

MEMO TO: ECE 4175 students

February 8, 2008

FROM: John Peatman

SUBJECT: First quiz on Wednesday, February 13th Open book, open notes.

Cody Planteen, Andrew Ray, and Bren Whitfield will hold a review session in room C241 (our classroom) on Monday from 7:00-9:00 PM. Also, review the first quiz from last semester, available from the piclab website. While the course is drastically different, at least you may get an idea of the kinds of questions I ask.

Both the 9:00 class and the 10:00 class will be taking the same quiz. If you take the quiz at 9:00, do not talk about the quiz or otherwise divulge information about the quiz as you leave. If you are in the 10:00 class, do not press the 9:00 students with questions about the quiz. Thanks!

I plan to begin each quiz at 3 minutes after the hour (i.e., 9:03 and 10:03) and end at 3 minutes before the hour (i.e., 9:57 and 10:57) unless someone comes to me no later than Monday to raise an objection. OK?

1. Be sure to bring your source files for all of the projects. I will assume that you are familiar with the kinds of things that arise in all of the projects up through Project Five. You will need to print out the source files for each of your team, regardless of who has the folder that you turned in. Also bring the qwik.lst file for Project Four.
2. Review David Bauer's "Hints on doing well in ECE 4175" posted on the piclab website.
3. Read through Chapters 1, 2, 3, 4, 5, 6, 7, Sections 9.1 to 9.6, and Appendix A2.
4. I do plan to ask you questions that have to do with the debugging that you have done, or should have done, in the lab.
5. You should be able to write C code to do the kinds of things you have had to write code for on the projects.
6. I will not ask you to write assembly code. However, given the information in Appendix A2, you should be able to determine the execution time of a short segment of assembly code that the C compiler has generated into the qwik.lst file.
7. For Project One, you used the pushbutton in a rudimentary way, taking action as long as the pushbutton was pressed. Subsequently, you took action in response to the leading edge of a pushbutton press.
8. All of our code examples use the brownout-reset functionality of Figure 4-3 assuming that the chip powers up into our code, to help ensure reliable startup. When we use the chip via QwikBug, this dealing with the brownout reset really doesn't serve this function since QwikBug has already turned off the brownout reset.
9. We have used the Sleep state of the CPU, waking up with either the watchdog timer's timeout or with Timer1's interrupt.
10. We have talked about the internal clock and how it can be controlled. Refer to Figure 2-3. We also talked about the role of clock rate versus the average current draw to execute a fixed number of instructions before returning to sleep for the remainder of the loop time.
11. We have also talked about the watchdog timer and its possible control with a Configuration option and the control we have actually used with a SWDTEN bit in the WDTCON register. Refer to Figure 2-6.
12. We have discussed the debouncing of a mechanical switch. Refer to Figure 2-7.
13. You should understand the rationale for the device connections of Figure 3-2 for the parts we have used.
14. Know how to set a bit, clear a bit, toggle a bit, and test a bit, for both a bit in a SFR (Special Function Register) and a flag bit defined in RAM (e.g., LCDFLAG).

15. Understand when and how to recast a variable from a char to an int, etc, in a mathematical expression.
16. The 4321 chip uses its Serial Peripheral Interface to send ASCII-coded characters to the starburst LCD. You should understand the role of the 4321's RD5 pin in this process as well as how the SPI works for these transfers. With our simple message protocol, we always send nine characters for an eight-character display. Why?
17. For Project Two, you used each push of the pushbutton to double the step rate of the stepper.
18. For Project Three, you used a rate multiplier scheme to increment the step rate from 1 s/s up to 62 s/s. You also measured how long the CPU remains awake during each loop time doing useful work.
19. For Project Four, you used high-priority interrupts to produce 1.6 ms tick times and then used these interrupts to reuse the rate multiplier scheme of Project Three to produce step rates from 10 s/s up to 620 s/s in increments of 10 s/s.
20. For Project Five, you used the output of the pot to set the step rate, no longer using the rate multiplier scheme. Now you translated the pot value into a step rate display and a number of counts of Timer3 between interrupts - and thereby the interval between steps.
21. Timer0 necessarily requires us to use its buffer register, TMR0H, if we want to read out the content of the timer, or write to it. What is the intent of the buffer register, in general? What is the effect of the buffer register if we stop the counter while we read from or write to the counter?
22. Consider the TXascii macro called from within the Measure.c file. What is the effect of waiting for the setting of the TRMT bit before exiting and going to sleep?
23. We have looked at a bunch of ways to convert a NUMBER of a given size into UNITS, TENS, HUNDREDS, etc. ASCII characters. You should have a pretty good idea of what helps to minimize the cycle count of an algorithm to do this.
24. On the projects, we have dealt with the LED, the pushbutton, the pot and its use with the ADC as both an 8-bit input and a 10-bit input device, the LCD, as well as individual output pins (e.g., RB0). None of these interactions should hold any mysteries for you.
25. Timer1 and Timer3 offer an option not available for Timer0 of buffering the upper byte of the counter. Whereas the circuitry for Timer0 forces the use of this buffering, we chose not to use either one with Timer1 or Timer3. What went into this choice, given how we used Timer1 and Timer3? When this buffering feature is used, what is it good for?
26. If the chip is asleep when the watchdog timer times out, the CPU just picks up with the execution of the next instruction after the Sleep macro. If the chip is awake when the watchdog timer times out, the CPU is reset and starts over from address 0x0000.
27. Timer1 and Timer3 generate interrupts whether or not the chip is asleep. The CPU awakens, vectors to the respective interrupt service routine, executes it and then returns from the interrupt, back to the next instruction in the code that was executing at the time of the interrupt, reenabling the interrupt for that timer in the process.

I added in a variable amount to the grades,
ranging from +5 for grades above 90 to +15 for grades below 71.

1. Assume that we have defined a flag variable called **READY** whose value is either non-zero (i.e., 1) or zero. Show the C code below to test **READY** and repeatedly execute
[15%] `<a sequence of instructions>`
until **READY** is non-zero. Presumably the sequence of instructions will eventually set **READY**, but you can assume that **READY** is initially zero.

```
while(!READY)           or           while(READY == 0)
{
    <a sequence of instructions>
}
```

2. Assume you have just added a new function, **NewFunction**, to code that previously worked correctly. Now you find that **NewFunction** causes the MCU to hang and never return to the main loop from
[10%] which it was called. Describe how you might use QwikBug to discern what is going wrong. Set a breakpoint at **NewFunction**. Single step through **NewFunction** to see where it hangs or loops. Set a breakpoint on a line that only executes when it breaks out of the loop. Then run to the breakpoint, to see if it ever breaks out. If not, set up a watch variable for any variable that affects the exit from the loop. Set a breakpoint within the loop and run to the breakpoint repeatedly to see what is happening to the watch variable from loop to loop.

3. The **TXascii** macro is used in the Measure.c template program of Chapter 6 to send a byte from the MCU to the PC. The **Send** function used there includes six successive executions of this **TXascii**
[9%] macro. If the **TXascii** macro definition did not include the statement

```
while(!TXSTAbits.TRMT)
```

then what would the effect be upon what would be displayed in QwikBug's console window? Explain your answer.
The key to this problem is Figure 6-4, which describes the role of TXREG, TSR, and the TRMT flag. When a byte is written to TXREG while TSR is empty, the byte is immediately dropped into TSR. Subsequent bytes written to TXREG can overwrite each other until the transfer from TSR has been completed (without messing with the content of TSR). Of course, shortly after the exit from the final Send occurs, the CPU goes to sleep, killing the reception of any byte in the process of being sent. For the Measure.c program, the first character sent is a carriage return code (which produces no display). Because of the delays caused by the computations, the CPU remains awake for some multiple of the half millisecond interval needed to send a single character. I checked and found that three displayable characters showed up in the Console window. I tried another experiment, sending the six digits 1,2,3,4,5,6 but then remaining awake thereafter. The 1 and the 6 showed up in the Console window.

4. The rate multiplier scheme used on Projects 3 and 4 produced an action (i.e., making the stepper motor take a step) at a rate determined by how often it was called and the value of **RATE** when it is called.
[16%] Assume that the rate multiplier code is executed each time around a mainline loop that has been set up to take ten milliseconds (using the Timer1 oscillator) and that the **RATE** parameter can take on any value between 1 and 120. When the subtraction of **RATE** from **ACCUM** produces a negative result, add 120 to **ACCUM** and take action.
 - a. What is the maximum number of times the action is taken per second, and what value of **RATE** produces this rate? Maximum = 100/second **RATE** = 120

 - b. What is the minimum number of times the action is taken per second, and what value of **RATE** produces this rate? Minimum = 0.833/second **RATE** = 1

5. Consider the code below, taken from the qwik.lst file for a modified version of the T3.c template program.

[25%]

```

// For high priority interrupts:
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
3
2   0008   EF24   GOTO   HiPriISR   _asm GOTO HiPriISR _endasm
   000A   F000
   .
   .
2   0048   CFDA   MOVFF   FSR2H,PREINC1   void HiPriISR()
   004A   FFE4
1   004C   52E6   MOVF   POSTINC1,F
                                   {
   004E   90B1   BCF   T3CON,0           T3CONbits.TMR3ON = 0;
1   0050   5007   MOVF   STEPCNTL,W       TMR3L = STEPCNT;
1   0052   6EB2   MOVWF  TMR3L
1   0054   5008   MOVF   STEPCNTH,W       TMR3H = STEPCNT/256;
1   0056   6EB3   MOVWF  TMR3H
   0058   80B1   BSF   T3CON,0           T3CONbits.TMR3ON = 1;
   005A   92A1   BCF   PIR2,1           PIR2bits.TMR3IF = 0;
   005C   8283   BSF   PORTD,1          PORTDbits.RD1 = 1;
   005E   9283   BCF   PORTD,1          PORTDbits.RD1 = 0;
   0060   7081   BTG   PORTB,0          PORTBbits.RB0 ^= 1;
   0062   52E5   MOVF   POSTDEC1,F      }
   0064   CFE5   MOVFF  POSTDEC1,FSR2H
   0066   FFDA
   0068   0011   RETFIE FAST

```

Consider the possible consequences of *not* stopping the counting of Timer3 *and* also of *reloading* (rather than adding into) **TMR3H:TMR3L**. Assume that it takes a maximum of three cycles of the CPU after Timer3 rolls over and before the CPU has fetched the instruction at the high-priority interrupt vector address 0x0008 and is ready to execute it.

- Cross out the instruction above that stops the counting of Timer3 and the instruction that starts the counting of Timer3 again.
- How many CPU clock cycles can occur after Timer3 rolls over and before the above code will have reloaded **TMR3H:TMR3L** with the content of **STPCNTH:STPCNTL**? In answering this, to the left of each line above that is executed until **TMR3H:TMR3L** has been reloaded, write the number of cycles executed for that line. Cycles = 12
- How many microseconds will pass after Timer3 rolls over until Timer3 is incremented again by the 32.768 kHz watch crystal? Microseconds = 30.5
- Given this, will reloading Timer3 rather than stopping Timer3 and adding into it always work? Explain your answer briefly.
Yes; we are assured that Timer3 will not be reloading at the same time that it is counting.

6. The T2.c template program of Chapter 5 uses the pushbutton to increment a number displayed on the LCD, one increment per press of the pushbutton. For this to work correctly, the maximum contact bounce time of the pushbutton must satisfy what requirement? Be specific.
[10%] It must be less than the loop time, which is 16 ms for T2.c.

7. Assume that in one of your projects you want to display the following message on the LCD display each time you detect a new condition has been fulfilled and before displaying the new data.

[15%]

NEW DATA

Show the code you would use to do this, making use of the **LoadLCDSTRING** macro.
LoadLCDSTRING("NEW DATA ");
Display();

MEMO TO: ECE 4175 students

March 14, 2008

FROM: John Peatman

SUBJECT: Second quiz on Wednesday, March 26th Open book, open notes.

Cody Planteen, Andrew Ray, and Bren Whitfield will hold a review session in room C241 (our classroom) on Monday from 7:00-9:00 PM. Also, review the quizzes and final exam from last semester, available from the piclab website.

Both the 9:00 class and the 10:00 class will be taking the same quiz. If you take the quiz at 9:00, do not talk about the quiz or otherwise divulge information about the quiz as you leave. If you are in the 10:00 class, do not press the 9:00 students with questions about the quiz. Thanks!

I plan to begin each quiz at 3 minutes after the hour (i.e., 9:03 and 10:03) and end at 3 minutes before the hour (i.e., 9:57 and 10:57) unless someone comes to me no later than Monday to raise an objection. OK?

1. Be sure to bring your source files for all of the projects. I will assume that you are familiar with the kinds of things that arise in all of the projects up through Project Ten. You will need to print out the source files for each of your team, regardless of who has the folder that you turned in.
2. Read through the remainder of the book, except for Chapter 14 on EEPROM and Chapter 16 on the LCD and the appendices A1, A3, and A4. Be sure to bring the book in either Lulu or Tech-printed form.
3. I will not ask you to write assembly code. However, given the information in Appendix A2, you should be able to determine the execution time of a short segment of assembly code that the C compiler has generated into the qwik.lst file. You should also understand the functioning of short assembly code sequences. I will avoid the cryptic stuff involving internal function calls like the math functions and the variables that are introduced by the compiler to deal with them.
4. For Project Six, you decreased the INTOSC frequency from 4 MHz to 1 MHz while sending bytes to the PC. What was this attempting to do and what was the result?
5. For Project Seven, you used the RPG to scroll an eight-character window over the sixteen-hex-character silicon serial number.
6. For Project Eight, you used a rate-sensitive RPG scheme to change the stepping rate of the stepper motor.
7. For Project Nine, you compared different implementations of a linear equation, looking at the qwik.lst file to discern how the c18 compiler is handling each implementation.
8. For Project Ten, you calibrated the HLVD feature of the PIC chip. You should go back and read the data sheet on this feature and understand what it is for and how it works.
9. Chapter 12 deals with interrupts. You should understand how to deal with an interrupt source with either a high or a low priority interrupt and how the structuring of the interrupt service routine can affect the code generated by the C compiler. The use of a "critical region" of code is a topic that should have been in Chapter 12. We discussed this last Monday in conjunction with the Timer1Check function where we wanted to take action within microseconds after an event occurred, so we disabled interrupts to make sure that code was not postponed momentarily by an intervening interrupt.
10. Chapter 15 deals with Dallas Semiconductor's 1-wire interface, used by the silicon serial number part and also by Dallas Semi's iButtons. We discussed the implementation of open-drain outputs from the MCU to communicate with these devices. We also discussed how a multiple-byte protocol that is specific to a device like the DS2415 time chip can be used to read from it or write to it.

- OVER -

11. Chapter 10 deals with two RPGs. The detented unit on the Qwik&Low board operates best with interrupt servicing. The Bourns unit available to you for the design project makes good use of a polling routine. You should understand this distinction and why each RPG is handled in its own way.
12. The latter part of Chapter 9 deals with a linearization process and how its careful implementation with integer arithmetic by the C compiler can drastically simplify its execution time.
13. Chapter 17 expands on the SPI concepts introduced in Chapter 5 to deal with the LCD. We discussed the choice of CKP and CKE to match the MCU to the needs of a device to which we want to send information. We also discussed the choice of CKP, CKE and SMP to read in data reliably from a device. Note that some devices want to receive a clock edge before the first data bit is read while others expect that first bit to be read before the first clock edge. And note the choice of parameters for either case. Some devices expect the MCU to carry on simultaneous input and output transfers using the MOSI/MISO scheme discussed in conjunction with the modes of Figure 17-11.
14. Chapter 8 explains how a bipolar stepper motor works, with a multiple-pole, magnetized rotor and a multiple-pole stator designed so that the appropriate energization of its two windings moves the poles CW or CCW a step at a time. It also explains how current-sensing resistors are used to set the current in the windings rather than having the current set by the motor's supply voltage.
15. Chapter 11 introduces macros for starting and stopping a pulse so that a scope can be used to measure time intervals. We have done the same thing on multiple occasions by just setting and clearing the output to a pin that can then be monitored by the scope. We also used this scheme in conjunction with the scope's Autostore mode to check worst-case time intervals.
16. The latter part of Chapter 13 introduces the cycle-counting Start, Stop and Send functions first used in the Measure.c template program of Chapter 6. (We haven't dealt with the first part of Chapter 13, wherein the INTOSC internal oscillator of the MCU is calibrated against the much more accurate 32,768 Hz, 50 ppm TMR1 oscillator. I won't address this calibration on the quiz.)
17. We have discussed (Chapter 16 taken from my previous book and) the I²C bus. Its use is discouraged with the Qwik&Low board because the same MCU pins that control the I²C bus are already in use with the SPI bus control of the LCD. You should understand how it frames multiple-byte transfers and how it uses the first byte to address a specific chip. A multiple-byte message is device specific, requiring reference to that device's data sheet for information on how to send information to, and receive information from, the device.

High	113
Upper quarter	87
Median	68
Lower quarter	58
Low	35

I added 10 points to everyone's grade.

1. Consider the following assembly code generated into a qwik.lst file:

[10%] `014C 5039 MOVF RPG,W
014E E103 BNZ L013
0150 0E0F MOVLW 0x0F
0152 6E39 MOVWF RPG
0154 D001 BRA L012
0156 0639 L013 DECF RPG,F
0158 9084 L012 BCF PORTE,0`

Describe what this code sequence does to the RAM variable, RPG.

If RPG is initially equal to zero, it is reloaded with 0x0F (i.e., 15). Otherwise it is decremented.

2. For Project Six, you decreased the INTOSC frequency from 4 MHz to 1 MHz while sending five bytes to the PC.

- [15%] a. If you had left INTOSC = 4 MHz, how long would it have taken to transmit the five bytes? Each byte, including its start and stop bits, takes 10/19200 seconds, or 10000000/19200 microseconds. This computes to 521 microseconds, or about 0.5 millisecond. Five bytes thus take about 2.5 ms.
- b. With INTOSC = 1 MHz, how long does it take to transmit the five bytes? The same 2.5 ms.
- c. Why does decreasing INTOSC from 4 MHz to 1 MHz while sending these five bytes to the PC decrease the average current draw on the coin cell? Because in either case there is a 2.5 ms time interval during which the chip draws a current determined by the clock rate. That current is about 1 mA for INTOSC = 4 MHz and about 0.5 mA for INTOSC = 1 MHz. (See Figure 2-4)

3. We have discussed how a *critical region* of code must not be permitted to be interrupted. That is, while that section of code is being executed, it must be allowed to proceed instruction by instruction with no intervening interrupt permitted to break the lock-step of the instruction sequence.

- [25%] a. For an application that makes use of no interrupt sources other than the single interrupt source, INT2 (the interrupt input from the RPG), this can be achieved by inserting the following line of C code before the critical region of code. Given that every interrupt has both a local enable control bit and a global enable control bit, show the code that will *locally* disable this single interrupt source.
INTCON3bits.INT2IE = 0;
- b. Show the line of C code to *locally* reenable INT2 interrupts at the completion of the critical region of code.
INTCON3bits.INT2IE = 1;
- c. If an RPG is incremented during this time that INT2 interrupts are momentarily disabled, will the CPU account for that increment of the RPG, or is it lost forever? The CPU will account for that increment of the RPG when INT2 interrupts are reenabled.
- d. For a different application that makes use of INT2 interrupts and also Timer3 overflow interrupts, show the line that will *globally* disable all possible interrupts by clearing a single control bit.
INTCONbits.GIEH = 0;
- e. Finally, show the line of C code that *globally* reenables all interrupts at the completion of the critical region of code.
INTCONbits.GIEH = 1;
- f. If a Timer3 overflow occurs during the time that all interrupts are globally disabled, will the CPU account for this overflow of Timer3, or is it lost forever? Using Figure 7-5, explain your answer. The TMR3IF flag is set by the overflow. Either the clearing of the local interrupt enable bit, TMR3IE, or the clearing of the global interrupt enable bit, GIEH, will block TMR3IF from reaching the CPU. When GIEH is set again (along with TMR3IE already being set), the signal gets through to the CPU and produces the interrupt whose interrupt service routine will then account for the overflow.

4. Serial interfaces that reach from the MCU to more than one peripheral device have two problems to be considered here:

- [20%]
- How does a specific device know that it is the one being addressed?
 - How are the beginning and the ending of multiple-byte messages marked?

- a. For the serial peripheral interface, how are these two problems handled?

The MCU designates the specific device that is to receive and act upon the multiple-byte message by making that device's chip enable pin active. Usually the chip enable pin is active low, so this means that the MCU pin that drives that chip enable pin has a zero written to it.

The MCU precedes the sending of the multiple-byte message with the activation of the specific device's chip enable pin. It terminates the message by deactivating that device's chip enable pin. Furthermore, because it is enabled, the SSPIF flag will be set after each received byte.

- b. For the I²C interface (a.k.a. the SMBus), how are these two problems handled?

The MCU indicates that a multiple-byte message is about to begin by transitioning the SDA and SCL lines through the I²C START condition. That is, the MCU first pulls the SDA line low and then the SCL line low (from their initial state of both being high). The end of the multiple-byte message is indicated when the MCU causes the SCL line to go high followed by the SDA line going high – a change that is called the I²C STOP condition.

Each device on the I²C bus has its own unique seven-bit address. Seven bits of the first byte of the multiple-byte message hold this seven-bit address. The addressed device responds to the rest of the message. All other devices on the bus ignore the rest of the message.

5. Consider a bipolar stepper motor whose rotor consists of just a single North pole and a single South pole. The stator has two windings, designated as A and B. Full stepping is carried out by the following sequence of winding excitations, where A and its complement represent current in one direction or the other through winding A.
- [10%]

... A B $\overline{\overline{A}}$ $\overline{\overline{B}}$ A B $\overline{\overline{A}}$ $\overline{\overline{B}}$...

- a. What is the angle of rotation produced by each step?
90°
- b. How many steps per revolution does this sequence produce?
4

6. Suppose that a transducer generates an output M ranging from 0 to 2047. The physical parameter N that is being transduced is related to M by the equation

[20%]
$$N = 4.321M + 342$$

- a. Reexpress this relationship to avoid floating-point math as well as division, so as to minimize computation time while maintaining 1 part in 2048 resolution. Multiply 4.321M by X/X, where X is a power of 256 so that the division by X can be achieved by moving the numerator to the right by one byte position for 256 or two byte positions for 256x256. To maintain 1 part in 2048 resolution, X = 256 is not sufficient whereas X = 256x256 = 65536 is. So

$$N = (65536/65536) \times 4.321 \times M + 342$$
or

$$N = 283181M/65536 + 342$$
- b. Express the computation in C. You can assume that M has been defined as a four-byte "long" variable.

$$N = ((283181 * M) >> 16) + 342;$$
If the C compiler implements this with bit shifts rather than byte moves, use the following instead.

$$N = (((283181 * M) >> 8) >> 8) + 342;$$
Michael Pribble and John Hamilton had this answer. Many students used

$$N = ((8849 * M) >> 11);$$
to get the 1 part in 2048 resolution. However this doesn't minimize the computation.

1. [] I have completed the CIOS course evaluation.
 [5%] [] I have not completed the CIOS course evaluation.

2. The QwikLow board that we have used in previous semesters included a three-digit seven-segment numeric LCD that had to be updated every ten milliseconds. The refreshing function might have employed the [15%] following macro to write the seven-segment coding for each digit out to three serially-connected 74HC595 shift registers via the Serial Peripheral Interface:

```
#define Shift(out)  SSPIF = 0; SSPBUF = out; while(!SSPIF)
Consider the following code and assume that the SPI has been set up to clock the SCK line at the maximum rate with a CPU clock rate of 1 MHz:
```

```
Shift(OUTH);          SUM          // Load three 74HC595 shift registers
Shift(OUTT);          SUM          // with seven-segment coding of
Shift(OUTU);          SUM          // hundreds, tens, and units digits
PORTCbits.RC0 = 0;    1           // Toggle the 74HC595 latch clocks
PORTCbits.RC0 = 1;    1
```

- a. Above each instruction in the macro, list your estimate of the number of microseconds taken to execute that instruction. Use one of the qwik.lst files (handed out with this final exam, in class, or obtained for one of the design projects) to see how instructions equivalent to those above are executed.

1 2 8-11 SUM = 11-14

```
#define Shift(out)  SSPIF = 0; SSPBUF = out; while(!SSPIF)
```

- b. To the right of each of the five lines above, list your estimate of the number of microseconds taken to execute that line.
 c. If the MCU draws 1.000 mA while executing instructions with a clock rate of 1 MHz, then determine the average current draw of the MCU contributed by this five-line sequence when it is executed every ten milliseconds, assuming that the MCU sleeps when it is done doing useful work each loop time. Show all your work clearly.

$$I_{\text{awake}} = 1000\mu\text{A} \times ((3 \times \text{SUM}) + 2) / 10000 = 0.1((3 \times \text{SUM}) + 2)\mu\text{A} \approx 5\mu\text{A}$$

3. Consider the interrupt service routine listed below for dealing with the magnetic card reader with the MCU [15%] connections discussed in class.

```

/***** HiPriISR interrupt service routine *****/
void HiPriISR()
{
    INTCON3bits.INT1IF = 0;          // Clear interrupt flag
    if (!PORTBbits.RB3)             // Read in active-low data
    {                                 // so RB3 == 0 represents bit = 1
1 →      MAG_BYTE |= 0b00100000;     // 6th bit position; will be shifted into 5th
        if (!MAG_STATE)             // This is the first 1 read from card
        {
3 →      MAG_CNT = 40;               // Initialize counter of no more than 40 bytes
          MAG_STATE = 1;            // Set flag to signal presence of data bytes
          MAG_BYTE = 0b00100000;    // Initial value
          MAG_BITCNT = 5;
        }
    }
    if (MAG_STATE)                  // Handle data bytes
    {
        MAG_BYTE >>= 1;            // Shift bits into position
        if (--MAG_BITCNT == 0)      // Act further only for a completed digit
        {
2 →      MAG_BITCNT = 5;            // Reset for next digit
          MAG_BYTE |= 0x30;         // Convert to ASCII
          if (MAG_BYTE == '?')     // End sentinel?
          {
              MAG_STATE = 2;       // Done with data
              TXascii(MAG_BYTE);    // Carriage return, line feed
              TXascii('\n');
          }
          else if (MAG_STATE == 2) // LRC character after end sentinel?
          {
              MAG_STATE = 0;
          }
          else                      // A data byte
          {
              if (MAG_CNT)         // Send no more than 40 characters
              {
                  TXascii(MAG_BYTE);
                  --MAG_CNT;       // Decrement scale of 40 counter
                  MAG_BYTE = 0;    // Reset byte for next data byte
              }
          }
        }
    }
}
}

```

- a. Mark with 1 → the place in the above code where you would insert a line of program code that will toggle the least-significant bit of a char variable, **PARITYFLAG**, if bit 3 of **PORTB** is read as a zero. Show that line of program code here:
- ```
PARITYFLAG ^= 0x01;
```
- b. Mark with 2 → the place in the above code where you would turn on a LED by setting bit 4 of **PORTD** (leaving the other bits of **PORTD** unchanged) if **PARITYFLAG** equals zero during the interrupt call when the fifth of each group of five bits occurs. Show the code here:
- ```
if (!PARITYFLAG)
{
PORTDbits.RD4 = 1;
}
```
- c. Mark with 3 → the place in the above code where you would turn off the LED by clearing bit 4 of **PORTD** (leaving the other bits of **PORTD** unchanged) when the swipe of a new card is begun, as detected by the first time bit 3 of **PORTB** is read as a zero. Show the code here:
- ```
PORTDbits.RD4 = 0;
```

4. A couple of groups in working on their design project had a part that expected bytes to be sent and received over the SPI interface least-significant bit first whereas the MCU's SPI module sends and receives bytes most-significant bit first. To handle this, it is convenient to have a **Reverse** function that will reverse the bits of a char global variable called **IO**. For this problem, you are to write the code for the **Reverse** function using the following algorithm that employs another char global variable, **TEMP**. To carry out this reversal, execute the following procedure seven times using a "for" loop:

Copy the least-significant bit of **IO** into the least-significant bit of **TEMP**.  
Shift **IO** right (one place)  
Shift **TEMP** left one place

When this has been carried out, copy **TEMP** into **IO** to complete the function.

```
void Reverse()
{
for (i=0; i<7; I++)
{
TEMP |= (IO & 0x01);
IO >>= 1;
TEMP <<= 1;
}
TEMP |= (IO & 0x01);
IO = TEMP;
}
```

5. Consider the attached page showing a 1/2-multiplexed LCD display that might be used to display a two-digit, 7-segment-coded decimal number directly from a microcontroller. The two 100 kΩ resistors used in the drive for each of the two COMi lines provide a way to drive the line to 3 V (the MCU supply voltage), 1.5 V, or 0 V. The backplane waveforms for COM1 and COM2 are shown in the middle of the page. Below these two waveforms are shown the four frontplane waveforms to be applied to SEG1. The bottom waveform is intended to turn on both pixel 11 and pixel 12, the leftmost two pixels illustrated. The other three turn one, or the other, or both of these same pixels off.
- a. The LCD itself draws essentially no current. What is the current draw on the coin cell due to the COM1 resistor circuit when its voltage is 1.5 V?
- $$I = 3V / 0.2M\Omega = 15 \mu A$$
- b. Can you drive COM1 to +3V with no current draw (due to this drive) at all? Explain your answer.  
Yes; set RC0 = RC1 = 1
- c. Can you drive COM1 to 0V with no current draw at all? Explain your answer.  
Yes; set RC0 = RC1 = 0
- d. As the COM1 and COM2 waveforms are continuously driven with a 50 Hz frame frequency, determine the average current draw on the coin cell due to the generation of these waveforms (ignoring the MCU's current draw)? Show your work clearly.
- $$\begin{aligned} \text{COM1 current} &= \frac{1}{2} \times 15\mu A = 7.5 \mu A \\ \text{COM2 current} &= \frac{1}{2} \times 15\mu A = 7.5 \mu A \\ \text{COM1 + COM2 current} &= 15 \mu A \end{aligned}$$
- e. What is the interval needed between interrupts so that the corresponding interrupt service routine can be used to generate these waveforms?  
5 ms
- f. Showing all work, determine the RMS voltage between the frontplane and the backplane for pixel 12 when both pixel 11 and pixel 12 are off.
- $$\text{RMS} = \sqrt{((1.5)^2 + (0)^2 + (1.5)^2 + (0)^2) / 4} = 1.06V$$

- g. Repeat part (e) for pixel 12 when pixel 12 is off but pixel 11 is on.  

$$\text{RMS} = \sqrt{((1.5)^2 + (0)^2 + (1.5)^2 + (0)^2) / 4} = 1.06\text{V}$$
- h. Repeat for pixel 12 when pixel 12 is on but pixel 11 is off.  

$$\text{RMS} = \sqrt{((1.5)^2 + (3)^2 + (1.5)^2 + (3)^2) / 4} = 2.37\text{V}$$
- i. Repeat for pixel 12 when pixel 12 is on and pixel 11 is also on.  

$$\text{RMS} = \sqrt{((1.5)^2 + (3)^2 + (1.5)^2 + (3)^2) / 4} = 2.37\text{V}$$

6. If we have set up the loop time to occur every ten milliseconds, then we can make an event take place at a rate of N events per second, where N is the content of an unsigned char variable, in a manner similar to what we have done on some of the lab projects. That is, we might call an **ActNow** function 2 times per second if N = 2, or 35 times a second if N = 35. Defining any global variables you need, write an **Action** function that is called from within the main loop that will call **ActNow** periodically, at a rate of N times per second.

```

Define signed char ACCUM = 0;

void Action()
{
 ACCUM -= N;
 if (ACCUM < 0)
 {
 ACCUM += 100;
 ACTNOW();
 }
}

```

7. Show a C-coded implementation of the equation

[10%]  $\text{SCALEDVALUE} = \text{TRANSDUCER} / 2.34$

that modifies this operation to a multiplication followed by a shift of 8 places twice (so that the C compiler will replace the shifts by the moves of bytes). The intent is to minimize the execution time of the scaling operation, with multiplications and shifts executing faster than a division. Assume that SCALEDVALUE and TRANSDUCER are unsigned int variables. Define any extra variables that you need.

```

First note that $2^{16} = 65536$ and that $65536 / 2.34 = 28007$.
Define unsigned long TEMP;

TEMP = TRANSDUCER;
TEMP *= 28007;
TEMP >>= 8;
TEMP >>= 8;
SCALEDVALUE = TEMP;

```

1. [ ] I have completed the CIOS course evaluation.  
[5%] [ ] I have not completed the CIOS course evaluation.
2. Two models of 16x2 character LCD displays are available with a serial interface. Model A uses a 19200 baud UART interface (similar to that used by the Qwik&Low board for its connection to a PC). Model B uses an SPI interface (similar to that used by the Qwik&Low board for its connection to the starburst LCD). For this problem you are to consider how long it takes to update the LCD with a five-character string consisting of a "cursor-position control character" followed by four displayable characters. Ignore the time taken for anything other than the five serial transfers.
- [15%] a. How long does it take the MCU on the Qwik&Low board to send a five-character string via its 19200 baud UART interface to Model A? Explain your answer.  
With a baud rate of 19,200, each character takes 10 bit times, where a bit time =  $1000000/19200 = 52.08\mu\text{s}$ . So, a character takes  $10 \times 52.08 = 520.8\mu\text{s}$  or about 0.5 ms. Five characters take  $5 \times 0.5\text{ms} = 2.5\text{ms}$
- b. How long does it take the MCU on the Qwik&Low board to send a five-character string via its SPI interface to Model B? Explain your answer.  
With a CPU clock rate of 1 MHz, the SPI interface transfers each byte in about  $10\mu\text{s}$ . Five characters take  $5 \times 10\mu\text{s} = 50\mu\text{s}$
3. The magnetic card reader discussed in class includes an LRC (longitudinal redundancy check) character for the data bytes read from the card. For this problem, assume that a char variable called **LRC** has been defined.
- [15%] a. When a magnetic card is scanned and after the first active-low "1" bit has been detected, show the code to initialize **LRC** to zero and to turn off the LED on the Qwik&Low board.  

```
LRC = 0;
PORTDbits.RD4 = 0;
```
- b. Each time a group of five bits has been shifted into a char variable called **MAG\_BYTE**, the **LRC** variable is to be updated by exclusive-ORing **MAG\_BYTE** into it. Show the code to do this.  

```
LRC ^= MAG_BYTE;
```
- c. After the "end sentinel" has been received and detected, the next five bits that are shifted into **MAG\_BYTE** are again to be exclusive-ORed into **LRC**. When this has been done, the lower five bits of **LRC** should equal zero if no LRC error has been detected. Show the code to make this check of the lower five bits of **LRC** and to turn on the LED on the Qwik&Low board if there is an LRC error. You may assume that the upper three bits of **LRC** are zero.  

```
if (LRC)
{
 PORTDbits.RD4 = 1;
}
```
4. On the back of the review sheet for the final exam, I included a circuit showing the SPI connection between a PIC microcontroller and an LCD character display that employs the standard Hitachi display controller interface. For this problem you are to consider replacing this circuitry with nothing but
- [65%] • another PIC microcontroller to serve as a display controller  
• the LCD character display with its Hitachi display controller interface
- The only connections to these two parts from the Qwik&Low board are
- an active-low interrupt input, INT0
  - SCK input
  - SDI input
  - VDD (power)
  - GND
- Upon powering up, the PIC display controller initializes the LCD character display with an eight-bit interface. The pins required of the PIC display controller are
- the five listed above
  - the eight data bits connecting it to the LCD
  - an output to the LCD's RS pin
  - an output to the LCD's E pin

## 4. (Continued)

The features required of the PIC display controller in addition to its SPI interface are

- internal clock
- interrupt input
- sleep mode

The PIC16F690 is a \$1.66 20-pin microcontroller that will serve this role. However for this problem, you will examine the code necessary assuming that the chip used is the PIC18LF4321.

- a. When an interrupt occurs via a falling edge on the **RB0/INT0** pin, the chip wakes up and gets ready to receive characters from the SPI interface. As each of up to 18 characters is received, it is put into a line buffer defined with the following global variable array, **LBUFFER**, and an index variable into it,

**LBINDEX**:

```
unsigned char LBUFFER[18]; // Line buffer
unsigned char LBINDEX; // A value between 0 and 17
```

Show the C coding for a function called **ReadIntoBuffer** that waits for each character to be received, reads it into the next position of **LBUFFER**, and then waits upon the reception of the next byte. You can assume that upon entering **ReadIntoBuffer**, the value of **LBINDEX** is zero. You are to exit from

**ReadIntoBuffer** after you have received and written the “null terminator” value of 0x00 into **LBUFFER**.

```
void ReadIntoBuffer()
{
 CHAR = SSPBUF; // Clear buffer
 PIR1bits.SSPIF = 0; // and flag bit initially
 DONE = 0;
 while (!DONE)
 {
 while (!PIR1bits.SSPIF); // Wait for byte to be received
 PIR1bits.SSPIF = 0; // Clear flag
 LBUFFER[LBINDEX++] = SSPBUF; // and read byte into buffer
 if (!SSPBUF)
 {
 DONE = 1; // Exit from while loop
 }
 }
}
```

- b. Now write a **WriteToLCD** function that will write each of the characters in **LBUFFER** to the LCD, stopping before writing the null terminator. Assume that **PORTD** is connected to the LCD’s 8-bit data input and that **RB4** and **RB5** are connected to the LCD’s RS and E pins, respectively. The first character to be sent from the buffer is to be treated as a “cursor position code” and written to the display with RS = 0. The subsequent characters are to be treated as displayable ASCII-coded characters and written to the display with RS=1. To write each character after the 8-bit data and the RS pin are in place, drive E high and then low again. After writing to the display, use the **Delay** macro (having the same definition as we have used all semester) to insert a delay of 40 microseconds before writing the next character to the display.

```
void WriteToLCD()
{
 PORTBbits.RB4 = 0; // Clear RS for cursor position code
 LBINDEX = 0;
 while (LBUFFER[LBINDEX] // Continue until null character
 {
 PORTD = LBUFFER[LBINDEX++];
 PORTBbits.RB5 = 1; // Clock display
 PORTBbits.RB5 = 0;
 PORTBbits.RB4 = 1; // Set RS for subsequent displayable chars
 Delay(4); // Wait for 40us
 }
}
```

- c. On the chance that data has not been sent to this PIC+LCD combination in the correct format, you are to rewrite **ReadIntoBuffer** to handle the following cases:

**1** An omitted cursor-positioning code can be detected as a value less than 0x80. If the first character received is less than 0x80, then write 0x80 to **LBUFFER[0]** and write the received character to **LBUFFER[1]**. This will result in the display of the received characters beginning in the upper left-hand corner of the display.

**2** If fewer than eighteen characters are received and no null terminator is received, time out after one millisecond (measured from the entry into the **ReadIntoBuffer** function), using Timer0 to monitor this time. Before returning, insert a null terminator at the end of the received characters so that **WriteToLCD** will function correctly with the characters that have been received.

4. (Continued)

3 If **RB0/INT0** goes high before the null terminator has been received, then write 0x00 to the next available location in **LBUFFER** and return.

As you rewrite **ReadIntoBuffer**, mark your lines to the left of new code with the digit 1, 2, or 3 to show code that has been added to deal with items 1, 2, and 3 above.

```

void ReadIntoBuffer()
{
2→ TMR0H = *; // * = upper byte of 65536-1000
 TMR0L = **; // ** = lower byte of 65536-1000
 CHAR = SSPBUF; // Clear buffer
 PIR1bits.SSPIF = 0; // and flag bit initially
 DONE = 0;
 while (!DONE)
 {
2→ while (!PIR1bits.SSPIF) // Wait for byte or for timeout
 {
 if (INTCONbits.TMR0IF)
 {
 LBUFFER[LBINDEX] = 0x00; // Write null character in buffer
 break; // and exit
 }
3→ if (PORTBbits.RB0)
 {
 LBUFFER[LBINDEX+1] = 0x00;
 break;
 }
 }
1→ PIR1bits.SSPIF = 0; // Clear flag
 if ((LBINDEX == 0) && (SSPBUF < 0x80;
 {
 LBUFFER[LBINDEX++] = 0x80;
 }
 LBUFFER[LBINDEX++] = SSPBUF; // and read byte into buffer
 if (!SSPBUF)
 {
 DONE = 1; // Exit from while loop
 }
 }
}

```