

You may use a calculator and the provided reference materials. If 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)*, *i2c_get(char ackbit)*, *char i2c_put_noerr(char byte)*. 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 an argument the bit value to be sent back as the acknowledge bit value.

You also have available the functions:

```
i2c_memwrite(char i2caddr, unsigned int addr, volatile unsigned char *buf)
i2c_memread(char i2caddr, unsigned int addr, volatile unsigned char *buf
```

Part I: (72 pts)

- a. (15 pts) Assume you have a 32.768 KHz clock as the clock source for TIMER1 with a prescale of 1 (timer1 overflows every 2 seconds). Every 30 seconds, do an ADC conversion and store the upper 8-bits in a buffer. Once the buffer has 64 entries write this to the 24LC515 EEPROM (A1, A0 on EEPROM both tied low). Do all of this work in an ISR, and use the *i2c_memwrite()* function to write the data to the EEPROM. Write the ISR that accomplishes this; do not assume that Timer1 is the only enabled interrupt. Assume the ADC has already been configured, and that it is left justified. Assume timer1 is already configured and that its interrupt is enabled. Each time a block of 64 bytes is written, increment the address passed to *i2c_memwrite()*. When the EEPROM fills up, disable the TIMER1 interrupt.

```
char time_count, index, buf[64];
int mem_addr;

interrupt my_isr() {
    if (TMR1IF) {
        TMR1IF = 0;
        time_count++;
        if (time_count == 15) { // 30 seconds
            time_count = 0;
            GODONE=1; // start ADC conversion
            while(GODONE); // wait until finished
            buf[index] = ADRESH;
            index++;
            if (index == 64) {
                i2c_memwrite(0xA0, mem_addr, buf); // write data to EEPROM
                mem_addr = mem_addr + 64; // increment to next page
                if (mem_addr == 0) { // if overflowed, then EEPROM full
                    TMR1IE = 0; // disable interrupt
                }
            } // end if (time_count..)
        } // end if (TMR1IF)
    } // end my_isr()
```

- b. (12 pts) Using the I2C functions, write a C code fragment that loops, polling a 24LC515 serial EEPROM for end-of-write. Assume that both the A1, and A0 pins on the EEPROM are tied high. The loop should not exit until the 24LC515 indicates the current write operation is finished. The function *char i2c_put_noerr(char byte)* is needed as it returns the value of the acknowledge bit returned after the byte write to the I2C bus.

```
// one solution
i2c_start(); //start transaction
// now, poll for end of write
while (i2c_put_noerr(0xA6)); // loop while ACKBIT == 1 (NAK)
// at this point write operation is finished
```

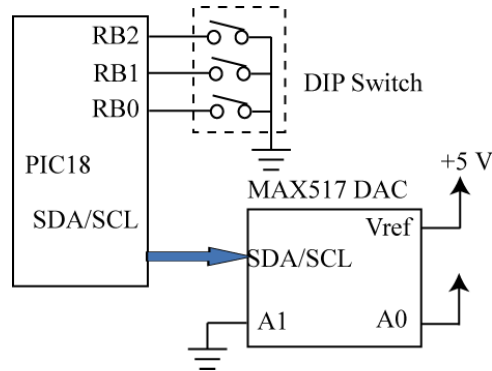
```
// another solution
do {
    i2c_start(); //start transaction
    ackbit = i2c_put_noerr(0xA6);
    i2c_stop();
} while(ackbit); //loop as long as ackbit returns as '1' (NAK)
```

- c. (6 pts) Assume FOSC = 40 MHz. Using the PWM module, give the PR2, PRE, and CCPR1L values for a 6 KHz square wave with a 20% duty cycle.

```
Timer2 PWM period = (PR2+1) * (4/FOSC) * PRE      (POST is NOT used for PWM period)
(1/6 KHz) = (PR2+1) * (4/40 MHz) * PRE
PR2 = [(40 MHz/4) / (6 KHz * PRE) ] - 1;
For PRE = 1, PR2 = 1666 (> 255, too large)
For PRE = 4, PR2 = 416 (> 255 too large)
For PRE = 16, PR2 = 103, so use PRE=16, PR2 = 103 = 0x67.
For 20% duty cycle, CCPR1 = 0.20 * 103 ~ 21 = 0x15.
```

- d. (12 pts) In the diagram below, a three position dip switch is connected to the lower three bits of PORTB. Assume the weak pullups have been enabled on PORTB. Thus when all switches are CLOSED, the three bit value is read as “000” = 0; when all switches are OPEN the three bit value is read as “111” = 7. Assume TIMER2 has been configured to generate a periodic interrupt. Write an ISR so that on each TIMER2 interrupt, read the switches and cause the DAC output voltage to be the values shown below. Use the individual I2C function calls to communicate with the DAC; you cannot use the ‘update_dac()’ function. When you read PORTB, you CANNOT assume any values for bits RB3-RB7. Do not assume that TIMER2 is the only enabled interrupt.

Dip switch value	DAC Vout
“000”,	0 V;
“001”,	1/8 VREF
“010”,	2/8 VREF
“011”,	3/8 VREF
....	
“111”,	7/8 VREF.



First, you had to recognize how to convert the PORTB three-bit value (0-7) to an 8-bit value to be sent to the DAC. You know that:

$$\text{DAC_code}/256 * \text{Vref} = \text{Vout.}$$

So, if $\text{Vout} = 1/8 \text{ Vref}$, then:

$$\text{DAC_code}/256 * \text{Vref} = (1/8) * \text{Vref}$$

$$\text{DAC_code} = 256*(1/8) = 32 \text{ to produce an output voltage of } 1/8 \text{ Vref}$$

Similarly

$$\text{DAC_code} = 256*(2/8) = 64 \text{ to produce an output voltage of } 2/8 \text{ Vref, etc.}$$

So the code is:

```

char dac_value;
interrupt my_isr() {
    if (TMR2IF) {
        TMR2IF = 0;
        dac_value = PORTB & 0x07; //force upper 5 bits to zero
        dac_value = dac_value * 32; // multiply by 32 to get DAC code
        i2c_start();
        i2c_put(0x5A); //DAC address 0101 1010
        i2c_put(0x00); //command byte that says to convert next value
        i2c_put(dac_value);
        i2c_stop();
    } //end TMR2IF
}

```

- e. (6 pts) How many TIMER1 tics is contained in 1 ms given an FOSC = 40 MHz and a

One Timer1 tic is $4/\text{FOSC} * \text{prescale} = 4/40 \text{ MHz} * 2 = 2 e^{-7} \text{ s} = 0.2 \text{ us}$

1 ms/ 1 Timer tic = $0.001 \text{ s} / 2 e^{-7} \text{ s} = 5000 \text{ tics}$.

- f. (7 pts) Write C configuration code that configures the ADC to select channel 4, left justified result, an ADC clock of FOSC/16, turns the ADC on, has VREF+=AN3, VREF-=AN2, and the other PORTA inputs as being analog inputs. Use individual bit assignments for clarity, do NOT assign 8-bit values to ADCON0 and ADCON1.

```
CHS2 = 1; CHS1 = 0; CHS0 = 0; // select channel AN4  
ADCS2 = 1; ADCS1 = 0; ADCS0 = 1; //FOSC/16  
ADFM = 0; // left justified  
PCFG3 = 1; PCFG2 = 0; PCFG1 = 0; PCFG0 = 0; // AN3-AN0 config  
ADON = 1;
```

- g. (6 pts) Assume the code used in lab to measure the pulse width of a pushbutton switch. On the falling edge (pushbutton pressed), the capture register captures the hex value 0x0700 from timer1. On the rising edge (pushbutton released), the capture register captures the value 0x0200 from Timer1, with TWO timer1 overflows between the falling and rising edge captures. Assuming a Timer1 prescale of 2, an FOSC = 40 MHz, and using the internal clock, how long is the pulse width in microseconds?

```
#of timer tics = (timer1_overflows - 1) * 216 + (0x0000-A) + B  
= (2-1) * 66536 + (0x0000 - 0x0700) + 0x0200  
= 65536 + 0xF900 + 0x0200  
= 65536 + 0xFB00 = 65536 + 64256 = 129792 timer tics
```

One Timer1 tic is $4/\text{FOSC} * \text{prescale} = 4/40 \text{ MHz} * 2 = 2 e^{-7} \text{ s} = 0.2 \text{ us}$

```
Pulse width = timer tics * period of 1 timer tic  
= 129792 * 0.2 us = 25958.4 us
```

- h. (8 pts) Explain EITHER the operation of a 4-bit successive approximation ADC or a 4-bit flash ADC. For both ADCs, use $V_{in} = 1.7\text{ V}$ and $V_{ref} = 4\text{ V}$. If you explain the successive approximation ADC, you have to give the internal VDAC 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.

Successive approximation:

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

step 2: VDAC code = 0100, VDAC = $4/16 * 4\text{ V} = 1\text{ V}$, $V_{in} = 1.7$. VDAC < V_{in} , correct guess, bit[2]=1

step 3: VDAC code = 0110, VDAC = $6/16 * 4\text{ V} = 1.5\text{ V}$, $V_{in} = 1.7$. VDAC < V_{in} , correct guess, bit[1]=1

step 4: VDAC code = 0111, VDAC = $7/16 * 4\text{ V} = 1.75\text{ V}$, $V_{in} = 1.7$. VDAC > V_{in} , wrong guess, bit[0] = 0.

Final output code 0110.

Flash:

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

Vref= 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 0 0 1 1 1 1 1 (comparator outputs)

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

Part II: (28 pts) Answer 7 out of the next 9 questions. Cross out the 2 questions that you do not want graded. Each question is worth 4 pts.

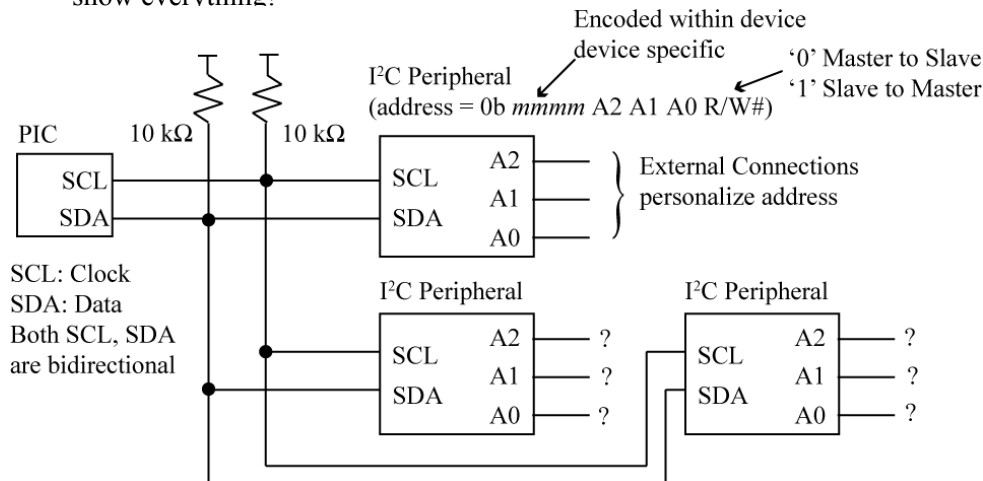
1. A 7-bit DAC has an input code of 0x3B and a $V_{REF} = 10\text{ V}$. What is the output voltage?

$$Dac_code / 2^N * V_{ref} = V_{out}; \quad 0x3B / 2^7 * 10\text{ V} = 59 / 128 * 10\text{ V} = 4.61\text{ V}.$$

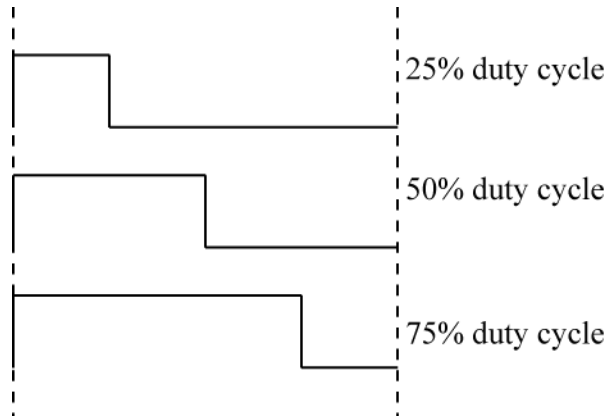
2. There are two important pieces of information specified in the first byte of any I2C transaction – what are they?

The address of the I2C device, and if it is a read or write transaction.

3. Draw a picture that shows two devices connected to the I2C bus of the PIC18. Be sure to show everything!



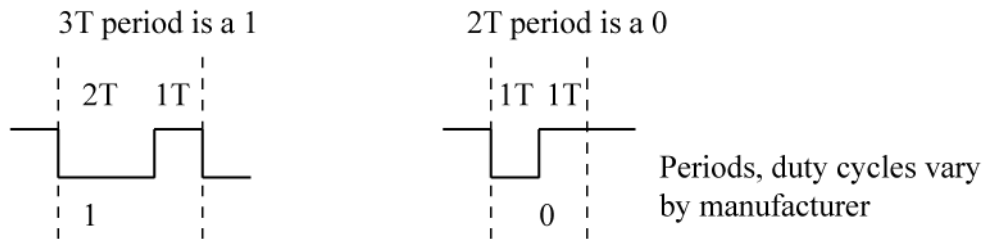
4. How would PWM be used to control the illumination of an LED? Show waveforms for 25% maximum illumination, 50% maximum illumination, and 75% maximum illumination.



PERIOD does NOT change. Only duty cycle changes.

5. Draw pictures that show how a “1” is distinguished from a “0” using SPACE-WIDTH encoding. Your pictures must be VERY clear in how the “1” and “0” bits are different – you can choose the actual ratios of the waveforms yourself.

(a) Space-Width Encoding, 1 and 0 distinguished by period length



6. In class and in the notes, we discussed the proper way to WRITE a 16-bit value to timer1 so that it is updated correctly. Show C code that updates TIMER1 with the value from ‘int my_value’ correctly.

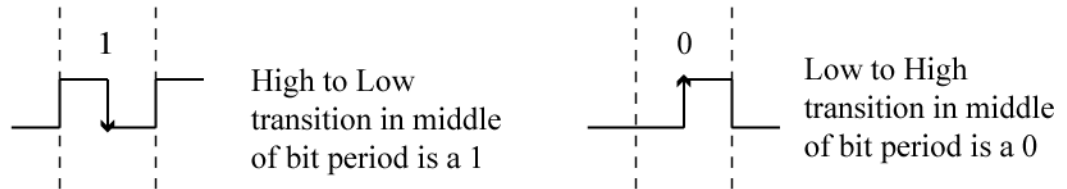
```
TIMER1 high byte must be written first!
TMR1H = (my_value) >> 8;
TMR1L = my_value & 0xFF;
```

7. Write C code that configures the capture system to capture every falling edge on CCP1, and to use TIMER3 as the timer register for CCP1 pin captures.

```
//CCP1CON[3:0] = 0100
CCP1CON3=0; CCP1CON2 =1; CCP1CON1 = 0; CCP1CON1 = 0;
T3CCP2 = 1;           //select timer3 as capture source
```

8. Draw pictures that show how a “1” is distinguished from a “0” using BIPHASE encoding. Your pictures must be VERY clear in how the “1” and “0” bits are different.

(a) Biphase encoding, 1 and 0 distinguished by transition in middle of bit period



9. Write a C code fragment that performs an acknowledge condition on the I2C bus (do not use the `i2c_ack(char ackbit)` function, I want to know what is inside the `i2c_ack` function!).

```
ACKDT = ackbit;           // set ack bit value
//initiate acknowledge cycle
ACKEN = 1;
// wait until acknowledge cycle finished
while(ACKEN);           //CCP1CON[3:0] = 0100
```