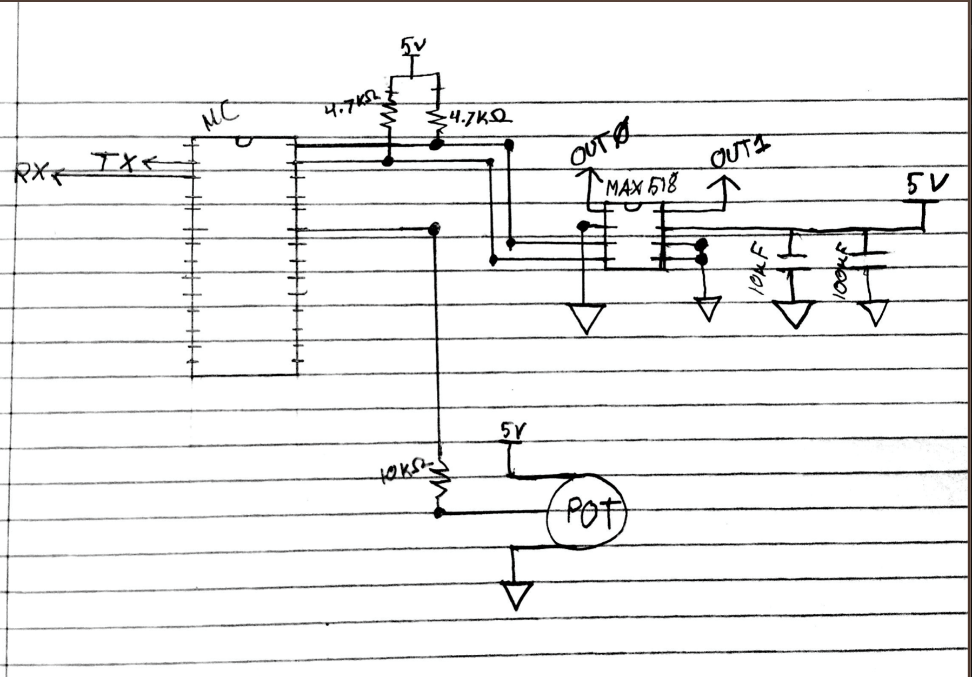
# Lab 5 – Remote Controllable Analog Data Logging System

## 1. Introduction

Our board provides a user with four primary commands to store voltage readings to EEPROM, as well as read them back out, and send them to a connected MAX518 DAC for output. We utilize the built-in A/D converter of the ATmega88PA controller to get voltage readings and communicate with the DAC over I2C serial communication.

## 2. Schematic



*Our Completed Schematic*

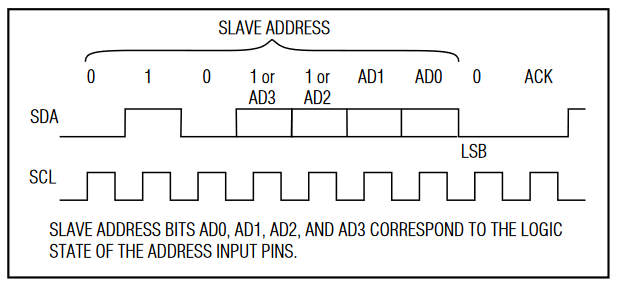
## 3. Discussion

The program begins by initializing a USART connection, configuring our ADC, and setting both DAC outputs. These tasks are isolated to respective functions for clarity.

*USART\_init();* configures the mode to “Async Normal Mode” by setting **UCSR0A** to 0x02, enables both transmission and reception by setting **RXEN0** and **TXEN0** in **UCSR0B**, and then assures a BAUD Rate of 9600 by setting **UBRR0** to 0x33. This represents 51d, which was calculated by dividing the clock rate of 8MHz by sixteen times the desired rate, as mandated by using the Async Normal Mode:

*ADC\_init();* enables the use of 5V as our voltage reference by setting the **REFS0** bit of **ADMUX**, and then sets **ADEN** in the **ADCSRA** in order to enable full functionality.

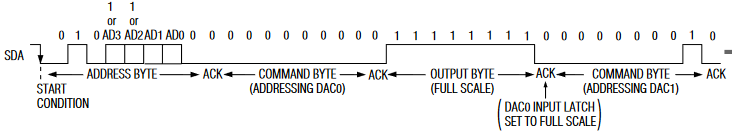
*write\_dac(float, int);* is used throughout our program in order to easily set both **OUT0** and **OUT1** of the MAX518, simply by providing a desired voltage and port preference. It utilizes the i2cmaster library[1] to send the slave address byte, command byte, and output byte in proper sequence. The slave address is set to 0x58 since there’s only one slave in use and **AD1** and **AD0** are grounded. This is shown in Figure 1 below.



*Fig 1: The slave address*

The command byte is set to either 0 or 1, as directly provided by the user from the second argument, which is validated for “bonus points” as mentioned in our command table below. The output byte is then calculated by the following equation:

By delivering these three bytes in order combined with the proper acknowledgment bits sent as needed from i2cmaster as shown in figure 2, the MAX518 can set voltages as needed.



*Fig. 2: Sequencing of byte delivery by i2cmaster to MAX518*

From this point onward, our main loop issues a prompt to the user, waiting for one of the following commands. As mentioned underneath the table, all arguments are validated, as well as refusal of any non-existing commands by evaluating the first character:

Our Commands:

|  |  |  |
| --- | --- | --- |
| Command | Function | Arguments |
| M | Get single voltage measurement ADC | No Arguments |
| S:a,n,t | Store ADC Measurements in EEPROM | a … start address (integer, 0≤ a ≤ 510)  n … number of measurements (integer, 1 ≤ n ≤ 20)  t ... time between measurements (integer, 1 ≤ dt ≤ 10 s) |
| R:a,n | Retrieve and display measurements from EEPROM | a … start address (integer, 0≤ a ≤ 510)  n … number of measurements (integer, 1 ≤ n ≤ 20) |
| E:a,n,t,d | Retrieve measurements from EEPROM and write results to DAC | a … start address (integer, 0≤ a ≤ 510)  n … number of measurements (integer, 1 ≤ n ≤ 20)  t ... time between measurements (integer, 1 ≤ dt ≤ 10 s)  d … DAC channel number (integer, d∈ {0,1})  Note that stored 10-bit ADC values must be converted to the closest 8-bit integer value such that the DAC quantization error is minimal. |

Our input validation includes bounds checking of individual variables through a generic function named check\_param, as well as EEPROM address validation. All commands are also validated to be within the appropriate range for each argument. When entering a command that’s out of range, the program will notify the user dynamically based on which argument was improper.

The user input is iterated through and placed character by character into a command array, from index two to the null terminating character ‘\0’. Characters are read in using the *read\_char\_from\_PC()*; method, which simply returns the contents of **UDR0** in sequence of waiting for the **RXC0** flag to be set, to insure proper polling, and avoid any **UDR** overwrite when switching between operations of transmission and reception. During this loop the received characters are loaded into a temporary parameter buffer until the loop reaches a comma, at which point the command is saved and the temporary buffer is reset. This pattern of saving off parameters continues until the null terminating character is reached. By this point we can begin validation of the parameters, before it is passed to its respective command calls, *execute\_M*, *execute\_R*, *execute\_S*, or *execute\_E*.

*execute\_M()* is the smallest execution function, as it requires no validation, and only utilizes the *read\_adc()* function as described previously. A small string is created in order to communicate the current voltage back to the user using the c function *sprintf* from the <avr/io.h> library. That string is then sent out character by character with the *print\_single\_line\_message()* which we created to elegantly print all characters from a provided string, with proper indentation through newline and carriage return escape characters.

In order to complete the (S)tore and (R)ead commands for lab five, we utilized that <avr/eeprom.h> library. This library drastically simplifies interfacing with the EEPROM registers. Because we represent our voltages as integers, we utilize the *eeprom\_write\_word(addr, val)* and *eeprom\_read\_word methods(addr)*. The following is what happens on the register level within each of the functions that we use.

In *eeprom\_write\_word(addr, val)*, we set the address bits that we want to write to inside of **EEARW**. The content we wish to write the needs to be stored into the EEPROM data register **EEDR**. Lastly, we set the master write enable bit **EEPME** and the write enable **EEPE** (within four clocks of each other) inside of the EEPROM control register **EECR**.

In the *eeprom\_read\_word(addr)* we again set the address bits that we want to read from inside of **EEARW**. We then toggle the EEPROM write enable bit **EERE**, inside of the EEPROM control register **EECR**.

Both *execute\_R()* and *execute\_S()* are simple for loops surrounding the *eeprom\_read\_word(addr)* and *eeprom\_write\_word(addr)* methods, in order to progress the memory addresses forward as needed.

*execute\_E()* utilizes the previously described *dac\_write()* to write the given voltage pulled from parameters, to the specified pin argument. Each of these interactions have been described in detail individually above, and *execute\_E()* adds them in combination, with a variable delay from *my\_delay()*, in order to leave the voltages set for the proper amount of times as specified.

## 4. Conclusion

In this lab, by using C programming over assembly, we were able to use many libraries in order to solve more complex problems faster. We grew much more comfortable in handling primitive types, including the conversions between them. We were happy with our methods of internal memory management and understanding cost, as in the end we never ran into memory constraints. The ability to create functions quickly and concisely allowed us to create more modular and reusable code. We’re looking forward to being able to use C in our upcoming final project.

## 5. Appendix A: Source Code

// ///////////////////////////////////////////////////////////////////

// Assembly language file for Lab 5 in ECE:3360 (Embedded Systems)

// Spring 2019, The University of Iowa.

//

// Desc: Lab 5 Analog Data Logging System

//

// Authors: B. Mitchinson, A. Powers

// ///////////////////////////////////////////////////////////////////

// Setting CPU Clock Speed

#ifndef *F\_CPU*

#define *F\_CPU* 8000000UL // 8 MHz -> CProgramming Notes, Slide 10

#endif

// ///////////////////////

// Required Imports

#include <avr/io.h>

#include <util/delay.h>

#include <stdio.h>

#include <avr/eeprom.h>

#include "i2cmaster.h"

// ////////////////

// Function Prototypes

// prints

void usart\_prints(const char \*);

void print\_new\_line(void);

void print\_single\_line\_message(const char \*);

// usart io

unsigned char read\_char\_from\_pc(void);

void usart\_init(void);

// using the adc

void adc\_init(void);

int read\_adc(void);

int check\_bounds(int);

// command interp

unsigned int read\_command(char \*, *size\_t*);

void interpret\_command(const char \*);

void my\_delay(int delay);

void execute\_M(void);

void execute\_R(int \*);

void execute\_S(int \*);

void execute\_E(int \*);

char parse\_args(const char \*, int \*);

// INPUT VALIDATION

char validate\_input(int \*);

char check\_param(int, int, int, char \*);

// ISR inits

void timer1\_init(void);

// DAC Write

int write\_dac(float, int);

int main(void)

{

usart\_init();

adc\_init();

write\_dac(0,0);

write\_dac(0,1);

//timer1\_init();

const *size\_t* arr\_len = 14; // max length of a command

char inputstring[arr\_len]; // input string to hold commands

unsigned int command\_code;

while (1)

{

command\_code = read\_command(inputstring, arr\_len);

if (command\_code != 0x00)

{

interpret\_command(inputstring);

}

}

}

// DAC Functions

// ///////////////////////////////////////////////////////////////////

int write\_dac(float voltage, int output)

{

int write\_val = (int)(voltage / 19.6);

i2c\_init();

i2c\_start(0x58 + I2C\_WRITE);

i2c\_write(output);

i2c\_write(write\_val);

i2c\_stop();

return write\_val;

}

// USART Functions

// ///////////////////////////////////////////////////////////////////

void usart\_init(void)

{

UCSR0A = UCSR0A & ~(0x02); // Set the mode to set "Async Normal Mode" (Slide 45 SerialComm)

UCSR0B = (1 << RXEN0) | (1 << TXEN0); // set to transmit and receive

UBRR0 = 0x33; // UBRR0 = [8000000 / 16(9600)] - 1 = 51.083 (51?)

}

void usart\_prints(const char \*sdata)

{

while (\*sdata)

{

// Wait for UDRE0 to become set (==1), which indicates

// the UDR0 is empty and can receive the next character (Slide 46, Serial Comm)

while (!(UCSR0A & (1 << UDRE0)))

; // Option A

//while (!(UCSR0A & (1<<TXC0))); // Option B

UDR0 = \*(sdata++);

}

}

// ///////////////////////////////////////////////////////////////////

// ADC Functions

// ///////////////////////////////////////////////////////////////////

// Configure ADC

void adc\_init(void)

{

ADMUX = ADMUX | (1 << REFS0); // Configure ADC Reference

ADCSRA = (1 << ADEN);

}

int read\_adc(void)

{

ADCSRA = ADCSRA | (1 << ADSC);

while (ADCSRA & (1 << ADIF))

;

short tmp = ADCW;

int voltage = tmp;

voltage \*= 5;

if (voltage > 5000)

{

voltage = 5000;

}

return voltage;

}

int check\_bounds(int voltage)

{

if (voltage < 0)

{

voltage = 0;

}

else if (voltage > 5000)

{

voltage = 0;

}

return voltage;

}

// ///////////////////////////////////////////////////////////////////

// command digits key

// invalid command

// invalid command --> 0x00

// M --> 0x01

// S:a,n,t --> 0x02

// R:a,n --> 0x03

// E:a,n,t,d --> 0x04

// Command Logic

// ///////////////////////////////////////////////////////////////////

void interpret\_command(const char \*command\_string)

{

int param\_arr[4];

char failure = 0x00;

failure |= parse\_args(command\_string, param\_arr);

failure |= validate\_input(param\_arr);

if (failure == 0x01)

{

print\_single\_line\_message("Failure to Parse!");

}

else

{

switch (command\_string[0])

{

case 'M':

execute\_M();

break;

case 'S':

execute\_S(param\_arr);

break;

case 'R':

execute\_R(param\_arr);

break;

case 'E':

execute\_E(param\_arr);

break;

}

}

}

// INPUT VALIDATION:

char validate\_input(int \*params)

{

char ret\_val = 0x00;

ret\_val |= check\_param(params[0], 0, 510, "0 =< a =< 510");

ret\_val |= check\_param(params[1], 1, 20, "1 =< n =< 20");

ret\_val |= check\_param(params[2], 1, 10, "1 =< t =< 10");

ret\_val |= check\_param(params[3], 0, 1, "0 =< d =< 1");

if (params[0] + (2 \* params[1]) > 510)

{

ret\_val = 0x01;

print\_single\_line\_message("a + (2 \* n) <= 510");

}

return ret\_val;

}

char check\_param(int value, int min\_v, int max\_v, char \*message)

{

if (value < min\_v || value > max\_v)

{

print\_single\_line\_message(message);

return 0x01;

}

return 0x00;

}

// parse args --> the return code can be passed by ref to reduce mem usage

// ///////////////////////////////////////////////////////////////////

char parse\_args(const char \*command, int \*arr)

{

// set all params to a valid initial state

for (int i = 0; i < 4; ++i)

{

arr[i] = 1;

}

// parse the command character array

if (command[0] != 'M')

{ // if the command isn't M

unsigned int param\_count = 0; // which parameter are we parsing

unsigned int command\_index = 1;

char param[3];

unsigned int param\_index = 0;

// loop through command array and parse args

do

{

command\_index += 1;

if (command[command\_index] == ',' || command[command\_index] == '\0')

{

// save param

arr[param\_count] = *atoi*(param);

param\_count += 1;

// reset param

param\_index = 0;

for (int i = 0; i < 3; ++i)

{

param[i] = '\0';

}

}

else

{

// add to param char buffer

if (command[command\_index] < '0' || command[command\_index] > '9'){

print\_single\_line\_message("Non-numeric value detected!");

return 0x01;

}

param[param\_index] = command[command\_index];

param\_index += 1;

}

} while (command[command\_index] != '\0');

}

return 0x00;

}

// ///////////////////////////////////////////////////////////////////

// specific functions for each sub-command

// M

// ///////////////////////////////////////////////////////////////////

void execute\_M(void)

{

read\_adc();

int adc\_output = check\_bounds(read\_adc());

char adc\_buff[10];

*sprintf*(adc\_buff, "v=%d.%d V", adc\_output / 1000, adc\_output % 1000);

print\_single\_line\_message(adc\_buff);

}

// ///////////////////////////////////////////////////////////////////

// S

// ///////////////////////////////////////////////////////////////////

void execute\_S(int \*params)

{

// blocking store

int adc\_val;

for (int current\_n = 0; current\_n < params[1]; ++current\_n)

{

read\_adc();

adc\_val = check\_bounds(read\_adc());

eeprom\_write\_word(params[0] + (current\_n \* 2), adc\_val);

char adc\_buff[30];

*sprintf*(adc\_buff, "t=%d s, v=%d.%d V, addr: %d", current\_n \* params[2], adc\_val / 1000, adc\_val % 1000, params[0] + (current\_n \* 2));

print\_single\_line\_message(adc\_buff);

if (current\_n < params[1]-1){

my\_delay(params[2]);

}

}

}

// ///////////////////////////////////////////////////////////////////

// R

// ///////////////////////////////////////////////////////////////////

void execute\_R(int \*params)

{

int adc\_val;

for (int current\_n = 0; current\_n < params[1]; ++current\_n)

{

adc\_val = check\_bounds(eeprom\_read\_word(params[0] + (current\_n \* 2)));

char adc\_buff[20];

*sprintf*(adc\_buff, "v=%d.%d V, addr: %d", adc\_val / 1000, adc\_val % 1000, params[0] + (current\_n \* 2));

print\_single\_line\_message(adc\_buff);

}

}

// ///////////////////////////////////////////////////////////////////

// E

// ///////////////////////////////////////////////////////////////////

void execute\_E(int \*params)

{

// Blocking E

int adc\_val;

for (int current\_n = 0; current\_n < params[1]; ++current\_n)

{

adc\_val = check\_bounds(eeprom\_read\_word(params[0] + (current\_n \* 2)));

char adc\_buff[50];

int write\_result = write\_dac(adc\_val, params[3]);

*sprintf*(adc\_buff, "t=%d s, DAC channel %d set to %d.%d V (%dd)", current\_n\*params[2], params[3], adc\_val / 1000, adc\_val % 1000, write\_result);

print\_single\_line\_message(adc\_buff);

if (current\_n < params[1]-1){

my\_delay(params[2]);

}

}

}

// ///////////////////////////////////////////////////////////////////

// command digits key

// invalid command

// invalid command --> 0x00

// M --> 0x01

// S:a,n,t --> 0x02

// R:a,n --> 0x03

// E:a,n,t,d --> 0x04

// ///////////////////////////////////////////////////////////////////

unsigned int read\_command(char \*command\_array, *size\_t* arr\_len)

{

print\_new\_line();

usart\_prints("Enter a command $> ");

const int CODE\_TO\_LENGTH[5] = {0, 0, 11, 8, 13}; // IF WE RUN INTO MEM TROUBLE REMOVE THIS

// reset command\_array

for (int i = 0; i < arr\_len; ++i)

{

command\_array[i] = "\0";

}

unsigned char first\_char = " ";

first\_char = read\_char\_from\_pc();

unsigned short max\_command\_length = 0x00;

unsigned int ret\_code = 0x00;

command\_array[0] = first\_char;

switch (first\_char)

{

case 'M':

ret\_code = 1;

break;

case 'S':

ret\_code = 2;

break;

case 'R':

ret\_code = 3;

break;

case 'E':

ret\_code = 4;

break;

default:

print\_single\_line\_message("Error: Invalid Command!!");

}

max\_command\_length = CODE\_TO\_LENGTH[ret\_code];

char curr\_char;

unsigned int curr\_read = 1;

while (curr\_read < max\_command\_length && curr\_char != '\r')

{

curr\_char = read\_char\_from\_pc();

if (curr\_char != '\r')

{

command\_array[curr\_read] = curr\_char;

}

curr\_read += 1;

}

return ret\_code;

} // Reading from serial input

// ///////////////////////////////////////////////////////////////////

unsigned char read\_char\_from\_pc(void)

{

while (!(UCSR0A & (1 << RXC0)))

;

return UDR0;

}

// ///////////////////////////////////////////////////////////////////

// utility prints

// ///////////////////////////////////////////////////////////////////

void print\_new\_line(void)

{

usart\_prints("\n\r");

}

void print\_single\_line\_message(const char \*message)

{

print\_new\_line();

usart\_prints(message);

print\_new\_line();

}

// ///////////////////////////////////////////////////////////////////

// my\_delay

// ///////////////////////////////////////////////////////////////////

void my\_delay(int delay)

{

switch (delay)

{

case 1:

*\_delay\_ms*(1000);

break;

case 2:

*\_delay\_ms*(2000);

break;

case 3:

*\_delay\_ms*(3000);

break;

case 4:

*\_delay\_ms*(4000);

break;

case 5:

*\_delay\_ms*(5000);

break;

case 6:

*\_delay\_ms*(6000);

break;

case 7:

*\_delay\_ms*(7000);

break;

case 8:

*\_delay\_ms*(8000);

break;

case 9:

*\_delay\_ms*(9000);

break;

case 10:

*\_delay\_ms*(10000);

break;

}

}

// ///////////////////////////////////////////////////////////////////

## 6. Appendix B: References

*[1] Peter Fleury’s i2cmaster:*

<http://homepage.hispeed.ch/peterfleury/doxygen/avr-gcc-libraries/group__pfleury__ic2master.html>

*[2]: MAX518 Documentation*

<https://datasheets.maximintegrated.com/en/ds/MAX517-MAX519.pdf>