ECE3073 Computer Systems

Real Time Systems Lab 2 – IPC with Concurrent Tasks

© Lindsay Kleeman, Monash University, Dept ECSEng.

0 Aims

- Design and test an alarm clock using an imbedded real time system running a real time kernel.
- Implement Inter-Process Communication (IPC) between concurrent tasks with semaphores.
- Understand the need and means of implementing mutual exclusion protection for data structures shared by multiple processes using semaphores.

1 Equipment

- DE2 board.
- Quartus/Nios II development software containing the uC/OS-II real time kernel.
- These notes are prepared for Quartus version 13.0sp1.

2 Preliminary Work

Print and read this document and bring it to the lab. During the lab, ensure you write answers next to these ☼ markers. Where you see a ☻ ask your demonstrator to initial your sheet in the lab to indicate you have answered the questions correctly and demonstrated your work.

Read the documentation and listen to lectures on semaphore functions *OSSemCreate(), OSSemPend(),* and *OSSemPost()*. See the unit web page resources section for the documentation uC/OS-II Real Time Kernel Reference Manual.

Start to plan the software and specifications for section 3 before the lab. You will implement and test this code during the lab session.

3 Method

3.1 Hardware Definition

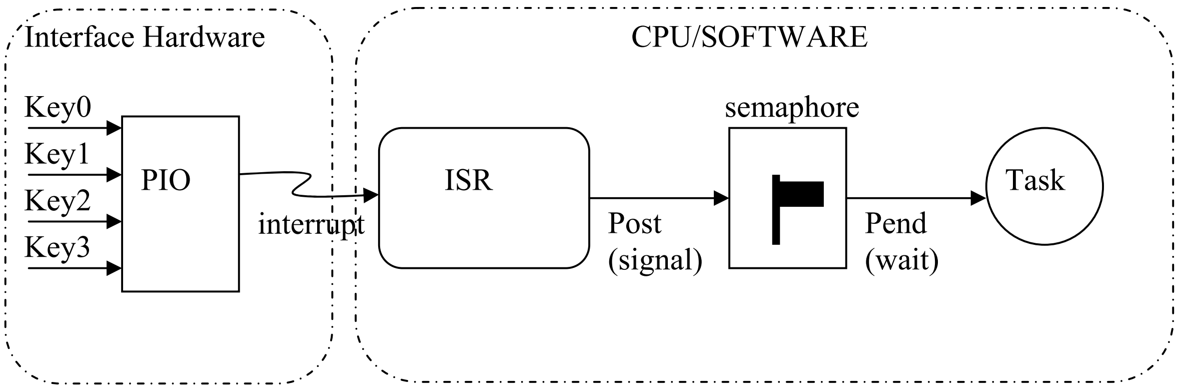
Start with the Quartus project archive from the first real time laboratory exercise available on the unit website. Within Qsys, edit the module KeyButtons (4 bit PIO interface to the 4 push buttons Key 0 to Key 3) by double clicking. Change the module to include an interrupt that is generated when a change in a button occurs – allow for both 0 to 1 and 1 to 0 changes.

You should consult the data sheet for the PIO module to determine what options to use when altering the KEY module. Click on the Documentation button on the top right of this parallel port wizard. Then click on the datasheet link which should open a PDF file datasheet. This will be also needed to work out how the software can clear the interrupt and determine which key is pressed or released.

Auto assign Interrupt numbers via the System menu. Generate the system and update the CPU Verilog instantiation in the Quartus project. Note that the ports for the CPU should not change, yet the project needs to be compiled and downloaded into the DE2 board from Quartus.

☻

3.2 Preparing for Interrupts and Writing the ISR



Pressing or releasing any of Key3, Key2, Key1 or Key0 will generate an interrupt. In the software we need to do the following in order:

1. globally declare a semaphore:
2. create a semaphore and check whether this has been successful by testing the returned error codes.

3. initialise the PIO hardware by clearing any pending edge captures
4. register the location of the Interrupt Service Routine (ISR) for the particular IRQ line driven by the PIO
5. enable interrupts for each bit of the edge capture register in the PIO.

Label each part against the code outline below.

The Interrupt Service Routine should be short so as not to delay other interrupts. The ISR performs the following actions:

- clears the interrupt source
- signals a semaphore.

A task should be written to wait on the semaphore and implement the response to a change in Key status.

```
#include <stdio.h>
#include "includes.h"
#include "system.h" // contains definitions of device addresses/IRQs
  // eg KEY_BASE, KEY_IRQ.  Use these symbols so your code will still
  // work after a change in SOPC addresses or IRQs
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
#include "sys/alt_irq.h"


void * context;
OS_EVENT * sem_key_change;  // semaphore declaration

static void KeyISR(void * isr_context, alt_u32 id)
{
  // TODO: clear interrupt source using IOWR(KEY_BASE, ??, ??)
  //              see documentation for PIO mentioned in section 3.1
  // TODO: signal semaphore using OSSemPost()
}
```

… task definitions etc including task that calls OSSemPend()

```
int main(void)
{
  // TODO: initialize, check semaphore create using ??=OSSemCreate(??)
  //        if (??? == 0) ??

  IOWR(KEY_BASE, 3, 0); // clear KEY PIO interrupt capture register,
                        // hence clears any pending interrupts

  alt_irq_register(KEY_IRQ, context, KeyISR);
            // registers and enables KEY_IRQ to interrupt

  IOWR(KEY_BASE, 2, ??? );
  // enable edge capture bits to generate interrupt for lowest 4 bits

  OSTaskCreateExt(……)
  OSStart();
}
```

Complete the code and test it by inserting a *printf* statement in the ISR temporarily (not a good thing to leave in an ISR that should minimize its execution time).
Remove the *printf* from the ISR and move it to the task waiting on the semaphore and also print out a message and the switch status using the integer

status=IORD(KEY_BASE,0); from the task.

What happens if the switch status changes between the ISR and the task waiting on the signal? Is this likely in practice and is it really a problem?
☼

Show your demonstrator.
☻

3.3 Task to process button changes

Design an appropriate user interface of your choice involving the Key buttons for the alarm clock that provides the facility to:

- change the clock time,
- change the alarm time
- turn the alarm on/off and show the alarm status
- select the time or alarm for display on the 7 segment displays.

Write a specification for your user interface here:
eg: Key0: Pressing Key0 increments the minutes of the current alarm or time determined by which ever is currently displayed. Initial rate is 4 minutes per second button pressed and increases rate to …..

☼ User Interface Specification:

Use the task that waits on the semaphore, referred to above, to process changes in the Keys. Your task should contain an infinite loop that waits on the semaphore signaled from the ISR, reads the Keys when they change and implements their actions. Note that the variables that store the current time and alarm time are also used in other tasks and therefore they must protected by mutual exclusion semaphores as discussed in lectures.

☻

3.4 Complete the code

Complete the code for the alarm clock to update the task that checks for an alarm event and communicates this to a dedicated task that handles the alarm function. For simplicity you may assume that the alarm stops after a fixed time. Remember that the alarm is implemented with the wave pattern of LEDs that you completed in the first laboratory. Also watch out for deadlock conditions in your implementation. Waiting on semaphores in the ***same order in all tasks*** is a good way to avoid common deadlock situations. Show a demonstrator your working system.

☻

4 Conclusions

This lab has introduced the use of semaphores in a real time kernel and how an event driven system is implemented. Multiple simple tasks have been organized to collaborate via semaphores to achieve a global specification. The practice of using a semaphore that signals in the ISR minimizes the processing performed by the ISR and thus reduces overall interrupt latency in the real time kernel. The lab also highlighted the practice of protecting shared data structures from corruption and inconsistent states due to simultaneously by more than one process. This has been achieved through the use of mutual exclusion semaphores.