Engineering Design for Embedded Systems: Assignment 3 (Polling and Interrupts)

Patrick Lam*

Suggested Completion Date: January 28, 2013

This assignment will allow you to practice writing C#-style pseudocode for polling and helps you understand how interrupts work.

Problem 1: Tight Polling, Pressure Sensor

[We discussed this in class on Tuesday, January 15.] A pressure sensor is interfaced with an embedded system using a memory-mapped register at address 0x1200. The pressure sensor readings are represented using the C# type `byte` (range 0..255).

Write a tight polling loop in C# that reads the content of the register and invokes the method `public void HandleOverpressure (byte currentPressure)` if the value reported by the pressure sensor exceeds 110.

```
const  byte MAXPRESSURE = 110;
unsafe  {
  // insert your code here
}
```

Note. You may need to review the section on C# Pointers from your ECE150 course. For your reference, a related extract from ECE150 overheads is available in this folder.

For an example solution, see the end of this document.

Problem 2: Tight polling, Touch Sensor

In an elevator control system, the position of a particular elevator door is sensed by a touch sensor that is interfaced with the control system using a memory mapped register at address 0x1420. When the elevator door is open, bit 3 of the register is 0; when the door is closed,

*Thanks to Rudolph Seviora for an earlier version of this assignment.

bit 3 is 1. The register is 8 bits wide (C# type: byte); bits 0–2 and 4–7 of the register contain values sensed by other sensors.

Write a tight polling loop in C# that repeatedly reads the elevator door position and calls a method `public void HandleElevDoorClosing()` when the door closes (i.e. bit 3 of the register changes from 0 to 1).

Problem 3: Writing Output to Actuators

In an embedded system, the current applied to a motor is controlled through a memory-mapped register at address 0x3100. The register is 8 bits wide (C# type byte). The current applied to the motor is equal to <register content>$\times$ 0.1 amperes. That is, the minimum current is 0 and the maximum current is 25.5 amperes.

Write a C# method `public bool SetMotorCurrent (float iMotor)`, where the value of `iMotor` is in amperes. If the value of the parameter `iMotor` is within the range of allowed values, the method writes the (scaled) parameter value into the register and returns true. If the parameter value is negative, the method writes 0 into the register and returns false. Finally, if the value is $>$ 25.5 amperes, the method writes the max value of current (255) into the register and returns false.

Problem 4: Periodic Polling I

This is a variant of Problem 1. Instead of tight loop polling, the pressure sensor readings are to be checked only once every 100 msec. Use Android `Handler`s and `Runnable`s (since we haven't talked about C# timers). Since we can't actually read sensors in Java, just call the static method `PressureSensor.readPressure()`, which returns a `byte` containing the sensor reading. Read the sensor register contents every 100 msec and call the method `handleOverpressure` if the value read exceeds 110.

Problem 5: Periodic Polling II

This is a variant of Problem 2. Instead of tight loop polling, the touch sensor readings are to be checked only once every 500 msec. Using an interval timer, write Java code that reads the register contents every 500 msec using the static method `ElevatorStatus.getStatus()`, extracts the bit corresponding to the elevator door status, and calls the method `public void handleElevDoorClosing()` when the door closes (bit 3 of the register changes from 0 to 1).

Problem 6: Smoothing Sensor Readings

Consider Problem 4. One of the "real-life" problems that affect such code is that the value the pressure sensor writes into the memory-mapped register may be corrupted by transient noise.

(Also a problem in Lab 2!) Such transients are typically short. A defence against transient noise is to wait for the next reading (100 msec later), and invoke the `HandleOverpressure` method only if the two consecutive readings are above maxpressure. Write a Java code fragment that would do such "noise filtering" and invokes the method `HandleOverpressure` only if both consecutive readings (i.e. the readings at T and at T+100 msec) exceed 110.

Problem 7: Interrupts vs Events

Imagine the following situation. Your Android phone is running an application which contains a click listener for a button. Now, the user clicks on the button. What happens? Describe a plausible sequence of interrupts and events generated by the touch screen interface, through the CPU's actions, all the way to the click listener. Include details on how Android is involved.

Problem 8: Handlers and Concurrency

Consider this Android code fragment:

```
Runnable r = new Runnable() {
  public void run() {
    System.out.println("X");
  }
};
Handler h = new Handler();
h.postDelayed(r, 5000);

for (int i = 0; i < 10; i++) {
  System.out.println(i);
  Thread.sleep(1000);
}
```

(a) What is the output of this code? (b) Now imagine that the program did not contain `"h.postDelayed(r, 5000);"`, but instead, an interrupt occurred 5000ms after the start of the loop. The interrupt handler runs `r`. What output would you expect, and why?

Example solution for Problem 1

```
using System;

class A3Q1
{
    public static void Main()
    {
         // start of solution --------------
         const byte MAXPRESSURE;
         unsafe{
            byte* ps;
            ps = (byte*)0x1200; //note: will fail in VS, no mem

            while (true)
            {
                byte pTemp = *ps;
                if (pTemp > MAXPRESSURE)
                    HandleOverpressure(pTemp);
            }
        }
        // end of solution -----------------
    }

    public static void HandleOverpressure(byte p)
    {
        Console.WriteLine("Overpressure! p={0}", p);
    }
}
```