BME 580 Microcontrollers I

Lab 5: MEMORY

Ross Rucho

In this lab you will continue to build upon previous labs by learning about the essential types of memory that are used in embedded systems:

- Internal memory types: RAM, ROM, EEPROM are the primary examples
- Mass storage memory in the form of external devices: SD Cards are one example

Memory can be writable (RAM – Random Access Memory) or traditionally non writable (ROM = Read Only Memory)

RAM and writable ROM (such as EEP*ROM*) come in two basic types: *volatile* and *non-volatile*

Volatile memory (many types of RAM, but usually not ROM) only retains the data values while the device is powered up. If power is interrupted or lost even for a brief instant, volatile memory basically vanishes. It resets to zero, or becomes corrupted. So this type of memory is usually used to store data during program execution, especially if that data is changed frequently.

Non-Volatile memory (both RAM and ROM) is designed to be retained during periods when no power is applied. The values stored in the memory are not lost when the power is shut off. This is handy for program code and for data and variables that may change occasionally, but that you must save between uses, such as calibration values, user settings, selectable pre-set values, and adjustable values that control device behavior at power-up. Examples of when you would need non-volatile memory include a calculator that remembers its most recent result from the last use, a digital radio tuner that remembers the volume and radio station it was last set to, a medical device that is recalibrated every few months for quality assurance, a digital temperature controller for a home or a freezer that must remember its setting after a power failure, or a personal security code that can be changed by the user. All of these examples require only a few bytes of non-volatile memory. Larger sets of data such as stored images, *.wav files, and scientific data also need to be stored using non-volatile memory. In some cases the data is never changed (such as play-only games or videos), but in some cases even huge data sets must be modified, added to, or corrected (such as a large set of scientific data). In the latter case, an SD Card is a very good solution: many GB of readable and writable data are available for a few dollars in a package only a few mm square.

Many embedded systems do not require a great deal of memory because most embedded systems basically perform just one or at most a few well-defined tasks that are neither computationally nor memory-intensive. A few kB is quite enough to control the temperature even in a large building. It is also enough to run a microwave oven, or even a small robotic toy. The core memory for the Apollo Guidance Computer (AGC) was called “core rope memory”, and Apollo only had 72 KB of core memory, which occupied an entire cubic foot of volume and was woven by hand, by skilled women (https://en.wikipedia.org/wiki/Core_rope_memory). This was enough embedded memory to guide the Apollo lunar module to a safe landing on the moon and a safe return to orbit for redocking with the command module prior to their return to Earth.

Memory has come a long way since Apollo, but because memory is still somewhat expensive, and most embedded applications do not require a lot of memory, and by microcontroller die standards memory takes up quite a lot of space, most microcontrollers do not have a lot of memory built in to their internal resources. On a typical microcontroller you generally get just a few kB of program memory (burned-in as non-volatile

BME 580 Microcontrollers I

and may or may not be writable during program execution), about 1kB of data memory (volatile, but can be changed many times as variable values are modified during program execution), and somewhat less than half of commercial microcontrollers also have a third special kind of memory called EEPROM, which is non-volatile, but may be modified thousands of times for semi-permanent data storage that persists after power has been removed. So EEPROM is an important kind of memory that microcontrollers can access to keep certain types of data for long periods of time, such as unique identifiers, updatable calibration data, pass codes, and semi-permanent records of events that happen during program execution. But EEPROM is expensive, so typical microcontrollers have either none, or just a few hundred bytes of EEPROM. Typical microcontrollers with EEPROM might have only 63, 128, 256, or 512 Bytes of EEPROM memory available.

READING ASSIGNMENT

https://en.wikipedia.org/wiki/Computer_memory

https://en.wikipedia.org/wiki/EEPROM

https://en.wikipedia.org/wiki/Secure_Digital

PIC16F829 DATASHEET, section 11.2: Using the Data EEPROM

Read about how to write data to and read data from the internal EEPROM:

Read these sections of the CCS C User Manual: write_eeprom(), read_eeprom()

Read about setting up your microcontroller to send serial data as RS232 in the CCS C User Manual:

Sections to read: RS232 I/O, #use rs232(),

Then read about the SD Card module we will use in this lab (SparkFun OpenLog):

https://www.sparkfun.com/products/9530

Read the following page from the CCS C User Manual: ***How can I use two or more RS-232 ports on one PIC?***

This is near the end of the User Manual, in the Commonly Asked Questions & Answers section

QUESTIONS FROM THE READINGS:

1- How much EEPROM memory is available on your microcontroller (PIC16F1829)?
PIC16F1829 microcontrollers have 256 bytes of data EEPROM.

2- Total number of Erase/Write cycles for the EEPROM memory? (also called EEPROM endurance)
Modern EEPROMs can execute approximately 1 million Erase/Write cycles during their lifetime, but specifically for the PIC16F1829, the EEPROM will last for a minimum of 100 thousand Erase/Write cycles.

BME 580 Microcontrollers I

3- What is the minimum number of wires that need to be connected to the SparkFun OpenLog to be able to write data to the SD Card module? List each connection and its function.

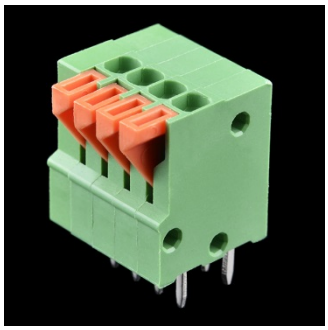
To be able to write data to the OpenLog SD Card module, a minimum of 3 wires need to be connected to the device. The connections are VCC -> +5 V (power), Ground -> 0 V (reference voltage), RXI -> RC5 (OpenLog signal receive).

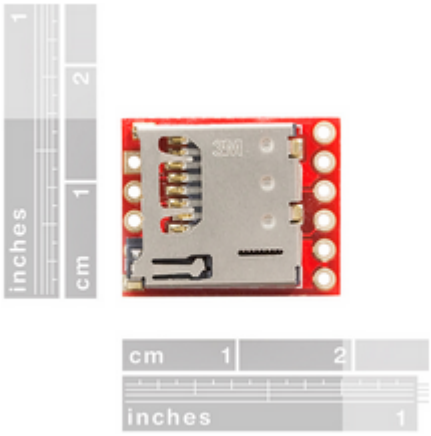
BME 580 Microcontrollers I

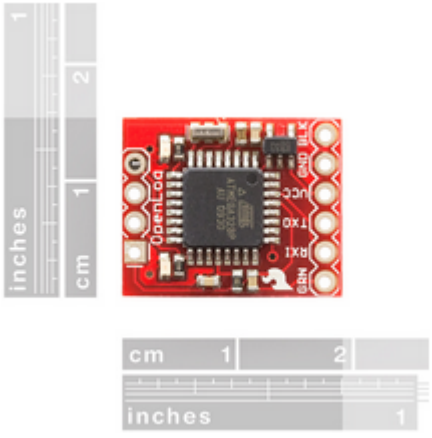
Supplies you will need for this lab:

You will be issued one micro SD Card, one OpenLog SD Card module from Sparkfun, and one 4-position spring terminal. The micro SD card fits into a larger full size SD Card as shown below. You may also be issued a USB-Micro SD Card reader if you do not own one. All of these items must be returned to the instructor at the end of the term before you will be issued a final grade.

BE SURE TO RETURN THE OPENLOG SD CARD MODULE AT THE END OF THE TERM (this is required to get credit).





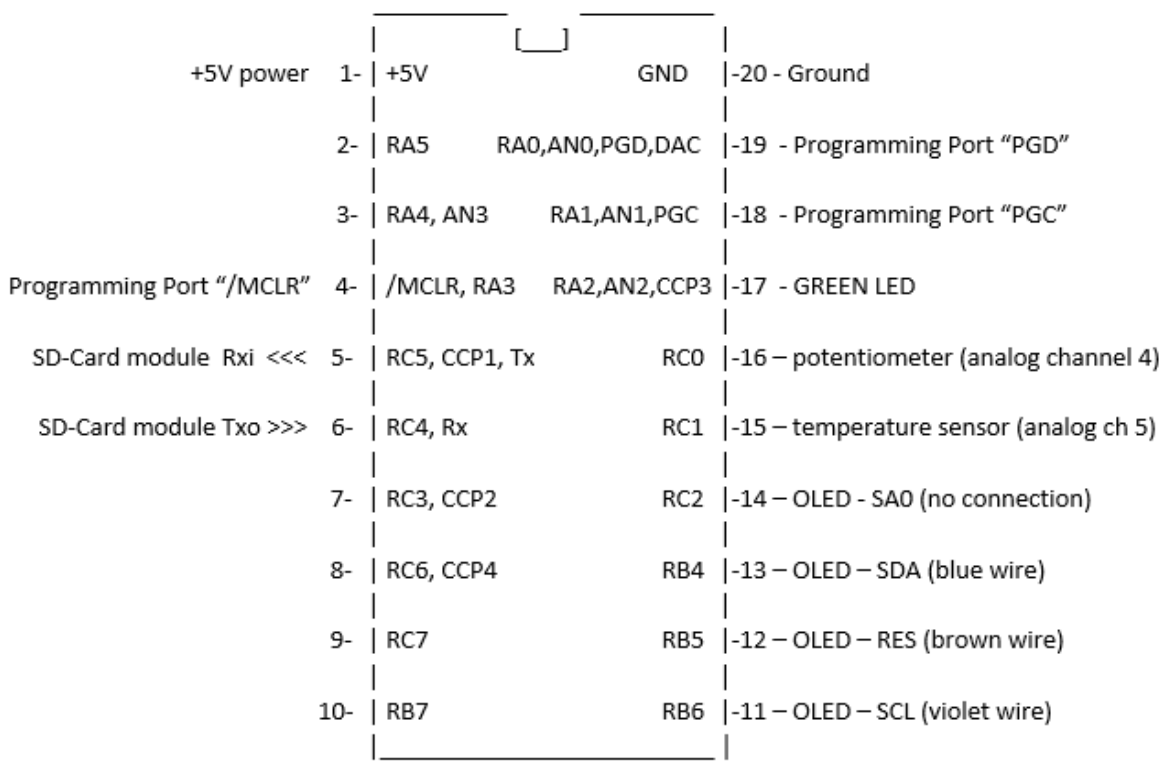


BME 580 Microcontrollers I

Microcontroller Connections:

Building upon your earlier circuits, add a 4-position spring terminal to your PCB to allow you to connect to the SparkFun OpenLog SD Card module.

PIC16F1829 8SOIC Pinout:



I have pre-soldered colored wires onto the SD Card module.

The color code for the wires on the SD Card module are:

GREEN = ground

YELLOW = +5V

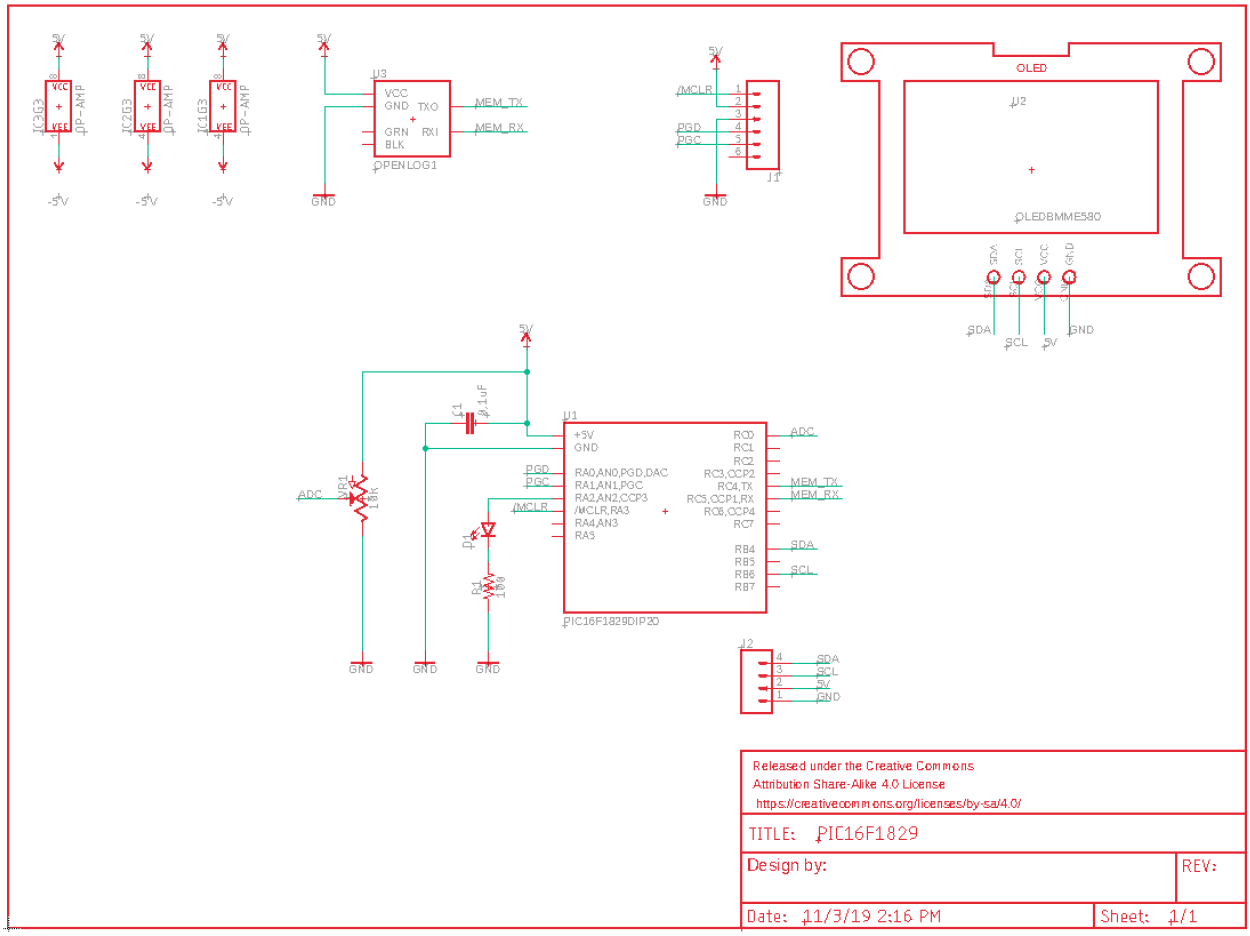
BLUE = RX-I on the OpenLog, pin #6 on the microcontroller (serial Tx: transmit data to SD Card)

GREY = TX-O on the OpenLog, pin #5 on the microcontroller (serial Rx: read data from SD Card)

Then modify your EagleCAD schematic to include the new component (OpenLog SD Card module).

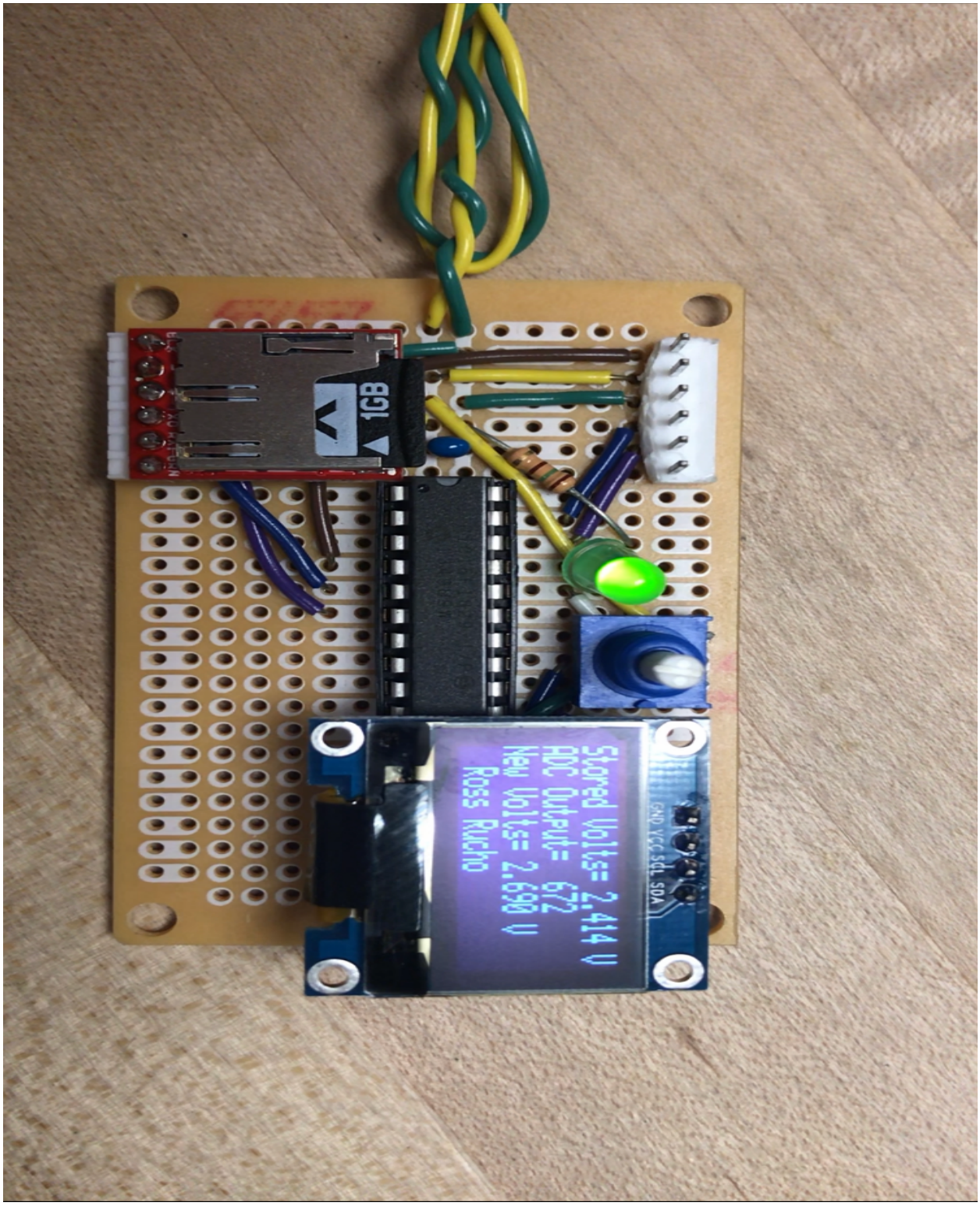
BME 580 Microcontrollers I

Your EagleCAD version of the modified circuit schematic with OpenLog SD Card:



BME 580 Microcontrollers I

Photograph of your completed circuit:



BME 580 Microcontrollers I

FIRMWARE:
Before you change your firmware, check to make sure your OLED display is still working properly using the firmware you installed for Lab #3. Also make sure that you can still read and display the voltage on the potentiometer as it changes when you rotate the potentiometer shaft, from Lab #4.

For this lab you will be converting the ADC value (an integer) into a standard length floating point number, and then working with those numbers.

USING internal memory: EEPROM
You should be able to use the CCS C User Manual to figure out how to write data to the internal EEPROM simply by using the commands read_eeprom() and write_eeprom(). Be sure to read about these commands in the CCS C User Manual.

Modify your source code from Lab #4 (ADC) to write the most recent voltage value to internal EEPROM. Do this once every second or so. Store the voltage in the form of a 16-bit integer, not as a floating point number. You have to do the conversion anyway, so it should not be a problem to just store the raw ADC reading as 16-bit integer data to the internal EEPROM. Even though the ADC only generates 10-bit data, storing it as 16-bit integer is relatively efficient. You will need to convert it to a floating point number for display on the OLED, but storing it as an integer in memory is much more efficient than storing it as a floating point.

Write your new source code such that before you turn OFF the power the last value displayed on your OLED should also be the last value you stored on the EEPROM. Use only one EEPROM memory location (using 2 bytes to store the full 16 bits of data) so that you just keep writing over the same location in memory with the new voltage value every time.

HINT: To store 16-bit integers in EEPROM you need to break the number into two halves, a lower byte (8-bits) and an upper byte (8-bits). CCS C has a simple command that allows you to do this easily:

```
make8();
```

You need to use this command twice to get the top and bottom bytes from the 16-bit number:

```
8_bit_lower_byte = make8(16_bit_integer_variable, 0);   // this returns the lower byte
8_bit_upper_byte = make8(16_bit_integer_variable, 1);   // this returns the upper byte
```

Then you store each byte in one location in EEPROM memory using the command:

```
write_eeprom();
```

Later, when you need to recover the data from EEPROM and restore the 16-bit number, you use the following command to retrieve each byte:

```
read_eeprom();
```

And the following command to re-build the 16-bit integer from the two 8-bit integers:

```
make16();
```

BME 580 Microcontrollers I

Look up both of these commands in the CCS C Compiler User Manual to see exactly how they are used. Verify that you are storing then retrieving the 16-bit numbers into EEPROM memory by displaying them before you break them apart to store them to EEPROM, then again after you have recovered the stored values and rebuilt the original 16-bit number.

You should be able to do this with a few simple lines of code just to make sure it is all working.

There are several other ways to convert 16-bit numbers into 8-bit numbers and vice versa, including bitwise operations, left shift and left shift operations, and integer math, but the makeX(); commands make this very simple to code.

Then change your source code to modify the text you display on the OLED so that each time power is applied (at start-up) your OLED permanently displays a line that says something like: "stored volts = x.xxxV"

Do not update this displayed value: it should stay the same while the new voltage you read from the ADC changes as you turn the shaft on the potentiometer. On a separate line on the OLED, display the new updated voltages once every second. Display something like: "new volts = x.xxxV". Keep updating the most recent voltage to both the OLED and the EEPROM, so that the next time you power up the circuit the OLED will display the last stored voltage from the previous power-up.

Now remove power from your circuit, then after a few seconds turn the power back on. Verify that the OLED displays the last voltage reading from your previous power-up. If the last voltage reading you saw updated the last time you used the device was 3.172 volts, then when you power up the device again the value "3.172 V" should be displayed as the last stored value from the most recent session before this one. This proves that you know how to store and retrieve data from the internal EEPROM.

BME 580 Microcontrollers I

Your source code using the internal EEPROM and displaying on the OLED:

```
#include <16F1829.H>
#device ADC=10               // Preprocessor directive to define the size of the ADC return value

// #include <stdlib.h>


//#device adc=8

#use delay (clock=8000000)    //  8 MHz, 0.5 us instruction cycle, this MUST come before the #include <oled_16F1829_i2c.h> to define delays in header file such as "delay_us(1)"
//   #use fast_io (A)         //

#include <oled_16F1829_i2c_Edit.h>

#fuses WDT, INTRC_IO, PUT     // internal oscillator operation with RA6 and RA7 as digital I/O, ref 16F1829.h, enable Power Up Timer
#fuses NOMCLR                 // disable /MCLR pin
#fuses NOPROTECT              // disable code protection
// #fuses NOLVP

// Bob's pin definitions for PIC16F1829 for I2C communication with OLED are in oled_16F1829_i2c.h

#define led pin_a2          // BE SURE TO REDEFINE THE LED PIN TO MATCH YOUR CIRCUIT

unsigned int i = 0;
unsigned int j = 0;

float t = 0;                // floating point variable for "temperature"
float v = 0;                // floating point variable for "voltage"
float vE = 0;               // floating point variable for "voltage" stored in EEPROM

// Ross' definitions for PIC16F1829 for ADC setup and usage
#define analog4 4           // define channel 4 of the ADC to be analog4
#define max_voltage 4.096   // define the maximum voltage to be Vss (This needs to be chagned if the voltage range changes)
#define num_states 1024     // Number of states for the given bit depth "ADC=8" (This needs to be changed if the bit depth changes)

unsigned int16 ADCout;      // 16 bit integer value for storing ADC output
unsigned int16 ADCoutE;     // 16 bit integer value for storing ADC output from EEPROM
unsigned int8 lower_byte;   // 8 bit integer value for storing lower 8 bits of ADCout
unsigned int8 upper_byte;   // 8 bit integer value for storing upper 8 bits of ADCout

////////////////////////////////////////
// SUBROUTINES

void init_ports(void) {

   SETUP_OSCILLATOR(OSC_8MHZ|OSC_INTRC);
   SETUP_WDT(WDT_8S);                        //  WDT set to 8 second period

   // Setup the ADC
   setup_adc(ADC_CLOCK_INTERNAL);            // ADC uses the internal clock
   setup_adc_ports(sAN4, VSS_FVR);           // ADC sets all ports to ANALOG and specifies the range to be 0 V to FVR
   set_adc_channel(analog4);                 // ADC reads from channel analog4
   delay_us(10);                             // A short delay is required after every time you change ADC channels

   // Setup the reference voltage
   setup_vref(VREF_ON | VREF_ADC_4v096);     // FVR is turned on and set to ADC 4.096 V
}



////////////////////////////////////////
// PIC12F1829 goes here at RESET

void main() {

   init_ports();                //  Initialize ports
   restart_wdt();
   delay_ms(50);                //  insert a short delay before starting the system up

   // Read the ADC value from EEPROM
   lower_byte = read_eeprom(0x0);
   upper_byte = read_eeprom(0x1);

   // Join the upper and lower bytes of ADC
   ADCoutE = make16(upper_byte, lower_byte);

   // Convert the last stored ADC value to floating point
   vE = ADCoutE * (max_voltage / (num_states-1) ); // Calculate the analog voltage

//  the "bit bang" code below and in the *.h file to emulate I2C
//  to run the OLED Display was developed by Kenny Donnelly
```

```
output_low(res);
   delay_ms(10);
   output_high(res);
   delay_ms(10);

initialise_screen();
   delay_ms(10);
   clear_screen();
   delay_ms(10);
   fill_screen();
   delay_ms(10);
   clear_screen();
   delay_ms(10);


CYCLE:

   i = 0;               //   counter for testing "moving text"
   clear_screen();      //   clear OLED screen

   while(true){

      // Control ADC to only read a new value at about 1 Hz
       ADCout = read_adc();

      // Split the ADC value into upper and lower bytes
      lower_byte = make8(ADCout, 0);
      upper_byte = make8(ADCout, 1);

      // Write the ADC value to EEPROM
      write_eeprom(0x0, lower_byte);
      write_eeprom(0x1, upper_byte);

      v = ADCout * (max_voltage / (num_states-1) ); // Calculate the analog voltage
      delay_ms(800);



      restart_wdt();

  oled_write_command(0xb0);
  oled_write_command(0x00);
  oled_write_command(0x10);

//  uncomment the following line to zoom in to text (top four lines of text only)
   oled_zoom();


//  the following command places the next character on line 0 (the 8 text lines are numbered 0 to 7)
//  and column 42 (there are 128 pixels from left to right on this 128 x 64 pixel display)
//  note that text character placement on the OLED display is therefore:
//       128 columns (individual pixels) numbered 0 to 127
//         8 lines, numbered 0 to 7, where each line is one character tall
//           each character is 8 pixels tall
//           8 x 8 = 64 pixels = the full display height
//  By placing the next character at this location the text "BMME580" is centered at the top
//  question: how many characters fit into the 128 characters across the OLED display

 oled_gotoxy(0,0);
 printf(oled_printchar, "Stored Volts = %2.3f ", vE);     // Displays the last recorded voltage from the previous session

 printf(oled_printchar, "V");

 oled_gotoxy(1,0);
 printf(oled_printchar, "ADC Output = %4Lu ", ADCout);  // Displays the digital representation of the voltage

 //oled_gotoxy(1,0);
 //printf(oled_printchar, "ADC Output = %4Lu ", ADCoutE); // Displays the digital representation of the voltage from EEPROM

 oled_gotoxy(2,0);
 printf(oled_printchar, "New Volts = %2.3f ", v);   //  see the PIC_C User Manual under "printf()" to format numbers in the display

 printf(oled_printchar, "V");     // notice how the "V" displays directly after the last printed character
```

BME 580 Microcontrollers I

```
// oled_gotoxy(2,0);
//printf(oled_printchar,"temperature = %1.1f $F", t);
//  look in the oled_16F1829_i2c.h file to see why the "$" character actually
//  displays a the temperature degree symbol "∞"
//  answer: I "remapped" the pixels in the BYTE const TABLE5 for "$".
//  So I changed the ASCII Character Table for the OLED Display in the *.h file

  oled_gotoxy(3,0);

  for (j=0; j<i; j++) printf(oled_printchar," ");


printf(oled_printchar,"Ross Rucho");


     OUTPUT_HIGH(LED);   //  RA5 - turn ON green LED
        delay_ms(100);
     OUTPUT_LOW(LED);    //  RA5 - turn OFF green LED
        delay_ms(100);

  t = t + 1.2;      //  increment the "temperature" variable
  i = i + 1;        //  nudge test to the right one character
  if(i>22) {
     i = 0;   // prevent moving text from scrolling down to next row on OLED display
     clear_screen();      // clear OLED screen
  }
}
     restart_wdt();

goto CYCLE;  //  cycle indefinitely until battery is removed or battery runs out of power
} // main
```

BME 580 Microcontrollers I

Using External Mass Storage Memory: SD Card

In this lab you will do something a bit complicated. The microcontroller will communicate with two different serial interfaces. One of them is a bit-bang hack version of an I2C interface, which goes to the OLED. The other is a standard RS-232 interface that will send data to and read data from the OpenLog SD Card module.

A good place to start with example code for using two serial interfaces is from the CCS C User Manual: *How can I use two or more RS-232 ports on one PIC?* This was in the Reading List at the beginning of this laboratory.

To begin with I suggest you just make sure the OpenLog module is working properly and ignore the OLED and the ADC for now. To do this I suggest you temporarily comment out the OLED code from your earlier source code and just use a short bit of code to stream RS-232 data to the OpenLog as a quick test.

Below I have included a snippet of example code that I have used for streaming data to the OpenLog module. This is set to the default serial parameters for use with both OpenLog SDCard (SparkFun):

```
#use rs232(baud=9600, xmit=PIN_C5, rcv=PIN_C4)
  delay_ms(20);                    //  allow slight delay for configuration of RS232 SPIO pins
printf("BMME 580");
puts( "" );                        // adds a reliable carriage return for OpenLog data streams
printf("Ross Rucho ");
puts( "" );                        // adds a reliable carriage return for OpenLog data streams
printf("Data Logging from PIC16F1829 to SparkFun OpenLog");
puts( "" );                        // adds a reliable carriage return for OpenLog data streams
```

then you need to write a line of code to "printf" each data point as a string to the SD card module

Write some fake data to the SD Card, then remove the SD Card from the OpenLog, insert it into your computer (you may need an SD Card reader), and check to see if a new file was created on the SD Card. Open the new file with a text editor to make sure that the file contains the simple data strings you sent to the SD Card.

Once you are sure that the OpenLog SD Card Module is working properly, write source code to stream your analog-to-digital data from the potentiometer to the SD Card. Modify your source code so that each ADC voltage data point is written to a new line in the file on the SD Card. The syntax for doing this should be quite similar to the syntax that you used to write text and variables to the OLED, except that you do not need to invoke the command that uses the OLED header file. This time you are just sending standard ASCII symbols to the SD card, not a stream of 5 hexadecimal numbers that you used to generate symbols on the graphic OLED display.

Then power OFF your circuit and turn the potentiometer back and forth for 10 or 20 seconds.

Shut off the power, remove the micro SD card, insert the card into your computer, and read the new text file that was generated.

BME 580 Microcontrollers I

If you put the micro SD card back into the OpenLog module, then run your circuit again, what happens?

*When I put the micro SD card back into the OpenLog module and run my circuit again, a new *.txt file is created and all data from the current trial is sent to this new file.*

Now, keep the power ON for your circuit while rotating the potentiometer. While your circuit is running, pull the SD Card out for a few seconds, then reinsert it. Do this (carefully) several times while your circuit is powered up and running. What does the resulting data look like?

*Each time the micro SD card is removed and reinserted, a new *.txt file is created and the data that is currently being generated is saved to that most recent *.txt file. The result was data for the entire trial being saved on multiple *.txt files. Additionally, none of the data generated was saved when the micro SD card was removed. This happened because the microcontroller that is located on the OpenLog module is programmed to generate a new *.txt file whenever the SD card is inserted while the power is on or whenever the power is turned on while the SD card is inserted. Finally, the OpenLog module microcontroller saves any received data on the most recent *.txt file.*

BME 580 Microcontrollers I

Your source code that streams ADC voltage data to an SD Card:

```
#include <16F1829.H>
#device ADC=10               // Preprocessor directive to define the size of the ADC return value

// #include <stdlib.h>


//#device adc=8

#use delay (clock=8000000)    //  8 MHz, 0.5 us instruction cycle, this MUST come before the #include <oled_16F1829_i2c.h> to define delays in header file such as "delay_us(1)"
//   #use fast_io (A)

#use rs232(baud=9600, xmit=PIN_C5, rcv=PIN_C4, stream=COM_MEM)

#include <oled_16F1829_i2c_Edit.h>

#fuses WDT, INTRC_IO, PUT     // internal oscillator operation with RA6 and RA7 as digital I/O, ref 16F1829.h, enable Power Up Timer
#fuses NOMCLR                 // disable /MCLR pin
#fuses NOPROTECT              // disable code protection
// #fuses NOLVP

// Bob's pin definitions for PIC16F1829 for I2C communication with OLED are in oled_16F1829_i2c.h

#define led pin_a2          // BE SURE TO REDEFINE THE LED PIN TO MATCH YOUR CIRCUIT

unsigned int i = 0;
unsigned int j = 0;

float t = 0;                // floating point variable for "temperature"
float v = 0;                // floating point variable for "voltage"
float vE = 0;               // floating point variable for "voltage" stored in EEPROM

// Ross' definitions for PIC16F1829 for ADC setup and usage
#define analog4 4           // define channel 4 of the ADC to be analog4
#define max_voltage 4.096   // define the maximum voltage to be Vss (This needs to be chagned if the voltage range changes)
#define num_states 1024     // Number of states for the given bit depth "ADC=8" (This needs to be changed if the bit depth changes)

unsigned int16 ADCout;      // 16 bit integer value for storing ADC output
unsigned int16 ADCoutE;     // 16 bit integer value for storing ADC output from EEPROM
unsigned int8 lower_byte;   // 8 bit integer value for storing lower 8 bits of ADCout
unsigned int8 upper_byte;   // 8 bit integer value for storing upper 8 bits of ADCout

////////////////////////////////////////
// SUBROUTINES

void init_ports(void) {

   SETUP_OSCILLATOR(OSC_8MHZ|OSC_INTRC);
   SETUP_WDT(WDT_8S);                        //  WDT set to 8 second period

   // Setup the ADC
   setup_adc(ADC_CLOCK_INTERNAL);            // ADC uses the internal clock
   setup_adc_ports(sAN4, VSS_FVR);           // ADC sets all ports to ANALOG and specifies the range to be 0 V to FVR
   set_adc_channel(analog4);                 // ADC reads from channel analog4
   delay_us(10);                             // A short delay is required after every time you change ADC channels

   // Setup the reference voltage
   setup_vref(VREF_ON | VREF_ADC_4v096);     // FVR is turned on and set to ADC 4.096 V
}

////////////////////////////////////////
// PIC12F1829 goes here at RESET

void main()
{
init_ports();               //  Initialize ports
restart_wdt();
delay_ms(50);               //  insert a short delay before starting the system up

// Read the ADC value from EEPROM
lower_byte = read_eeprom(0x0);
upper_byte = read_eeprom(0x1);

// Join the upper and lower bytes of ADC
ADCoutE = make16(upper_byte, lower_byte);

// Convert the last stored ADC value to floating point
vE = ADCoutE * (max_voltage / (num_states-1) ); // Calculate the analog voltage
```

BME 580 Microcontrollers I

```
fprintf(COM_MEM, "BMME 580");
fputs("", COM_MEM);  // Adds a reliable carriage return for OpenLog data streams
fprintf(COM_MEM, "Ross Rucho");
fputs("", COM_MEM);  // Adds a reliable carriage return for OpenLog data streams
fprintf(COM_MEM, "Data Logging from PIC16F1829 to SparkFun OpenLog");
fputs("", COM_MEM);  // Adds a reliable carriage return for OpenLog data streams

// the "bit bang" code below and in the *.h file to emulate I2C
// to run the OLED Display was developed by Kenny Donnelly

output_low(res);
  delay_ms(10);
  output_high(res);
  delay_ms(10);

initialise_screen();
  delay_ms(10);
  clear_screen();
  delay_ms(10);
  fill_screen();
  delay_ms(10);
  clear_screen();
  delay_ms(10);


CYCLE:

i = 0;  // counter for testing "moving text"
clear_screen();  // clear OLED screen

while(true)
{
  // Control ADC to only read a new value at about 1 Hz
  ADCout = read_adc();

  // Split the ADC value into upper and lower bytes
  lower_byte = make8(ADCout, 0);
  upper_byte = make8(ADCout, 1);

  // Write the ADC value to EEPROM
  write_eeprom(0x0, lower_byte);
  write_eeprom(0x1, upper_byte);

  // Write the ADC value to external memory
  fprintf(COM_MEM, "%4Lu", ADCout);
  fputs("", COM_MEM);  // Adds a reliable carriage return for OpenLog data streams

  v = ADCout * (max_voltage / (num_states-1) ); // Calculate the analog voltage
  delay_ms(800);

  restart_wdt();

  oled_write_command(0xb0);
  oled_write_command(0x00);
  oled_write_command(0x10);

  // uncomment the following line to zoom in to text (top four lines of text only)
  oled_zoom();


  // the following command places the next character on line 0 (the 8 text lines are numbered 0 to 7)
  // and column 42 (there are 128 pixels from left to right on this 128 x 64 pixel display)
  // note that text character placement on the OLED display is therefore:
  //       128 columns (individual pixels) numbered 0 to 127
  //         8 lines, numbered 0 to 7, where each line is one character tall
  //           each character is 8 pixels tall
  //           8 x 8 = 64 pixels = the full display height
  // By placing the next character at this location the text "BMME580" is centered at the top
  // question: how many characters fit into the 128 characters across the OLED display

  oled_gotoxy(0,0);
  printf(oled_printchar, "Stored Volts= %2.3f ", vE);  // Displays the last recorded voltage from the previous session

  printf(oled_printchar, "V");

  oled_gotoxy(1,0);
  printf(oled_printchar, "ADC Output= %4Lu ", ADCout);  // Displays the digital representation of the voltage

  //oled_gotoxy(1,0);
  //printf(oled_printchar, "ADC Output = %4Lu ", ADCoutE);  // Displays the digital representation of the voltage from EEPROM
```

BME 580 Microcontrollers I

```
  oled_gotoxy(2,0);
  printf(oled_printchar, "New Volts= %2.3f ", v);  // see the PIC_C User Manual under "printf()" to format numbers in the display
  printf(oled_printchar, "V");  // notice how the "V" displays directly after the last printed character

  //oled_gotoxy(2,0);
  //printf(oled_printchar,"temperature = %1.1f $F", t);

  // look in the oled_16F1829_i2c.h file to see why the "$" character actually
  // displays a the temperature degree symbol "∞"
  // answer: I "remapped" the pixels in the BYTE const TABLE5 for "$".
  // So I changed the ASCII Character Table for the OLED Display in the *.h file

  oled_gotoxy(3,0);

  for (j=0; j<i; j++) printf(oled_printchar," ");

  printf(oled_printchar,"Ross Rucho");


  OUTPUT_HIGH(LED);  // RA5 - turn ON green LED
  delay_ms(100);
  OUTPUT_LOW(LED);  // RA5 - turn OFF green LED
  delay_ms(100);

  t = t + 1.2;  // increment the "temperature" variable
  i = i + 1;  // nudge test to the right one character

  if(i>22)
  {
    i = 0;  // prevent moving text from scrolling down to next row on OLED display
    clear_screen();  // clear OLED screen
  }
}

restart_wdt();

goto CYCLE;  //  cycle indefinitely until battery is removed or battery runs out of power
} // main
```

BME 580 Microcontrollers I

Run your circuit for a few minutes to generate a good set of data, then paste your data set (the resulting text file on the SD card) here:

2.410	*0.004*	*2.690*	*2.690*	*1.941*
2.406	*0.004*	*2.690*	*2.694*	*1.657*
2.422	*0.004*	*2.690*	*2.694*	*1.381*
2.966	*0.056*	*2.690*	*2.690*	*1.153*
3.275	*0.208*	*2.690*	*2.694*	*0.896*
3.627	*0.364*	*2.690*	*2.690*	*0.604*
4.003	*0.600*	*2.694*	*2.690*	*0.372*
4.096	*0.796*	*2.694*	*2.818*	*0.016*
4.096	*1.081*	*2.694*	*3.171*	*0.004*
4.096	*1.217*	*2.690*	*3.551*	*0.004*
4.096	*1.573*	*2.690*	*3.967*	*0.004*
3.999	*1.909*	*2.690*	*4.096*	*0.004*
3.995	*2.138*	*2.690*	*4.096*	*0.004*
3.795	*2.522*	*2.690*	*4.096*	*0.116*
3.283	*2.706*	*2.690*	*4.096*	*0.484*
2.822	*2.926*	*2.690*	*4.096*	*0.960*
2.414	*3.435*	*2.690*	*3.707*	*1.541*
2.033	*4.096*	*2.690*	*3.331*	*1.849*
1.781	*4.096*	*2.690*	*3.030*	*2.450*
1.441	*4.051*	*2.690*	*2.954*	*2.454*
1.077	*4.075*	*2.694*	*2.666*	*2.446*
0.644	*4.096*	*2.690*	*2.574*	*2.446*
0.300	*3.575*	*2.694*	*2.234*	
0.004	*2.690*	*2.690*	*1.965*	

BME 580 Microcontrollers I

Using two serial communication ports simultaneously

Now, using the sample code from the CCS C User Manual: *How can I use two or more RS-232 ports on one PIC?*, figure out how to stream the voltage data to the SD card while simultaneously saving the most recent previous value to internal EEPROM and displaying on the OLED the last saved voltage data (from EEPROM) and the new voltage data that is generated every second. This is just like the data you displayed on the OLED that was stored on EEPROM from the beginning of this lab, but now you are also streaming each data point to an SD Card for permanent storage and later analysis on a separate computer. The EEPROM only has enough memory for a couple of hundred data points, the SD Card has much more capacity than the EEPROM.

QUESTIONS:

Read through the CCS C User Manual to determine how much memory is required (bytes) for a standard floating point variable. Read the PIC16F1829 Datasheet to determine the memory capacity of the internal EEPROM memory (bytes). Look at the microSD Card (or open the SD Card's "properties" on a computer) to determine its memory capacity. For these questions assume you store every ADC value in both the internal EEPROM and on the SD Card.

Capacity of internal EEPROM (bytes): The internal EEPROM has a capacity of 256 bytes.

Capacity of the SD Card (bytes): The microSD Card has a capacity of 1 billion bytes (1 GB).

Then compare storage capacity between the two memory types:

Storing your data efficiently on internal EEPROM as integers, how many 16-bit data values could you store internally in EEPROM? *EEPROM can store 128 16-bit data values.*

Storing the same data as strings of characters on the SD card (each character is a single ASCII 8-bit value), how many data values could you store on the SD Card? *The microSD Card can store 500 million 16-bit data values.*

Think about memory capacity in terms of time. If you were storing one value point every second…

How long could you collect data before the internal EEPROM memory was full? *I could collect data for 2 minutes and 8 seconds (128 seconds).*

How long could you collect data before the SD Card memory was full? *I could collect data for 15 years 312 days 53 minutes 20 seconds (500 million seconds).*

BME 580 Microcontrollers I

Your source code that streams ADC voltage data to an SD Card and the OLED:

```
#include <16F1829.H>
#device ADC=10               // Preprocessor directive to define the size of the ADC return value

// #include <stdlib.h>


//#device adc=8

#use delay (clock=8000000)    //  8 MHz, 0.5 us instruction cycle, this MUST come before the #include <oled_16F1829_i2c.h> to define delays in header file such as "delay_us(1)"
//   #use fast_io (A)

#use rs232(baud=9600, xmit=PIN_C5, rcv=PIN_C4, stream=COM_MEM)

#include <oled_16F1829_i2c_Edit.h>

#fuses WDT, INTRC_IO, PUT     // internal oscillator operation with RA6 and RA7 as digital I/O, ref 16F1829.h, enable Power Up Timer
#fuses NOMCLR                 // disable /MCLR pin
#fuses NOPROTECT              // disable code protection
// #fuses NOLVP

// Bob's pin definitions for PIC16F1829 for I2C communication with OLED are in oled_16F1829_i2c.h

#define led pin_a2          // BE SURE TO REDEFINE THE LED PIN TO MATCH YOUR CIRCUIT

unsigned int i = 0;
unsigned int j = 0;

float t = 0;                // floating point variable for "temperature"
float v = 0;                // floating point variable for "voltage"
float vE = 0;               // floating point variable for "voltage" stored in EEPROM

// Ross' definitions for PIC16F1829 for ADC setup and usage
#define analog4 4           // define channel 4 of the ADC to be analog4
#define max_voltage 4.096   // define the maximum voltage to be Vss (This needs to be chagned if the voltage range changes)
#define num_states 1024     // Number of states for the given bit depth "ADC=8" (This needs to be changed if the bit depth changes)

unsigned int16 ADCout;      // 16 bit integer value for storing ADC output
unsigned int16 ADCoutE;     // 16 bit integer value for storing ADC output from EEPROM
unsigned int8 lower_byte;   // 8 bit integer value for storing lower 8 bits of ADCout
unsigned int8 upper_byte;   // 8 bit integer value for storing upper 8 bits of ADCout

////////////////////////////////////////
// SUBROUTINES

void init_ports(void) {

   SETUP_OSCILLATOR(OSC_8MHZ|OSC_INTRC);
   SETUP_WDT(WDT_8S);                        //  WDT set to 8 second period

   // Setup the ADC
   setup_adc(ADC_CLOCK_INTERNAL);            // ADC uses the internal clock
   setup_adc_ports(sAN4, VSS_FVR);           // ADC sets all ports to ANALOG and specifies the range to be 0 V to FVR
   set_adc_channel(analog4);                 // ADC reads from channel analog4
   delay_us(10);                             // A short delay is required after every time you change ADC channels

   // Setup the reference voltage
   setup_vref(VREF_ON | VREF_ADC_4v096);     // FVR is turned on and set to ADC 4.096 V
}

////////////////////////////////////////
// PIC12F1829 goes here at RESET

void main()
{
init_ports();               //  Initialize ports
restart_wdt();
delay_ms(50);               //  insert a short delay before starting the system up

// Read the ADC value from EEPROM
lower_byte = read_eeprom(0x0);
upper_byte = read_eeprom(0x1);

// Join the upper and lower bytes of ADC
ADCoutE = make16(upper_byte, lower_byte);

// Convert the last stored ADC value to floating point
vE = ADCoutE * (max_voltage / (num_states-1) ); // Calculate the analog voltage
```

BME 580 Microcontrollers I

```
fprintf(COM_MEM, "BMME 580");
fputs("", COM_MEM);  // Adds a reliable carriage return for OpenLog data streams
fprintf(COM_MEM, "Ross Rucho");
fputs("", COM_MEM);  // Adds a reliable carriage return for OpenLog data streams
fprintf(COM_MEM, "Data Logging from PIC16F1829 to SparkFun OpenLog");
fputs("", COM_MEM);  // Adds a reliable carriage return for OpenLog data streams

// the "bit bang" code below and in the *.h file to emulate I2C
// to run the OLED Display was developed by Kenny Donnelly

output_low(res);
  delay_ms(10);
  output_high(res);
  delay_ms(10);

initialise_screen();
  delay_ms(10);
  clear_screen();
  delay_ms(10);
  fill_screen();
  delay_ms(10);
  clear_screen();
  delay_ms(10);


CYCLE:

i = 0;  // counter for testing "moving text"
clear_screen();  // clear OLED screen

while(true)
{
  // Control ADC to only read a new value at about 1 Hz
  ADCout = read_adc();

  // Split the ADC value into upper and lower bytes
  lower_byte = make8(ADCout, 0);
  upper_byte = make8(ADCout, 1);

  // Write the ADC value to EEPROM
  write_eeprom(0x0, lower_byte);
  write_eeprom(0x1, upper_byte);

  // Write the ADC value to external memory
  fprintf(COM_MEM, "%4Lu", ADCout);
  fputs("", COM_MEM);  // Adds a reliable carriage return for OpenLog data streams

  v = ADCout * (max_voltage / (num_states-1) ); // Calculate the analog voltage
  delay_ms(800);

  restart_wdt();

  oled_write_command(0xb0);
  oled_write_command(0x00);
  oled_write_command(0x10);

  // uncomment the following line to zoom in to text (top four lines of text only)
  oled_zoom();


  // the following command places the next character on line 0 (the 8 text lines are numbered 0 to 7)
  // and column 42 (there are 128 pixels from left to right on this 128 x 64 pixel display)
  // note that text character placement on the OLED display is therefore:
  //       128 columns (individual pixels) numbered 0 to 127
  //         8 lines, numbered 0 to 7, where each line is one character tall
  //           each character is 8 pixels tall
  //           8 x 8 = 64 pixels = the full display height
  // By placing the next character at this location the text "BMME580" is centered at the top
  // question: how many characters fit into the 128 characters across the OLED display

  oled_gotoxy(0,0);
  printf(oled_printchar, "Stored Volts= %2.3f ", vE);  // Displays the last recorded voltage from the previous session

  printf(oled_printchar, "V");

  oled_gotoxy(1,0);
  printf(oled_printchar, "ADC Output= %4Lu ", ADCout);  // Displays the digital representation of the voltage

  //oled_gotoxy(1,0);
  //printf(oled_printchar, "ADC Output = %4Lu ", ADCoutE);  // Displays the digital representation of the voltage from EEPROM
```

BME 580 Microcontrollers I

```
  oled_gotoxy(2,0);
  printf(oled_printchar, "New Volts= %2.3f ", v);  // see the PIC_C User Manual under "printf()" to format numbers in the display
  printf(oled_printchar, "V");  // notice how the "V" displays directly after the last printed character

  //oled_gotoxy(2,0);
  //printf(oled_printchar,"temperature = %1.1f $F", t);

  // look in the oled_16F1829_i2c.h file to see why the "$" character actually
  // displays a the temperature degree symbol "∞"
  // answer: I "remapped" the pixels in the BYTE const TABLE5 for "$".
  // So I changed the ASCII Character Table for the OLED Display in the *.h file

  oled_gotoxy(3,0);

  for (j=0; j<i; j++) printf(oled_printchar," ");

  printf(oled_printchar,"Ross Rucho");


  OUTPUT_HIGH(LED);  // RA5 - turn ON green LED
  delay_ms(100);
  OUTPUT_LOW(LED);  // RA5 - turn OFF green LED
  delay_ms(100);

  t = t + 1.2;  // increment the "temperature" variable
  i = i + 1;  // nudge test to the right one character

  if(i>22)
  {
    i = 0;  // prevent moving text from scrolling down to next row on OLED display
    clear_screen();  // clear OLED screen
  }
}

restart_wdt();

goto CYCLE;  //  cycle indefinitely until battery is removed or battery runs out of power
} // main
```

BME 580 Microcontrollers I

OPTIONAL EXERCISE: Master the SD Card Module

(that's right, you do not have to do this, but I suggest that you do it if you can)

NOTE: Most students find this very difficult or impossible. This exercise is optional.

If you really want to know how to make full use of the SD Card Module you need to do a bit more than just mindlessly streaming data to the module. The module is designed to allow you to do this with minimal code for simple bulk data storage, but let's say you want to have access to the resulting data during operation of your system, while the data is on the SD Card, for use by your microcontroller during program execution.

Let's assume you will start by streaming data to the SD Card module ten times each second. After just a minute or so you would have too much data to store in the microcontroller memory. But what if your device required you to be able to go back and use the stored data? What if you wanted to *append* more data to the same file every time you began to store the data to the SD Card, instead of creating a new file every time?

FIRST: rewrite your source code to read the ADC and store data to the SD card ten times each second.

THEN: modify your code so that you update the OLED only once each second so that it is not flickering with updates too frequently. So, basically every second you will collect and store ten data points, but only display one of those data points on the OLED each second.

THEN, once you have 100 seconds worth of data, begin reading the values that have been stored on the SD Card. Read the 100 most recent values, calculate the average voltage value from that data, and display it on the OLED as: "average = X.xxx volts)

Update and display this average value once every 50 data points (once every 5 seconds)

THEN, modify your code to append your voltage data to the same text file every time you restart the system, rather than creating a new file automatically as the OpenLog SD Card module typically does. To figure out how to do this you will have to read the user manual for the OpenLog module (sparkfun.com)

After taking 100 data points, when you start reading values from the SD Card and calculating the average, do you notice that the microcontroller is beginning to take data less frequently than 10 times per second? If you just used firmware delays to set your timing you will find out that the timing gets screwed up when the computational demand on the microcontroller increases, in this case, when you begin accessing the SD Card, reading values, converting them from strings into floating point values, then doing floating point math, which is very computationally intensive for a simple microcontroller.

FINALLY, display a line of code at the bottom of the OLED that states:

Your first name: memory master

BME 580 Microcontrollers I

Your source code for the OPTIONAL EXERCISE:

Paste source code here