EECS 461, Fall 2009, Problem Set 7[1]

issued: Thursday, November 12, 2009 due: Thursday, November 19, 2009

To receive full credit for your answers to the following questions, please explain your reasoning carefully.

1. In the following code, the function `lSecondsSinceMidnight` returns the number of seconds since midnight. A hardware timer asserts an interrupt signal every second, which causes the microprocessor to run the interrupt routine `vUpdateTime` to update the static variables that keep track of time. (Note that an RTOS is not used in this problem.)

```
static int iSeconds, iMinutes, iHours
void interrupt vUpdateTime (void)
{
    ++iSeconds;
    if (iSeconds >= 60)
    {
        iSeconds = 0;
        ++iMinutes;
        if (iMinutes >= 60)
        {
            iMinutes = 0;
            ++iHours;
            if (iHours >= 24)
                iHours = 0;
        }
    }

          !! service the hardware

}

long lSecondsSinceMidnight (void)
{
    return( (((iHours * 60) + iMinutes) * 60) + iSeconds);
}
```

   (i) Does this code always return the correct answer? Explain.

   (ii) The ANSI C standard allows the compiled C code to read in the variables in any order, including the following: first `iHours`, then `iMinutes`, and lastly `iSeconds`. What is the maximum amount by which the returned value of time may be off? (Hint: Suppose that the time is 5:59:59.)

   (iii) Suggest a way to fix this code so that it returns the correct time.

[1] Revised November 12, 2009.

2. The interrupt routine in the following code is the same as that in Problem 1. The subroutine `vSetTimeZone` changes the time zone by changing the `iHours` variable. This subroutine takes into account the difference in the two time zones and then makes adjustments to deal with the fact that one or both of the two time zones may currently be observing daylight savings time. To reduce the period during which this subroutine must disable interrupts, the subroutine copies the `iHours` variable into the local, nonshared `iHoursTemp` variable, does the calculation, and copies the final result back at the end. Does this work? Explain.

```
static int iSeconds, iMinutes, iHours;
void interrupt vUpdateTime (void)
{
    ++iSeconds;
    if (iSeconds >= 60)
    {
        iSeconds = 0;
        ++iMinutes;
        if (iMinutes >= 60)
        {
            iMinutes = 0;
            ++iHours;
            if (iHours >= 24)
                iHours = 0;
        }
    }

          !! service the hardware

}
void vSetTimeZone (int iZoneOld, int iZoneNew)
{
    int iHoursTemp;

    /* Get the current 'hours' of the time */
    disable ();
    iHoursTemp = iHours;
    enable ();

    /* Adjust for the new time zone */
    iHoursTemp = iHoursTemp + iZoneNew---iZoneOld;

    /* Adjust for daylight savings time, since not all places in the world */
    /* go on daylight savings at the same time */
    if (fIsDaylightSavings (iZoneOld))
       ++iHoursTemp;
    if (fIsDaylightSavings (iZoneNew))
       --iHoursTemp;

    /* Save the new 'hours' of the time   */
    disable ();
    iHours = iHoursTemp;
    enable ();
 }
```

3. The task code and the interrupt routine below share the variable `fTaskCodeUsingTempsB`. Is the task code's use of that variable atomic? Is it necessary for it to be atomic for the system to work?

```
static int iTemperaturesA[2];
static int iTemperaturesB[2];
static BOOL fTaskCodeUsingTempsB = FALSE;

void interrupt vReadTemperatures (void)
{
   if (fTaskCodeUsingTempsB)
   {

          iTemperaturesA[0] =  !! read data from hardware
          iTemperaturesA[1] =  !! read data from hardware

   }
   else
   {

          iTemperaturesB[0] =  !! read data from hardware
          iTemperaturesB[1] =  !! read data from hardware

   }
}

void main (void)
{
   while  (TRUE)
   {
      if (fTaskCodeUsingTempsB)
         if (iTemperaturesB[0] != iTemperaturesB[1])

           !! Set off alarm;

      else
         if (iTemperaturesA[0] != iTemperaturesA[1])

           !! Set off alarm;


      fTaskCodeUsingTempsB = !fTaskCodeUsingTempsB;
   }
}
```

4. The routines below are called by Tasks A, B, and C, but they don't work. How would you fix the problem?

```
static int iRecordCount;

void increment_records (int iCount)
{
   OSSemGet (SEMAPHORE_PLUS);
   iRecordCount += iCount;
}

void decrement_records (int iCount)
{
   iRecordCount -= iCount;
   OSSemGive (SEMAPHORE_MINUS);
}
```

5. A *reentrant* function is one that may be called by more than one task, and will always work correctly even if the RTOS switches from one task to another in the middle of executing the function. Where would you need to take and release the semaphores to make the function below reentrant?

```
static int iValue;

int iFixValue (int iParm)
{
   int iTemp;

   iTemp = iValue;
   iTemp += iParm * 17;

   if (iTemp > 4922)
      iTemp = iParm;
   iValue = iTemp;

   iParm = iTemp + 179;
   if (iParm < 2000)
      return 1;
   else
      return 0;
}
```

6. Write a Simulink diagram and S-function to convert one 32 bit number into four 8 bit numbers, as shown in Figure 1. You may modify the S-function for converting four 8 bit numbers into one 32 bit number, found on the website. Hand in the Simulink diagram and the C-code.

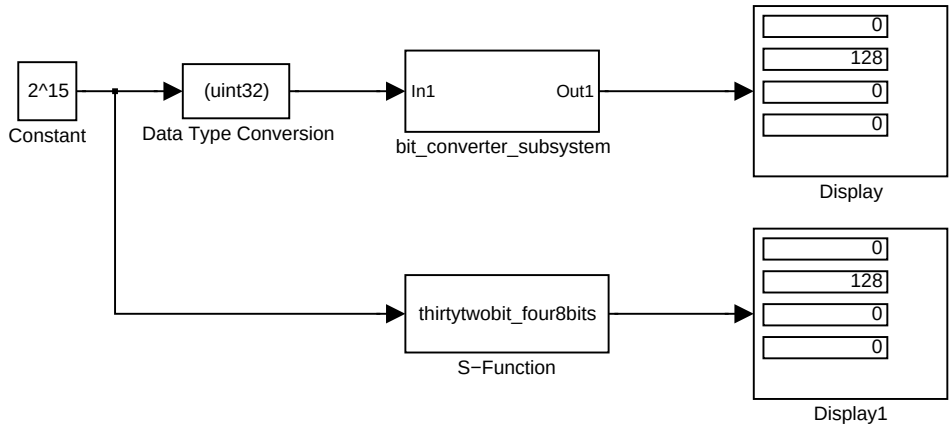

Figure 1: S-function to convert one 32 bit number to four 8 bit numbers.

NOTE: On the website you will find a Simulink .mdl file as well as a .c file that contains the C code for the S-function. To execute the Simulink model, you will need to create a .mexw32 file. To do so, type "mex filename.c" at the Matlab command prompt. If you are asked for a compiler, choose one from the specified list. When you type the name of the S-function into the Simulink block, omit the .c suffix.