

You may use only the provided reference materials. You may use a calculator, either a four-function or a scientific calculator. You may not use a programmable calculator. The test is worth 100 pts, you are given 1 pt for free. For any required I2C functionality, use subroutine calls *i2c\_start()*, *i2c\_rstart()*, *i2c\_stop*, *i2c\_put(char byte)*, *i2c\_put\_noerr(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. Recall that *i2c\_put\_noerr* returns the value of the ACK bit that is returned by the slave device, while *i2c\_put* does a software reset if the slave device returns an ACK of '1'. If you use *i2c\_get*, you must pass in an as argument the bit value to be sent back as the acknowledge bit value. You also have *DelayUs()* and *DelayMs()* functions available. Show all your work in any computations done or formulas used.

## Part I: (80 pts)

- a. (10 pts). Write a function called *void write\_rdy()* that does not returns until the 24LC515 serial EEPROM has finished with its last write. You must poll the device for 'end-of-write' to determine when the device has finished; the *i2c\_put\_noerr()* function will come in handy for this. Assume that both the A0, and A1 lines of the EEPROM are tied low.

```
void write_rdy(void)
{
    char ack_bit;

    do {
        i2c_start();
        ack_bit = i2c_put_noerr(0xA0); //send write command, get ack bit
        i2c_stop();
    }
    while(ack_bit); //only exit when ack_bit returns as '0'
```

- b. (5 pts) The function below is communicating with a 24LC515 Serial EEPROM. Give the range of addresses that are read in the EEPROM from the viewpoint of the 64K address space (the range must be within the range 0x0000 – 0xFFFF).

```
myfunc() {  
  
    int i;  
    char x;  
  
    i2c_start();  
    i2c_put(0xAE);  
    i2c_put(0x00);  
    i2c_put(0x00);  
    i2c_restart();  
    i2c_put(0xAF);  
    i = 0;  
    do {  
        x = i2c_get(0);  
        i++;  
    }while (i < 0x8000);  
    i2c_stop();  
}
```

0xAE command is write to upper block,  
internal address register is zeroed.

0xAF command is read, will read 0x8000  
bytes from UPPER block.

So locations 0x8000 through 0xFFFF are  
read.

- c. (15 pts) Write a C function named `char adc_check()` that performs conversions on channels AN1, AN2, and AN3 and returns a '1' if any of them are outside of the range 2V to 4V. Assume a VREF of 5 V, that the ADC is configured for left justification, and that we are only interested in the upper 8-bits of the conversion. Do not assume that a particular channel is selected on entering the function. Delay for 20 us before beginning a conversion after selecting a channel.

```
// 2V is 2/5 * 256 = 102.      4V is 4/5*256 = 204

char adc_check()
{
    char x;

    CHS2=0;CHS1=0;CHS0 =1; //select channel 1
    DelayUs(20);
    GODONE = 1;
    while(GODONE); //wait for conversion
    x = ADRESH;
    if (x < 102 || x > 204) return(1);

    CHS2=0;CHS1=1;CHS0 =0; //select channel 2
    DelayUs(20);
    GODONE = 1;
    while(GODONE); //wait for conversion
    x = ADRESH;
    if (x < 102 || x > 204) return(1);

    CHS2=0;CHS1=1;CHS0 =1; //select channel 3
    DelayUs(20);
    GODONE = 1;
    while(GODONE); //wait for conversion
    x = ADRESH;
    if (x < 102 || x > 204) return(1);

    return(0);
}
```

- d. (15 pts) Write a function called `dac_step(void)` that tests the MAX517 DAC by stepping its output value from 0 V to its full range in steps of 1 least significant bit. Assume that both AD1 and AD0 of the MAX517 are tied high.

```
void dac_step(void) {
char  dac_value;
  dac_value = 0;          //start at 0
  do {
    i2c_start();
    i2c_put(0x5E);        //dac address byte, assume A1=A0=1
    i2c_put(0x00);        //dac command byte
    i2c_put(dac_value);   //value to output
    i2c_stop();
    dac_value++;          //increment to next voltage
  }while(dac_value);     //exit when we wrap to zero
}
```

- e. (15 pts) The delay function `DelayMs(char x)` that we have used in labs is a software delay loop function. Write an equivalent function that uses Timer1 and compare mode (CCPR1) to accomplish the same thing. Follow the approach of using a loop that zeros Timer1, then writes a value to CCPR1 that is equivalent to a 1 ms delay, then waits for the CCPR1IF flag to be set. To wait for  $x$  ms, execute this basic loop  $x$  times. Assume an FOSC of 40 MHz. In addition to the code, show calculations that determine the value to write to the CCPR1 register – you pick the prescale value that you will use for this calculation. You do not have to write any configuration code for compare mode, assume that this has already been done.

```
Let prescale = 1 for Timer 1.
time = timer_tics * clock_period
timer_tics = time / clock_period
timer_tics = 0.001 / ( (4/FOSC) * Prescale)
timer_tics = 0.001 / ( (4/40e6) * 1)
timer_tics = 10000

void DelayMs(char x) {

    CCPR1 = 10000;          //set CCPR1
    while(x) {             //exit when x is zero
        TMR1H = 0;
        TMR1L = 0;         //zero Timer1, write low byte last
        CCPR1IF = 0;       //clear CCPR1IF flag
        while(!CCPR1IF);  //wait for match
        x--;
    }
}
```

- f. (10 pts) Write C code that will configure the CCP1 output and the PWM module of the PIC18 to generate a square wave with 250 us period and a 40% duty cycle assuming an FOSC of 30 MHz. Show the calculations first, then show the code.

```

250 us = (PR2+1) 4/30 Mhz * PRE
PR2 = [250 us * 30 MHz/(4 * PRE)] - 1;
for PRE=1, PR2 = 1874 (too large)
for PRE=4, PR2 = 468 (too large)
for PRE=16, PR2 = 116 use this
CCPR1L = 0.4 * (PR2+1) = 46.8 = 47

main() {
    PR2 = 468; CCPR1L = 47; TRISC2 = 0;
    T2CKPS1 = 1; //prescale 16
    CCP1M3 = 1; CCP1M2 = 1; //pwm mode
    TMR2ON = 1;
    while(1);
}

```

- g. (10 pts) Explain EITHER the operation of a 3-bit successive approximation ADC or a 3-bit flash ADC. For both ADCs, use  $V_{in} = 1.7\text{ V}$  and  $V_{ref} = 6\text{ V}$ . 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 3-bit output code.

#### Successive Approximation:

First guess: 100, produces voltage  $4/8 * 6V = 3V$ . This is greater than  $V_{in}$  (1.7V), so first bit is 0  
 2<sup>nd</sup> guess: 010 produces voltage  $2/8 * 6V = 1.5V$ . This is less than  $V_{in}$  (1.7V), so 2<sup>nd</sup> bit is 1  
 3<sup>rd</sup> guess 011 produces voltage  $3/8 * 6V = 2.25V$ . This is greater than  $V_{in}$  (2.25V), so last bit is 0.  
 Final answer: 010.

Flash: 7 comparators, 8 resistors. Each resistor represents  $6V/8 = 0.75$  steps. The comparator input voltages are (from top comparator to bottom comparator): 5.25 V, 4.5, 3.75, 3.0, 2.25, 1.5, 0.75.

Thus, the output of the comparators are 0 0 0 0 0 1 1,  
 with the 3 bit output being 010 (2) since the output of two comparators are equal to '1'.

Part II: (20 pts) Answer 5 out of the next 8 questions. Cross out the 3 questions that you do not want graded. Each question is worth 4 pts.

1. Assume an FOSC of 20 MHz. What is the maximum length (in milliseconds) of a periodic interrupt generated using Timer2?

```
Timer2 interrupt period = 4/FOSC (PR2+1) *PRE * POSTSCALE
max PRE=16, max POST = 16, max PR2 = 255
max Timer2 intrpt period = 4/20 MHz * 256 * 16 * 16
                          = 13.1 ms
```

2. What does the least significant bit of the first byte of every I2C transaction signify?

```
if '0' then write transaction
if '1' then read transaction
```

3. Write a C code fragment that returns the upper 8-bits of the PIC18 ADC result value in the char variable *c* regardless of whether the ADC is configured as left justified or right justified.

```
char c;

if (!ADFM) c = ADRESH; //left justified
else {
    //right justified
    c = ADRESH;
    c = c << 6;
    c = c | (ADRESL >> 2);
}
```

4. We discussed three different types of A/D architectures in class. Describe how the SLOWEST of these A/D architectures functioned. Why is it slow?

The counter ramp is the slowest because it starts at the internal DAC voltage comparison voltage at zero and counts up, until it finds the DAC voltage that is greater than the input voltage. Its conversion time is proportional to the input voltage.

5. In using PWM to control either voltage or current, what is varied within the PWM waveform? What is kept constant?

The high pulse width of the PWM waveform is varied, the period is kept constant.

6. Write PIC18 code that configures TIMER2 for prescale of 4, and postscale of 11, internal clock as the clock source, and turns the timer on.

```
// one line statement:  
T2CON = 0x2D; // 0 0101 1 01  
  
or  
//multiple lines  
TOUTPS3 = 1; TOUTPS2 = 0; TOUTPS1 = 1; TOUTPS0 = 0;  
T2CKPS1 = 0; T2CKPS0 = 1;  
TMR2ON = 1;
```

7. If the Vref of the PIC18 is 4.1 V, and I read a 10-bit code of 0x200, what is the input voltage value?

$$0x200 / 1024 * 4.1 = 512 / 1024 * 4.1 = 2.05 \text{ V}$$

8. For an FOSC of 6 MHz, what is the fastest ADC clock configuration (other than the internal oscillator) that can be selected and still not violate the 1.6 us minimum period constraint? Show your work.

1.6 us is a 625 KHz frequency  
6 MHz / 32 = 187.5 KHz  
6 MHz / 16 = 375 KHz      Use this (FOSC/16)  
6 MHz / 8 = 750 KHz      TOO FAST!