2.2.3 ADS1298 ECG/EEG Analog Front End Register Level Programming

In this task I was asked to do register level programming for an ECG/EEG analog front end. Here my supervisor gave me ADS1298RECG-FE ECG front end performance demonstration kit to do the register level programming. This ADS1298RECG-FE Evaluation Module (EVM) consists with an ADS1298 microcontroller which is capable of patient monitoring, high end Electrocardiogram (ECG) and Electroencephalogram (EEG) applications.

Normally ADS1298RECG-FE Evaluation Module is powered up using MMB0 Modular EVM Motherboard. Since the MMB0 Modular in the laboratory did not function well, my supervisor asked me to use Arduino MEGA board to power up the ADS1298RECG-FE EVM.

Before starting the programming part, I went through the data sheet of ADS1298 microcontroller and the manual of EVM to gather necessary information. Here I was asked to do register level programming for the Serial Peripheral Interface (SPI) connection between ADS1298 and Arduino MEGA.

In order to build up the SPI communication between EVM and Arduino MEGA, I had to connect the pins in the following configuration order shown in table 2.1.

Table 2.1 Pin Configuration Order

Arduino MEGA Pin	ADS1298RECG-FE EVM Pin(s)
52 (SCK)	J3.3 (SCLK)
51 (MOSI)	J3.11 (DIN)
50 (MISO)	J3.13 (DOUT)
53 (SS)	J3.7 ($\overline{\text{CS}}$)
3 (Interrupt Pin)	J3.15 ($\overline{\text{DRDY}}$)
5V	J4.10
3.3V	J4.9
GND	J4.5 and J3.4

Above pin details are available in the Serial Interface Pinout table and the MMB0 schematic drawing in the EVM manual.

With the guidance of the table 2.1, I connected the pins as shown in figure 2.6.

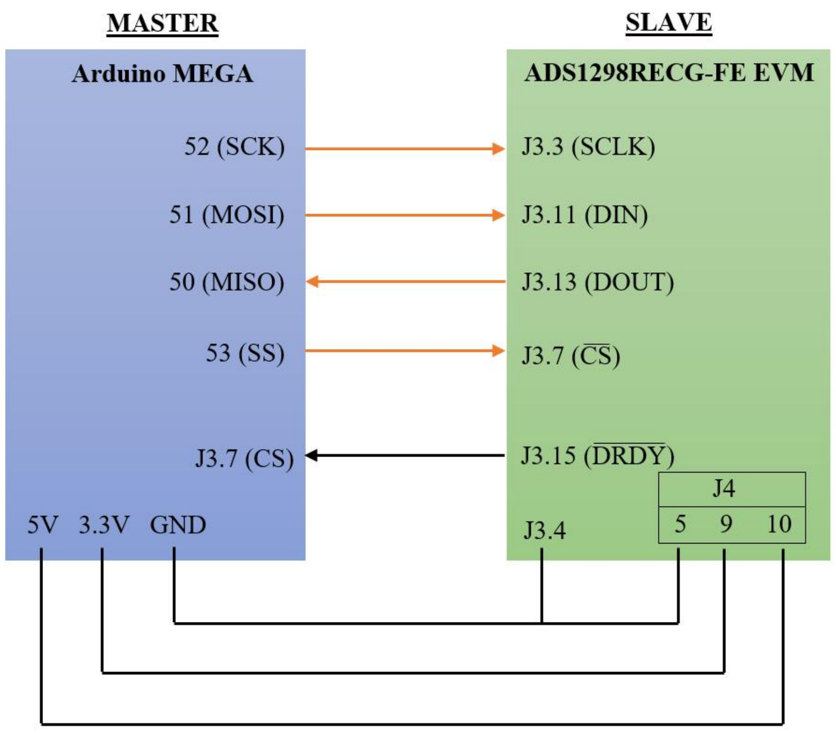


Figure 2.6 Pin Configuration Order

Since Arduino MEGA operates in 5 volts and ADS1298 operates in 3.3 volts, outputs from the Arduino MEGA should not be connected directly to EVM. So, when constructing the circuit, I had to use voltage dividers to all EVM input connections.

Following figure 2.7 shows how I decided the voltage divider circuit.

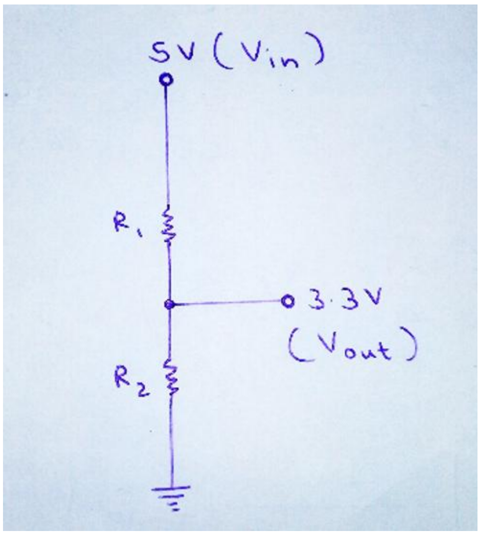


Figure 2.7 Voltage Divider Circuit

The voltage divider circuit calculations were done in the following way.

$$V_{OUT} = \frac{V_{IN}}{R_1 + R_2} \text{x}\, R_2$$

$$3.3\ V = \frac{5\ V}{R_1 + R_2} \text{x}\, R_2$$

$$3.3\ V = \frac{5\ V}{R_1 + R_2} \text{x}\, R_2$$

$$3.3R_1 + 3.3R_2 = 5R_2$$

$$3R_1 = 1.7R_2$$

$$\frac{R_1}{R_2} = \frac{1.7}{3} = \frac{17}{30}$$

After calculating the ratio between R1 and R2, I came up with two resister values which were less than 500 ohms which satisfy the ratio. But my supervisor told me it is not a good practice to use that much low resister values. So, he asked me to use resister values larger than 10 kilo ohms, when constructing this kind of circuits. But since there were no that larger resistors available at the time. So, I constructed the voltage divider circuit by using resistors serially.

Further I had to connect the jumpers in the EVM according to the functions which were available in the EVM manual. The constructed circuit is available in following figure 2.8 and figure 2.9.

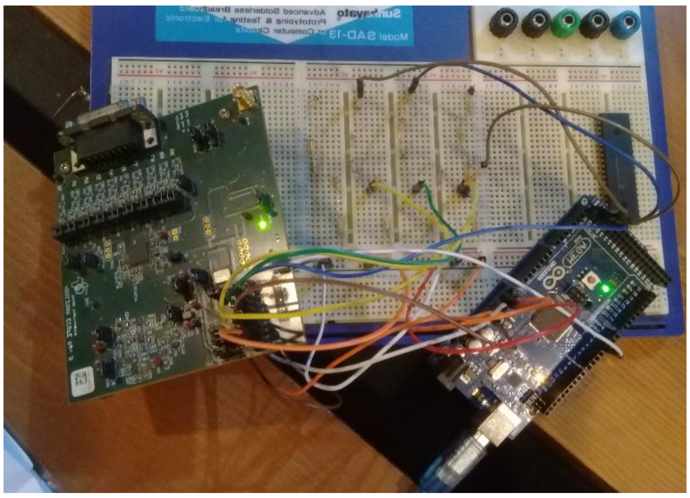


Figure 2.8 Constructed Circuit

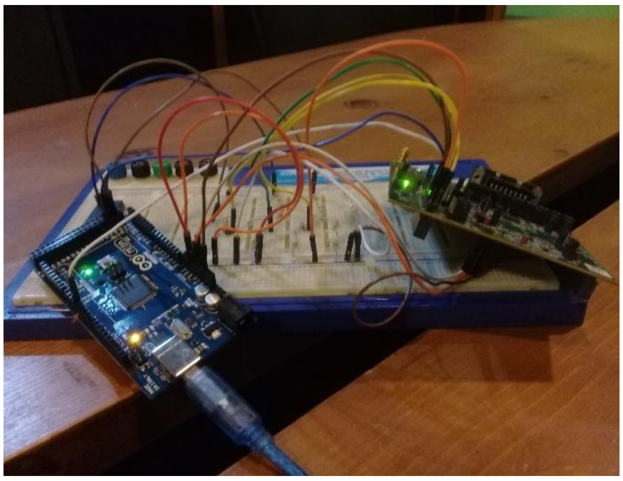


Figure 2.9 Constructed Circuit

After constructing the circuit, I had to verify whether the power supplies are connected correctly. For that there were inbuilt test points on EVM board to check the connectivity. Those test points voltages were available in the EVM manual. So, I measured the voltage values of the test points using the multimeter and verified the circuit I constructed was powered correctly.

As for the programming, I had to follow the ADS1298 microcontroller datasheet and get the information. Since that was a SPI communication, I had to write data on ADS1298 microcontroller registers. When writing data on a ADS1298 microcontroller register, WREG command should be sent first.

WREG command is a two-byte opcode. First byte is assigned for the command opcode and the registered address. Second byte is to specify the number of registers that need to be written. So, the "number of registers to write -1" value was assigned to the second byte.

ADS1298 microcontroller datasheet mentions that throughout the communication, the inverse chip selection pin should be kept logic low. So, to write data on the register, at first, I made the inverse chip selection pin logic low. Next, I send two opcodes as mentioned in the datasheet before sending data. OPCODE1 is written in the following format.

OPCODE1 => 010r rrrr

Here r rrrr refers to the starting register address. Then the OPCODE2 is written in the following format.

OPCODE2 => 000n nnnn

Here n nnnn refers to the "number of registers to write -1" value.

After the OPCODEs are sent, I sent the register input data in the Most Significant Bit (MSB) first format. My supervisor advised me to send data for CONFIG1, CONFIG2 and CONFIG3 at first. CONFIG1, CONFIG2 and CONFIG3 register bit assignments were also available in the datasheet. When sending data for three registers I can use WREG command one time to send data for all three registers. For that I can change the OPCODE2 value to 00010.

$$\begin{aligned} \text{number of registers to write - 1} &= 3-1 \\ &= 2_{10} \\ &= 00010_2 \end{aligned}$$

Instead of doing it separately, by assigning 00000 to the OPCODE2, it is possible to send data to each register separately.

$$\begin{aligned}\text{number of registers to write - 1} &= 1-1 \\ &= 0_{10} \\ &= 00000_{2}\end{aligned}$$

So, I used that method for my program in order to get a better understanding for myself. Following Arduino code lines shows that method.

```
SPI.transfer(0b01000001);
delayMicroseconds(5);
SPI.transfer(0b00000000);
delayMicroseconds(5);
int a = SPI.transfer(0b10100000);
delayMicroseconds(10);
```

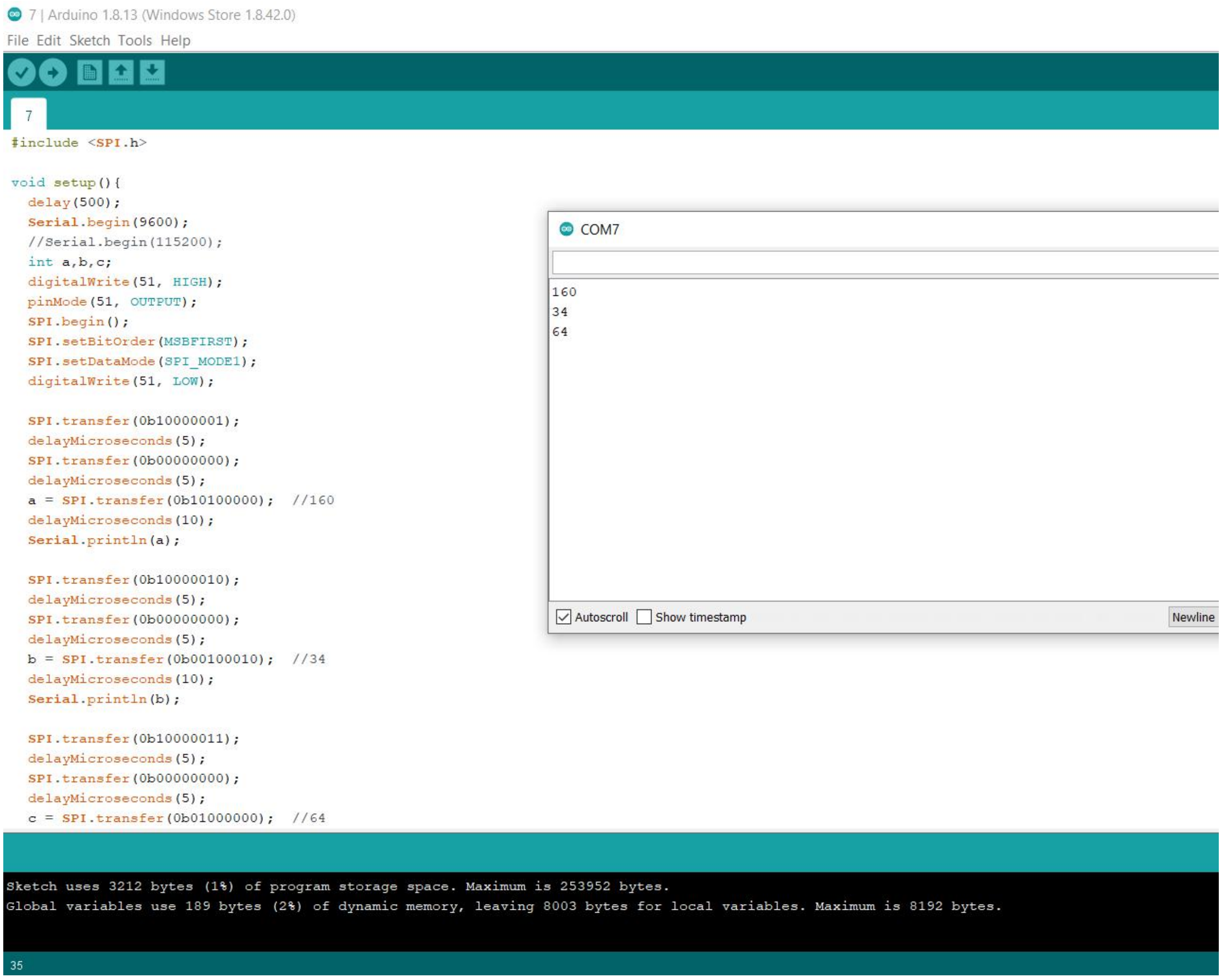
As shown in above, I firstly transferred the OPCODE1 in binary and have a 5 microseconds delay. Then I transferred the OPCODE2 in binary and have another 5 microseconds delay. Next, I transferred the necessary data in binary.

According to the above part of the code it is clear that I have sent the WREG command because the first binary value I sent **010**00001 is in the format of OPCODE1 in WREG register (**010**r rrrr) and the second binary value I sent 00000000 is in the format of OPCODE2 in WREG register (010n nnnn).

Here the last five bits of the OPCODE1 (00001) refers to the starting register address. Then the last five bits of the OPCODE2 (00000) refers to the "number of registers to write – 1" value, which I was shown earlier in the calculations

After sending opcodes, I sent the data as per the requirement. Since I only sent data for three registers, and since I was going to send separate OPCODE2 bits for the three registers, I wrote three sets of programs which is in similar format as thee above Arduino program lines. Then I initialized the required other settings in the program.

Further there was a Power-Up flowchart in the datasheet which indicates the sequence of programming. According to that diagram, I completed the rest of my SPI communication Arduino program and uploaded it to the Arduino MEGA board. Then I was able to observe the output using the Arduino Serial Monitor as shown in figure 2.10. The complete register level program I created is attached in the appendices.


7 | Arduino 1.8.13 (Windows Store 1.8.42.0)
File Edit Sketch Tools Help
7
COM7
Autoscroll
Show timestamp
Newline
35

```
#include <SPI.h>

void setup(){
  delay(500);
  Serial.begin(9600);
  //Serial.begin(115200);
  int a,b,c;
  digitalWrite(51, HIGH);
  pinMode(51, OUTPUT);
  SPI.begin();
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE1);
  digitalWrite(51, LOW);

  SPI.transfer(0b10000001);
  delayMicroseconds(5);
  SPI.transfer(0b00000000);
  delayMicroseconds(5);
  a = SPI.transfer(0b10100000);  //160
  delayMicroseconds(10);
  Serial.println(a);

  SPI.transfer(0b10000010);
  delayMicroseconds(5);
  SPI.transfer(0b00000000);
  delayMicroseconds(5);
  b = SPI.transfer(0b00100010);  //34
  delayMicroseconds(10);
  Serial.println(b);

  SPI.transfer(0b10000011);
  delayMicroseconds(5);
  SPI.transfer(0b00000000);
  delayMicroseconds(5);
  c = SPI.transfer(0b01000000);  //64
```

```
160
34
64
```

```
Sketch uses 3212 bytes (1%) of program storage space. Maximum is 253952 bytes.
Global variables use 189 bytes (2%) of dynamic memory, leaving 8003 bytes for local variables. Maximum is 8192 bytes.
```

Figure 2.10 SPI Communication Program Output