2012

Car Alarm System

BY

David Lane

davwillane@yahoo.com

California Polytechnic State University

San Luis Obispo, CA

Advisor: David Braun

Revision 7
3/20/2012

Table of Contents

Table Index

Circuit Index

Simulation Index

Figure Index

Abstract

This report details the design and implementation of a Car Alarm System specifically for vintage cars. It covers every aspect of the design process, from the starting design specifications and concerns to the final product. Project goals included a system that would provide adequate car security without the annoyance of general car alarms, such as sounding when lightning strikes or a dog barks. This project entailed programming, circuit designing, and application of various sensors aspects. It demonstrates an Electrical Engineers' knowledge and abilities upon graduation from California Polytechnic State University in San Luis Obispo, CA.

Introduction

Objective

The initial goal for this project was to fill my immediate need for an alarm system. I own a vintage car, and wanted to install a security system that would protect my car and anything I put in it. A senior project creating my own alarm system seemed convenient. However, after researching current car alarm systems, I found that the general public disliked car alarm systems. When asked why, two main reasons came up: Obnoxious sounding sirens that lasted for a long time, and false alarms. My goal then turned from not only filling my personal need, but also to improving the design to compensate for these complaints. I was now presented with the task of designing a car alarm system with unique and less annoying siren noise that would adequately protect vehicles from being stolen and from basic break-ins, but would not trigger any otherwise.

Design Requirements

A car alarm system with unique and less annoying siren noise than current models that would protect vehicles against window breaks and door openings and would also have wireless transmitters for Enabling and Disabling the alarm system.

Project Specifications

- A wireless button transmitter smaller than 3” X 3” X 1” that can be attached to a key chain. It will need two buttons, one for enabling and the other disabling the security system.
- Door Sensors for both driver side and passenger side door that will sense when the doors are open. These sensors need to be placed in the door jamb to allow them to be concealed when the doors are shut. Therefore, they need to be no larger than 2” X 2” and embedded in the frame of the car.
- Window Sensors that will detect when any of the six windows are broken. The sensors should not trigger off of anything except a window break.
- A button module with at least four buttons that deactivates the alarm system as a backup to the wireless transmitter remote. It will need to be powered only when the alarm system is active, must be smaller than 6” X 4”, and must be able to connect to a microcontroller. It will be concealed under the back hatch by the motor. Pressing the correct code will deactivate the security system.

The Planning

Having taken EE 460 in the Fall, I decided to use PERT Charts to plan out this project. A preliminary chart was drawn up with time and cost estimates. Adjustments were made to both estimates after I had started my research. I created a PERT Chart at the very beginning of the project after I had completed some preliminary research. I then re-evaluated my time line and adjusted the PERT Chart in-between Winter and Spring Quarter 2011. I used this chart to complete my project, but the time line needed to be adjusted due to difficult Winter quarter and working full time for Hewlett Packard. The final PERT Chart shows an accurate time line of how long the entire project took. All PERT Charts are at the end of the document.

The Design

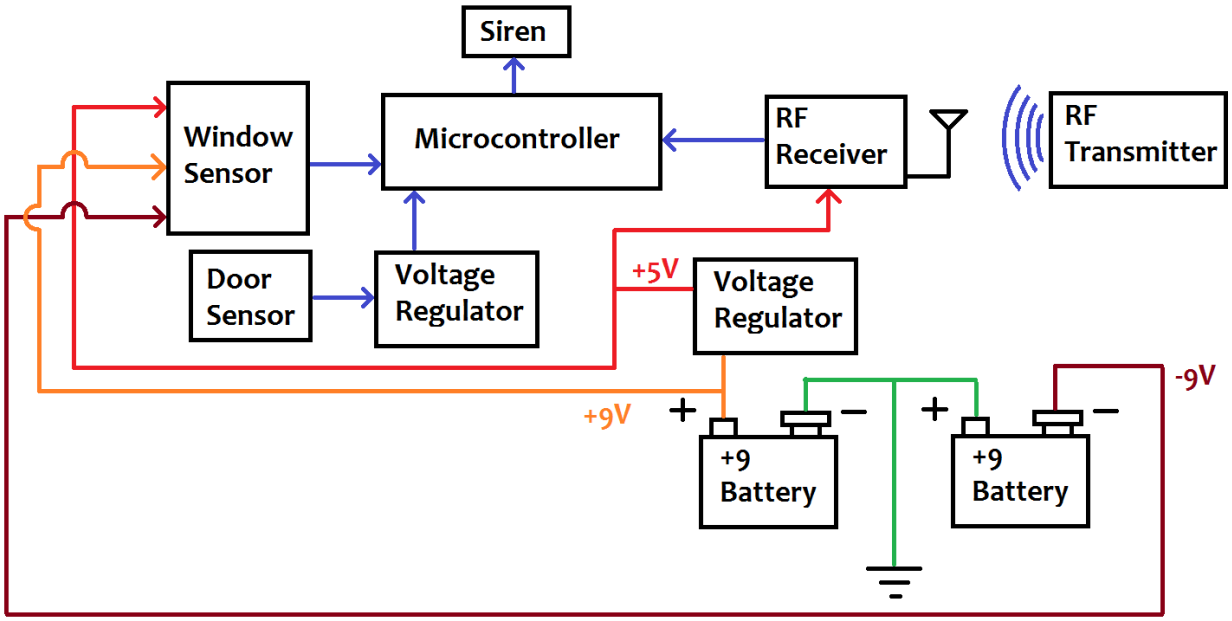
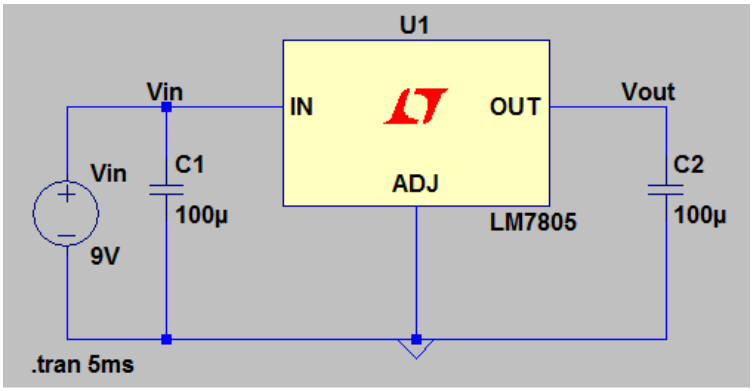


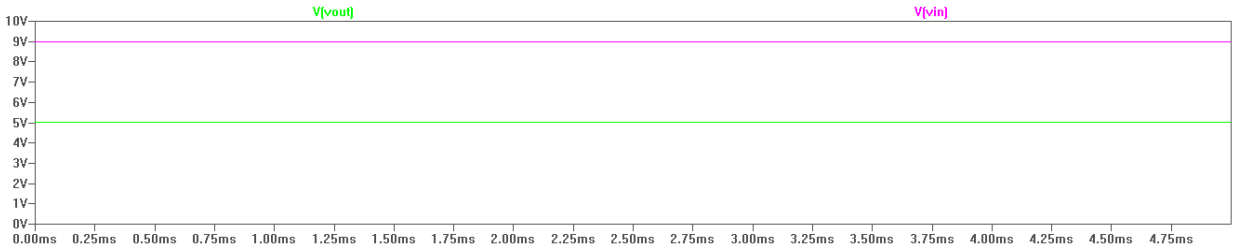
Figure 1: Alarm System Block Diagram

Powering the Alarm System

Originally I thought I would simply use the car battery, but I decided instead to use standard 9V lithium batteries. One was used to be the negative rail on the LM741 Op Amps, the second was used for the positive rail on the LM741 Op Amps as well as the power for the various components. However, most of the components I chose needed +5V. I used a LM7805 voltage regulator in the following circuit to step down the DC voltage from 9V to 5V. The Bill of Materials for this circuit is on page 33



Circuit 1: 9V to 5V Voltage Regulator



Simulation 1: 9V to 5V Voltage Regulator Simulation

Component Designs

Button Module: Design Choices

There were three design choices I considered for the Button Module. My first option was to design my own button module and circuit. I would need to purchase individual buttons, and I could then either use an analog process to check for a correct code or I could send each button signal to the microcontroller and check the code digitally. Either way would not be difficult, however a lot of time would need to be designated on manufacturing the device.

My second option was to purchase an external USB number pad. This option eliminated the time I would need to spend on manufacturing the button module, but it created a problem on the microcontroller end. I would need a USB input into the microcontroller, which would add to the cost and complexity of my microcontroller.

My third option was to purchase an external keypad. These are relatively cheap and extremely basic. By applying a Vdd signal to one of the pins, pressing a button would simply connect that button's corresponding pin to Vdd. The problem was that the button module would have as many outputs as I had buttons in my code.

	Design Choice	Pros	Cons
1	Design personal button keypad	Design Flexibility, Uniqueness	Difficulty, Time
2	Utilize computer external number pad	Could be made wireless, More buttons -> More security	USB connection, Cost
3	Utilize basic external keypad	Simplicity, Cheap	Output for each button

Table I: Button Module Design Choices

Button Module Design Choice: KEYPAD

I chose the third design choice for the button module because of its simplicity. The only downside was that since it had an output for each button, the microcontroller would need that many more inputs. However, I had no requirements for the size of the microcontroller, so that would not be a problem. I purchased the external keypad for $5.70 from Electronic Parts Supermart located in Santa Maria.

Figure 2: Keypad Purchased

Symbol	Output Pin
(V_{dd})	1
	2
1	6
2	10
3	14
4	5
5	9
6	13
7	4
8	8
9	12
0	7
*	3
#	11

Table II: Keypad Symbols and Corresponding Pins

Window Sensor: Design Choices

I also had three design options for my window sensor. My first option was to design and implement my own vibration sensor. In order to break a window, a blunt force would need to be applied. This force would create vibrations through the car, which could be read by a vibrations sensor. Downsides would be that the car alarm system would be prone to false alarms. The vibration sensor would not be able to tell the difference between the force of a window breaking and the force of anything else bumping into the car.

My second option was to use a microphone to capture the sound of glass breaking. I could then compare it to a recorded signal of glass breaking or I could simply filter the microphone using a bandpass filter. Downsides would again be that the car alarm system would be prone to false alarms. The microphone would not be able to tell the difference between the sound of glass breaking and any other sounds at the same frequency.

My third design option involved laser detectors. By placing a small mirror on each window, I would bounce a laser off each window and eventually back to a sensor. If any window was broken, the laser ray would be broken. Downsides would be the difficulty in accurately placing mirrors and reflecting the laser.

	Design Choice	Pros	Cons
1	Vibration Sensor	Cheap, Easy to Program	Prone to False Alarms
2	Microphone	Cheap, Easy to Program	Prone to False Alarms, Extra Circuit
3	Laser Detector	Reliability	Installment Difficulty

Table III: Window Sensor Design Choices

Window Sensor Design Choice: Vibration Sensor + Microphone

I chose to create window break sensor that incorporated both the vibration sensor and the microphone. By using both sensors, the system reliability is greatly improved and the problem with false alarms is eliminated. For the microcontroller to read a broken window, both the vibration sensor will need to trigger and the microphone will need to read a sound at the correct frequency. Only when both these event occur will the alarm go off.

After speaking with my ME Professor Terry Cook, I decided to use an Accelerometer for a vibration sensor. However, I changed to using multiple basic vibration sensors due to their size and simplicity.

Figure 3: Piezo Electric Vibration Sensor Purchased

Figure 4: Microphone Purchased

Wireless Transmitter: Design Choices

I had two design choices for my wireless transmitting device. My first design choice was for a radio frequency transmitter. However, the receivers listed online were bulky and had large antennas. I was suggested to look into garage door company's remotes. They would contain buttons already configured to the transmitter. This would require configuring to meet my need, but would provide simpler manufacturing.

My second choice was using Bluetooth drop-in modules, which was suggested by ME Professor Terry Cook. By having Bluetooth modules both at the Microcontroller and the remote, not only can the remote talk to the microcontroller, but the microcontroller can talk to the remote. Using Bluetooth for the remote would allow more design flexibility, but would need button installment, a microcontroller in the remote and housing manufacturing.

	Design Choice	Pros	Cons
1	Radio Frequency Transmitter	Simpler configuration	Size, low transmission distance
2	Bluetooth Drop-in Module	Size, long transmission distance	Complicated housing manufacturing

Table IV: Wireless Transmitter Design Choices

Wireless Transmitter Design Choice: RF Transmitter

I originally chose to use two Bluetooth drop-in modules to wirelessly transmit a signal from my remote unlocking device to the microcontroller. However, I soon realized the difficulty in programing these devices. I felt that I was spending too much time on something that should be simple, so I changed to using a simple radio transmitter. I chose a universal garage door opener shown below.

Figure 5: Radio Frequency Transmitter

Figure 6: Radio Frequency Receiver

Alarm Noise: Design Choices

I had two design choices for my alarm noise. My first design choice utilized the current car horn. However, I struggled with this option and the microcontroller being reset whenever the horn beeped in the alarmed state.

My second choice used an external buzzer or siren. This option is much simpler, required no extra circuitry, but did require a second noise maker when the car horn was already present.

	Design Choice	Pros	Cons
1	Car Horn	Cheaper, utilized current car horn	Complicated switching circuit
2	External Buzzer or Siren	Simplicity	Required extra horn

Table V: Alarm Noise Design Choices

Alarm Noise Design Choice: 102 dB Piezo Electric Siren

I originally chose to use the first design choice utilizing the existing car horn. However, due to growing complexity, time constraints, and the inability to solve an issue of resetting the microcontroller, I changed to the second design choice. I purchased the 102 dB Piezo Electric Siren.

Figure 5: 102 dB Piezo Electric Siren

Component Information

KEYPAD

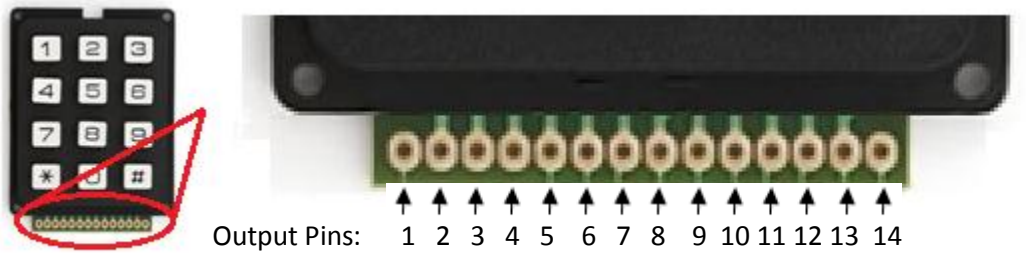


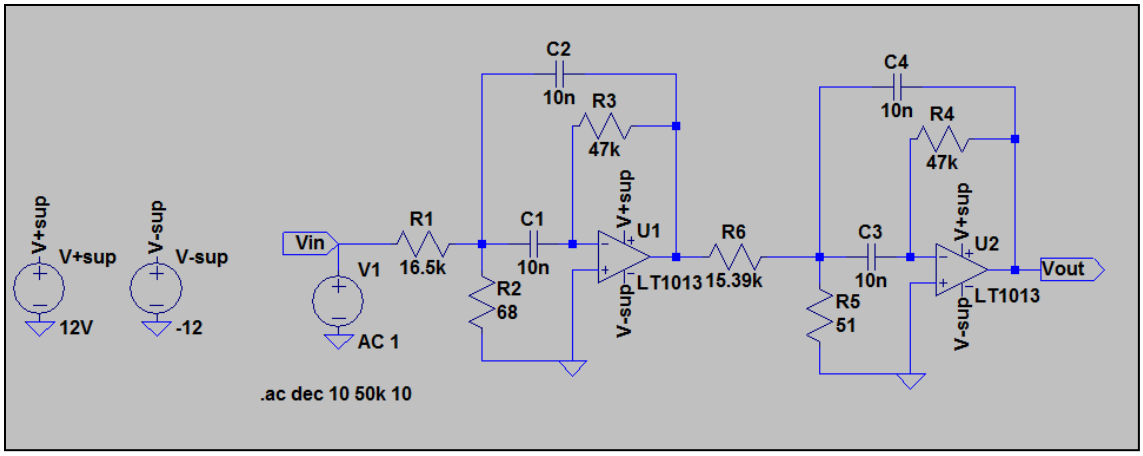
Figure 8: Keypad Output Pins

V_{dd} is connected to Pin 1. For testing purposes, Vdd was set to +5V. When configured to work with microcontroller, V_{dd} will need to be set to V_{high} of the microcontroller inputs. Each button output pin from the keypad is wired to an input pin on the microcontroller. When a button is pressed, the corresponding keypad output pin will become a high.

MICROPHONE

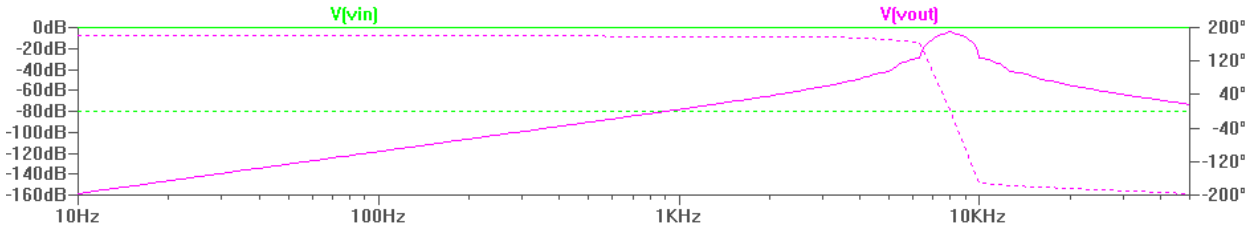
There are three pins on the Electric Microphone: AUD, GND, and VCC. VCC needs to be within 2.7V and 5V. The Breakout board the microphone is mounted on comes with a 100x operational amplifier to amplify the sounds. Since the window sensor should only trigger off of noises within the same frequency as that of breaking glass, a bandpass filter was used to capture the signal only within that frequency.

I chose a Sallen Key Bandpass filter. My goal for a center frequency was 8kHz and corner frequencies of 7.5kHz and 8.5kHz. Below is the circuit I designed to do this.



Circuit 2: Sallen Key Bandpass Filter for Microphone

The Bill of Materials for this circuit is on page 33



Simulation 2: Sallen Key Bandpass Filter

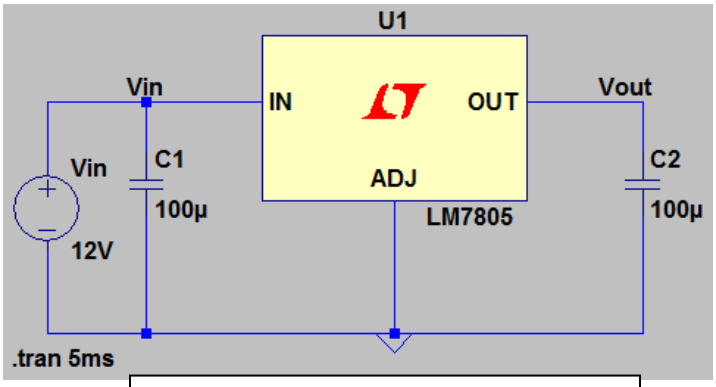
The output of the bandpass filter is connected to an interrupt pin on the microcontroller to ensure the microcontroller notices any input from the microphone

DOOR SENSOR

For the door sensor in my car alarm system, I decided to use the existing door sensors that are used for the head lamps shown below.

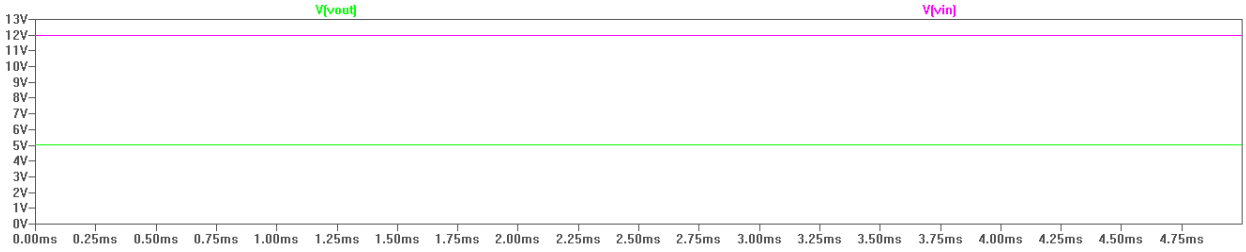
Figure 9: Door Sensor

There are two of these sensors in my car, one on each door. When the doors are shut the output of the door sensor is 12V from the car battery. When the doors are open, the output of the door sensor is grounded. Since the microcontroller is working off of 5V, I needed to bring this 12V down to 5V. To do this, I used a LM7805 voltage regulator in the circuit shown below. The output of the voltage regulator is connected to an input on the microcontroller.



Circuit 3: 12V to 5V Voltage Regulator

The Bill of Materials for this circuit is on page 33



Simulation 3: 12V to 5V Voltage Regulator Simulation

VIBRATION SENSOR

Chosen Option: Piezo Electric Film Vibration Sensor

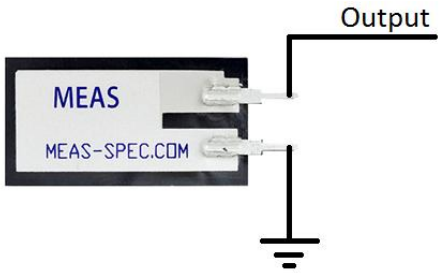


Figure 10: Wiring Single Vibration Sensor

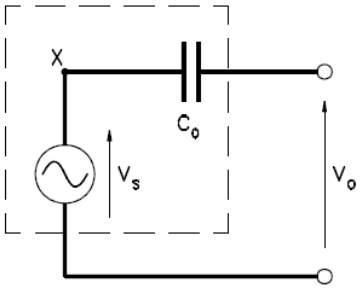


Figure 11: Piezo Electric film element as a voltage generator

I was having difficulty tuning my vibration sensors to read window breaks, so I decided to further test different mounting methods. I first obtained scrap pieces of glass from San Luis Glass and Window Company, and then I built a simple glass frame. I tried multiple different mounting techniques for the vibration sensor to the window. I connected the output of the vibration sensor to a summing amplifier, then the output of the summing amplifier to an interrupt on the microcontroller. I wrote a simple program that turned an LED on when an input was read in the interrupt service routine.

1) I tried mounting the sensor flat against the window, but that not only failed to read striking the glass, but it also failed to read breaking the glass.
2) I tried mounting the sensor in an arc shape against the window, but this too failed.
3) I found the following mounting technique to work the best.

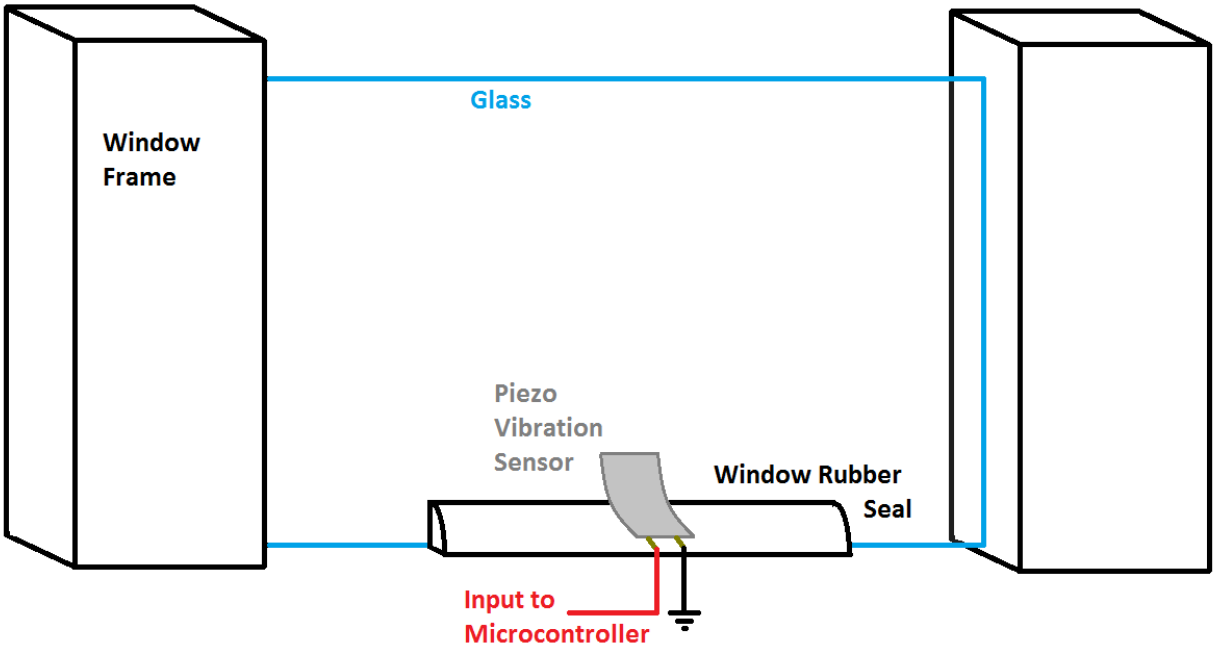
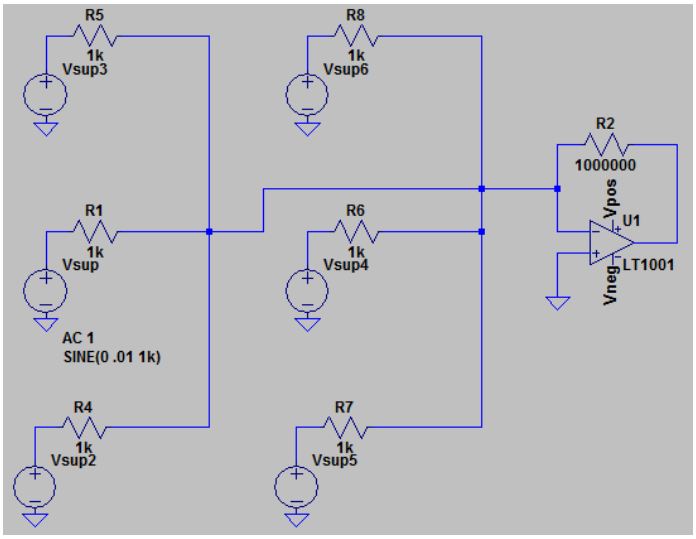


Figure 12: Testing Station for Vibration Sensor

By placing the vibration sensor on the rubber seal around the window, a window break would bend the vibration sensor rather than just vibrating it. Doing this triggered the microcontroller interrupt and turned the LED on.

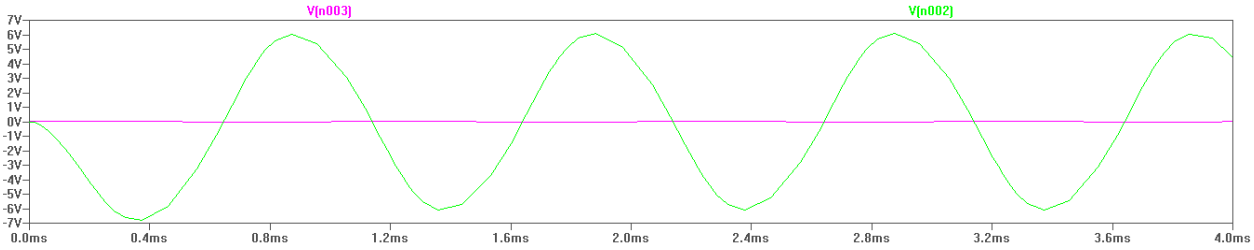
Since these vibration sensors are small and easily hidden, I decided to purchase and install multiple of them. I have installed one in-between each of the windows. The outputs of each of the vibration sensors go to a summing amplifier, and then the output of the summing amplifier goes to the microcontroller.

Figure 13: Location of Vibration Sensors



Circuit 4: Inverting Summing Amplifier

The Bill of Materials for this circuit is on page 34



Simulation 4: Inverting Summing Amplifier Simulation

Original Option: Accelerometer

In the beginning of this project, I decided to use an accelerometer as my vibration sensor. In order for the accelerometer voltage to be read by the microcontroller, there were two options. The voltage could be processed digitally by the microcontroller, or it could be processed with an analog circuit before the microcontroller. Either option would have worked, but the problem I foresaw with the Accelerometer was its size and the fact there was only one of them. The accelerometer was larger and harder to install than the Piezo Electric film vibration sensor. Also, since there was only one accelerometer, and suppose it

was installed in the front of the car, I was concerned that it would not read a vibration in the back of the car. I have shown the two design options for the accelerometer below.

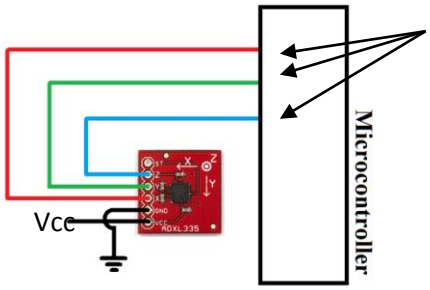


Figure 14: First Design Choice for Connecting Accelerometer to Microcontroller

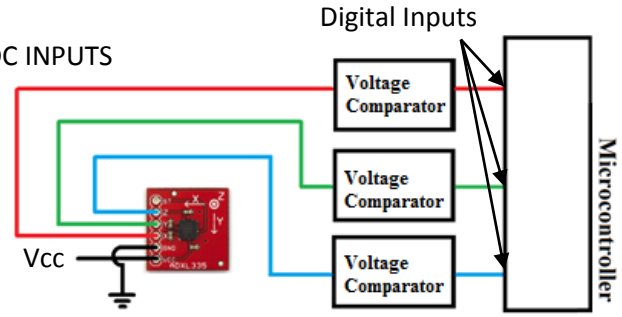


Figure 15: Second Design Choice for Connecting Accelerometer to Microcontroller

WIRELESS TRANSMITTER

Chosen Option: Universal Garage Door Opener

The SKYLINK G6VR UNIVERSAL GARAGE DOOR REMOTE is a universal remote that can be programed to work with different SKYLINK receivers and is intended to be used as a garage door opener. The receiver is powered by a DC voltage (5V – 12V). When the remote button is pressed, the two wires from the top are shorted. Below is the diagram for how I have wired this universal remote to work with my microcontroller.

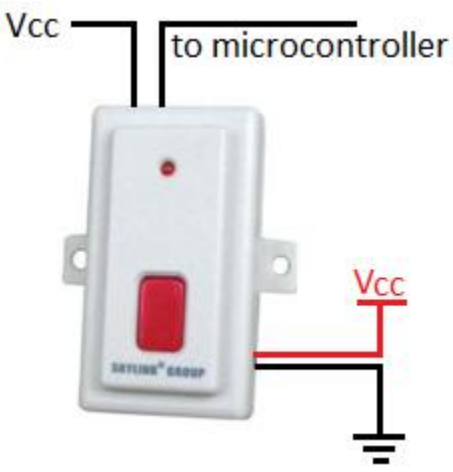


Figure 16: Wiring Universal Garage Door RF Receiver

I connected both Vcc connection shown above to my +5V rail, and the output to an input on the microcontroller. After making these connections, I tested the remote by using LEDs to show the current state and having the remote switch between states.

Original Option: Bluetooth Devices

I originally chose to use RN41 Bluetooth devices for the remote device because I thought it would make my project more unique. However, I found the code for programing the devices unnecessarily difficult. Whenever the remote button for unlocking the alarm system was pressed, the Bluetooth device in the remote would have to find the Bluetooth device in the car, check to see if it had already established a connection, and if there was no connection already established then connect to it. Once connected, it would then need to send the Bluetooth device in the car the unlock code and wait to receive confirmation. The Bluetooth device in the car would have to constantly check if it was connected and if had received the unlock code. I dubbed the constant checking and sending and receiving bits to be unnecessary for such a simple task. That is why I changed to the much simpler RF transmitter/receiver.

Figure 17: Top View of RN41

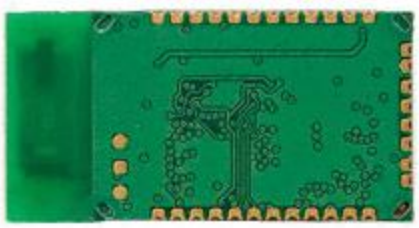


Figure 18: Bottom View of RN41

ALARM NOISE

Chosen Option: 102dB Piezo Electric Siren

The 102dB Piezo Electric Siren was chosen as a replacement option to the intial effort of utilizing the existing car horn. It is a very simple device that only requires a 5-10 DC Voltage. This was perfect for wiring to the microcontroller. As shown in the figure below, the red wire connects to the microcontroller output and black wire connects to ground.

Figure 19: Piezo Electric Siren

After connecting the Piezo Electric Siren to the microcontroller, it was tested by simply moving to the alarm system enabled state, then by opening the door. This moves to the alarming state and sounded the siren.

Original Option: Car Horn

The original design used the car horn for my alarm. I struggled for two days with this design choice. John S. Henry's BugShop FAQs shows the following schematic for the horn.

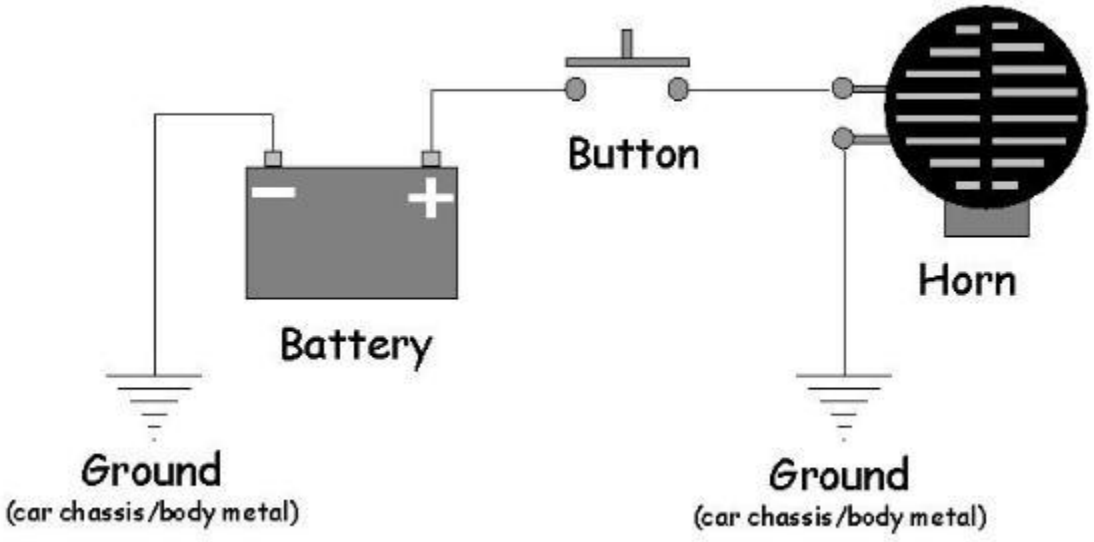


Figure 20: Vintage VW Bug Horn Schematic

This shows that for the horn to make noise, 12V must be applied from the battery. However the microcontroller can only output 5V. In order to have the microcontroller sound the horn, I decided to use an Auto Relay. The microcontroller's 5V output would trigger the relay connecting Vcc and the horn, where Vcc is the car battery. Below is the relay I purchased from Radio Shack, and below that is the design for connecting the relay to the microcontroller and the car battery.

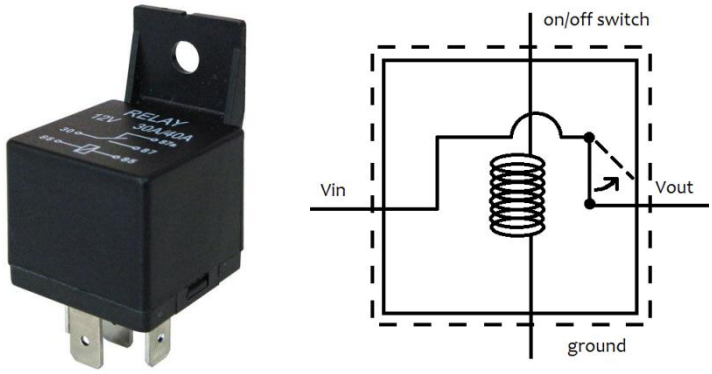


Figure 21: 12V SPDT 30 AMP AUTOMOTIVE RELAY

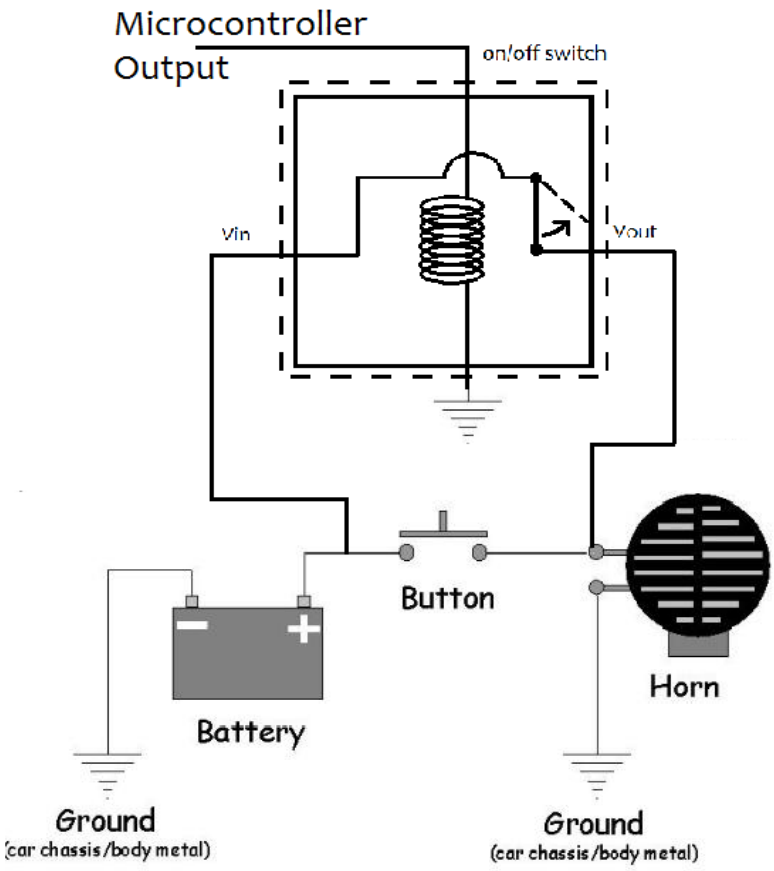
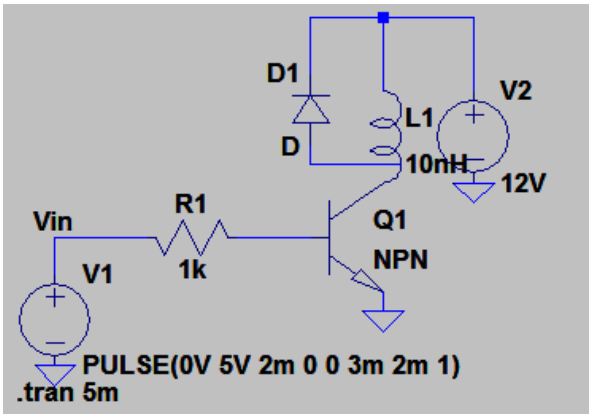


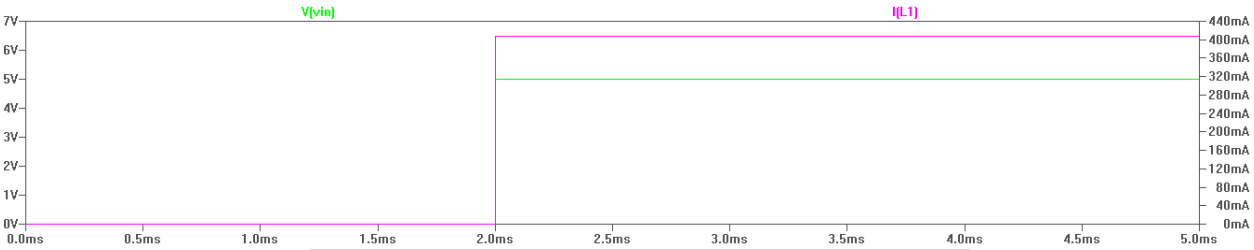
Figure 22: Wiring Relay with Horn

The problem with this design is the microcontroller is unable to put out enough amps to switch the relay. In order to switch the relay, I designed the following circuit.



Circuit 5: BJT Switch for Horn Relay

The Bill of Materials for this circuit is on page 34



Simulation 5: BJT Switch for Horn Relay Simulation

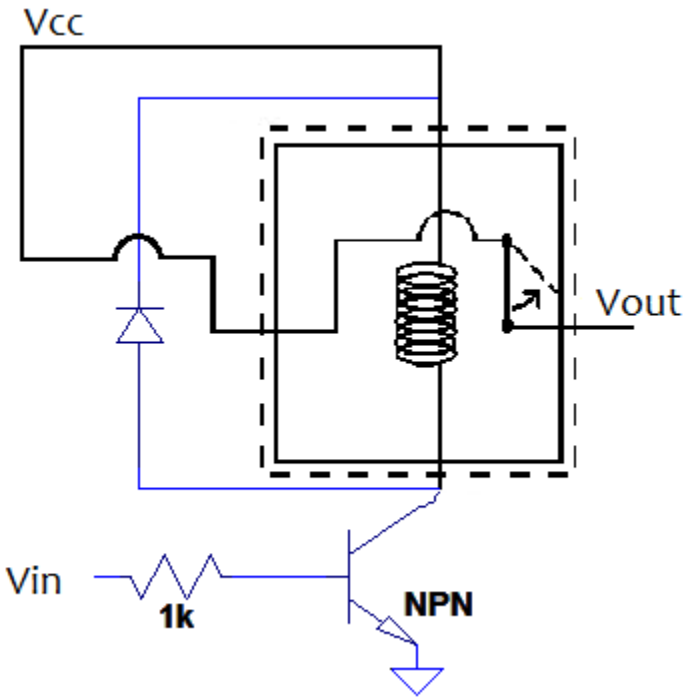


Figure 23: Wiring Relay with BJT Switch

Vin is the the 5V output from the microcontroller, Vcc is the car battery 12V, and Vout is the voltage applied to the horn.

Problem with Using Car Horn

After opening a door and setting off the car alarm, the horn would beep once but not beep continuously like my program intended. This was because after transferring to the alarming state and beeping the horn, the microcontroller would reset. As seen in Figure 20 on page 22, the relay is powered by the car battery. Therefore, I needed a common ground. I to connect the car battery ground with the ground of the two 9V batteries. The horn beeping when in the alarming state creates a large current spike. This current spike was then translated to a voltage spike seen on the connected ground terminals of the two 9V

batteries. That spike caused the voltage of the batteries to momentarily drop below the needed 9V supplied to the LM7805 voltage regulators to create the 5V that powers the microcontroller. This caused the microcontroller reset.

I spent one whole day diagnosing the issue and another day trying to find a solution.

- I tried connecting the negative terminal of the horn all the way to the car battery negative terminal rather than connecting it to the alarm system circuit ground, hoping this would reduce the voltage spike seen on the microcontroller.
- I tried connecting a 22uF capacitor in series with the ground connection from the horn to the ground terminal of the battery, hoping this would reduce the current spike.

Due to an inability to find a solution, growing complexity, and mainly time constraints, I decided to change my design choice to an external siren.

Component Placement

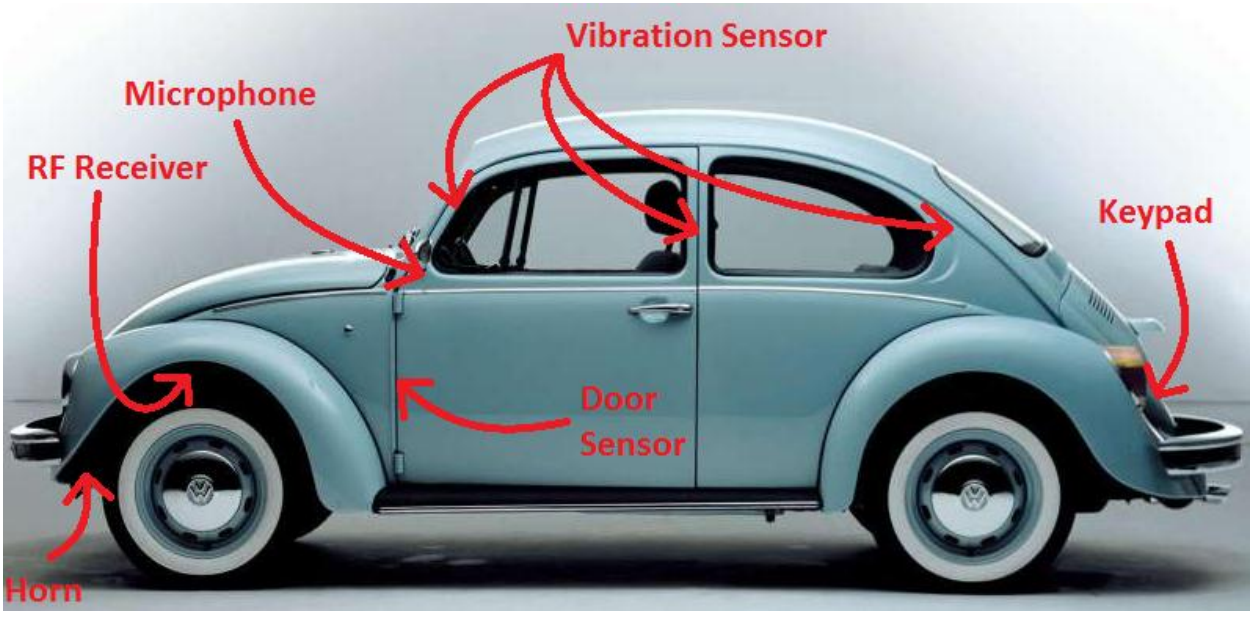
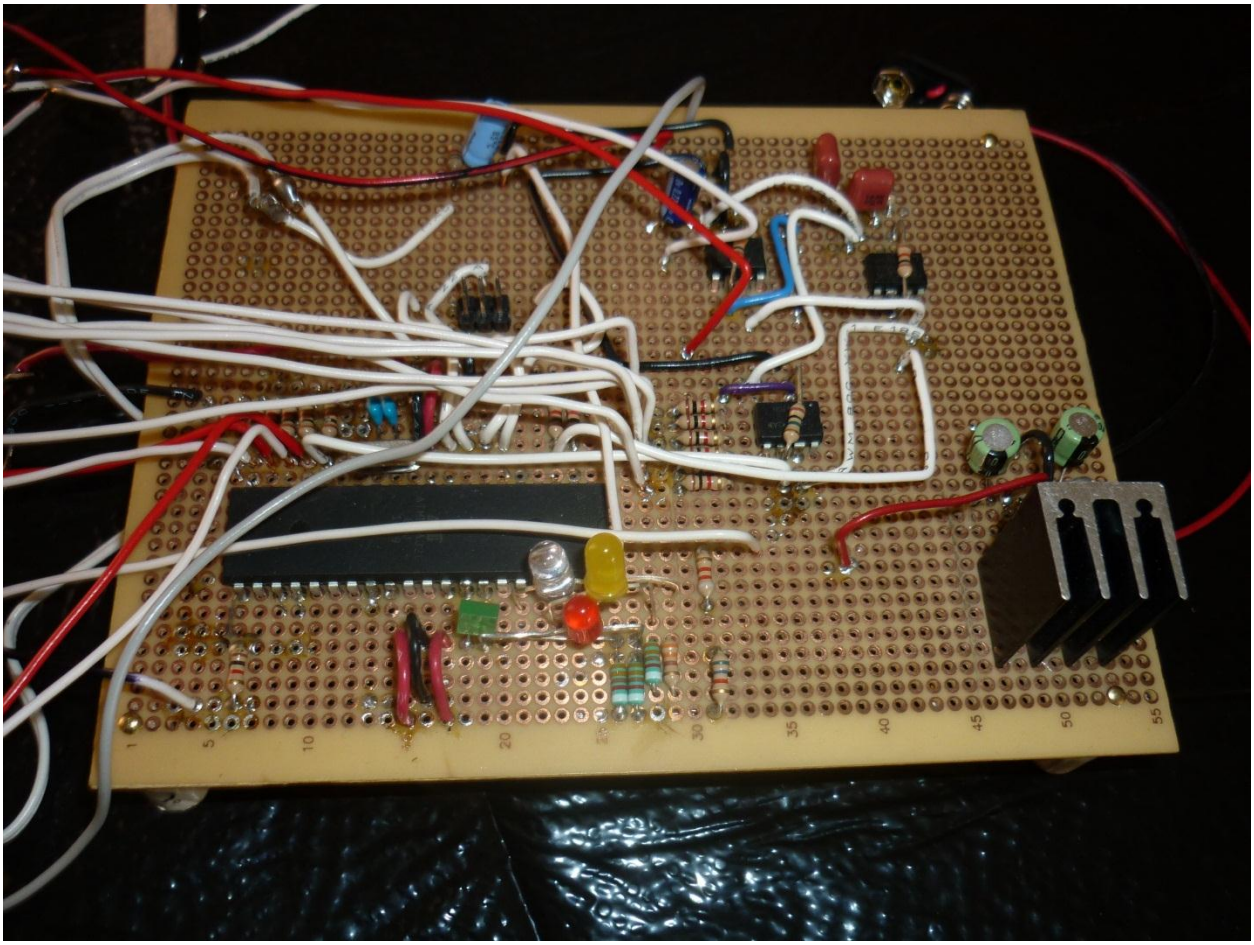


Figure 24: Component Layout in Car



Picture 1: Picture of Assembled Alarm System

The Programming

Since I started this project while I was enrolled in a mechatronics course, and we used C++ to program microcontrollers, I decided to do the same for my senior project. The programming section has been divided into subsections for each object and task file. There is one basic main file, one task file, and two object files.

Alarm_System_Test.cpp

The basic purpose of this file is to create objects, tasks, and then call them within main. It creates all the objects first, and then when it creates the alarm task, it passes it pointers to all the different objects just created. Once within main, it enters an infinite loop where it calls the schedule function of alarm task. Alarm _System_Test.cpp code is on page 58.

	Header Files Included	Description
1	stdlib.h	Standard C library
2	rs232int.h	Header for serial port class
3	stl_timer.h	Header for timer class
4	keypad_driver.h	Header file for keypad driver
5	door_sensor_driver.h	Header file for door sensor driver
7	alarm_task.h	Header file for alarm task

Table VI: Header File List in Alarm_System_Test.cpp

Alarm_task.cpp

This file is made a descendant of the stl_task class in order to utilize its scheduling functions. The purpose of the alarm_task file is to keep track of the alarm system state and to call functions from the different objects based on that state. The file's constructor saves the various pointers locally, initializes variables, and sets up the ports. Within the Run function, there are three different states. The alarm system can either be Disabled, Enabled, or Alarming. Alarm_task.cpp code is on page 60 and its header file Alarm_task.h is on page 66.

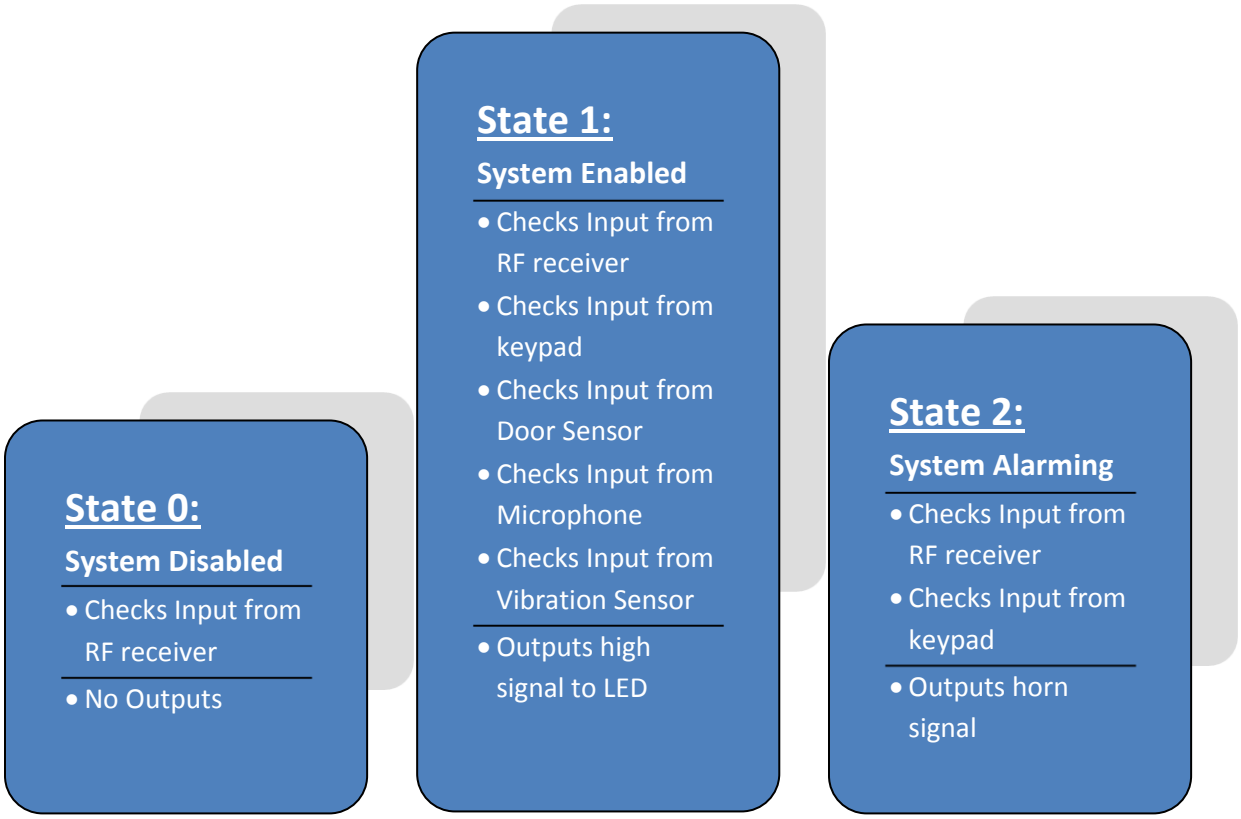


Figure 25: Actions within Alarm System States

When the object's functions within each state are called, they return values that determine the next state. For instance, when in State 1, the alarm system is enabled but not sounding. If the door sensors function check sensors returns true, then the state transitions to State 2, System Alarming. The state transition diagram below summarizes all the possible transitions.

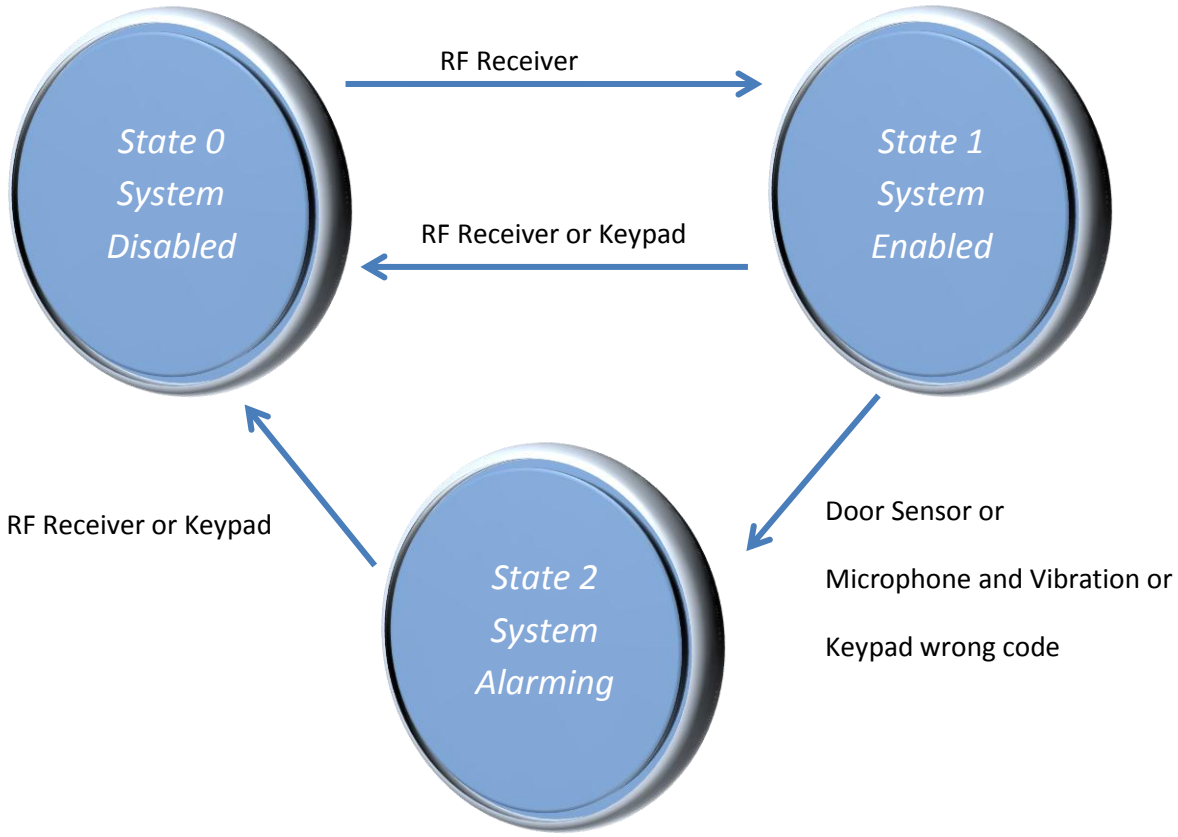


Figure 26: State Transition Diagram for Alarm_task.cpp

	Header Files Included	Description
1	stdlib.h	Standard C library
2	rs232int.h	Header for serial port class
3	stl_timer.h	Header for timer class
4	keypad_driver.h	Header file for keypad driver
5	door_sensor_driver.h	Header file for door sensor driver
7	alarm_task.h	Header file for alarm task

Table VII: Header File List in Alarm_task.cpp

Keypad_driver.cpp

This file contains a program which reads inputs from an external keypad. The constructor sets up the ports it will use for the keypad inputs as well as initializes the count variables and the different flags. The check inputs function checks the different ports for the correct code. The correct code is hard coded and takes into account the order at which the buttons are pressed. If entered correctly, the function will return a '2' for disarming the alarm system. If entered incorrectly three times, the function will return a '1' for sounding the alarm system. The code also allows the user three seconds in-between pressing the buttons. A more detailed description of how this is done is below the header table, while the actual code for Keypad_driver.cpp is on page 70 and its header file Keypad_driver.h is on page 74.

	Header Files Included	Description
1	stdlib.h	Standard C library
2	rs232int.h	Header for serial port class
3	stl_timer.h	Header for timer class
4	keypad_driver.h	Header file for keypad driver

Table VIII: Header File List in Keypad_driver.cpp

Whenever the keypad_driver function check_inputs is called, the port corresponding to the first button of the correct code is checked. If it reads a low, that button has not been pressed and the function returns a '0' signifying no change in state. If it reads a high, the function enters a three second while loop where it constantly checks the port corresponding to the next correct code input. If either an incorrect button is pressed or the check times out, the wrong code count variable is incremented by one. If the wrong code count is greater than three, the function returns a '1' signifying the keypad triggered an alarm. If during the three second check, the next code entered is correct, this process until the last button for the correct code has been pressed. When this happens, the function returns a '2', signifying the keypad received the correct code and the alarm system is disabled.

Door_Sensor_driver.cpp

This file contains a simple program which reads inputs from two door sensors. The constructor sets up the two ports it will use for the door sensor inputs. The check sensor function checks both ports. If either is high, then the function returns a true which signifies the doors opening. Otherwise, this function will return false. The Door_Sensor_driver.cpp code is on page 67 while its header file Door_Sensor_driver.h is on page 69.

	Header Files Included	Description
1	stdlib.h	Standard C library
2	rs232int.h	Header for serial port class
3	stl_timer.h	Header for timer class
4	door_sensor_driver.h	Header file for door sensor driver

Table IX: Header File List in Door_Sensor_driver.cpp

Senior Project Analysis

Summary of Functional Requirements

A car alarm system with unique and less annoying siren noise than current models that would protect vehicles against window breaks and door openings and would also have wireless transmitters for Enabling and Disabling the alarm system.

Primary Constraints

All primary constraints for this project were met.

- The remote is within the size limits and it successfully transfers between the alarm system states.
- The Door Sensors are on both driver side and passenger side doors and are hidden when the doors are shut.
- The Window Sensor comprising of the combination of a microphone and the use of multiple vibration sensors successfully distinguishes broken windows from anything else.
- The external keypad is within the size constraints, it fits in the back engine compartment, and it successfully transfers alarm system state from enabled or alarming to disabled.

Economic

Costs

Component	Original Cost Estimate	Actual Cost
	All Projected Purchases	*Current Purchases*
External Keypad	$10.00	$5.70
Alarm Siren	$15.00	Using Car Horn
Microphone	$15.00	$7.95
Accelerometer	$10.00	$24.95
Bluetooth Module	$60.00	24.95
Universal Garage Remote	Not in Original Estimates	27.99
Vibration Sensor	Not in Original Estimates	17.70
Main Microcontroller	$10.00	$5.00
Horn Relay	Not in Original Estimates	$5.00
Piezo Electric Siren	Not in Original Estimates	$5.79
BJT and OP Amps	Not in Original Estimates	$5.00
Capacitors	Not in Original Estimates	$8.00
Resistors	Not in Original Estimates	$6.99
Total	$120.00	$145.02

Table X: Original Costs Estimate vs. Actual Costs

Equipment

Fortunately, the only development equipment I purchased was the Pocket AVR Programmer for $15.00. The other equipment I used included the following:

- DC Power Supply constructed from IME 156
- Agilent E3630 Power Supply
- HP 3478A Multi-meter
- Agilent DSO6012A Oscilloscope
- Toshiba Laptop
- Soldering Iron

Time Line

1. I originally estimated the project to take two quarters to complete spanning 1/14/2011 to 4/25/2011

2. I did another PERT chart and time estimate in-between Winter and Spring quarter 2011. This estimate also predicted the project to be completed in two quarters, spanning 1/12/2011 to 5/12/2011. This second estimation was flawed due to the intensity of my Spring quarter class schedule. I then thought I would work on this project during my summer vacation, but then I was hired at Hewlett Packard for a six month internship. I neglected to work on this project while working full time, and returned to school in January 2012.

3. My final PERT chart shows the actual time line of the entire project. Although the project took three quarters and over fifteen months to complete, it compiled of only 83 work days.

Bill of Materials

9V to 5V Voltage Regulator			**Page 8 and 16**
Component	**Supplier**	**Quantity**	**Price per Unit**
LM7805 Voltage Regulator	RadioShack	2	$2.00
100uF Capacitor	RadioShack	4	$1.50

Table XI: Voltage Regulator Bill of Materials

Microphone and Filter			**Page 15**
Component	**Supplier**	**Quantity**	**Price per Unit**
LM741 OpAmp	RadioShack	2	$0.50
10nF Capacitor	RadioShack	4	$0.10
Spark fun BOB-09964 Microphone	Sparkfun	1	$7.95

Table XII: Microphone and Filter Bill of Materials

Vibration Sensors and Summing Amplifier			Page 19
Component	**Supplier**	**Quantity**	**Price per Unit**
LM741 OpAmp	RadioShack	1	$0.50
SEN-09196 Piezo Electric Vibration Sensors	Sparkfun	6	$2.95

Table XIII: Inverting Summing Amplifier Bill of Materials

Alarm Noise			Page 17
Component	**Supplier**	**Quantity**	**Price per Unit**
12V/40A SPDT Relay	RadioShack	1	$5.00
TIP120 NPN BJT	RadioShack	1	$1.50
273-079 102dB Piezo Electric Siren	RadioShack	1	$5.79

Table XIV: Alarm Noise Bill of Materials

Microcontroller			Page 17
Component	**Supplier**	**Quantity**	**Price per Unit**
Atmega32 Microcontroller	Sparkfun	1	$5.00
16MHz Crystal COM-00535	Sparkfun	1	$0.95
Ceramic 16pF Capacitor	Sparkfun	2	$0.25
276-149A Proto Board	RadioShack	1	$2.95

Table XV: Microcontoller Bill of Materials

Remote			Page 17
Component	**Supplier**	**Quantity**	**Price per Unit**
Skylink g6vr RF Transmitter/Receiver	Frys Electronics	1	$27.99

Table XVI: Remote Bill of Materials

Keypad			Page 17
Component	**Supplier**	**Quantity**	**Price per Unit**
Sparkfun COM-08653 12 Button Keypad	Santa Maria Electronics SuperMart	1	$5.70
Multi-Mac 1-Gang Receptacle Cover/Enclosure	HomeDepot	1	$7.37

Table XVII: Keypad Bill of Materials

Environmental

Noise pollution is growing problem in cities. Every night, the absurd sound of car alarms can be heard miles from the car. It is a sound that is simply associated with city life. That is why my car alarm system will need to incorporate an alarm that deters car thieves, but limits noise pollution. Fortunately, my car already has a horn that is quieter than most. By sending the horn a pulse signal for a unique sound with a smaller magnitude to lessen the volume, an adequate alarm siren may be achieved while reducing noise pollution. If I took this project to a commercial basis, I would look to making a unique sounding alarm that said phrases such as, "Step Away From the Vehicle". This would not only be a great alarm because it would show that this is no ordinary car alarm system, but would also be great for noise pollution. People hearing this alarm would pay more attention to it and would not be annoyed by a siren.

Manufacturability

As with any product, a major issue with manufacturing on a commercial basis is entering an already established market. It is very difficult to create a product that stands out from the crowd and has a competitive price. Matching the competitors' prices cuts into profits, this makes mass manufacturing less plausible. After completing the cost analysis for the car alarm system designed in this project, I do not believe this is viable product to manufacture on a large scale.

Manufacturing Estimates	Estimates
Number of Devices sold per year	1000
Manufacturing cost for each device	
Parts Costs	$50.00
Manufacturing Costs	$5.00
Installation Costs	$50.00
Marketing Costs	
Purchase Price for Each Device	$150.00
User Operation Costs	$5.00
Total Profits per year	$45,000.00

Table XVIII: Commercial Basis Manufacturing Figures

Based solely on the costs of the components, this project seems very feasible on a commercial basis. However, the installation of all the sensors would require trained human hands. A great deal of effort would need to be spent making this product easily installed by owners before it will ever see a store shelf.

Sustainability

A car alarm system will only use natural resources that are already being used in electronics and all of which are considered safe. It also does not improve or harm any materials. The main effect the car alarm system will have on the environment will be the disposal of the batteries. "Bell Canada, a leading telecom services provider, in its Centralized Collection Programme, collected 960 metric tons of wet cell lead acid batteries for recycling in 2003" [Frost]. The current car alarm system design includes low power consumption and will prove insignificant in battery drain.

The only maintenance for the car owner would be to occasionally replace the battery in the remote transmitter as well as the batteries for the microcontroller. However, both the microcontroller and the remote have very little power consumption. The battery will have a very long life span and will only need to be replaced on a yearly basis.

The main beneficiary of my car alarm system is the owner. The price will be in the range of a few hundred dollars, the potential savings are in the thousands. It will enable owners to sleep knowing their vehicle is safe. The cost of a car theft is a financial burden not many people can afford. It is national problem that affects everyone. "Using the FBI's average valuation of $6,505 per stolen vehicle, the 794,616 vehicles stolen during 2009 caused estimated property losses of $5.2 billion" [RMIIA]. All of the parts in my costs estimate total one hundred dollars. This makes the designed car alarm system a worthy investment for any car owner.

Ethical and Health

Car alarms create a conflict between car owners and their neighbors as stated in the Health and Environmental sections. Owners want to ensure the security of their car, but neighbors don't want to listen to constant alarms. Since my project is based on car security, does is make it ok for me to disturb the neighborhood? Is one of these needs greater than the other? This conflict has made one of my project priorities to ensure that a car alarm is successful in protecting the car while keeping noise to minimum.

The primary goal of protecting the car owner has definitely been achieved. The alarm system protects against window breaks, door openings, and starting the car. I believe that it also achieves the second goal of keeping noise to a minimum. The horn is only programmed to go for a few minutes, deciding to use the car's horn keep's the volume down, and sending a pulse signal creates a slightly less irritating siren. I believe that these reasons make this project succeed on both fronts.

The Health issue with car alarms is also negated by the same reasons stated above. A problem with car alarm systems is how the noise created is actually causing harm to people. The World Health Organization, National Institutes of Health, and numerous other scientific and

medical publications are recognizing noise pollution and its deleterious effects [CDC]. The noises from alarm systems trigger an involuntary stress response with an increase in heart rate, blood pressure and muscle tension. Having the quieter and shorter alarm helps to solve this health issue.

Safety

Webster Dictionary defines Safety as the condition of being safe from undergoing or causing hurt, injury, or loss. A car alarm system may not prevent injuries, but it certainly provides immediate protection against loss of property by preventing car thefts. However, a car alarm system also provides indirect protection to a neighborhood.

Car alarms deter criminals from breaking and stealing automobiles. If criminals know certain neighborhoods have car alarm systems, they will avoid those neighborhoods. These neighborhoods will become free of car thefts without actually sounding any car alarms, making these areas that much safer.

Social and Political

In 2010, $4.5 billion was lost to motor vehicle thefts in 2010 totally 737,142 motor vehicles were stolen in the United States [VinTrack]. That averages roughly $6,000 dollars per stolen vehicle. Almost three quarters of million people are negatively affected by car thefts. In New York City, there were over 65,000 noise complaints made to the police in 2009, with an estimated 70% of them due to car alarms. In 2010, New York City had an estimated population of 8.2 million and there were 10 cities in the US with populations over 1 million. If we multiply the number of noise complaints due to car alarms in New York City by the number of cities with comparable populations, we see that there were 455,000 total noise complaints. Although this is a very rough estimation, it does show that for every noise complaint made in 2010, 1.6 cars were stolen. This fact combined with the fact that sleep deprivation due to excessive car alarm noise is incomparable to having your car stolen shows that car alarms provide a greater good for the general population. Even so, there is a considerable effort being made to ban car alarms in rural areas.

Legislators in New York City tried to pass a bill to outlaw noisy car alarms, because of the environmental and health impacts [Friedman]. My project therefore must entail the possibility of ban on noisy alarm systems. My security system must either have a quieter alarm, a less annoying noise, or some other way of alarming intruders.

Since my alarm system includes a quieter and shorter alarm siren, it will be allowed even with this new bill. The only opponent for this bill would be the impracticality of alarm systems meeting this standard. If my alarm system can provide the same security as other alarm systems while meeting this bill's standards, it would have a huge selling edge over other alarm systems.

Bibliography

- "Auto Theft Statistics" **Rocky Mountain Insurance Information Association**. 24 February 2012 http://www.rmiia.org/auto/auto_theft/statistics.asp

- "Auto Theft Statistics" **Vin Track: Stolen Vehicle Detection** Bender Enterprises Inc. 3 March 2012 http://www.vintrack.com/moststolenvehicles.htm

- Friedman, Aaron. "The Case Against Car Alarms" **Gotham Gazette**. 7 July 2003 http://www.gothamgazette.com/print/445

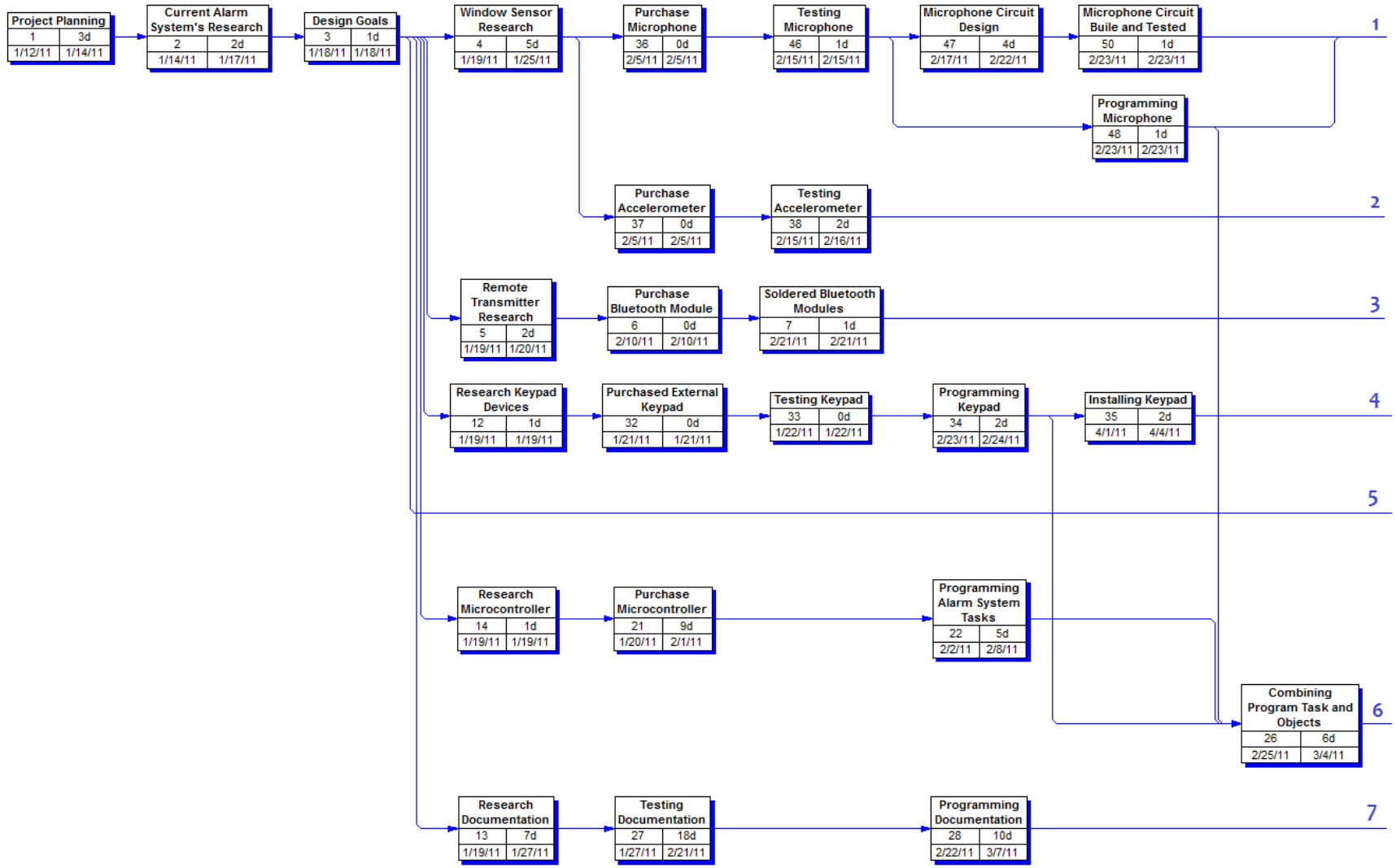
- Harrison, Adelia Honeywood. "City Turns Deaf Ear to Noisy Neighbors" **Gotham Gazette**. February 2010. http://www.gothamgazette.com/article/Environment/20100223/7/3191

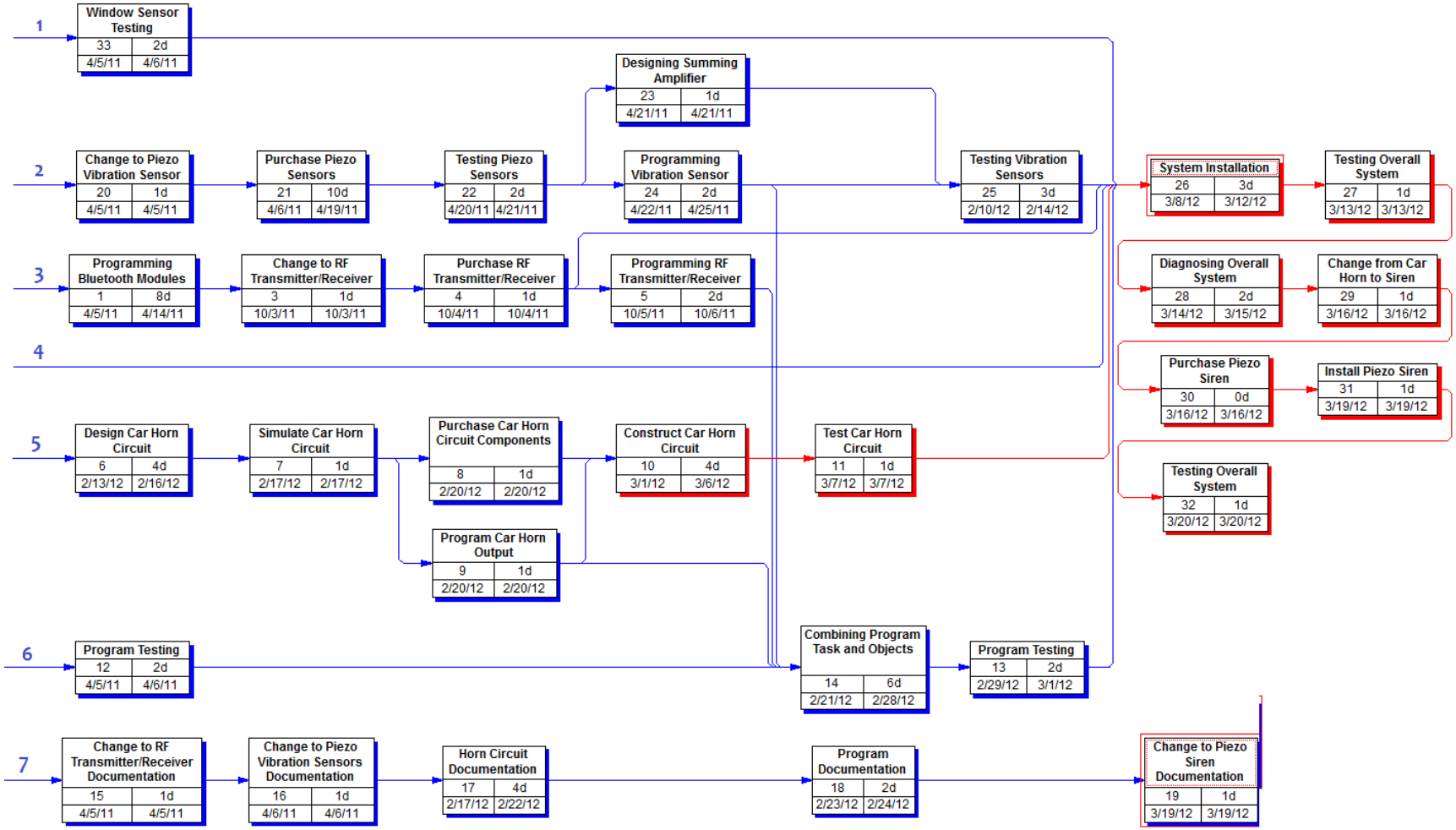
- "Health Studies: Noise" **Center of Disease Control and Prevention** US Government Department of Health and Services. 24 February 2012 http://www.cdc.gov/nceh/hsb/noise/faq.htm

- Malavika, C.R. "Environmental Effects Associated with Battery Disposal." **Frost and Sullivan** 28 Jun 2004 http://www.frost.com/prod/servlet/market-insight-top.pag?docid=20759887

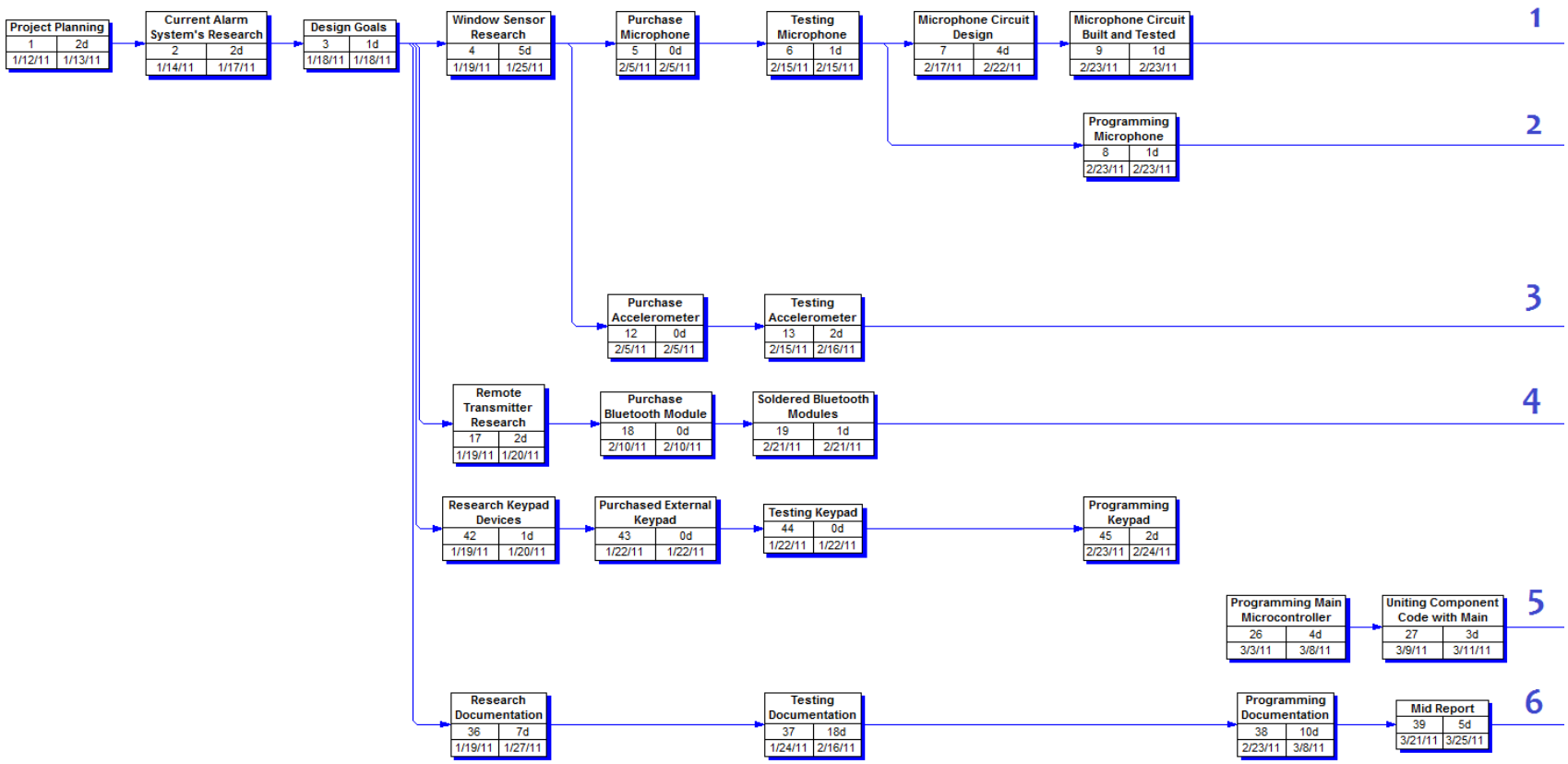
- "safety." *Merriam-Webster.com*. Merriam-Webster, 2012. 18 March 2011 http://www.merriam-webster.com/dictionary/safety

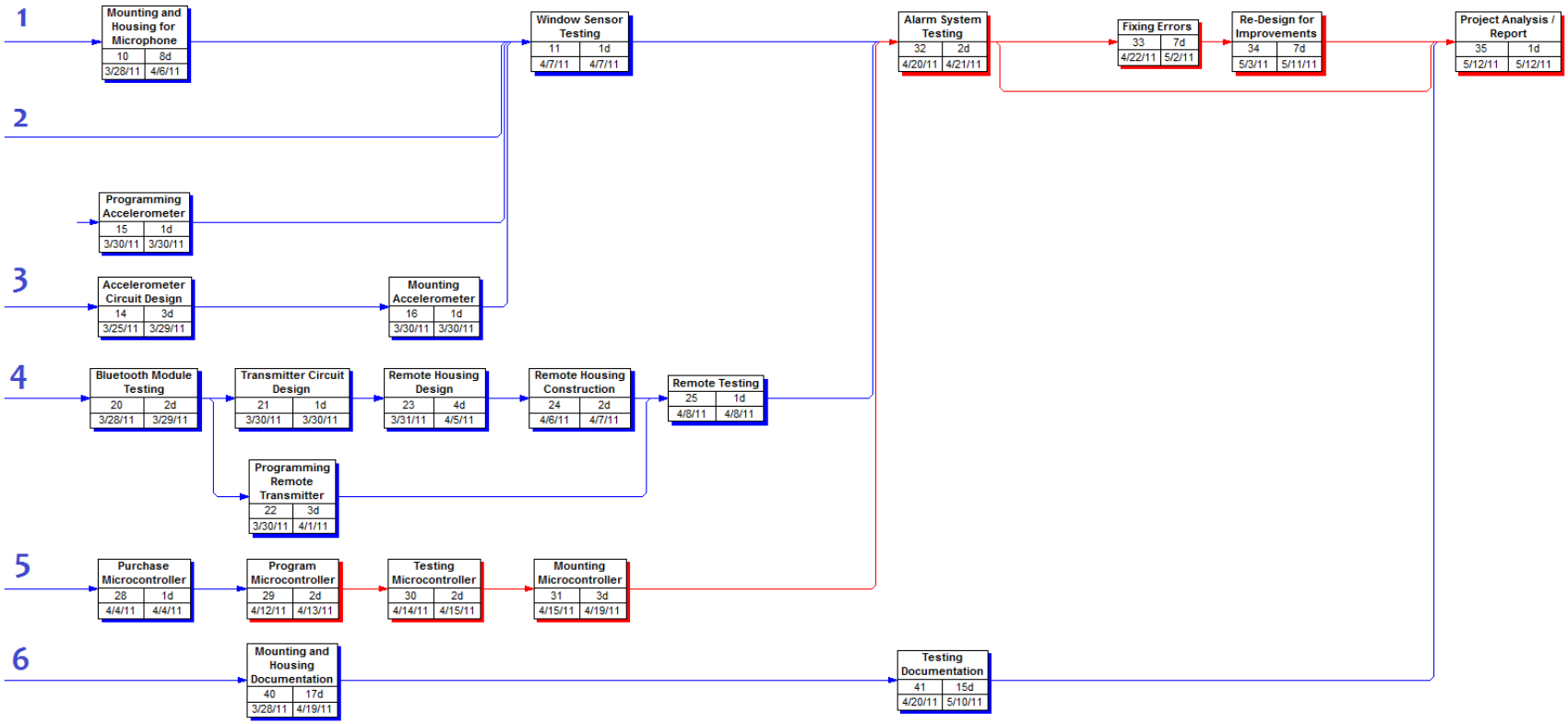
Final PERT Chart



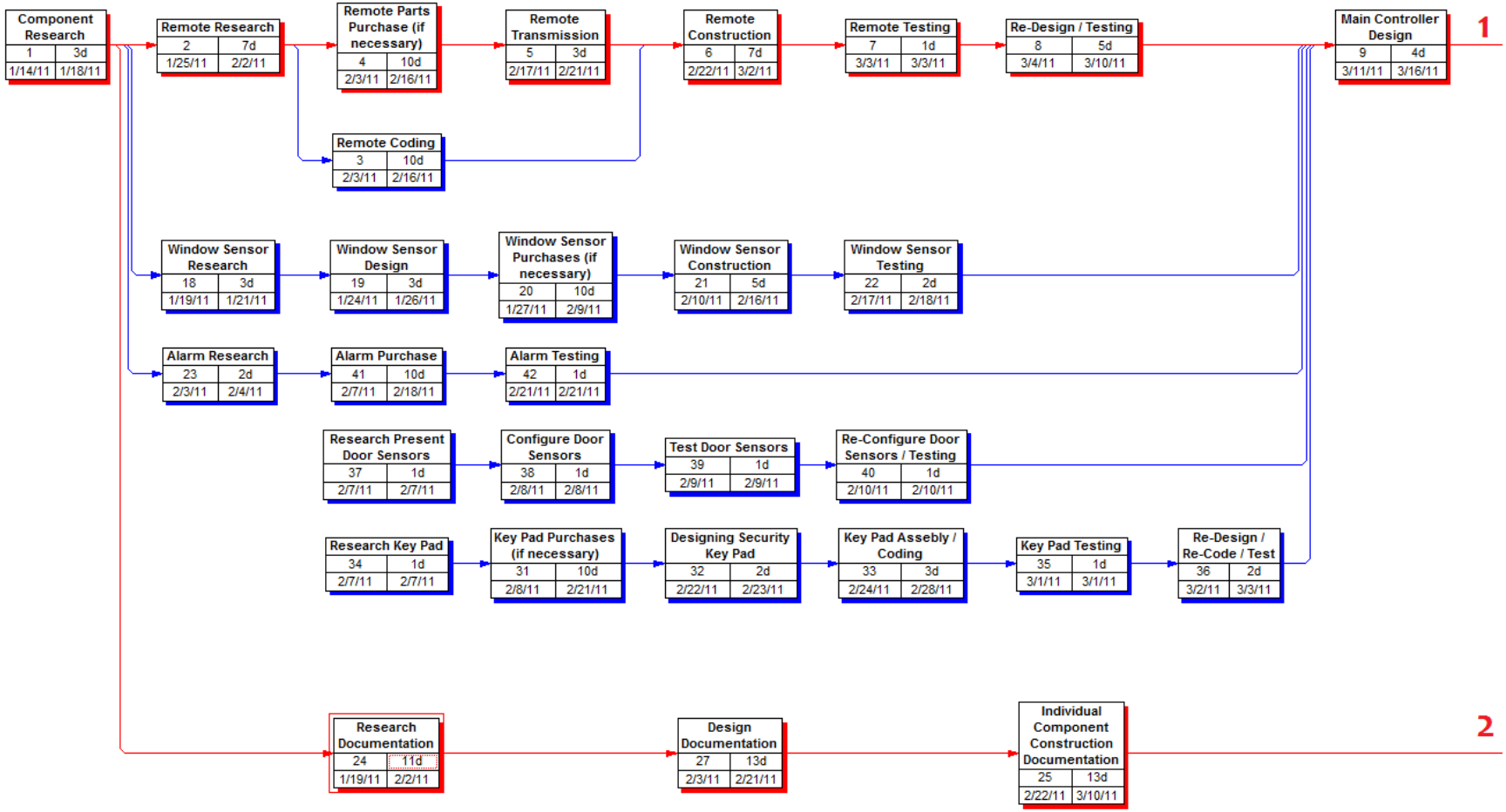

1
Window Sensor Testing
33 2d
4/5/11 4/6/11
2
Change to Piezo Vibration Sensor
20 1d
4/5/11 4/5/11
Purchase Piezo Sensors
21 10d
4/6/11 4/19/11
Testing Piezo Sensors
22 2d
4/20/11 4/21/11
Designing Summing Amplifier
23 1d
4/21/11 4/21/11
Programming Vibration Sensor
24 2d
4/22/11 4/25/11
Testing Vibration Sensors
25 3d
2/10/12 2/14/12
3
Programming Bluetooth Modules
1 8d
4/5/11 4/14/11
Change to RF Transmitter/Receiver
3 1d
10/3/11 10/3/11
Purchase RF Transmitter/Receiver
4 1d
10/4/11 10/4/11
Programming RF Transmitter/Receiver
5 2d
10/5/11 10/6/11
4
5
Design Car Horn Circuit
6 4d
2/13/12 2/16/12
Simulate Car Horn Circuit
7 1d
2/17/12 2/17/12
Purchase Car Horn Circuit Components
8 1d
2/20/12 2/20/12
Program Car Horn Output
9 1d
2/20/12 2/20/12
Construct Car Horn Circuit
10 4d
3/1/12 3/6/12
Test Car Horn Circuit
11 1d
3/7/12 3/7/12
6
Program Testing
12 2d
4/5/11 4/6/11
Combining Program Task and Objects
14 6d
2/21/12 2/28/12
Program Testing
13 2d
2/29/12 3/1/12
7
Change to RF Transmitter/Receiver Documentation
15 1d
4/5/11 4/5/11
Change to Piezo Vibration Sensors Documentation
16 1d
4/6/11 4/6/11
Horn Circuit Documentation
17 4d
2/17/12 2/22/12
Program Documentation
18 2d
2/23/12 2/24/12
Change to Piezo Siren Documentation
19 1d
3/19/12 3/19/12
System Installation
26 3d
3/8/12 3/12/12
Testing Overall System
27 1d
3/13/12 3/13/12
Diagnosing Overall System
28 2d
3/14/12 3/15/12
Change from Car Horn to Siren
29 1d
3/16/12 3/16/12
Purchase Piezo Siren
30 0d
3/16/12 3/16/12
Install Piezo Siren
31 1d
3/19/12 3/19/12
Testing Overall System
32 1d
3/20/12 3/20/12

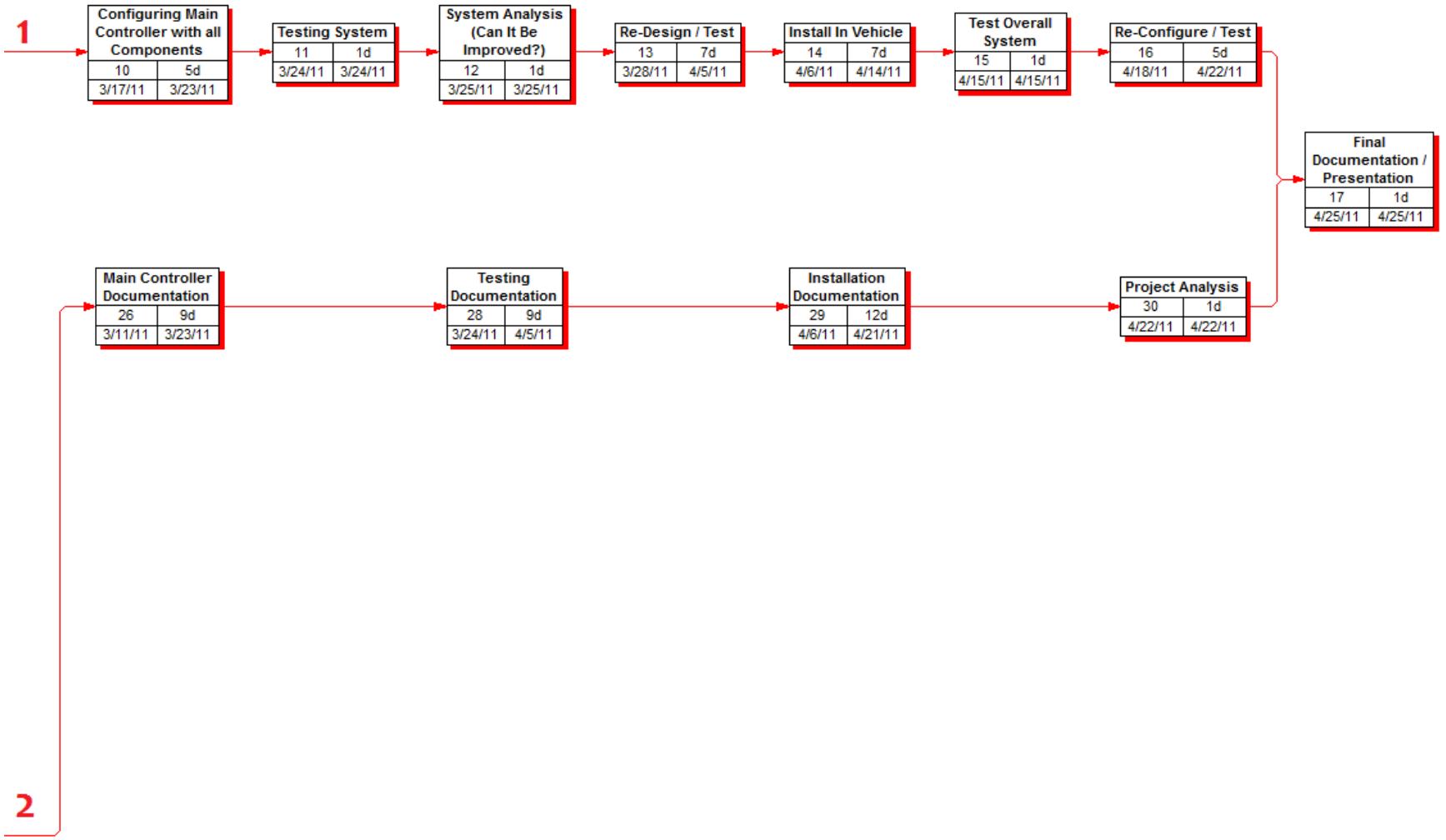
Mid-Point PERT Chart




1
Mounting and Housing for Microphone
10 | 8d
3/28/11 | 4/6/11
Window Sensor Testing
11 | 1d
4/7/11 | 4/7/11
Alarm System Testing
32 | 2d
4/20/11 | 4/21/11
Fixing Errors
33 | 7d
4/22/11 | 5/2/11
Re-Design for Improvements
34 | 7d
5/3/11 | 5/11/11
Project Analysis / Report
35 | 1d
5/12/11 | 5/12/11
2
Programming Accelerometer
15 | 1d
3/30/11 | 3/30/11
3
Accelerometer Circuit Design
14 | 3d
3/25/11 | 3/29/11
Mounting Accelerometer
16 | 1d
3/30/11 | 3/30/11
4
Bluetooth Module Testing
20 | 2d
3/28/11 | 3/29/11
Transmitter Circuit Design
21 | 1d
3/30/11 | 3/30/11
Remote Housing Design
23 | 4d
3/31/11 | 4/5/11
Remote Housing Construction
24 | 2d
4/6/11 | 4/7/11
Remote Testing
25 | 1d
4/8/11 | 4/8/11
Programming Remote Transmitter
22 | 3d
3/30/11 | 4/1/11
5
Purchase Microcontroller
28 | 1d
4/4/11 | 4/4/11
Program Microcontroller
29 | 2d
4/12/11 | 4/13/11
Testing Microcontroller
30 | 2d
4/14/11 | 4/15/11
Mounting Microcontroller
31 | 3d
4/15/11 | 4/19/11
6
Mounting and Housing Documentation
40 | 17d
3/28/11 | 4/19/11
Testing Documentation
41 | 15d
4/20/11 | 5/10/11

Original PERT Chart




1
Configuring Main Controller with all Components
10
5d
3/17/11
3/23/11
Testing System
11
1d
3/24/11
3/24/11
System Analysis (Can It Be Improved?)
12
1d
3/25/11
3/25/11
Re-Design / Test
13
7d
3/28/11
4/5/11
Install In Vehicle
14
7d
4/6/11
4/14/11
Test Overall System
15
1d
4/15/11
4/15/11
Re-Configure / Test
16
5d
4/18/11
4/22/11
Final Documentation / Presentation
17
1d
4/25/11
4/25/11
Main Controller Documentation
26
9d
3/11/11
3/23/11
Testing Documentation
28
9d
3/24/11
4/5/11
Installation Documentation
29
12d
4/6/11
4/21/11
Project Analysis
30
1d
4/22/11
4/22/11
2

Code

Makefile

```
#---------------------------------------------------------------------------
# File:    Makefile for an AVR project
#          The makefile is the standard way to control the compilation and linking of
#          C/C++ files into an executable file. This makefile is also used to control
#          the downloading of the executable file to the target processor and the
#          generation of documentation for the project.
#
# Version:  4-11-2004 JRR Original file
#           6-19-2006 JRR Modified to use AVR-JTAG-ICE for debugging
#          11-21-2008 JRR Added memory locations and removed extras for bootloader
#          11-26-2008 JRR Cleaned up; changed method of choosing programming method
#          11-14-2009 JRR Added make support to put library files into subdirectory
#
# Relies   The avr-gcc compiler and avr-libc library
# on:      The avrdude downloader, if downloading through an ISP port
#          AVR-Insight or DDD and avarice, if debugging with the JTAG port
#          Doxygen, for automatic documentation generation
#
# Copyright 2006-2009 by JR Ridgely.  This makefile is intended for use in educational
# courses only, but its use is not restricted thereto. It is released under the terms
# of the Lesser GNU Public License with no warranty whatsoever, not even an implied
# warranty of merchantability or fitness for any particular purpose. Anyone who uses
# this file agrees to take all responsibility for any and all consequences of that use.
#---------------------------------------------------------------------------

# The name of the program you're building, and the list of object files. There must be
# an object file listed here for each *.c or *.cc file in your project. The make
# program will automatically figure out how to compile and link your C or C++ files
# from the list of object files. TARGET will be the name of the downloadable program.
TARGET = alarm_system_test
OBJS = $(TARGET).o alarm_task.o keypad_driver.o door_sensor_driver.o

# Clock frequency of the CPU, in Hz. This number should be an unsigned long integer.
```

```
# For example, 16 MHz would be represented as 16000000UL. For ME405 boards, clocks are
# usually 4MHz for Ministrone and Swoop sensor boards, 8MHz for single-motor ME405
# boards, and 16MHz for dual-motor ME405 boards
F_CPU = 8000000UL

# These codes are used to switch on debugging modes if they're being used. Several can
# be placed on the same line together to activate multiple debugging tricks at once.
# -DSERIAL_DEBUG       For general debugging through a serial device
# -DSTL_TRACE          For printing state transition traces on a serial device
# -DSTL_PROFILE        For doing profiling, measurement of how long tasks take to run
DBG =

# Other codes, used for turning on special features in a given project, can be put here
OTHERS =

# This define is used to choose the type of programmer from the following options:
# bsd        - Parallel port in-system (ISP) programmer using SPI interface on AVR
# jtagice    - Serial or USB interface JTAG-ICE mk I clone from ETT or Olimex
# bootloader - Resident program in the AVR which downloads through USB/serial port
# PROG = bsd
# PROG = jtagice
# PROG = bootloader
PROG = usbtiny

# These defines specify the ports to which the downloader device is connected.
# PPORT is for "bsd" on a parallel port, lpt1 on Windows or /dev/parport0 on Linux.
# JPORT is for "jtagice" on a serial port such as com1 or /dev/ttyS0, or usb-serial
#       such as com4 or /dev/ttyUSB1, or aliased serial port such as /dev/avrjtag
# BPORT is for "bootloader", the USB/serial port program downloader on the AVR
# The usbtiny programmer doesn't need a port specification; it has a USB identifier
PPORT = /dev/parport0
JPORT = /dev/ttyUSB1
BPORT = /dev/ttyUSB0

# These are the name of the library file and of the subdirectory in which it is kept
LIB_FILE = me405.a
LIB_DIR = lib
```

```
#-------------------------------------------------------------------------------
# This section specifies the type of CPU; uncomment one line for your processor. To add
# a new chip to the file, put its designation here and also define a code for CHIP in
# the section below. See the avrdude manual page for the code to use as CHIP.

# MCU = atmega128
MCU = atmega32
# MCU = atmega324p
# MCU = atmega644
# MCU = atmega8
# MCU = at90s2313
# MCU = atmega8535

################# End of the stuff the user is expected to need to change ###############
#-------------------------------------------------------------------------------
# In this section, various variables are automatically set to match the MCU choice
# above. The CHIP variable is used by the downloader and must match MCU; fuse bytes are
# set some common settings for each processor and sometimes need to be adjusted.
# New chip specifications can be added to this file as needed.

# ATmega128 set up for ME405 board with JTAG enabled
ifeq ($(MCU), atmega128)
  CHIP = m128
  EFUSE = 0xFF
  HFUSE = 0x11
  LFUSE = 0xEF
# ATmega32 configured for Swoop sensor board with JTAG disabled to save power
else ifeq ($(MCU), atmega32)
  CHIP = m32
  EFUSE =
  HFUSE = 0xD9
  LFUSE = 0xEF
# ATmega324P configured for Swoop sensor board with JTAG disabled
# Standard fuses FF19EF, bootloader fuses FFC8EF, low power fuses FF11EF
else ifeq ($(MCU), atmega324p)
  CHIP = m324p
  EFUSE = 0xFF
  HFUSE = 0x11
```

```
  LFUSE = 0xEF
# ATmega644 (note: the 644P needs a different MCU)
else ifeq ($(MCU), atmega644)
  CHIP = m644
  EFUSE = 0xFF
  HFUSE = 0x11
  LFUSE = 0xEF
# ATmega8 configured for Ministrone strain gauge interface boards
else ifeq ($(MCU), atmega8)
  CHIP = m8
# ATmega2313 configuration in case somebody has a few old ones lying around
else ifeq ($(MCU), at90s2313)
  CHIP = 2313
# ATmega8535 configuration for use with old chips gathering dust
else ifeq ($(MCU), atmega8535)
  CHIP = m8535
endif

#----------------------------------------------------------------------------
# Tell the compiler how hard to try to optimize the code. Optimization levels are:
# -O0  Don't try to optimize anything (even leaves empty delay loops in)
# -O1  Some optimizations; code usually smaller and faster than O0
# -O2  Pretty high level of optimization; often good compromise of speed and size
# -O3  Tries really hard to make code run fast, even if code size gets pretty big
# -Os  Tries to make code size small. Sometimes -O1 makes it smaller, though(?!)
OPTIM = -O2

# Define which compiler to use (CC) and some standard compiler options (STD).
CC = avr-gcc
STD = _GNU_SOURCE

# Usually this is just left at debugging level 0
DEBUGL = DEBUG_LEVEL=0

# The JTAG bitrate and IP port are used to configure the JTAG-ICE debugger
JTAGBITRATE = 1000000
IPPORT = :4242
```

```
# Any other compiler switches go here, for example short enumerations to save memory
OTHERS += -fshort-enums

#---------------------------------------------------------------------------
# This command exports the definitions of variables in this file so that Makefiles in
# subdirectories can make use of these settings. If you add variables above, they may
# need to be added here. Exception: stuff for the downloader is only used in this file.
export CC STD DEBUGL OPTIM MCU F_CPU DBG OTHERS
#---------------------------------------------------------------------------
# Inference rules show how to process each kind of file.

# How to compile a .c file into a .o file
.c.o:
        $(CC) -c -g $(OPTIM) -mmcu=$(MCU) -I ./$(LIB_DIR) \
        -DF_CPU=$(F_CPU) $(DBG) $(OTHERS) $<

# How to compile a .cc file into a .o file
.cc.o:
        $(CC) -c -g $(OPTIM) -mmcu=$(MCU) -I ./$(LIB_DIR) \
        -DF_CPU=$(F_CPU) $(DBG) $(OTHERS) $<

# How to compile a .cpp file into a .o file
.cpp.o:
        $(CC) -c -g $(OPTIM) -mmcu=$(MCU) -I ./$(LIB_DIR) \
        -DF_CPU=$(F_CPU) $(DBG) $(OTHERS) $<

# This rule controls the linking of the target program from object files. The target
# is saved as an ELF debuggable binary, downloadable .hex, and raw binary .bin.  Also
# created is a listing file .lst, which shows generated machine and assembly code.
$(TARGET).elf:  $(OBJS)
        $(CC) $(OBJS) $(LIB_FILE) -g -mmcu=$(MCU) -o $(TARGET).elf
        @avr-objdump -h -S $(TARGET).elf > $(TARGET).lst
        @avr-objcopy -j .text -j .data -O ihex $(TARGET).elf $(TARGET).hex
        @avr-objcopy -j .text -j .data -O binary $(TARGET).elf $(TARGET).bin
        @avr-size $(TARGET).elf

# This rule makes the files in the library subdirectory. It is normally called when
# one is first building stuff after downloading the project tree, or after updating
```

```
# library files
$(LIB_FILE):
	$(MAKE) -C $(LIB_DIR)
	mv -f $(LIB_DIR)/me405.a .

#--------------------------------------------------------------------------------
# Make the main file of this project.  This target is invoked when the user just types
# 'make' as opposed to 'make <target>.'  This should be the first target in Makefile.

all:  $(LIB_FILE) $(TARGET).elf

#--------------------------------------------------------------------------------
#--------------------------------------------------------------------------------
# This is a dummy target that doesn't do anything. It's included because the author
# belongs to a faculty labor union and has been instilled with reverence for laziness.
nothing:
#--------------------------------------------------------------------------------
# 'make install' will make the project, then download the program using whichever
# method has been selected -- ISP cable, JTAG-ICE module, or USB/serial bootloader

install:  $(LIB_FILE) $(TARGET).elf
 ifeq ($(PROG), bsd)
	avrdude -p $(CHIP) -P $(PPORT) -c bsd -V -E noreset -Uflash:w:$(TARGET).hex
 else ifeq ($(PROG), jtagice)
	avarice -e -p -f $(TARGET).elf -j $(JPORT)
 else ifeq ($(PROG), bootloader)
	ruby bootloader.rb $(BPORT) $(TARGET).bin
 else ifeq ($(PROG), usbtiny)
	avrdude -p $(CHIP) -c usbtiny -V -E noreset -Uflash:w:$(TARGET).hex
 else
	@echo "ERROR: No programmer" $(PROG) "in the Makefile"
 endif

#--------------------------------------------------------------------------------
# 'make debug' will make the project, then download it with the JTAG-ICE-I interface
# and start up a debugger. The JTAG-ICE-I only works with the ATmega32, ATmega128, and
# a few other somewhat older processors; it won't work with the ATmega324/644.
```

```
debug:  $(TARGET).elf $(GDBINITFILE)
	@avarice --capture -B $(JTAGBITRATE) -j $(JPORT) $(IPPORT) &
	@sleep 1
	@ddd --debugger "avr-gdb -x $(GDBINITFILE)"
	@killall -q avarice || /bin/true

$(GDBINITFILE): $(TARGET).hex
	@echo "file $(TARGET).elf"          > $(GDBINITFILE)
	@echo "target remote localhost:4242" >> $(GDBINITFILE)
	@echo "load"                        >> $(GDBINITFILE)
	@echo "break main"                  >> $(GDBINITFILE)
	@echo "continue"                    >> $(GDBINITFILE)

#----------------------------------------------------------------------------
# 'make term' will run the serial port emulator GTK-Term, a hacked version of which
# can be used to download programs to the bootloader with the ctrl-D command.

term:  $(TARGET).elf
	@gtkterm --to-send $(TARGET).bin &

#----------------------------------------------------------------------------
# 'make fuses' will set up the processor's fuse bits in a "standard" mode. Standard is
# a setup in which there is no bootloader but the ISP and JTAG interfaces are enabled.

fuses: nothing
 ifeq ($(PROG), bsd)
	avrdude -p $(CHIP) -P $(PPORT) -c $(PROG) -V -E noreset -Ulfuse:w:$(LFUSE):m
	avrdude -p $(CHIP) -P $(PPORT) -c $(PROG) -V -E noreset -Uhfuse:w:$(HFUSE):m
	avrdude -p $(CHIP) -P $(PPORT) -c $(PROG) -V -E noreset -Uefuse:w:$(EFUSE):m
 else ifeq ($(PROG), jtagice)
	avarice -e -j $(JPORT) --write-fuses EF19EF
	# FF11EF
 else ifeq ($(PROG), usbtiny)
	avrdude -p $(CHIP) -c usbtiny -q -V -E noreset -Ulfuse:w:$(LFUSE):m
	avrdude -p $(CHIP) -c usbtiny -q -V -E noreset -Uhfuse:w:$(HFUSE):m
	avrdude -p $(CHIP) -c usbtiny -q -V -E noreset -Uefuse:w:$(EFUSE):m
 else
	@echo "ERROR: Only bsd/ISP, JTAG-ICE, or USBtiny can program fuse bytes"
```

```
 endif

#--------------------------------------------------------------------------------
# 'make readfuses' will see what the fuses currently hold

readfuses: nothing
 ifeq ($(PROG), bsd)
        @echo "ERROR: Not yet programmed to read fuses with bsd/ISP cable"
 else ifeq ($(PROG), jtagice)
        @avarice -e -j $(JPORT) --read-fuses
 else ifeq ($(PROG), bootloader)
        @echo "ERROR: Not yet programmed to read fuses via bootloader"
 else
        @echo "ERROR: No known device specified to read fuses"
 endif

#--------------------------------------------------------------------------------
# 'make reset' will read a byte of lock bits, ignore it, and reset the chip
# This can be used to un-freeze the chip after a 'make freeze'
reset:
 ifeq ($(PROG), bsd)
        avrdude -c bsd -p $(CHIP) -P $(PPORT) -c $(PROG) -V -E noreset \
                -Ulfuse:r:/dev/null:r
 else ifeq ($(PROG), usbtiny)
        avrdude -p $(CHIP) -c usbtiny -q -V -E noreset -Ulfuse:r:/dev/null:r
 else
        @echo "ERROR: make reset only works with parallel ISP cable"
 endif

#--------------------------------------------------------------------------------
# 'make freeze' will read a byte of lock bits and leave the chip in reset
freeze:
 ifeq ($(PROG), bsd)
        avrdude -c bsd -p $(CHIP) -P $(PPORT) -c $(PROG) -V -E reset \
                -Ulfuse:r:/dev/null:r
#   else ifeq ($(PROG), usbtiny)
#        avrdude -p $(CHIP) -c usbtiny -q -V -E reset -Ulfuse:r:/dev/null:r
 else
```

```
		@echo "ERROR: make freeze only works with parallel ISP cable"
 endif

#-------------------------------------------------------------------------------
# 'make doc' will use Doxygen to create documentation for the project.

doc:  $(TARGET).elf
		doxygen doxygen.conf

#-------------------------------------------------------------------------------
# 'make clean' will erase the compiled files, listing files, etc. so you can restart
# the building process from a clean slate. It's also useful before committing files to
# a CVS or SVN or Git repository.
clean:
		rm -f *.o $(TARGET).hex $(TARGET).lst $(TARGET).elf $(TARGET).bin *~

#-------------------------------------------------------------------------------
# 'make cleaner' will clean the current directory and subdirectories as well. This
# cleanup should be performed before archiving things in an SVN/CVS/Git repository.
cleaner:  clean
		$(MAKE) -C $(LIB_DIR) clean
		rm -f $(LIB_FILE)
		rm -fr html rtf latex

#-------------------------------------------------------------------------------
# 'make help' will show a list of things this makefile can do
help:
		@echo 'make         - Build program file ready to download'
		@echo 'make install  - Build program and download with parallel ISP cable'
		@echo 'make debug    - Build program and debug using JTAG-ICE module'
		@echo 'make freeze   - Stop processor with parallel cable RESET line'
		@echo 'make reset    - Reset processor with parallel cable RESET line'
		@echo 'make doc      - Generate documentation with Doxygen'
		@echo 'make clean    - Remove compiled files in topmost directory'
		@echo 'make cleaner  - Clean subdirectories too; use before archiving files'
		@echo ' '
		@echo 'Notes: 1. Other less commonly used targets are in the Makefile'
		@echo '       2. You can combine targets, as in "make clean all"'
```

Alarm_System_Test.cpp

```
//=============================================================================================
/** \file alarm_system_test.cpp
 *     This file contains a test program which creates objects and tasks for the Car
 *                      Alarm System and calls the tasks within main.
 *
 *  Revisions:
 *    \li  02-23-11  Created main_test file from motor_test from ME405
 *    \li  03-03-11  Edited main_test file to create objects and tasks
 */
//=============================================================================================
#include <stdlib.h>                                                     // Standard C library
#include "rs232int.h"                                                   // Include header for serial port class
#include <avr/io.h>                                                     // You'll need this for SFR and bit names
#include "stl_timer.h"                                                  // Microsecond-resolution (approx.) timer#include "stl_task.h"
#include "keypad_driver.h"                                              // Include header for keypad_driver
#include "door_sensor_driver.h"                                         // Include header for keypad_driver
#include "alarm_task.h"                                                 // Include header for alarm_task
//---------------------------------------------------------------------------------------------

int main ()
{
        // Create a serial port object
        rs232 the_serial_port (9600, 1);

        // Create a time stamp object. This is a data structure which holds measured times
        time_stamp the_time;
        // Create a task timer object.
        task_timer the_timer;
        // Create a keypad object. This object must be given a
        // pointer to the serial port object so that it can print debugging information
        keypad_driver keypad (&the_serial_port);
        door_sensor_driver door_sensor (&the_serial_port);
        // Creates a alarm task.  This needs the task timer and time stamp for the schedule
        // function.  It also needs the keypad driver and the serial port for
        // debugging information
```

```
	alarm_task my_alarm_task (the_timer, the_time, &keypad, &door_sensor, &the_serial_port);
	// Enable interrupts. This is necessary because the timer uses a timer interrupt
	sei ();
	// Enters Infinite loop
	while(true)
	{
		// use schedule() to run tasks
		my_alarm_task.schedule();
	}
return (0);
}
```

Alarm_task.cpp

```
//*****************************************************************************
/** \file alarm_task.cpp
 *    This file contains case transition statements for car alarm state, and calls
 *            the different drivers.
 *
 *  Revisions:
 *    \li 03-03-2011 Created alarm_task file from ME405 format
 *    \li 03-03-2011 Created case and transition statements
 *
 */
//*****************************************************************************
#include <stdlib.h>                                                     // Standard C library
#include "rs232int.h"                                                   // Include header for serial port class
#include <avr/io.h>                                                     // You'll need this for SFR and bit names
#include "stl_timer.h"                                                  // Microsecond-resolution (approx.) timer#include "stl_task.h"
#include "door_sensor_driver.h"                                         // Include header for keypad_driver
#include "keypad_driver.h"                                              // Include header for keypad_driver
#include "alarm_task.h"                                                 // Include header for alarm_task
//-----------------------------------------------------------------------------
/** This constructor creates a alarm task. Sor far it only needs a pointer to the keypad
 * driver, but will eventually need pointers to microphone, bluetooth, accelerometer, and
 * door sensor drivers.
 */

bool door_returns, microphone_returns, vibration_returns, window_break;
        time_stamp vib_time;
        time_stamp mic_time;
        task_timer timer;

        alarm_task::alarm_task (task_timer& t_timer, time_stamp& t_stamp, keypad_driver* p_key,
                                door_sensor_driver* p_door, base_text_serial* p_ser)
        : stl_task (t_timer, t_stamp)
{
        p_door_sensor = p_door;
        p_keypad = p_key;                                               // Save pointers locally
```

```
	p_serial = p_ser;

	// Set up Port A Pins 2,4,6,and 7 as outputs
	DDRA |= (1 << DDA2) | (1 << DDA4)| (1 << DDA6)| (1 << DDA7);
	PORTA |= (1 << PORTA0) | (1 << PORTA2) | (1 << PORTA4)| (1 << PORTA6)| (1 << PORTA7);
	PORTA = 0x00;
	// Set up the interrupts
	MCUCR |= (1 << ISC00);
	MCUCSR |= (1 << ISC2);
	SREG |= 0b1000000;
	GICR |= (1 << INT2) | (1 << INT0);
	// Initialize the different sensor flags
	keypad_returns = false;
	door_returns = false;
	microphone_returns = false;
	window_break = false;
	remote_returns = 0;

}
//---------------------------------------------------------------------------
/** This is the function which runs when it is called by the task scheduler. There are
 *  three states for the car alarm system, in which different driver functions are called
 *  @param state The state of the task when this run method begins running
 *  @return The state to which the task will transition, or STL_NO_TRANSITION if no
 *          transition is called for at this time
 */

char alarm_task::run (char state)
{
	switch (state)
	{
		// State 0 is Car Alarm System Disabled
		case (0):

			// If the remote input is high (remote button was pressed)
			if(	remote_returns == 1)
			{
				// Set flags to false
```

```
                window_break = false;
                microphone_returns = false;
                vibration_returns = false;
                keypad_returns = 0;
                // Delay before transitioning to the next state
                for(i = 0; i < 5; i++)
                        p_serial->puts ("DELAYING BEFORE TRANSITION\r\n\n");
                // Turn on LED showing system is enabled
                PORTA = 0x00;
                PORTA |= (1 << PIN7);
                // Transitions to Enabled State
                return (1);
        }
        // If anything else is pressed, stay in Case 0
        else
                return(STL_NO_TRANSITION);
        break;

// In State 1 is Car Alarm System Enabled
case (1):
        // If the remote input is high (remote button was pressed)
        if(PINA &0b00000001)
                remote_returns = 0;
        // Runs check_inputs function from keypad_driver, sets keypad_returns to
        // the returned value.
        keypad_returns = p_keypad->check_inputs();
        // Runs check_sensors function from door_sensor_driver, sets door_returns to
        // the returned value.
        door_returns = p_door_sensor->check_sensors();
        // Enters if statement when keypad_returns is 2 or remote returns is 0
        // and when door_returns is false and window break is false
        if((remote_returns == 0 || keypad_returns == 2) && !door_returns && !window_break)
        {
                // Set flags to false
                microphone_returns = false;
                vibration_returns = false;
                window_break = false;
                // Delay before transitioning to the next state
```

```
                for(i = 0; i < 5; i++)
                        p_serial->puts ("DELAYING BEFORE TRANSITION\r\n\n");
                // Turn off LEDs
                PORTA = 0x00;
                // Transitions to Disabled State
                return (0);

        }
        // Enters if statement when keypad_returns is 1 or when door_returns is true
        // or when window break is true
        else if(keypad_returns == 1 || door_returns || window_break)
        {
                // Set flags to false
                microphone_returns = false;
                vibration_returns = false;
                window_break = false;
                remote_returns = 2;
                keypad_returns = 0;
                // Delay before transitioning to the next state
                for(i = 0; i < 5; i++)
                        p_serial->puts ("DELAYING BEFORE TRANSITION\r\n\n");
                // Turn on LED and Horn
                PORTA = 0x00;
                PORTA |= (1 << PIN6);
                // Transitions to Alarm Sounding State
                return (2);
        }
        // If anything else is pressed, stay in Case 1
        else
                return(STL_NO_TRANSITION);
        break;

// In State 2 is Car Alarm System Sounding
case (2):
        // Runs check_inputs function from keypad_driver, sets keypad_returns to
        // the returned value.
        keypad_returns = p_keypad->check_inputs();
        // Enters if statement when keypad_returns is equal to 2 or when remote pin is high
```

```
                        if(PINA &0b00000001 || keypad_returns == 2)
                        {
                                // Set flags to false
                                window_break = false;
                                microphone_returns = false;
                                vibration_returns = false;
                                // Delay before transitioning to the next state
                                for(i = 0; i < 5; i++)
                                        p_serial->puts ("DELAYING BEFORE TRANSITION\r\n\n");
                                // Turn off LED and horn
                                PORTA = 0x00;
                                // Transitions to Disabled State
                                return (0);
                        }
                        // If anything else is pressed, stay in Case 1
                        else
                                return(STL_NO_TRANSITION);
                        break;

                // If the state isn't a known state
                default:
                        return (0);
        };
        // If we get here, no transition is called for
        return (STL_NO_TRANSITION);
}
// Interrupt service routine for microphone input
ISR(INT2_vect)
{
        // Set mic flag to true and save time
        microphone_returns = true;
        mic_time = timer.get_time_now();
        // Turn on LED
         PORTA |= (1 << PIN2);
        // If time of mic is within a few seconds of vib time
        if( (vibration_returns) && (mic_time <= vib_time + 100000) )
                window_break = true;
}
```

```
// Interrupt service routine for vibration sensor input
ISR(INT0_vect)
{
	// Set vib flag to true and save time
	vibration_returns = true;
	vib_time = timer.get_time_now();
	// Turn on LED
	 PORTA |= (1 << PIN4);
	// If time of vib is within a few seconds of mic time
	if( (microphone_returns) && (vib_time <= mic_time + 100000) )
		window_break = true;
}
```

Alarm_task.h

```
//*****************************************************************************
/** \file alarm_task.h
 *    This file declares alarm task and its functions
 *
 *  Revisions:
 *    \li 03-03-2011 Created alarm_task file from ME405 format
 *
 */
//*****************************************************************************
#ifndef _ALARM_TASK_H_
#define _ALARM_TASK_H_
#include "stl_task.h"
#include "door_sensor_driver.h"                         // Include header for keypad_driver
#include "keypad_driver.h"                              // Include header for the keypad_driver class
//-----------------------------------------------------------------------------
class alarm_task : public stl_task
{
        protected:
                door_sensor_driver* p_door_sensor;
                keypad_driver* p_keypad;                ///< Pointer to one motor controller
                base_text_serial* p_serial;                     ///< Pointer to a serial port for messages
                uint8_t keypad_returns;
                uint8_t remote_returns;

                uint32_t i;
        public:
                // The constructor creates a new task object
                alarm_task (task_timer&, time_stamp&, keypad_driver*, door_sensor_driver*, base_text_serial* = NULL);
                // The run method is where the task actually performs its function
                char run (char);

};

#endif // _ALARM_TASK_H_
```

Door_Sensor.cpp

```
//=============================================================================================
/** \file door_sensor_driver.cpp
 *     This file contains a program which reads inputs from two door sensors
 *
 *  Revisions:
 *    \li  03-04-11  Created door sensor driver from ME405 template
 */
//=============================================================================================
#include <stdlib.h>                                                     // Standard C library
#include "rs232int.h"                                                   // Include header for serial port class
#include <avr/io.h>                                                     // You'll need this for SFR and bit names
#include "stl_timer.h"                                                  // Microsecond-resolution (approx.) timer
#include "door_sensor_driver.h"                                         // Include header for keypad_driver
//---------------------------------------------------------------------------------------------

door_sensor_driver::door_sensor_driver (base_text_serial* p_serial_port)
{
        ptr_to_serial = p_serial_port;        // Store the serial port pointer locally
        // Enable PortD
        DDRD |= (0 << DDD7);
        PORTD |= (1 << PORTD7);

}
//---------------------------------------------------------------------------------------------
/** This function reads inputs from two door sensors
 *  \param  void
 *  \return bool
 */
bool door_sensor_driver::check_sensors (void)
{
        // Checks PortD Pin7,if high -> enter if statement
        if(PIND & 0b10000000)
        {
                // Returns true for triggering alarm
                return (false);
```

```
        }
        else
        {
                // Returns false for not triggering alarm
                return (true);
        }

}
```

Door_Sensor.h

```
//=============================================================================================
/** \file  door_sensor_driver.h
 *    This file declares keypad_driver class and its functions
*  Revisions:
 *    \li  03-04-11  Created door sensor header file from ME405 template
 */
//=============================================================================================
/// This define prevents this .h file from being included more than once in a .cc file
#ifndef _DOOR_SENSOR_DRIVER_H_
#define _DOOR_SENSOR_DRIVER_H_
class door_sensor_driver
{
        protected:
                // The door sensor class needs a pointer to the serial port for testing purposes
                base_text_serial* ptr_to_serial;
        public:
                // The constructor sets up Ports and initializes variables
                door_sensor_driver (base_text_serial*);
                // Checks the sensors
                bool check_sensors (void);
};

#endif // _DOOR_SENSOR_DRIVER_H_
```

Keypad_Driver.cpp

```
//=============================================================================================
/** \file keypad_driver.cpp
 *     This file contains a program which reads inputs from an external keypad
 *                      for a Car Alarm System
 *
 *  Revisions:
 *    \li  02-23-11  Created main_test file from motor_test from ME405
 *    \li  03-03-11  Edited main_test to become an object keypad_driver
 */
//=============================================================================================
#include <stdlib.h>                                                     // Standard C library
#include "rs232int.h"                                                   // Include header for serial port class
#include <avr/io.h>                                                     // You'll need this for SFR and bit names
#include "stl_timer.h"                                                  // Microsecond-resolution (approx.) timer
#include "keypad_driver.h"                                              // Include header for keypad_driver
//---------------------------------------------------------------------------------------------
keypad_driver::keypad_driver (base_text_serial* p_serial_port)
{

            // Enable PortD as Inputs
            DDRD |= (0 << DDD3) | (0 << DDD4) | (0 << DDD5) | (0 << DDD6);
            PORTD |= (1 << PORTD3) | (1 << PORTD4) | (1 << PORTD5) | (1 << PORTD6);

            // Initializes count variable i to zero, wrong_code_count to 0, and wrong_code_f
            uint32_t i = 0;
            uint8_t wrong_code_count = 0;
            bool wrong_code_f = false;
}
//---------------------------------------------------------------------------------------------
/** This function reads inputs from an external keypad from PortC
 *  \param  void
 *  \return uint8_t
 */
uint8_t keypad_driver::check_inputs (void)
{                       in3
```

```
// Checks for input from PortD 4
if(PIND & 0b00001000)
{
        // Resets wrong code flag and count variable i
        wrong_code_f = false;
        i = 0;
        // Loop that checks for next code input
        while(i <= 4800000 && !wrong_code_f)
        {
                // Checks for input from PortD Pin4
                if (PIND & 0b00010000)
                {
                        // Resets count variable i
                        i = 0;
                        // Loop that checks for next code input
                        while(i <= 4800000 && !wrong_code_f)
                        {
                                // Checks for input from PortD Pin3
                                if(PIND & 0b00001000)
                                {
                                        // Resets count variable i
                                        i = 0;
                                        // Loop that checks for next code input
                                        while(i <= 4800000  && !wrong_code_f)
                                        {
                                                // Checks for input from PortD Pin5
                                                if (PIND & 0b00100000)
                                                {
                                                        // Resets count variable i
                                                        i = 0;
                                                        // Loop that checks for next code input
                                                        while(i <= 4800000 && !wrong_code_f)
                                                        {
                                                                // Checks for input from PortD Pin6
                                                                if (PIND & 0b01000000)
                                                                {
                                                                        // Resets the wrong code count to zero
                                                                        wrong_code_count = 0;
```

```
                                        // System is Disarmed due to correct code entered
                                        return (2);
                                }
                                // Any input other than PortD Pin6
                                else if((PIND & 0b00001000 || PIND & 0b00010000 || PIND & 0b00100000) &&
i >=160000)

                                {
                                        // Sets the Wrong code flag to true
                                        wrong_code_f = true;
                                        // Increments wrong code count by one
                                        wrong_code_count++;
                                }
                                // if nothing is pressed, increment counting variable
                                else
                                        i++;
                        }
                }
                // Any input other than PortD Pin5
                else if((PIND & 0b00001000 || PIND & 0b00010000 || PIND & 0b01000000) && i >=160000)
                {
                        // Sets the Wrong code flag to true
                        wrong_code_f = true;
                        // Increments wrong code count by one
                        wrong_code_count++;
                }
                // if nothing is pressed, increment counting variable
                else
                        i++;
        }
}
// Any input other than PortD Pin3
else if((PIND & 0b00010000 || PIND & 0b00100000 || PIND & 0b01000000) && i >=160000)
{
        // Sets the Wrong code flag to true
        wrong_code_f = true;
        // Increments wrong code count by one
        wrong_code_count++;
}
```

```
                                        // if nothing is pressed, increment counting variable
                                        else
                                                i++;
                                }

                        }
                        // Any input other than PortD Pin4
                        else if((PIND & 0b00001000 || PIND & 0b00100000 || PIND & 0b01000000) && i >=160000)
                        {
                                // Sets the Wrong code flag to true
                                wrong_code_f = true;
                                // Increments wrong code count by one
                                wrong_code_count++;
                        }
                        // if nothing is pressed, increment counting variable
                        else
                                i++;
                }
        }
        // If code entered incorrectly four times ->wrong_code_count = 4
        else if(wrong_code_count >= 4)
        {
                // Resets wrong code count to zero
                wrong_code_count = 0;
                // Alarm is Set off due to mistyping code four times
                return (1);
        }
// System State has not changed
return (0);
}
```

Keypad_Driver.h

```
//=============================================================================================
/** \file  keypad_driver.h
 *    This file declares keypad_driver class and its functions
 *
 *  Revisions:
 *    \li  03-03-11  Created keypad_driver header file from ME405 template
 */
//=============================================================================================
/// This define prevents this .h file from being included more than once in a .cc file
#ifndef _KEYPAD_DRIVER_H_
#define _KEYPAD_DRIVER_H_
//---------------------------------------------------------------------------------------------

class keypad_driver
{
        protected:
                // The keypad class needs a pointer to the serial port for testing purposes
                base_text_serial* ptr_to_serial;
                // General count variable
                uint32_t i;
                // Variable to keep track of number of times the code was incorrect
                uint8_t wrong_code_count;
                // Flag for when the code was entered incorrectly
                bool wrong_code_f;
        public:
                // The constructor sets up Ports and initializes variables
                keypad_driver (base_text_serial*);
                // Check_inputs checks for correct inputs in the correct order and returns:
                // 0 for no transistion
                // 1 for alarm triggered when code entered incorrectly four times
                // 2 for alarm system disabled
                uint8_t check_inputs (void);
};

#endif // _KEYPAD_DRIVER_H_
```