**TEMPERATURE SENSING VIA ADC & TMP36**

**&**

**READING THE TIME VIA AN RTC OVER I2C**

**Ammon Dodson**

**Alex Marlow**

**Jake McKenzie**

**December 11, 2018**

**Introduction**

In this project we are attaching a temperature sensor to the TM4C launch pad and reading in the values using an ADC.

The ADC (Analog to Digital converter) turns a voltage into a number. In our TM4C launch pads, the value is read in as an input voltage, and then is made into a 12-bit number and stored into a register. To do this, the maximum voltage is divided by the maximum value that the 12-bit register can hold. This gives us a resolution. The input voltage can be divided by this resolution to get the number it should be in the 12-bit register.

In this project, Alex worked on the temperature sensor, while Jake and Ammon worked on the real time clock.

The temperature sensor we are using the is TMP36. This temperature sensor has three pins, 5v in, ground and the analog output for the temperature. The sensor is rated to sense -40 to 125 degrees Celsius, with a conversion rate of 10 mv/degree and 750 mv at 25 degrees.

The largest obstacle for understanding I2C is understanding the interconnection between the two main wires SDA (serial data) and SCL (serial clock), which are open drain that connect multiple devices in a multi-drop bidirectional low-speed bus. Jake and Ammon hit the RTC portion of the project as hard as we could. Jake wrote some co If idle, both SDA and SCL are high. Transactions begin when SDA goes low, followed by SCL. This indicates to all receivers on the bus that a packet transmission is starting. While SCL is low, SDA transitions (high or low) for the first valid data bit, which is the start condition. I2C does this weird protocol to take advantage of properties of capacitors. You see, capacitors discharge at a faster rate than they charge, which is why the on (low) and off (high) is the way that it is. With each bit that is sent across the line, the bit must become valid on SDA while SCL is 0. The bit is sampled each rising edge of SCL and must remain valid until SCL goes high once more. Then SDA switches bits before SCL goes 1 once more.

**Temperature sensor**

In this project we used register writing for the temperature sensor. The following list of commands which were used to write to the appropriate registers.

ui32SysClkFreq = **SysCtlClockFreqSet**((SYSCTL\_XTAL\_25MHZ | SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL | SYSCTL\_CFG\_VCO\_480), 120000000);

// Enable the clock to port D

SYSCTL\_RCGCGPIO\_R |= SYSCTL\_RCGCGPIO\_GPIOD;

// configuration of port D pin2

GPIO\_PORTD\_AHB\_DIR\_R |= PORTD\_DIR; //input set

GPIO\_PORTD\_AHB\_DEN\_R &= POTD\_DEN; //digital enable off

GPIO\_PORTD\_AHB\_AMSEL\_R |= PORTD\_AMSEL; //set alternate function registers

GPIO\_PORTD\_AHB\_AFSEL\_R |= PORTD\_AFSEL;

// Enable the clock for ADC0

SYSCTL\_RCGCADC\_R |= SYSCTL\_RCGCADC\_ADC0;

//turns off ADC0 for configuration

ADC0\_ACTSS\_R |= 0x0;

// delay

Delay (3);

//sets continuous triggering

ADC0\_EMUX\_R |= EMUX\_SS3\_DEFUALT;

// Select AN13 ( PD2 ) as the analog input

ADC0\_SSMUX3\_R |= SSMUX3\_AIN13\_SET;

// quarter conversion rate; 48\*Tadc periods pause

ADC0\_PC\_R |= PC\_WAIT48;

// 1st sample is end of sequence and source of interrupt

ADC0\_SSCTL3\_R |= SSCTL3\_INTEND;

// 16x oversampling and then averaged

ADC0\_SAC\_R |= SAC\_16X\_SET;

// Unmask ADC0 sequence 3 interrupt

ADC0\_IM\_R |= IM\_SS3\_ENABLE;

// Clear the interrupt for ADC0 sequencer 3

ADC0\_ISC\_R |= ISC\_SS3\_CLEAR;

// Enable ADC0 sequencer 3 interrupt in NVIC

NVIC\_EN0\_R |= NVIC\_SS3;

// Enable ADC0 module for sequencer 3

ADC0\_ACTSS\_R |= ACTSS\_SS3\_ENABLE;

// Initiate sequencer 3

ADC0\_PSSI\_R |= PSSI\_SS3\_START;

// wait certain time for ADC module do the conversion

Delay(100);

//loop for program to continue running

**while** (1) {

}

The first command used is,

ui32SysClkFreq = **SysCtlClockFreqSet**((SYSCTL\_XTAL\_25MHZ | SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL | SYSCTL\_CFG\_VCO\_480), 120000000);

This command initiates the system clock, so that the ADC clock can be enabled and work properly.

The second command,

SYSCTL\_RCGCGPIO\_R |= SYSCTL\_RCGCGPIO\_GPIOD;

Enables the clock for the port D, by writing a 1 to its appropriate spot in the register.

The next set of commands

GPIO\_PORTD\_AHB\_DIR\_R |= PORTD\_DIR;

GPIO\_PORTD\_AHB\_DEN\_R &= POTD\_DEN;

GPIO\_PORTD\_AHB\_AMSEL\_R |= PORTD\_AMSEL;

GPIO\_PORTD\_AHB\_AFSEL\_R |= PORTD\_AFSEL;

Make the port D direction input, turn the digital mode off for port D as well as turn on the alternate function mode and enable the ADC circuitry.

The next command,

SYSCTL\_RCGCADC\_R |= SYSCTL\_RCGCADC\_ADC0;

Enables the clock for the ADC functionality, by writing a 1 to the part of the register for the ADC0 converter.

The following commands are both safety measures,

ADC0\_ACTSS\_R |= 0x0;

Delay (3);

The first disables the ADC0 temporarily so that we can change the setting without risking the possibility of it doing things, and the second is a short delay for after the clock enable so that everything is running before we try and change anything.

The next command,

ADC0\_EMUX\_R |= EMUX\_SS3\_DEFUALT;

Sets the ADC so that it continuously triggers, so that it is always taking another reading after it has completed the last. It does this by writing a 0xF000 to the appropriate register.

The next command,

ADC0\_SSMUX3\_R |= SSMUX3\_AIN13\_SET;

Sets the ADC sequencer to use AIN13 with corresponds with the GPIO pin 2.

The following command,

ADC0\_PC\_R |= PC\_WAIT48;

Sets a wait time after each sequence is taken, before the next sample starts.

The next command,

ADC0\_SSCTL3\_R |= SSCTL3\_INTEND;

Sets the end of sample to be the source of the interrupt, as well as making it trigger after only one sequence.

The next command,

ADC0\_SAC\_R |= SAC\_16X\_SET;

Sets the sequencer to do 16x sampling, which makes it read in 16 values, average them, then use that value to give to the 12-bit final value register.

The next commands,

ADC0\_IM\_R |= IM\_SS3\_ENABLE;

ADC0\_ISC\_R |= ISC\_SS3\_CLEAR;

Enable the ADC0 interrupt by writing a mask to it, as well as clearing the interrupt to make sure it is triggered currently.

The following command,

NVIC\_EN0\_R |= NVIC\_SS3;

Enables interrupt 17, which corresponds with the interrupt for ADC0.

The next commands,

ADC0\_ACTSS\_R |= ACTSS\_SS3\_ENABLE;

ADC0\_PSSI\_R |= PSSI\_SS3\_START;

Enable the ADCO sequencer 3, which we had disabled close to the begging in of the code. And starts the sequencer by writing the appropriate value to the starter.

The last commands

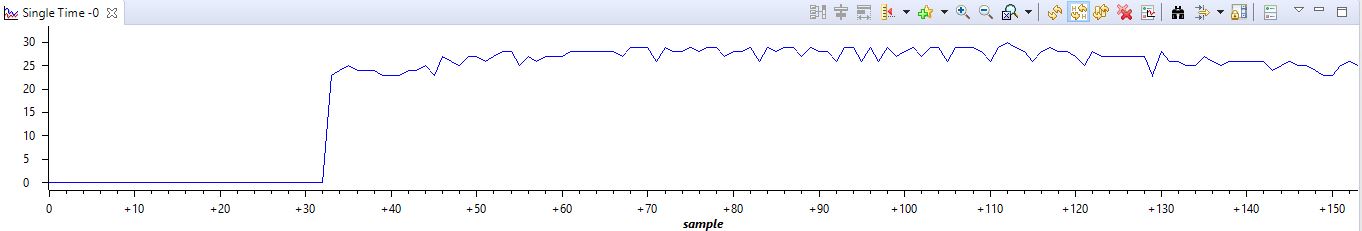
Delay(100);

**while** (1) {

}

Gives a delay for the sequencer to start working, then the while loop makes the program run indefinitely while it is taking in temperature data.

To make sure that our code was working we added the Data and Temperature variables to the watch list and turned on auto updating. We also made a graph for the temperature and turned on auto updating while the program was running. The graph we got from this is shown below.



This picture shows the temperature of the room and it is changing when a finger is put on the sensor and then taken off.

To read the data from the destination register we used interrupts in the following code.

**void** **ADC0SS3\_Handler** (**void**) {

ADC0\_ISC\_R |= ISC\_SS3\_CLEAR; //clear interrupt

**int** DATA = 0; //input data

DATA = ((ADC0\_SSFIFO3\_R) & 0x0FFF); //read data in

Temperature = (((DATA\*.806) - 750)/10 + 25); //do conversions

ADC0\_ISC\_R |= ISC\_SS3\_CLEAR; //clear interrupt again, (for some reason this had to be done)

}

In this code the interrupt is cleared twice because that is what was found to work. The temperature conversion is done by getting the given value first in mV, which we can do by getting the resolution, which in this case is 3.3 V divided by the maximum value of the 12-bit register which is 4096. This gives us the resolution number of .000806. So, if we want our DATA value in mV, we multiply it by .806. Then to center it we minus 750 mV, so any leftover voltage is the error from 25 degrees. Then we divide by 10, because we get 10 mV per degree, now we have the error in degrees. Finally, we add 25 degrees to the error and we get our final temperature.

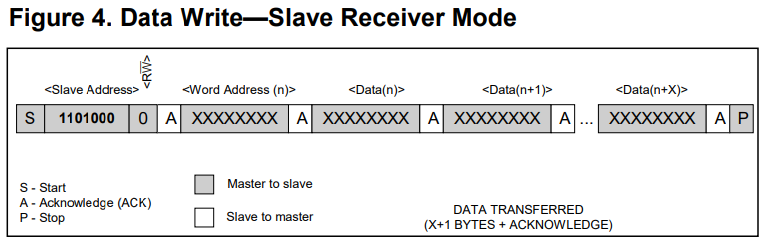
**Reading Time via a Real Time Clock over I2C**

Start and stop bits, I prefer to think of them as conditionals, must be sent from the master along with the address bits. The data bits on the other hand can be sent by the slave or the master. This particular protocol allows for multiple masters and multiple slaves but they all share the same SDA and SCL serial line. This protocol is good because there is no intermediate value, unlike some other protocols, because the SDA and SCL is connected to the bus via a large resistor the current will immediately drain whenever voltage goes over the line. This makes the protocol very responsive and resistant to noise. Just to reiterate, the master initiates and terminates transmission and generates the SCL. The slave is addressed by the master. Orthogonal to the master and slave are the transmitter and receiver. The transmitter places data on the bus while the receiver reads data from the bus. The master can be both a transmitter and receiver, and so can the slave. Every node is either a master or slave, or either a transmitter or receiver.

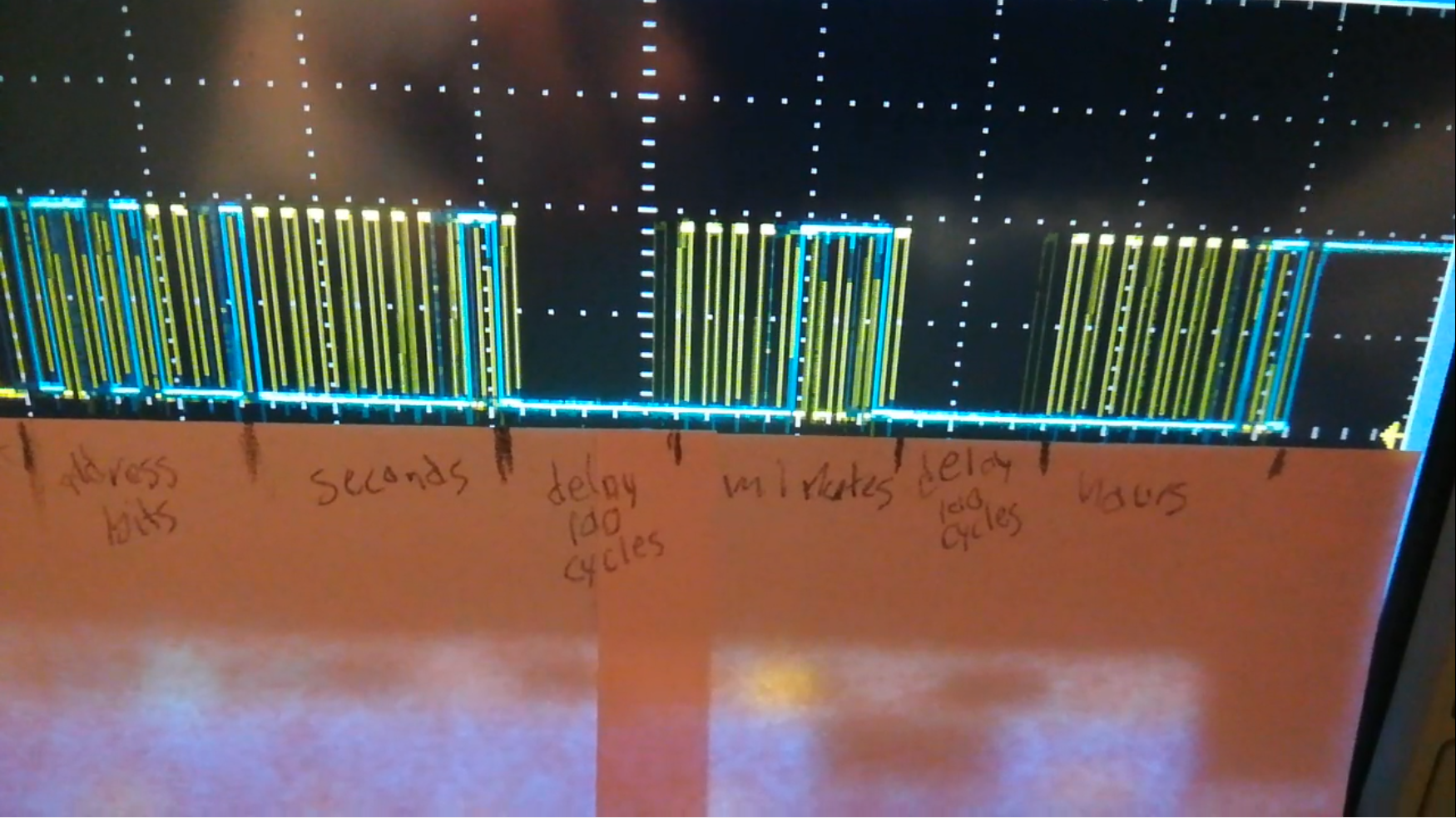
The largest obstacle in accomplishing this project was dealing with TI’s crummy data sheets and documentation for their peripherals. The datasheet explicitly said to use I2CMasterBusy to wait for when the master was busy but in their own example code they state to use I2CMasterBusBusy to wait for the bus busy bit but instead of the busy bit. This issue was the hardest problem we had and gave us the most trouble. When debugging hardware and it does nothing, but it tells you that it is working, that can be the most frustrating thing. We decided at this point that anything we did would need to be looked at on the oscilloscope. If you’re ever having trouble with I2C the only real proper way to find what is wrong is by capturing traffic on the bus and look at what is happening bit by bit. You can stare at code all day until you’re blue in the face but until you look at what the hardware is doing, you’re not going to know what is happening!

We also ran into hardware that was broken. The DS1307 we bought appears to be able to be able to receive address bits but NOT receive data bits. We’ve tested the device on the same piece of code and on one the device receives the address bits but fails. Since we were far along, another group lent us their clock, we tested our code on it and it was successfully sending and receiving data sort of correctly. The important part is that the code was correct at that point. Once we could see what was being transmitted over the oscilloscope, the real debugging could begin. While reading the oscilloscope manual on edge triggering we learned that Tektronix’s claim to fame is discovering edge triggering.

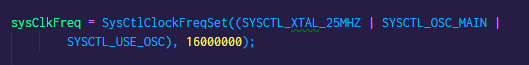
It is worth noting that we used the peripheral library to implement this portion of the project and that “SYSCTL\_USE\_OSC” was used in setting the clock frequency, not “SYSCTL\_USE\_PLL”. This decision was made after we noticed that this resulted in a much more stable signal coming from the device itself.



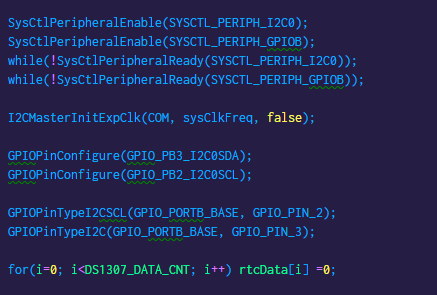
The code was structured to match the standard on the left written in a simple and crisp way to try to reduce the redundancy in our code. Before this fact we were just testing things and were a bit too grandiose in what we could accomplish in the short amount of time.



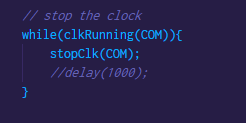
We ended up needing to add a more than 100 cycle delay (typo in the picture) between every instruction because we could not find, within the time limit, what instruction was causing us issues. If this was reimplemented an interrupt-based method would be employed with no delays. Having to delay with polling caused serious problems for the communication. Slowing it down significantly helped tremendously, but sometimes a race condition gets hit. The only way to fix this is to do it properly with interrupts or to go through meticulously finding which line needs a delay then tuning the code to what works. As we can se above, our output matches perfectly with the I2C format as specified in the documentation on the previous page.



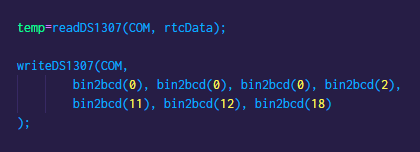
Above is where we changed the system clock frequency. By using SYSCTL\_USE\_OSC we were able to see the signal over the line clearly and crisply.



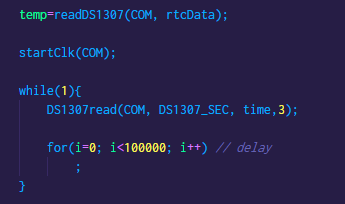
Above is the setup ritual for enable the peripherals. We ended up using the GPIO B and I2C0. They require a busy loop right after they are enabled to prevent bad behavior. After this the I2C clock is initialized in slow mode. We configured PB3 and PB2 for our SDA and SCL lines respectively. Then enabled the pins to be of type I2C. To the best of our knowledge this is the correct way to do so. We then write to the RTC data zero initially to reset the clock.



The above loop was a debugging method. It essentially means, “while the clock is running, try your hardest to stop that darn clock”. The clock is perpetually in a state of not wanting to be stopped and we could never figure out why.

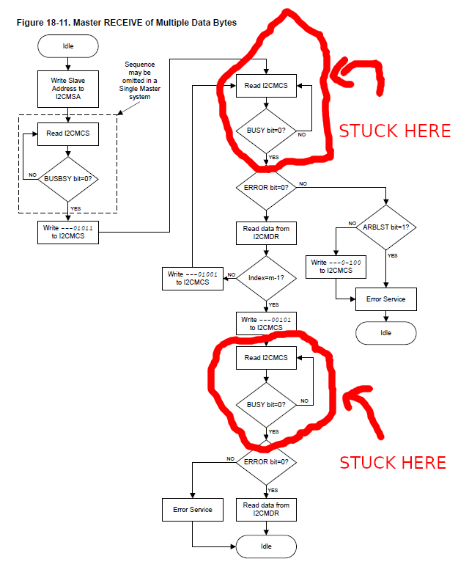


In the above line we are reading from the DS1307. Due to the complexity of that function we will not go into all the details of the function. Essentially, we give it an I2C location and a data pointer. That line can also be seen mirrored in the code below where we do the same. After this we write to the DS1307, which does a binary to BCD, functions we also implemented on our own.



After this do a massive delay, that reads every hundred thousand cycles. This was the end of main.

Things started working better for our project when we compartmentalized the write and read to the DS1307 into their own functions. Before this we were doing both reading and writing together and there was something wrong in the steps where we would get stuck at this stage of the burst process. At first when we talked to you, we thought we were stuck in the top loop in the figure below where it says stuck here but that was just a hardware fault. Once we got hardware that was working we got stuck in the bottom loop where it says stuck here.



**Conclusion**

The TMP36 is a specialized transistor used for measuring temperature designed in such a way to produce specific voltage range that has a linear reference to temperature over a given range. Alex mostly did the temperature section on his own and it did not take him too much time. In comparison doing this on the TM4C1294 was quite straightforward compared to communicating over the I2C bus. The documentation for doing the I2C part was quite haphazardly written with contradictory information laid throughout the documentation. Honestly, we did not feel acclimated with the peripheral library enough to adequately use all the functions and documentation correctly because we spent so much time writing directly to bits. We also ran into broken hardware for the DS1307. This was hardware we bought ourselves came from Amazon and was provided by Brooke, who despite it being broken, really helped us get started. The next RTC clock was provided by Aric, who did not say where he acquired the RTC clock, but he did have one that worked. We ran into a problem with the I2C part where a race condition would trigger undesirable. There is a video of the clock working in action on the oscilloscope for the teacher’s prerogative. The seconds, minutes and hours (we did not check hours due to limited time) but do all seem to be incrementing correctly. The temperature sensing also seemed correct. All and all this was a fun, hectic project. We learned a lot and accomplished what we needed to accomplish.