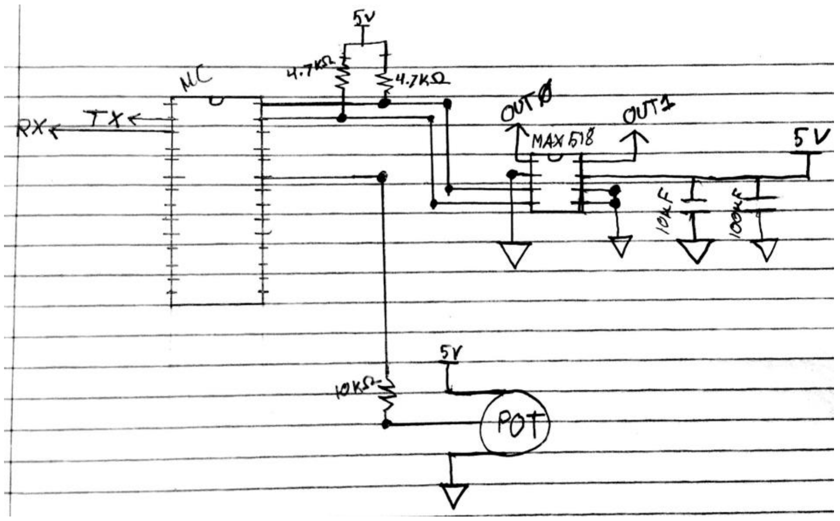
Team 23: Alex Powers and Ben Mitchinson
ECE:3360
Post-Lab Report 5

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:

$$\mathbf{UBRR0} = \frac{8 * 10^6}{(16 * \mathbf{BAUD})}$$

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.

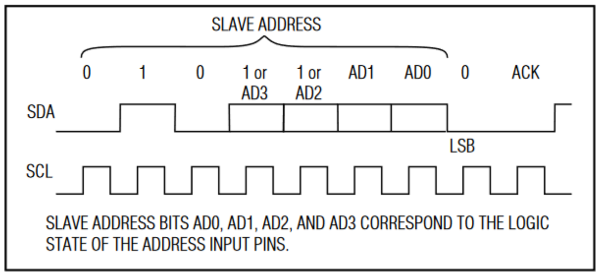

SLAVE ADDRESS
0
1
0
1 or AD3
1 or AD2
AD1
AD0
0
ACK
SDA
LSB
SCL
SLAVE ADDRESS BITS AD0, AD1, AD2, AND AD3 CORRESPOND TO THE LOGIC STATE OF THE ADDRESS INPUT PINS.

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:

$$\textbf{Command Byte} = \left(\frac{5000}{256}\right) * \boldsymbol{V}$$

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.

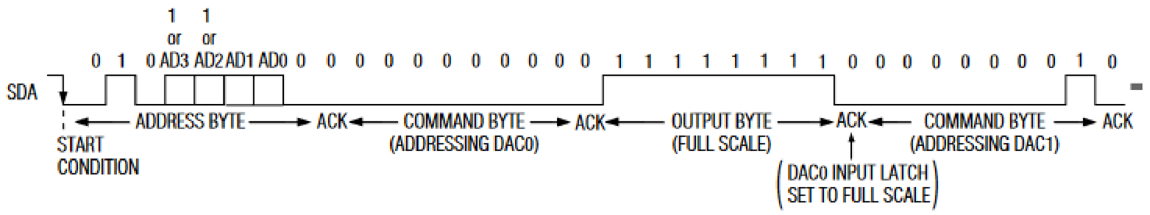

SDA
0 1 0 1 or AD3 1 or AD2 AD1 AD0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 0
START CONDITION
ADDRESS BYTE
ACK
COMMAND BYTE (ADDRESSING DAC0)
ACK
OUTPUT BYTE (FULL SCALE)
ACK
DAC0 INPUT LATCH SET TO FULL SCALE
COMMAND BYTE (ADDRESSING DAC1)
ACK

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:

Team 23: Alex Powers and Ben Mitchinson
ECE:3360
Post-Lab Report 5

<u>Our Commands:</u>

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;
```

```c
			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