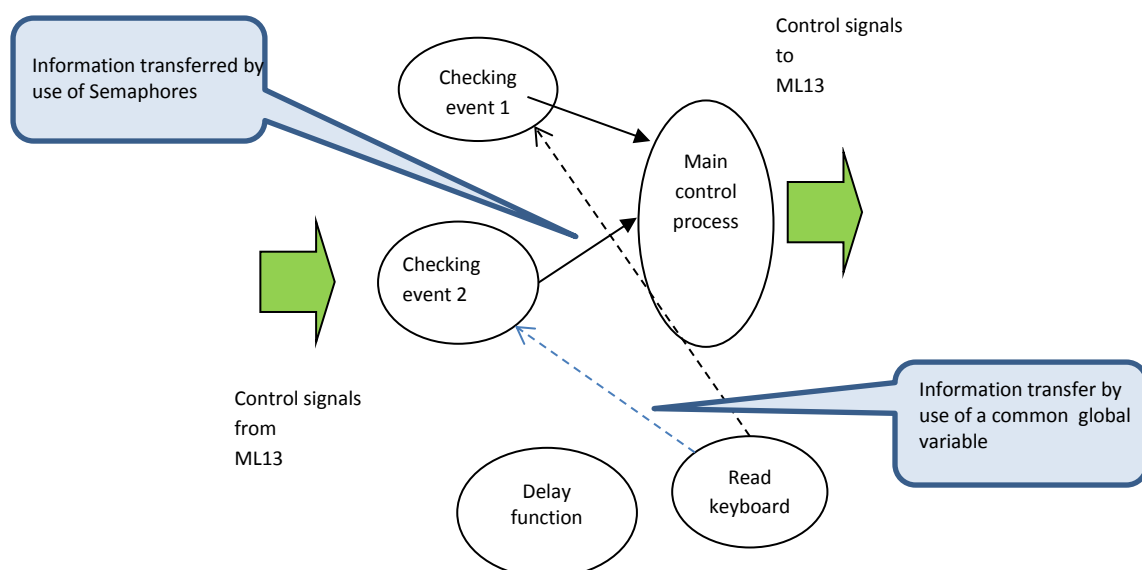
The door shall be possible to 'lock' by pushing the key 'l' on the PC keyboard and unlocked if pushing the 'u' key. Locking/unlocking the door may be done at any time and shall be manage by a separate process reading the keyboard. ( Tip : to read the key use the special XCC function `_tstchar()` ). A way to indicating the status locked / unlocked of the door is to use a global variable (*int locked*).

With "locked" door it means that it can't be opened by pressing any of the buttons on the ML13. The door should be locked from the start.

Below you will find a proposal on how to fundamentally solve the task. You are not bound to in detail follow these steps, however, the "execution model" (see below) should be used in some way and the idea of several processes shall be adopted.

#### Execution Model

The program will be designed by a number of processes that are synchronized with help of semaphores and has a structure as below figure:



In the center we have a main process that provides all control signals to the door (open, close). The main process can be synchronized with the processes that check the status of the door unit, preferably with semaphores. The following pseudo-code indicates a possible solution:

In the setup phase in main all semaphores are locked at the beginning. All processes except `manage_door()` calls for a semaphore in beginning and will be moved to a queue for the called semaphore. `Manage_door()` releases one semaphore and by that the process waiting for that specific semaphore can continue when.

```
PROCESS manage_door () {  
    systemSetup  
    do forever {  
        enable "open_door_event" -process // (release the semaphore)  
        wait for "open-door" signal // ( wait for the event to occur)  
        open the door;  
        enable "door_opened_event" -process  
        wait for "door-opened signal"  
        wait for 5 seconds  
        close the door  
    }  
}
```

Note below especially how semaphores can be used to "enable-events".

The execution model above shows how a number of processes is monitoring the various events that can occur in the door unit. For example:

- ✓ Sensor is activated
- ✓ The door is wide open
- ✓ The door is fully closed ( not necessary to use in this solution)

Such a process can be designed as follows:

```
begin do forever:  
    wait for the specific semaphore for activate this process;  
    begin do forever  
        if (sensor pushed and door not locked)  
            signal (open_the_door_sem)  
            break inner loop  
        else  
            call yield() function// shift to next process  
        endif  
    end  
end
```

### The "yield" function

Most RT operating Systems has a yield-function which when it's called leaves the remaining time of the calling process TIMESLICE to next executable process in the ready queue. In RTK12 there is no such feature, so you have to write it by yourselves. It is a rather easy function but you have to use a number of functions from the RT Kernel so you just get it here and you can copy it to your program.

The code for the function is as below:

```
Void    yield(int dummy)
{        // Dummy parameter is for the 'suspend' to work well

        DISABLE;                // can't take process switch here
        insert_last(Running, &ReadyQ); // Calling Process last in Readyque
        suspend(Running);           // Saving context for Calling process
        dispatch();                // Start next process in ready que
    }
```

### The Sleep() function

To create a delay time, in our case a time during which the door is fully open, you can create a function that waits until the RT-kernel system clock reaches a certain time. The function consists of an infinite loop in which it tests if the clock has reached the defined end time. If not the function call yield() and by that start the next process in the queue. Next time the calling process get execution time the function again tests the time. When the time at last reach the end time the function will end and the calling process can execute further. In our hardware the sleep function will give a delay of (delay\*100) ms but in the simulator mode it will be longer delays, approximately (delay)\* 1 seconds.

The following code shows a simple implementation of the "sleep" function:

```
Void    sleep(int delay){
    int    wakeup;
    wakeup = get_rtk_time() + delay; /* sleep for delay * TIMESLICE*/

    while(1){
        if( wakeup > get_rtk_time() )
            yield(0);                /* not morning yet... */
        else
            return;
    }
}
```

### What to do:

You shall develop a C-program that controls the opening and closing of the door with the same demand as in the exercise 6\_4. Additional it shall be possible to, via a key on the PC keyboard, at any time lock or unlock the possibility to open the door. The program shall follow the above described principles of using several processes. If you think there is some missing in the description of how the opening/closing of the door shall function feel free to decide by yourself but of course it shall be a realistic way of function.

----- End of exercises for XCC12 users -----

For alternative WP 6 exercises se next page.

### Alternative WP 6 exercises for those not using the XCC12 IDE for a MCC12 CPU.

Below you find alternative exercises 6\_3alt to 6\_5alt that you can do if you not having access to XC12 IDE.

#### Exercise 6\_3alt (*exerc 6\_3alt.c*) Time based control program (2p)

In this first exercise you should develop a draft for a program, possible to compile but not possible to run or test without access to XCC12. **You should just show and explain the solution for the TA.**

You shall develop a time-based implementation of an automatic door. The door will operate in the following manner:

- ✓ If someone approaches the door space (button S1 /S2 pushed) the door will open.
- ✓ After the door had been open (in several seconds, controlled by a simple loop) the door should close automatically.
- ✓ The door is closing very slowly, takes several seconds to close. If someone approaches the door space while the door is closing, it shall immediately reopen.
- ✓ The main structure of the program is described roughly by the following flowchart:

The implementation will be done entirely in the programming language C and in one single module (file).

#### Control of the door unit ML13.

The door is connected to the computer system and controlled via four different 8 bits registers which is possible to read or write to depending on the register.

#### About the registers for this task:

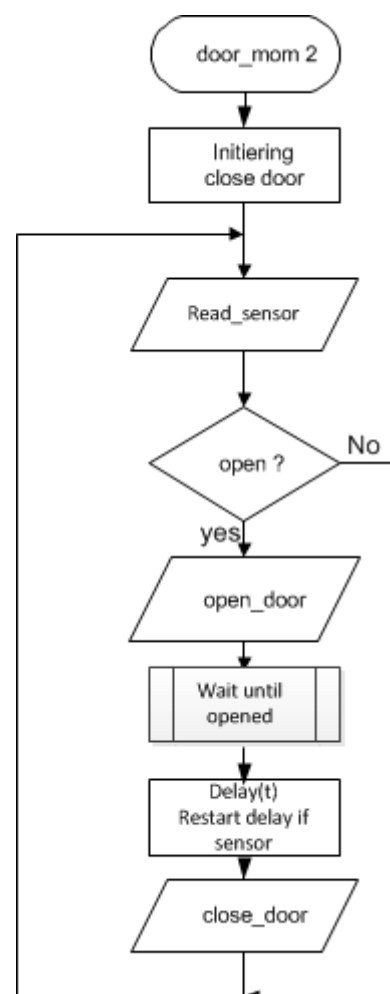
**Control register:** Address 0xB00 (write operation).

By bit 0 you can start opening the door, it will open slowly so it will take some time before it is fully open. By bit 1 you can start closing the door, it will close slowly. Closing and opening of the door could be done anytime even if the door is in its opening or closing phase.

Bit 7-2, Not used.

Bit 1, CLOSE write 1 => Actuate "Close the Door"

Bit 0, OPEN write 1 => Actuate "Open the Door"



**Status register :** Address 0xB00 ( read operation). Each bit is represented the status of the door. For example, door is closed, door is opened. You can by reading the bits in this register check the actual status of the door.

Bit 7,	CLOSING.	1 = Door is closing slowly.
Bit 6,	OPENING.	1 = Door is opening slowly.
Bit 5,	S2: Sense 2.	1 = Door is closed
Bit 4,	Not of interest	
Bit 3,	S1: Sense 1.	1 = Door is wide open.
Bit 2,	Not of interest	
Bit 1,	RIGHT: Sensor B.	1 = Sensor B is activated (Button S2)
Bit 0,	LEFT: Sensor A.	1 = Sensor A is activated (Button S1)

**To do:**

Your task is to write a draft for a control program. Below you have a program head with some declarations and other initiations. Try to make a draft for a control program that solve the problem with a structure as flowchart above views. The Delay() function can be implemented as a simple for-loop with a counter.

Compile the program to ensure the syntax. **You can't test the program.** The only thing you can do is to explain the program to the TA and they will respond if it seems to be a working solution or not. If rather good he will give you a pass code and you can hand in the solution.

```
#define ML13_Status    0x0B00
#define ML13_Control  0x0B00

void main () {
...

}
```

**Exercise 6\_4alt (exerc 6\_4alt.c)**

**(2p)**

**Timer-controlled I/O management**

To control events in a program, you can use a "process" (in this case a thread) that manage a global system time variable tick. For example, the time could tick with a resolution of 0,1 seconds.

With help of this function you can read the system-time in ms :

```
#include <sys/time.h>

double get_time_ms(){
    struct timeval t;
    gettimeofday(&t, NULL);
    return (t.tv_sec + (t.tv_usec / 1000000.0)) * 1000.0;
}
```

If you let a thread function call this get\_time\_ms() and by help of this count a global program time counter variable *program\_time* with a resolution of 1 s.

Then it is possible to control all other events with help of the *program\_time*. For example, you can read a special in-port every x seconds.



**To do:** Write a program with the following structure:

```
int program_time; // The global time, start value 0

int main(){

    // Start up the thread time_count.
    // Start up the thread read_inport.

    While ( program_time < 50){

        //Print out system time every second.

    }
}
// --- End of main thread -----

// ----- Tread functions --
//-----
void *time_count( ....) {

while ( program_time < 50){
    // Check system-time ( get_time_ms())
    // Increase program_time by one every second.
}
// exit thread;
} //End-----

void *read_inport(...){
    while (program_time<50){
        // Read Inport every 5 second.
        ( Simulate this just by print out a text : Reading Inport now)
    }
    Exit thread
} //End-----

// ----- Function get_time_ms -----
double get_time_ms(){
..
...
} // ----- End -----
```

Tip: Try to do the printing of program\_time every second on same row until a printout of reading inport is done.

Example of console text:

**Program time 26**  
**Reading inport now**

**Program time 31**  
**Reading inport now**

**Exercise 6\_5alt**      (*exerc 6\_5alt.c*)

**(5p)**

**Testing synchronization of threads in C-programming**

By using a library pthread.h you can create a number of threads (light weight processes) from your program. This task for you is to test threads and some synchronization primitives (semaphores and condition variables).

You can find related information about this at:

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

The main task for you is to try to develop a program demonstrating a circular buffer in which it should be possible for threads to write in buffer ( in this case a character) and to read a character (take out) from buffer. All writing and reading from buffer should be without losing data or getting it out in wrong order.

The Buffer and its two index number is global

Global / Shared variables

```
char buffer[MAX];           // circular buffer . Test for MAX 5 and 10.
int write_pos;              // index for next character to be put in buffer (put)
int read_pos;               // index for next character to be read ( fetch )
int count;                  // the number of characters in buffer not fetched.
```

For synchronization there are some function in the pthread library you can use:

pthread\_mutex\_init(); pthread\_mutex\_lock(); pthread\_mutex\_unlock() and  
pthread\_cond\_signal(), pthread\_cond\_wait.

To compile the program you have to add the library pthread when compiled as follows:

```
gcc filename.c -o program -lpthread
```

**The program should have the following structure and function.**

A main program for setting up and start running two threads as described below. The main program should after started up the threads continue in a loop as you can see in the program skeleton below. (You find the code at the homepage in file **skeleton\_wp6\_6\_4.c**)

The first thread, **producer()**, should create characters a,b,c...z ; a,b,c.... and store them in the buffer in a infinite loop. For every character stored it should call a cond\_signal telling other threads that buffer not empty. If buffer full (MAX number of characters in buffer) the thread should wait for a *cond\_signal* 'not full' from the other consuming thread telling that buffer is not full. The second thread, **consumer()**, should fetch out the character in first position to be fetched from the buffer and

print it out in the console window. If no character in buffer it should wait for a *cond\_signal* 'not empty' from the producer thread that signal every time it stores a new character in buffer. The result should be that the program prints out 'abcdef...z'abcd...z' in correct alphabetical order. Between the letters the main() loop prints out the textstring when it is running as the picture in figure below shows one possible example of output.

You can find additional information regarding circular buffer, pthread and so on internet

// Skeleton for exercise nr 4 in WP 6 course DIT165.

// File skeleton\_wp6\_6\_4.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

pthread\_mutex\_t count\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t not\_empty =

PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t not\_full =

PTHREAD\_COND\_INITIALIZER;

// Global buffer and corresponding counters.

char letter = 'a'; //the starting letter

#define MAX 10//buffer size

unsigned short count = 0;

char buffer[MAX]; // circular buffer

int write\_pos = 0; // index for next character to be  
put in buffer (put)

int read\_pos = 0; // index for next character to be  
read ( fetch )

void \*put();

void \*fetch();

int main(){

int i;

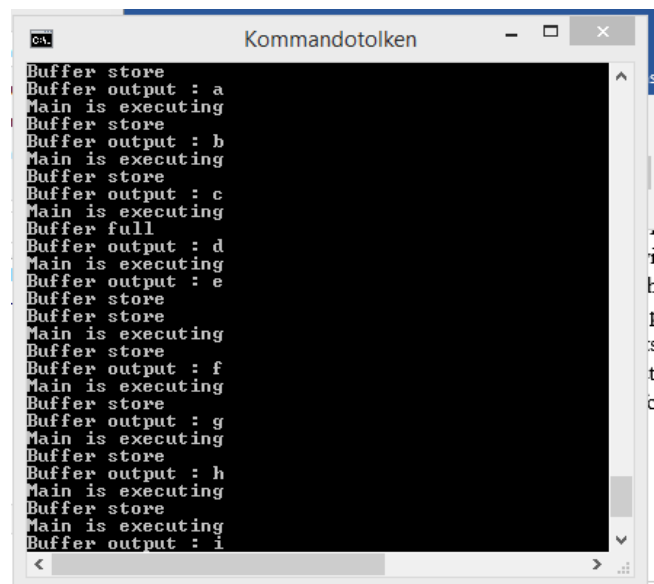
while(1){

....

....

}

}



```
Buffer store
Buffer output : a
Main is executing
Buffer store
Buffer output : b
Main is executing
Buffer store
Buffer output : c
Main is executing
Buffer full
Buffer output : d
Main is executing
Buffer output : e
Buffer store
Buffer store
Main is executing
Buffer store
Buffer output : f
Main is executing
Buffer store
Buffer output : g
Main is executing
Buffer store
Buffer output : h
Main is executing
Buffer store
Buffer output : i
```

```
void *put(){  
    while(1){  
        ....  
        ....  
    }  
}
```

```
void *fetch(){  
    while(1){  
        ....  
        ....  
    }  
}
```

----- End of WP 6 assignments exercises -----

## Some theory exercises regarding real-time systems for course week no 6.

Below you find some theory exercises regarding Real-Time systems suitable to solved or answered before the final exam. Solution of this should not be handed in or demonstrated for any TA.

After all exercises you find proposals for answers or solutions.

### RT exercise 1

Suppose we have a system consisting of a number of periodic processes that are executed with support of a real-time operating system. As usual, the processes can be described with values according to  $P_n (p_n, c_n, d_n)$ .

Assume that the processes are executed with the support of a dynamic scheduler and that we have at least three processes P1 to P3 where the processes are prioritized according to P1 highest priority and P3 lowest of the three. The scheduling uses pre-emptive scheduling.

P1 and P3 shares a semaphore to ensure mutual exclusion of their critical regions.

In connection with the use of semaphores and different priorities, something called priority inversion may arise. Try to briefly explain how the problem can arise and also indicate how one can handle and partly solve the problem in a real-time operating system.

### RT exercise 2

Suppose we have a set of processes with data according to the table below.

The processes should be scheduled with a static schedule. The processes are of the pre-emptive type.

a) Calculate the LCM (Least Common Multiple) number for the process time periods. Then adjust the period times in a suitable manner so we get a lower LCM number for the period times suitable for the scheduling and then calculate the new LCM number.

b) Draw a possible static schedule for the processes based on the adjusted periodic times.

Process	p	c	d
P1	6	2	4
P2	10	2	5
P3	16	5	10

### RT exercise 3

Suppose we have a set of processes with data according to the table to the right. The processes P1 and P3 do include critical regions for which mutual exclusion is managed by a semaphore S1.

Proc.	Pri.	p (ms)	d(ms)	c(ms)	Semaphore S1
P1	0	6	6	2	2 ms
P2	1	13	8	3	
P3	2	25	15	5	3 ms

Decide if the system is possible to schedule according to a RMSA analysis.

#### RT exercise 4

Suppose we have a system with three processes with program code as figure to the right. The processes are executed in a system supported by a Real-time operating system with support for semaphores. Process two and three share a semaphore to ensure mutual exclusion for the respective critical regions. The processes have fix priority according to P3, P1 and P2, where P3 has the highest priority.

Execution times for the functions are: compute1 (): 5 sec, compute2 (): 4 sec and compute3 () 2 sec.

The functions: parkForEvent\_x () means that the process awaits a certain event no x to occur and is not assigned execution time during this wait. When an event occurs, the process changes to Ready and is given execution time based on the fixed priority it has, which may mean that it might either wait until a higher priority processes is completed or given execution time immediately if it has a higher priority than the one currently executed.

Assume that events # 1 - 3 occur in the following order, event #2 , event #3 and last event #1, with one second between each event. We must assume that no other processes affect the execution of the three processes.

We also assume that we start with all processes in the parkForEvent () mode.

```
Process 1() {
    while(1) {
        parkForEvent_1();
        compute1();
    }
}
//-----

process2(){
    while(1) {
        parkForEvent_2();

        waitsem(S1);
        comput2();
        signalsem(S1)
    }
}
//-----

Process3(){
    while(1) {
        parkForEvent_3();
        waitsem(S1);
        compute3();
        signalsem(S1)
    }
}
//-----
--
```

#### To do :

a) Draw a time diagram for the execution of all processes from the first event until all three processes have performed one iteration of their executions, ie the function computex() has been executed for one period.

Draw the time diagram so that the state of the processes (**waiting and running**) is shown.

b) Due to the use of the semaphore, one will have an undesirable effect in the execution of the processes. Describe this effect and specify what it is called.

c) There is a method to minimize the impact of this effect. Briefly describe this method and what it usually is called.

---

### Proposals for answers or solutions for the theory exercises regarding Real-Time systems.

#### Answer to RT exercise 1

Suppose the following scenario together with the use of semaphore S1:

P3 executes, calls, and blocks S1, executes a part of it's critical region.

Process change, P1 starts executing, calls S1, S1 is already blocked and P1 moves to the wait queue for S1.

Process change, P2 executes its entire time period.

Process change to P3, continue executes the critical region, releases semaphore S1.

Process switch to P1 which now can block the semaphore and execute further.

As a result of P3:s blocking of the semaphore P1 will be delayed in it's execution by this lower process P3. In this case P3 is working as it had a higher or equal priority than P1. This effect is called (Priority inversion).

The problem is solved by minimizing the delay (blocking) by temporarily assigning P3 to the same or higher priority than P1 when P1 calls S1. P1 can then execute further and then release the temporary priority at same time as it releases the semaphore. The method is called Priority Heritage

#### Answer to RT exercise 2

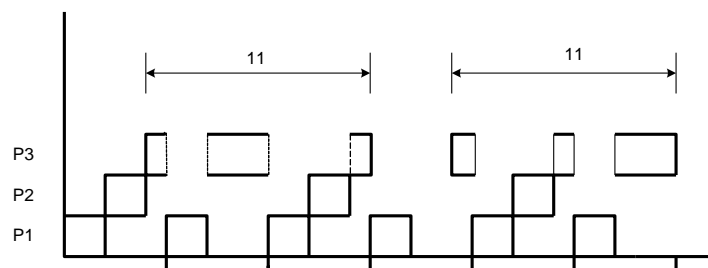
$$\text{LCM}(6,10,16) = \text{LCM}(\text{LCM}(6,10),16) = \text{LCM}((6*10)/2,16) = 30*16/2 = 240$$

[ Based on the fact that  $\text{LCM}(a,b,c) = \text{LCM}(\text{LCM}(a,b),c)$  and

$\text{LCM}(a,b) = (a*b)/\text{lcd}$ ; lcd : least common denominator (minsta gemensamma nämnaren )

Adjust the periodic times to 5,10,15 witch gives  $\text{LCM} = 30$  (Alt. 6,8,16 with  $\text{LCM} = 48$ )

b) Possible static schedule for the period times 5,10,15



Note: If we assume that the system is scheduled by a dynamic scheduler under run-time you can with a RMSA analyses calculate the response time for P3 to 15 ms. This result because RMSA gives the worst-case response-time. Worst case is when all three processes are ready at the same time (here

$t=0$ ). P1 will run first then P2 and last P3 starts and it takes 11 ms to execute finished as the schedule views.

### Answer RT exercise 3

Blocking of the semaphore by P1 and P3 gives the blocking factors:  $b_1 = 3$  ms and  $b_2 = 3$  ms.

R3 is then calculated to 12 ms with RMSA analyses which is less than deadline  $d_3 = 15$  ms

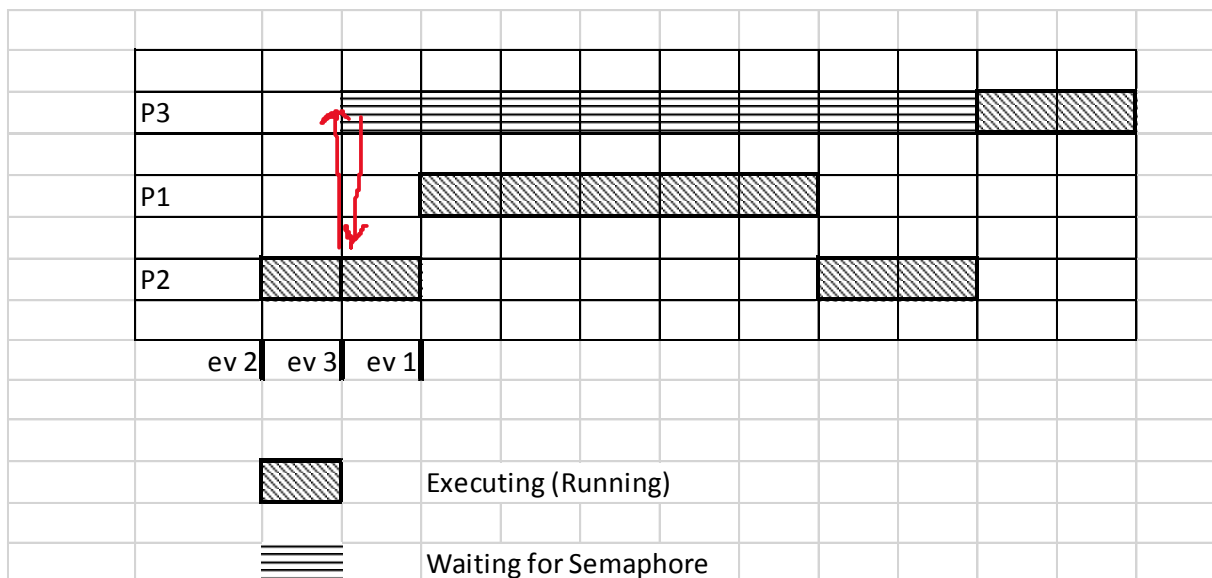
R2 is calculated to 10 ms which is greater than the deadline  $d_2 = 8$  ms.

The system is by this not possible to schedule with a dynamic scheduler with the given fix priority (RMSA) due to the fact that one process don't meet the requirement for the deadline.

### Answer RT exercise 4

P2 starts running and blocks the semaphore S1, at event 3 P3 starts running due to its higher priority. P3 asks for blocking the semaphore but it's already blocked by P2 and P3 sets to waiting for semaphore. Only P2 is in ready mode now so it continues running until event 1 occur which make system switch over to run P1. P2 still holds the semaphore so P3 stays in waiting mode. P1 ends and P2 gets into run mode and release the semaphore and ends. P3 is now getting running time and can block the semaphore and execute until its end.

As the figure shows, P3, despite its high priority, will have to wait for both P1 and P2. P1 and P2 operate in this position as if they had a higher priority than P3. The effect is called Priority inversion.



c) By then, when P2 takes the semaphore let P2 obtain the highest priority of the processes that handle the semaphore, P2 will have to execute the critical region (event function) uninterrupted (in this case) which minimizes the delay of P3. The method is called Priority Heritage



