Jonathan W. Valvano

First Name: _______________ Last Name:____________________

April 4, 2008, 10:00 to 10:50am

Open book, open notes, calculator (no laptops, phones, devices with screens larger than a TI-89 calculator, devices with wireless communication). Please don't turn in any extra sheets.

(10) Question 1. A CAN system has a baud rate of 100,000 bits/sec and a message protocol with frame sizes of exactly 100 bits. The average time between frames is only 1 frame/sec, but there are times when up to 20 frames are transmitted one right after another (followed by long pauses when no frames are transmitted). The receive process requires real-time solution, because no CAN messages should be lost, and the CAN network will stall if all the CAN receiver frame buffers (receiver frame buffers are in the CAN hardware) are full. The Receive CAN Flag (RXF) is set when there is at least one message in its frame buffers; this flag is armed so it will request an interrupt; and the CAN ISR will remove all receive messages from its buffers (clearing the entire receive fifo). Calculate the worst possible interface latency, which is the maximum allowable time between the Receive CAN Flag (RXF) being set and the execution of the CAN interrupt 38 that reads the messages. Show your work.

(30) Question 2. In this question we will consider implementing the FFT on the 9S12C32. Because, there is no floating support, we will implement the calculations in fixed-point. There will be **n** data points (where **n** must be a power of 2, such as 16, 32, 64, 128, 256, 512, 1024,… etc.) The number **nn** is the number of elements in the input buffer, which is converted to the output buffer by the FFT

```
#define n ???
#define nn (2*n)
```

The input/output data stored in RAM will be

```
long Idata[nn];  // signed binary fixed point
```

Part a) The input data, initially stored into the even elements of **`Idata`**, will be voltages measured by the 10-bit ADC (0 to +5V). The odd elements of **`Idata`** will be initially zero. We will utilize 32-bit signed binary fixed-point. We will use *signed* fixed-point because the output can be negative. We will use *binary* fixed-point because there will be lots of calculations. Binary fixed-point means $\Delta=2^m$ (basically, I am asking, what is m?) What is the best fixed-point resolution for **`Idata`**? Justify your choice.

Part b) What is the largest possible value for size **`n`** on the 9S12C32. Justify your choice.

Part c) Consider this portion of the FFT software, shown here in floating point.

```
        theta = -6.283185307179586476925286766559/mmax;
        wtemp = sin(0.5*theta);
```

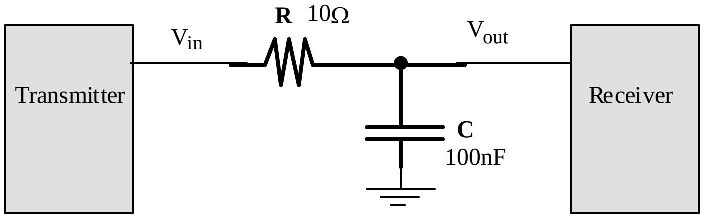
mmax is an integer taking on values of 2, 4, 8, 16, 32, …, **n**/2. Once we convert this to fixed-point, you may assume **Itemp** will be the integer portion of **wtemp**, which is a 16-bit signed binary fixed-point number with a resolution of 2^{-13} (the value of **wtemp** equals **Itemp**$*2^{-13}$, the range of **wtemp** values will be -4 to +3.9999). Assume for this part, **theta** is not used elsewhere in the program, just **Itemp**. Without showing any code, briefly explain how you would implement this part of the FFT (calculating **Itemp** from **mmax**)

Part d) Consider this portion of the FFT software, shown here in floating point.

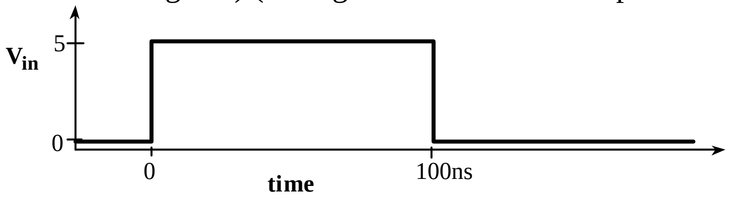
```
    tempr = wr*data[j-1]-wi*data[j]; // Danielson-Lanczos formula
```

j is an integer, guaranteed to properly index into the **data[]** buffer. Once we convert this to fixed-point, you may assume **Iwr** and **Iwi** will be the integer portions of **wr** and **wi**, which are 16-bit signed binary fixed-point numbers with a resolution of 2^{-13} (range of values will be -4 to +3.9999). In particular, **Iwr** and **Iwi** are of type **short**. Assume **Itempr** and **Idata** are the integer parts of **tempr** and **data**, which are both 32-bit signed fixed-point numbers with a resolution defined by you in part a). In particular, **Itempr** is of type **long**. Show the C code to implement this portion of the FFT. You may assume that neither **Iwr*Idata[j-1]** nor **Iwi*Idata[j]** will overflow a 32-bit signed integer. In particular, calculate **Itempr** from **Idata j Iwr** and **Iwi**.

(30) Question 3. We will model the physical communication channel, such as USB, CAN, or RS232 as a single resistor in series with a capacitor, as shown in the figure below.



For this question, assume an ideal transmitter (output impedance of 0) and an ideal receiver (input impedance of infinity). Let **R**=10Ω, and **C**=100nF. Consider a 5V 100ns pulse on the output of the transmitter (labeled as $\mathbf{V_{in}}$ in the figures) (as might occur with a 5 Mbps transmission)



Part a) Derive an equation for $\mathbf{V_{out}}$ as a function of time. Show your work and plug in values for R and C.

Part b) Make a rough sketch of $\mathbf{V_{out}}$ as a function of time.

Part c) There is a serious problem with this channel. How do you fix the problem?

(30) Question 4. Consider a system with two LCD message displays in the context of a preemptive thread scheduler with blocking semaphores. To display a message, the OS can call either **`LCD1_OutString`** or **`LCD2_OutString`** passing it an ASCII string. These routines have critical sections but must run with interrupts enabled. The foreground threads will not call **`LCD1_OutString`** or **`LCD2_OutString`** directly; rather, the threads call a generic OS routine **`OS_Display`**. If a LCD is free the OS passes the message to the free LCD. If both LCDs are busy, the thread will block. There are many threads that wish to display messages, and the threads do not care or know onto which LCD their message will be displayed. You are given the **`LCD1_OutString`** or **`LCD2_OutString`** routines, the OS and the blocking semaphores with the following prototypes.

```
void LCD1_OutString(char *string); // up to 20ms to complete
void LCD2_OutString(char *string); // up to 20ms to complete
int OS_InitSemaphore(Sema4Type *semaPt, short value);
void OS_Wait(Sema4Type *semaPt);   // counting semaphore
void OS_Signal(Sema4Type *semaPt); // counting semaphore
```

Part a) List the semaphores and private global variables needed for your solution. For each semaphore define what it means and what initial value it should have. Give the meaning and initial values for any private global variables you need. The threads will not directly access these semaphores or variables.

Part b) Write the generic OS display routine that the foreground threads will call (you may not disable interrupts or call any other functions other than the five functions shown above)

```
void OS_Display(char *string){
```