# Refrigerator Monitoring System

Michael Musick, Gavin Fielder

EECE 344 Digital Systems Design, CSU Chico

Dr. Hadil Mustafa

May 12th, 2018

# Introduction, Motivation

We built a monitoring system for a refrigerator. It monitors whether the door is closed as well as the internal temperature of the fridge, generating an alarm when either the door is left open or the temperature inside the refrigerator goes above a threshold.

This can be adapted into two categories of product:

- As a component for manufacturers of home refrigerators.
- As a standalone system that can be installed on commercial refrigerators.

In either case, it provides a layer of security against food spoilage by alerting the user when the integrity of the refrigerator system is compromised.

As this is a reasonably simple device that any company with investment in electrical engineers could produce, the second category of product is more likely to attract customers.

# Related Work

Similar products already exist. Our implementation is neither a new product nor an improvement. Since, ultimately, this device is a security feature, the primary way our product can be competitive in the marketplace is in its reliability. In this way, simplicity is its best feature.

# Specification

The target user is any company or store that has a refrigerator in which significant stock is stored and does not have a built-in monitoring system. It would likely be sold in two ways: 1) by contacting the business directly and delivering a sales pitch, and 2) providing units to construction companies building new restaurants or stores. For the home refrigerator route, sales would likely be made by bidding contracts to refrigerator manufacturers.

The minimum specifications for a saleable product is:

- Can be installed on an existing refrigerator
- Runs on a reliable power source (AC adapter, or battery system that can detect low battery)
- Accurately reads temperature within 2 deg F
- Produces a notification when the fridge door is left open
- Produces a notification when the internal temperature rises above a threshold

The competitive advantage for this product is made on these factors, in order of importance:

1. Reliability
2. Configurability of thresholds

3. Energy consumption
4. Price
5. Fine temperature accuracy

In addition, significant market value is added as well if the unit can function for freezers as well as it can for refrigerators.

Reliability is a complicated measure. While the entire field of reliability engineering could not be simplified into a simple strategy, simplicity and modularity are key strategies to high reliability.

Configurability of thresholds can be achieved.

Energy consumption can be optimized by reducing the clock speed, which is possible because this device should not need a fast clock, and also by reducing the number of ports used, or in general using a simpler microcontroller.

Price can also be reduced by using a simpler microcontroller and by using a thermistor of the minimum viable resolution for the 0 to 60 degree range Fahrenheit, though conversely, fine temperature accuracy is primarily achieved by a higher quality thermistor.

### Specifications for the first iteration

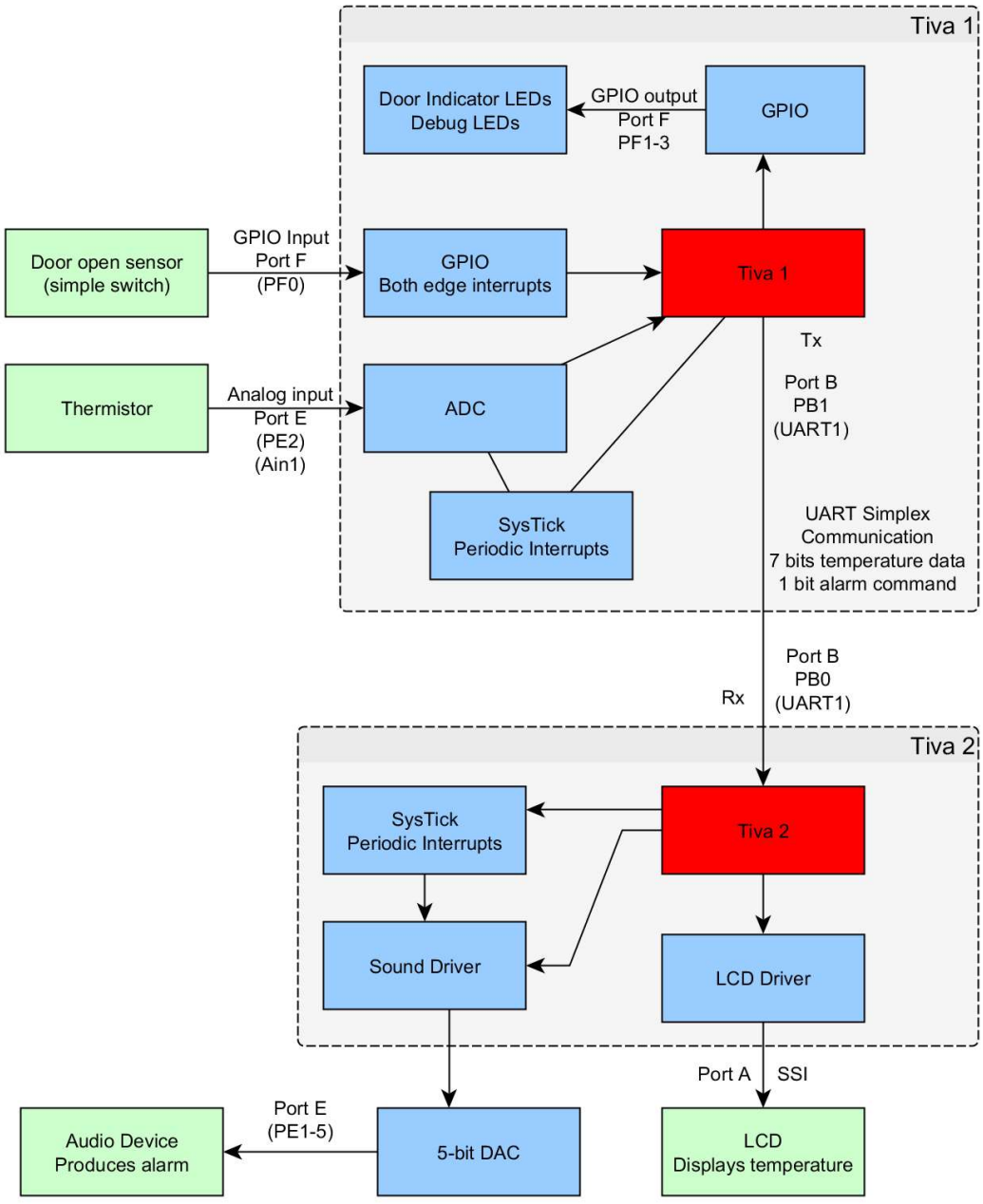
For the first minimum viable product, our specifications are as follows:

- Accurately reads temperature within 3 deg F
- Produces an alarm when the fridge door is left open
- Produces an alarm when the internal temperature rises above a threshold

This provides a good starting point for future improvements.

# Design

## Block Diagram




# Hardware

## LCD

In order to give the user feedback regarding the state of the system, we connected a Nokia 5110 LCD to port A (as per Valvano's driver) of the receiver Tiva board to use to output the current ambient temperature of the thermistor.

### 7 Segment Display

Because of a problem relating to the system clock, part way through the project development we could no longer get the Nokia LCD to display the appropriate output. As such, we considered the alternative of using two common anode 7 segment displays to display the temperature data instead. We built a test circuit for the displays by connecting each cathode to port B of the Tiva board through 220Ω resistors, while the common anode pins were each connected to the emitter pin of a BJT transistor. We ran 5V from the Tiva clock bus to the collector pins and we connected two pins on port A to the base pins of the transistors to trigger the switching. We then proceeded to write drivers to control these displays through multiplexing. Ultimately, however, we were able to solve the clock issue and we migrated back to the LCD for the sake simplicity and modularity, should we ever want to add more visual output to the system.

### Door Switch

The switch used to simulate the door opening and closing is a dpst push button wired with negative logic to the Tiva board.

### Thermistor

In order to measure temperature, we purchased a negative temperature coefficient thermistor from the IEEE chapter at Chico. Unfortunately, the component did not have a data sheet or any markings to indicate its type or characteristics. We decided to use the component anyways and went to work characterizing it. We determined how its resistance relates to temperature (see ADC and Thermistor Characterization under Development Plan) and connected it in a voltage divider with a reference resistor to generate a predictable voltage, which we collect with the Tiva's ADC.

#### Considerations for future iterations

Thermocouples provide a significantly faster response time and create a voltage difference to measure without drawing current from the Tiva, though their outputs may need to be amplified. Similarly, thermostats require no current to operate, and could be used without the ADC (since they operate like switches), but only measure a single temperature.

### DAC and Sound Output

Since we wanted to sound an alarm in the event that the internal refrigerator temperature gets too warm or if the door is left open, we needed a method to produce sound. We did this by using a 5-bit DAC connected to a powered speaker through a conventional 3.5mm jack plug. In order to drop the output voltages we used resistor values of 1.5kΩ, 3kΩ, 6kΩ, 12kΩ, and 24kΩ.

## Software

### Bus frequency

Both boards use PLL at 80 MHz.

### Considerations for future iterations

Reducing the clock speed will reduce energy consumption.

## FriendlyWriter and generalized initialization

In order to improve the speed and accuracy of development, we wrote a module called FriendlyWriter that reads and writes memory addresses. It consists of 3 functions, Read, Write, and FriendlyWrite. Each takes a 32-bit unsigned integer address, Write also takes an unsigned integer value to write, and FriendlyWrite also takes an unsigned integer representing a mask such that only the bits of the passed value corresponding to the high bits in the mask will be written to the register, and the bits in the register corresponding to the zero bits of the mask retain their value. With this functionality, we wrote generalized initialization functions for both UART simplex communication and the door switch of the sensor unit.

The way generalized initialization works is by defining base addresses of different modules within constant arrays, and defining the offsets for registers in each module. By passing the number of the desired module, the base address is looked up in an array and then the offset is added to that base address to derive the address of a desired register. It then reads and writes to these dynamically-determined addresses using the FriendlyWriter module.

### UART Library

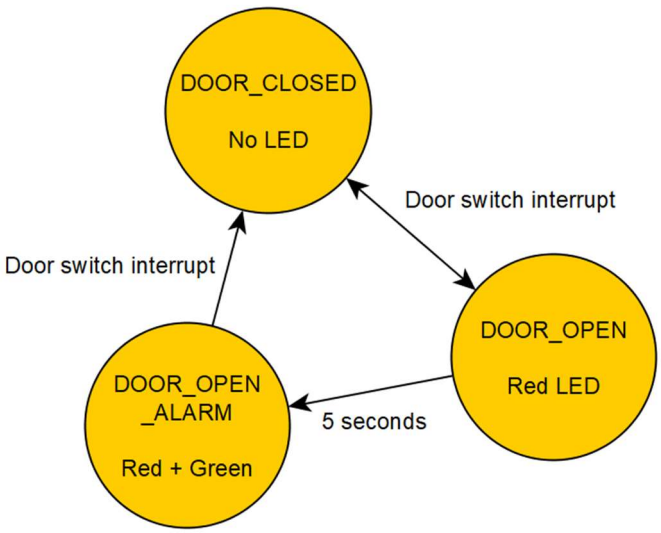
The UART initialization function allows us to specify the port and pin number, the UART number, and whether it is Rx or Tx, such that when we wish to switch pins or switch UART modules, we need only change a single line in the main program. By including functions for both Rx and Tx, we also were able to use identical files in both the receiver and the transmitter. The UART library assumes a bus clock frequency of 80 MHz and sets the baud rate at 9600 bits/s.

### Considerations for future iterations

We could have used the normal path for initialization in which specific registers are hard coded into the initialization. This has the advantage of being fast in initialization as well as easy to read for code maintenance of the initializing functions, but initialization time is not a concern for this product. Instead, generalized initialization allows for faster development and turnaround time, reduces the frequency of programmer error, and much less time is spent bug-fixing when a pin change is deemed necessary. The tradeoff makes it worth implementing generalized initializations for other peripherals, but only when the next big “strange bug” is encountered.

## Tiva 1 (Sensor unit)

The goal for communication to the interface unit is to send 7 bits of temperature data in Fahrenheit and 1 bit of alarm request. The goal internal to the sensor unit, therefore, is to continually update the status of the fridge monitor with regards to the state of the door and the internal temperature of the fridge.



## Main Program

The software is implemented as a simple state machine with three states: DOOR_CLOSED and DOOR_OPEN, in which the alarm bit is cleared, and DOOR_OPEN_ALARM in which the alarm bit is set, indicating the door has been left open. The main loop of the program handles the current state and checks the conditions for transitioning to its other states, such as whether the door has been opened or closed or whether the door has been open for more than a specified time interval. The way the state machine checks the time elapsed is by reading a counter variable which is incremented by the SysTick periodic interrupt at 5 Hz. At 25 counts threshold, the time from the door being open to the alarm being requested is $5 \pm 0.2$ seconds.

The state machine uses two LEDs onboard the Tiva. If the state machine reads the door as open, it turns on the red LED (PF1). This is intended as a user interface on the fridge itself, so that the user can see when the door is reading as open or closed. If the state machine is setting the alarm bit, it turns on the green LED (PF3). We included this as debug functionality.

Outside of the state machine framework, the main loop checks if the ADC has new data from the thermistor, converts the ADC sample to Fahrenheit if so, using a fixed-point format according to a conversion function determined by characteristic data.

Next, if the most recent calculation of temperature in Fahrenheit is above the configured threshold, the alarm bit is set regardless of door state.

Lastly, the main loop sends both the most recent temperature data and the alarm bit to the UART module to be communicated.

## Thermistor

Since the thermistor is an analog device, we must use one of the Tiva's onboard ADC units. To set this up, we initialize ADC0 to channel Ain1 on pin PE2, and then we initialize SysTick with periodic interrupts at 5 Hz. Every SysTick interrupt request, we take an ADC reading. When the ADC is finished, it stores the result in a global variable to be retrieved by the main program, and sets a flag that new data is available.

While the ADC is reading, we turn on the onboard blue LED (PF2) to enable heartbeat debugging. Since the ADC takes some appreciable time and the frequency is low enough to be visible to human vision, we can then ensure that the ADC is generating input by visibly seeing the blue LED flicker at some low duty cycle.

SysTick also increments a global counter variable. We use this as a general purpose timer.

## Temperature Conversion

For demonstration purposes, we implemented a linear fit between ADC sample and degrees farenheit on the range of 52°F to 90°F.

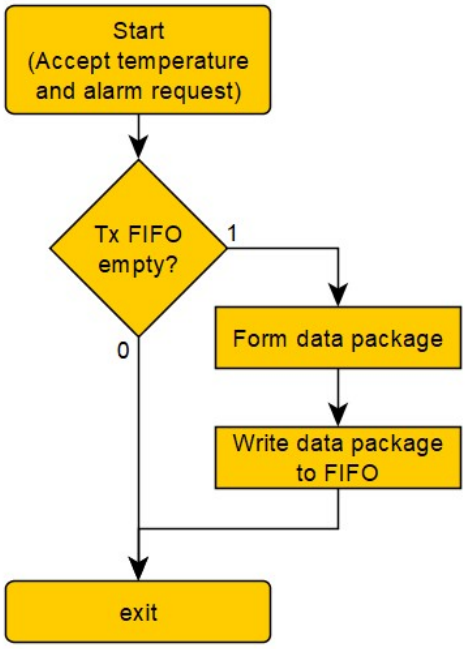
The table below is a subset of the linear fit data. For the full data set, see ADC and Thermistor Characterization under Development Plan.

| Measurements | | Calculations | | | Linear Fit | |
|---|---|---|---|---|---|---|
| Temp (°C) | Resistance (Ω) | Temp (°F) | Volts (V) | Theoretical ADC sample | linear °F | error |
| 10.8 | 1680 | 51.44 | 1.21696 | 1511 | 52 | 0.32% |
| 13 | 1549 | 55.4 | 1.27996 | 1589 | 56 | 1.15% |
| 15 | 1471 | 59 | 1.32067 | 1639 | 59 | -0.16% |
| 19 | 1319 | 66.2 | 1.40793 | 1748 | 65 | -1.75% |
| 21.5 | 1226 | 70.7 | 1.46725 | 1821 | 69 | -2.10% |
| 24 | 1108 | 75.2 | 1.55011 | 1924 | 75 | -0.20% |
| 25.5 | 1004 | 77.9 | 1.63130 | 2025 | 81 | 3.67% |
| 27 | 984 | 80.6 | 1.64790 | 2045 | 82 | 1.65% |
| 29 | 930 | 84.2 | 1.69445 | 2103 | 85 | 1.20% |
| 33 | 849 | 91.4 | 1.76943 | 2196 | 90 | -1.00% |

Here voltage is the voltage calculated by the voltage divider equation with a resistor we tested at 981.5 Ω. For more information, see the full data set and procedure in ADC and Thermistor Characterization under Development Plan.

## UART transmission

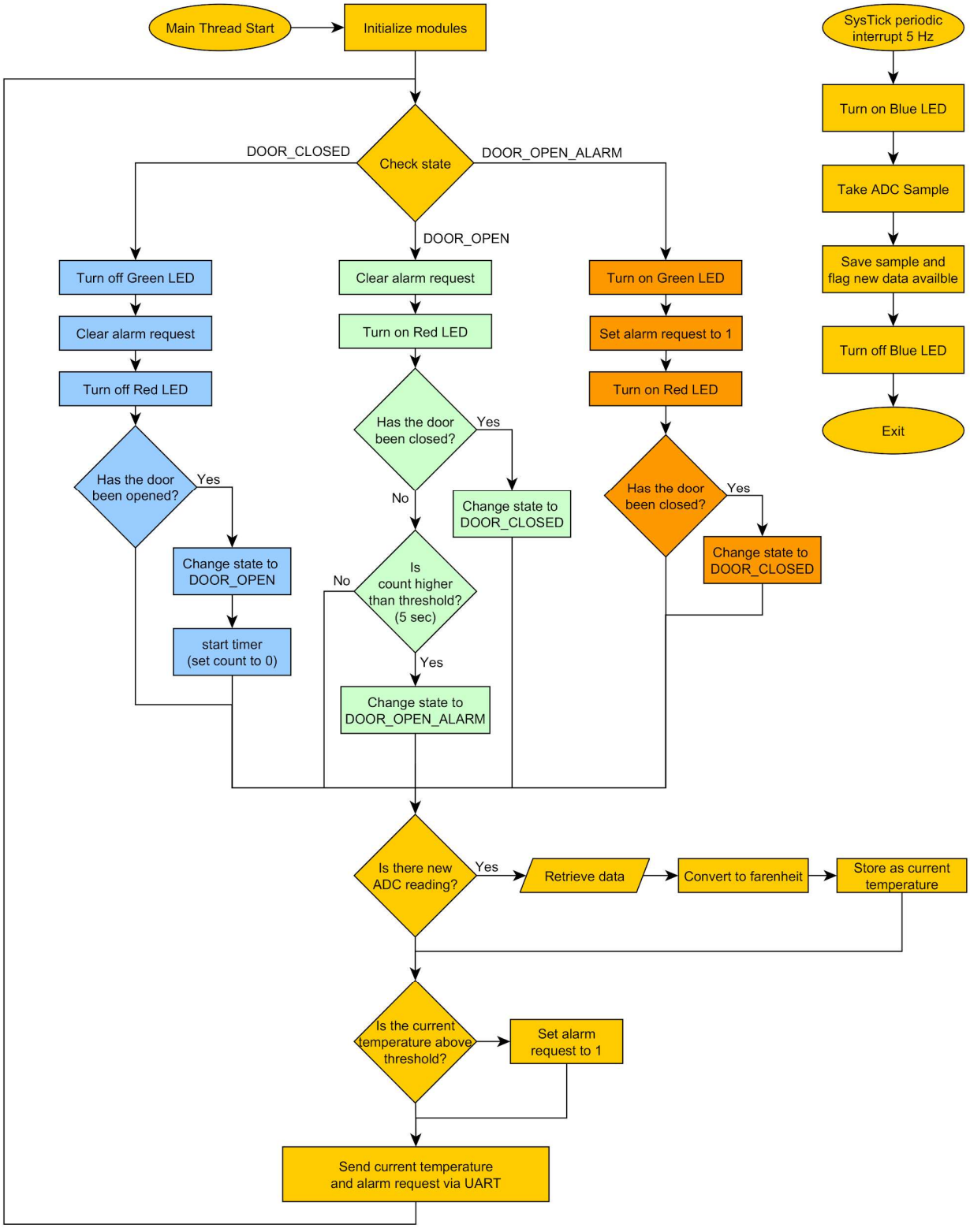
For a passive temperature sensor, the accuracy of the reading at the immediate time is of more importance than every single reading being sent. For this reason, we decided our transmit function to only transmit data if the UART is currently idle. The way it does this is upon being called by the main program (and accepting temperature data and alarm request), it checks if the transmit FIFO is empty, and if so, it formulates a unsigned 8-bit integer with the lower 7 bits temperature in Fahrenheit and the most significant bit, and then writes this data package to the FIFO. If the FIFO is not empty, then the previous data has not yet been sent, and so it exits, discarding the data. Since the bus speed runs at 80 MHz and the frequency of new ADC readings is only 5 Hz, the same data is inevitably repeatedly sent anyway even in this lazy transmission model. But more importantly, by sending only the most recent data, there is less time lag between the reading of the data by the sensor unit and display of the data by the interface unit.




## Door Switch

The door is a simple negative logic switch--if the door is closed, the switch is closed, so a raw data reading of 1 implies the door is closed and a reading of 0 implies the door is open. When the switch state changes, an interrupt is generated which sets a global doorStateChanged variable. When this flag is set, the main program knows to call a function which reads the current state of the door switch, via a function which returns 1 if the door is open and 0 otherwise. By calling this read function, the flag is also reset.

Total software flowchart for sensor unit



## Considerations for future iterations

While the main program could be implemented in ways other than a state machine, it would inevitably have to use several conditional statements that approach the same functionality, and so representing the problem as a state machine clarifies the issue and simplifies the project.

In future iterations, the green LED debug function for the door open alarm state can be taken out.

The decision for SysTick to have double-duty in both triggering ADC samples and acting as a general purpose timer for the open door alarm state violates the single responsibility principle in good software design and can introduce problems if, for example, the ADC sampling rate is changed. By only using one timer, however, this reduces energy consumption and simplifies the project in the sense that it reduces likelihood of programmer error in total system integration.

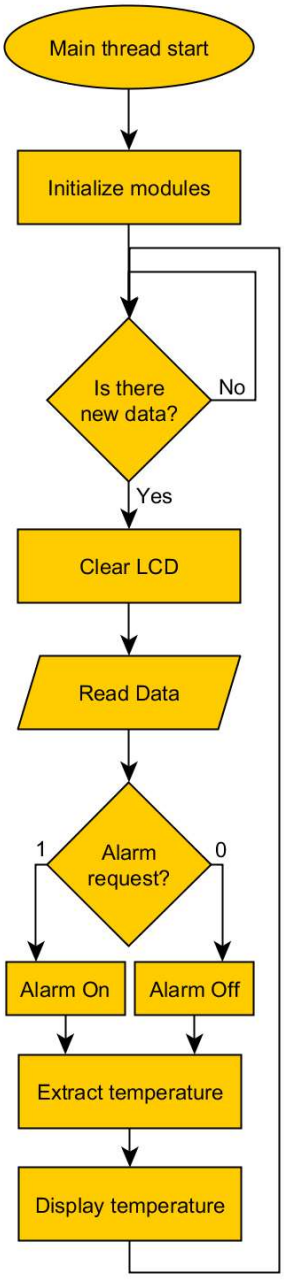
The largest limitation of this implementation is the fact that the temperature is an unsigned value, and so cannot handle values below 0°F. If temperatures below 0°F are needed, one possible implementation that uses 7 bits as currently is using a biased temperature, i.e., set 0 to correspond to -50°F and the max value of 127 shall then refer to 77°F and above, which is clearly outside the refrigerator functioning range and should therefore be more than sufficient to handle both refrigerators and freezers.

The linear fit we used is only appropriate in the room temperature range, and the thermistor characterization has nonlinear resistance with temperature especially outside of room temperature range, and so for future iterations the temperature conversion needs to be changed from a linear calculation to a 2nd- or 3rd-order polynomial at least. The Steinhart test mentioned in ADC and Thermistor Characterization under Development Plan, while potentially accurate, uses a logarithmic function, which has the added complexity of needing double precision data types in order to use the math.h standard library functions. As our microcontroller was hard-faulting on any attempt to use double type, we abandoned this approach in favor of implementing a simple linear model for demonstration and proof of concept. Should the double implementation be refactored and made to work, however, it could produce accurate results.

The decision to put the alarm request and the temperature data in the same frame has both advantages and drawbacks. The drawback is that it’s a less readily understood by any developer working with the interfacing device. The advantage is that every UART frame means the same thing. One alternative, by contrast, would be to send the alarm request in one frame and then the temperature in another frame, but this would drastically increase the complexity of communication. If there’s one clear viable alternative, it is that the alarm status might not be determined by the sensor unit, but instead by the interface unit. This reduces the sensor unit to have the single responsibility of collecting and transmitting data. The complication in this alternative is that the sensor unit would also have to transmit the door open or closed state in order for the time-based alarm for the door being left open to be determined by the interface unit, so the issue of multiple data needing to be sent per frame would not actually be solved, but this alternative may still be preferable as it allows the interface to be the sole point of configurability of thresholds when this feature is implemented, while maintaining the simplicity of simplex communication.

The main loop currently maintains its state for many cycles before changing to a new state, and repeat data is sent over the communication wire, which is not energy efficient. A change that should be made in future iterations is moving to a completely interrupt-based system using SysTick periodic interrupts to take the ADC sample, do any processing, and transmit the data. This will enable the interstice to run in a low-power wait mode.

# Tiva 2 (Interface unit)

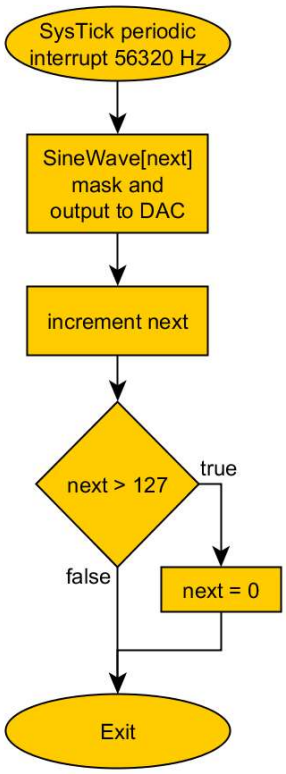



## Main Program

The main program busy waits on a function reading the UART Rx FIFO empty flag to detect when new data is available.

When new data is available, it is read and processed.

- The alarm request bit (bit 7) is read and the alarm is turned on or off accordingly.
- The temperature data (bits 0-6) is written to the LCD as a decimal number with a degF unit specifier.




## Alarm

The alarm is produced by a 5-bit DAC outputting an approximated sine wave to an audio device. It does this by iteratively outputting values to the DAC from a table. SysTick periodic interrupts trigger each new value output to the DAC.

The table consists of 128 values from 0-31 approximating one period of a sine wave. New values are output at a frequency of 56.320 KHz. Therefore the frequency of the sine wave output is 440 Hz.

When AlarmOn is called, the SysTick reload value is reset to 1419, producing the 56.320 KHz. The outmask is also reset to allow all DAC bits to pass through.

When AlarmOff is called, the maximum reload value for SysTick is set and the outmask is set to 0. This forces each DAC output to be zero.

## LCD

The LCD uses the supplied Nokia 5110 library.

#### Considerations for future iterations

Since the main loop only busy waits on UART communication, this can be worked to be interrupt-driven for much lower energy consumption.

# Development Plan

## Planned Development

#### Acquire parts

All parts not provided in class (such as the Nokia LCD, 3.5mm jack plug, and resistor network necessary to create a DAC) were purchased from the Chico IEEE chapter. This includes the thermistor, BJT transistors for the 7 segment display multiplexer, and various resistors used throughout the project.

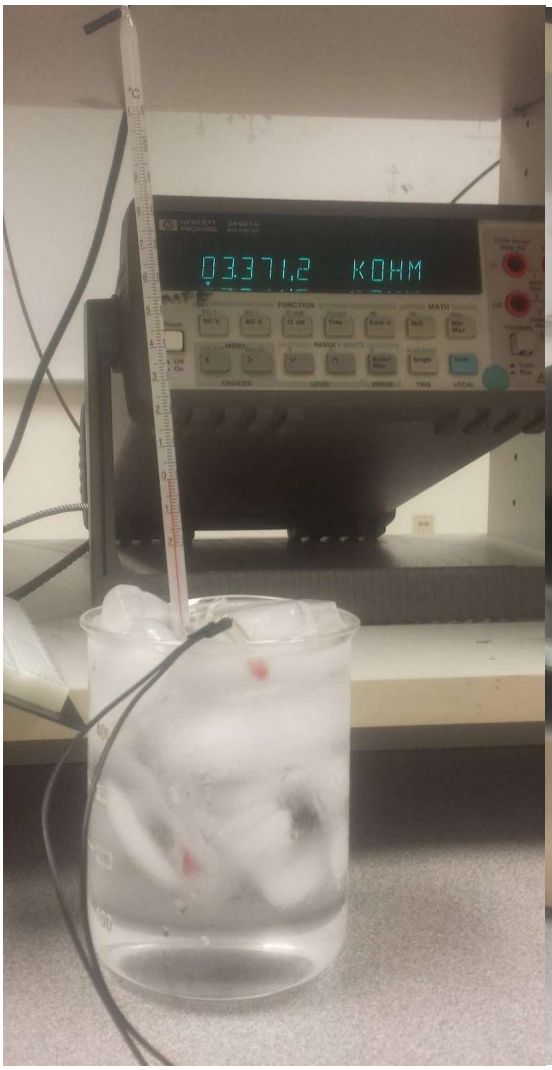
#### Thermistor circuit

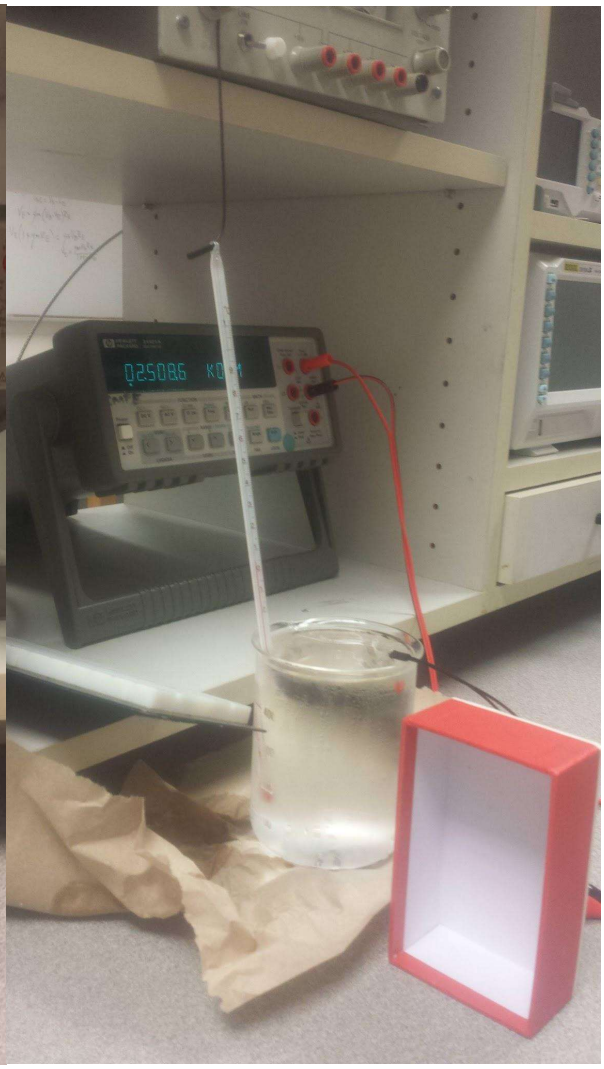
In order to use the thermistor as a temperature sensor, we could have used two primary circuits; either we could use the variable resistive properties of the thermistor to short a circuit path to trigger an input when temperature exceeded a predetermined value (since the thermistor resistance decreases with rising temperature), or we could use a voltage divider to generate a predictable voltage in a range as the temperature changes. The latter involves using the Tiva's ADC to collect data rather than simply using a "boolean" GPIO input pin, but allows us to give the user more feedback regarding the system.

#### 7 segment display circuit

Since the temperatures we sought to display were all two digit decimal values (though an actual installed unit could manage with one digit numbers if measuring in Celsius), we needed a way to drive at least two 7 segment displays. Instead using at least 14 pins on the Tiva board, it made more sense to only use 9 and multiplex the displays, that is alternate displaying data on one, then the other, at a rate faster than humans can perceive, such that both displays appear to be on. In order to do this, the selection pins on each of the 7 segment displays are tied together and sent data. Then the common anode of one display is sent a signal while the anode of the other display is pulled to ground. This anode switching behavior is accomplished using two BJT transistors (though MOSFETS have better switching performance, I believe it would be negligible in this context). The transistors are just toggled between the on and off states (it should not matter if the transistor is in the saturation region or the forward active region since all of the collector current will be directed to the emitter path in the circuit) to power each LED array one at a time.

ADC and Thermistor Characterization



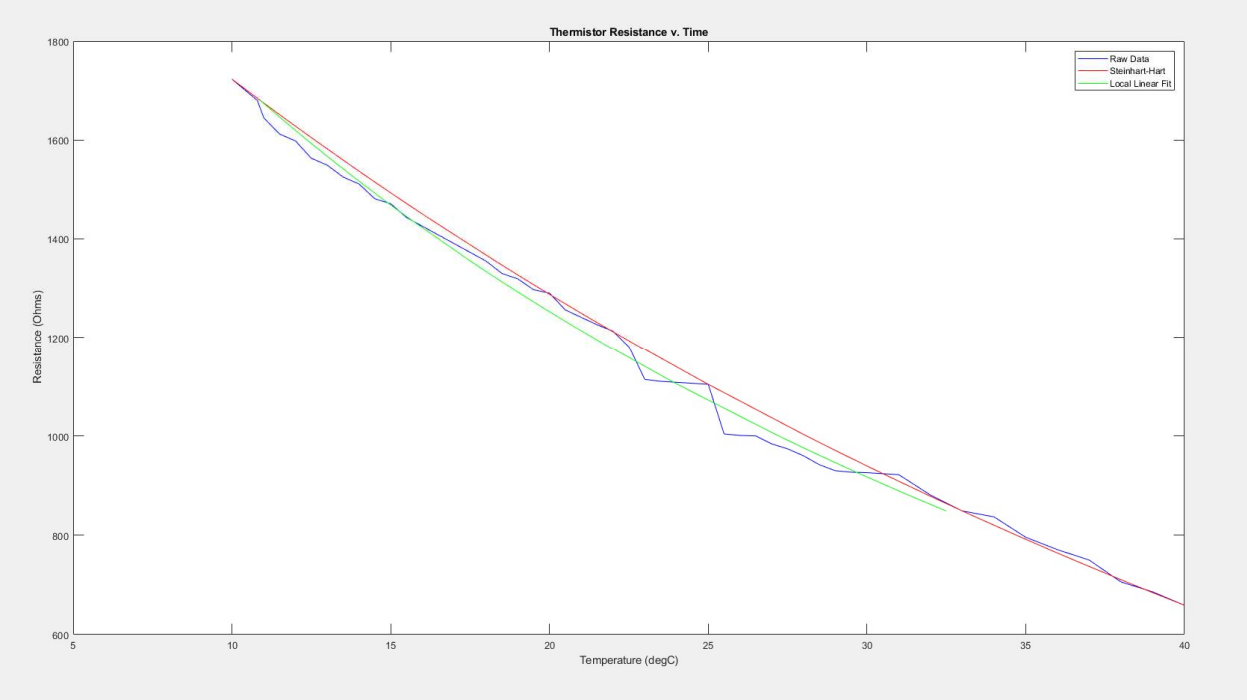
We did not have a data sheet for the thermistor we used. In order to characterize the thermistor, we used a beaker of water and a thermometer, and starting from ice water, took periodic measurements as the water warmed. Once the water reached room temperature (about 25°C), the beaker was emptied and refilled with boiling water. We waited for the temperature to lower well below 100°C to protect the insulation (hot glue, with a melting point near 120°C) that kept the thermistor from shorting in the water. Measurements were then periodically taken until the water settled at room temperature once again.

**All Measurements**

| Temp (°C) | Resistance (Ω) | | Temp (°C) | Resistance (Ω) | | Temp (°C) | Resistance (Ω) |
|---|---|---|---|---|---|---|---|
| 0.8 | 3909 | | 22.5 | 1181 | | 48 | 531 |
| 4.5 | 2620 | | 23 | 1114 | | 49 | 525 |
| 5 | 2375 | | 23.5 | 1110 | | 50 | 514 |
| 5.5 | 2221 | | 24 | 1108 | | 51 | 495 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 2187 | | 24.5 | 1106 | | 52 | 481 |
| 6.5 | 2102 | | 25 | 1104 | | 53 | 469 |
| 7 | 1994 | | 25.5 | 1004 | | 54 | 452 |
| 7.8 | 1923 | | 26 | 1001 | | 55 | 439 |
| 8 | 1894 | | 26.5 | 1000 | | 56 | 428 |
| 8.5 | 1854 | | 27 | 984 | | 57 | 415 |
| 9 | 1811 | | 27.5 | 974 | | 58 | 411 |
| 9.5 | 1770 | | 28 | 960 | | 59 | 394 |
| 10 | 1723 | | 28.5 | 942 | | 60 | 382 |
| 10.8 | 1680 | | 29 | 930 | | 61 | 378 |
| 11 | 1645 | | 29.5 | 927 | | 62 | 365 |
| 11.5 | 1612 | | 30 | 926 | | 63 | 355 |
| 12 | 1598 | | 31 | 922 | | 64 | 348 |
| 12.5 | 1563 | | 32 | 881 | | 65 | 342 |
| 13 | 1549 | | 33 | 849 | | 66 | 334 |
| 13.5 | 1525 | | 34 | 837 | | 67 | 326 |
| 14 | 1511 | | 35 | 796 | | 68 | 322 |
| 14.5 | 1481 | | 36 | 771 | | 69 | 314 |
| 15 | 1471 | | 37 | 750 | | 70 | 311 |
| 15.5 | 1443 | | 38 | 706 | | 71 | 305 |
| 18 | 1355 | | 39 | 686 | | 72 | 298 |
| 18.5 | 1330 | | 40 | 659 | | 73 | 295 |
| 19 | 1319 | | 41 | 641 | | 74 | 287 |
| 19.5 | 1297 | | 42 | 618 | | 75 | 285 |
| 20 | 1290 | | 43 | 603 | | 76 | 282 |
| 20.5 | 1256 | | 44 | 576 | | 77 | 279 |
| 21 | 1241 | | 45 | 563 | | 78 | 275 |
| 21.5 | 1226 | | 46 | 550 | | | |
| 22 | 1213 | | 47 | 538 | | | |

This data was analyzed on a spreadsheet program, but the best-fit curve approximation was too high of an order to be useful. Instead we used a common thermistor equation, a simplified form of the Steinhart-Hart equation $\frac{1}{T} = A + Bln(R) + C[ln(R)]^3$, where *A, B, C* are experimentally determined coefficients that tend to be small and require high precision. This relationship gave a much closer approximation at a considerably lower order. Unfortunately, we encountered errors dealing with the doubles needed to store the Steinhart coefficients, so we instead took a small operating region and created a linear model to use for ADC conversion. The following MatLab plot demonstrates the accuracy of the various approximations.




## Write software

Since the software uses many parts from previous lab assignments, much of the peripheral code was recycled from previous projects in the first version of the software. The ADC module was kept identical as we did not switch the pin or any options in initialization other than the sample frequency in setting up SysTick interrupts.

While we used previous project files as a base for each unit, the main program in each was written from scratch.

While some UART libraries were provided, we opted to write our own UART functions, especially as we were sending specialized packages of data rather than generic ascii data. Our design also calls for only sending data when the UART is idle, which is unsupported by the provided UART library. Our first choice for the UART modules were UART3 for both boards, PC7 on the sensor unit and PC6 on the interface unit. Both of these had to change before all the hard fault issues were resolved.

Apart from serious bugs in system integration, all the software modules operated, independently, as expected reasonably quickly. Some defects were injected into the code, such as:

- Forgetting to clear the interrupt flag of the door switch
- Testing a masked value against 1 instead of 1 shifted

These errors were caught during simulator debugging before causing any significant issue.

## Debugging

### Bug-fixing ADC and temperature sensing

The steinhart test module, using a natural log, required, in its simplest possible implementation, using double precision floating point data type. We attempted implementing it this way, but every time the software ran on the board it would hard fault the first time it entered the temperature conversion function. Hypothesizing that double floating point type was the cause of the hard fault, we decided to postpone implementation of the Steinhart test model and try to find a linear model for demonstration purposes.

For demonstration, it was assumed we would be testing the device under room temperature conditions, and so we found a linear fit on the range of 52°F to 90°F using the LINEST function of google sheets, comparing measured degrees Fahrenheit versus theoretical ADC sample based on measured resistance and computed voltage across a voltage divider. Of the 39 data points we had in this range, only one conversion had an error greater than 3% between the measured temperature and the modelled temperature, and we believe this is due to measurement error.

The LINEST function returned the line

$$degF = (0.05669595055)*ADC + (-34.0350707)$$

To rewrite the conversion using fixed-point format, we decided on a binary fixed point precision of 16 bits, because both the slope and the range of inputs is small. This resulted in the following conversion:

$$degF = ((3716)*ADC + (-2230522)) / 65536$$

We believe this model accurately reads temperatures in the room temperature range.

### General bug-fixing

Since one of our most common errors were unexplained hard faults, and that fixing this error commonly required switching pins of different modules, we decided that in order to speed up development we needed generalized initialization such that changing the pin or port of a peripheral only needed one line of code in the main program rather than changing the entire implementation file to accommodate the hardware change. To facilitate this, we first wrote a module called FriendlyWriter which reads and writes data to a provided address. Using this, we implemented dynamic address lookup with base addresses and register offsets and wrote logic that navigates the particular setup of the Tiva board--for example, in the door switch generalized initialization, if port F is selected, it activates interrupt 30 in the NVIC, but if anything else is selected, it can be linearly determined from port A at interrupt 0, port B at interrupt 1, etc.

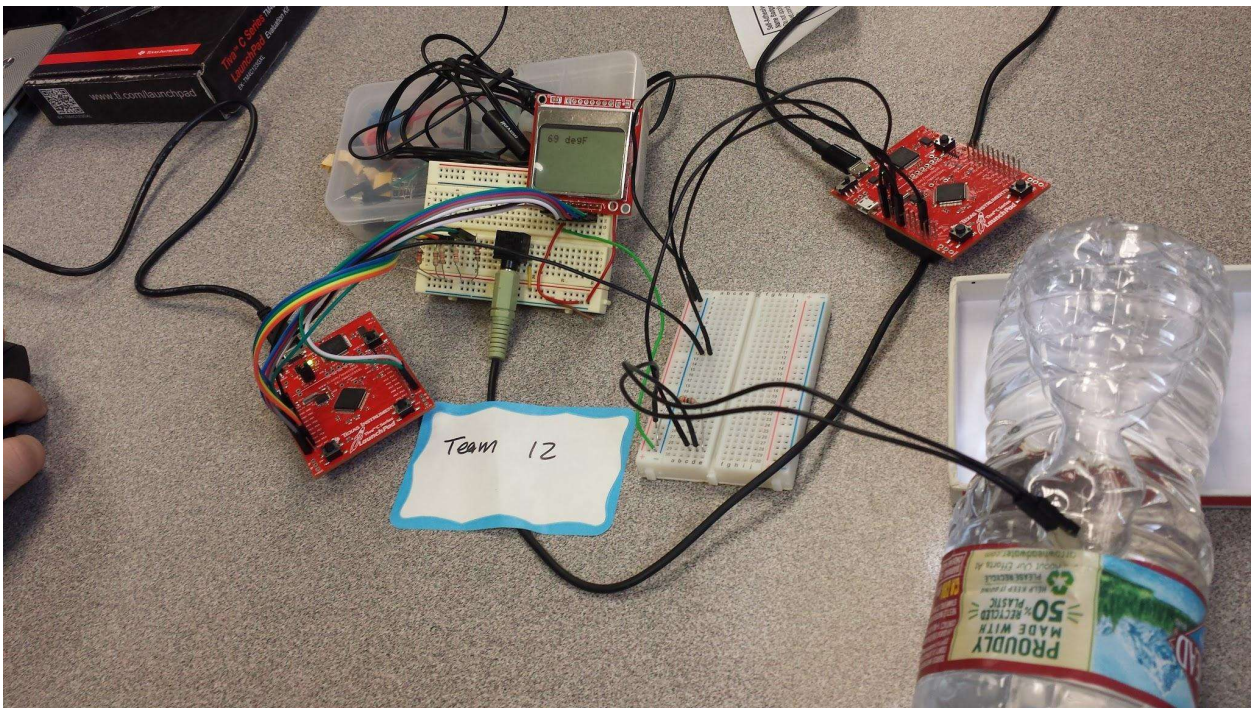
This had positive results. Our initial selection for the door switch was PA2,  but after this caused problems with the ADC initialization, we switched to PF0 relatively quickly. Since the UART

initialization was generalized, we could import the new library into the interface unit and immediately use it. Thus with a single line of code, we switched from using UART3 Rx on PC6 to UART1 Rx on PB0, which resolved another hard fault integration bug.

We also found that hard faults were being caused by, or, at least, being resolved by, switching the order of initialization. For example, the last hard fault bug in the interface unit was resolved by moving the Nokia LCD initialization and the UART initialization to before the PLL initialization.

At one point, we were reading strange values from the UART communication before realizing there was not a common ground at that point. Upon reconnecting the common ground, UART worked as expected.

### First Iteration results




The device works as designed. We demonstrated using a temperature threshold of 80 deg F, and the device both passively displays the current temperature on the LCD screen as well as outputs an alarm if either the temperature rises above 80 deg F or the PF0 switch on the sensor unit Tiva (modeeling the door) is pressed (left open) for more than 5 seconds.

# Conclusions

The fact that our major issues were caused by internal behavior of the Tiva rather than problems we injected into the product ourselves emphasizes the need for fail-fast development and it highlights the value of tools that can enable rapid turnaround upon failure and assist in debugging. It also highlights that modules sharing clocks and other features of the board must represent a common problem in developing embedded systems.

Other than this, the only new thing about this project for us was UART communication, which we didn't have any problems with directly.

The first improvement of this product is a more appropriate thermistor that more accurately reads temperature at the -10°F to 50°F range and with less noise. Using a better thermistor, we could then accurately characterize it using a polynomial model, or by finding a way to implement the Steinhart test model to calculate temperature. Related to this, by taking the temperature data to a bias, we can represent negative values.

The next most useful change to make to this product in order to bring it to salable status is configurability of thresholds. This might best be accomplished by moving the threshold comparison to the interface unit and have the interface unit accept configuration by the user. This would mean the data package can be 7 bits of temperature data and 1 bit of door status instead of the high bit being an alarm request, and doing it this way maintains simplex communication.

Finally, we would decide on a reliable power source as well as a housing for the sensor unit.

Optionally, getting the 7 segment display to work might be better than the LCD because it would entail lower energy consumption. It might interfere with how configurability is implemented for the user, though.

# Credits and Acknowledgments


- Hardware
    - Acquiring parts - Michael Musick
    - Designing circuitry - Michael Musick
    - Assembling circuitry - Michael Musick
- Temperature sensing
    - Temperature sensor development planning - Michael Musick
    - ADC and thermistor characterization measurements - Michael Musick and Gavin Fielder
    - ADC sample conversion, Steinhart test model - Michael Musick
    - ADC sample conversion, linear model for demonstration - Gavin Fielder
- Writing software
    - Program flow design - Gavin Fielder
    - Sensor unit main - Gavin Fielder
    - Interface unit main - Gavin Fielder
    - ADC driver - Michael Musick
    - DAC alarm driver - Gavin Fielder
    - Door switch driver - Gavin Fielder
    - Sensor unit UART driver, first attempt - Gavin Fielder
    - Interface unit UART driver, first attempt - Gavin Fielder
    - FriendlyWriter module - Gavin Fielder

    - Generalized initialization - Gavin Fielder
    - 7-Segment display driver - Michael Musick
- Debugging
    - System integration refactoring and general debugging - Michael Musick and Gavin Fielder
    - Dr. Mustafa for various advice
- Documentation
    - Introduction, Motivation - Gavin Fielder
    - Related Work - Gavin Fielder
    - Specification - Gavin Fielder
    - Design
        - Hardware section - Michael Musick
        - Software section - Gavin Fielder
    - Development plan
        - Acquire parts section - Michael Musick
        - Design circuitry section - Michael Musick
        - ADC and Thermistor Characterization section - Michael Musick
        - Write Software section - Gavin Fielder
        - 7-Segment display section - Michael Musick
        - Bug-fixing ADC and temperature sensing section - Gavin Fielder
        - General bug-fixing section - Gavin Fielder
    - Conclusions - Michael Musick and Gavin Fielder
    - Credits and Acknowledgements - Gavin Fielder

# Appendix: Code

## Common Libraries

### friendlyWriter.c

```
/**
 * @file friendlyWriter.c
 * @brief This file holds general purpose read/writer functions
 *
 * @author Gavin Fielder
 * @date 5/12/2018
 */

#include "stdint.h"
#include "friendlywriter.h"

//Ports
const uint32_t GPIO_BASE[6] = {   0x40004000,   //port A
                                  0x40005000,   //port B
```

```
                                    0x40006000,   //port C
                                    0x40007000,   //port D
                                    0x40024000,   //port E
                                    0x40025000 }; //port F

//Writes to an address
void Write(uint32_t address, uint32_t value) {
    (*((volatile unsigned long *)address)) = value;
}

//Writes to an address using friendly code
void FriendlyWrite(uint32_t address, uint32_t value, uint32_t onlyOn) {
    (*((volatile unsigned long *)address)) |= (value & onlyOn);
    (*((volatile unsigned long *)address)) &= (value | (~onlyOn));
}

//Reads an address
unsigned int Read(uint32_t address) {
    return (*((volatile unsigned long *)address));
}
```

uart.c

```
/**
 * @file uart.c
 * @brief This file holds generalized initialization for
 *     UART simplex communication
 *
 * @author Gavin Fielder
 * @date 5/12/2018
 *
 * Hardware Connections
 *     Thermistor with 1K ohm resister voltage divider on PE2
 *     UART Tx on PB1
 */

#include "stdint.h"
#include "tm4c123gh6pm.h"
#include <stdio.h>
#include <stdlib.h>
#include "uart.h"
#include "friendlywriter.h"

//Ports
extern const uint32_t GPIO_BASE[6];

//UART module base addresses
```

```
const uint32_t UART_BASE[8] = {   0x4000C000,   //UART0
                                  0x4000D000,   //UART1
                                  0x4000E000,   //UART2
                                  0x4000F000,   //UART3
                                  0x40010000,   //UART4
                                  0x40011000,   //UART5
                                  0x40012000,   //UART6
                                  0x40013000 }; //UART7
//UART Register offsets
#define CTL_R0x030
#define DR_R 0x000
#define IBRD_R   0x024
#define FBRD_R   0x028
#define LCRH_R   0x02C
#define CC_R 0xFC8
#define FR_R 0x018

//Saved UART Number
unsigned int selected_UART;

/**
 * Generalized UART Initialization
 *
 * @param  portNum   PA, PB, PC, PD, PE, or PF (defined in header)
 * @param  pinNum   0-7 pin select
 * @param  UARTNum   0-7 UART module select
 * @param  dir      Tx or Rx (defined in header)
 */
void UART_Init(unsigned int portNum, unsigned int pinNum,
              unsigned int UARTNum, unsigned int dir) {

       int delay;
       unsigned int GPIO_Using = 1 << pinNum;
       unsigned int PCTLValue = 1;
       unsigned int RCGCAddress;
       unsigned int DIRValue = dir << pinNum;
       if (UARTNum < 3)
       RCGCAddress = 0x400FE104; //use legacy SYSCTL_RCGC1_R
       else
       RCGCAddress = 0x400FE618; //current SYSCTL_RCGCUART_R

       //PC4 and PC5 on UART1 get PCTL of 2
       if (portNum == 2 && UARTNum == 1 && (pinNum == 4 || pinNum == 5))
       PCTLValue = 2;

       //Send clock signals
       FriendlyWrite(RCGCAddress, 1 << UARTNum, 1 << UARTNum); //Enable UART clock
       for (delay = 0; delay < 100; delay++);
```

```
	SYSCTL_RCGCGPIO_R |= (1 << portNum); //Enable GPIO Clock
	for (delay = 0; delay < 10; delay++);

	//Initialize GPIO pin
	FriendlyWrite(GPIO_BASE[portNum] + DIR_R, DIRValue, GPIO_Using); //DIR
	FriendlyWrite(GPIO_BASE[portNum] + AFSEL_R, 1 << pinNum, GPIO_Using); //AFSEL
	FriendlyWrite(GPIO_BASE[portNum] + AMSEL_R, 0, GPIO_Using); //AMSEL
	FriendlyWrite(GPIO_BASE[portNum] + DEN_R, 1 << pinNum, GPIO_Using); //DEN
	FriendlyWrite(GPIO_BASE[portNum] + PCTL_R, PCTLValue << (pinNum * 4), (0xF <<
(pinNum * 4))); //PCTL

	//Initialize UART at 9600 baud rate for 80 MHz clock
	FriendlyWrite(UART_BASE[UARTNum] + CTL_R, 0, 1); //disable during setup
	Write(UART_BASE[UARTNum] + IBRD_R, 520); //integer part of baud rate clock
divisor
	Write(UART_BASE[UARTNum] + FBRD_R, 53); //fraction part of baud rate clock
divisor
	Write(UART_BASE[UARTNum] + LCRH_R, 0x70); //enable FIFO and set word length to
8
	Write(UART_BASE[UARTNum] + CC_R, 0); //select system clock source
	FriendlyWrite(UART_BASE[UARTNum] + CTL_R, 1, 1); //enable UART

	//Save settings
	selected_UART = UARTNum;
}

//Transmits data if the UART is idle
void UART_TransmitIfIdle(uint8_t alarmStatus, uint8_t temperature) {
	uint8_t data = (alarmStatus << 7) | temperature;
	if (Read(UART_BASE[selected_UART] + FR_R) & 0x80) { //if FIFO empty
	(*((volatile unsigned char *)(UART_BASE[selected_UART] + DR_R))) = data;
	}
}


//Returns 1 if data exists in FIFO, 0 otherwise
uint8_t UART_NewDataAvailable() {
	if ((Read(UART_BASE[selected_UART] + FR_R) & 0x10) == 0) {
		//FIFO NOT empty. Return 1 for new data detected
		return 1;
	}
	else return 0;
}

//Extracts data from FIFO
uint8_t UART_Read(void) {
	return (Read(UART_BASE[selected_UART] + DR_R) & 0xFF);
```

```
}
```

## Sensor Unit

### Main Program

```
/**
 * @file Lab6.c
 * @brief This file holds the main program for the sensor unit
 *     of the fridge monitoring system
 *
 * @author Gavin Fielder
 * @date 5/12/2018
 *
 * Hardware Connections
 *     Thermistor with 1K ohm resister voltage divider on PE2
 *     UART Tx on PB1
 */

#include "PLL.h"
#include "ADC.h"
#include "tm4c123gh6pm.h"
#include "Lab6peripherals.h"
#include <stdio.h>
#include "math.h"
#include "uart.h"
#include "doorSwitch.h"

void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
long StartCritical (void);// previous I bit, disable interrupts
void EndCritical(long sr);// restore I bit to previous value
void WaitForInterrupt(void);  // low power mode

//Global variables
extern unsigned int ADCMail;
extern unsigned int ADCStatus;
unsigned int Data;        // 12-bit ADC
unsigned int temperature;
extern int doorStateChanged;

//Function prototypes
unsigned int Temp_Convert(unsigned int data);

//Global
int alarm; //1 if alarm should be on, 0 if alarm should be off
extern int doorStateChanged; //door switch change event flag
extern unsigned int count; //incremented by systick
```

```
//Simple state machine
#define DOOR_CLOSED 0
#define DOOR_OPEN 1
#define DOOR_OPEN_ALARM 2
#define ALARM_TIME_THRESHOLD 25 //number of systick periods
#define TEMPERATURE_THRESHOLD 80 //deg F
int state;

int __main(void){

	DisableInterrupts();	//Disable global interrupts

	//Initializations
	PLL_Init();		//Bus clock is 80 MHz
	SysTick_Init();	//5 Hz periodic interrupt
	DoorSwitch_Init(PF,0); //enable door switch with interrupts
	PortF_Init(0xe);	//Enables PF1-3 LEDs
	ADC_Init();		//turn on ADC, set channel to 1 (PE2)
	UART_Init(PB,1,1,Tx); //UART Tx on PB1. 9600 baud rate.
	state = DOOR_CLOSED; //initial state
	count = 0;

	EnableInterrupts();	//Enable global interrupts

	while(1){
	//Handle the current state and see if transitions need to be made
	if (state == DOOR_CLOSED) {
		GPIO_PORTF_DATA_R &= ~0x8; //Debug alarm status indicator LED
		alarm = 0; //can be overridden by temperature reading
		DoorOpenIndicator_Off();
		if ((doorStateChanged) && (IsDoorOpen())) {
		count = 0; //start timer
		state = DOOR_OPEN;
		}
	}
	else if (state == DOOR_OPEN) {
		alarm = 0; //can be overridden by temperature reading
		DoorOpenIndicator_On();
		if ((doorStateChanged) && (!(IsDoorOpen())))
		state = DOOR_CLOSED;
		else if (count > ALARM_TIME_THRESHOLD)
		state = DOOR_OPEN_ALARM;
	}
	else if (state == DOOR_OPEN_ALARM) {
		GPIO_PORTF_DATA_R |= 0x8; //Debug alarm status indicator LED
		alarm = 1;
		if ((doorStateChanged) && (!(IsDoorOpen())))
```

```
			state = DOOR_CLOSED;
		}
		//Check status of ADC
		if (ADCStatus) {
			ADCStatus = 0;
			Data = ADCMail;
			temperature = Temp_Convert(Data);
		}
		//If temperature is above the threshold, alarm is always 1
		if (temperature > TEMPERATURE_THRESHOLD)
			alarm = 1;
		//If the FIFO is ready to transmit, send current data
		UART_TransmitIfIdle(alarm, temperature);
		}


		return 0;
}

//----------Temp_Convert---------
// Converts 12-bit digital ADC value into useable data
// Input: 12-bit result of ADC conversion
// Output: temperature value in farenheit
unsigned int Temp_Convert(unsigned int data){
		return (3716 * data - 2230522) / 65536;
}
```

## ADC.c

```
/**
 * @file ADC.c
 * @brief This file holds the ADC initialization
 *
 * @author Gavin Fielder
 * @date 5/12/2018
 *
 * Hardware Connections
 *		Thermistor with 1K ohm resister voltage divider on PE2
 *		UART Tx on PB1
 */

#include "tm4c123gh6pm.h"


// This initialization function
// Max sample rate: <=125,000 samples/second
// Sequencer 0 priority: 1st (highest)
// Sequencer 1 priority: 2nd
// Sequencer 2 priority: 3rd
```

```
// Sequencer 3 priority: 4th (lowest)
// SS3 triggering event: software trigger
// SS3 1st sample source: Ain1 (PE2)
// SS3 interrupts: flag set on completion but no interrupt requested
void ADC_Init(void){
	unsigned int PE2;
	int delay;
	SYSCTL_RCGC2_R |= 0x10;
	PE2 = 0x04;
	//Configure as analog input
	GPIO_PORTE_DIR_R &= ~PE2;
	GPIO_PORTE_AFSEL_R |= PE2;
	GPIO_PORTE_DEN_R &= ~PE2;
	GPIO_PORTE_PCTL_R &= ~PE2;
	GPIO_PORTE_AMSEL_R |= PE2;

	SYSCTL_RCGCADC_R |= 1;
	SYSCTL_RCGCADC_R &= ~2;
	for (delay = 0; delay < 50; delay++);  //needs longer delay to not hard fault
	ADC0_PC_R = 0x1;
	ADC0_SSPRI_R = 0x3210;		//8) Set sequencer priority
	ADC0_ACTSS_R &= ~0x8;		//9) Disable sequencer 3
	ADC0_EMUX_R &= ~0xF000;		//10) Set software trigger
	ADC0_SSMUX3_R = 0x1;		//11) Select channel AIN1
	ADC0_SSCTL3_R = 0x06;		//12) enable interrupt, flag last step
	ADC0_ACTSS_R |= 0x08;		//13) Enable sequencer 3

}

//------------ADC_In------------
// Busy-wait Analog to digital conversion
// Input: none
// Output: 12-bit result of ADC conversion
unsigned int ADC_In(void){
	unsigned int data;
	ADC0_PSSI_R |= 0x08; //Start ADC
	while ((ADC0_RIS_R & 0x08)==0);
	data = ADC0_SSFIFO3_R & 0xFFF;
	ADC0_ISC_R |= 0x08; //Clear flag
	return data;
}
```

doorSwitch.c

```
/**
 * @file doorSwitch.c
 * @brief This file holds the generalized door switch functions
 *
 * @author Gavin Fielder
 * @date 5/12/2018
 *
 * Hardware Connections
```

```
 *     Thermistor with 1K ohm resister voltage divider on PE2
 *     UART Tx on PB1
 */


#include "tm4c123gh6pm.h"
#include "stdint.h"
#include "friendlywriter.h"

//Globals
int doorStateChanged;
unsigned int selected_port;
unsigned int selected_pin;

//Ports
extern const uint32_t GPIO_BASE[6];


// Initializes a pin with both-edge triggered interrupts
void DoorSwitch_Init(unsigned int portNum, unsigned int pinNum){
	int delay;
	unsigned int GPIO_Using = 1 << pinNum;
	SYSCTL_RCGCGPIO_R |= ( 1 << portNum);
	delay = 0;
	//Configure as GPIO
	Write(GPIO_BASE[portNum] + LOCK_R, GPIO_LOCK_KEY);
	FriendlyWrite(GPIO_BASE[portNum] + CR_R, GPIO_Using, GPIO_Using);
	FriendlyWrite(GPIO_BASE[portNum] + DIR_R, 0, GPIO_Using);
	FriendlyWrite(GPIO_BASE[portNum] + AFSEL_R, 0, GPIO_Using);
	FriendlyWrite(GPIO_BASE[portNum] + DEN_R, GPIO_Using, GPIO_Using);
	FriendlyWrite(GPIO_BASE[portNum] + PCTL_R, 0, (0xF << (pinNum*4)));
	FriendlyWrite(GPIO_BASE[portNum] + AMSEL_R, 0, GPIO_Using);

	//Configure Interrupts
	FriendlyWrite(GPIO_BASE[portNum] + IM_R, 0, GPIO_Using); // Disarm the
interrupt
	FriendlyWrite(GPIO_BASE[portNum] + PUR_R, GPIO_Using, GPIO_Using); //enable
pull up resistor
	FriendlyWrite(GPIO_BASE[portNum] + IS_R, 0, GPIO_Using); // is edge-sensitive
	FriendlyWrite(GPIO_BASE[portNum] + IBE_R, GPIO_Using, GPIO_Using); // is both
edges
	FriendlyWrite(GPIO_BASE[portNum] + IEV_R, 0, GPIO_Using); //
	Write(GPIO_BASE[portNum] + ICR_R, GPIO_Using); // clear flag
	FriendlyWrite(GPIO_BASE[portNum] + IM_R, GPIO_Using, GPIO_Using); //Enable the
interrupt


	//Enable NVIC
```

```
	if (portNum == PF)
		NVIC_EN0_R |= (1 << 30);
	else
		NVIC_EN0_R |= (1 << portNum);

	//Set priority 5
	if (portNum <= 3)
	FriendlyWrite(NVIC_PRI0_R, (5 << 5) << (portNum * 8), (0x7 << 5) << (portNum *
8));
	else if (portNum == 4)
		FriendlyWrite(NVIC_PRI1_R, (5 << 5), (0xe << 5));
	else
		FriendlyWrite(NVIC_PRI7_R, (5 << 5) << (24), (0x7 << 5) << (24));

	//Set initial flag value
	doorStateChanged = 0;

	//Save settings
	selected_port = portNum;
	selected_pin = pinNum;
}


//Returns 1 if door is open, 0 if door is closed
//Also resets doorStateChanged flag.
uint8_t IsDoorOpen() {
	if (Read(GPIO_BASE[selected_port] + DATA_R) & ( 1 << selected_pin) ) {
		//reading 1, door is closed
		doorStateChanged = 0;
		return 0;
	}
	else {
		doorStateChanged = 0;
		return 1;
	}
}

//Comment out the handlers that are unneeded

//Interrupt handler
//void GPIOPortA_Handler(void) {
//	doorStateChanged = 1;
//	GPIO_PORTA_ICR_R |= (1 << selected_pin);
//}

////Interrupt handler
//void GPIOPortB_Handler(void) {
//	doorStateChanged = 1;
```

```
//     GPIO_PORTB_ICR_R |= (1 << selected_pin);
//}

////Interrupt handler
//void GPIOPortC_Handler(void) {
//     doorStateChanged = 1;
//     GPIO_PORTC_ICR_R |= (1 << selected_pin);
//}

////Interrupt handler
//void GPIOPortD_Handler(void) {
//     doorStateChanged = 1;
//     GPIO_PORTD_ICR_R |= (1 << selected_pin);
//}

////Interrupt handler
//void GPIOPortE_Handler(void) {
//     doorStateChanged = 1;
//     GPIO_PORTE_ICR_R |= (1 << selected_pin);
//}

//Interrupt handler
void GPIOPortF_Handler(void) {
       doorStateChanged = 1;
       GPIO_PORTF_ICR_R |= (1 << selected_pin);
}
```

Lab6peripherals.c

```
/**
 * @file Lab6peripherals.c
 * @brief This file holds peripheral functions for
 *     the fridge monitoring system.
 *
 * @author Gavin Fielder
 * @date 5/12/2018
 *
 * Hardware Connections
 *     Thermistor with 1K ohm resister voltage divider on PE2
 *     UART Tx on PB1
 */


#include "Lab6peripherals.h"
#include "tm4c123gh6pm.h"
#include "ADC.h"
#include "friendlywriter.h"
```

```
//Global variables
unsigned long ADCMail;
unsigned int ADCStatus;
unsigned int count;

// **************SysTick_Init*********************
// Initialize Systick periodic interrupts
// Input: none
// Output: none
// Note: Enables SysTick periodic interrupts at 40 Hz
void SysTick_Init(){
       SYSTICK_STCTRL = 0; // disable SysTick during setup
       SYSTICK_STRELOAD = 0xF42400 - 1; //5 Hz at clock speed 80 MHz
       SYSTICK_STCURRENT = 0; // any write to current clears it
       NVIC_PRI3_R = (NVIC_PRI3_R&0x00FFFFFF)|0x40000000; //priority 2
       NVIC_EN0_R |= 0x00008000; //enable interrupt 15 (SysTick)
       SYSTICK_STCTRL = 0x07; // enable with system clock and interrupts
}

// Interrupt service routine
// Executed periodically, the actual period
// determined by the current Reload.
void SysTick_Handler(void){
       GPIO_PORTF_DATA_R |= 0x04; //Turn on LED
       ADCMail = ADC_In(); //get ADC input
       ADCStatus = 1; //flag that new output is ready
       GPIO_PORTF_DATA_R &= ~0x04; //Turn off LED
       count = count + 1;
}

// Initializes Port F for output
// Inputs: None
// Outputs: None
// Notes: Sets all used pins as outputs
void PortF_Init(unsigned int portF_using){
       SYSCTL_RCGC2_R |= 0x20; //send clock signal
       portF_using = portF_using; //delay
    GPIO_PORTF_LOCK_R = GPIO_LOCK_KEY;
    GPIO_PORTF_CR_R |= portF_using; //set bits
    GPIO_PORTF_PCTL_R &= ~portF_using; //clear bits
       GPIO_PORTF_DIR_R |= portF_using; //set bits for output
    GPIO_PORTF_PUR_R |= portF_using; //set bits
       GPIO_PORTF_AFSEL_R &= ~portF_using; //clear bits
       GPIO_PORTF_AMSEL_R &= ~portF_using; //clear bits
       GPIO_PORTF_DEN_R |= portF_using; //set bits
}
```

```
//Turns LED on
void DoorOpenIndicator_On() {
	GPIO_PORTF_DATA_R |= 0x02;
}

//Turns LED off
void DoorOpenIndicator_Off() {
	GPIO_PORTF_DATA_R &= ~0x02;
}
```

## Interface Unit

### Main Program

```
/**
 * @file Lab5.c This file holds the main program for the
 *           interface unit of the fridge monitoring system.
 *
 * @author Gavin Fielder
 * @date 5/12/2018
 *
 * Hardware Connections
 *     UART Rx on PB0
 *     5-bit DAC connected to PE1-5, output to audio device
 *           1.5 kohm PE5
 *           3.0 kohm PE4
 *           6.0 kohm PE3
 *           12.0 kohm PE2
 *           24.0 kohm PE1
 *     Nokia LCD on Port A
 *           // Red SparkFun Nokia 5110 (LCD-10168)
 *           // -------------------------------------
 *           // Signal    (Nokia 5110) LaunchPad pin
 *           // 3.3V      (VCC, pin 1) power
 *           // Ground    (GND, pin 2) ground
 *           // SSI0Fss   (SCE, pin 3) connected to PA3
 *           // Reset     (RST, pin 4) connected to PA7
 *           // Data/Command  (D/C, pin 5) connected to PA6
 *           // SSI0Tx    (DN,  pin 6) connected to PA5
 *           // SSI0Clk   (SCLK, pin 7) connected to PA2
 *           // back light(LED, pin 8) not connected
 */

#include "tm4c123gh6pm.h"
#include "PLL.h"
#include "stdint.h"
#include "Nokia5110.h"
```

```
#include "alarm.h"
#include "uart.h"
#include "stdio.h"

// basic functions defined at end of startup.s
void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
void WaitForInterrupt(void);  // Low power wait

int __main(void){
	char str[16];
	uint8_t data_in = 0;

	DisableInterrupts();

	UART_Init(PB,0,1,Rx); //Select PB0, UART1, Rx
	Nokia5110_Init();   //LCD
	PLL_Init();         // bus clock at 80 MHz
	InitializeAlarm();  //Initialize DAC, SysTick
	Nokia5110_SetCursor(0,0);
	Nokia5110_Clear();

	EnableInterrupts(); //Enable global interrupts

	while(1){
		//Check if new data received
		if (UART_NewDataAvailable()) {
			Nokia5110_Clear();
			Nokia5110_SetCursor(0,0);
			data_in = UART_Read();
			//Get alarm status (bit 7)
			if (data_in & 0x80)
			AlarmOn();
			else
			AlarmOff();
			//Read temperature data (bits 0-6)
			data_in &= ~0x80;
			//print data
			sprintf(str, "%u degF", data_in);
			Nokia5110_OutString(str);
		}
	}
}
```

alarm.c

```
/**
```

```
 * @file alarm.c
 * @brief This file holds functions for the alarm system
 *
 * @author Gavin Fielder
 * @date 5/12/2018
 *
 * Hardware Connections
 *     UART Rx on PB0
 *     5-bit DAC connected to PE1-5, output to audio device
 *            1.5 kohm PE5
 *            3.0 kohm PE4
 *            6.0 kohm PE3
 *            12.0 kohm PE2
 *            24.0 kohm PE1
 *     Nokia LCD on Port A
 *            // Red SparkFun Nokia 5110 (LCD-10168)
 *            // ---------------------------------
 *            // Signal    (Nokia 5110) LaunchPad pin
 *            // 3.3V      (VCC, pin 1) power
 *            // Ground    (GND, pin 2) ground
 *            // SSI0Fss   (SCE, pin 3) connected to PA3
 *            // Reset     (RST, pin 4) connected to PA7
 *            // Data/Command  (D/C, pin 5) connected to PA6
 *            // SSI0Tx    (DN,  pin 6) connected to PA5
 *            // SSI0Clk   (SCLK, pin 7) connected to PA2
 *            // back light(LED, pin 8) not connected
 */

#include "stdint.h"
#include "tm4c123gh6pm.h"
#include "alarm.h"

//Holds the sine wave for a 5-bit DAC
const uint16_t NUMBER_OF_DATA_POINTS = 128;
const uint8_t SineWave[NUMBER_OF_DATA_POINTS] = {
       0x10,0x10,0x11,0x12,0x13,0x13,0x14,0x15,
       0x15,0x16,0x17,0x17,0x18,0x19,0x19,0x1a,
       0x1a,0x1b,0x1b,0x1c,0x1c,0x1d,0x1d,0x1e,
       0x1e,0x1e,0x1e,0x1f,0x1f,0x1f,0x1f,0x1f,
       0x1f,0x1f,0x1f,0x1f,0x1f,0x1f,0x1e,0x1e,
       0x1e,0x1e,0x1d,0x1d,0x1c,0x1c,0x1b,0x1b,
       0x1a,0x1a,0x19,0x19,0x18,0x17,0x17,0x16,
       0x15,0x15,0x14,0x13,0x13,0x12,0x11,0x10,
       0x10,0xf,0xe,0xd,0xc,0xc,0xb,0xa,
       0xa,0x9,0x8,0x8,0x7,0x6,0x6,0x5,
       0x5,0x4,0x4,0x3,0x3,0x2,0x2,0x1,
       0x1,0x1,0x1,0x0,0x0,0x0,0x0,0x0,0x0,
       0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x1,0x1,
```

```
	0x1,0x1,0x2,0x2,0x3,0x3,0x4,0x4,
	0x5,0x5,0x6,0x6,0x7,0x8,0x8,0x9,
	0xa,0xa,0xb,0xc,0xc,0xd,0xe,0xf
};
//Sets frequency of sound
const uint32_t period = 1420;

//Tracks the next data point
int next;
//Masks the output to toggle signal on/off
uint32_t outmask;

//Initializes DAC and SysTick
void InitializeAlarm(){

	int delay;
	unsigned int portE_using = 0x3e;

	//Initialize PE1-5 as 5-bit DAC output
	SYSCTL_RCGC2_R |= 0x10; //send clock signal
	delay = 0;
	GPIO_PORTE_DIR_R |= portE_using; //set bits (output)
	GPIO_PORTE_AFSEL_R &= ~portE_using; //clear bits
	GPIO_PORTE_AMSEL_R &= ~portE_using; //clear bits
	GPIO_PORTE_DEN_R |= portE_using; //set bits

	//Initialize SysTick with periodic interrupts
	NVIC_ST_CTRL_R = 0; // disable SysTick during setup
	NVIC_ST_RELOAD_R = period - 1;
	NVIC_ST_CURRENT_R = 0; // any write to current clears it
	NVIC_PRI3_R = (NVIC_PRI3_R&0x00FFFFFF)|0x40000000; //priority 2
	NVIC_EN0_R |= 0x00008000; //enable interrupt 15 (SysTick)
	NVIC_ST_CTRL_R = 0x07; // enable with system clock and interrupts

	//initial value
	next = 0;
}

//Turns off alarm
void AlarmOff(void) {
	NVIC_ST_RELOAD_R = 0xFFFFFF;
	outmask = 0;
}

//Turns on alarm
void AlarmOn(void) {
	NVIC_ST_RELOAD_R = period - 1;
	outmask = 0x3e;
```

```
}

// Interrupt service routine
// Executed periodically, the actual period
// determined by the current Reload.
void SysTick_Handler(void){
	DAC_Out(SineWave[next] & (outmask >> 1)); //output next value
	next = next + 1; //set next index value
	if (next >= NUMBER_OF_DATA_POINTS) next = 0; //wrap around
}

// **************DAC_Out*********************
// output to DAC
// Input: 5-bit data, 0 to 31
// Output: none
// Note: outputs on PA2-PA6
void DAC_Out(unsigned long data) {
	GPIO_PORTE_DATA_R = data << 1;
}
```