

FINAL REPORT

Tom Aprik
Natalie Hannon
Jeremy Konz
Don VanderLaan

Emergency Alert Response System (EARS) for the 21st Century Automobile
ECE 480: Capstone Design Team 9
Michigan State University

December 3, 2008

Contents

1	Introduction and Background	2
2	Exploring the solution space and selecting a specific approach	3
2.1	Matched Filtering Algorithm and MATLAB	3
2.2	First Microcontroller: dsPIC30F	3
2.2.1	Data Acquisition	4
2.2.2	External Oscillator	4
2.2.3	Serial Peripheral Interface	4
2.2.4	Issues with dsPIC30F	6
2.3	Second Microcontroller: dsPIC33F	7
3	Technical description of work performed	7
3.1	Main Software Functionality	7
3.2	ADC Function and DMA Interrupt	8
3.3	Vector Dot Product	8
3.4	Change Notice pin	9
3.5	Thresholding	9
3.6	Dynamic Scaling	9
3.7	Hardware Design	10
4	Test data with proof of functional design	13
5	Final cost, schedule, summary and conclusions	16
5.1	Final Budget	16
5.2	Conclusion	17
A	Technical roles, responsibilities, and work accomplished	18

A.1	Don VanderLaan	18
A.2	Jeremy Konz	19
A.3	Natalie Hannon	19
A.4	Tom Aprik	20
B	Literature and website references	21
C	Detailed technical attachments	24
C.1	Software Flowcharts	24
C.2	PCB Board Schematic	29



Executive Summary

Team 9 has designed and built an emergency alert response system, referred to as EARS, intended for use in General Motors products. EARS listens for an emergency siren external to the automobile using an audio sensor. Successful detection results in notification to the driver of the presence of an emergency vehicle. The notification takes several forms reduction in stereo volume and LED indicator display. The purpose of EARS is twofold to reduce emergency response time by alerting drivers and reduce the number of emergency vehicle crashes with unaware drivers.

Acknowledgement

Team 9 acknowledges Dr. Deller, professor at Michigan State University, for his guidance and support throughout the semester. Team 9 also acknowledges General Motors for generous sponsorship of the project.

1 Introduction and Background

With 37,248 fatal vehicle crashes in the United States in 2007¹¹ increasing vehicle safety is an important initiative for automotive companies. Currently, traffic signal preemption¹⁵ is used to alert drivers of the presence of emergency vehicles in conjunction with the emergency sirens and lights. Traffic signal preemption relies on an emitter on the emergency vehicle to communicate with a detector mounted on a signal arm of a traffic light. Traffic signal preemption is used to insure that emergency vehicles have a green light and thereby the right-of-way through an intersection. However, traffic signal preemption is inherently problematic. It is only available in select geographic areas where the technology is installed at intersections. When it is available, traffic signal preemption relies on line-of-sight transmission from the emergency vehicle. This implies that the vehicle is traveling straight towards the intersection, which may not always be the case. Lastly, traffic signal preemption can be a source of confusion for drivers, thereby negating the potential benefits gained from its use.

Another factor to consider is that not all emergency vehicles may be equipped with a traffic signal preemption emitter. However, it is reasonable to assume that all emergency vehicles have emergency lights and sirens. The U.S. Fire Administration (USFA) has partnered with the Society of Automotive Engineers (SAE) to determine the effectiveness of emergency lights on drivers.³ EARS bolsters the effectiveness of emergency sirens to provide advanced warning to drivers of proximal emergency vehicles.

EARS uses a matched filter to confirm the presence of a siren external to a vehicle. While matched filtering is not new technology, its use in this realm is unprecedented.

2 Exploring the solution space and selecting a specific approach

2.1 Matched Filtering Algorithm and MATLAB

The main concept behind EARS is matched filtering. Matched filtering is essentially convolution, which is a repetition of dot products used to determine the presence of a pre-known signal. A dot product, in mathematics, is an operation which takes a vector of two real numbers and returns a real-valued scalar quantity, defined as $a \cdot b = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$

A disadvantage to matched filtering is time-domain signal processing, requiring computationally intensive convolution for a large sample count. However, this feat was overcome by selecting a powerful microcontroller, as described in Section 2.2. A major advantage of matched filtering is a high detection rate. This was the primary reason matched filtering was selected as the means by which to detect an emergency siren.

2.2 First Microcontroller: dsPIC30F

Initially the microcontroller selected for EARS was the dsPIC30F3014 by Microchip[®]. This 40-pin microcontroller had eight analog to digital (A/D) ports and performed operations at 30 million instructions per second (MIPS). The dsPIC30F contained a digital signal processing (DSP) library, including a vector dot product function. As specified by the manufacturer the dsPIC30F was able to complete a dot product on two vectors of size N in $17+3N$ cycles.²

The dsPIC30F also provided a cost benefit, averaging less than \$5 per unit when purchased in bulk. Traditional DSP chips can range between four to ten times this amount. The controller area network (CAN) communication bus functionality of the dsPIC30F will be used to implement EARS into an automobile. The CAN bus is widely used in automotive applications. It will allow EARS to communicate with other electronic modules within the

automobile.

2.2.1 Data Acquisition

Analog to digital conversion (ADC) is required for interpretation of the external audio noise by the microcontroller. For the dsPIC30F, the A/D Control Register 1 (ADCON1) was configured so that Timer3 controlled the sampling and conversion. An interrupt was triggered after a variable number of samples had been converted to initiate the matched filter function within the main function. However, the dsPIC did not have sufficient internal memory to store the A/D data, leading to the implementation of an external memory chip.

2.2.2 External Oscillator

An external oscillator was implemented for use with the dsPIC30F. Advantages of an external oscillator include reduced cost and lower run temperature for the microcontroller. The dsPIC30F has an oscillator crystal input (OSC1) allowing it to interface with an external oscillator. The dsPIC30F external clock output is connected to the OSC1 input, which is routed to the phase locked loop (PLL) multiplier. The PLL will increase the frequency of the external clock. After the PLL multiplier, the signal is sent to the clock switching and control block.

2.2.3 Serial Peripheral Interface

A one-megabyte electrically erasable programmable read-only memory (EEPROM) was selected to meet the memory requirements of EARS. This model by Microchip[®], 25LC1024, communicates via the serial peripheral interface (SPI) bus. Implementation of SPI was simplified by using the dedicated hardware buffers in the dsPIC30F. An output bit rate of 20 megabits/second was deemed sufficient to transfer data. Functioning via four wires, the SPI protocol was maximized for speed. The external wires included a 20MHz serial clock,

serial data input, serial data output, and chip select to enable the EEPROM. Implementing serial input and serial output allowed for simultaneous read/write operations in the SPI module. The external memory was intended to store information as the internal memory of the dsPIC30F filled.

Considerable time was spent to implement the SPI in order to determine the viability of the dsPIC30F and EEPROM setup. A dedicated communication module on the dsPIC30F reduced the amount of system resources required to run SPI. This module automatically sent a 16-bit word when written to the transmit buffer. This also activated the serial communication clock and synchronized it to the serial output. At the same time that data was transmitted from the dsPIC30F data was read in via the serial in line. This input data was shifted into a receive buffer bit-by-bit on the active clock edge.

The first version of the SPI did not feature a transaction request queue and resulted in substandard performance. Figure 1 shows the original program flow, which required the CPU to sit idle while waiting for an I/O transmission to complete.

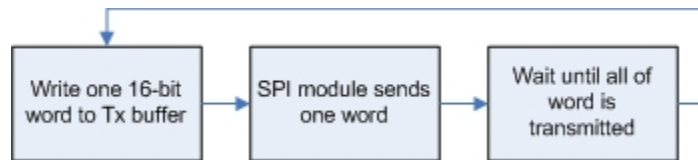


Figure 1: Low performance SPI transaction flow

The second version of the SPI included a circular queue containing read/write requests for the EEPROM. Each request is pushed on the queue and performed as soon as the SPI module is ready.

The circular transaction request queue improved performance. The queue featured minimal wait time and exploited benefits of interrupt driven programming. An interrupt was triggered manually and also automatically when one complete word was sent. This topology, shown in Figure 2, activated the SPI module by generating a request to send and receive information. The interrupt continued with each sequential item until the queue was empty.

At this point, all requested transactions had been performed and the interrupt will only trigger manually. The key to the success of this topology was that the SPI module only consumed resources when interrupt-triggered.

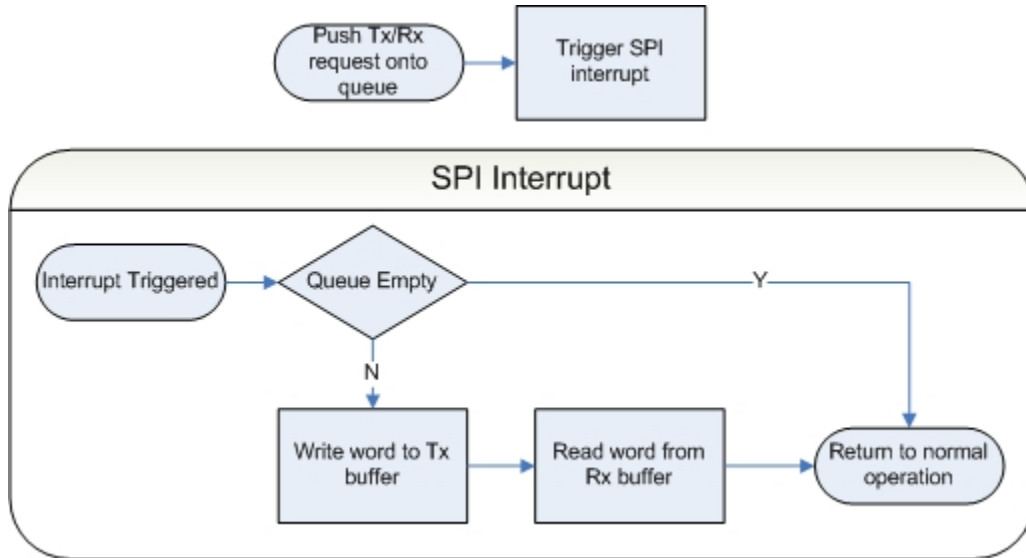


Figure 2: High level view of high performance, interrupt driven SPI transaction flow

2.2.4 Issues with dsPIC30F

The functionality of the SPI communication with the EEPROM was first verified at a serial clock rate of 100kHz. Read/write operation to and from the EEPROM was successful. However, results were not replicated when the clock rate was increased to 10MHz, representing 50% of maximum speed. The problem was contained within the chip select pin use. The write enable latch command required that the chip select pin be enabled (set low) for precisely 8-bits of transfer. The EARS system sends 16-bits at one time, which meant that the chip select must be set low half-way through the transfer. Figure 3(a) specifies the signals the EEPROM must receive in order to set the write enable latch. After the serial clock was increased to 10MHz, the chip select signal was performing at random as shown in figure 3(b).

The dsPIC30F needed to run at 30MIPS and the SPI module needed to run at 20MHz

to suit the requirements for EARS. This meant that over every three instruction cycles, two bits were sent and received over the SPI bus for the EEPROM. The timer counted eight serial clock pulses before interrupting to set the write enable latch. However, after eight pulses the dsPIC30F was left with $\frac{3}{2}$ of an instruction cycle to execute the interrupt. The overhead required to call an interrupt was too large for it to execute. At this point it was determined that the dsPIC30F and EEPROM combination would not suffice for EARS.

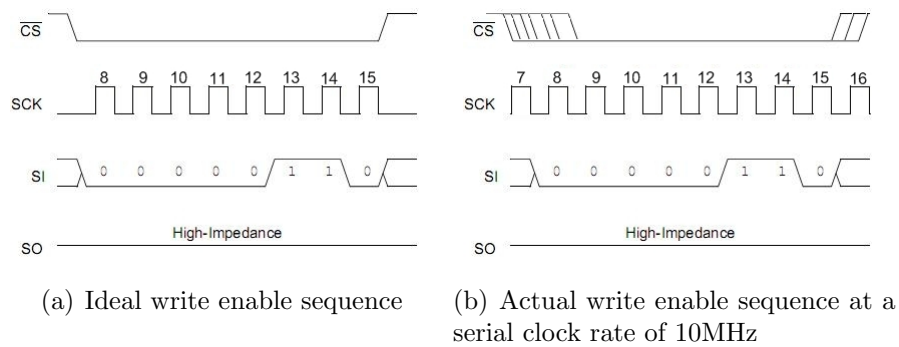


Figure 3: EEPROM Write Enable Sequences

2.3 Second Microcontroller: dsPIC33F

With the option of using external memory eliminated, the 28-pin dsPIC33FJ128GP802 by Microchip[®] was selected based on its small size, 40MIPS processing power and 16k byte internal memory. The dsPIC33F is the microcontroller that is used in the final design, as outlined in Section 3.1.

3 Technical description of work performed

3.1 Main Software Functionality

. The software implemented on the dsPIC33F is interrupt driven. The “main” section of software loops continuously, waiting to perform the vector dot product (VDP) function.

The VDP function executes for each new conversion result acquired from the ADC function. Detection of a siren will occur quickly because the program executes in as close to real time as possible. There is not data backup because incoming data is processed immediately and completely before additional data arrives.

3.2 ADC Function and DMA Interrupt

To communicate with the main function, the dsPIC33F utilizes interrupt-driven direct memory access (DMA) for data transfer. DMA moves data to and from the RAM without interrupting the central processing unit (CPU). DMA is most useful when more than one conversion result needs to be buffered before triggering an interrupt and can allocate up to 128 words of buffer for each analog input. For EARS, the DMA interrupts after every conversion result. This happens so that EARS will detect the siren almost instantly. DMA is set in continuous ping pong mode so that it alternates writing data between two registers. This happens so that data is not accidentally overwritten.

3.3 Vector Dot Product

Vector dot product from the DSP library takes two fractionals in memory locations and returns the scalar fractional result. The output of the vector dot product is a high fractional when a match is found. However, if two high fractionals are summed, overflow occurs. Section 3.6 addresses this issue.

Another design issue with the VDP function was performing each vector dot product as the signal was sampled. As each sample was taken, a shift was needed to balance the starting point of the vector dot product. To compensate for this, the VDP function computes two vector dot products corresponding to where the shift occurs and then sums them.

3.4 Change Notice pin

A push button was implemented to store a siren template. This button was connected to a change notice (CN) pin on the dsPIC33F. CN pins are designed to generate an interrupt whenever a digital voltage change is detected. When the button is pressed an interrupt occurs that sets the appropriate variables to store the incoming A/D data in the siren template location. Also, a LED is lit while the siren template is stored. At completion of the interrupt the scaling factor is calculated, as explained in 3.6.

3.5 Thresholding

After the VDP function is complete the result is compared to a threshold. A match is detected if the result is greater than the threshold value. Each match increments the match counter, while each non-match decrements the match counter. Once the match counter has reached a predefined limit EARS renders a siren detected.

3.6 Dynamic Scaling

Since EARS repeatedly sums and multiplies data in the VDP function, it is imperative that overflow does not occur. Overflow causes inaccurate results, leading to faulty siren detection. A dynamic scaling approach is applied to the input signal in order to eliminate any possibility of overflow. The scaling factor is calculated based on the maximum values in the siren template. Each A/D conversion result is divided by the scaling factor. Resolution is lost when scaling is applied, although the high resolution of the 12-bit A/D module makes this loss negligible. Prior to implementing dynamic scaling, overflow occurred on 100% of vector dot products performed.

3.7 Hardware Design

The electret microphone contains a simple transistor amplifier inside its packaging. This internal amplifier is likely a single JFET which is the reason the microphone requires a bias current to operate. To suffice this requirement the amplifier sets the DC offset voltage across the electrets terminals to be approximately 6.5V. A simple way to bias the electret is to use a pullup resistor to the power supply. Any oscillation on the power supply will directly affect the electrets output voltage. Since EARS is powered from a car battery and given the high Thevenin impedance of the battery, there could be considerable amount of noise. Therefore, circuitry was designed to indirectly power the electret microphone from the power supply. An 8.5V Zener diode with a pullup resistor to the battery was used to provide a steady DC voltage for biasing the electret microphone. The pullup resistor needs to be small enough such to allow enough current under maximum load conditions yet large enough so the Zener diode does not draw too much current at no load. The diagram for the electret microphone biasing network is shown in Figure 4. EARS involves detecting a siren in the

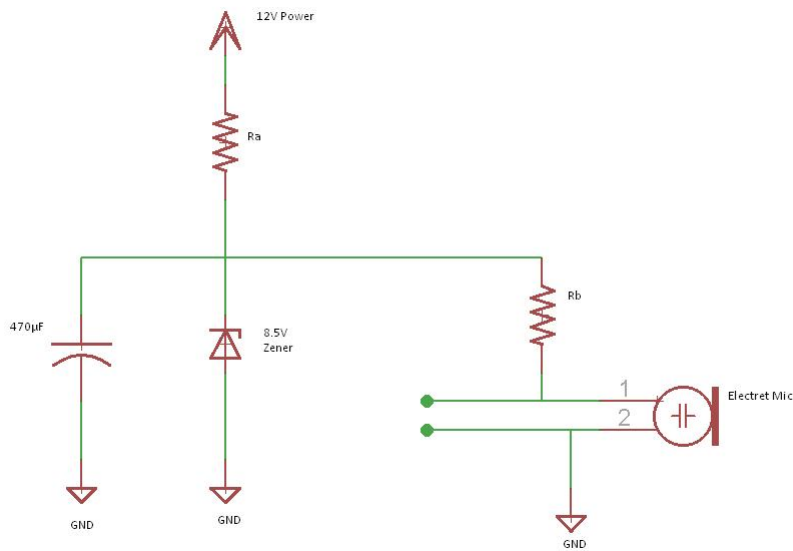


Figure 4: Electret microphone biasing network

acoustic environment external to the vehicle with the electret microphone. The amplifier circuitry will be inside the vehicle a distance away from the electret microphone. The long microphone leads will be prone to electrical noise pickup, especially if the leads are a twisted pair without shielding. Signals from all two-wire microphones are differential, justifying the use of a differential amplifier. Most of the electrical noise picked up is common-mode, thereby making it easy to suppress. The gain of common-mode signals is 20dB below that of differential-mode signals. This is a result of a modification of the amplifier to provide a DC offset at the output.

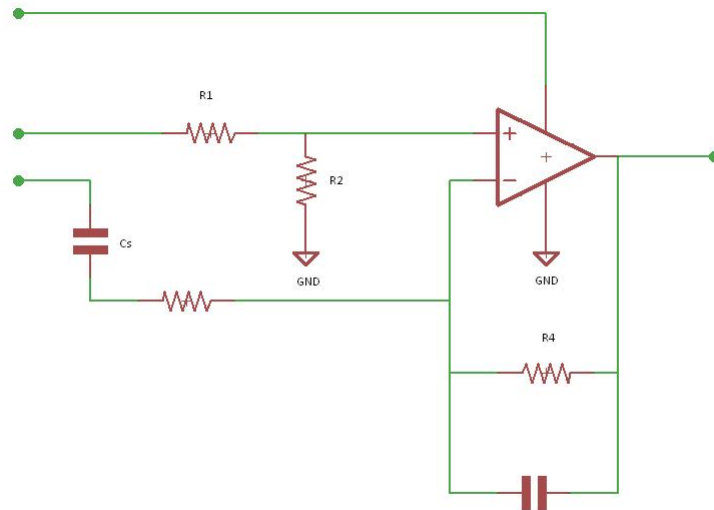


Figure 5: Differential Amplifier capable of being driven by single supply

The next stage is a filtering stage. It is necessary to optimally condition the signal for input into the microcontroller. The data acquisition is both discrete-time and digital. Aliasing is one reason to block high frequency content. Time sampling a high frequency signal below the Nyquist rate and a low frequency signal at the same rate could yield the same vector of data. Therefore, acoustic noise sampled below its Nyquist rate after matched filtering could produce a false positive. A low pass filter prevents this case.

A second reason for filtering is to minimize digitization errors of the A/D converter.

The siren will contain spectral content solely within the range of 0.5 to 2 kHz. While superposition of the spectral content within 0.5 to 2 kHz may not exceed the PIC's input voltage range, superposition of the entire audio band's spectral content will result in a larger voltage range. Removing the out-of-band spectral content will allow us to increase the gain of the amplifier to minimize digitization errors. Therefore, to reduce the potential number of false-positives, the passband of the amplifier was set to 0.5 to 2 kHz. Both sides of the passband are second-order with a Butterworth filter blocking high frequency content. The filter is shown in Figure 6.

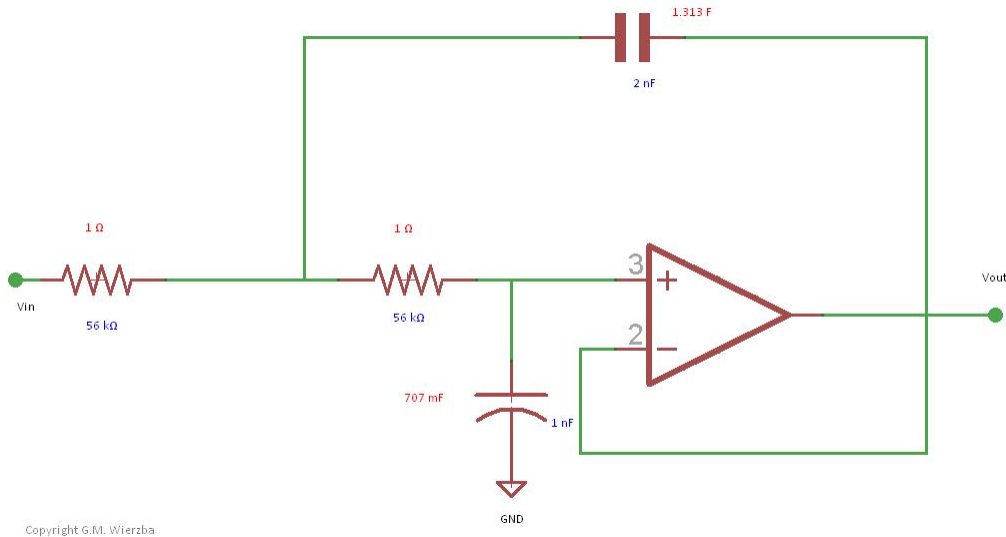


Figure 6: Butterworth filter with pre- and post- scaling values shown¹⁶

Testing the electret microphone indicated the overall gain of the amplifier should be around 350, or 50dB. The first stage provided a gain of 10; therefore, the final stage must have a gain of 35. The final stage also must set the final DC offset to 2.05V for the microcontroller. Design of the final stage originally started with a differential amplifier. The final DC offset was set using a voltage divider with the stable Zener reference from the electrets biasing network. A series capacitor was used to adjust the DC offset of the signal from the electret microphone. The resistors, including a pot, were selected to set the gain to 35.

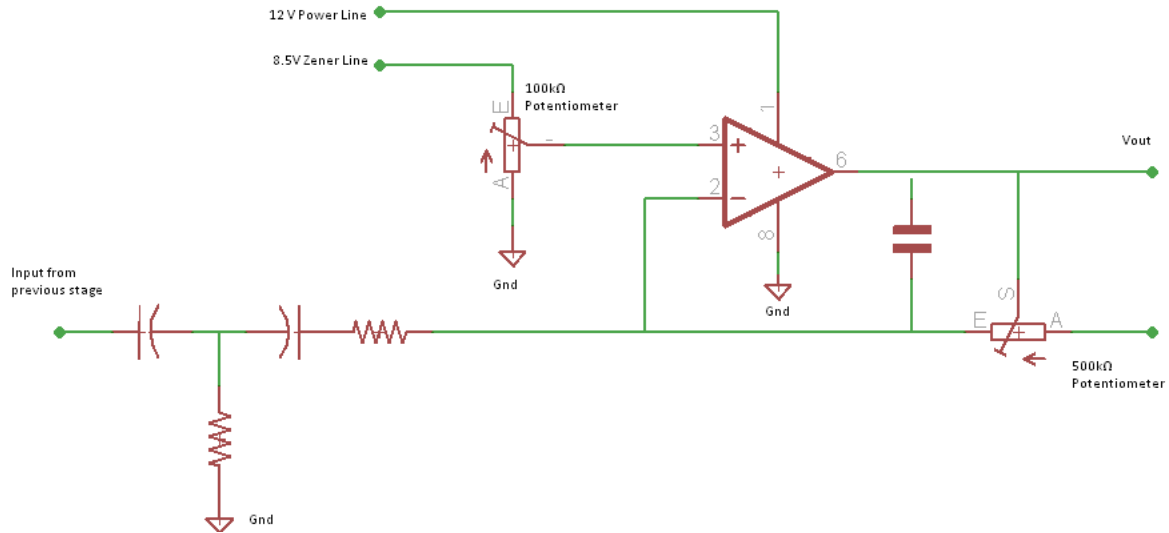


Figure 7: Third Stage

To block the low frequency content, capacitors were added to the existing stages.

A voltage regulator was used to provide a constant 3.3V power line for the microcontroller¹⁰.

The entire amplifier circuit is shown in Appendix C.2.

4 Test data with proof of functional design

EARS was tested to determine how well it was able to detect a siren. For any siren template within the frequency range of 0.5 to 2 kHz, EARS has the capability to detect it. This was tested by observing the amplifier output and LED output on the oscilloscope for multiple siren templates. The LED will turn on whenever the dsPIC33F meets conditions to detect a siren. Observing the LED output and amplifier output simultaneously allows for a visual reference. Figure 8 displays test data from the lab. In this case, EARS detects a siren, turns on the LED and then turns it off after the siren is off. This is not to say that EARS is fully functional. Currently EARS is too sensitive to impulse signals and determines false positives on some sharp loud noises. However, if the sensitivity is decreased false-negatives

are also likely. To operate ideally, more filtering will be necessary to achieve near perfect functionality. The current hardware implemented does not have this capability.

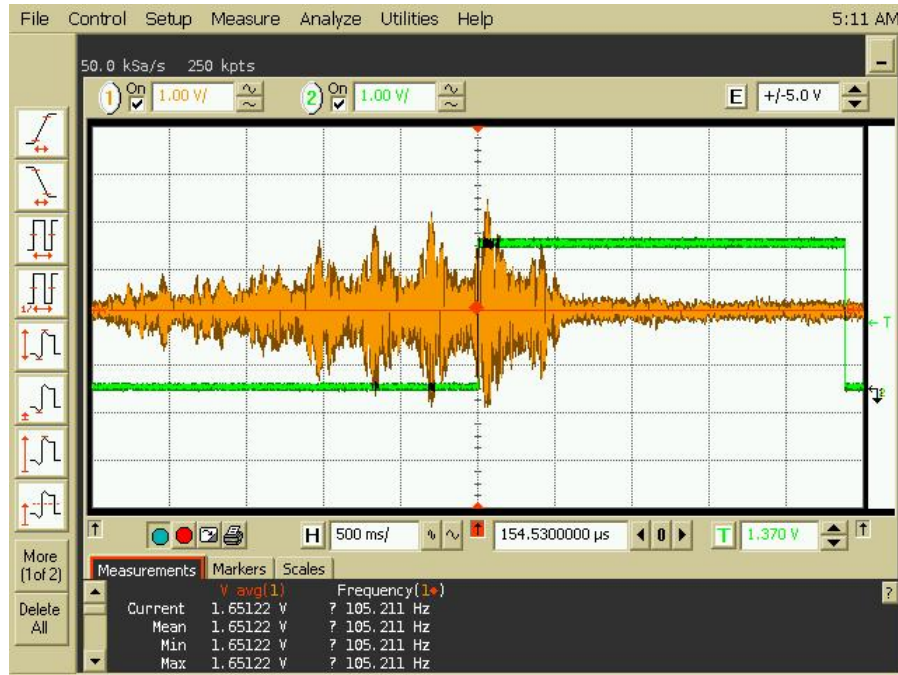


Figure 8: Complete siren detection and removal cycle.

Once a siren template is stored, EARS normalizes future A/D reads and this allows the volume of siren to have less effect on the algorithm results. Figure 9 displays how small of a magnitude the siren can be to detect a match. While figure 10 shows the same siren input, but with much greater magnitude. Both of the examples in figures 9 and 10 are matching with the template stored.

EARS does have ability to filter out noise that is not relevant. Figure 11 shows a 1200Hz input signal into the microphone, and EARS does not cross the threshold of detection.

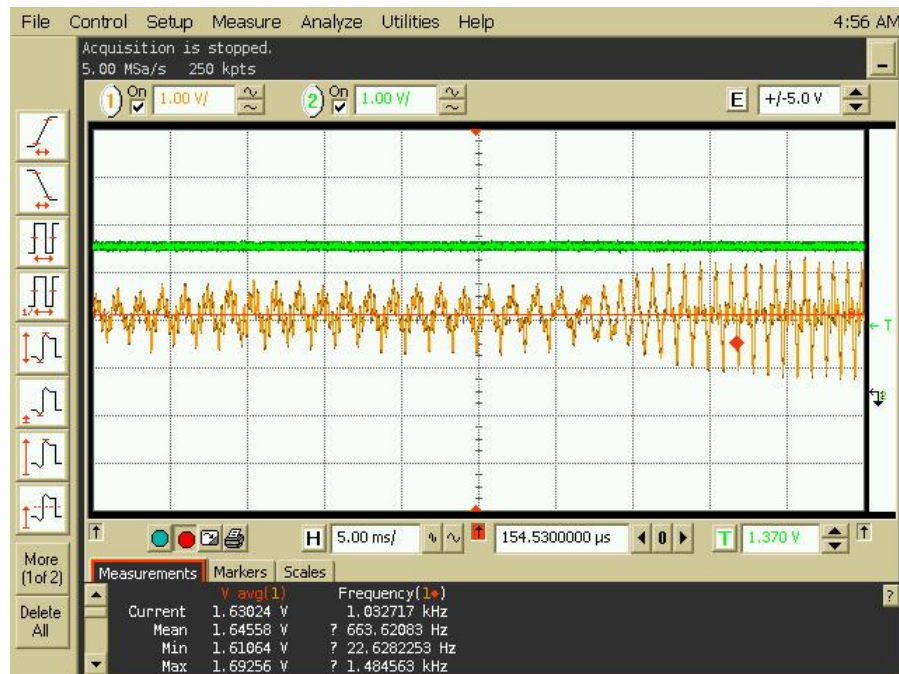


Figure 9: Minimum siren input needed to detect.

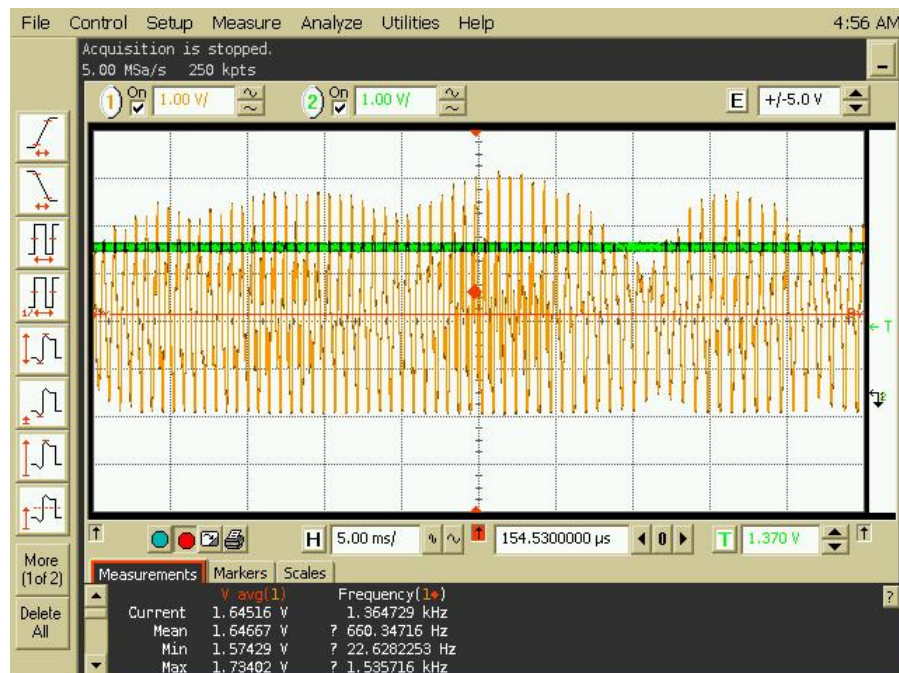


Figure 10: Maximum input magnitude still leads to detection

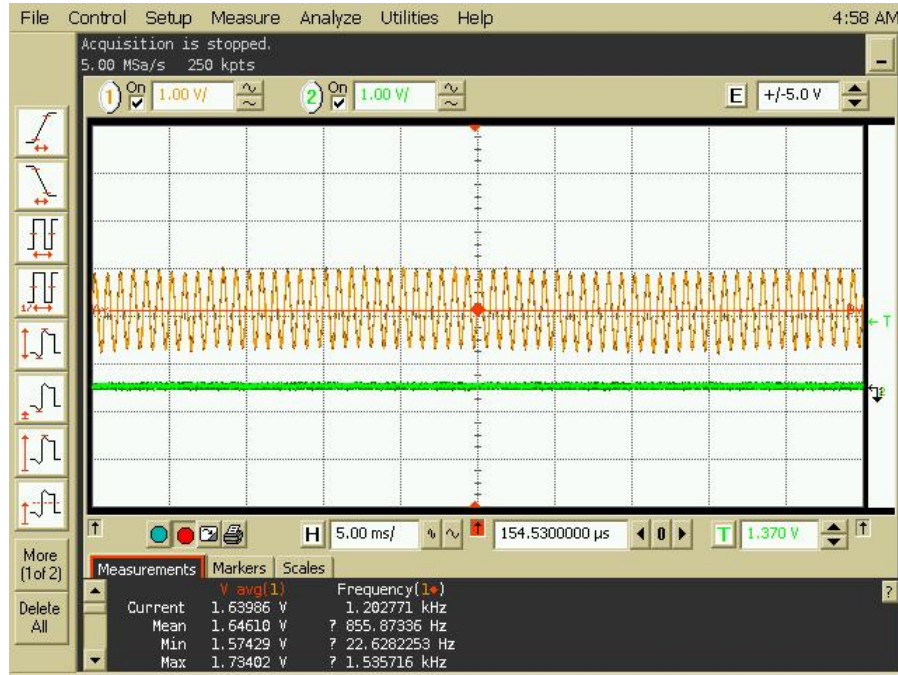


Figure 11: 1200Hz input signal does not activate siren.

5 Final cost, schedule, summary and conclusions

5.1 Final Budget

Category	Quantity	Description / Part Number	Cost
Main Processing	1	Microchip dsPIC33FJ128GP802 Microcontroller	\$5.88
Acoustics	1	Electret Microphone / MD9745APZ-F	\$1.01
Acoustics	1	Siren Template - iTunes App	\$1.06
Acoustics	2	Operational Amplifier / LF411CN	\$2.16
Power	1	3.3V Voltage regulator / LM2576T-3.3	\$3.05
Housing	1	Display Box / 1591ETCL	\$12.30
Various	-	Other low power components (R, C, L)	≤ \$5
Total			≤ \$29.40

5.2 Conclusion

The work on siren detection produced an algorithm and prototype that provides a foundation for future work on the project. The team found that while matched filtering is easily implementable and provides reasonable detection rates, the computational and memory resources are quite demanding.

The end result did not function optimally due to the large number of false positives. These false positives could be the result of higher spectral content picked up by the amplifier and aliased down in to the band that the siren template occupies. However this case is unlikely, because the microphone amplifier was band limited to prevent this. Another possibility is that the ambient acoustic noise picked up when recording the template is matching with the ambient acoustic noise present in the testing environment. This would be a more difficult case to reject, as it would require recording the siren template in a low noise recording studio. Simple thresholding on the matched filter output is used to determine whether a siren is present. If the Matlab algorithm was used, then an autocorrelation of the output could be performed and the distance between peaks would be compared to the siren templates fundamentals period. Only when the spacing was similar would a siren be detected. This works well in Matlab, however is too computationally intensive to implement in the PIC microcontroller used. This leads to the opportunity for future work on the project: alternate hardware could be used to implement the more advanced algorithm to improve detection rates. The project was accomplished well under budget, further justifying the possibility of future work.

A Technical roles, responsibilities, and work accomplished

A.1 Don VanderLaan

Dons technical work included theoretical justification for matched filtering and design of the analog circuitry required to interface the microphone with the microcontroller digitizer.

Theoretical simulation of the matched filtering technique was done in Matlab. A siren audio file was used as the template for matching. Then, a superposition of the template and various noise profiles was used to see if simple matched filtering work. Various distortions on the input were also tried, to further determine the algorithms robustness. Tested distortions include Doppler shift and various fade in and out and their combinations.

Simple matched filtering worked sufficiently for the non-distorted case, but the following improvements were made to the algorithm: The fundamental frequency of the template was determined by the inverse of the distance between peaks in an autocorrelation of the template. An autocorrelation is computed from the output of the simple matched filtering, and if the peaks in this vector are spaced the same distance as the period of the fundamental frequency of the template, the siren is deemed present. This resulted in superior detection for all inputs.

Don also designed the microphone amplifier. The amplifier was used to interface an electret microphone to the microcontrollers analog to digital converter. The overall gain of the amplifier needed to be 350, therefore two gain stages were used. A low-pass Butterworth filter was designed using G. M. Wierzbas procedure of magnitude and frequency scaling a normalized Butterworth filter.

The electrets bias network included a zener held below the supply to provide a stable DC voltage for biasing the electret microphone. This will reduce the susceptibility of the design

to oscillations on the power supply.

A.2 Jeremy Konz

As one of two computer engineering students on the team, the technical role that Jeremy held was highly based on software design. He was tasked with creation of the SPI communication bus module for the dsPIC30F. Jeremy created a SPI function library to communicate with the EEPROM. After writing the original SPI implementation, he wrote an alternative design which featured a circular transaction request queue. The second version of the module decreased use of system resources drastically. However, SPI module testing ultimately led to the consideration of a new microcontroller for EARS.

For the dsPIC33F Jeremy wrote the main functionality and program flow with the help of Natalie and Tom. This included memory management, change notice interrupts for push buttons, and PLL control to achieve 40MIPS operation. Jeremy is the main author of `main.c` and `common.h`, included in Appendix C. He was also involved with implementing dynamic scaling which eliminated overflow.

A.3 Natalie Hannon

Natalie was responsible for data acquisition. This required knowledge of analog to digital conversion (ADC) for interpretation of external audio noise by the dsPIC. She was the main author of `adc.c`, included in Appendix C. She learned how to configure individual registers within the A/D module to set reference voltages, configure data types and determine function control. The ADC function for the dsPIC30F was interrupt-driven, requiring a separate ISR to communicate with the main code. The switch from the dsPIC30F to the dsPIC33F required her to re-write `adc.c` to use direct memory access (DMA). She learned how to configure the DMA buffer to interrupt after every conversion result to achieve real time

operation.

As part of the software team, Natalie assisted in debugging code. She looked at data and interpreted the validity of the results and tested the ADC function for accuracy. She worked with Jeremy and Tom to design the program flow, including the structure of the main function, ADC and VDP functions.

A.4 Tom Aprik

Toms contribution to the project involved developing code for the vector dot product function. He assisted with the overall programming for the skeleton code of the microcontroller. He also assisted in debugging all portions of the code, include the SPI, analog to digital conversion, external oscillator, and DMA interrupt. The SPI portion was coded by Jeremy and then tested and debugged by all group members, including Tom. The analog to digital conversion was coded by Natalie and then debugged by Natalie, Jeremy, and Tom. The integration of the external oscillator was done by Tom and tested by Jeremy. After weeks of testing and much deliberation, it was decided by the team members that a different PIC was needed to better suit the needs of the project. With the new microcontroller the SPI and external oscillator were omitted from the project. The interrupt system had to be rewritten for the new microcontroller as well. Tom assisted with rewriting all the parts to work correctly with the new microcontroller. Tom was also in charge of designing the complete layout of the circuitry in Eagle and creating the PCB for the final display.



(a) Don VanderLaan (b) Jeremy Konz (c) Natalie Hannon (d) Tom Aprik

Figure 12: Team Members

B Literature and website references

References

- [1] “Crash Avoidance Metrics Partnership. December 2002. National Highway Traffic Safety Administration. 16 November 2008. <http://www.nrd.nhtsa.dot.gov/pdf/nrd-12/CAMPS.pdf>
- [2] “dsPIC30F Digital Signal Controllers” Microchip. 2008. 04 Nov 2008. http://ww1.microchip.com/downloads/en/DeviceDoc/dspbrochure_70095G.pdf
- [3] “Emergency Vehicle Safety Initiative. August 2004. FEMA. 16 November 2008. <http://www.usfa.dhs.gov/downloads/pdf/publications/fa-272.pdf>
- [4] Grimm, Don. “DSRC: An International Perspective. 2 May 2008. Online PowerPoint. General Motors. 16 November 2008. http://www.acma.gov.au/webwr/_assets/main/lib310662/23futurebandwidthrequirementsforitspart2.pdf
- [5] Holstege, Sean. “GM shows off latest smart-car technology in S.F. Oakland Tribune. 7 November 2005. SIRS BNET. 16 November 2008. http://findarticles.com/p/articles/mi_qn4176/is_20051107/ai_n15767653

- [6] “Intelligent Transportation Systems: Research and Innovative Technology Administration. 2008. U.S. Department of Transportation. 16 November 2008. <http://www.its.dot.gov/index.htm>
- [7] “Intelligent Vehicle Initiative. 2008. U.S. Department of Transportation. 16 November 2008. <http://www.its.dot.gov/ivi/ivihf/index.html>
- [8] “Microchip dsPIC30F3014/4013 Data Sheet” Microchip. 2008. 04 Nov 2008. <http://ww1.microchip.com/downloads/en/DeviceDoc/70138F.pdf>
- [9] “Microchip dsPIC33FJ32GP302/304, dsPIC33FJ64GPX02/X04, and dsPIC33FJ128GPX02/X04 Data Sheet” Microchip. 2008. 04 Nov 2008. <http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en532298>
- [10] “National Semiconductor LM2576 Data Sheet” National Semiconductor. 2008. 01 Nov 2008. <http://www.national.com/ds/LM/LM2576.pdf>
- [11] “National Statistics. Fatality Analysis Reporting System (FARS) Encyclopedia. 16 November 2008. <http://www-fars.nhtsa.dot.gov/Main/index.aspx>
- [12] Murray, Charles J. “GM Shows off Safety Technology. 2 June 2008. Design News. 16 November 2008. http://www.designnews.com/article/46449-GM_Shows_off_Safety_Technology.php
- [13] Perel, Michael. “IVI Human Factors Research. 12 January 2004. U.S. Department of Transportation. Online PowerPoint. 16 November 2008. http://www.its.dot.gov/ivi/ivihf/docs/IVI_HF.ppt
- [14] Resendes, Raymond. “Intelligent Vehicle Initiative. U.S. Department of Transportation. Online PowerPoint. 16 November 2008. http://ivsource.net/public/ppt/021001_USIVI-Resendes.ppt

- [15] “Traffic Signal Preemption for Emergency Vehicles: A Cross-Cutting Study. January 2006. U.S. Department of Transportation. 16 November 2008. http://www.itsdocs.fhwa.dot.gov/JPODOCS/REPTS_TE/14097_files/14097.pdf
- [16] Wierzba, G.M. “ECE 831 e-Notes.” Michigan State University. 2008.

C Detailed technical attachments

C.1 Software Flowcharts

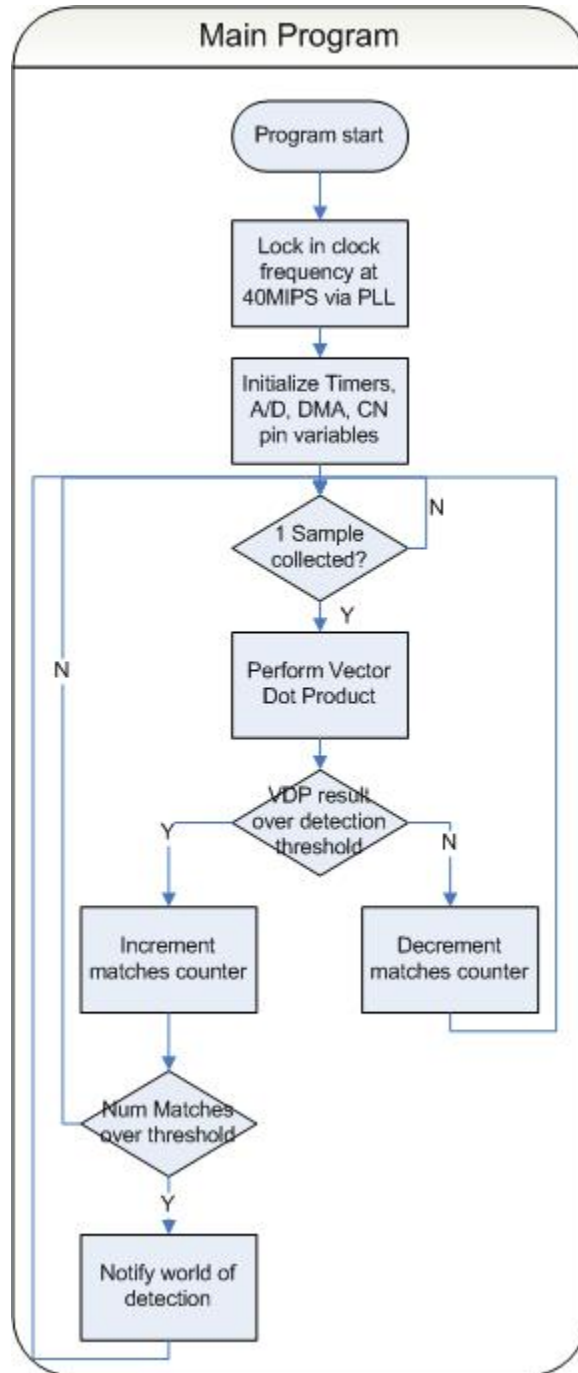


Figure 13: Main Function Flowchart

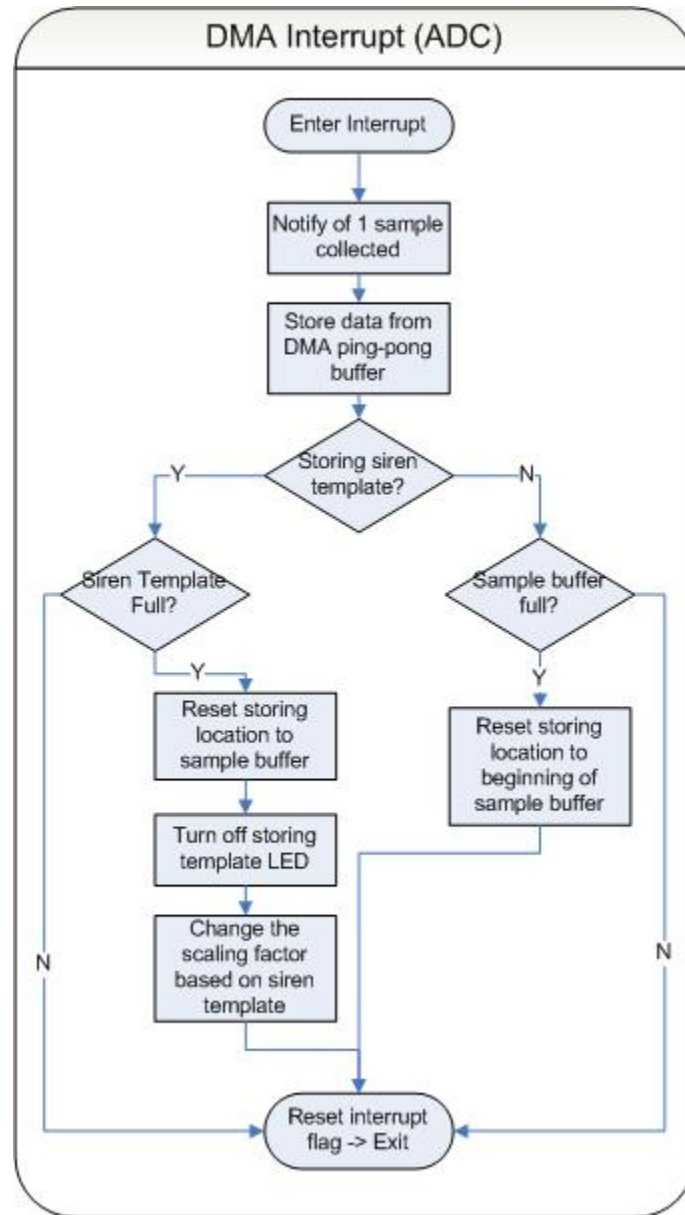


Figure 14: DMA Interrupt (from ADC) Flowchart

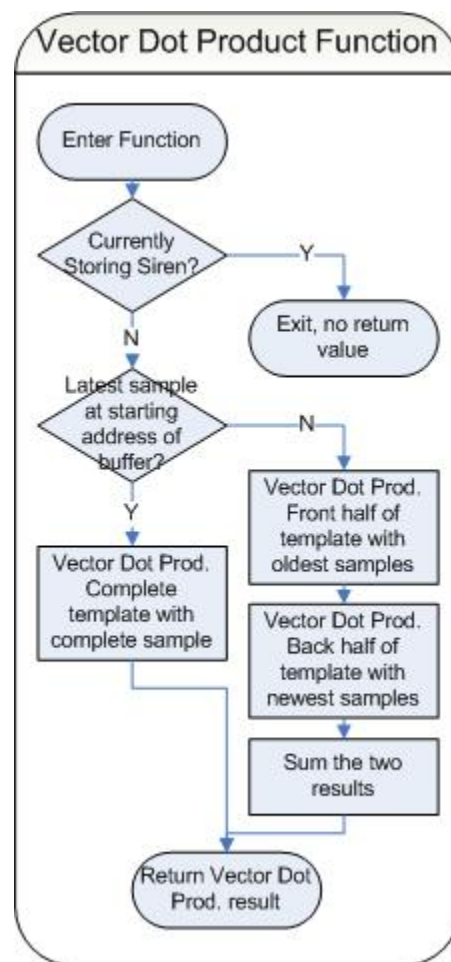


Figure 15: Vector Dot Product Function Flowchart

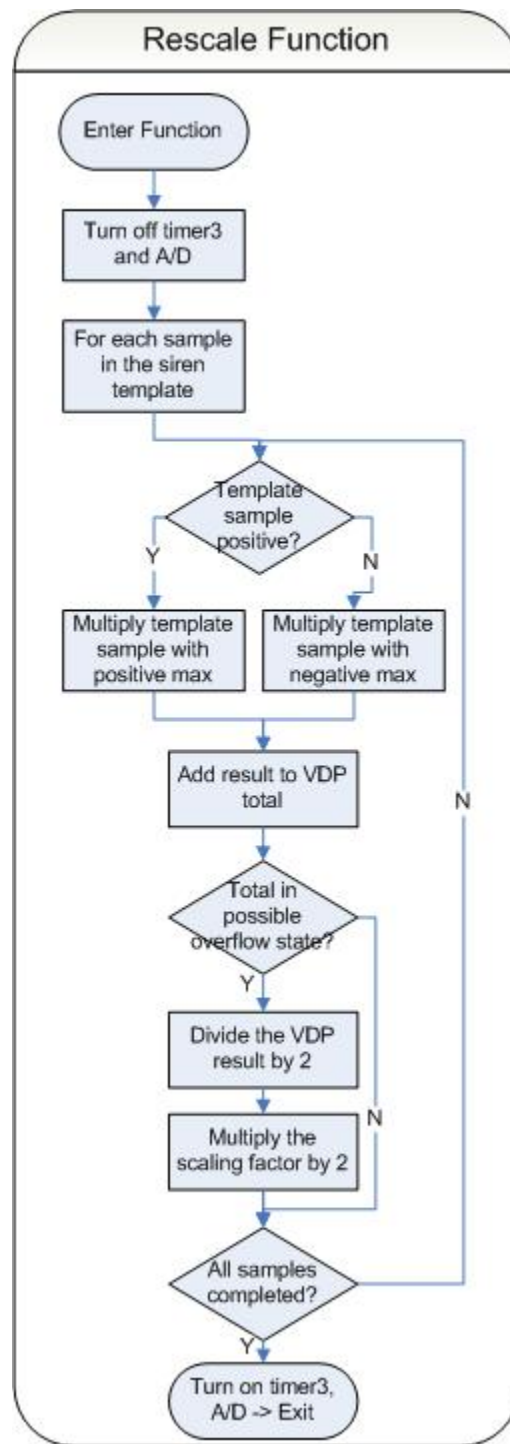


Figure 16: Rescale Function Flowchart

C.2 PCB Board Schematic

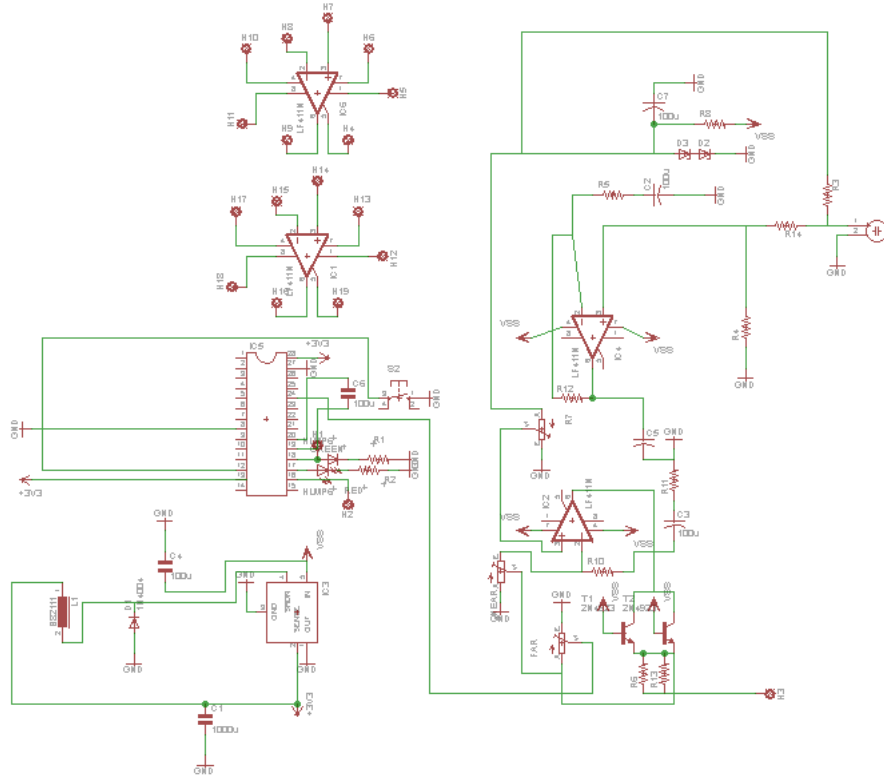


Figure 17: PCB Board Schematic

Listing 1: Project Code for dsPIC33F

```
1  /* File: main
2  *
3  * Written by: ECE 480 Design Team 9, Fall 08
4  *
5  * Contains: Main function, also timer initialization, change notice interrupt,
6  *           rescaling function, Vector Dot Product
7  */
8  #include "common.h"
9  #include "adc.c"
10
11 // Select Internal FRC at POR
12 _FOSCSEL(FNOSC_FRC);
13 // Enable Clock Switching and Configure
14 _FOSC(FCKSMCSECMD & OSCIOFNC_OFF);
15 _FWDT(FWDTEN_OFF); // Watchdog Timer Enabled/disabled
16
17 int main()
18 {
19     TRISB = 0x8000; //set up the B port as all outputs
20     CORCONbits.PSV = 1; //enable Program Space Visibility
21
22
23     /***** Functions to set up hardware *****/
24     CNinit(); //initialize the change notice pins
25     TMRinit(); //initialize the timers
26     CLOCKinit(); //set the clock frequency to 40MIPS
27
28     ADC(); //initialize the A/D
29     DMA(); //initialize the DMA
30
31     /***** Variable initialization *****/
32     ADCstore = &sample[0]; //initialize the location to store the ADC
33     info as //the outside sample
34     ind = 0;
35     RunFilter = 0;
36     matches = 0;
37     MATCHLED = 0;
38
39
40     while (1) // code execution - interrupt driven */
41     {
42         if (RunFilter) // flag set upon receiving new sample */
43         {
44             result = VDP(); //perform vector dot product matched
45             filtering
46             RunFilter = 0; //Reset flag to get more ADC data
47
48             //if the result is negative, take the 2s complement so the
```

```

48         magnitude can be evaluated
49         if (result >= 0b1000000000000000)
50         {
51             result = (result ^ 0xFFFF) + 1;
52         }
53         if ((result >= 0b0001000000000000) && (matches < 300))
54             //if result is above threshold
55         {
56             matches += 10;                                //increment
57             the matches counter
58         }
59         else if ((result < 0b1000000000000000) && (matches > 0))
60         {
61             matches--;                                    //
62             decrement the matches counter
63         }
64         if (matches >= MATCHTHRESHOLD) //display a match to the world
65         {
66             MATCHLED = 1;                                //turn on LED
67             //timer is set to turn off the match led after 1
68                 second...
69             //assuming it is not reset again here on the next
70                 match
71             TMR5 = 0x00;                                    // Clear 32-
72                 bit Timer (msw)
73             TMR4 = 0x00;                                    // Clear 32-
74                 bit Timer (lsw)
75             T4CONbits.TON = 1;                            // Start Timer
76         }
77     }
78 }
79 return 0;
80 }
81
82 void CLOCKinit(void)
83 {
84     /*      Name: CLOCKinit (clock initialization)
85     *      Function: To set up the clock to achieve maximum speed
86     *      Explanation: This step is necessary in order to get 40MIPS operation.
87     *      The onboard RC oscillator is sent through the PLL to get this
88     speed.
89     *
90     *      Taken from http://ww1.microchip.com/downloads/en/DeviceDoc/70216B.pdf
91     *      Page 22
92     */
93     // Configure PLL prescaler , PLL postscaler , PLL divisor

```

```
89 PLLFBD = 41; // M = 43
90 CLKDIVbits.PLLPOST=0; // N1 = 2
91 CLKDIVbits.PLLPRE=0; // N2 = 2
92
93 // Initiate Clock Switch to Internal FRC with PLL (NOSC = 0b001)
94 __builtin_write_OSCCONH(0x01);
95 __builtin_write_OSCCONL(0x01);
96
97 // Wait for Clock switch to occur
98 while (OSCCONbits.COSC != 0b001);
99
100 // Wait for PLL to lock
101 while(OSCCONbits.LOCK!=1) {};
102 }
103
104 fractional res1, res2; //results fractionals from the two halves of the VDP
105
106 fractional VDP(void)
107 {
108 /*      Name: VDP (vector dot product)
109 *      Function: To perform a single vector dot product
110 *      Returns: The fractional result of the VDP matched filtering, ranging
111 *      from -1 to 1
112 *      Explanation: As long as the siren template is not currently being
113 *      stored
114 *      we will use the sample and the siren template arrays and
115 *      calculate the
116 *      VDP. This result will range between -1 and 1 in a signed 16-
117 *      bit fractional
118 *      because of the scaling function. The VectorDotProduct was
119 *      included in the
120 *      DSP library by microchip, it is fast.
121 */
122 if (!storing_siren)
123 {
124     if (ADCstore == &sample[0]) //if the latest sample is the
125         beginning of the sample buffer
126     {
127         //only
128         one VDP is needed
129         return VectorDotProduct(SAMPLE_SIZE, &sample[0], &
130             siren_template[0]);
131     }
132     else
133     {
134         res1 = VectorDotProduct(SAMPLE_SIZE-ind, &sample[ind],
135             &siren_template[0]);
136         res2 = VectorDotProduct(ind, &sample[0], &siren_template[SAMPLE_SIZE - ind]);
137         ;
138         return ( res1 + res2 );
139     }
140 }
```

```
129     }
130 }
131 return NULL;
132 }
133
134 void CNinit(void)
135 {
136     /*      Name: CNinit (change notice initialization)
137     *      Function: To set up the change notice pin
138     *      Explanation: The change notice pin is used to detect a button press
139     */
140
141     CNEN1bits.CN0IE = 1;           //enable the interrupt on the CN11 pin
142     CNPU1bits.CN0PUE = 1;         //enable the internal pull-up resistor for
        button
143     IEC1bits.CNIE = 1;             //enable the CN interrupts
144     IFS1bits.CNIF = 0;            //clear CN interrupt flag
145 }
146
147 void TMRinit(void)
148 {
149     /*      Name: TMRinit (timer initialization)
150     *      Function: To set up the timer that keeps the match LED on for
151     *                  an extended period of time
152     *      Explanation: This timer controls the matched led. Since we want the
153     *                  LED to be on for 5 seconds, the timer needs to be 32-bits
154     */
155
156     T5CONbits.TON = 0;             // Stop any 16-bit Timer3 operation
157     T4CONbits.TON = 0;             // Stop any 16/32-bit Timer3 operation
158     T4CONbits.T32 = 1;             // Enable 32-bit Timer mode
159     T4CONbits.TCS = 0;             // Select internal instruction cycle clock
160     T4CONbits.TGATE = 0;           // Disable Gated Timer mode
161     T4CONbits.TCKPS = 0b00;        // Select 1:1 Prescaler
162     TMR5 = 0x00;                  // Clear 32-bit Timer (msw)
163     TMR4 = 0x00;                  // Clear 32-bit Timer (lsw)
164     PR5 = 0x04c4;                 // Load 32-bit period value (msw)
165     PR4 = 0xb400;                 // Load 32-bit period value (lsw)
166     IPC7bits.T5IP = 0x01;         // Set Timer3 Interrupt Priority Level
167     IFS1bits.T5IF = 0;            // Clear Timer3 Interrupt Flag
168     IEC1bits.T5IE = 1;            // Enable Timer3 interrupt
169     T4CONbits.TON = 0;            // Start 32-bit Timer
170 }
171
172 void __attribute__((__interrupt__, __auto_psv__)) _CNInterrupt(void)
173 {
174     /*      Name: _CNInterrupt
175     *      Function: This interrupt executes upon any digital change on a CN pin
176     *      Explanation: This interrupt will execute when any CN button is pressed
177     *                  . The interrupt must determine which pin,
178     *                  and execute the according command, either store a siren or
179     *                  acknowledge one
180     */
181 }
```

```
177
178 //this case is the store siren button
179 if (PORTAbits.RA4 == 0)                                //if the CN11 pin has
    indeed been pressed
180 {
181     PORTBbits.RB8 = 1;                                //turn on LED
        to show storing as template
182     ADCstore = &siren_template[0];                    //tell the ADC to store the
        shit in the template
183     storing_siren = 1;
184     shift_scale = 0;
185     scalar = 1;
186 }
187
188 IFS1bits.CNIF = 0;    //clear interrupt flag
189 }
190
191 void __attribute__((__interrupt__, __shadow__)) _T5Interrupt(void)
192 {
193     /*      Name: _T5Interrupt
194     *      Function: This interrupt executes when the 32-bit timer counts to the
        specified amount
195     *      Explanation: The functionality of this timer is to turn off the led
        and relay indicating a match
196     *      after a specified amount of time.
197     */
198     MATCHLED = 0;                                //turn off LED
199     T5CONbits.TON = 0;                            //Disable Timer
200     IFS1bits.T5IF = 0;    //Clear Timer1 interrupt flag
201 }
202
203 void rescale (void)
204 {
205     /*      Name: rescale
206     *      Function: This function will alter the scaling factor according to the
        current siren template
207     *      Explanation: The scaling factor must be changed according to the
        template so that no overflow occurs.
208     *      The scaling ensures that the result from the VDP will range
        between -1 and 1. It functions by
209     *      running a VDP with the vector which will achieve the maximum
        VDP result and then scaling
210     *      every successive sample based on that. This is done by
        watching for overflow, if it is about to
211     *      occur, then multiply the scaling factor by 2 and dividing the
        VDP result by 2.
212     */
213     int i;
214     fractional vdp_res = 0;
215     scalar = 4;
216     //turn off the A/D so that it does not interrupt this operation and overwrite
```

```

    sample values
217 AD1CON1bits.ADON = 0;          //turn off both A/D and timer3
218 T3CONbits.TON = 0;
219
220 for (i = 0; i < SAMPLE_SIZE ; i++)
221 {
222     //determine whether positive or negative maximum should be used
223     if (siren_template[i] > 0)
224     {
225         sample[i] = 0x7FFF;          //positive maximum
226     }
227     else
228     {
229         sample[i] = 0x8000;          //negative maximum
230     }
231
232     //add the new value
233     vdp_res += (sample[i] * siren_template[i]) / scalar;
234
235     //if the vdp_result is above 0.5, then overflow is possible and must
        scale accordingly
236     if (vdp_res > 0b0100000000000000)
237     {
238         scalar = scalar << 1;          //multiply by 2
239         vdp_res = vdp_res / 2;          //divide by 2
240     }
241 }
242
243 //resume normal functionality of A/D
244 AD1CON1bits.ADON = 1;          //turn on both A/D and timer3
245 T3CONbits.TON = 1;
246 }
```

Listing 2: Project Code for dsPIC33F

```
1  /* File: adc.c
2  *
3  * Written by: ECE 480 Design Team 9, Fall 08
4  *
5  * Contains: function descriptions to use the analog to digital port of the
        dsPIC33FJ128GP802
6  */
7
8
9
10 fractional RegisterA[ADC_NUMSAMP] __attribute__((space(dma)));           //ping
        pong buffer for DMA
11 fractional RegisterB[ADC_NUMSAMP] __attribute__((space(dma)));           //ping
        pong butter for DMA
12
13 void ADC(void)
14 {
15     /*      Name: ADC
16     *      Function: To initialize the analog to digital port and timer3, which
        drives sampling rate
17     *      Explanation: The appropriate configuration registers are set to
        achieve the settings we are looking for.
18     */
19
20     AD1CON1bits.ADON = 0;           //make sure ad is off
21     AD1CON1bits.ADSIDL = 0;         //continue operation in idle mode
22     AD1CON1bits.ADDMABM = 1;        //DMA written in order of conversion
23     AD1CON1bits.AD12B = 1;          //12-bit values
24     AD1CON1bits.FORM = 3;           //fractional = 2, integer = 0
25     AD1CON1bits.SSRC = 2;           //choose timer3 to sample
26     AD1CON1bits.ASAM = 1;           //auto sample
27
28     AD1CON2bits.VCFG = 0;           //use Avdd and Avss
29     AD1CON2bits.CSCNA = 0;          //no scanning
30     AD1CON2bits.CHPS = 0;           //channel 0 only
31     AD1CON2bits.SMPI = 0;           //interrupt after every sample
32     AD1CON2bits.BUFM = 0;           //start filling at start every time
33     AD1CON2bits.ALTS = 0;           //always use sample A from channel
34
35     AD1CON3bits.ADRC = 0;           //use system clock for AD conv clock
36     AD1CON3bits.ADCS = 26;          //668ns/25ns = 26.72, where Tcy = 1/(80 MHz/2)
        = 25ns
37
38                                     //Want to minimize
        ADCS for faster A/
        D
39                                     //MIPS = Fosc/2 = Fcy
        = 80 MHz/2 = 40
        MIPS
        //Tad = Tcy*(ADCS+1) =
        1600 ns > 668 ns
```

```

40                                     limit
                                     //ADC Conversion Time
                                     for 12-bit Tc=14*
                                     Tad = 22.4 us
41
42
43 AD1CHS0bits.CH0NA = 0;           //use Vref-
44 AD1CHS0bits.CH0SA = 11;         //use AN11
45
46 AD1PCFGL = 0xFFFF;              //set up AN11 as analog input, all
    rest digital
47 AD1PCFGLbits.PCFG11 = 0;        // AN11 as Analog Input
48
49 //Timer3
50 TMR3 = 0x0000;
51 PR3 = TMR_SAMP_COUNT;           //counts of how long to count until new sample
    is taken
52 IFS0bits.T3IF = 0;
53 IEC0bits.T3IE = 0;
54
55 //Clear the A/D interrupt flag bit
56 IFS0bits.AD1IF = 0;
57
58 //Set the A/D interrupt enable bit
59 IEC0bits.AD1IE = 0;
60
61 //Turn on the A/D converter
62 AD1CON1bits.ADON = 1;
63
64 //Start Timer 3
65 T3CONbits.TON = 1;
66 }
67
68 void DMA (void)
69 {
70     /*      Name: DMA
71      *      Function: To initialize the Dynamic Memory Access
72      *      Explanation: The configuration registers are set to enable correct
    storing of the A/D samples through the DMA module
73      */
74
75 DMA0CONbits.SIZE = 0;            //transfer in words
76 DMA0CONbits.DIR = 0;            //read from peripheral to SRMA
77 DMA0CONbits.HALF = 0;           //initiate interrupt when all is transferred
78 DMA0CONbits.NULLW = 0;          //no null data write
79 DMA0CONbits.AMODE = 0;          //register indirect with post-increment
    addressing
80 DMA0CONbits.MODE = 2;           //continuous ping pong mode
81
82 DMA0REQ = 13;                   //trigger on ADC1
83

```

```
84 DMA0STA = __builtin_dmaoffset(RegisterA);
85 DMA0STB = __builtin_dmaoffset(RegisterB);           //this is registers a
    and b
86
87 DMA0PAD = (volatile unsigned int)&ADC1BUF0;         //get info from adc
    buffer
88
89 DMA0CNT = ADCNUMSAMP;           //number of transfers from adc
90
91 IFS0bits.DMA0IF = 0;           //Clear the DMA interrupt flag bit
92 IEC0bits.DMA0IE = 1;           //Set the DMA interrupt enable bit
93
94 DMA0CONbits.CHEN = 1;           //enable channel
95 }
96
97 unsigned int DmaBuffer = 0;
98 void __attribute__((__interrupt__, __auto_psv__)) _DMA0Interrupt(void)
99 {
100     /*      Name: _DMA0Interrupt (Dynamic Memory Access interrupt)
101     *      Function: To store the A/D samples to the correct location in memory
102     *      Explanation: The DAM is triggered after every sample so that matched
103     *                    filtering can be run.
104     *                    When this interrupt is triggered the DMA module chooses where
105     *                    to store the sample.
106     *                    The sample is either stored as the siren template or the
107     *                    outside sample.
108     *                    The ADCstore pointer is set outside of this function to define
109     *                    where to locate the new samples.
110     */
111
112     int i;
113     RunFilter = 1;
114
115     //this is the implementation of the ping-pong DMA buffers, one can be written
116     //to while the other is being stored
117     if (!DmaBuffer) //storing to RegA currently
118     {
119         for (i = 0; i < ADCNUMSAMP; i++)
120         {
121             *ADCstore++ = (RegisterB[i] / scalar);
122             ind++;
123         }
124     }
125     else
126     {
127         for (i = 0; i < ADCNUMSAMP; i++)
128         {
129             *ADCstore++ = (RegisterA[i] / scalar);
130             ind++;
131         }
132     }
133     DmaBuffer ^= 1;           //toggle which buffer to write to
```

```

128
129 if (storing_siren)           //are we storing the siren currently?
130 {
131     //check whether the ADCstore pointer has incremented past the array (
        array is full)
132     if (ADCstore >= &siren_template[SAMPLE_SIZE])
133     {
134         ADCstore = &sample[0]; //start storing samples to the outside
            noise location
135         storing_siren = 0;           //reset the flag specifying
            that we are storing the siren
136         PORTBbits.RB8 = 0;         //turn off the LED saying that
            the siren is being stored
137         ind = 0;                   //reset the index
138
139         rescale();                 //run the function to
            scale all of the following inputs based on the maximum
140                                     //siren
                                     template
                                     VDP value
141     }
142 }
143 //the siren is not being stored, check if the ADCstore pointer has reached the
    end of the outside sample queue
144 else if (ADCstore >= &sample[SAMPLE_SIZE])
145 {
146     ADCstore = &sample[0];         //reset the ADCstore pointer to the
        beginning of the outside noise buffer
147     ind = 0;                       //reset the index in
        use
148 }
149
150
151 IFS0bits.DMA0IF = 0;             //Clear the DMA interrupt flag bit
152 }

```

Listing 3: Project Code for dsPIC33F

```
1  /* File: common.h
2  *
3  * Written by: ECE 480 Design Team 9, Fall 08
4  *
5  * Contains: Function headers, global variable declarations
6  */
7
8
9  #include <p33FJ128GP802.h>
10 #include <dsp.h>
11
12 /***** Variable Declarations *****/
13 #define MATCHLED PORTBbits.RB9
14 #define MATCHLTHRESHOLD 90
15 #define Hold_time 1 //hold time in seconds for how
    long to signify match
16 #define FCY 4000000 //40MHz
17 #define TCY 1/FCY
18 #define Hold_cycles Hold_time/TCY
19
20 #define ADC_NUMSAMP 1
21 #define SAMPLINGRATE 6000
22 #define TMR_SAMP_COUNT (FCY/SAMPLINGRATE)-1
23
24 #define SAMPLE_SIZE SAMPLINGRATE/2
25
26 /***** Global Variables *****/
27 //this specifically states the siren to be in Y data space, and the sample to
    be
28 //in X data space, doing this maximizes the speed of the Multiply ACcumulator
    function (MAC)
29 //which is used repeatedly in the Vector Dot Product
30 fractional _XDATA(32) siren_template[SAMPLE_SIZE];
31 fractional _YDATA(32) sample[SAMPLE_SIZE];
32
33 fractional result; //holds the result of the VDP
34 fractional * ADCstore; //pointer to where the ADC stores its info
35 char RunFilter; //flag to specify when VDP runs
36 char storing_siren = 0; //flag to speify whether the siren template is
    being stored
37 unsigned int matches; //counter to say how many matches we have had
38 int ind; //index value of the sample
    which is next to be stored
39 int scalar = 1; //the value to scale all of the read A
    /D values by
40 /* end global variables */
41
42 /***** Function Declarations *****/
43 /* The VectorDotProduct function was written by Microchip and is
    optimized for speed. */
```

```

44 extern fractional VectorDotProduct
45         (          int numElems,
46                   fractional* srcV1,
47                   fractional* srcV2      );
48
49 /*      Name: CLOCKinit (clock initialization)
50 *      Function: To set up the clock to achieve maximum speed
51 *      Explanation: This step is necessary in order to get 40MIPS operation.
52 *                  The onboard RC oscillator is sent through the PLL to get this
                    speed.
53 *
54 *                  Taken from http://ww1.microchip.com/downloads/en/DeviceDoc/70216B.pdf
55 *                  Page 22
56 */
57 void CLOCKinit(void);
58
59 /*      Name: CNinit (change notice initialization)
60 *      Function: To set up the change notice pin
61 *      Explanation: The change notice pin is used to detect a button press
62 */
63 void CNinit(void);
64
65 /*      Name: TMRinit (timer initialization)
66 *      Function: To set up the timer that keeps the match LED on for
67 *                  an extended period of time
68 *      Explanation: This timer controls the matched led. Since we want the
69 *                  LED to be on for 5 seconds, the timer needs to be 32-bits
70 */
71 void TMRinit(void);
72
73 /*      Name: VDP (vector dot product)
74 *      Function: To perform a single vector dot product
75 *      Returns: The fractional result of the VDP matched filtering, ranging
                    from -1 to 1
76 *      Explanation: As long as the siren template is not currently being
                    stored
77 *                  we will use the sample and the siren template arrays and
                    calculate the
78 *                  VDP. This result will range between -1 and 1 in a signed 16-
                    bit fractional
79 *                  because of the scaling function. The VectorDotProduct was
                    included in the
80 *                  DSP library by microchip, it is fast.
81 */
82 fractional VDP(void);
83
84 /*      Name: rescale
85 *      Function: This function will alter the scaling factor according to the
                    current siren template

```

```

86 *      Explanation: The scaling factor must be changed according to the
      template so that no overflow occurs.
87 *      The scaling ensures that the result from the VDP will range
      between -1 and 1. It functions by
88 *      running a VDP with the vector which will achieve the maximum
      VDP result and then scaling
89 *      every successive sample based on that. This is done by
      watching for overflow, if it is about to
90 *      occur, then multiply the scaling factor by 2 and dividing the
      VDP result by 2.
91 */
92 void rescale(void);
93
94 /*      Name: ADC
95 *      Function: To initialize the analog to digital port and timer3, which
      drives sampling rate
96 *      Explanation: The appropriate configuration registers are set to
      achieve the settings we are looking for.
97 */
98 void ADC(void);
99
100 /*      Name: _DMA0Interrupt (Dynamic Memory Access interrupt)
101 *      Function: To store the A/D samples to the correct location in memory
102 *      Explanation: The DAM is triggered after every sample so that matched
      filtering can be run.
103 *      When this interrupt is triggered the DMA module chooses where
      to store the sample.
104 *      The sample is either stored as the siren template or the
      outside sample.
105 *      The ADCstore pointer is set outside of this function to define
      where to locate the new samples.
106 */
107 void __attribute__((__interrupt__)) _ADC1Interrupt(void);

```