Jonathan W. Valvano

First Name: _____ Last Name:_____

April 17, 2009, 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.

(15) Question 1. A CAN system with 3 nodes has a baud rate of 50,000 bits/sec. The message protocol has frame sizes of exactly 4 data bytes per frame. The time between frame transmissions varies from 10 ms to 50 ms, with an average of 20 ms. This means each node starts a CAN transmission on average every 20 ms. The CAN uses 11-bit IDs.

Part a) This system will work. I.e., this CAN baud rate can support the traffic on this network. However, determine the slowest baud rate that could support this traffic. You may ignore stuff bits. Show your work.

Part b) Assuming a baud rate of 50,000 bits/sec, what is the actual average data bandwidth on this network. Show your work.

Part c) What are stuff bits and why are they used?

(20) Question 2. Consider a 256-point FFT calculated on 12-bit ADC data sampled at 10 Hz. I.e., the input data are numbers from 0 to 4095 collected every 100 ms. The ADC range is 0 to 10 V. This means if the input is 5V, the ADC data will be 2048.

Input	x[2i] = sampled data for i = 0 to 255	input is real
	x[2i+1] =0	
Output	X[2k] = real part of FFT for k = 0 to 255	output is complex
	X[2k+1] = imaginary part of FFT	

Part a) What frequency does k=64 represent?

Part b) For k=64, what does it mean if the real part (X[128]) equals the imaginary part (X[129])?

Part c) What is the relationship between the FFT output at k, and the FFT output at 256-k, assuming k is not 0 or 128?

Part d) Assume the input signal is a constant 5V, what will be the real part and imaginary part of the FFT output as calculated by the **`fft()`** used in Lab 4, were **`nn`** is 256? There is no noise. I.e., every input data sample is 2048.

(15) Question 3. Consider the following 16-bit FIFO implementation.

```
#define FIFOSIZE 100
short static *PutPt;  // Pointer of where to put next
short static *GetPt;  // Pointer of where to get next
short static Fifo[FIFOSIZE];  // statically allocated data
void Fifo_Init(void){  PutPt = GetPt = &Fifo[0];}
```

```
int Fifo_Put(short data){
short *tempPt;
  tempPt = PutPt;
  tempPt++;
  if(tempPt==&Fifo[FIFOSIZE]){/
    tempPt = &Fifo[0];
  }
  if(tempPt == GetPt ){
    return(1);  // Failed, full
  }
  else{
    *PutPt = data;  // save
    PutPt = tempPt; // Success
    return(0);
  }
}
```

```
int Fifo_Get(short *datapt){
  if(PutPt == GetPt ){
    return(1);     // Empty
  }
  else{
    *datapt = *GetPt;  // store
    GetPt++;
    if(GetPt==&Fifo[FIFOSIZE]){
      GetPt = &Fifo[0];
    }
    return(0);
  }
}
```

Write a C function that returns a true (nonzero) if the FIFO is more than 75% full, and a false (0) if the FIFO is less than or equal to 75% full. You will be graded on style and effectiveness. You can not modify **`Fifo_Put`** or **`Fifo_Get`**.

(25) Question 4. Consider a system that employs a preemptive real-time OS like Lab 5. There are multiple threads that need to update a shared LCD display. Consider this example with two foreground threads (**thread1 thread2**) and one background thread (**isr**) that all output to a 3-line LCD. **Free** is a global variable, initialized to 1.

```
void thread1(void){
unsigned short data;
  init1();
  for(;;){
    data = calc1();
    Display(1,data);
  }
}
```

```
void thread2(void){
unsigned short data;
  init2();
  for(;;){
    data = calc2();
    Display(2,data);
  }
}
```

```
interrupt 15
void isr(void){
unsigned short data;
  data = calc3();
  Display(3,data);
}
```

The first parameter of **Display** is the line number, and the second parameter is a 16-bit number. You may call the two **LCD_GoTo LCD_OutDec** functions without writing them. Your **Display** function will effectively perform the following (this program has a critical section.)

```
unsigned char Free=1;
void Display(int line, unsigned short num){
  if(Free){Free=0; LCD_GoTo(line,1); LCD_OutDec(num); Free = 1;}
}
```

Rewrite this **Display** function to remove the critical sections. You CAN NOT disable interrupts at all. You should not introduce new critical sections. You CAN NOT allow threads to block or spin. If the LCD is busy, then the output is simply skipped. You may not change the thread code or the **Display** function prototype. Basically you will add software to this existing **Display** function, but not **thread1**, **thread2** or **isr**.

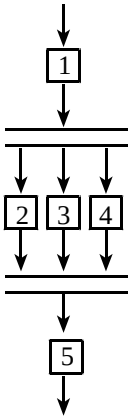
(25) Question 5. Implement the following fork and join synchronization. Each of the boxes in the figure represent one of 5 predefined functions (**fun1**, **fun2**, **fun3**, **fun4**, and **fun5**), which you do not need to implement. You will however define six different code segments so that **fun2 fun3** and **fun4** run simultaneously. There is a master thread that exists before the fork and after the join. You may assume initially the master thread is active at priority level 3, but the two slave threads have not yet been added.

```
void master(void){
  for(;;){
    fun1();
    segment1A
    fun2();
    segment1B
    fun5();
  }
}
```

```
void slave3(void){
  segment3A
  fun3();
  segment3B
}
```

```
void slave4(void){
  segment4A
  fun4();
  segment4B
}
```

Assume the OS is preemptive priority scheduler with blocking semaphores like the solution to Lab 5. In this question, you can call any of the following OS commands without implementing the function. The fun functions need about 50 bytes of stack space. You can create semaphores by defining them as variables with **SemaType**.




```
// ******** OS_InitSemaphore ************
// initialize semaphore
// input:  pointer to a semaphore
// output: none
void OS_InitSemaphore(Sema4Type *semaPt, short value);
// ******** OS_Wait ************
// decrement semaphore and block if less than zero
// input:  pointer to a counting semaphore
// output: none
void OS_Wait(Sema4Type *semaPt);
// ******** OS_Signal ************
// increment semaphore, wakeup blocked thread if appropriate
// input:  pointer to a counting semaphore
// output: none
void OS_Signal(Sema4Type *semaPt);
//******** OS_AddThread ***************
// add a foregound thread to the scheduler
// Inputs: pointer to a void/void foreground function
//         number of bytes allocated for its stack
//         priority (0 is highest)
// Outputs: 1 if successful, 0 if this thread can not be added
short OS_AddThread(void(*fp)(void),
   unsigned short stackSize, short priority);
// ******** OS_Sleep ************
// place this thread into a dormant state
// input:  number of ms to sleep
// output: none
void OS_Sleep(unsigned short sleepTime);
// ******** OS_Kill ************
// kill the currently running thread, release its TCB memory
// input:  none
// output: none
void OS_Kill(void);
```

Part 0) List the semaphore(s) needed

Part a) Give the C code labeled **`segment1A`** for the master to execute between **`fun1`** and **`fun2`**

Part b) Give the C code labeled **`segment1B`** for the master to execute between **`fun2`** and **`fun5`**

Part c) Give the C code labeled **`segment3A`** for the slave3 to execute before **`fun3`**

Part d) Give the C code labeled **`segment3B`** for the slave3 to execute after **`fun3`**

Part e) Give the C code labeled **`segment4A`** for the slave4 to execute before **`fun4`**

Part f) Give the C code labeled **`segment4B`** for the slave4 to execute after **`fun4`**