• You may use a non-programming calculator only. You may use only the provided reference materials.
• When a binary result is required, give the value in **HEX**.
• For any required I2C functionality, use subroutine calls to `i2c_start()`, `i2c_rstart()`, `i2c_stop()`, `i2c_put(char byte)`, `char i2c_get(char ackbit)`. If you use `i2c_put`, you must pass in as an argument the byte that is to be written to the I2C bus. If you use `i2c_get`, you must pass in as argument the bit value to be sent back as the acknowledge bit value.
• Unless stated otherwise, all multi-byte data values are stored in **little-endian** ordering.
• Please note the relative value of each problem in the table below.
• Answers should be clearly indicated. Placing them in a **BOX** would be ideal.
• Be as neat and well organized as possible. This is in your grade’s best interest.
• If you need additional space to work, do so on the backside of the page. Make sure it is clear where your work continues.
• Absolutely NO cheating is allowed. If you are caught in the attempt of, the act of, or the past action of academic dishonesty, you will receive the maximum punishment allowed by University policy.
• No panicking allowed!

<table>
<thead>
<tr>
<th>Page</th>
<th>Maximum</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6-7</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

**MY CALCULATOR MODEL:** ___________________________________________
Part I: (79 pts)

a. (9 pts) Complete the C code loop that causes the MAX 517 to output a voltage of VREF (or as close as you can get), and decrease all of the way to 0 V in steps of 1 LSb (least significant bit). You must use I2C functions when causing the MAX517 to perform a conversion. Assume A1 is wired to +5V and A0 is wired to GND.

```c
char dac_value;
dac_value = 0xFF; // initial value is as close to VREF as possible
while (dac_value != 0xFF) { // exit when we wrap around back
    i2c_start();
    i2c_put(0x5C); // dac address byte
    i2c_put(0x00); // dac command byte
    i2c_put(dac_value); // value to output
    i2c_stop();
    dac_value--; // decrement to next voltage

    if (!RI) {
        RI = 1;
        printf("Software reset detected!");
    }
```
For the next several questions, consider the following C code for an ISR and main loop.

```c
interrupt my_isr() {
    if (INT1IF) {
        INT1IF = 0;
        RB5 = 1;
    }
} //end my_isr()

main() {
    TRISB5 = 0; TRISB1 = 1;
    RB5 = 1;
    INTEDG1 = 0; IPEN = 0; INT1IF = 0; INT1IE = 1; GIE = 1; PEIE = 1;
    while (1) {
        RB5 = 0;
    } // end while()
} //end main()
```

d. (6 pts) Describe the response of the system that runs the code above. Be specific in describing what stimuli cause what responses.

A falling edge on RB1 (for example, from a pushbutton) triggers the ISR, which sets RB5 back high again. After the ISR exits, execution is picked back up in the while(1) loop, causing RB5 to go low again.

e. (6 pts) What determines the HIGH pulse width time in the interrupt code above? Be as specific and inclusive as possible.

High pulse time is the time it takes to (i) exit ISR, (ii) restore registers from shadow registers, (iii) return to while(1){} loop, and (iv) get back to RB5=0

f. (6 pts) What determines the LOW pulse width time in the interrupt code above? Be as specific and inclusive as possible.

Low pulse time is the time it takes to (i) recognize interrupt, (ii) save registers in shadow registers, and (iii) execute code up to RB5=1 in ISR
g. (10 pts) Your PIC18 has “normally-open” pushbuttons connected to port pins RB1 and RB2 with internal pullups enabled. Assume both pins have been configured to generate rising-edge-triggered, high-priority interrupts. Write an ISR that increments a variable named `edge_count` by 2 if the INT1 interrupt occurs, or increments `edge_count` by 1 if the INT2 interrupt occurs. After an interrupt occurs, disable that interrupt using its enable flag, clear the interrupt flag bit, and set the semaphore variable switch_pressed to a ‘1’. Don’t debounce the switches.

```c
interrupt isr() {
    if (INT1IF) {
        INT1IF = 0; INT1IE = 0;
        edge_count += 2;
        switch_pressed = 1;
    } // end INT1F if-block
    if (INT2IF) {
        INT2IF = 0; INT2IE = 0;
        edge_count += 1;
        switch_pressed = 1;
    } // end INT2F if-block
} // end interrupt_isr()
```

h. (8 pts) The PIC18 automatically saves the registers BSR, W, and STATUS into the shadow registers when an interrupt occurs for either a low or high priority interrupts. Assume that you had a system with both low and high priority interrupts. What is the first thing that your low priority interrupt service routine should do with the values saved in the BSR, W, STATUS shadow registers? Explain why. Is this also necessary for the high priority ISR? Why or Why not?

The low priority ISR should save the BSR, W, and STATUS in some temporary locations. If a high priority interrupt occurs, the shadow registers are used to save the BSR/W/Status registers. Saving BSR, W, and STATUS to temporary registers is not needed by the high priority interrupt because the high priority ISR will not be interrupted.
i. (10 points) Write C code that will configure the A/D module for left justification, AN2, AN1 as analog inputs, AN3 as VREF+. VSS as VREF-. Use the internal FOSC clock, and configure the A/D clock such that it meets the minimum clock period constraint of 1.6μs assuming an FOSC of 12 MHz. (Use the fastest internal clock choice that meets this constraint). For the configuration code, use individual bit names ADCS2, ADCS1, ADCS0, ADON, ADFM, PCFG3, PCFG2, PCFG1, PCFG0. You do not have to configure the channel select bits, that is done next.

Before we write code, we must first determine the best A/D clock divider choice:
FOSC/16 = 750 KHz, is period of 1.3 us < 1.6 us. Clock period is too short!
FOSC/32 = 375 KHz, is period of 2.7 us > 1.6us, use this choice!

// AD Configuration, ADCON0 register
ADCS2 = 0; ADCS1 = 1; ADCS0 = 0; // configure with FOSC/32
ADON = 1; // A/D turned on
ADFM = 0; // left justified
// AN2, AN1 analog inputs, AN3 = VREF, multiple solutions
PCFG3 = 0; PCFG2 = 0; PCFG1 = 0; PCFG0 = 1; // Other valid configs:
// 0011, 0101, 1010

j. (10 points) Write a function called char analog_sum() that will perform a conversion on each analog input (AN2, AN1) and return the sum of these values as a char value. Do not let the sum exceed 255 (0xFF) (Hint: You will need to use an unsigned int variable to hold the sum, then clip this to 255). When changing A/D channels, use the DelayUs function to delay 20 μs to give the A/D input a chance to settle. Since you don’t know how often this function will be called, use this delay before starting each conversion.

char analog_sum(){
    unsigned int tmp;
    char tmpc;
    CHS2=0;CHS1=0;CHS0=1; // select channel 1
    DelayUs(20); // wait for settling
    GODONE= 1; //godone bit = 1
    while (GODONE);
    tmp = ADRESH; // read result
    CHS2=0;CHS1=1;CHS0=0; // select channel 2
    DelayUs(20);
    GODONE= 1; //godone bit = 1
    while (GODONE);
    tmp = tmp + ADRESH; // add new result to old result
    if (tmp > 255) tmp = 255;
    tmpc = tmp;
    return(tmpc);
}
Part II: (21 points) Answer the 7 out of the next 9 questions. Each question is worth 3 points.

YOU MUST CLEARLY CROSS OUT THE QUESTIONS YOU DO NOT WANT GRADED!!!!

a. What is the \( \text{SPRG} \) value for a baud rate of 19200 assuming an \( F_{\text{OSC}} \) of 15 MHz and high speed mode?

\[
\text{SPRG} = \frac{15 \text{ MHz}}{16 \times 19200 \text{ baud}} - 1 = 47.8, \text{ round to 48}
\]

b. Can you hook the PIC TX and RX pins directly to pins 2, 3 of the DB9 connector to implement the serial port connection to the PC? Why or why not? Explain.

No, RS232 levels are \(-3\) V to \(-25\) V for logic 1, and \(+3\) V to \(+25\) V for logic 0. You need an interface IC like the MAX232 to convert from CMOS levels to RS232 levels and vice-versa.

c. Your 7-bit DAC has an input code of 0x2B and a \( V_{\text{REF}} = 8 \text{ V} \). What is the output voltage?

\[
\text{DAC\_code}/2^7 \times V_{\text{REF}} = V_{\text{out}}; \quad 0x2B/2^7 \times 8 \text{ V} = 43/128 \times 8 \text{ V} = 2.6875 \text{ V}
\]

d. There are two important pieces of information specified in the first byte of any \( \text{I}^2\text{C} \) transaction – what are they? Where are they located relative to each other?

The address of the I2C device and whether the transaction is a read or a write. The R/W# bit is the least significant bit of the “8-bit” I2C address.

e. Write a C code fragment that performs an acknowledge condition on the \( \text{I}^2\text{C} \) bus (do not use the \text{i2c\_ack(char ackbit)} function, I want to know what is inside the \text{i2c\_ack function}).

\[
\text{void i2c\_ack(ackbit) \{}
\begin{align*}
\text{ACKDT} &= \text{ackbit}; \quad \text{// set ack bit value} \\
\text{// initiate acknowledge cycle} \\
\text{ACKEN} &= 1; \\
\text{// wait until acknowledge cycle finished} \\
\text{while(ACKEN); \quad// CCP1CON[3:0] = 0100}
\end{align*}
\]

\] // end i2c\_ack
f. Explain *EITHER* the operation of a 4-bit *successive approximation* ADC or a 4-bit *flash* ADC. For both ADCs, use $V_{in} = 1.7\, V$ and $V_{ref} = 4\, V$. If you explain the successive approximation ADC, you have to give the internal $V_{DAC}$ voltage used at each comparison step, and list all steps. If you explain a flash ADC, you have to give the number of comparators and resistors, the output value (1 or 0) of all comparators. For either ADC, you have to give the final 4-bit output code.

4-bit Successive approximation:

step 1: $V_{DAC}$ code = 1000, $V_{DAC} = \frac{8}{16} \times 4\, V = 2\, V$, $V_{in} = 1.7$. $V_{DAC} > V_{in}$, wrong guess $bit[3] = 0$

step 2: $V_{DAC}$ code = 0100, $V_{DAC} = \frac{4}{16} \times 4\, V = 1\, V$, $V_{in} = 1.7$. $V_{DAC} < V_{in}$, correct guess, $bit[2] = 1$

step 3: $V_{DAC}$ code = 0110, $V_{DAC} = \frac{6}{16} \times 4\, V = 1.5\, V$, $V_{in} = 1.7$. $V_{DAC} < V_{in}$, correct guess, $bit[1] = 1$

step 4: $V_{DAC}$ code = 0111, $V_{DAC} = \frac{7}{16} \times 4\, V = 1.75\, V$, $V_{in} = 1.7$. $V_{DAC} > V_{in}$, wrong guess, $bit[0] = 0$.

Final output code 0110.

4-bit Flash:

15 comparators, 16 resistors. The reference voltage of each comparator increases by $\frac{1}{16} \times 4\, V = 0.25\, V$ each step up the resistor string. So, the $V_{ref}$ voltages from top (left) to bottom (right) of the 15 comparators are:

$V_{ref}$ = 3.75 3.5 3.25 3.0 2.75 2.5 2.25 2.0 1.75 1.5 1.25 1.0 0.75 0.5 0.25

$V_{in}$ = 1.7 1.7 1.7 1.7 1.7 1.7 1.7 1.7 1.7 1.7 1.7 1.7 1.7 1.7 1.7

$V_{in} > V_{ref}?$ 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 (comparator outputs)

Output code is 0110 (1.7/4 $V / 16 = 6.8$, truncate to 6 = 0110 b).

g. How many bit times are there in the $I^2C$ transaction to the MAX517 DAC for a conversion? Count the start and stop conditions each as one bit time.

1 start+ 1 stop + addr byte (9) + cmd byte(9) + data byte(9) = 29 bit times total

h. Assume the PIC18 A/D is configured with a $V_{ref^+} = 4\, V$, and a $V_{ref^-} = 0\, V$, and that two successive reads of the ADC are done, with the first value returning 0xA0, and the next value returning 0x1A (only the 8 MSb are read). What is the change in voltage?

\[
\text{Difference is } 0xA0 - 0x1A = 0x86 = 134. \\
134/256 \times 4\, V = 2.09375\, V
\]

i. In asynchronous serial communication, why can’t we send large numbers of bits at a time? (RS232 uses a maximum of 10 bits).

Sender and receiver do not share common clocks so any clock mismatch error accumulates with each bit that is sent. Eventually the receiver will sample input bit in wrong place resulting in communications errors.
"DEJA VU" BONUS QUESTION (5 points)
You must answer this question completely correct to receive the bonus points.

After the execution of ALL of the C code below, fill in the memory location values. Assume little-endian order for multi-byte values.

```c
signed char a[2];
signed int b;
signed long c;
signed int *ptrb;

a[0] = 3;
a[1] = -7;
ptrb = &a[0];
b = *ptrb + 2;
ptrb++;
c = *ptrb >> 1; // Computes *ptr >> 1 as int, then converts to a long
```

Location | Contents (MUST be given in hex) |
---|---|
0x1A0 | 0x03 | ; a[0] = 3 |
0x1A1 | 0xF9 | ; a[1] = -7 = 0xF9 |
0x1A2 | 0x05 | ; b = *ptrb + 2. ptrb points to a[0 – 1]. 0xF905 + 1 = 0xF906. 0x05 is low byte, 0xF9 is high byte. |
0x1A3 | 0xF9 | ; c = *ptrb >> 1. ptrb now points to b, so c = b >> 1. 0xF905 >> 1 = 0x82 |
0x1A4 | 0x82 | 1111 1001 0000 0101 >> 1 = 1111 1001 0000 010 = |
0x1A5 | 0xFC | 1111 1100 1000 0010 = 0xFC82 |
0x1A6 | 0xFF | As a long, this is 0xFFFF FC82 |
0x1A7 | 0xFF | ; ptrb points to b, at 0x1A2 |
0x1A8 | 0xA2 |  |
0x1A9 | 0x01 |  |