|  |
| --- |
| Project Design Document |
|  |
| **Project: Alarm Clock Radio**  **Prepared By:**  **Nick Schrock,**  **Kevin Sager** |
| **Submitted to:** Dr. Robert Bossemeyer and Dr. Heidi Jiao  EGR 315/326 Term Project  Fall Term 2014  12/5/2014 |

Contents

[Abstract (or Executive Summary) iii](#_Toc371249642)

[1.0 Introduction and Design Background 1](#_Toc371249643)

[2.0 Requirements and Functional Specifications 3](#_Toc371249644)

[3.0 System Architecture 6](#_Toc371249645)

[3.1 Hardware Specifications 6](#_Toc371249646)

[3.1.1 Hardware Design 6](#_Toc371249647)

[3.1.2 Hardware Implementation 6](#_Toc371249648)

[3.2 Software Specifications 6](#_Toc371249649)

[3.2.1 Software Design 6](#_Toc371249650)

[3.2.2 Software Implementation 19](#_Toc371249651)

[4.0 Validation 19](#_Toc371249655)

[5.0 Conclusions 20](#_Toc371249656)

[6.0 Project Budget and Schedule 20](#_Toc371249657)

[6.1 Budget 20](#_Toc371249658)

[6.2 Schedule 20](#_Toc371249659)

[7.0 Revision History 20](#_Toc371249660)

[Appendix A – Electrical Schematics A](#_Toc371249661)

[Appendix B – Bill of Materials B](#_Toc371249662)

[Appendix C – Detailed Mechanical Drawings C](#_Toc371249663)

[Appendix D – Source Code D](#_Toc371249664)

# Abstract

Embedded systems are widely implemented in virtually all corners of the consumer product design industry. This document describes a specific application of an embedded system as a design a fully functional alarm clock radio. The finished product is designed to be marketed to the general public as a consumer good. The system design involves the integration of a real-time-clock, and FM radio receiver through the use of the ATmega328P microcontroller. These components, along with the necessary circuitry involved with them, are assembled as a single printed circuit board housed in a plastic casing. Outside of the casing, the user interface includes an LCD display screen, a solenoid alarm system, speaker, two push buttons and a slide switch for customization. The system is powered from the standard 120V AC wall socket. The system also employs a battery backup to maintain power when it is unplugged from the wall. This project represents a clear demonstration of a functional and useful embedded system. The main results of prototyping the design fluently agreed with the system requirements. After running final validation tests, it is determined that the final system design meets all of the listed requirements of the customer.

# Introduction and Design Background

The objective of the alarm clock radio design is to build and create a useful device that employs the main concepts of embedded system architecture. The primary goal of the design is to meet this objective by creating an easy-to-use implementation of an alarm clock radio that contains extra features that are typically found outside the scope of the standard alarm clock radio model. Through this design, a wide array of embedded systems subjects are implemented and used on this single device. The expectation of the customer is analogous with this goal as it requires an alarm clock radio that employs implementations of embedded systems.

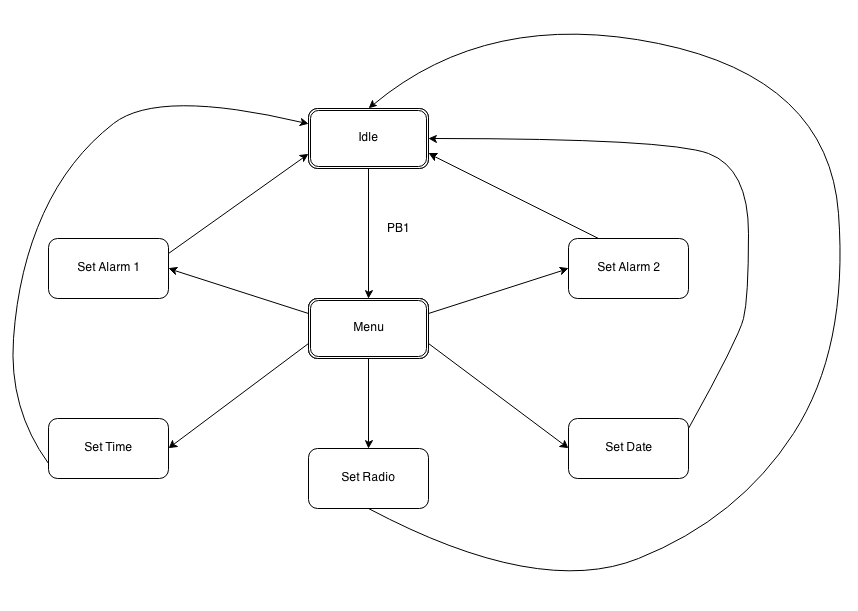
This design uses the ATmega328P microcontroller board as the primary processor of information. This is a relatively inexpensive microcontroller that is easily programmed and has plenty of functionality for this implementation. The 328P is equipped with 32 Kbytes of flash memory, two analog to digital converters, three timers plus a watchdog timer, 2 Kbytes of internal SRAM, and 1 Kbyte of non-volatile EEPROM memory. This microcontroller is interfaced with different integrated circuit peripherals that combine to a fully functional device. The integrated circuits included in the design are an FM radio receiver, a real-time clock, and an LCD display.

The FM radio receiver used in the design is the SI4703 made by Silicon Labs. This integrated circuit interfaces with the microcontroller using two-wire I2C communication. The output signal of the chip is amplified and connected to a speaker to output audio. The specific implantation of the chip is controlled by the user through two way communication with the microcontroller following a series of state changes through external button pushes from the user.

The real-time-clock IC is the DS1307 made by Maxim Integrated. The primary function of this peripheral is to keep track of the time using an external oscillator. Like the radio receiver, the RTC communicates with the microcontroller via I2C. This communication allows for the user to customize the current time and date using push buttons.

The LCD display is the PCD8544 made by Philips Semiconductors. The display screen is 48x84 pixels. The display is mounted on the primary circuit board along with the driver IC. This IC communicates with the microcontroller over the SPI communication protocol. The purpose of this device is to display the information of the current state to the user.

The system functions as a finite state machine. Upon boot-up, the user is presented with the idle state which displays the time in large font and the date in small font. With a single push of button 1, the system jumps to a menu state. From there the user is able to scroll through and select which option they want. There are a total of 5 states to choose from listed in the following order: set radio, set alarm 1, set alarm 2, set current time, and set current date. Each of those options represents a different state in which the user can manipulate data using the push buttons and slide switches to create a uniquely customized idle state. In any state, the watchdog timer keeps track of user inactivity and will jump back to the idle state after a certain period of time.



**Figure 0:** Top Level System Flowchart

In the remaining text an in depth analysis of the system will be thoroughly reviewed. The system requirements will be outlined and numbered for reference. The hardware design will be reviewed in detail. Following the hardware analysis, the software portion of the system will be explained function by function. A validation plan for each of the listed requirements will be outlined and analyzed. Concurrently, a conclusion and final overview of the projects budget and schedule will be visited. Appended to the end of the document are all of the schematics, simulations, and source code used to complete the final product.

# Requirements and Functional Specifications

**Requirement 1:** LCD displays readable information to the user

1. Auto-Adjusting Backlighting

**Technical Specification:**

* LCD Screen adjusts in environmental conditions.  The screen will dim in bright light conditions and intensify in low light conditions.

1. Display of Time and Date

**Technical Specification:**

* MCU reads date and time from RTC and displays it correctly on LCD

1. Display of Radio Station and Frequency Strength

**Technical Specification:**

* MCU reads date and time from RTC and displays it correctly on LCD

1. Menu State Listing different user states for customization

**Technical Specification:**

* Menu will project each individual menu one at a time, one push button lets user change the display and another push button scrolls through menu.

**Requirement 2:** System is powered effectively and efficiently

1. Amplifier system for Audio output on speakers

**Technical Specification:**

* Gain from operational amplifier controls voltage output according to adjustable gain.

1. Power Supply circuit

**Technical Specification:**

* System receives correct power supply input.

1. Rechargeable Battery Backup System

**Technical Specification:**

* Battery will provide energy when unit is disconnected from wall source power supply.

1. Adjustable Volume for Audio Signal

**Technical Specification:**

* Adjustable sound given by speakers controlled by potentiometer input.

**Requirement 3:** System is controlled by an easy-to-use external user interface

1. Alarm Snooze Button

**Technical Specification:**

* After alarm set, and alerts user, snooze button can be activated to reset alarm for a ten minute window.

1. Alarm Silence Button

**Technical Specification:**

* After alarm set, and alerts user, silence button will turn off alarm completely.

1. Alarm Select Switch

**Technical Specification:**

* Slide switches enable user to set alarm to a specific time and station

1. User Cancellation Button

**Technical Specification:**

* Pushbutton will allow user to escape from any given menu to change back to default menu.

1. Power LED Indicator

**Technical Specification:**

* LED will illuminate when radio tuner receives power.

1. Signal LED Indicator

**Technical Specification:**

* LED will illuminate when a strong signal is received.

**Requirement 4:** Software design allows for enhanced usability

1. Utilization of built-in watchdog timer for User interface time outs

**Technical Specification:**

* Keeps track of usage so LCD Screen defaults after a period of inactivity.

1. Customizable alarm settings

**Technical Specification:**

* Each alarm is able to be programmed to a certain respective activation time. The user also has the option to pre-set the station for when the alarm goes off.

1. Preset stations

**Technical Specification:**

* By one pushbutton, station will be chosen and another pushbutton will set the 1 through 5 station select channel for future use.

**Requirement 5:** System is fully assembled as its own functional device

1. Printed circuit board containing all internal circuitry

-**Technical Specification**

**-**A PCB will house all of the internal circuitry of the system

-Final Design and Manufacturer Plan

-All functional requirements will be re-verified to ensure proper design on PCB.

1. Enclosure for Unit

-**Technical Specification**

**-**A PCB will house all of the internal circuitry of the system

-Final Design and Manufacturer Plan

-All functional requirements will be re-verified to ensure proper design on PCB.

# 3.0 System Architecture

## Hardware Specifications (how hardware will be used)

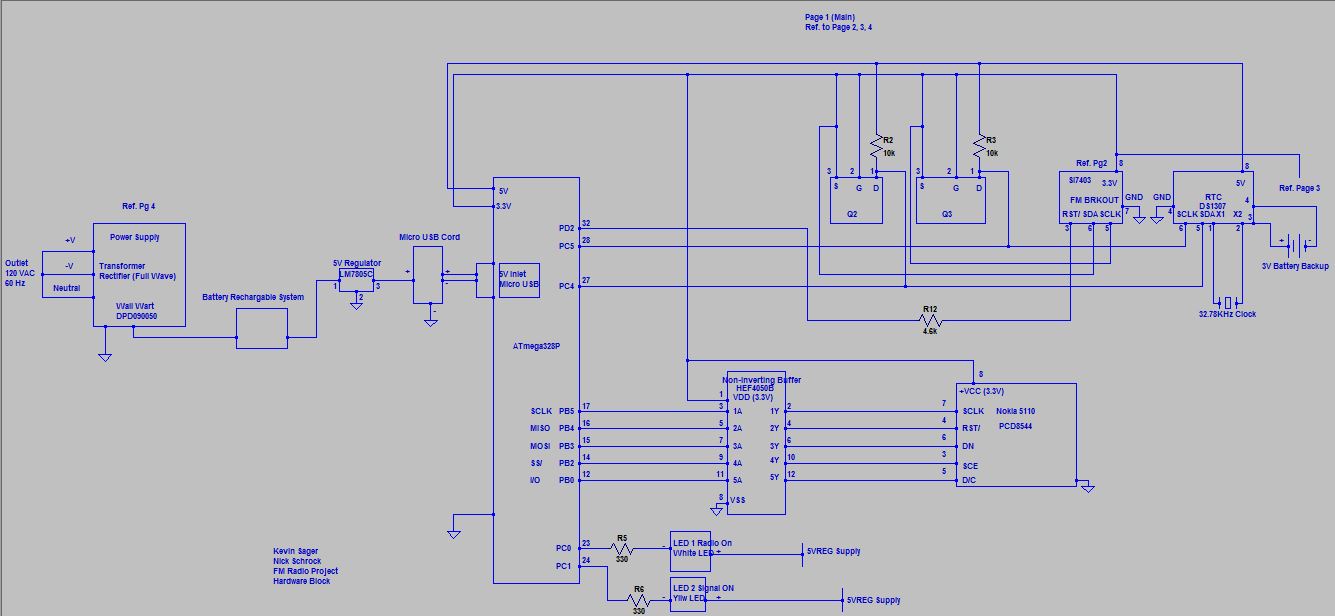


Figure 1. Hardware Block Diagram (Page 1 of 4)

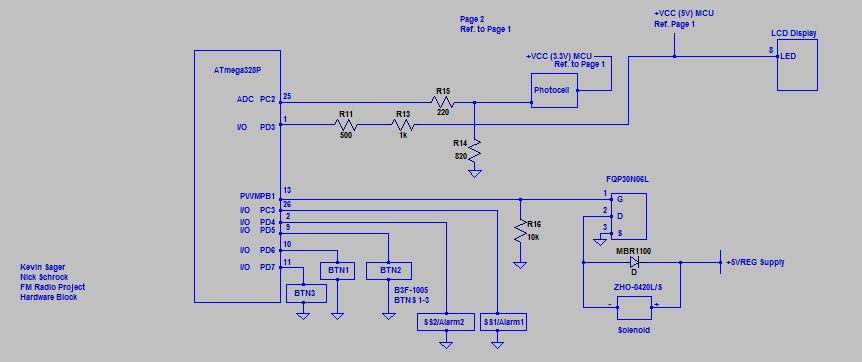


Figure 2. Hardware Block Diagram (Page 2 of 4)

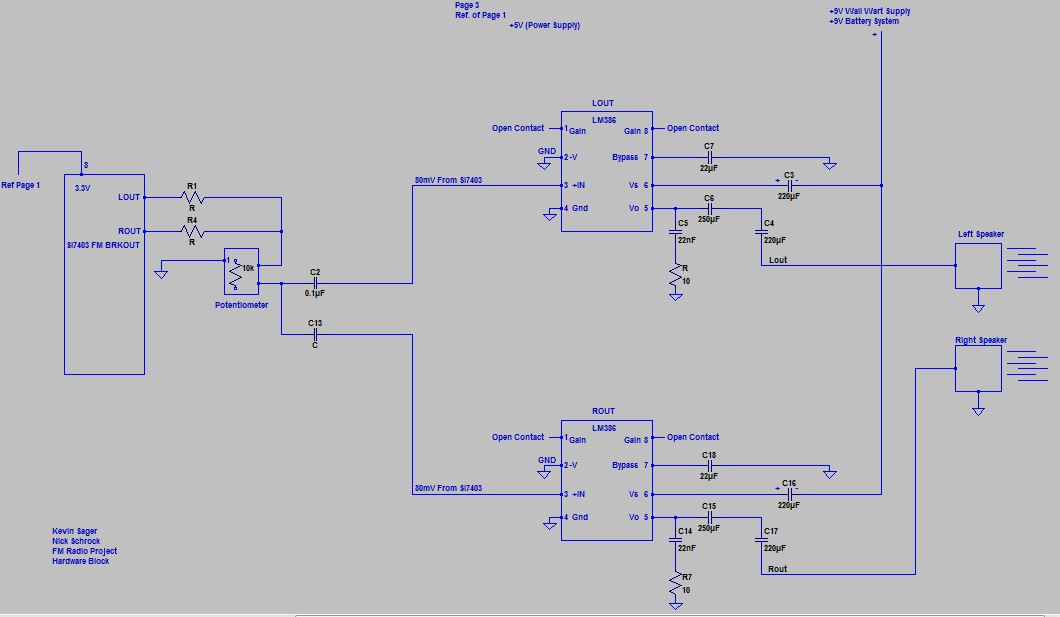


Figure 3. Hardware Block Diagram (Page 3 of 4)

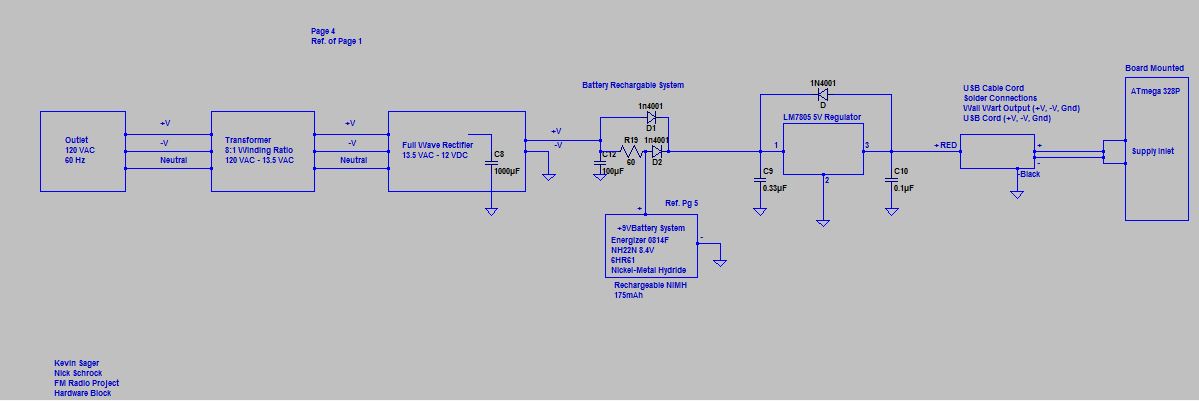


Figure 4. Hardware Block Diagram (Page 4 of 4)

The alarm clock FM radio has specified functional requirements such as those listed above in the specification requirements section and are met with hardware specifications. These hardware specifications include an 8-bit CPU 20MHz internal clock microcontroller, operating between 1.8 to 5.5 volts (supplying low current circuits), with programmable memory, EEPROM and RAM (to store constant variables and retain uploaded programs from Studio Software). A microcontroller by ATMEL model 328P will have necessary capabilities to perform inter-processing and controlling functions including peripherals with timers (such as PWM for both the solenoid circuit and LED backlighting for the LCD), analog to digital conversions (6-channel 10-bit resolution for LED backlighting on the LCD), digital to analog conversions, watchdog timers if program gets caught inside a infinite loop, sleep modes to save power, countless interrupts to free up its processing (during button pushes), and 23 I/O lines to send and receive all of the analog and digital communicating processes. This MCU ATMEL 328P will also use two protocols (I2C and SPI for RTC and LCD, respectively) to communicate between slave and master devices (master is MCU). The MCU will inter-process all the inputs and outputs of the design and aid in specifications of the customer.

A real time clock (RTC) will be used with an internal clock oscillator at 32.7 KHz to provide time and date for the user and allow user to set alarms, times and dates. It uses an I2C bus with a 7-bit address (0x68). The RTC is a low-power, full binary-coded decimal (BCD) clock/calendar and has 56 bytes of NV SRAM. Address and data are transferred serially through an I2C, bidirectional bus. This RTC includes clock/calendar and provides seconds, minutes, hours, day, date, month, and year information operating on backup 3.0V battery supply. The RTC will maintain the time, day, month for the design and specifications of the customer.

Two circuit components will be used to take voltages from a higher range to a lower range. One component that is used will be a HEF4050B non-inverting buffer to provide 5 buffers (capable of 6) from high to low level logic conversion of 5.0 volts to 3.3 volts to interface the MCU and the LCD. The buffer has high current output capability; converts logic levels, and has parametric ratings of 5V, 10V and 15V (although the design will use logic levels of 3.3V from 5.0V). Another circuit involved with providing 3.3V from 5.0V consists of the level shifter. A level shifter circuit will convert 5.0V to 3.3V by two 2N7000 n-channel enhancement mode field effect transistors and 2 10kilo-ohm resistors as pull-up resistors (protective circuitry). 2N7000 FET’s provide voltage controlled small signal switches with low resistance (on state) for reduced power consumption of 5 ohms maximum with drain current of maximum 75mA at VGS of 4.5V which is applicable for the project design(VDS max of 0.4V). The FET also has a gate threshold of maximum 3.0V with drain current at 1mA. The voltage converters and buffer will provide alternative voltages to be utilized for different ICs such as the FM IC, and RTC which will satisfy the design and specification requirements.

An LCD Nokia 5110 (Model PCD8544) will be used for user display and interfacing the interaction between the user and MCU. Displaying 48 by 84 RAM pixels (resolution), the LCD runs on 2.7V to 3.3V. This module includes on-chip generation of LCD supply, a driver and bias voltages and results in a minimum of external components necessary for use and uses low power consumption suitable for battery use. This PCD8544 Interfaces to the ATMEL 328P through a serial bus Interface and the PCD8544 uses CMOS technology (again, low power consumption technology). The module uses SPI protocol. A temperature compensation feature can adjust the liquid crystals (internally programmed) to operate during different environmental temperatures to adjust display contrast (not associated with back lighting, this is a different feature). Normal supply current for this model is maximum 300 micro amps. Its “digital contract” includes a voltage (input high) of 0.7V and low level input voltage of 0.3V. This hardware will allow the customer to interface with the designed system and allow specification requirements to be met.

A Si4703 IC module is used for receiving frequency modulation signals of 64-108 MHz and FM digital tuning. The IC uses 2-wire I2C protocol for ease of programming, in this design (Between Slave and Master). It features complete tuner function from antenna input to digital audio output and digital and I/O supply voltage takes 1.62V to 3.6V for the IC, with a supply current of 12.8mA for low power consumption (desirable for battery use portability). The IC’s maximum power up time is 110ms for the user with a maximum input frequency of 75.9 MHz and offers noise immunity. This FM IC will provide radio stations for the customer and output analog frequency for the auxiliary control board (done with analog circuitry) to amplify the frequency and provide sound to the speakers for specification requirements.

Photocells convert light into voltage with decreasing its internal resistance with intensity in environmental lighting allowing an increase in photocurrent to flow. This provides a green type application to lower the power consumption of the circuitry with 0.1W at 25 degrees Celsius (model VT900), again for use of portability and a battery system application (using more battery power when environmental dim lighting conditions exist). It provides low cost to overall system. This hardware will be designed to adjust the intensity of the LEDs inside the LCD based off lighting in the environment and meet the specifications required.

Another piece of hardware is the solenoid alarm. It uses a ‘flyback’ diode and FQP30N06L n-Channel MOSFET rated for 60V, 32A, 35mΩ maximum resistance with a typical resistance of 27mΩ (reduced on state resistance) which will protect the circuit from large voltage spikes and minimize power consumption during operation of the circuit. Also, the device has a low gate charge of 15nC to activate very fast response times reducing the time constant. And the device can provide a drain current of 16A during switching operation and 32A maximum current during continuous operation. The FQP30N06L provides switching performance for pulse width modulation application (used with a peripheral of the MCU) and provides current to the solenoid. This switching action will provide enough power for the solenoid to actuate very fast while protecting the MCU. The gate threshold voltage has a maximum of 2.5V and minimum of 1.0V to enable the Q-point of operation at a drain current of 250 micro amps. It also has a trans-conductance of 24 Siemens which is desirable. During operation the device supplies a current of about 1.25A, which is enough for the solenoid to induce and actuate, according to the on state characteristics in the data sheet, and at a VGS of 5V has a 27mΩ resistance which lowers power consumption of the circuitry (ideal for battery portability). The ‘flyback’ diode application (Shcottky diode model MBR1100 Axial Lead Rectifier) provides polarity protection, low forward voltage, low reverse current, and high surge capacity used with rectifiers in low-voltage and high frequency inverters. MBR1100 has maximum ratings of peak repetitive reverse voltage and DC blocking voltage of 100V. This ‘flyback’ diode will most certainly protect the MCU against voltage spikes coming from the solenoid. It includes non-repetitive peak surge current of 50A limiting large currents during high loads. Other electrical characteristics, of the MBR1100 diode, include maximum instantaneous forward voltage of 0.79V during operation and a reverse current of 0.5mA with a forward current of up to 5A. Both components provide protection and minimal power consumption for the embedded system to meet specification requirements of the overall system.

The solenoid provides stroke distance and force capable of hitting a light resonating structure (for design purposes) and creating a sounding alarm for the customer to announce any event of some kind. It can provide a 2.5mm stroke at 20gf force with 1.2W continuous operating condition (100% duty cycle) and an ideal 2.4W 50% duty cycle at 1.5mm stroke distance with 45gf force. This hardware will be designed to provide a sounding alarm for the customer to use and meet the specification requirements.

A power supply will provide constant regulated voltage of 5V supplied from unregulated 9V (9V during load operation) from the wall wart and 9V from the rechargeable battery system. The first part of the power supply uses a step down transformer ratio of 9:1 stepping down the voltage from 120VAC 60Hz to 13.5VAC 60Hz to four 1N4004 (1A, 400V) rectifying diodes. These rectifying diodes convert the alternating current 13.5V to a full wave. After the full wave rectification, the 12VDC is supplied after a coupling capacitor of 1,000 micro-farads to a LM7805 5V regulator (also supplied with 9V NiMH rechargeable battery). The LM7805 regulator converts the 12VDC to a constant 5V without very much ripple to ultimately supply the 5V to other sensitive voltage devices (eliminating 89% of voltage ripple). This LM7805 will keep the noise in the circuitry to a minimum, an output a current up to 1A, output voltage of 5V with an output maximum of 5.25V, line regulation of 4mV with an input voltage greater than 7V, a load regulation of 9mV at 1.5A, and short circuit protection of 230mA for protection of the sensitive electronic devices. This hardware will be used for the main supply for the designed embedded system and meet the specification requirements for constant 5V to the sensitive circuitry and components.

The power amplifier will be accompanied by capacitors and resistors to reduce noise (also couple AC and DC voltages) and use for needed voltage drops. A LM386 power amplifier will be used. This IC is widely used for low voltage audio power amplification which is what the project is designed for during portability and yields voltage gains from 20 to 200. It is accompanied by low distortion of 0.2% and output power of 125mW for applications of both AM-FM radio amplifiers. Input voltage range of -0.4V and 0.4V, supply voltage up to 15V but will only use 5V to 9V input supply, and operating supply voltage of maximum 12V. This designed hardware will give greater amplification than the LM741 to get greater sound out of the speakers and is used as standalone IC’s to meet specification requirements.

The speakers load resistance is 8Ω and is reasonable for the application. Reasonably, the lower the output resistance, the volume increases. However, to receive better sound, an upgrade would be suggested. This hardware provides sound to the customer during reception of frequency signals and will meet the specification requirements.

Tactile switches such as the side-operated, high-force, gold plated contact operating component, it provides extended lifetime and durability preventing corrosion on the contacts and increased depressed cycles. This hardware will provide greater lifetime to the overall system and meet the specification requirements.

A battery 9V system is used for portable application of the FM radio. The Energizer NiMH 175mAh rechargeable battery (model NH22N) can be recharged a hundred times before it is no longer usable and during times when no wall power is available, the battery will provide power. Two diodes of 1N4001 type at a forward voltage of 1.0V, according to the data sheet, will act as a self-activating switch either choosing wall power when plugged in or battery power when unplugged from wall power. It also is a rechargeable circuit when wall power is supplied (wall wart voltage supply) providing constant current to the NiMH battery recharging the battery. Two diodes activate by a higher potential voltage (either from wall wart voltage supply when plugged in or battery voltage supply when wall wart unplugged). This is an added feature of the embedded system and will be designed to supply the system with power during times when the system is unplugged from main line power of the wall wart and charging the battery when main power is connected which will meet specification requirements.

All supply power will route by means of a generic micro USB cable custom modified to fit the application of power in order to route power to the MCU. This will provide ease of connectivity.

### Hardware Design (how meet functional requirements)

Each piece of hardware contributes to the control, protection, and generation of electrical signals in the embedded system design of the FM radio project. The interconnected electrical components sum up to meet functional requirements stated above in 2.0 Requirements and Functional Specifications.

Firstly, the microcontroller will interface between the user and the embedded system with timer and ADC peripheral capabilities including memory functions. It is the brain of the system and allows inputs such as analog sensors such as the photocell, buttons, and digital input information such as the real time clock IC, the Si4073 FM chip, power supply voltage, and battery system support for unit portability. The ATMEGA 328P also processes and controls these inputs to display outputs on the liquid crystal display and the actuating solenoid (during alarm settings).

In order to provide functional requirements such time, day, date, station frequency and strength, and including the stereo indication (AM/FM) and backlighting intensity for the customer, associated circuitry and hardware is involved. With devices such as the non-inverting buffer, level shifter, photocell, real time clock IC and Si7403 FM IC chip all design for interconnection circuitry to help display to the liquid crystal display and meet the functional requirements for user interface (alarm settings, station selections, presets stations, adjustable backlighting). A non-inverting buffer HEF4050B provide low level logic for the Nokia 5110 from the ATMEGA 328P (5.0V to 3.3V). Associated pins for SPI protocol from the MCU to the LCD provide for communication and these necessary lines used have different voltages. To provide this buffer, the IC HEF4050B, a mirror of the input voltage to the IC (3.3V from the MCU supply) is seen at the output lines of the buffer (output to the LCD). This concept is similar to the level shifter for both the real time clock and the FM IC. Level shifter provides interconnection of different supplied voltage IC’s such as the real time clock and Si4705 FM IC. Given the FM IC operates at 3.3V and the RTC operates at 5.0V, the level shifter operates the circuit that is communicating (SDA, and SCLK I2C protocol) to the MCU of either the 5.0V or 3.3V at one time by the three 2N7000 n-channel enhancement mode field effect transistors activating either circuit when in use. Therefore, I2C protocol is allowed for communication between two different supply voltages for two different IC’s (real time clock and FM IC) allowing communication to provide the time, day, date, station frequency, station strength, and FM/AM stereo indication.

Adjusting the LED intensity backlighting of the Nokia 5110 required is done by a circuit with a photocell (Bulk effect type). The photocell takes the lighting from the environment and converts it to a voltage sensed by the MCU ADC peripheral. Based on the voltage produced by the photocell sensed by the MCU, an algorithm process used in conjunction with the PWM mode of the MCU, linearly relates the lack of light (or abundance thereof) in the room and provides more (or less) voltage to the LED’s inside the Nokia 5110 LCD due to the PWM duty cycle adjustment (done by the MCU algorithm). A voltage divider circuit takes into account the dark resistance (maximum resistance with minimum voltage 26mV supplied by the photocell) of the device and the ambient room lighting resistance within the photocell (average voltage supply of 2.5V) to provide some contrast of voltage coming from the photocell to the MCU (this algorithm is accounted for in the ADC peripheral). Adjusting the intensity of the backlighting will meet the functional requirements.

Switches, such as the standard B3F-1000 flat type tactile switch and slide switches, allow user interface to the LCD through menu select including desired exits of menu options and control the outputs such as the alarms, indicator LED’s (station strength and status of radio on) and thus meet functional requirements. These inputs are managed by the MCU. The gold plated model increase contact reliability and lifetime allowing for a decrease in de-bounce time upon user depress. These circuits are active low to the MCU and provide ease of depression for menu select and alarm setting options. The switching circuits simply provide auxiliary control, specifically, to the functions of the solenoid alarm, station selection, and preset stations (also other included alarms), FM IC (station select).

The solenoid alarm consists of a protective ‘flyback’ diode for voltage spikes and allows the functional requirement of an actuating alarm for the user without system malfunction. This solenoid also consists of a 2mm stroke at 40gf force operating near 2.4W 50% duty cycle used for the customer when striking the resonating structure. This meets the functional design requirements providing an actuating alarm for the user to set whenever desired.

A power supply is designed for disconnecting from USB PC power supply and allowing the customer to receive power from standard AC wall outlets in any facility. Stepping down the 120VAC wall power to a reasonable 13.5V, then the voltage rectifies the alternating current for a full wave application to the whole embedded system in order to use direct current. Finally, after full wave rectification, the voltage ripple hazardous to sensitive electronics requiring smooth constant 5.0V is attained by the LM7805 regulator converting the 12VDC (actual output after step-down, rectification of wall power) to a constant 5.0V maintain the integrity of the electronic system. The power supply design meets the converting AC-DC functional requirement for the embedded system design. Also, the real time clock incorporates a small 3.0V battery for maintaining timed, day, and date functionality of the system given any shutdown or power outage for the customer. Thus the hardware will meet help meet functional requirements specified above.

Volume control and output of the FM IC radio frequencies are designed with a 10 kilo-ohm potentiometer and power amplifiers. The 10kΩ potentiometers adjust each of the speakers to the desired level for the customer, allowing for customized adjustment of the desired speaker volume. The design then meets the volume control functional requirements of the specification design allowing the customer this necessary option.

Indicating LEDs illuminate by the output signal of the MCU by communicating to the customer signal strength, and radio on, thus, meeting functional requirements for the indication of both the signal strength and radio on.

Also, the main power two diode battery circuit will provide power at all times for the user in the use of facility power and portability. This meets the functional requirements of providing power to which the user may have the option of utilizing the unit when unplugged.

The customized micro USB cable router will be used for connectivity for both cases in which the unit is powered by the wall wart or battery. This will help meet functional requirements for power connection of wall wart or battery operating conditions.

### Hardware Implementation

The power supply will supply main power to the embedded system FM radio project. It includes a step down transformer (figure 6), full-wave rectifier (figure 7), and a LM7805 voltage regulator for constant voltage supply to the system. Corresponding simulations to the given circuit can be seen in each figure following a schematic diagram. In figure 6, the step-down transformer, with a winding ratio of 8:1 will step the outlet VAC of 120 at 60 Hertz to 13.5 VAC which can be seen in Figure 7. Then, the voltage will go through a full-wave rectification as shown in Figure 8 with four 1N4004 diodes to control the current’s direction and provide AC to DC conversion shown in Figure 9. As can be seen in Figure 9, a constant DC voltage of close to 12VDC can be observed (however, the simulation is ideal and voltage ripple must be considered). A 1000 micro-Farad capacitor will aid in DC coupling (minimizing ripple voltage) the output voltage to the LM7805 regulator as can be seen in Figure 8. Figure 10 shows the fully implemented power supply, with the LM7805 5V regulator, to supply a constant 5VDC to the entire embedded system application (also seen in Figures 1, 3, 4 in above section). The simulation of the constant 5VDC supply can be seen in Figure 11 as opposed to the voltage output of the transform rectifier circuit (in blue). The LM7805 component, as discussed in previous section, regulates this transform rectifier circuit to 5V and will not output an exceeding maximum voltage of 5.25V according to the data sheet (given protective circuitry within the LM7805 and also a 'flyback' diode to prevent damage in case of short circuitry).

The load regulation, shown in figure 8, for the rectified circuit of the wall wart calculated by equation (1) with a nominal load voltage found in the data sheet to be 9V at full load, minimum load of 100mA and a max load of 500mA is found to be 64.9%.

(1)

The load regulation for the regulated circuit (including the LM7805 IC shown in figure 10) is calculated by equation (2) with a nominal load voltage found in the data sheet to be 5V nominal voltage (5mA to 1A) and a simulated minimum and maximum load of 250mA and 750mA is found to be 99.9% finding the circuit capable of handling load applications for the project design.

(2)

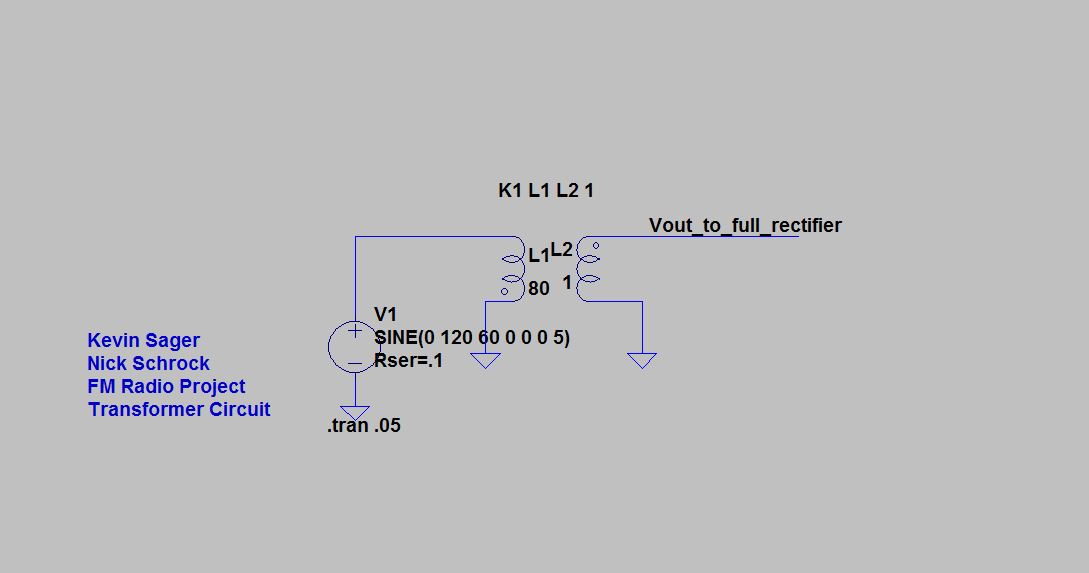


Figure 6. Power Supply (Step-Down Transformer)

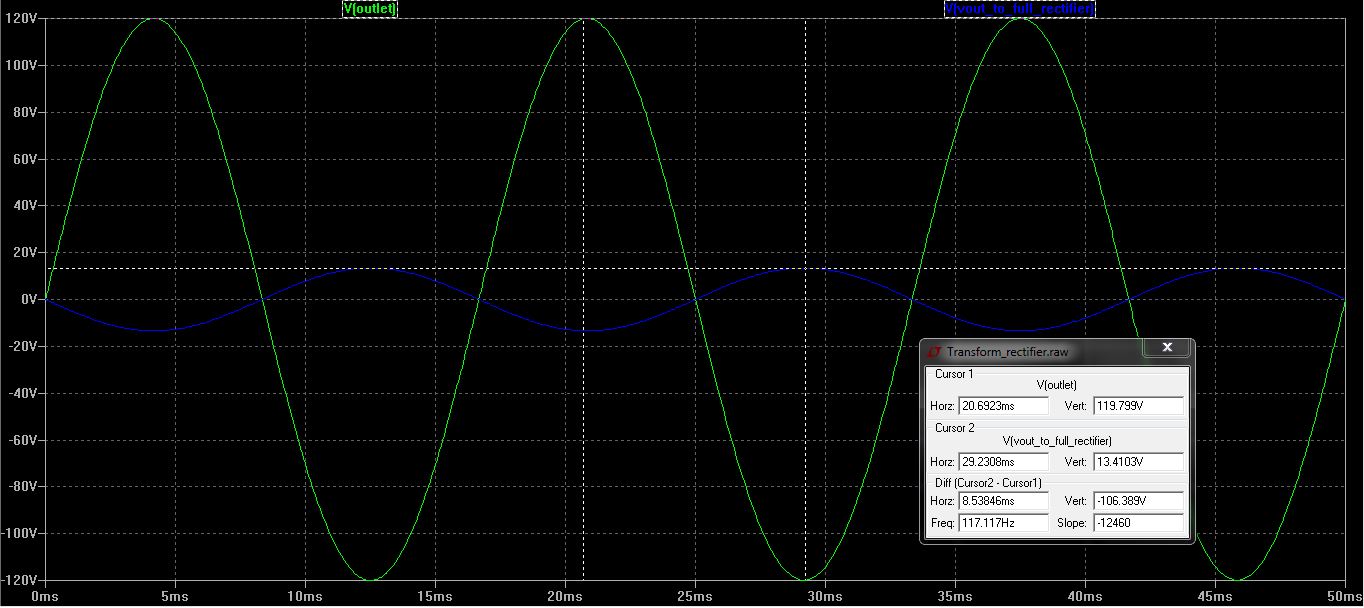


Figure 7. Simulation Step-Down Transformer

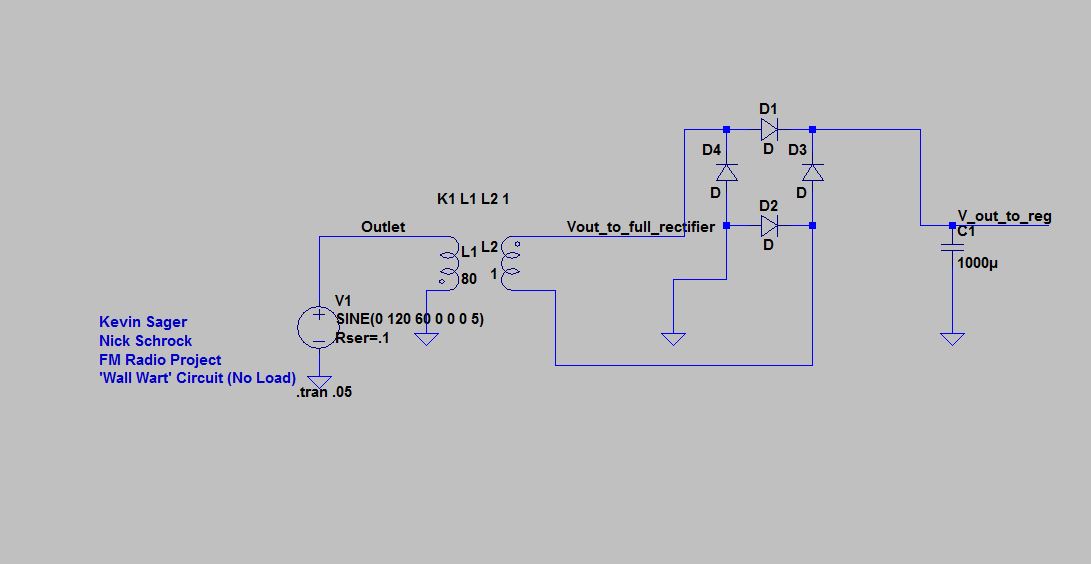


Figure 8. Power Supply (Full-Wave Rectification with DC Coupling Capacitor)

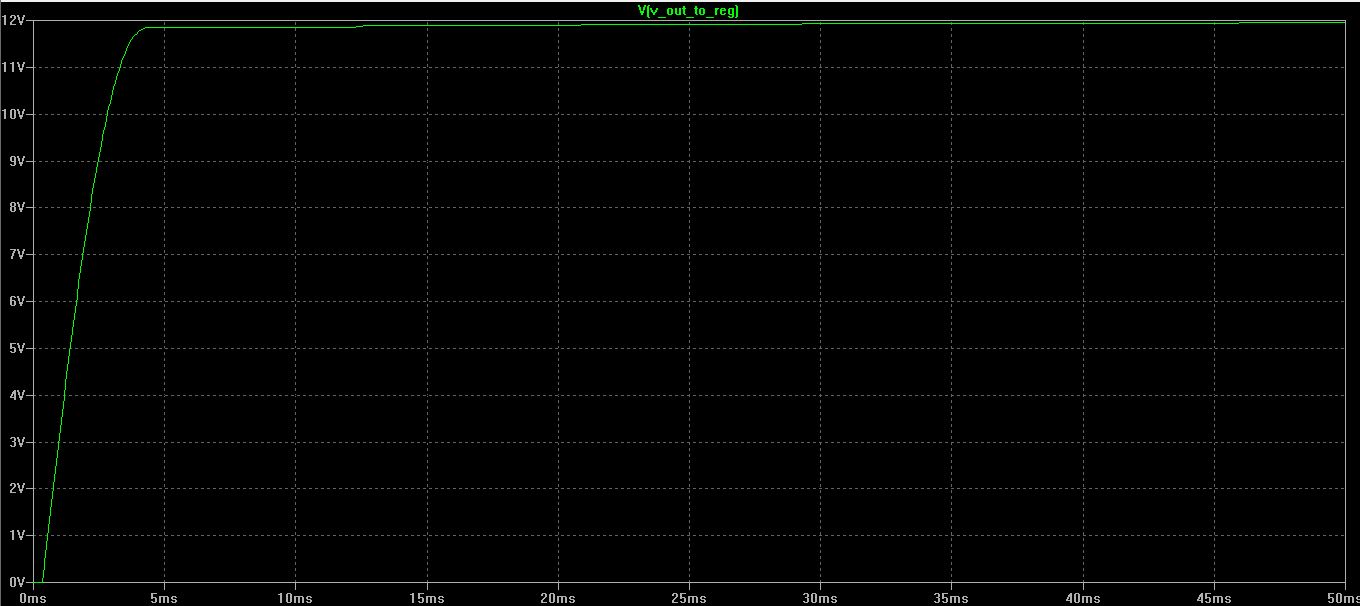


Figure 9. Power Supply Simulation (Full-Wave Rectification)

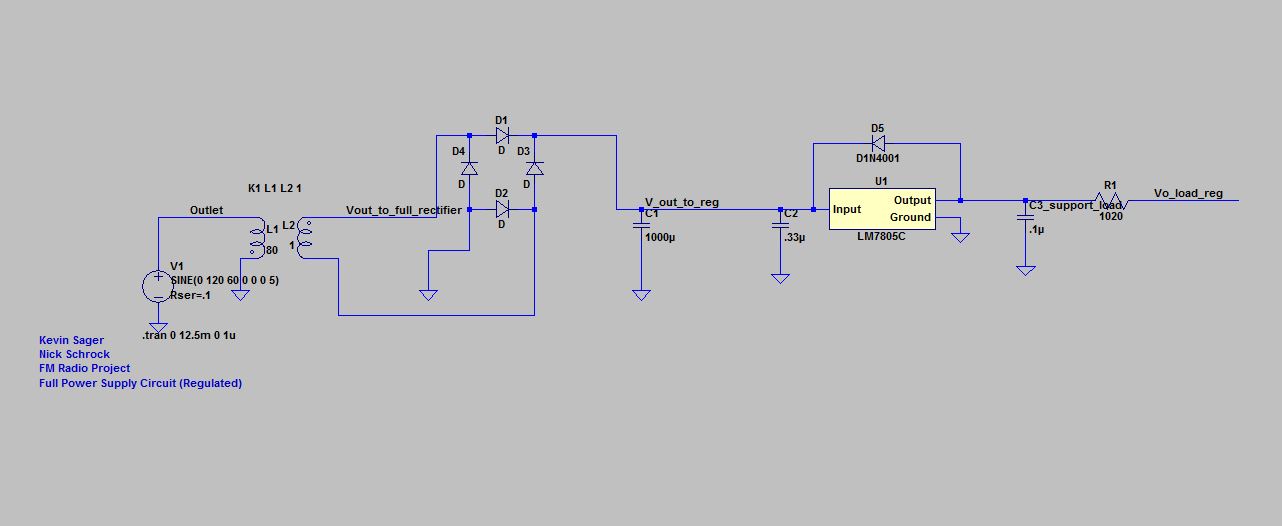


Figure 10. Fully Implemented Power Supply (With LM7805 Regulator)

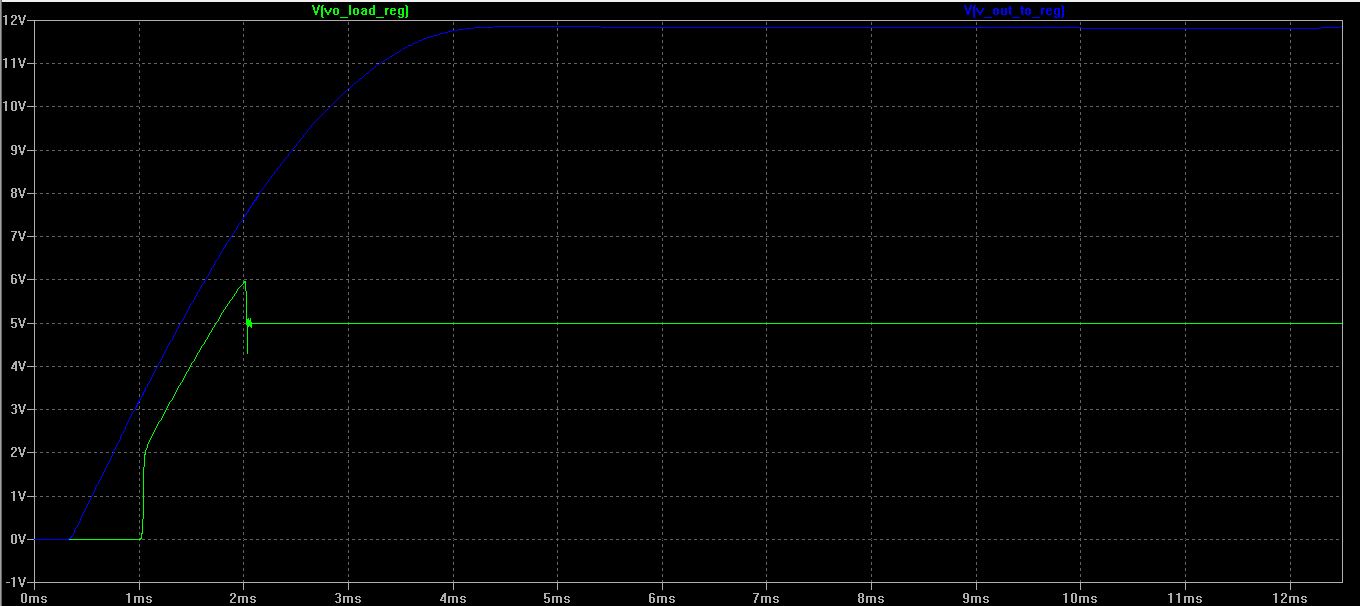


Figure 11. Fully Implemented Power Supply Simulation (With LM7805 Regulator (Green))

The RTC, FM IC circuitry is shown in Figure 12. An integrate chip DS1307 and Si4703 will be implemented to display through an LCD the time, date, day, and station frequency (by programmed registers) to the customer (a 3V battery to maintain RTC time, day and date). Customer setup of the time, date and day with the RTC will be maintained by the 3V battery in the circuit while the embedded system is powered off. Voltages of 5V and 3.3V from MCU will provide power for the IC’s Drain voltage and Gate voltage, respectively for the NMOSFETS (2N700) during level shifting operation to provide logic communication to the respective IC’s. Separately, supply voltage of 3.3V and 5V from the MCU will provide each IC with the necessary voltage (VCC of 5V for RTC, and 3.3V for Si4703). When SDA and SCL, 3.3V (3.3V side pulls circuit low) on the FM IC side of the level shifter circuit, circuitry is pulled low and the gate to source voltage rises above the threshold voltage (Voltage Threshold) which drives the Q-point for the NMOSFET and conducts. During this operation the NMOSFET acts as a switch and allows data and clock speeds, for the I2C protocol, to communicate to the pins on the MCU via FM IC and MCU. Also, during 5V operation (5V side pulls circuit low) for the RTC, the bus line is pulled low until VGS on the lower voltage section passes the threshold and the NMOSFET becomes conducting allowing SDA and SCLK to communicate to the MCU.

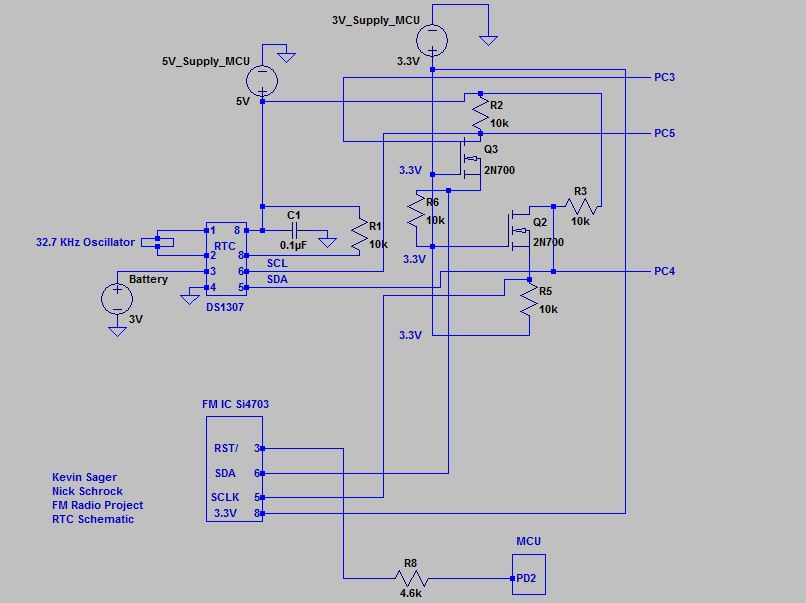


Figure 12. RTC Circuit with Level Shifting Circuit

The buffer circuit, shown in figure 13, with the LCD takes the input, by the MCU of 3.3V, and creates a logic level based on this input. It is necessary for MCU and LCD SPI communication that the 5V rated MCU be converted to a 3.3V rated Nokia 5110. For the given application, a 3.3V is supplied to Vcc of the Buffer by the MCU, as shown in figure 13, creates an output voltage of 3.3V for the Nokia 5110. In this way, the use of SPI protocol with the MCU can communicate to the display for the customer and the customer may interact with it using push buttons (I/O pins on the MCU shown in figure 24). The supply voltage to the LCD is now 3.3V through each pin 3, 4, 5, 6, and 7. These pins are necessary for SPI protocol.

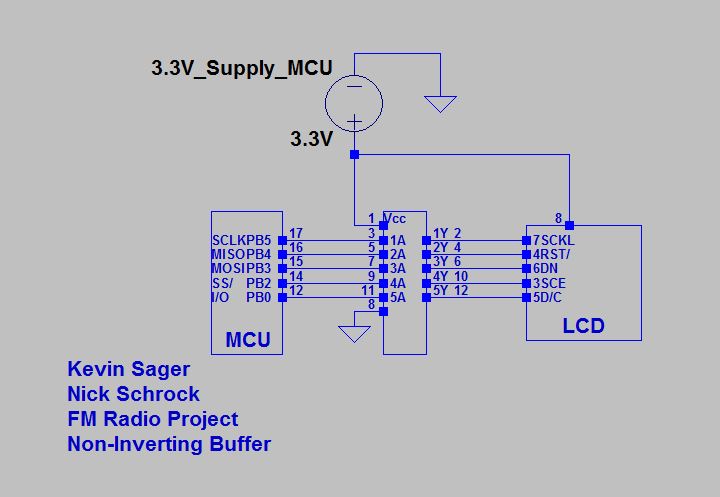


Figure 13. Non-Inverting Buffer Circuit (Communication with Nokia 5110)

Amplification circuitry can be seen in figure 14 for both the left and right speakers. This application of the LM386 was designed to amplify the rough estimate of 80mV analog signal voltage output of the Si7403 FM IC for a gain of at least 20 according to the recommended application with minimum parts. In figure 15, the amplification can be seen as a voltage gain, which according to equation (3) was calculated to be roughly 32. This gain, greater than the data sheet, had been augmented by additional DC and AC coupling capacitors. Calculations of voltage gain, current gain, and power gain can be seen in equation (3), (4) and (5). Providing 80mV to the power amplification (LM386) circuit with a -0.4V to 0.4V supply as required to the input pin reveals a large power gain of 690 which is needed for the small amount of current at the input of the LM386. This application is to provide the 8 ohm speakers enough voltage for a maximum 2 Watt design application for the speakers. The application will provide gain for the customer for ample auditory sound in the designed application.

(3)

(4)

(5)

Power amplification, with use of the LM386, has total harmonic distortion and can be seen in Figure 16 and 17 in either the left or right speaker which are considerably close to each other. Both the left and right speaker has 14.11 percent THD. In Figure 15, the lowest gain is simulated (by use of the volume control 10 kilo-ohm potentiometer) providing enough amplification to give sound to the speakers. However, the potentiometer sends signal to ground at one extreme adjustment of the potentiometer and the simulated circuitry cannot show this, thus at minimum gain the adjustment can further minimized to an amount of gain of zero for no sound if desired. Figure 17 is the simulation of the LM386 power amplifier with the highest gain with the least amount of resistance of the potentiometer.

The overall circuit for the system of amplification can be seen in Figure 14. The application will meet the specified requirements of providing sound through the speakers with a volume adjustment potentiometer.

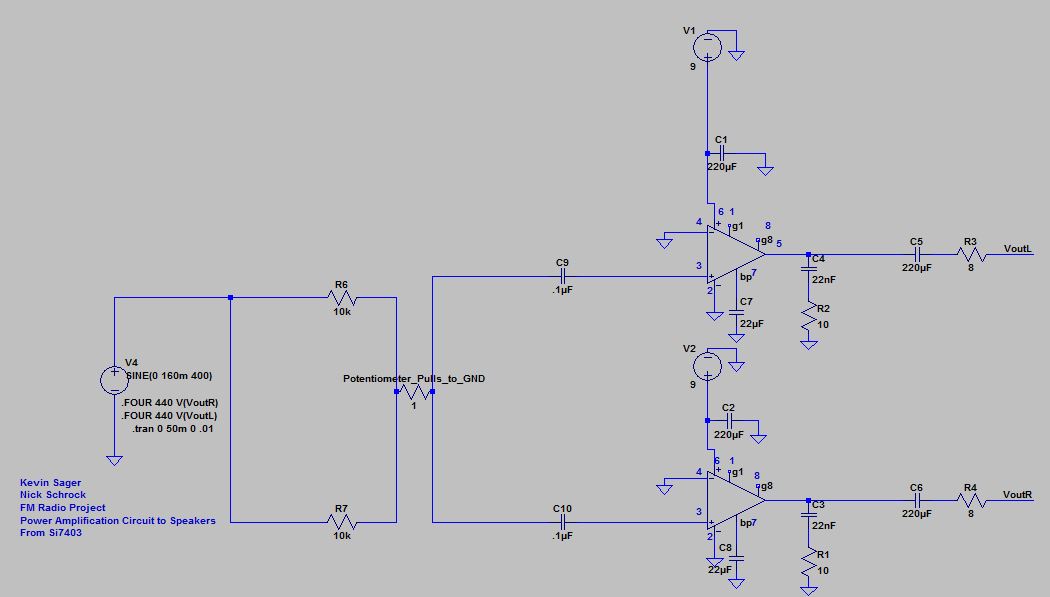


Figure 14. Power Amplifier Circuit to LM386 Power Amplifier (For Speakers)

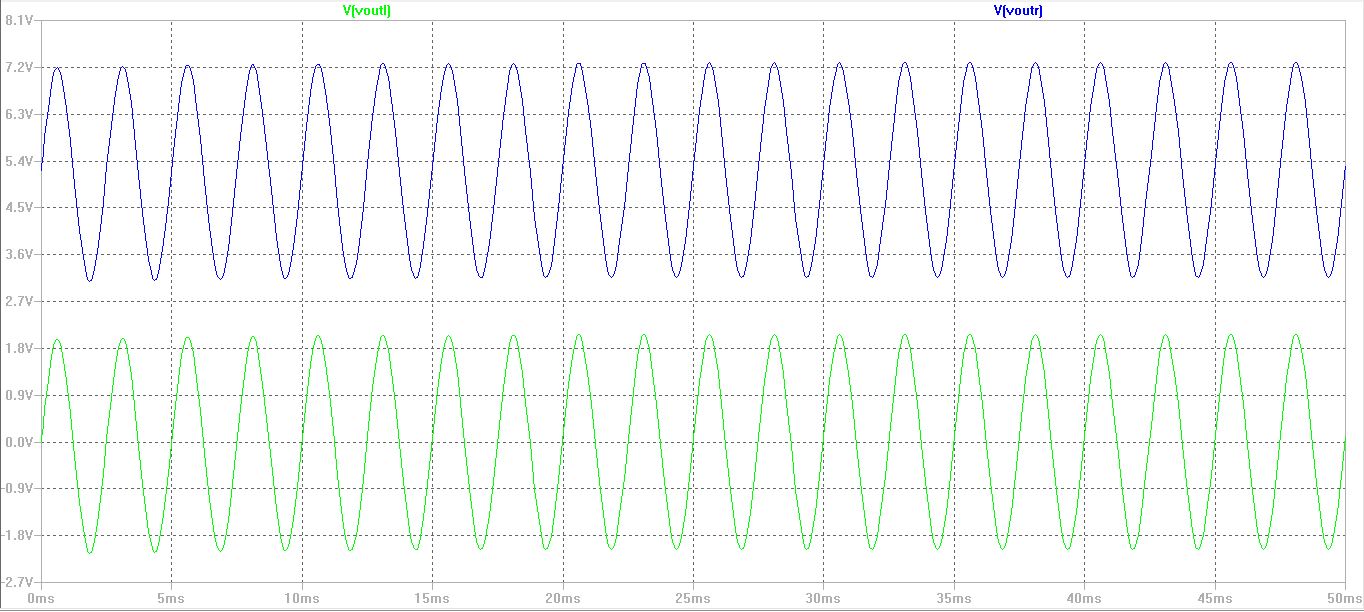


Figure 15. Power Amplifier Circuit to LM386 Power Amplifier (Simulation of Minimum Gain (Pot at 10K))

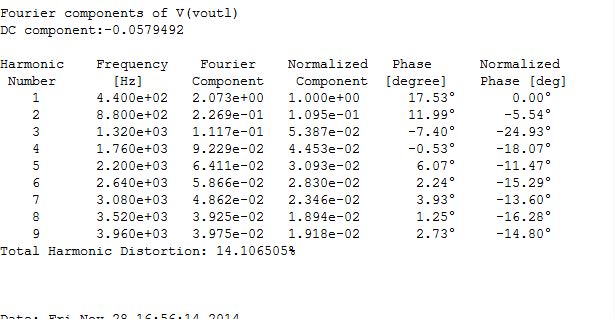


Figure 16. Power Amplifier Circuit to Speaker (Total Harmonic Distortion Left Speaker)

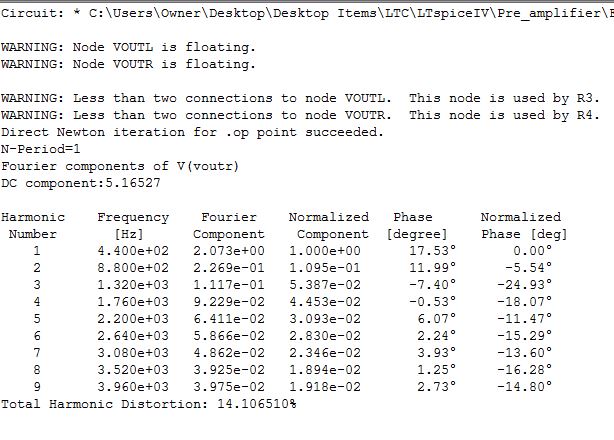


Figure 17. Power Amplifier Circuit to Speakers (Total Harmonic Distortion Right Speaker)

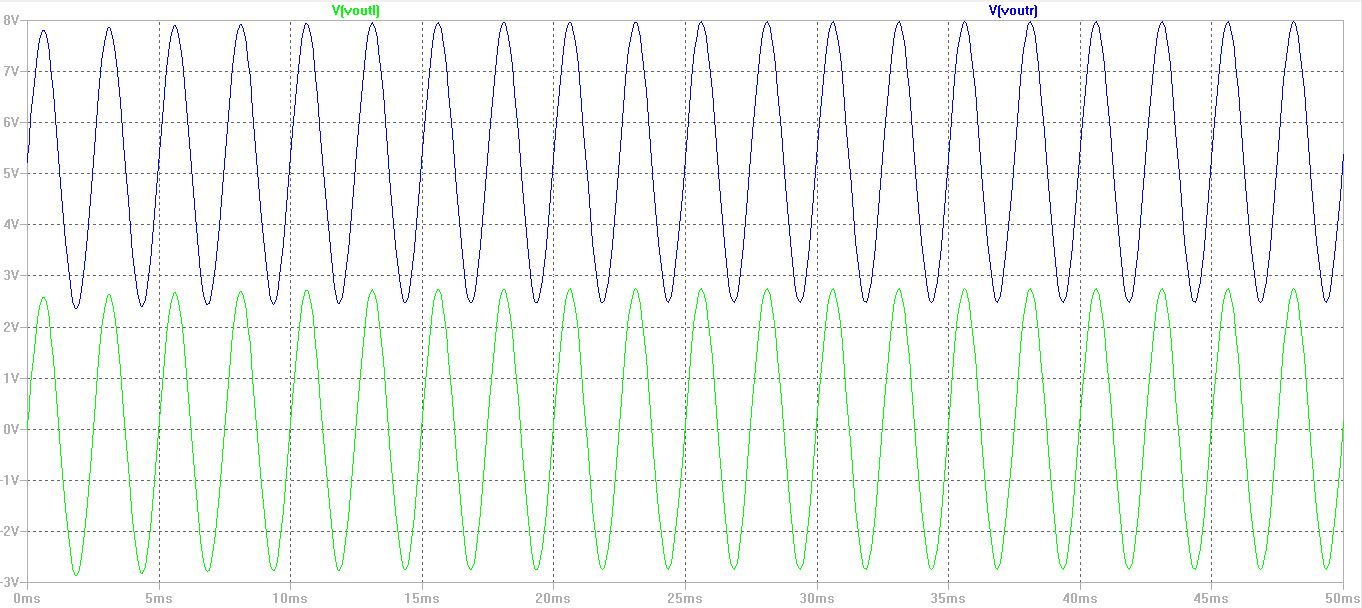


Figure 18. Full Amplification to Speakers (Simulation of Full Gain (Potentiometer at 1 ohm))

The solenoid (ZHO-042L/S) circuit in Figure 19 is used as a design application for the customer to set an alarm with the MCU. It uses a Schottky MBR1100 to prevent large reverse voltage spikes from the solenoid inductance (Back EMF) to protect the devices such as the MCU from damage. This design is tailored to the solenoid operation using MCU peripheral of Pulse Width Modulation for 10% duty cycle with a 72mA flow to the inductor at 20 Hertz and period of 50ms. At 10% the solenoid actuates fairly well with the resonating structure of thin metallic material (not shown in Figure). The NMOSFET acts, again in the circuitry, as a switch and as the MCU pulses the NMOSFET opens (very rapidly). The pull up resistor R1 is another circuit protection device used for leakage current coming back to the MCU and is necessary to prevent damage to the MCU. These protective circuits and design applications provide the customer the option for use of a fun mechanical circuit to use as an alarm.

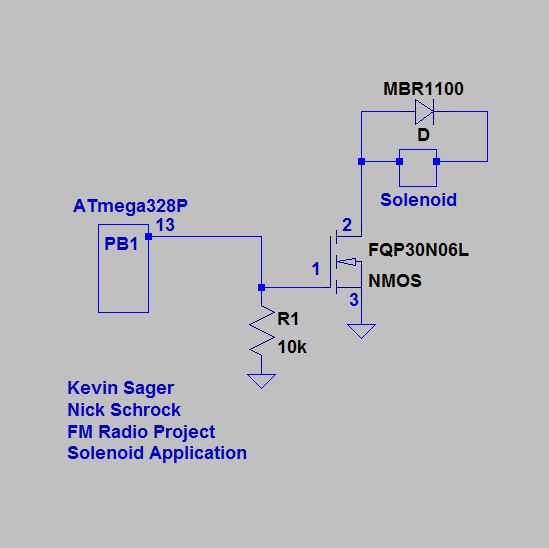


Figure 19. Solenoid Circuit for Customer Set Alarm

The photocell circuit changes the intensity of the LED’s on the Nokia 5110. An Atmega328P controls the amount of voltage to pin 8 (dependent on the ADC conversion integer sent to the MCU from the photocell) of the LCD by ADC peripheral used through PC2 pin on the MCU. This ADC converter will compute the voltage received from the photocell circuit and output the voltage algorithm out of PD3 in PWM mode of the MCU. A simulation can be seen in Figure 21 and as the MCU senses the voltage from the photocell (depending on how much); the PWM function will adjust the duty cycle of its output from PD3 to pin 8 on the LCD.

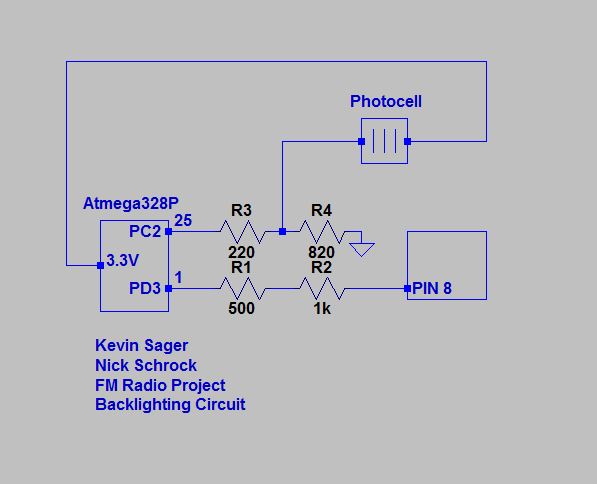


Figure 20. Photocell Circuit (Nokia 5110 Backlighting)

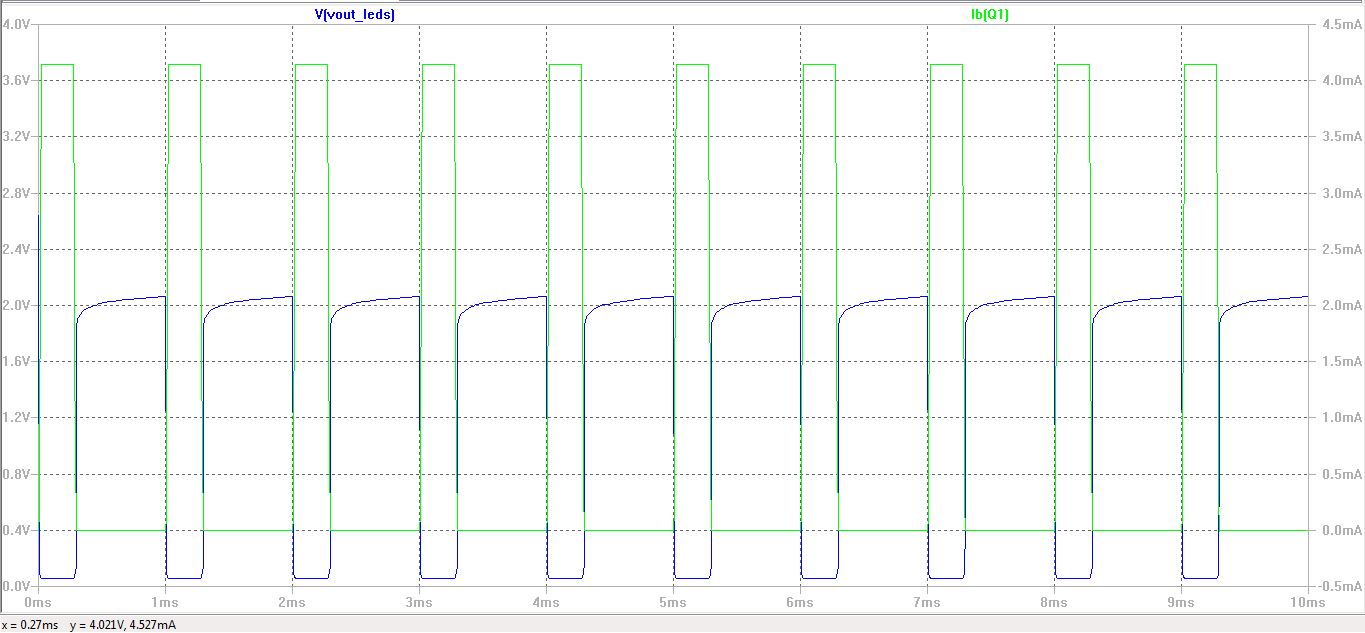


Figure 21. Simulation Photocell (MOSFET Acting as MCU in PWM Mode)

The LED circuitry can be seen in figure 22 and (typical standard light emitted diodes) indicates radio on and station strength to the customer. In the design application a threshold voltage (forward bias) is provided by the MCU upon input of the Si7403 frequency by use of customer. A design application of setting the MCU input (of both input pins 23 and 24) of active low will determine the LED’s active state. When no voltage is sensed at the pins, the respective LED will illuminate by overcoming the forward voltage of the diode by a larger potential at the anode of each LED.

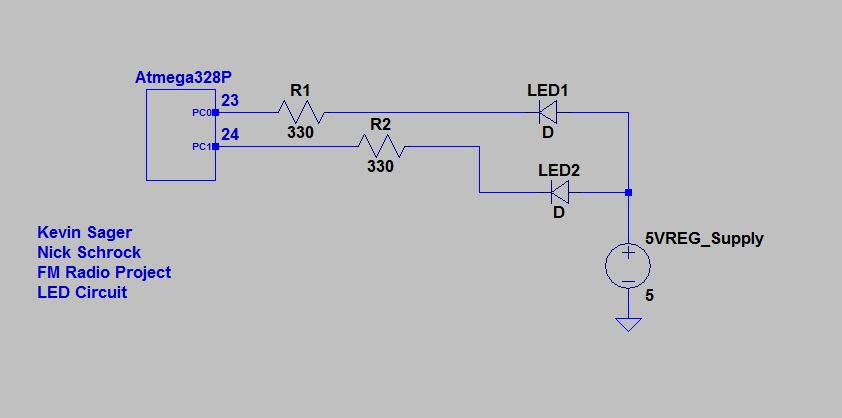


Figure 22. LED Circuit Interfacing with MCU

The switch circuitry can be seen in figure 23 and is used to interface, directly, the user and the embedded system. Buttons are programmed to be active low circuits and button 1 is used when the costumer want to select through menus. Button 2 is used then the customer wants to select a station and set the time, date, day of either the alarms or initial settings of the unit’s time, day and date. A third button is used to back out of any menu, station, or selection the customer desires in order to aid in navigation.

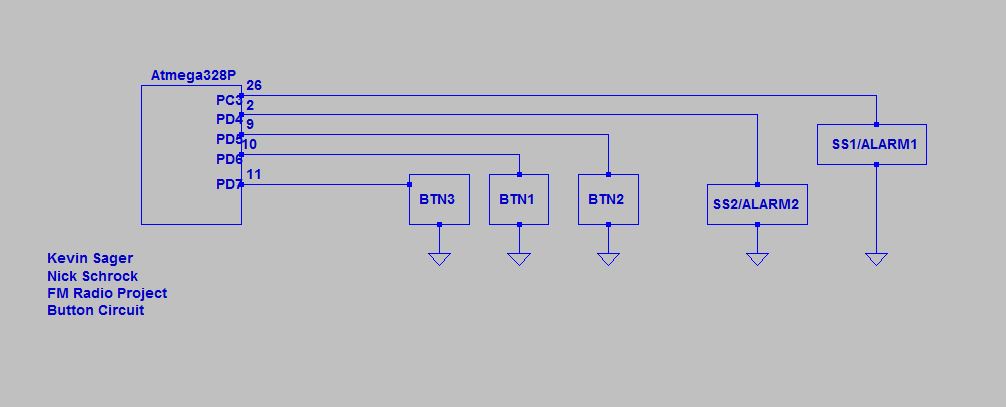


Figure 23. Button Circuit Interfacing with MCU

The rechargeable battery circuit seen in figure 24 is a part of the power supply system. A rechargeable NiMH battery is in parallel with the wall wart power circuit. Two diodes of 1N4001, having a forward voltage of 1V, are used to activate during a higher potential voltage. During wall wart supply voltage D1 will be forward biased (and turned on while D2 is turned off) allowing wall wart supply voltage of 13.5V (9V nominal during full loads) to enter the LM7805 regulator IC and also provide current of around 200mA (dropped across the 60 ohm resistor) to the positive plate of the 9V NiMH rechargeable battery to recharge the battery. 200mA is 30mA greater than the 170mAh needed however, no damage will occur and is used for higher expectations of limitations of the battery’s life span (after long use the battery will still except and maintain its charge). Equation (6) shows the amount of resistance needed to provide a maximum current (shown in figure 25) to the battery in order to fully charge the battery which is also used as a trickle charge if necessary to the battery in order to maintain a fully charged battery for maximum use, if portable. This design application allows the user full use of portability.

(6)

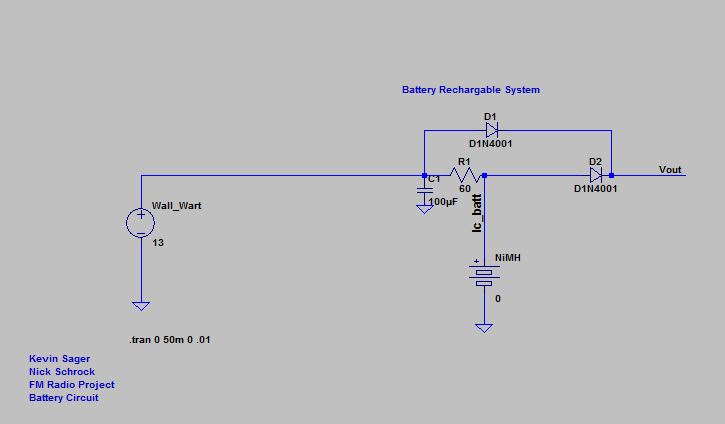


Figure 24. Rechargeable Battery Circuit

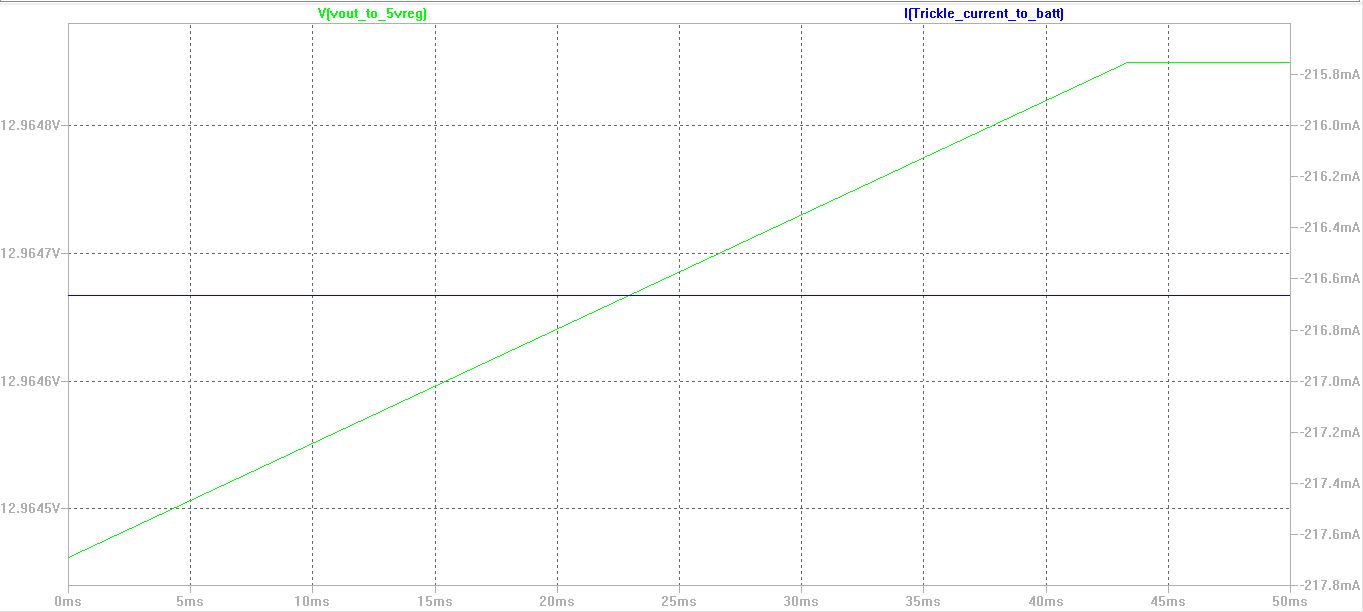
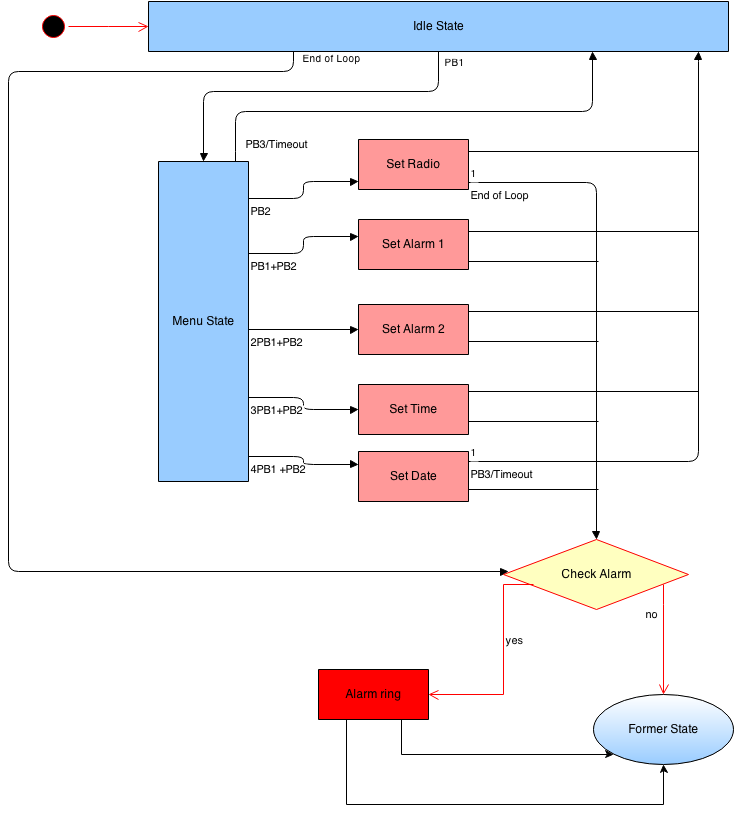


Figure 25. Rechargeable Battery Circuit (LT Spice Simulation) Constant Current Charge

## Software Specifications



**Figure 26:** Top Level System UML

**Top Level System Architecture:**

The broad design of the system software implements the concept of a finite state machine. The software is meant to be programmed once and then subsequently entered in to an infinite loop. Within in this loop are different states of operation. The user navigates through these states to customize the settings of the system using three push buttons.

**Idle State**

The system boots and defaults to the idle state which displays all pertinent information. It defaults to displaying the time and date, but will update its display based on the user configured settings. If the FM chip is receiving a signal, the radio station frequency along with an indicator of signal strength is displayed along with the time and date. If an alarm is set, a small bell will appear in the bottom left corner of the screen. If two alarms are set, two bells will appear. If the alarm is set to play music instead of sound an alarm, a small musical note will be displayed instead of the bell. The radio is turned on and off by pressing button 3. If button 2 is pressed while the on it will tune to the next preset in the preset cue. Presets are added in the set-radio state. The system defaults back to idle every time push-button 3 is pressed in any state. If the user is without activity for 60 seconds in any state, the system will time out and default back to the idle state.

**Menu State**

When the user hit button 1 the system is thrown into the main menu state. From this state the user has the ability to access any of the other states. Displayed on the screen is a list of different states that the user can enter in to. The user scrolls through these options by hitting button 1 repeatedly. If the user hits button 2, the system goes into the selected state. If the user hits button 3 or is inactive for 60 seconds, the system will default back to the idle state.

**Set Radio State**

The first option on the menu is the set radio state. Here the user can turn the radio on or off and can select a specific station to listen to by pressing button 1. The user can also seek through radio stations at certain threshold signal strengths by pressing button 2. The user can also set up to 5 presets by holding down button 2.

**Set-Alarm States**

Set-Alarm 1 and Set-Alarm 2 are identical states that set alarms at whatever time the user inputs. When the system is in this state, it displays a flashing time (just hour and minute). The user can increment the hour by pushing button 1 and the minute by button 2. The seconds are then defaulted to 0. If the user hits button 3 or is inactive for 60 seconds, the system will default back to the idle state. However, if button three is held, it will toggle the ring between turning the radio on or sounding the alarm.

**Set Time and Date States**

These states have an identical user interface to the Set Alarm states; however they are used to update the current time and the current date respectively to whatever the user enters. The same screen and button sequence is used for the set time state while a flashing screen with a month day and year is shown for the set date state.

**Alarm Check/Ring**

In any of these states there is the potential for the alarm to be set off if the current time matches the time that the alarm is set to, and the alarm is activated using the slide switch. At the end of every loop in every state the check\_alarm() function is called and the system verifies that the alarm should not be activated. In the event that the alarm is activated, the system goes into the alarm ring state. The alarm ring state can be exited by either hitting button 1 to silence the alarm and return to default or hitting button 2 to snooze the alarm for a ten minute period and return to default. If no user action is taken the alarm ring state will be exited after 60 seconds and the system will default back to idle.

### Software Design

**ACR\_Main.c**

This is the main module for the entire system. The following functions are executed in the order they are described. Subsequently, the program is thrown into an infinite loop that runs the state machine. The steps taken to initialize the state machine are all completed prior to entrance into the infinite loop.

* void timer0\_init(void)

Timer 0 is an 8-bit timer peripheral on the ATmega328P. This application uses Timer 0 as counter to implement a hardware switch debounce. This function initializes Timer 0 in CTC mode at a clk/1024 prescaler. The current implementation checks for a debounced button press by calling the Timer 0 Interrupt Service Routine every 5ms. OCR0A is set to 78 cycles to generate the 5ms increments.

* void timer1\_init(void)

Timer1 is a 16-bit timer on the ATmega328P. Its use in this application is to generate a PWM waveform output that is used to drive the solenoid alarm. OCR1A is initialized to 0 because the alarm is initially silenced.

* void timer2\_init(void)

Timer 2 is another 8-bit timer. In this application timer2 is initialized to generate a PWM waveform at a clk/64 prescaler. It is used to drive the backlighting on the LED in response to the ADC output from the photocell resistivity circuit that responds to the natural lighting.

* void WDT\_Init(void)

This function initializes the Watchdog Timer that is built in to the ATmega328P. The watchdog timer is used to monitor periods of inactivity to generate user timeouts. It is initialized to generate an interrupt every 4 seconds.

* uint8\_t read\_alarm\_minute()

This is a short function used to read the value of the alarm minute from EEPROM.

alarm\_minute = eeprom\_read\_byte (( uint8\_t \*) ALARM\_MINUTE\_ADDR) ;

* uint8\_t read\_eeprom(int addr){

This function reads a byte from EEPROM.

byte = eeprom\_read\_byte ((uint8\_t\*)addr) ;

* void write\_eeprom(uint8\_t byte, int addr)

This function a byte to EEPROM

eeprom\_update\_byte (( uint8\_t\*)addr, byte );

* uint8\_t read\_radio\_station()

This function is reads the pre-set radio station from EEPROM . This is a separate function due to the radio stations needing to be stored in a two byte word.

* void write\_radio\_station(uint8\_t channel){

This writes the user entered radio station to a specific location in EEPROM

* void ADC\_init(void){

This function initializes the Analog to Digital Converter that is built in to the ATmega328P. The converter is initialized to free-run mode and is used to convert the analog signal input from the photocell circuit. The conversion is thereafter used to modify the PWM from timer2 and control the backlighting of the LED as per **Req. 1a**

* void setup(void)

This is the main setup function for the entire system. It is the first thing called in the main function. The first step in the setup function is to define each pin as either an input or an output. Once that is completed, the GPIO pins that are used as button inputs are pulled high. Then every \_init function is called not just from the main module, but from all of the different driver modules. The last step is to set the time to default at 2:00.

* ISR(TIMER0\_COMPA\_vect)

This is the Interrupt Service Routine that gets called when Timer 0 hits a compare match. The basic function of this ISR is to check if a button has been pressed.

b1\_state = (b1\_state<<1) | (PIND & (1<<PD6)) >> 6 | 0xfc00;

b2\_state = (b2\_state<<1) | (PIND & (1<<PD5)) >> 5 | 0xfc00;

The variables b1\_state and b2\_state represent the duration of time in which the respective buttons are pulled low. For this implementation, a requirement of 10 consecutive low values must be read for the button state variable to compare successfully. Once this happens, either the button 1 or button 2 global flag variable is set. This information is frequently used in later functions To implement button holds, a state is kept track of to tell whether the button is continuously being held down.

b3\_hold\_state = (b3\_state<<1) | (PIND & (1<<PD6)) >> 6 | 0xfc00;

b2\_hold\_state = (b2\_state<<1) | (PIND & (1<<PD5)) >> 5 | 0xfc00;

b1\_hold\_state = (b1\_state<<1) | (PIND & (1<<PD7)) >> 7 | 0xfc00;

the duration of the hold is kept track of using a count variable. This is how button holds are implemented later in the program.

* uint8\_t get\_conversion(void)

This is a short function that pulls the value of the ADCH register from the Analog to Digital Converter. This value is used to determine the voltage output of the photocell resistivity circuit

* ISR(WDT\_vect)

This is the Interrupt Service Routine called every time the watchdog timer hits its peak. The only thing happening here is the incrementing of a timeout variable used in various states to return to idle

* void backlight\_adjust()

The backlight adjust function takes the value from the get\_conversion function and does the correct math to convert it into a duty cycle percentage. This percentage is then used to update the value OCR2B which in turn adjusts the backlight on the LCD screen per **Req 1a.**

* void one\_sec\_delay(void)

Simple Function that uses the built in delay function to delay one second

* void reset\_variables()

This function is used to clear all the button flags and counts. Along with that it also clears the LCD screen and resets all pre-existing variables on the RTC.

* void set\_alarm\_state()

This is the fundamental loop function for the setting alarm state. This code loops through and updates what time is displaying on the screen while flashing the time. The user updates the screen by hitting either of the two buttons. After a period of inactivity, the loop times out per the watchdog timer (**Req 4a).** Outside of the loop, the alarm is set with the time displayed on the screen and the variables are stored in EEPROM memory. The state then reverts back to Idle. This state is shared by each alarm in the system. Logic is used to delineate which alarm is set and which alarm is ringing.

* void check\_Alarm()

The check\_Alarm function is called during every loop in every state. This function calls another function to sound the alarm if the alarm\_check function in the Alarm\_Driver module returns true.

* void idle\_state()

The idle state is the home state for the finite state machine. This is the state that is initialized on boot. In this state a large clock and small date are displayed on the screen (**Req 1b)**. If the alarm is on then a small bell is displayed in the bottom left corner of the screen. If the alarm is set to turn on the radio, a small musical note is displayed instead of a bell. Each alarm has a separate icon with alarm 1 coming first and alarm 2 coming second. The idle state also displays the radio station (**Req 1c)** if the radio is on. A small icon next to the radio station indicates the signal strength of the station (**Req 1c)**.

* void menu\_state()

By hitting button 1 in the idle state, the system goes to the menu state **(Req 1c)**. The menu state displays a list of other states that can be selected. There are 5 total states that can be scrolled through by pressing button 1 and selected by pressing button 2. The menu state is timed out by the watchdog timer as per **Req 4a.**

* void set\_time\_state()

When the set time state is selected the user can change the current time displayed on the clock. The functionality is the same as the set alarm state, however different variables are updated outside of the loop. These variables are then sent to the DS1407 RTC via I2C communication

* void set\_date\_state(){

The set date state has the same functionality as the set time state. The difference is that it changes the date variables rather than the timer variables.

* void set\_radio\_state()

The set radio state employs many different capabilities of the SI4703 FM receiver. Depending on the users specification, the peripheral can be tuned to a specific frequency and set to output an audio signal on the speakers. The user can also seek through the different stations that provide a strong signal. If the user holds down the push button, they have the ability to save the selected channel to a specific location in EEPROM memory as a preset.

* int main(void)

This is the function that is executed by the compiler. Everything must be called from this function. Ideally this function remains as small and simple as possible. The main architecture for this state machine relies on a switch statement that is used to update the state. A global variable called “State” is used to determine the case of the switch statement. From there the different states are called. With this implementation all that has to happen for the state to change is a simple update of the State variable. This can be done in any state and the state will update as long as the main loop is allowed to continue executing. An example of the main switch statement is shown below.

switch(State)

{

case 'I':

idle\_state();

break;

case 'M':

menu\_state();

break;

case 'A':

set\_alarm\_state();

break;

case 'T':

set\_time\_state();

break;

case 'R':

set\_radio\_state();

break;

case 'D':

set\_date\_state();

break;

}

Prior to the execution of the switch statement the setup function is called. Subsequently there is a check to see if button 1 has been pressed. If so the state changes to the menu state. All the other states on the FSM are accessed from the menu state.

**I2C\_Driver.c**

* This is the driver module for the two wire I2C communication protocol. The functions in this module are used by the microcontroller to send and receive data from the SI4703 FM receiver and the DS1407 Real-Time clock. This module is vital for the overall system because it enables communication between the microcontroller and the various peripheral devices.
* void I2C\_WriteRegister(uint8\_t device\_address, uint8\_t deviceRegister, uint8\_t data)

Takes arguments device address, register, and data to be written and stores it in the data register. Using TWI protocol sends the data to the slave device and waits for an acknowledgement. This is how data is written to a slave.

* uint8\_t I2C\_ReadRegister(uint8\_t device\_address, uint8\_t deviceRegister)

Function communicates with a specific register on a specific device. Takes the data and stores it in the TWI data register. Sends an acknowledge bit if still reading data. Stops reading from slave when no acknowledge bit is sent.

* void I2C\_Init();

Initializes I2C communication with various devices. This configures the I2C communication protocol so that the microcontroller can talk to the SI4703 and the DS1407 via I2C.

* uint8\_t I2C\_Start();

Sets the TWI enable bit and waits for data be either sent or received on the SDA lines. The SCK clock is started.

* uint8\_t I2C\_SendAddress(uint8\_t address);

Sends the address of the device to ensure communication is being done with the correct device. If the address matches, the device will acknowledge that communication has started.

* uint8\_t I2C\_Write (uint8\_t data)

Function that actually writes the data to the TWI data register and sends it to the slave. Waits for an acknowledgement from the slave when the transaction has finished.

* void I2C\_Stop();

Stops TWI communication by clearing the Enable bit

* uint8\_t I2C\_ReadNACK()

Reads a register from a slave but does not send an acknowledge bit back. This is to stop reading data from the slave.

* uint16\_t I2C\_ReadRegister\_16bit()

Does the same thing as the 8 bit function but does two reads back to back and merges the data into a 16-bit word.

* void I2C\_WriteRegister\_16bit(uint16\_t data);

Same functionality as the 16-bit read but for writing to a slave. These functions are used for communicating with the since it contains 16 bit registers.

**SPI\_Driver.c**

* This is a short module that is called to initialize the SPI protocol and send data. It is relatively short because a lot of the work is done by the only SPI device in the system (the LCD Display).
* void SPI\_Master\_Init(void)

This initializes the SPI protocol in master out slave in mode by setting specific bits in the SPI control register.

SPCR |= \_BV(SPE) | \_BV(MSTR) | \_BV(SPR0);

This enables SPI and sets it into master mode since there is only one way communication between the LCD display and the microcontroller.

* void SPI\_Master\_Send(unsigned char data)

The only thing this function does is load the SPI data register with the argument passed and wait for the acknowledgement that the data was sent to the device. It is called from the LCD\_Driver module.

**LCD\_Driver.c**

* This module is the primary driver for the LCD Display screen. These functions are used and called in various parts of the system to draw the correct images on to the LCD display. The functions can get fairly intricate but their functionality is mostly self-explanatory.
* void LCD\_Init(void)

Initializes the LCD by toggling pins and writing commands to the LCD control registers. This sets up the LCD into high contrast mode which is ideal for the display.

* void LCD\_WriteCommand(unsigned char command)

Function to write setup commands to the LCD control register

* void LCD\_WriteData(unsigned char data)

Writes an 8-bit “bar” of data onto the LCD screen. In vertical aligned mode the bar goes from top to bottom. This bar enables pixels wherever the cursor is located.

* void LCD\_GoTo(uint8\_t x, uint8\_t y)

Places the cursor at a specific location on the screen.

* void LCD\_Clear(void)

Clears the LCD screen entirely by looping through and clearing each individual pixel.

* void LCD\_Update( void )

Refreshes the LCD screen with any data that might have been written to the buffer but not actually drawn on the screen.

* void LCD\_DrawSplash(void)

Draws a fun screen that is configurable in code. The splash is stored as a variable in EEPROM.

* void LCD\_SetPixel(uint8\_t x, uint8\_t y)

Sets one individual pixel at a certain location.

* void LCD\_ClearPixel(uint8\_t x, uint8\_t y)

Clears one individual pixel at a certain location.

* void LCD\_WriteChar (unsigned char ch)

Writes a character to the screen. Loops through and updates the buffer then proceeds to loop and call LCD\_WriteData() on the updated buffer as shown in the code below.

for(j=0; j<7; j++){

LCD\_WriteData(lcd\_buffer[current\_row][current\_col++]);

}

* void LCD\_WriteString(const char \*string)

Loops throught the LCD\_WriteChar() function to draw a string of data on the screen.

* void LCD\_DisplaySmallInt(uint8\_t i)

Displays an integer in small font on the screen. Called in LCD\_WriteInt().

* void LCD\_WriteInt(uint8\_t i, bool large)

This function is used to write variables of type int to the screen. It is called frequently as much of the data that is written to the screen are integers. The function takes arguments for the data and whether it is large or small font.

if(large){

if(second\_digit == 0)

LCD\_ClearLargeSpace();

LCD\_DisplayLargeInt(first\_digit);

LCD\_GoTo(current\_col+13, current\_row);

LCD\_ClearLargeSpace();

LCD\_DisplayLargeInt(second\_digit);

}

else{

if(second\_digit == 0)

LCD\_ClearSmallSpace();

LCD\_DisplaySmallInt(first\_digit);

LCD\_ClearSmallSpace();

LCD\_DisplaySmallInt(second\_digit);

}

This part of the function is executed if the integer is over 9. The digitAtPos() function is called prior to this to separate the number into two digits.

* void LCD\_DisplayLargeInt(uint8\_t num)

Displays a Large integer to the screen. Function called in Write Int function.

* void LCD\_ClearSmallSpace(void)

Clears the data from a small font size space on the screen. Used when changing the seconds on the clock display.

* int digitAtPos(int number, int pos);

Function called in the LCD\_Write\_Int() function that splits a two digit number into two distinct digits. The method to do so is shown below.

return ( number / (int)pow(10.0, pos-1) ) % 10;

* void LCD\_ClearLargeSpace(void);

Clears the data from a large font size space on the screen. Used when incrementing the minutes and hours numbers on the clock display.

* void LCD\_DrawBell(int x, int y);

Draws a small bell at a certain location on the screen. Called when either alarm is set.

* void LCD\_WriteColon();

Writes a large colon to the screen. Used to display the large clock.

* void LCD\_InvertRow();

Reverses the polarity on the pixels for an entire row of data. This is used to scroll through the menu bar to make a selection. The method uses an exclusive OR operator and is shown below.

for(int i=0; i<84; i++)

lcd\_buffer[row][i] ^= 0xFF;

* void LCD\_DrawNote(int x, int y)

Draws a 9x8 icon of a musical note on the screen. Used when an alarm is set to turn on the radio rather than activate the solenoid.

* void LCD\_DrawBar(int y, int x, uint8\_t height)

Draws a bar with a specific height on the screen. Used to display signal strength when the radio is on in the idle state.

**RTC\_Driver.c**

* This module is dedicated to controlling the functions of the DS1407 Real-Time Clock IC. The primary functions used in the main state are the function that displays the large clock. This is the primary display for the idle state so it must be accurate.
* uint8\_t DecimalToBCD (uint8\_t decimalByte);

This function is called when data is being sent to the RTC. The RTC formats its data in Binary-coded Decimal format. In order to send the RTC the correct value, it must first be converted into BCD format.

* uint8\_t BCDToDecimal (uint8\_t bcd)

When data is received from the RTC, it is in BCD format. This function converts the Binary-coded Decimal back into standard decimal format so that it can be used throughout the system.

* void set\_Time(uint8\_t hours, uint8\_t minutes, uint8\_t seconds)

This function sends the hours, minutes, and seconds arguments to the RTC via I2C communication. The data is first converted to decimal format before it is sent to the RTC.

* uint8\_t get\_Hours()

Function to retrieve the hours byte from the RTC.

* uint8\_t get\_Minutes();

Function to retrieve the minutes byte from the RTC.

* uint8\_t get\_Seconds();

Function to retrieve the seconds byte from the RTC.

* void RTC\_DisplayLargeClock(int xpos, int ypos)

This function gets called repeatedly in the idle state. The primary function is to display the hours and minutes in large font and the seconds in small font. Since this function is constantly updated, there is a fair amount of logic to make sure the correct values are being refreshed and written in the correct spot.

switch(update)

{

case 'H':

//Write Hours to screen

if(displayed\_hours>9){

hours\_pos2 = hours\_pos1;

}

LCD\_Clear();

LCD\_GoTo(hours\_pos2, ypos);

LCD\_WriteInt(displayed\_hours,true);

LCD\_GoTo(colon\_pos, ypos);

LCD\_WriteColon();

//Write Minutes to Screen

LCD\_GoTo(minutes\_pos1,ypos);

if(displayed\_minutes<10){

LCD\_WriteInt(0,true);

LCD\_GoTo(minutes\_pos2,ypos);

}

LCD\_WriteInt(displayed\_minutes,true);

//Write Seconds to Screen

LCD\_GoTo(70,1);

if(displayed\_seconds<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(76,1);

}

LCD\_WriteInt(displayed\_seconds,false);

break;

case 'M':

//Write Minutes to Screen

LCD\_GoTo(minutes\_pos1,ypos);

if(displayed\_minutes<10){

LCD\_WriteInt(0,true);

LCD\_GoTo(minutes\_pos2,ypos);

}

LCD\_WriteInt(displayed\_minutes,true);

//Write Seconds to Screen

LCD\_GoTo(70,1);

if(displayed\_seconds<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(76,1);

}

LCD\_WriteInt(displayed\_seconds,false);

break;

case 'S':

//Write Seconds to Screen

LCD\_GoTo(70,1);

if(displayed\_seconds<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(76,1);

}

LCD\_WriteInt(displayed\_seconds,false);

case 'N':

break;

}

This is the switch case used within the function to make sure that only the correct values are being updated. If all three positions need to be updated, the case is H, if two, M, if just seconds the case is S. The cases are determined by keeping track of the variables that are displayed on the screen and comparing them with the values that are received from the RTC. This implementation creates a seamless update of the displayed clock at the correct time.

* void RTC\_DisplaySmallClock(int xpos, int ypos);

This is a function used to display a small clock on the screen. The same method is used as in displaying the large clock however the LCD\_WriteInt functions take a false size argument.

* void RTC\_DisplayDate(int xpos, int ypos);

Displays a small date on the screen using the method previously explained.

* void RTC\_ClearVariables();

Clears the variables that are kept for indicating what is displayed on the screen. This is necessary to call whenever a state change occurs. If it is not called only the seconds will show up on the screen because the minutes and hours will still register as their old values unless it is the turn of the minute or the hour.

**FM\_Driver.c**

* This is the module that drives the SI4703 FM receiver IC. This code is yet to be completed and therefore requires a more in depth analysis than is currently given. The current code is for the . Due to a sudden design change, the SI4703 will now be implemented and the code will change. These are unwritten stub functions that will likely be used to control the FM receiver chip.
* void FM\_Init();

This function initializes the SI4703 FM receiver IC and sets it into I2C mode. This enables communication between the microcontroller and the SI4703. To set the SI4703 into I2C mode, the data line must be low upon reset. This is done in the following code sample.

//Clear SDIO and RESET

PORTC &= ~\_BV(SDA);

PORTD &= ~\_BV(RST);

\_delay\_ms(10);

//Set RESET

PORTD |= \_BV(RST);

* void FM\_ReadRegisters();

This function is responsible for receiving the data from the SI4703 and placing it into shadow an array of shadow registers that are used in other functions in the FM driver code. The SI4703 registers are read by starting at 0x0A and cycles through all 16 registers, returning to 0x09 as the final register read. The following code shows how the registers are read and placed into shadow register variables.

for(int i=0; i<16; i++){

if(i<6){

index = i+10;

}

else{

index = i-6;

}

registers[index] = I2C\_ReadRegister\_16bit();

}

TWCR = TW\_NACK;

while(!TW\_READY);

When the all the registers are read the control register is set to not ask for another bit and communication is thereby halted.

* void FM\_UpdateRegisters();

Registers 2 through 8 are written to by looping through and putting the data stored in the shadow registers into the data register for I2C communication. It writes to each 16-bit register and then sends a stop bit ending the communication.

* void FM\_Reset()

This is a utility function used to reset the registers back to the initial conditions. It is not used in the current implementation but is a useful function for debugging problems with I2C communication.

* void FM\_DisplayChannel(int channel, int xpos, int ypos);

This function displays a channel to the LCD screen. It is used to display to the user what channel is currently being tuned to.

* int FM\_TuneToChannel(int channel);

This function is used to tune the radio to a specific channel in which it is passed in integer form. For example 979 translates to 97.9. The function manipulates the CHANNEL register on the SI4703. The channel is masked in and the tune bit is set. After waiting until the chip is properly tuned, a signal is then outputted from the SI4703. This signal is amplified in hardware and sounds a radio station through the speakers.

* int FM\_GetChannel()

This short function returns the current channel that the radio is tuned to.

* void FM\_PowerDown()

This powers down the radio. It is used to toggle the radio on and off with user input.

registers[SYSCONFIG1] &= ~(1<<RDS);

registers[POWERCFG] |= \_BV(0) | \_BV(6);

Configuring the SYSCONFIG1 and POWERCFG registers to the correct values commences the power down the SI4703 without completely clearing the registers.

* bool FM\_GetRadioState()

returns weather the radio is on or off.

* uint8\_t FM\_GetSignalStrength()

Returns the signal strength of the radio pulled from the status register.

* void FM\_DisplaySignalStrength(int xpos, int ypos)

Displays the signal strength in the form of bars on the screen. 4 bars means clearest signal while zero bars means most distorted signal.

* void FM\_Seek(uint8\_t direction);

This function loops through radio stations in ascending order and stops on a station that has at least 2 bars of signal strength. The looping function is shown below

while(FM\_GetSignalStrength()<20){

current\_channel += 2;

if(current\_channel < MIN\_CHANNEL || current\_channel > MAX\_CHANNEL){

current\_channel = MIN\_CHANNEL;

}

FM\_DisplayChannel(current\_channel, 35, 3, true);

FM\_TuneToChannel(current\_channel);

}

**Alarm\_Driver.c**

* The primary function of this module is to factor out some of the code responsible for controlling the alarm system. This is a relatively short module used to set, sound, and silence the alarm.
* void alarm\_Set(int hours, int minutes, int seconds)

This function takes arguments and sets them as global variables in the alarm driver module.

* void alarm\_Sound(float volume);

Sounding the alarm is done by setting the value of OCR1A to control the duty cycle of the PWM output driving the solenoid alarm. The alarm goes off for 60 seconds before timing out.

* void alarm\_Silence(void)

Silencing the alarm is simply setting OCR1A to 0 so there is no PWM signal on the solenoid.

* void alarm\_Display(int hours, int minutes, int xpos, int ypos);

This displays hour and minute on to the screen. This function is also used in the set time state.

* int alarm\_Check()

This function returns one when the current time is equivalent to the values of the global alarm setting variables. This is called at the end of every loop in the entire system to monitor the alarm.

* void alarm\_Snooze()

When button 2 is pushed in the alarm ring state, the alarm will be silenced and set to ring ten minutes after its previous time. The original alarm time does not change due to the snooze function being activated.

if(alarm\_ring==1){

alarm\_minute += 10;

if(alarm\_minute>60){

alarm\_hour++;

alarm\_minute -=60;

}

}

This is executed for both alarms depending on which one is activated.

### Software Implementation

The software for this system was developed using Atmel Studio 6.0 IDE and Sublime Text Editor 3. Testing and implementation was done using UART communication between a PC hosting the development environment and the ATmega328p. The transfer of data was done on a universal serial bus line. Terra Term was also used to aid in the debugging of the system. Terra-Term provided a terminal that displayed output commands throughout the code that aided in the debugging process.

The software is written in C-code and uses the standard AVR GCC compiler. Included in the code is the standard AVR library. This interfaces with the ATmega328p and allows for a more elegant style of C.

# Validation

* **Req. 1a:** LCD Screen adjusts in environmental conditions.  The screen will dim in bright light conditions and intensify in low light conditions.

**Validation Plan:** A cover will be placed across the photocell sensing unit and the brightness of the LCD screen will be at full strength. The cover will then be removed and a bright light will be shown directly at the photocell. A distinct drop in brightness of the LCD backlighting will be expected to occur.

**Possible Risks:**

1. The photocell sensor could be obstructed by the shadow of the enclosure causing a disproportionate relationship between ambient light and backlight brightness.
2. The photocell circuit could potentially draw too much current and overheat the backlight LED’s causing them to burn out.

**Mitigation Plan:**

1. The photocell receptor will be designed to sense light coming directly from the top of the enclosure. In most situations this design will sample the ambient room light adequately.
2. The photocell circuit is current limited using forward biasing diodes. The software is also limited to duty cycles for the waveform between 10% and 90% so that there is not too current drawn from the LED’s.

* **Req. 1b:** MCU reads date and time from RTC and displays it correctly on LCD

**Validation Plan:** A thorough test of the Real time clock will be completed to set and reset the clock for different times of day. The behavior of the display will be observed at key turning points of the clock such as changes in minutes, hours, days, months, and years. The results will be tested and recorded.

**Possible Risks:** There could be a certain circumstance at an extremely specific time where the RTC does not update the display correctly.

**Mitigation Plan:** If this situation does occur and is truly outside of the scope of testing it will likely return to normalcy on the next refresh of the clock. This is because data is continuously being pulled from the real-time clock and displayed on the screen. Any software malfunction that slips through the testing procedure would quickly be corrected on the next update of the clock.

* **Req. 1c:** MCU reads frequency and signal strength from FM receiver and displays it correctly on LCD

**Validation Plan:** The output received from the FM chip will be compared to a verified radio receiver. If the information matches the data will be tested and recorded. The FM frequency should match the radio station that is being played through the speakers. The signal strength indicator will be verified by the quality of sound that is output from the speakers.

**Possible Risks:** Radio could be playing the incorrect station at any certain frequency

**Mitigation Plan:** The signal coming from the FM chip will be compared with signals from a third party radio to verify that each station input is the correct radio station.

* **Req. 1d:** Menu will project each individual state one at a time, one push button lets user change the display and another push button scrolls through menu.

**Validation Plan:** The menu state will be tested by scrolling through each option using button 1. Once it is verified that every item can be scrolled through, each state will be entered in to by selecting the state using button 1 and entering it using button 2.

* **Req. 2a:** Gain from operational amplifier controls voltage output according to adjustable gain

**Validation Plan:** The potentiometer on the external part of the enclosure will be adjusted and the loudness of the audio signal will be measured. The gain is directly related to the volume of the signal.

**Possible Risks:** The volume can be controlled in software as well as hardware so during testing it must be verified that the two are completely independent of one another.

**Mitigation Plan:** The volume is software will be set at a specific level allowing for an appropriate range of volume to be adjusted by the external potentiometer. Therefore the only shift is volume will be from the adjustment of the gain of the amplifier circuit using the external hardware.

* **Req. 2b:** System receives correct power supply input.

**Validation Plan:** The power supply circuit will be verified before the final unit is assembled. After assembly the system power will be monitored by an internal resettable fuse. In the event where power levels exceed the limitations of the device, the power fuse will disconnect power from the device keeping all of the components safe.

**Possible Risks:** In specific situations in certain parts of the design, there could be a surge of current that could go untested.

**Mitigation Plan:** Any unexpected imbalance of voltage and current that could happen outside of the initial resettable fuse is considered in the design. The combination of biasing diodes and resistors in the hardware design make the system virtually indestructible in regards to how the power is distributed.

* **Req. 2c:** Battery will provide energy when unit is disconnected from wall source power supply.

**Validation Plan:** The unit will be disconnected from the wall and all functionality will be tested and confirmed to continue to work normally.

**Possible Risks:** In low battery situations brown-out conditions could occur which could drastically affect the functionality of the user interface.

**Mitigation Plan:** A lower limit on battery output will be verified and when the battery reaches its lower limit, the system will shut down. The system will also be designed so that any brown-out condition will not harm any of the components in the system.

* **Req. 2d:** Adjustable sound given by speakers controlled by potentiometer input.

**Validation Plan:** The potentiometer on the external part of the enclosure will be adjusted and the loudness of the audio signal will be measured. The gain is directly related to the volume of the signal.

**Possible Risks:** The volume can be controlled in software as well as hardware so during testing it must be verified that the two are completely independent of one another.

**Mitigation Plan:** The volume is software will be set at a specific level allowing for an appropriate range of volume to be adjusted by the external potentiometer. Therefore the only shift is volume will be from the adjustment of the gain of the amplifier circuit using the external hardware.

* **Req. 3a:** After alarm ring, snooze button can be activated to reset alarm for a ten minute window.

**Validation Plan:** Alarm will be set to a specific time. That time will be recorded. When the alarm goes off, the snooze button will be hit and the alarm will be silenced. When the alarm goes off again, the time will be recorded and the difference between the two recorded times should be ten minutes.

**Possible Risks:** There could be a certain situation in which the change in date/time rollover could be at a point where addition of ten minutes to the alarm time does not work with expected functionality.

**Mitigation Plan:** Test cases will have to be applied for every scenario in which the time rolls over to a new value and the software will have to reflect these scenarios. It is essential that the snooze functionality works 100% of the time therefore many test cases will be recorded to cover all bases.

* **Req. 3b:** After alarm set, and alerts user, silence button will turn off alarm completely.

**Validation Plan:** When alarm is ringing the silence button will be pressed and the alarm will turn off. The time will be set for the next day and the alarm should go off again.

* **Req. 3c:** Slide switches enable user to set a specific alarm

**Validation Plan:** Each slide switch position will be set and the alarms should go off at their respective times. When alarm 1 is set, alarm two should not go off, and when alarm 2 is set, alarm 1 should not go off. When both alarms are set, they should both go off at their respective times.

**Possible Risks:**

1. Silencing one alarm could cause another alarm to not go off
2. The second alarm could go off while the first alarm is still ringing

**Mitigation Plan:**

1. Both alarms will share the same hardware so the silence mechanism will be shared. This means that both alarms work with the same functionality and do not affect each other in any way besides in the variables there time is stored in.
2. Unless the alarms are set for the exact same time, this will never occur since the alarms time out after 60 seconds. In the case where both alarms are set for the same time, there will be no difference than if only one alarm was set for that time since the alarms share the same hardware. This is not a problem because there is no practical application for two alarms to be set at the exact same time.

* **Req. 3d:** Pushbutton will allow user to escape from any given menu to change back to default menu.

**Validation Plan:** Button 3 will be pushed in every state and it will be recorded whether the system defaults back to the idle state.

* **Req. 3e:** LED will illuminate when radio tuner receives power.

**Validation Plan:** The LED power indicator light will be tested to illuminate when the system receives power from either the battery or the wall wart.

* **Req. 3f:** LED will illuminate when a strong signal is received.

**Validation Plan:** The LED status indicator light will be tested to illuminate when the system receives a clear signal on the FM receiver.

**Possible Risks:** The threshold for receiving a clear signal may vary based on external conditions. These conditions could change causing an inaccurate reading of sampled signal.

**Mitigation Plan:** The signal strength will be sampled periodically allowing for any necessary adjustment in the reading of signal strength.

* **Req 4a:** System tracks usage so LCD Screen defaults after a period of inactivity.

**Validation Plan:** The watchdog timer is implemented to track user inactivity. Every state will be tested to ensure that time outs occur after 60 seconds of inactivity and the screen defaults back to the idle state.

**Possible Risks:** Certain combinations of push buttons could either cancel each other out or could be ineffective in resetting the time out condition of the watchdog timer.

**Mitigation Plan:** The software is designed so that every push button generates a flag when set. These flags are checked in every loop in every state and if any of the flags are set, the watchdog timer will reset. This renders it impossible for the buttons to cancel each other out. To validate that all buttons are effective, every state will be tested with every push button.

* **Req 4b:** Each alarm is able to be programmed to a certain respective activation time. The user also has the option to pre-set the station for when the alarm goes off.

**Validation Plan:** Each slide switch position will be set and the alarms should go off at their respective times. When alarm 1 is set, alarm two should not go off, and when alarm 2 is set, alarm 1 should not go off. When both alarms are set, they should both go off at their respective times.

**Possible Risks:**

1. Silencing one alarm could cause another alarm to not go off
2. The second alarm could go off while the first alarm is still ringing

**Mitigation Plan:**

1. Both alarms will share the same hardware so the silence mechanism will be shared. This means that both alarms work with the same functionality and do not affect each other in any way besides in the variables there time is stored in.
2. Unless the alarms are set for the exact same time, this will never occur since the alarms time out after 60 seconds. In the case where both alarms are set for the same time, there will be no difference than if only one alarm was set for that time since the alarms share the same hardware. This is not a problem because there is no practical application for two alarms to be set at the exact same time.

* **Req 4c:** By one pushbutton, station will be chosen and another pushbutton will set the 1 through 5 station select channel for future use.

**Validation Plan:** When button 1 is held when selecting a certain radio frequency, that station will be saved in EEPROM as a preset. When choosing stations the user can choose to either hit one button to seek to stations with high signal strength, or hit button 2 to jump to a preset radio station. 5 presets will be stored in EEPROM

**Possible Risks:** What happens when more than 5 presets are attempted to be saved?

**Mitigation Plan:** Presets will be stored in an array and the location in that array will be incremented until it hits five in which case it will start back over at zero. This means that the when the sixth preset is saved, it takes the place of the first preset that was saved. The user must be able to keep track of the saved presets.

* **Req 5a:** A Printed Circuit Board houses all circuitry for entire system

**Validation Plan:** Printed circuit board will be ordered from a fab house and rendered based on the GERBER files generated from the EAGLE layout. The board will be soldered and extensively tested using all of the tests already noted above. The functionality of the printed circuit board is a key part in the success of any of the other dependent tests.

**Possible Risks:** The PCB could be incorrectly fabricated or soldered

**Mitigation Plan:** The PCB will be extensively tested and all the hardware circuits will be simulated, prototyped and tested before the PCB is fabricated. Safety measures will be taken when soldering the board together so that any mistake made by the technician can be easily corrected.

* **Req 5b:** An enclosure houses all internal parts of the system.

**Validation Plan:** After the system is built and verified, it will be mounted inside of a prefabricated enclosure that will house all of the internal circuitry. The LCD display and the user-interface push buttons, slide switch, and volume control potentiometer will be part of the external part of the enclosure.

**Possible Risks:** All of the components do not correctly fit and wont mount in a safe manner inside the enclosure

**Mitigation Plan:** Prior to the design of the enclosure, all dimensions will be verified and tested and the enclosure will be designed around the system prototype. This eliminates any possible error of having an enclosure that does not properly house the internal components.

# Conclusions

The purpose of the design is to implement a functional and useable alarm clock radio using the key elements of embedded systems design. The objective of the alarm clock radio design is to build and create a useful device that employs the main concepts of embedded system architecture. After thorough testing and validation, it can be concluded that the system design was successful. The initial requirements have sustained a few tweaks but the overall functionality of the project remains intact with the initial requirements and expectations. The prototyping of the design showed a completely functional system that met all of the specified requirements. Barring any major setbacks or design changes, this design will meet all of the requirements and expectations of the customer. Testing of the prototype was done thoroughly before the final product was produced and assembled. The final product was then built and run through a rigorous validation process and was found to meet all of the initial requirements within the given constraints. The project was built, verified, and validated to meet the needs of the consumer. Steps can now be taken to implement the design as a full-fledged product ready for market development. It can be concluded that the overall system design, prototyping, testing, and building of the finished product was a success.

# 5.0 Project Budget and Schedule

## 5.1 Budget

Table 1. Added Item Costs (Extra Features)



Table 2. Total Package (Cost Bottom Line)



## 5.2 Schedule

Table 1. Project Milestones

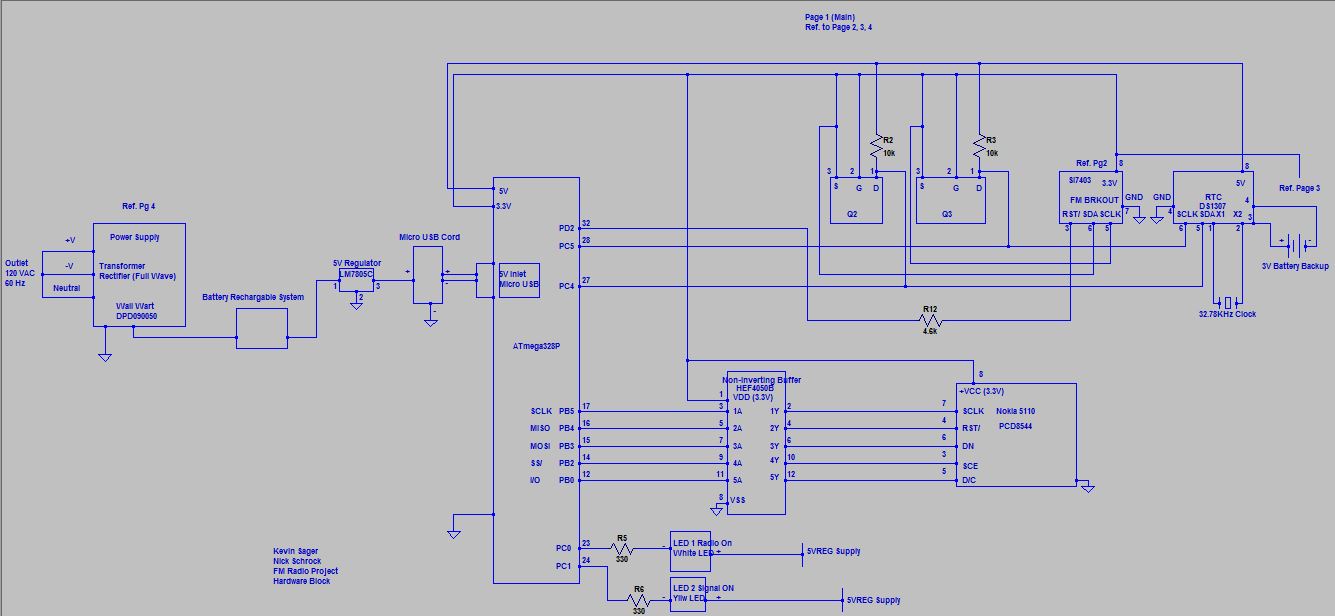


# 6.0 Revision History

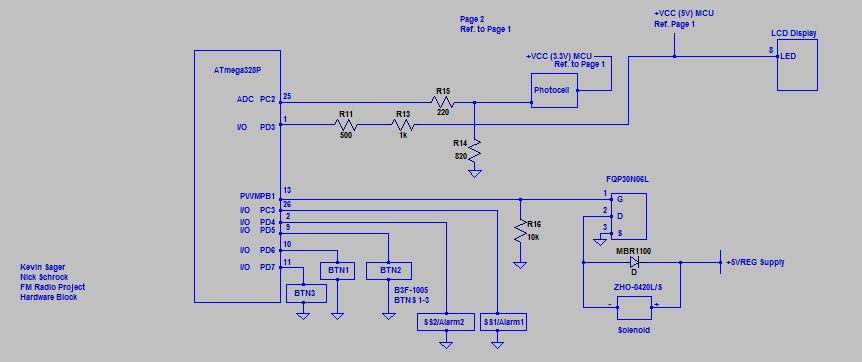
|  |  |  |  |
| --- | --- | --- | --- |
| Revision Number | Date | Author | Changes |
| Rev. 1.0 | 11-3-2014 | Kevin Sager  Nick Schrock | Documentation Designs |
| Rev. 2.0 | 11-28-2014 | Kevin Sager  Nick Schrock | 3.1 Hardware Specifications, 3.2 Software Specifications, 6.0 Schedule & Budget,  Appendix (A-D) |
|  |  |  |  |
|  |  |  |  |

# Appendix A – Electrical Schematics

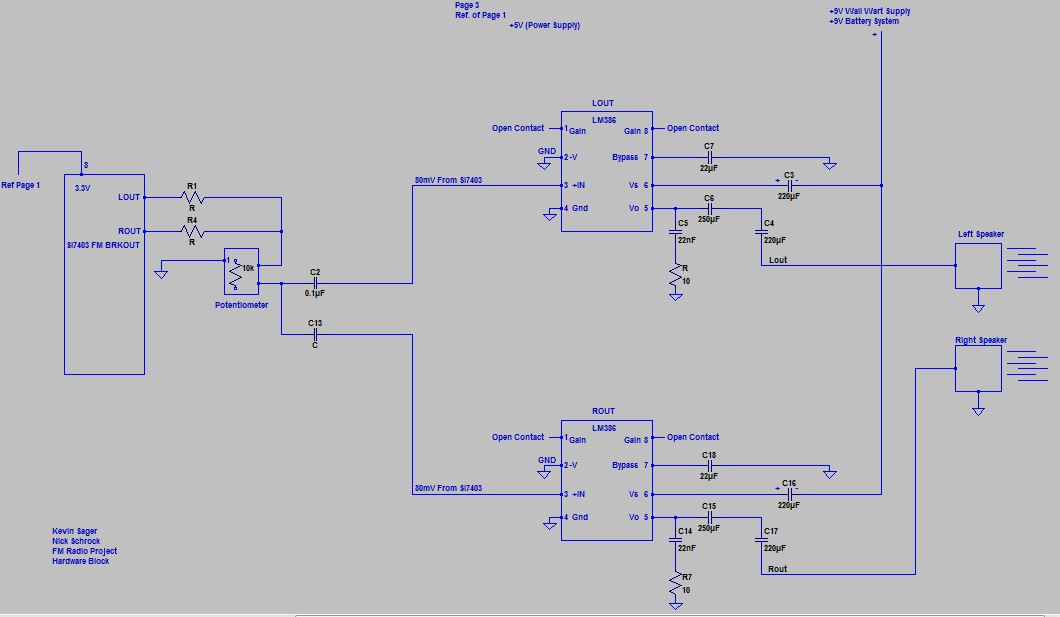
The electrical schematics in this appendix were used throughout the design process for simulation and/or designed circuitry applications. Some circuits do not reflect exactly those designed to use due to software simulation limits, although these circuits provide an application purpose for use.



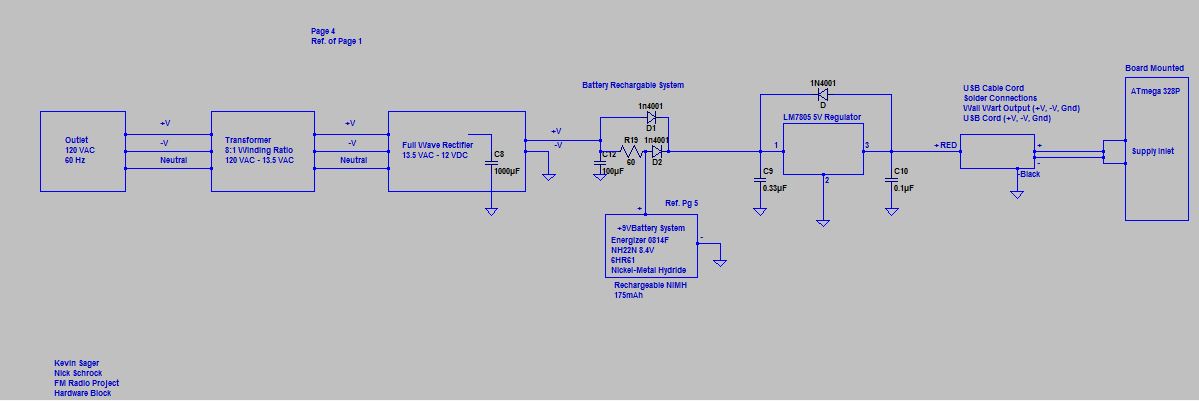
A-1 Overall Block Diagram Part 1



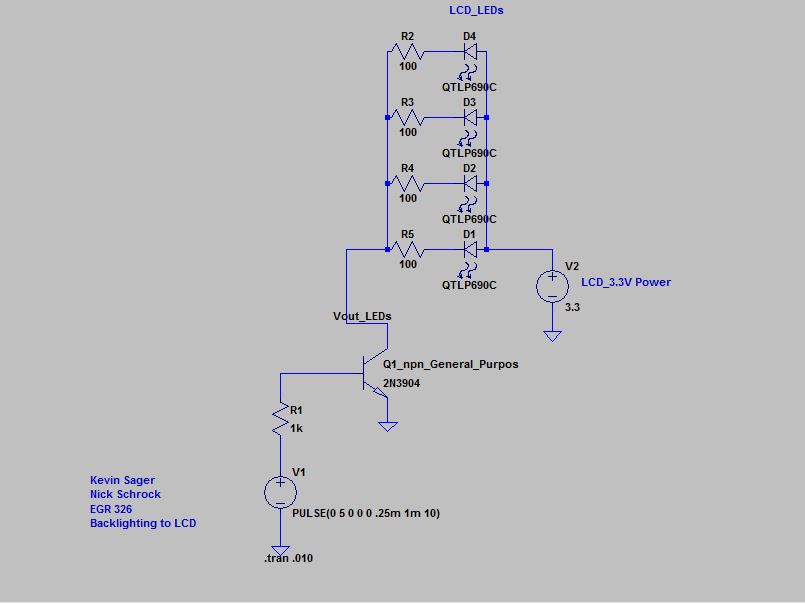
A-2 Overall Block Diagram Part 2



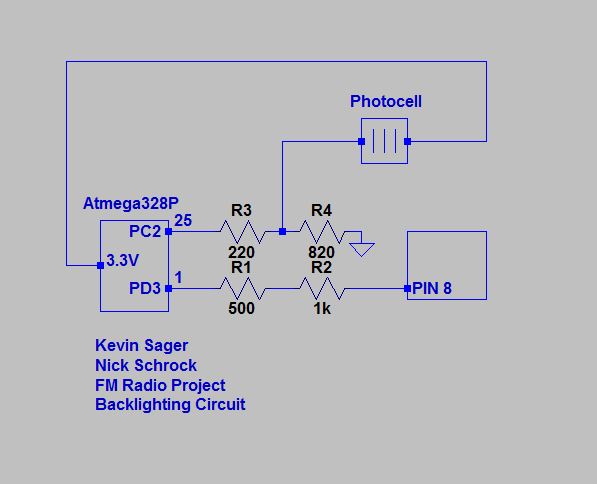
A-3 Overall Block Diagram Part 3



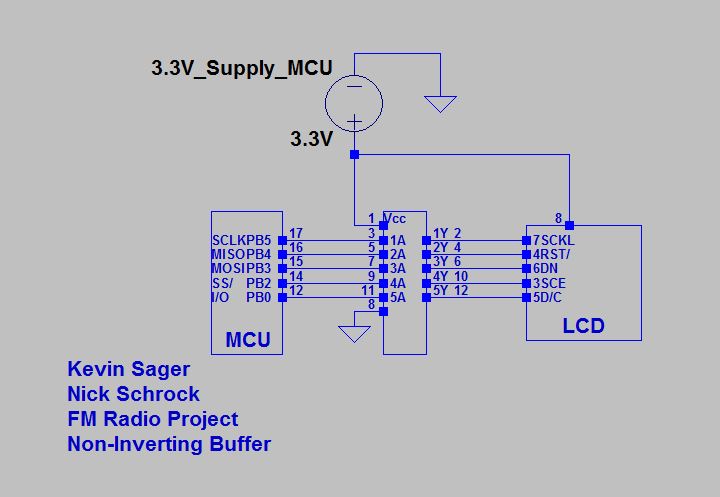
A-4 Overall Block Diagram Part 4



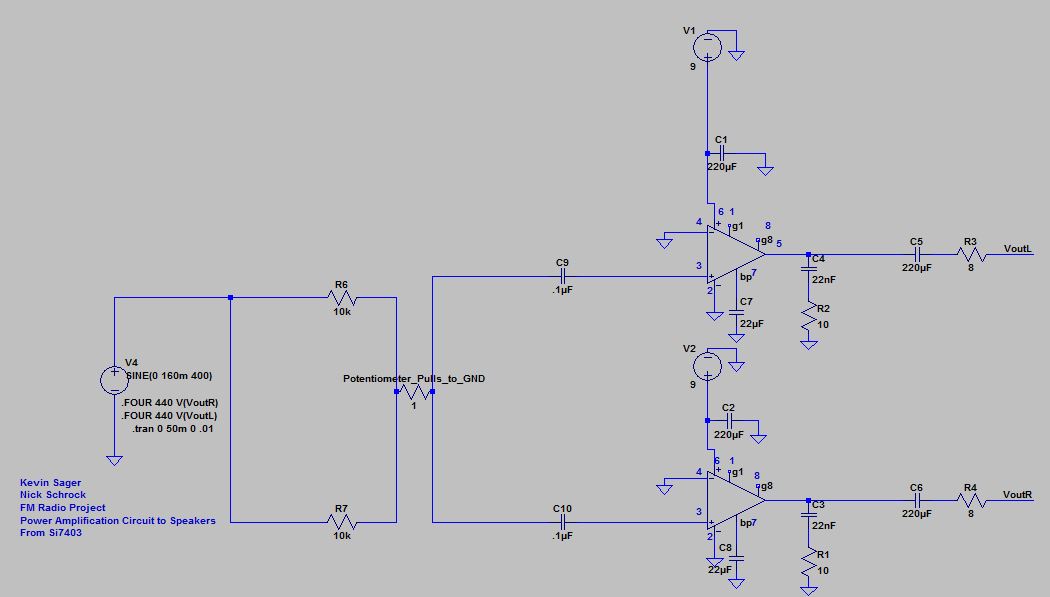
A-5 Back Lighting of LCD (For Simulation Purposes)



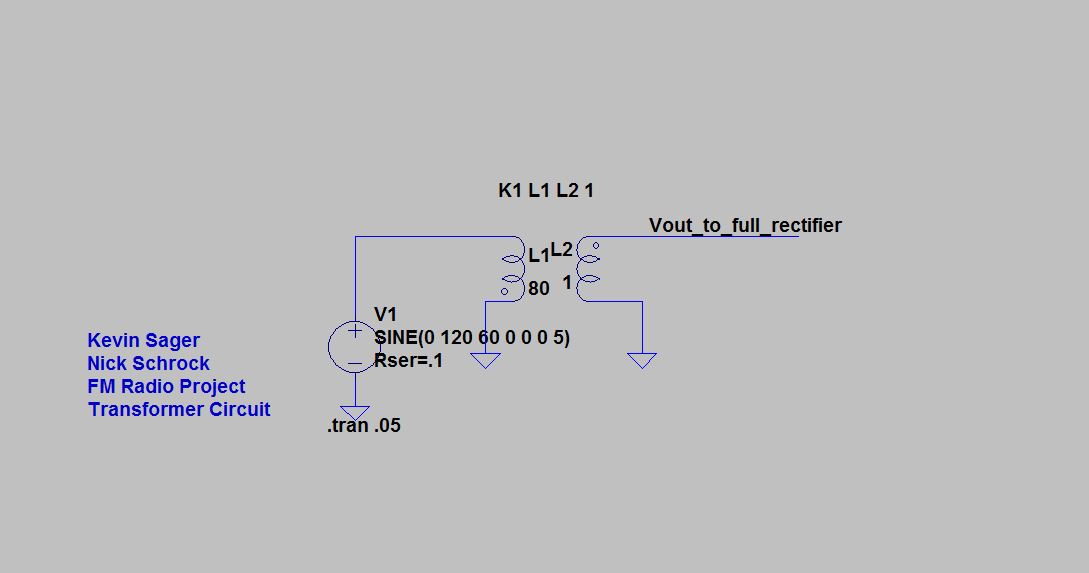
A-6 Backlighting Circuit Used in Design (MCU in PWM)



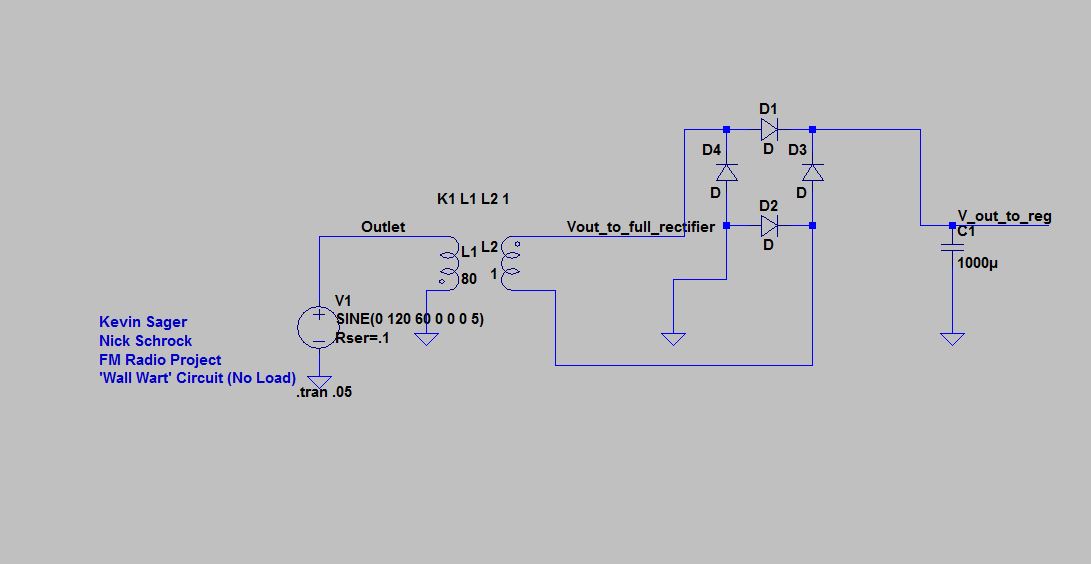
A-7 LCD Circuit (SPI Communication and 3.3V Logic Conversion with Non-Inverting Buffer)



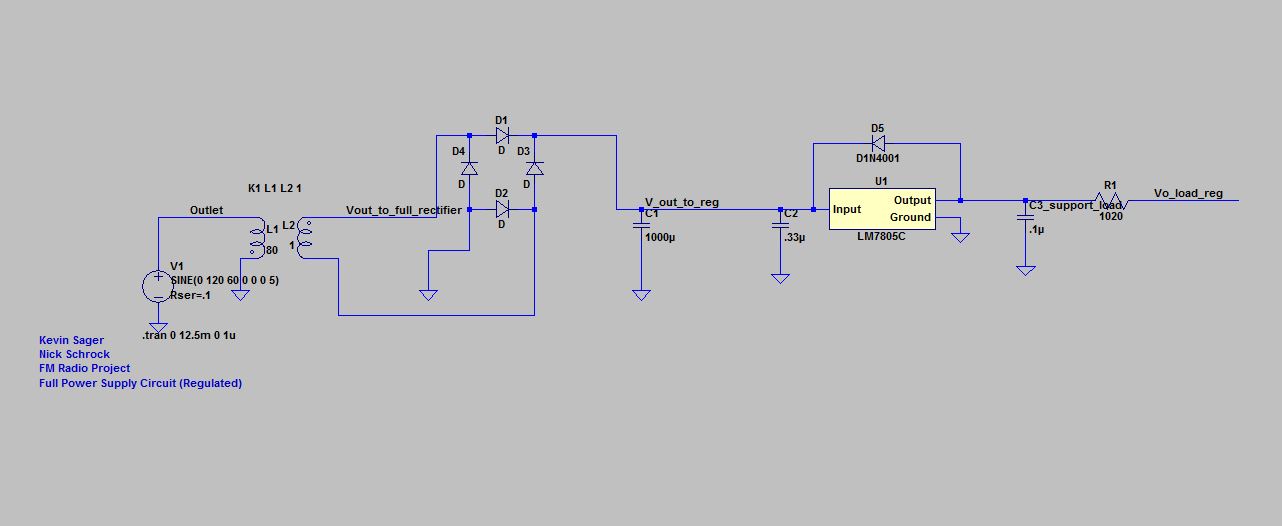
A-8 Power Amplification Circuit (Overall Circuit)



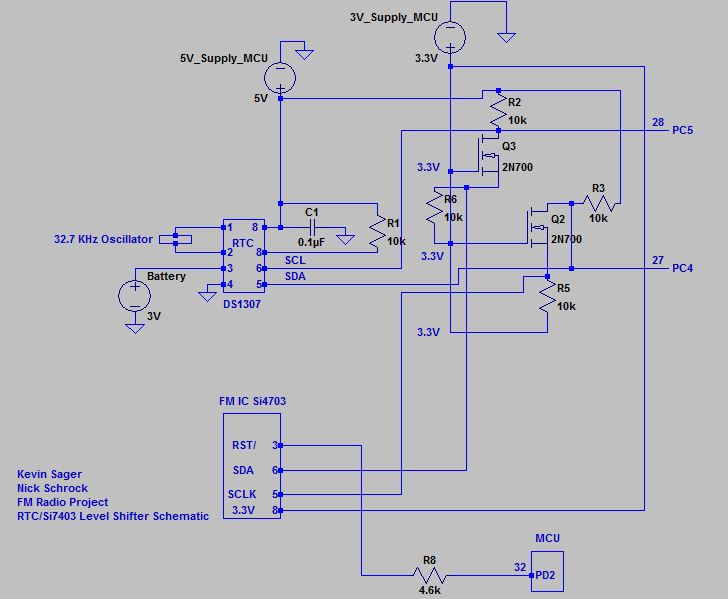
A-9 Step-Down Transformer



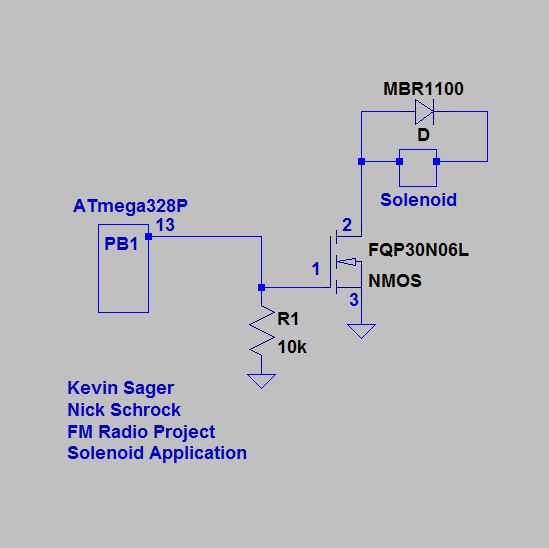
A-10 Full Wave Rectifier



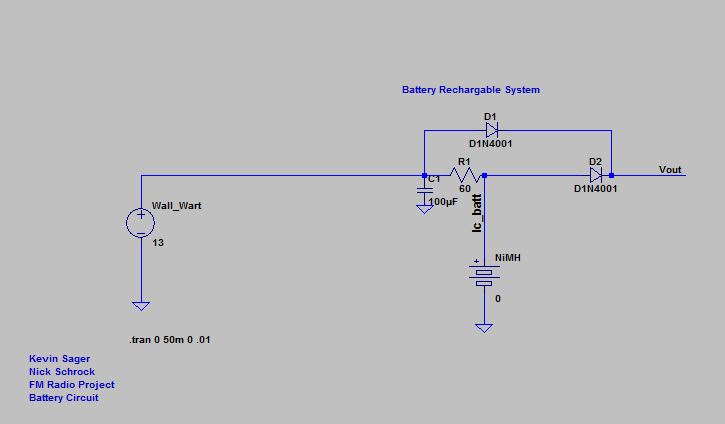
A-11 Full Power Supply (With Regulator)



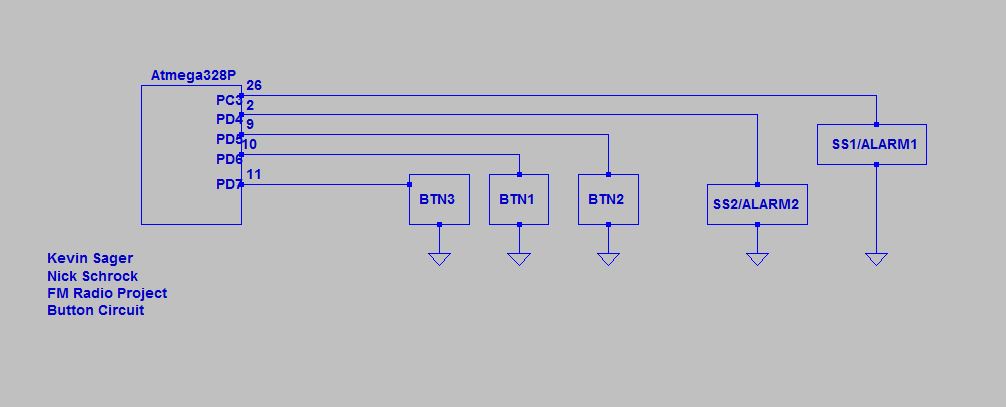
A-12 Level Shifter with Si4703, and Real Time Clock



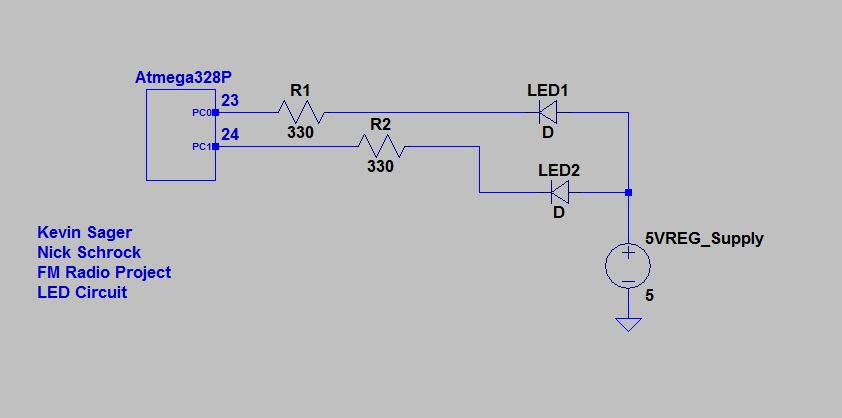
A-13 Solenoid Alarm Circuit



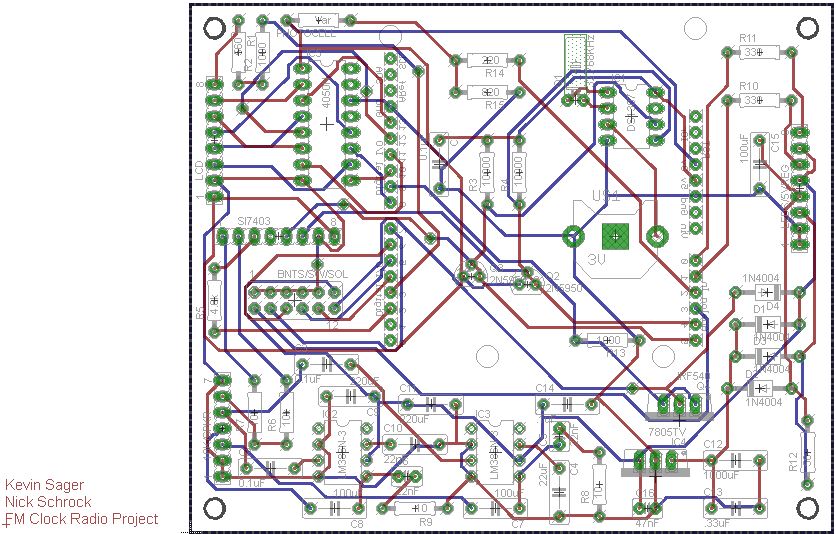
A-14 Rechargeable Battery Circuit



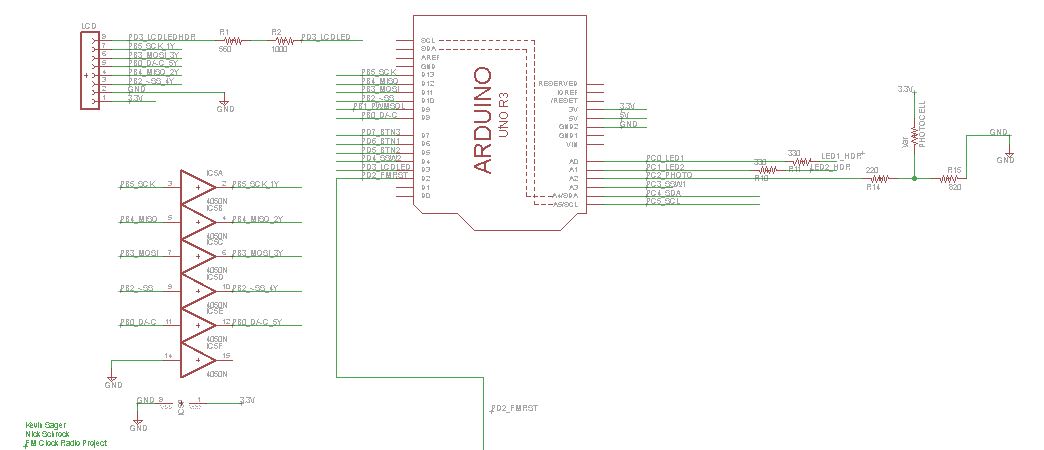
A-15 Circuit for Buttons/Slide Switches for User Interface



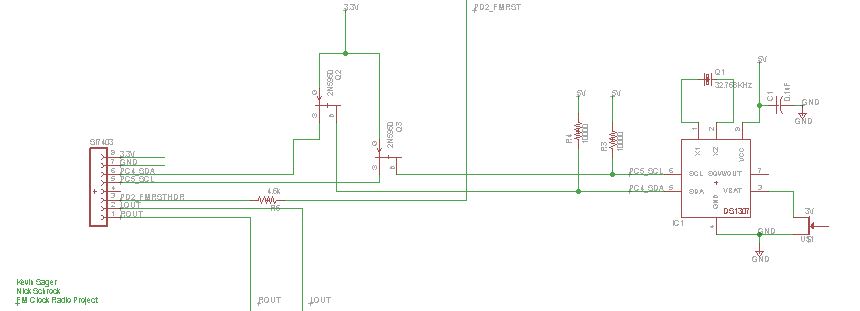
A-16 LED Circuit for Indication of Signal Received/Radio On



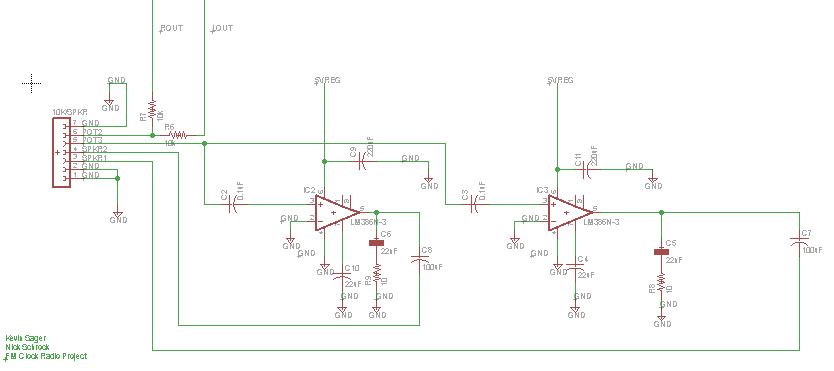
A-17 PCB Component Layout



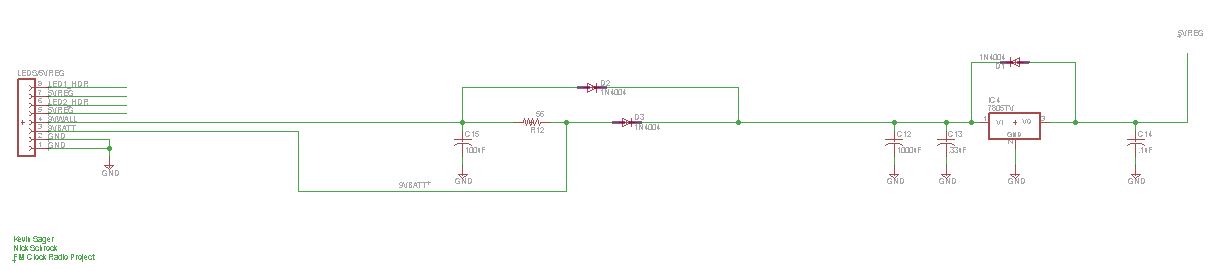
A-18 PCB Schematic: MCU, LCD Header, Non-Inverting Buffer (Page 1 of 5)



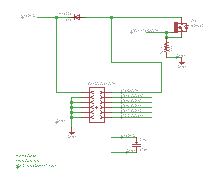
A-19 PCB Schematic: Si7403 Headers, Level Shifter, RTC (Page 2 of 5)



A-20 PCB Schematic: Speakers and 10k Potentiometer Headers, Amplifier Circuits (Page 3 of 5)



A-21 PCB Schematic: Power Routing Headers, Regulation Circuit (Page 4 of 5)



A-22 PCB Schematic: Push Button Headers, Solenoid Circuit (1 of 5)

# Appendix B – Bill of Materials

Table B-1 Total Components List (Cost)



Table B-2 Major Items (Cost)



Table B-3 Additional Features (Cost)



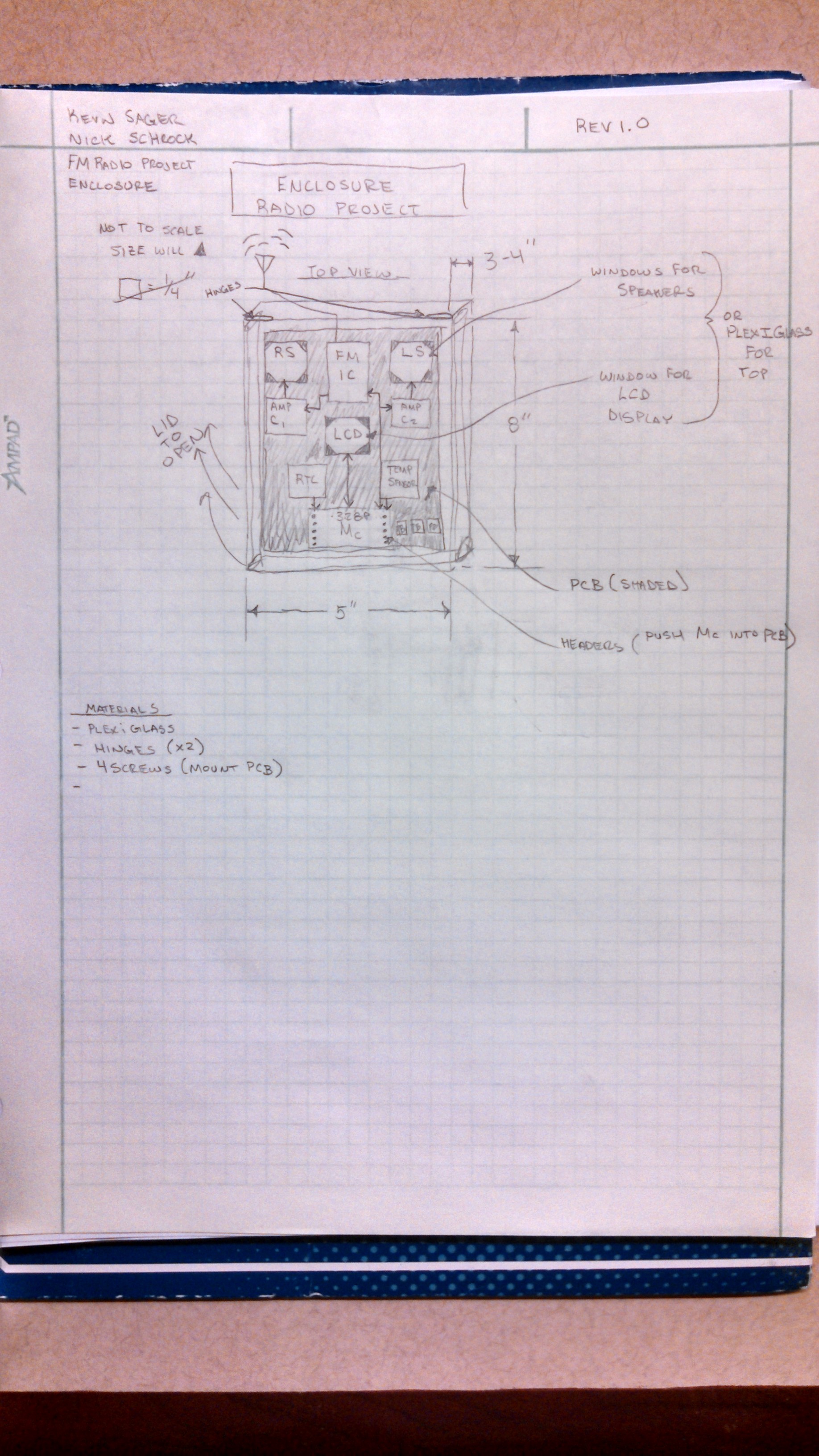
Table B-4 Major Items and Additional Features (Cost)



Table B-5 Total Package Cost (Bottom Line Cost)



# Appendix C – Mechanical Drawings



# Appendix D – Source Code

*/\**

*\*ACR\_Main.c*

*\**

*\* Created: 10/9/2014 1:28:37 PM*

*\* Author: Nick Schrock, Kevin Sager*

*\*/*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*This is the main module for the entire system. The execution of the code begins here.*

*The following functions are executed in the order they are described. Subsequently,*

*the program is thrown into an infinite loop that runs the state machine. The steps*

*taken to initialize the state machine are all completed prior to entrance into the*

*infinite loop.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <stdio.h>

#include "uart.h"

#include <string.h>

#include <avr/io.h>

#include "LCD\_Driver.h"

#include "SPI\_Driver.h"

#include "TWI\_Driver.h"

#include "RTC\_Driver.h"

#include "Alarm\_Driver.h"

#include "avr/interrupt.h"

#include "FM\_Driver.h"

#define F\_CPU 16000000UL

#define BAUD 9600

#define MYUBRR F\_CPU/16/BAUD-1

*//Alarm Addresses*

#define ALARM\_HOUR\_ADDR 46

#define ALARM\_MINUTE\_ADDR 12

#define ALARM2\_HOUR\_ADDR 20

#define ALARM2\_MINUTE\_ADDR 23

#define ALARM2\_TONE\_ADDR 30

#define ALARM\_TONE\_ADDR 31

#define INITIAL\_ALARM\_HOUR 1

#define INITIAL\_ALARM\_MINUTE 0

*//Radio Preset Addresses*

#define CHANNEL1\_ADDR1 1

#define CHANNEL1\_ADDR2 2

#define CHANNEL2\_ADDR1 3

#define CHANNEL2\_ADDR2 4

#define CHANNEL3\_ADDR1 5

#define CHANNEL3\_ADDR2 6

#define CHANNEL4\_ADDR1 7

#define CHANNEL4\_ADDR2 8

#define CHANNEL5\_ADDR1 9

#define CHANNEL5\_ADDR2 10

#define TIMEOUT\_SECONDS 60

#define MAX\_CHANNEL 1079

#define MIN\_CHANNEL 879

#include <stdbool.h>

#include <avr/eeprom.h>

#include <util/delay.h>

**static** uint16\_t b1\_state = 0;

**static** uint16\_t b2\_state = 0;

**static** uint16\_t b3\_state = 0;

**static** int b1\_flag = 0;

**static** int b2\_flag = 0;

**static** int b3\_flag = 0;

**static** int b3\_hold = 0;

**static** int b3\_hold\_state = 0;

**static** int b1\_hold = 0;

**static** int b1\_hold\_state = 0;

**static** int b2\_hold = 0;

**static** int b2\_hold\_state = 0;

**static** int b1\_count = 0;

**static** int b2\_count = 0;

**static** char State = 'I';

**static** int alarm = 0;

**static** int alarm2 = 0;

**static** int set\_alarm2;

**static** int timeout = 0;

**static** float volume = 10;

**static** uint16\_t preset[5];

**static** int months[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

*//Initialize Timer 0 for 5 ms Push-Button Debounce*

void timer0\_init(void){

cli();

TCCR0A |= \_BV(WGM01); *//Set to CTC mode*

TIFR0 |= \_BV(OCF0A); *//flag to set on compare match*

TCCR0B |= \_BV(CS02) | \_BV(CS00); *//Prescalar1024*

OCR0A = 78;

TIMSK0 |= \_BV(OCIE0A);

sei();

}

*//Initialize Timer 1 for alarm PWM gen*

void timer1\_init(void){

TCCR1A |= \_BV(COM1A1) | \_BV(WGM11);

TCCR1B |= \_BV(CS12) | \_BV(WGM13);

ICR1 = 3125;

OCR1A = 0;

}

*//Initialize Timer 2 for photocell PWM generation*

void timer2\_init(void){

TCCR2A |= \_BV(COM2B1) | \_BV(COM2B0) | \_BV(WGM20);

TCCR2B |= \_BV(WGM22) | \_BV(CS22); *//clk/64 prescalar*

OCR2A = 125;

OCR2B = 62;

}

*//Initialize Watchdog Timer*

void WDT\_Init(void)

{

*//disable interrupts*

cli();

*//set up WDT interrupt*

WDTCSR = (1<<WDCE)|(1<<WDE);

*//Start watchdog timer with 4s prescaller*

WDTCSR = (1<<WDIE)|(1<<WDP3);

*//Enable global interrupts*

sei();

}

*//Reads from EEPROM*

uint8\_t read\_eeprom(int addr){

uint8\_t byte ;

byte = eeprom\_read\_byte ((uint8\_t\*)addr) ;

**return** byte;

}

*//Writes to EEPROM*

void write\_RadioStation(uint16\_t word, int addr1, int addr2){

uint8\_t low\_byte = word & 0x00FF;

uint8\_t high\_byte = word >> 8;

eeprom\_update\_byte (( uint8\_t\*)addr1, high\_byte );

eeprom\_update\_byte (( uint8\_t\*)addr2, low\_byte );

}

uint16\_t read\_RadioStation(int addr1, int addr2){

uint8\_t high\_byte;

uint8\_t low\_byte;

high\_byte = eeprom\_read\_byte((uint8\_t\*)addr1);

low\_byte = eeprom\_read\_byte((uint8\_t\*)addr2);

uint16\_t word = high\_byte << 8;

word |= low\_byte;

**return** word;

}

*//Writes to EEPROM*

void write\_eeprom(uint8\_t byte, int addr){

eeprom\_update\_byte (( uint8\_t\*)addr, byte );

}

void write\_presets(){

write\_RadioStation(preset[0], CHANNEL1\_ADDR1, CHANNEL1\_ADDR2);

write\_RadioStation(preset[1], CHANNEL2\_ADDR1, CHANNEL2\_ADDR2);

write\_RadioStation(preset[2], CHANNEL3\_ADDR1, CHANNEL3\_ADDR2);

write\_RadioStation(preset[3], CHANNEL4\_ADDR1, CHANNEL4\_ADDR2);

write\_RadioStation(preset[4], CHANNEL5\_ADDR1, CHANNEL5\_ADDR2);

}

void display\_presets(){

LCD\_GoTo(0,0);

*//LCD\_WriteString("Presets");*

**for**(int i=0; i<5; i++){

LCD\_GoTo(0,i+1);

*//LCD\_WriteInt(i+1,false);*

*//LCD\_GoTo(6,i+1);*

*//LCD\_WriteChar(')');*

FM\_DisplayChannel(preset[i], 0, i+1, false);

}

}

*//Set up analog to digital converter in Free-Run mode*

*//Used for Photocell voltage conversion*

void ADC\_init(void){

ADMUX = 0;

ADMUX |= \_BV(REFS0) | \_BV(ADLAR) | 2;

ADCSRA = 0;

ADCSRA |= \_BV(ADEN) | \_BV(ADPS0) | \_BV(ADPS1) | \_BV(ADPS2);

ADCSRB = 0;

}

void setup(void){

cli();

*//Initialize GPIO pins as inputs or outputs*

DDRB |= \_BV(1); *//Alarm PWM OUT*

DDRC &= ~\_BV(2); *//Photocell ADC IN*

DDRC &= ~\_BV(3); *//Slide Switch*

DDRC |= \_BV(1); *//LED*

DDRC |= \_BV(0); *//LED*

DDRD &= ~\_BV(4); *//Slide Switch*

DDRD &= ~\_BV(6); *//Button*

DDRD &= ~\_BV(5); *//Button*

DDRD &= ~\_BV(7); *//Button*

DDRD |= \_BV(3); *//Backlight Adjust PWM OUT*

DDRD |= \_BV(2); *//RST FM\_Chip*

PORTD |= \_BV(6);

PORTD |= \_BV(5);

PORTD |= \_BV(7);

PORTD |= \_BV(3);

PORTD |= \_BV(2);

PORTC |= \_BV(3);

PORTD |= \_BV(4);

PORTC |= \_BV(1);

PORTC |= \_BV(0);

*//Initialize all peripherals*

timer0\_init();

timer1\_init();

timer2\_init();

WDT\_Init();

ADC\_init();

SPI\_Master\_Init();

I2C\_Init();

LCD\_Init();

FM\_Init();

*//Initialize time to 2:00PM and enable interrupts*

set\_Time(2,0,40);

printf("Time set**\n**");

set\_Date(11,5,14);

write\_eeprom(INITIAL\_ALARM\_HOUR, ALARM\_HOUR\_ADDR);

write\_eeprom(INITIAL\_ALARM\_HOUR, ALARM2\_HOUR\_ADDR);

write\_eeprom(INITIAL\_ALARM\_MINUTE, ALARM2\_MINUTE\_ADDR);

write\_eeprom(INITIAL\_ALARM\_MINUTE, ALARM\_MINUTE\_ADDR);

write\_eeprom(0x01, ALARM\_TONE\_ADDR);

write\_eeprom(0x01, ALARM2\_TONE\_ADDR);

sei();

}

*//Push Button Interrupt Handler*

ISR(TIMER0\_COMPA\_vect)

{

b3\_state = (b3\_state<<1) | (PIND & (1<<PD6)) >> 6 | 0xf800;

b2\_state = (b2\_state<<1) | (PIND & (1<<PD5)) >> 5 | 0xf800;

b1\_state = (b1\_state<<1) | (PIND & (1<<PD7)) >> 7 | 0xf800;

b3\_hold\_state = (b3\_state<<1) | (PIND & (1<<PD6)) >> 6 | 0xfc00;

b2\_hold\_state = (b2\_state<<1) | (PIND & (1<<PD5)) >> 5 | 0xfc00;

b1\_hold\_state = (b1\_state<<1) | (PIND & (1<<PD7)) >> 7 | 0xfc00;

*//B1 pushed*

**if**(b1\_state==0xfc00){

b1\_flag = 1;

}

**if**(b1\_hold\_state==0xfc00){

b1\_hold++;

}

**else**{

b1\_hold = 0;

}

*//B2 pushed*

**if**(b2\_state==0xfc00){

b2\_flag = 1;

}

**if**(b2\_hold\_state==0xfc00){

b2\_hold++;

}

**else**{

b2\_hold = 0;

}

*//B3 Pushed*

**if**(b3\_state==0xfc00){

b3\_flag = 1;

}

**if**(b3\_hold\_state==0xfc00){

b3\_hold++;

}

**else**{

b3\_hold = 0;

}

TIFR0 |= \_BV(OCF0A);

TCNT0 = 0;

}

*//Grab ADC conversion*

uint8\_t get\_conversion(void){

ADCSRA |= \_BV(ADIF) | \_BV(ADSC);

**while**((ADCSRA & ADIF) ==0);

**return** ADCH;

}

*//Watchdog timer interrupt handler*

ISR(WDT\_vect){

timeout++;

WDTCSR |= \_BV(WDIE);

}

*//Adjust LED's on display*

void backlight\_adjust(){

float sampled\_voltage=0;

float max\_voltage = 2.5;

float duty\_cycle = 0.5;

sampled\_voltage = get\_conversion()\*0.0195;

duty\_cycle = (sampled\_voltage/max\_voltage);

OCR2B = duty\_cycle\*OCR2A;

}

*//Waits 1 second*

void one\_sec\_delay(void){

\_delay\_ms(1000);

}

*//Used when jumping out of states to reset all flags, counts and buffers*

void reset\_variables(){

RTC\_ClearVariables();

LCD\_Clear();

b1\_flag=0;

b2\_flag=0;

b1\_count=0;

b2\_count=0;

}

*//TODO: Implement functionality for second alarm set*

void set\_alarm\_state(){

uint8\_t hour;

uint8\_t minute;

uint8\_t tone;

*//Initialize alarm time, alarm and time-out*

**if**(alarm2){

hour = read\_eeprom(ALARM2\_HOUR\_ADDR);

minute = read\_eeprom(ALARM2\_MINUTE\_ADDR);

tone = read\_eeprom(ALARM2\_TONE\_ADDR);

}

**else**{

hour = read\_eeprom(ALARM\_HOUR\_ADDR);

minute = read\_eeprom(ALARM\_MINUTE\_ADDR);

tone = read\_eeprom(ALARM\_TONE\_ADDR);

}

alarm = 1;

bool pm = false;

int displayed\_hour = hour;

**if**(hour>12){

displayed\_hour -= 12;

}

**if**(hour >= 12){

pm = true;

}

**while**(timeout < TIMEOUT\_SECONDS/4){

*//Clear button flags*

b2\_flag = 0;

b1\_flag = 0;

b3\_flag = 0;

*//Draw Alarm time and flash*

alarm\_Display(minute, displayed\_hour, 13, 2);

**if**(tone == 1){

LCD\_DrawBell(3,35);

}

**else**{

LCD\_DrawNote(3,35);

}

**if**(pm){

LCD\_GoTo(70,2);

LCD\_WriteString(" ");

LCD\_GoTo(70,2);

LCD\_WriteString("PM");

}

**else**{

LCD\_GoTo(70,2);

LCD\_WriteString(" ");

LCD\_GoTo(70,2);

LCD\_WriteString("AM");

}

LCD\_GoTo(5,5);

LCD\_WriteString("Set Alarm");

\_delay\_ms(100);

*//Set minute with PB1*

**if**(b1\_flag==1){

minute++;

**if**(minute==60)

minute=0;

timeout=0;

b1\_flag=0;

}

**while**(b1\_hold > 100){

alarm\_Display(minute, displayed\_hour, 13, 2);

**if**(tone){

LCD\_DrawBell(3,35);

}

**else**{

LCD\_DrawNote(3,35);

}

**if**(pm){

LCD\_GoTo(70,2);

LCD\_WriteString(" ");

LCD\_GoTo(70,2);

LCD\_WriteString("PM");

}

**else**{

LCD\_GoTo(70,2);

LCD\_WriteString(" ");

LCD\_GoTo(70,2);

LCD\_WriteString("AM");

}

LCD\_GoTo(5,5);

LCD\_WriteString("Set Alarm");

minute++;

**if**(minute==60)

minute=0;

timeout=0;

b1\_flag=0;

\_delay\_ms(50);

}

*//Set hour with PB2*

**if**(b2\_flag==1){

hour++;

displayed\_hour = hour;

**if**(hour>12){

displayed\_hour -= 12;

}

**if**(displayed\_hour == 12){

pm ^= true;

}

**if**(hour==24){

hour=0;

}

timeout=0;

b2\_flag = 0;

}

**while**(b2\_hold > 100){

alarm\_Display(minute, displayed\_hour, 13, 2);

**if**(tone){

LCD\_DrawBell(3,35);

}

**else**{

LCD\_DrawNote(3,35);

}

**if**(pm){

LCD\_GoTo(70,2);

LCD\_WriteString(" ");

LCD\_GoTo(70,2);

LCD\_WriteString("PM");

}

**else**{

LCD\_GoTo(70,2);

LCD\_WriteString(" ");

LCD\_GoTo(70,2);

LCD\_WriteString("AM");

}

LCD\_GoTo(5,5);

LCD\_WriteString("Set Alarm");

hour++;

displayed\_hour = hour;

**if**(hour>12){

displayed\_hour -= 12;

}

**if**(displayed\_hour == 12){

pm ^= true;

}

**if**(hour==24){

hour=0;

}

timeout=0;

b2\_flag = 0;

\_delay\_ms(100);

}

**while**(b3\_hold != 0){

**if**(b3\_hold > 38){

tone ^= 1;

**if**(tone){

LCD\_DrawBell(3,35);

}

**else**{

LCD\_DrawNote(3,35);

}

\_delay\_ms(1000);

b3\_flag = 0;

}

b3\_hold = 0;

\_delay\_ms(200);

}

**if**(b3\_flag == 1){

b3\_flag = 0;

**break**;

}

}

*//Initialize alarm and go back to Idle state*

reset\_variables();

**if**(alarm2){

write\_eeprom(hour, ALARM2\_HOUR\_ADDR);

write\_eeprom(minute, ALARM2\_MINUTE\_ADDR);

write\_eeprom(tone, ALARM2\_TONE\_ADDR);

alarm2\_Set(hour,minute,0);

alarm2=0;

}

**else**{

write\_eeprom(hour, ALARM\_HOUR\_ADDR);

write\_eeprom(minute, ALARM\_MINUTE\_ADDR);

write\_eeprom(tone, ALARM\_TONE\_ADDR);

alarm\_Set(hour, minute, 0);

}

State = 'I';

}

*//Check to see if alarm should be going off*

void check\_Alarm(){

**if**(alarm\_Check()==1){

alarm\_Sound(volume);

}

}

*//Idle state displaying Large time*

*//TODO: Get a small date, station frequency, and temperature on here*

*//I doubt it will all fit so we may have to a scrolling row of this info.*

void idle\_state(){

RTC\_DisplayLargeClock(13,2);

RTC\_DisplayDate(13,3);

**if**(FM\_GetRadioState()){

FM\_DisplayChannel(FM\_GetChannel(), 40, 5, false);

FM\_DisplaySignalStrength(70,5);

}

**else**{

LCD\_GoTo(40,5);

LCD\_WriteString(" ");

}

**if**(alarm==1 && !(PINC & \_BV(3))){

**if**(read\_eeprom(ALARM\_TONE\_ADDR)==1){

LCD\_DrawBell(5,0);

}

**else**{

LCD\_DrawNote(5,0);

}

}

**else**{

LCD\_GoTo(0,5);

LCD\_WriteString(" ");

}

**if**(set\_alarm2==1 && !(PIND & \_BV(4))){

**if**(read\_eeprom(ALARM2\_TONE\_ADDR)==1){

LCD\_DrawBell(5,14);

}

**else**{

LCD\_DrawNote(5,14);

}

}

**else**{

LCD\_GoTo(14,5);

LCD\_WriteString(" ");

}

check\_Alarm();

}

*/\*Displays all options to user. User is able to scroll through list using*

*\*PB1. User can enter the selected state by pressing PB2\*/*

*//TODO: Implement watchdog timer user timeouts and third push button to exit ALL states*

void menu\_state(){

State = 'I';

*//Draw Menu*

LCD\_Clear();

LCD\_GoTo(1, 0);

LCD\_WriteString("Set Radio");

LCD\_GoTo(1,1);

LCD\_WriteString("Set Alarm 1");

LCD\_GoTo(1,2);

LCD\_WriteString("Set Alarm 2");

LCD\_GoTo(1,3);

LCD\_WriteString("Set Time");

LCD\_GoTo(1,4);

LCD\_WriteString("Set Date");

int loop = 0;

**while**(timeout<TIMEOUT\_SECONDS/4){

*//Reset Button Flags every loop*

b1\_flag = 0;

b2\_flag = 0;

b3\_flag = 0;

\_delay\_ms(100);

*//Increment button counts on button flag*

**if**(b1\_flag==1){

timeout=0;

b1\_count++;

}

**if**(b2\_flag==1){

timeout=0;

b2\_count++;

}

**if**(b3\_flag == 1){

b3\_flag=0;

State = 'I';

**break**;

}

*//Radio Select*

**if**(b1\_count==1){

**if**(b1\_flag ==1){

LCD\_InvertRow(0);

**if**(loop>0){

LCD\_InvertRow(4);

}

}

**if**(b2\_flag == 1){

State = 'R';

**break**;

}

}

*//Set Alarm 1 Select*

**if**(b1\_count == 2){

**if**(b1\_flag == 1){

LCD\_InvertRow(0);

LCD\_InvertRow(1);

}

**if**(b2\_flag == 1){

State = 'A';

**break**;

}

}

*//Set Alarm 2 Select*

**if**(b1\_count == 3){

**if**(b1\_flag ==1){

LCD\_InvertRow(1);

LCD\_InvertRow(2);

}

**if**(b2\_flag == 1){

alarm2 = 1;

set\_alarm2=1;

State = 'A';

**break**;

}

}

*//Time-set Select*

**if**(b1\_count == 4){

**if**(b1\_flag ==1){

LCD\_InvertRow(2);

LCD\_InvertRow(3);

}

**if**(b2\_flag == 1){

State = 'T';

**break**;

}

}

*//Date-set Select*

**if**(b1\_count == 5){

**if**(b1\_flag ==1){

LCD\_InvertRow(3);

LCD\_InvertRow(4);

b1\_count = 0;

loop++;

}

}

**if**(b1\_count == 0 && loop>0){

**if**(b2\_flag == 1){

State = 'D';

**break**;

}

}

check\_Alarm();

}

*//Reset all button counts and flags*

reset\_variables();

}

*//Allows user to set time. A lot of code is borrowed from the set alarm state*

void set\_time\_state(){

*//Initialize alarm time, alarm and time-out*

int hour = get\_Hours();

**if**(RTC\_GetPM()){

hour += 12;

}

**if**(hour == 24){

hour=0;

}

int minute = get\_Minutes();

int displayed\_hour = get\_Hours();

**while**(timeout < TIMEOUT\_SECONDS/4){

*//Draw Time*

alarm\_Display(minute, displayed\_hour, 13, 2);

LCD\_GoTo(25,4);

LCD\_WriteString("Set Time");

\_delay\_ms(100);

*//Set minute with PB1*

**if**(b1\_flag==1){

minute++;

**if**(minute==60)

minute=0;

timeout=0;

b1\_flag=0;

}

**while**(b1\_hold > 100){

alarm\_Display(minute, displayed\_hour, 13, 2);

LCD\_GoTo(25,4);

LCD\_WriteString("Set Time");

minute++;

**if**(minute==60)

minute=0;

timeout=0;

b1\_flag=0;

\_delay\_ms(50);

}

*//Set hour with PB2*

**if**(b2\_flag==1){;

hour++;

displayed\_hour = hour;

**if**(hour>12){

displayed\_hour -= 12;

}

**if**(hour == 24){

hour=0;

}

timeout=0;

b2\_flag = 0;

set\_Time(hour, minute, 0x00);

}

**while**(b2\_hold > 100){

alarm\_Display(minute, displayed\_hour, 13, 2);

LCD\_GoTo(25,4);

LCD\_WriteString("Set Time");

hour++;

displayed\_hour = hour;

**if**(hour>12){

displayed\_hour -= 12;

}

**if**(hour == 24){

hour=0;

}

timeout=0;

b2\_flag = 0;

set\_Time(hour, minute, 0x00);

get\_Hours();

\_delay\_ms(100);

}

**if**(b3\_flag ==1){

b3\_flag = 0;

**break**;

}

check\_Alarm();

}

*//Initialize time and go back to Idle state*

reset\_variables();

set\_Time(hour, minute, 0x00);

State = 'I';

}

*//TODO: Make this state functional*

void set\_date\_state(){

int day = get\_Day();

int month = get\_Month();

int year = get\_Year();

LCD\_Clear();

**while**(timeout<TIMEOUT\_SECONDS/4){

RTC\_DisplayDate(13,2);

LCD\_GoTo(25,4);

LCD\_WriteString("Set Date");

*//Set day with PB1*

**if**(b1\_flag==1){

day++;

**if**(day>months[month-1])

day=1;

timeout=0;

set\_Date(month,day,year);

b1\_flag=0;

}

**while**(b1\_hold > 100){

RTC\_DisplayDate(13,2);

LCD\_GoTo(25,4);

LCD\_WriteString("Set Date");

day++;

**if**(day>months[month-1])

day=1;

timeout=0;

set\_Date(month,day,year);

b1\_flag=0;

\_delay\_ms(100);

}

*//Set month with PB2*

**if**(b2\_flag==1){

month++;

**if**(month==13)

month=1;

timeout=0;

set\_Date(month,day,year);

b2\_flag=0;

}

**while**(b2\_hold > 100){

RTC\_DisplayDate(13,2);

LCD\_GoTo(25,4);

LCD\_WriteString("Set Date");

month++;

**if**(month==13)

month=1;

timeout=0;

set\_Date(month,day,year);

b2\_flag=0;

\_delay\_ms(100);

}

**if**(b3\_flag == 1){

b3\_flag = 0;

**break**;

}

check\_Alarm();

}

reset\_variables();

State = 'I';

}

*//Still in its infant stages*

*//TODO: Make this state functional*

void set\_radio\_state(){

uint16\_t channel = FM\_GetChannel();

**if**(!FM\_GetRadioState())

FM\_PowerOn();

**if**(channel < MIN\_CHANNEL || channel > MAX\_CHANNEL){

channel = MIN\_CHANNEL;

}

b1\_flag = 0;

b2\_flag = 0;

b3\_flag = 0;

LCD\_Clear();

FM\_DisplayChannel(channel,35,2, true);

FM\_TuneToChannel(channel);

timeout = 0;

**while**(timeout<TIMEOUT\_SECONDS/4){

*//printf("In Loop\n");*

display\_presets();

FM\_DisplayChannel(channel, 35,2, true);

**if**(b1\_flag == 1){

channel += 2;

**if**(channel < MIN\_CHANNEL || channel > MAX\_CHANNEL){

channel = MIN\_CHANNEL;

}

display\_presets();

FM\_DisplayChannel(channel, 35,2, true);

FM\_TuneToChannel(channel);

timeout = 0;

b1\_flag = 0;

}

**while**(b1\_hold > 100){

channel += 2;

**if**(channel < MIN\_CHANNEL || channel > MAX\_CHANNEL){

channel = MIN\_CHANNEL;

}

display\_presets();

FM\_DisplayChannel(channel, 35,2, true);

timeout = 0;

b1\_flag = 0;

\_delay\_ms(100);

}

*//Set preset if button held*

**while**(b2\_hold != 0){

**if**(b2\_hold > 38){

**for**(int i=4; i>0; i--){

preset[i] = preset[i-1];

}

preset[0] = channel;

write\_presets();

display\_presets();

\_delay\_ms(1000);

b2\_flag = 0;

}

b2\_hold = 0;

\_delay\_ms(200);

}

**if**(b2\_flag == 1){

FM\_Seek();

timeout = 0;

channel = FM\_GetChannel();

b2\_flag = 0;

}

*//Add preset if B2 is held*

**if**(b3\_flag == 1){

b3\_flag = 0;

**break**;

}

}

write\_RadioStation(channel, CHANNEL1\_ADDR1, CHANNEL1\_ADDR2);

reset\_variables();

State = 'I';

}

void preset\_select(int num){

**if**(num==0){

FM\_TuneToChannel(read\_RadioStation(CHANNEL1\_ADDR1, CHANNEL1\_ADDR2));

}

**if**(num==1){

FM\_TuneToChannel(read\_RadioStation(CHANNEL2\_ADDR1, CHANNEL2\_ADDR2));

}

**if**(num==2){

FM\_TuneToChannel(read\_RadioStation(CHANNEL3\_ADDR1, CHANNEL3\_ADDR2));

}

**if**(num==3){

FM\_TuneToChannel(read\_RadioStation(CHANNEL4\_ADDR1, CHANNEL4\_ADDR2));

}

**if**(num==5){

FM\_TuneToChannel(read\_RadioStation(CHANNEL5\_ADDR1, CHANNEL5\_ADDR2));

}

}

void alarm\_Sound(float volume){

*//Alarm 1 is ringing*

**if**(volume>90){

volume = 90;

}

**if**(get\_Alarm\_Minute() == get\_Minutes()){

set\_AlarmRing(1);

**if**(read\_eeprom(ALARM\_TONE\_ADDR)){

OCR1A = (volume/100)\*ICR1;

}

**else**{

FM\_PowerOn();

}

}

*//Alarm 2 is ringing*

**else**{

set\_Alarm2Ring(1);

**if**(read\_eeprom(ALARM2\_TONE\_ADDR)){

OCR1A = (volume/100)\*ICR1;

}

**else**{

FM\_PowerOn();

}

}

LCD\_Clear();

timeout = 0;

**while**(timeout<TIMEOUT\_SECONDS/4){

RTC\_DisplayLargeClock(13,2);

RTC\_DisplayDate(13,3);

\_delay\_ms(300);

reset\_variables();

\_delay\_ms(200);

**if**(b1\_flag){

alarm\_Silence();

alarm\_Set(read\_eeprom(ALARM\_HOUR\_ADDR), read\_eeprom(ALARM\_MINUTE\_ADDR),0);

alarm2\_Set(read\_eeprom(ALARM2\_HOUR\_ADDR), read\_eeprom(ALARM2\_MINUTE\_ADDR),0);

b1\_flag = 0;

**break**;

}

**if**(b2\_flag){

alarm\_Snooze();

b2\_flag = 0;

**break**;

}

}

OCR1A=0;

set\_Alarm2Ring(0);

set\_AlarmRing(0);

reset\_variables();

State = 'I';

}

int main(void){

*//Initialize UART communication with computer*

USART\_Init(MYUBRR);

stdout = &uart\_output;

stdin = &uart\_input;

printf("START**\n**");

setup();

**while**(1){

**if**(b1\_flag==1){

State = 'M';

}

**if**(b2\_flag==1){

b2\_flag=0;

**if**(FM\_GetRadioState()){

preset\_select(b2\_count);

b2\_count++;

**if**(b2\_count == 5){

b2\_count = 0;

}

}

}

**if**(b3\_flag){

**if**(FM\_GetRadioState())

FM\_PowerDown();

**else**{

FM\_PowerOn();

}

b3\_flag = 0;

}

**switch**(State)

{

**case** 'I':

idle\_state();

**break**;

**case** 'M':

timeout = 0;

menu\_state();

**break**;

**case** 'A':

timeout = 0;

set\_alarm\_state();

**break**;

**case** 'T':

timeout = 0;

set\_time\_state();

**break**;

**case** 'R':

timeout = 0;

set\_radio\_state();

**break**;

**case** 'D':

timeout = 0;

set\_date\_state();

**break**;

}

check\_Alarm();

backlight\_adjust();

}

**return** 0;

}

*/\**

*\* Alarm\_Driver.h*

*\**

*\* Created: 10/9/2014 1:32:03 PM*

*\* Author: Nick Schrock, Kevin Sager*

*\*/*

void alarm\_Set(int hours, int minutes, int seconds);

void alarm\_Sound(float volume);

void alarm\_Silence(void);

void alarm\_Display(int hours, int minutes, int xpos, int ypos);

int alarm\_Check();

void one\_sec\_delay(void);

void alarm2\_Set(int hours, int minutes, int seconds);

void alarm\_Snooze();

uint8\_t get\_Alarm\_Minute();

uint8\_t get\_Alarm\_Hour();

uint8\_t get\_Alarm\_Second();

uint8\_t get\_Alarm2\_Minute();

uint8\_t get\_Alarm2\_Second();

uint8\_t get\_Alarm2\_Hour();

void set\_Alarm2Ring(bool ring);

void set\_AlarmRing(bool ring);

*/\**

*\* Alarm\_Driver.c*

*\**

*\* Created: 10/9/2014 1:28:37 PM*

*\* Author: Nick Schrock, Kevin Sager*

*\*/*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*The primary function of this module is to factor out some of the code responsible*

*for controlling the alarm system. This is a relatively short module used to set,*

*sound, and silence the alarm.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <avr/io.h>

#include <stdbool.h>

#include "RTC\_Driver.h"

#include "Alarm\_Driver.h"

#include "LCD\_Driver.h"

#include "FM\_Driver.h"

#include <stdio.h>

#define LARGE\_SPACE 13

uint8\_t alarm\_hour=0;

uint8\_t alarm\_minute=0;

uint8\_t alarm\_second=0;

uint8\_t alarm2\_hour=0;

uint8\_t alarm2\_minute=0;

uint8\_t alarm2\_second=0;

int snooze\_num;

bool alarm\_ring = false;

bool alarm2\_ring = false;

bool alarm\_set = false;

bool alarm2\_set = false;

uint8\_t get\_Alarm\_Minute(){

**return** alarm\_minute;

}

uint8\_t get\_Alarm\_Second(){

**return** alarm\_second;

}

uint8\_t get\_Alarm\_Hour(){

**return** alarm\_hour;

}

uint8\_t get\_Alarm2\_Minute(){

**return** alarm2\_minute;

}

uint8\_t get\_Alarm2\_Second(){

**return** alarm2\_second;

}

uint8\_t get\_Alarm2\_Hour(){

**return** alarm2\_hour;

}

void set\_Alarm2Ring(bool ring){

alarm2\_ring = ring;

}

void set\_AlarmRing(bool ring){

alarm\_ring = ring;

}

void alarm\_Set(int hours, int minutes, int seconds){

alarm\_hour = hours;

alarm\_minute = minutes;

alarm\_second = seconds;

alarm\_set = true;

}

void alarm2\_Set(int hours, int minutes, int seconds){

alarm2\_hour = hours;

alarm2\_minute = minutes;

alarm2\_second = seconds;

alarm2\_set = true;

}

void alarm\_Display(int minutes, int hours, int xpos, int ypos){

int hours\_pos1 = xpos - LARGE\_SPACE;

int hours\_pos2 = xpos;

int colon\_pos = hours\_pos2 + LARGE\_SPACE;

int minutes\_pos1 = colon\_pos + LARGE\_SPACE;

int minutes\_pos2 = minutes\_pos1 + LARGE\_SPACE;

*//Write Hours to screen*

**if**(hours>9){

hours\_pos2 = hours\_pos1;

}

LCD\_Clear();

LCD\_GoTo(hours\_pos2, ypos);

LCD\_WriteInt(hours,true);

LCD\_GoTo(colon\_pos, ypos);

LCD\_WriteColon(colon\_pos, ypos);

*//Write Minutes to Screen*

LCD\_GoTo(minutes\_pos1,ypos);

**if**(minutes<10){

LCD\_WriteInt(0,true);

LCD\_GoTo(minutes\_pos2,ypos);

}

LCD\_WriteInt(minutes,true);

**if**(RTC\_GetPM()){

LCD\_GoTo(70,2);

LCD\_WriteString("PM");

}

**else**{

LCD\_GoTo(70,2);

LCD\_WriteString("AM");

}

}

void alarm\_Silence(void){

alarm\_ring = false;

alarm2\_ring = false;

**if**(FM\_GetRadioState()){

FM\_PowerDown();

}

OCR1A = 0;

}

int alarm\_Check(){

**if**(get\_Hours() == alarm\_hour && get\_Minutes() == alarm\_minute && get\_Seconds() == alarm\_second && !(PINC & \_BV(3))){

**return** 1;

}

**if**(get\_Hours() == alarm2\_hour && get\_Minutes() == alarm2\_minute && get\_Seconds() == alarm2\_second && !(PIND & \_BV(4))){

**return** 1;

}

**else**{

**return** 0;

}

}

void alarm\_Snooze(){

**if**(alarm\_ring==1){

alarm\_minute += 10;

**if**(alarm\_minute>60){

alarm\_hour++;

alarm\_minute -=60;

}

}

*//alarm\_Set(alarm\_hour, alarm\_minute, 0);*

**if**(alarm2\_ring==1){

alarm2\_minute += 10;

**if**(alarm2\_minute>60){

alarm2\_hour++;

alarm2\_minute -=60;

}

*//alarm2\_Set(alarm2\_hour, alarm2\_minute, 0);*

}

alarm\_Silence();

}

*/\**

*\* FM\_Driver.c*

*\**

*\* Created: 10/9/2014 1:28:37 PM*

*\* Author: Nick Schrock, Kevin Sager*

*\* Written in reference to https://www.sparkfun.com/products/11083*

*\*/*

#include <stdbool.h>

#define SI4703 (0x10<<1)

*//Device Registers*

#define DEVICEID 0x00

#define CHIPID 0x01

#define POWERCFG 0x02

#define CHANNEL 0x03

#define SYSCONFIG1 0x04

#define SYSCONFIG2 0x05

#define STATUSRSSI 0x0A

#define READCHAN 0x0B

#define RDSA 0x0C

#define RDSB 0x0D

#define RDSC 0x0E

#define RDSD 0x0F

*//Register 0x02 - POWERCFG*

#define SMUTE 15

#define DMUTE 14

#define SKMODE 10

#define SEEKUP 9

#define SEEK 8

*//Register 0x03 - CHANNEL*

#define TUNE 15

*//Register 0x04 - SYSCONFIG1*

#define RDS 12

#define DE 11

*//Register 0x05 - SYSCONFIG2*

#define SPACE1 5

#define SPACE0 4

*//Register 0x0A - STATUSRSSI*

#define RDSR 15

#define STC 14

#define SFBL 13

#define AFCRL 12

#define STEREO 8

#define SCK 5

#define SDA 4

#define RST 2

#define MAX\_CHANNEL 1079

#define MIN\_CHANNEL 879

void FM\_PrintRegisters();

void FM\_Init();

void FM\_SetVolume(uint16\_t level);

void FM\_ReadRegisters();

void FM\_UpdateRegisters();

*//int digitAtPos(int number, int pos);*

void FM\_DisplayChannel(int channel, int xpos, int ypos, bool large);

int FM\_TuneToChannel(int channel);

int FM\_ReadChannel();

void FM\_Reset();

int FM\_GetChannel();

uint8\_t FM\_GetSignalStrength();

void FM\_DisplaySignalStrength(int xpos, int ypos);

void FM\_Seek(void);

void FM\_PowerOn();

bool FM\_GetRadioState();

void FM\_PowerDown();

*/\**

*\* FM\_Driver.c*

*\**

*\* Created: 10/9/2014 1:28:37 PM*

*\* Author: Nick Schrock, Kevin Sager*

*\* Written in reference to https://www.sparkfun.com/products/11083*

*\*/*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*//TODO: Write this Driver for the SI4705 FM Receiver*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#define LARGE\_SPACE 13

#define SMALL\_SPACE 6

#define F\_CPU 16000000UL

#define BAUD 9600

#include <avr/io.h>

#include <util/delay.h>

#include "FM\_Driver.h"

#include "TWI\_Driver.h"

#include "LCD\_Driver.h"

#include <stdio.h>

uint16\_t registers[16];

int current\_channel = 879;

bool radio\_on = false;

*//To get the Si4703 inito 2-wire mode, SEN needs to be high and SDIO needs to be low after a reset*

*//The breakout board has SEN pulled high, but also has SDIO pulled high. Therefore, after a normal power up*

*//The Si4703 will be in an unknown state. RST must be controlled*

void FM\_Init(){

DDRC |= \_BV(SDA);

*//Clear SDIO and RESET*

PORTC &= ~\_BV(SDA);

PORTD &= ~\_BV(RST);

\_delay\_ms(10);

*//Set RESET*

PORTD |= \_BV(RST);

\_delay\_ms(10);

*//Set SDIO and SCK*

DDRC &= ~(\_BV(5) | \_BV(4));

}

void FM\_PowerOn(){

*//FM\_Reset();*

radio\_on = true;

FM\_ReadRegisters();

registers[0x07] = 0x8100; *//Enables the oscillator*

FM\_UpdateRegisters();

\_delay\_ms(500);

FM\_ReadRegisters();

registers[POWERCFG] = 0x4001; *//Enables the IC*

registers[POWERCFG] |= \_BV(SMUTE) | \_BV(DMUTE);

registers[SYSCONFIG1] |= (1<<RDS); *//Enable RDS*

registers[SYSCONFIG2] &= ~(1<<SPACE1 | 1<<SPACE0); *//200kHz channel spacing in the US*

registers[SYSCONFIG2] &= (0xFFF0); *//Clear Volume bits*

registers[SYSCONFIG2] |= 0x0F;

*//printf("\n\nUpdating registers after config initializations\n\n");*

FM\_UpdateRegisters();

\_delay\_ms(110); *//Wait for power-up*

*//printf("\n\nInitialization Complete\n\n");*

PORTC &= ~\_BV(0);

}

void FM\_SetVolume(uint16\_t level){

I2C\_WriteRegister(SI4703, SYSCONFIG2, level);

}

void FM\_ReadRegisters(){

int index = 0;

I2C\_Start(); *//set\_up-->TWINT, TWSTA, TWEN*

*//printf("I2C started\n");*

I2C\_SendAddress(SI4703 | READ);

*//printf("Address sent\n");*

**for**(int i=0; i<16; i++){

**if**(i<6){

index = i+10;

}

**else**{

index = i-6;

}

registers[index] = I2C\_ReadRegister\_16bit();

}

TWCR = TW\_NACK;

**while**(!TW\_READY);

}

void FM\_UpdateRegisters(){

I2C\_Start();

I2C\_SendAddress(SI4703);

**for**(int i=0x02; i<0x08; i++){

*//printf("Writing %i to register %i\n", registers[i], i);*

I2C\_WriteRegister\_16bit(registers[i]);

}

I2C\_Stop();

}

void FM\_Reset(){

registers[POWERCFG] = 0x0000;

registers[CHANNEL] = 0x0000;

registers[SYSCONFIG1] = 0x0000;

registers[SYSCONFIG2] = 0x0000;

registers[0x06] = 0x0000;

registers[0x07] = 0x0100;

registers[0x08] = 0x0000;

FM\_UpdateRegisters();

}

void FM\_DisplayChannel(int channel, int xpos, int ypos, bool large){

**if**(channel == 1011 || channel == 879){

*//LCD\_Clear();*

}

int dig1 = 0;

**if**(channel<900){

dig1 = 8;

}

**else**{

dig1 = 9;

}

LCD\_GoTo(xpos,ypos);

**if**(large){

**if**(channel<1000){

LCD\_WriteInt(dig1, true);

*//printf("Channel: %i \nfirst digit: %i\n",channel, digitAtPos(channel,3));*

LCD\_GoTo(xpos+LARGE\_SPACE-2, ypos);

LCD\_WriteInt(digitAtPos(channel,2), true);

LCD\_GoTo(xpos+(2\*LARGE\_SPACE)-2,ypos);

LCD\_WriteChar('.');

LCD\_GoTo(xpos+((2\*LARGE\_SPACE)+2), ypos);

LCD\_WriteInt(digitAtPos(channel,1),true);

}

**else**{

xpos -=6;

LCD\_GoTo(xpos, ypos);

LCD\_WriteInt(digitAtPos(channel,4), true);

LCD\_GoTo(xpos+LARGE\_SPACE-2, ypos);

LCD\_WriteInt(digitAtPos(channel,3), true);

LCD\_GoTo(xpos+2\*(LARGE\_SPACE-2), ypos);

LCD\_WriteInt(digitAtPos(channel,2),true);

LCD\_GoTo(xpos+(3\*LARGE\_SPACE-2)+2,ypos);

LCD\_WriteChar('.');

LCD\_GoTo(xpos+(3\*LARGE\_SPACE-2)+4, ypos);

LCD\_WriteInt(digitAtPos(channel,1),true);

}

}

**else**{

**if**(channel<1000){

LCD\_WriteInt(digitAtPos(channel,3), false);

LCD\_GoTo(xpos+SMALL\_SPACE, ypos);

LCD\_WriteInt(digitAtPos(channel,2), false);

LCD\_GoTo(xpos+2\*SMALL\_SPACE, ypos);

LCD\_WriteChar('.');

LCD\_GoTo(xpos+2\*SMALL\_SPACE+4,ypos);

LCD\_WriteInt(digitAtPos(channel,1),false);

}

**else**{

LCD\_WriteInt(digitAtPos(channel,4), false);

LCD\_GoTo(xpos+SMALL\_SPACE, ypos);

LCD\_WriteInt(digitAtPos(channel,3), false);

LCD\_GoTo(xpos+SMALL\_SPACE+SMALL\_SPACE, ypos);

LCD\_WriteInt(digitAtPos(channel,2),false);

LCD\_GoTo(xpos+3\*SMALL\_SPACE, ypos);

LCD\_WriteChar('.');

LCD\_GoTo(xpos+3\*SMALL\_SPACE+4,ypos);

LCD\_WriteInt(digitAtPos(channel,1),false);

}

}

}

int FM\_GetChannel(){

**return** current\_channel;

}

int FM\_TuneToChannel(int channel){

*//TODO: write function to select a channel at a certain frequency*

*//eg: 973 > 97.3*

**if** (channel>MAX\_CHANNEL || channel<MIN\_CHANNEL){

channel = MIN\_CHANNEL;

}

current\_channel = channel;

int step = 0;

channel \*= 10;

channel -= 8750;

channel /= 20;

*//printf("\n\nAttempting to Tune to Channel: %i\n\n", channel );*

*//printf("\n\nReading registers after initialization\n\n");*

FM\_ReadRegisters();

*//FM\_PrintRegisters();*

registers[CHANNEL] &= 0xFE00; *//Clear all channel bits*

registers[CHANNEL] |= channel; *//Enter in new channel*

registers[CHANNEL] |= (1<<TUNE); *//Start the tune bit*

*//printf("\n\nUpdating Channel Register 3\n\n");*

FM\_UpdateRegisters();

**while**(!(registers[STATUSRSSI] & (1<<STC))){

FM\_ReadRegisters();

\_delay\_ms(1);

*//printf("waiting for tuning to complete...\n");//wait for tuning to complete*

}

*//printf("\n\nReading Registers after Channel Update\n\n");*

FM\_ReadRegisters();

*//FM\_PrintRegisters();*

*//printf("\n\nUpdating registers: Clearing TUNE bit\n");*

registers[CHANNEL] &= ~(1<<TUNE); *//Clear tune bit*

FM\_UpdateRegisters();

**while**(registers[STATUSRSSI] & (1<<STC)){

*//printf("Waiting for STC bit to clear...\n ");*

FM\_ReadRegisters();

}

*//printf("\n\nTuning Complete\n\n");*

*//FM\_PrintRegisters();*

PORTC &= ~\_BV(1);

**return** step;

}

uint8\_t FM\_GetSignalStrength(){

FM\_ReadRegisters();

uint8\_t strength = 0x00;

strength |= registers[STATUSRSSI];

**return** strength;

}

void FM\_DisplaySignalStrength(int xpos, int ypos){

LCD\_GoTo(xpos, ypos);

int strength = FM\_GetSignalStrength();

**if**(strength>10){

LCD\_DrawBar(xpos, ypos, 0xC0);

xpos++;

xpos++;

LCD\_GoTo(xpos,ypos);

LCD\_WriteChar(' ');

}

**if**(strength>20){

LCD\_DrawBar(xpos, ypos, 0xF0);

xpos++;

xpos++;

LCD\_GoTo(xpos,ypos);

LCD\_WriteChar(' ');

}

**if**(strength>30){

LCD\_DrawBar(xpos, ypos, 0xFC);

xpos++;

xpos++;

LCD\_GoTo(xpos,ypos);

LCD\_WriteChar(' ');

}

**if**(strength>40){

LCD\_DrawBar(xpos, ypos, 0xFF);

xpos++;

xpos++;

}

}

void FM\_Seek(void){

current\_channel += 2;

FM\_TuneToChannel(current\_channel);

**while**(FM\_GetSignalStrength()<20){

current\_channel += 2;

**if**(current\_channel < MIN\_CHANNEL || current\_channel > MAX\_CHANNEL){

current\_channel = MIN\_CHANNEL;

}

FM\_DisplayChannel(current\_channel, 35, 3, true);

FM\_TuneToChannel(current\_channel);

}

}

bool FM\_GetRadioState(){

**return** radio\_on;

}

void FM\_PowerDown(){

radio\_on = false;

FM\_ReadRegisters();

registers[SYSCONFIG1] &= ~(1<<RDS);

registers[POWERCFG] |= \_BV(0) | \_BV(6);

FM\_UpdateRegisters();

PORTC |= \_BV(1);

PORTC |= \_BV(0);

}

void FM\_PrintRegisters(void){

**for**(int i=0; i<16; i++){

printf("%i: %x**\n**", i, registers[i]);

}

}

*//Header file for LCD Driver*

*//Author: Nick Schrock, Kevin Sager*

*//written in reference to https://github.com/mspiceland/avr-spiceduino-3310-thermistor*

#define SET\_DC\_PIN PORTB |= 0x01

#define CLEAR\_DC\_PIN PORTB &= ~0x01

#define SET\_SCE\_PIN PORTB |= 0x04

#define CLEAR\_SCE\_PIN PORTB &= ~0x04

#define SET\_RST\_PIN PORTB |= 0x10

#define CLEAR\_RST\_PIN PORTB &= ~0x10

#include <avr/pgmspace.h>

#include <stdbool.h>

**static** **const** unsigned char smallFont[] PROGMEM =

{

0x00, 0x00, 0x00, 0x00, 0x00, *// sp*

0x00, 0x00, 0x2f, 0x00, 0x00, *// !*

0x00, 0x07, 0x00, 0x07, 0x00, *// "*

0x14, 0x7f, 0x14, 0x7f, 0x14, *// #*

0x24, 0x2a, 0x7f, 0x2a, 0x12, *// $*

0xc4, 0xc8, 0x10, 0x26, 0x46, *// %*

0x36, 0x49, 0x55, 0x22, 0x50, *// &*

0x00, 0x05, 0x03, 0x00, 0x00, *// '*

0x00, 0x1c, 0x22, 0x41, 0x00, *// (*

0x00, 0x41, 0x22, 0x1c, 0x00, *// )*

0x14, 0x08, 0x3E, 0x08, 0x14, *// \**

0x08, 0x08, 0x3E, 0x08, 0x08, *// +*

0x00, 0x00, 0x50, 0x30, 0x00, *// ,*

0x10, 0x10, 0x10, 0x10, 0x10, *// -*

0x00, 0x60, 0x60, 0x00, 0x00, *// .*

0x20, 0x10, 0x08, 0x04, 0x02, *// /*

0x3E, 0x51, 0x49, 0x45, 0x3E, *// 0 17*

0x00, 0x42, 0x7F, 0x40, 0x00, *// 1 18*

0x42, 0x61, 0x51, 0x49, 0x46, *// 2 19*

0x21, 0x41, 0x45, 0x4B, 0x31, *// 3 20*

0x18, 0x14, 0x12, 0x7F, 0x10, *// 4 21*

0x27, 0x45, 0x45, 0x45, 0x39, *// 5 22*

0x3C, 0x4A, 0x49, 0x49, 0x30, *// 6 23*

0x01, 0x71, 0x09, 0x05, 0x03, *// 7 24*

0x36, 0x49, 0x49, 0x49, 0x36, *// 8 25*

0x06, 0x49, 0x49, 0x29, 0x1E, *// 9 26*

0x00, 0x36, 0x36, 0x00, 0x00, *// :*

0x00, 0x56, 0x36, 0x00, 0x00, *// ;*

0x08, 0x14, 0x22, 0x41, 0x00, *// <*

0x14, 0x14, 0x14, 0x14, 0x14, *// =*

0x00, 0x41, 0x22, 0x14, 0x08, *// >*

0x02, 0x01, 0x51, 0x09, 0x06, *// ?*

0x32, 0x49, 0x59, 0x51, 0x3E, *// @*

0x7E, 0x11, 0x11, 0x11, 0x7E, *// A*

0x7F, 0x49, 0x49, 0x49, 0x36, *// B*

0x3E, 0x41, 0x41, 0x41, 0x22, *// C*

0x7F, 0x41, 0x41, 0x22, 0x1C, *// D*

0x7F, 0x49, 0x49, 0x49, 0x41, *// E*

0x7F, 0x09, 0x09, 0x09, 0x01, *// F*

0x3E, 0x41, 0x49, 0x49, 0x7A, *// G*

0x7F, 0x08, 0x08, 0x08, 0x7F, *// H*

0x00, 0x41, 0x7F, 0x41, 0x00, *// I*

0x20, 0x40, 0x41, 0x3F, 0x01, *// J*

0x7F, 0x08, 0x14, 0x22, 0x41, *// K*

0x7F, 0x40, 0x40, 0x40, 0x40, *// L*

0x7F, 0x02, 0x0C, 0x02, 0x7F, *// M*

0x7F, 0x04, 0x08, 0x10, 0x7F, *// N*

0x3E, 0x41, 0x41, 0x41, 0x3E, *// O*

0x7F, 0x09, 0x09, 0x09, 0x06, *// P*

0x3E, 0x41, 0x51, 0x21, 0x5E, *// Q*

0x7F, 0x09, 0x19, 0x29, 0x46, *// R*

0x46, 0x49, 0x49, 0x49, 0x31, *// S*

0x01, 0x01, 0x7F, 0x01, 0x01, *// T*

0x3F, 0x40, 0x40, 0x40, 0x3F, *// U*

0x1F, 0x20, 0x40, 0x20, 0x1F, *// V*

0x3F, 0x40, 0x38, 0x40, 0x3F, *// W*

0x63, 0x14, 0x08, 0x14, 0x63, *// X*

0x07, 0x08, 0x70, 0x08, 0x07, *// Y*

0x61, 0x51, 0x49, 0x45, 0x43, *// Z*

0x00, 0x7F, 0x41, 0x41, 0x00, *// [*

0x55, 0x2A, 0x55, 0x2A, 0x55, *// 55*

0x00, 0x41, 0x41, 0x7F, 0x00, *// ]*

0x04, 0x02, 0x01, 0x02, 0x04, *// ^*

0x40, 0x40, 0x40, 0x40, 0x40, *// \_*

0x00, 0x01, 0x02, 0x04, 0x00, *// '*

0x20, 0x54, 0x54, 0x54, 0x78, *// a*

0x7F, 0x48, 0x44, 0x44, 0x38, *// b*

0x38, 0x44, 0x44, 0x44, 0x20, *// c*

0x38, 0x44, 0x44, 0x48, 0x7F, *// d*

0x38, 0x54, 0x54, 0x54, 0x18, *// e*

0x08, 0x7E, 0x09, 0x01, 0x02, *// f*

0x0C, 0x52, 0x52, 0x52, 0x3E, *// g*

0x7F, 0x08, 0x04, 0x04, 0x78, *// h*

0x00, 0x44, 0x7D, 0x40, 0x00, *// i*

0x20, 0x40, 0x44, 0x3D, 0x00, *// j*

0x7F, 0x10, 0x28, 0x44, 0x00, *// k*

0x00, 0x41, 0x7F, 0x40, 0x00, *// l*

0x7C, 0x04, 0x18, 0x04, 0x78, *// m*

0x7C, 0x08, 0x04, 0x04, 0x78, *// n*

0x38, 0x44, 0x44, 0x44, 0x38, *// o*

0x7C, 0x14, 0x14, 0x14, 0x08, *// p*

0x08, 0x14, 0x14, 0x18, 0x7C, *// q*

0x7C, 0x08, 0x04, 0x04, 0x08, *// r*

0x48, 0x54, 0x54, 0x54, 0x20, *// s*

0x04, 0x3F, 0x44, 0x40, 0x20, *// t*

0x3C, 0x40, 0x40, 0x20, 0x7C, *// u*

0x1C, 0x20, 0x40, 0x20, 0x1C, *// v*

0x3C, 0x40, 0x30, 0x40, 0x3C, *// w*

0x44, 0x28, 0x10, 0x28, 0x44, *// x*

0x0C, 0x50, 0x50, 0x50, 0x3C, *// y*

0x44, 0x64, 0x54, 0x4C, 0x44, *// z*

0x00, 0x06, 0x09, 0x09, 0x06 *// º*

};

**static** **const** unsigned char number[14][3][16] PROGMEM = {

{{0,128,192,224,224,96,224,224, *//'0'*

192,128,0,0,0,0,0,0}

,

{112,255,255,1,0,0,0,0,

255,255,254,0,0,0,0,0}

,

{0,15,31,60,56,48,56,56,

31,15,3,0,0,0,0,0}

},

{

{0,0,0,0,128,224,224,0, *//'1'*

0,0,0,0,0,0,0,0}

,

{0,0,3,3,3,255,255,0,

0,0,0,0,0,0,0,0}

,

{0,0,56,56,56,63,63,56,

56,56,0,0,0,0,0,0}

},

{

{0,192,192,224,96,96,224,224, *//'2'*

192,128,0,0,0,0,0,0}

,

{0,1,0,0,128,192,224,249,

63,31,0,0,0,0,0,0}

,

{0,60,62,63,63,59,57,56,

56,56,56,0,0,0,0,0}

},

{

{0,192,224,224,96,96,224,224, *//'3'*

192,192,0,0,0,0,0,0}

,

{0,1,0,0,48,48,56,125,

239,207,0,0,0,0,0,0}

,

{0,28,56,56,48,48,56,60,

31,15,1,0,0,0,0,0}

},

{

{0,0,0,0,0,128,192,224, *//'4'*

224,0,0,0,0,0,0,0}

,

{224,240,248,222,207,199,193,255,

255,192,192,0,0,0,0,0}

,

{0,0,0,0,0,0,0,63,

63,0,0,0,0,0,0,0}

},

{

{0,224,224,224,224,224,224,224, *//'5'*

224,224,224,0,0,0,0,0}

,

{0,63,63,63,56,56,48,112,

240,224,0,0,0,0,0,0}

,

{0,28,56,56,48,48,56,60,

31,15,1,0,0,0,0,0}

},

{

{0,0,128,192,192,224,96,96, *//'6'*

224,224,0,0,0,0,0,0}

,

{224,254,255,55,57,24,24,56,

240,240,192,0,0,0,0,0}

,

{0,15,31,28,56,48,48,56,

31,15,7,0,0,0,0,0}

},

{

{0,224,224,224,224,224,224,224, *//'7'*

224,224,224,0,0,0,0,0}

,

{0,0,0,0,128,224,248,126,

31,7,1,0,0,0,0,0}

,

{0,0,56,62,31,7,1,0,

0,0,0,0,0,0,0,0}

},

{

{0,128,192,224,224,96,96,224, *//'8'*

192,192,0,0,0,0,0,0}

,

{0,207,255,127,56,48,112,112,

255,239,199,0,0,0,0,0}

,

{3,15,31,60,56,48,48,56,

31,31,15,0,0,0,0,0}

},

{

{0,128,192,224,224,96,224,224, *//'9'*

192,128,0,0,0,0,0,0}

,

{12,63,127,241,224,192,192,225,

255,255,254,0,0,0,0,0}

,

{0,0,56,48,48,56,56,30,

15,7,0,0,0,0,0,0}

},

{

{0,0,0,0,0,0,0,0, *//'.'*

0,0,0,0,0,0,0,0}

,

{0,0,0,0,0,0,0,0,

0,0,0,0,0,0,0,0}

,

{60,60,60,0,0,0,0,0,

0,0,0,0,0,0,0,0}

},

{

{0,0,0,0,0,0,0,0, *//'+'*

0,0,0,0,0,0,0,0}

,

{0,0,64,64,64,64,64,254,

254,64,64,64,64,64,0,0}

,

{0,0,0,0,0,0,0,15,

15,0,0,0,0,0,0,0}

},

{

{0,0,0,0,0,0,0,0, *//'-'*

0,0,0,0,0,0,0,0}

,

{0,64,64,64,64,64,64,0,

0,0,0,0,0,0,0,0}

,

{0,0,0,0,0,0,0,0,

0,0,0,0,0,0,0,0}

},

{

{0,0,0,0,0,0,0,128,128,0,0,0,0,0,0,0}, *//':'*

{0,0,0,0,0,0,0,129,129,0,0,0,0,0,0,0},

{0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0}

}

};

void LCD\_Init(void);

void LCD\_WriteCommand(unsigned char command);

void LCD\_WriteData(unsigned char data);

void LCD\_GoTo(uint8\_t x, uint8\_t y);

void LCD\_Clear(void);

void LCD\_Update( void );

void LCD\_DrawSplash(void);

void LCD\_SetPixel(uint8\_t x, uint8\_t y);

void LCD\_ClearPixel(uint8\_t x, uint8\_t y);

void LCD\_WriteChar (unsigned char ch);

void LCD\_WriteString(**const** char \*string);

void LCD\_DisplaySmallInt(uint8\_t i);

void LCD\_WriteInt(uint8\_t i, bool large);

void LCD\_DisplayLargeInt(uint8\_t num);

void LCD\_ClearSmallSpace(void);

int digitAtPos(int number, int pos);

void LCD\_ClearLargeSpace(void);

void LCD\_DrawBell(int x, int y);

void LCD\_WriteColon();

void LCD\_InvertRow();

void LCD\_DrawNote(int x, int y);

void LCD\_DrawBar(int x, int y, uint8\_t height);

*//LCD Display driver for NOKIA 5110*

*//Author: Nick Schrock, Kevin Sager*

*//written in reference to https://github.com/mspiceland/avr-spiceduino-3310-thermistor*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*This module is the primary driver for the LCD Display screen. These functions are*

*used and called in various parts of the system to draw the correct images on to the LCD*

*display. The functions can get fairly intricate but their functionality is mostly*

*self-explanatory.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <avr/io.h>

#include "LCD\_Driver.h"

#include "SPI\_Driver.h"

#include <avr/pgmspace.h>

#include <math.h>

#include <stdbool.h>

int current\_row = 0;

int current\_col = 0;

**static** unsigned char lcd\_buffer[8][84];

*//Splash Screen data. Currently Thermometer*

*//TODO: Customize a cool Splash screen for startup*

unsigned **const** char splash[7][84] PROGMEM = {

{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xc0,0x20,0x20,0xc0,0x00,0xe0,0x20,0x20,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x80,0xc0,0x60,0x20,0x60,0xc0,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,},

{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x01,0x00,0x00,0x1f,0x02,0x02,0x00,0x00,0x80,0x84,0x84,0x84,0x84,0xff,0x00,0x00,0x00,0x00,0x00,0xff,0x84,0x84,0x84,0x04,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,},

{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x20,0x20,0x20,0xff,0x00,0x00,0x00,0x00,0x00,0xff,0x20,0x20,0x20,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,},

{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x84,0x84,0x84,0x84,0xff,0x00,0x00,0x00,0x00,0x00,0xff,0x84,0x84,0x84,0x84,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,},

{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x10,0x10,0x90,0xd0,0x7f,0x00,0x00,0x00,0x00,0x00,0x7f,0xd0,0x90,0x10,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,},

{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x3e,0xe3,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x80,0xe3,0x3e,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,},

{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x01,0x03,0x02,0x02,0x02,0x03,0x01,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,},

};

*//Data Array for drawing a bell*

unsigned **const** char bell[9] PROGMEM = {0x40,0x60,0x7C,0xFE,0xFF,0xFE,0x7C,0x60,0x40};

unsigned **const** char note[9] PROGMEM = {0x07,0x01,0x4D,0xE4,0xE0,0xE0,0x7F,0x06,0x0C};

*//Data Array for drawing a large colon*

*//TODO: Make it look better / more defined*

unsigned **const** char colon[3][16] = {

{0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0}, *//':'*

{0,0,0,0,0,0,0,129,129,0,0,0,0,0,0,0},

{0,0,0,0,0,0,0,128,128,0,0,0,0,0,0,0}

};

*//Initialize LCD by clearing and resetting pins*

void LCD\_Init(void){

CLEAR\_RST\_PIN;

CLEAR\_SCE\_PIN;

SET\_RST\_PIN;

SET\_SCE\_PIN;

LCD\_WriteCommand( 0x21 ); *// LCD Extended Commands.*

LCD\_WriteCommand( 0xC0 ); *// Set LCD Vop (Contrast).*

LCD\_WriteCommand( 0x04 ); *// Set Temp coefficent.*

LCD\_WriteCommand( 0x13 ); *// LCD bias mode 1:48.*

LCD\_WriteCommand( 0x20 ); *// LCD Standard Commands, Horizontal addressing mode.*

LCD\_WriteCommand( 0x0c ); *// LCD in normal mode.*

LCD\_Clear();

}

*//Sends commands to LCD control register*

void LCD\_WriteCommand(unsigned char command){

CLEAR\_SCE\_PIN; *//disable LCD*

CLEAR\_DC\_PIN; *//set to command mode*

SPI\_Master\_Send(command);

}

*//Writes 8 vertically aligned pixels*

void LCD\_WriteData(unsigned char data){

CLEAR\_SCE\_PIN;

SET\_DC\_PIN;

SPI\_Master\_Send(data);

}

*//Places cursor at specific location*

void LCD\_GoTo(uint8\_t x, uint8\_t y){

LCD\_WriteCommand(0x80 | x);

LCD\_WriteCommand(0x40 | y);

current\_row = y;

current\_col = x;

}

*//Clears Screen*

void LCD\_Clear(void){

int i,j;

LCD\_GoTo(0,0); *//start with (0,0) position*

**for**(i=0; i<8; i++)

{

**for**(j=0; j<90; j++)

{

LCD\_WriteData( 0x00 );

**if** ((i < 6) && (j < 84))

lcd\_buffer[i][j] = 0x00;

}

}

LCD\_GoTo(0,0); *//bring the XY position back to (0,0)*

}

*//Write Zero's to 5 spaces and return to original position*

void LCD\_ClearSmallSpace(void){

**for**(int i=0; i<6; i++){

lcd\_buffer[current\_row][current\_col+i] = 0x00;

}

}

*//Writes 0's to large space and returns to original position*

void LCD\_ClearLargeSpace(void){

**for**(int i=0; i<3; i++){

**for**(int j=0; j<10; j++){

**if**((current\_row+i)<10 && (current\_col+j)<94)

lcd\_buffer[current\_row-i][current\_col+j] = 0x00;

}

}

}

*//Updates screen to whatever is contained in the buffer*

void LCD\_Update( void )

{

int i,j;

LCD\_GoTo(0,0); *//start with (0,0) position*

**for**(i=0; i<7; i++)

{

LCD\_GoTo(0,i);

**for**(j=0; j<84; j++)

{

LCD\_WriteData(lcd\_buffer[i][j]);

}

}

LCD\_GoTo(0,0); *//bring the XY position back to (0,0)*

}

*//Draws whatever is in the Splash data array onto the screen*

void LCD\_DrawSplash(void){

int i,j;

**for**(i=0; i<7; i++){

**for**(j=0; j<84; j++){

lcd\_buffer[i][j] = pgm\_read\_byte(&(splash[i][j]));

}

}

LCD\_Update();

}

*//Draws Colon data array*

void LCD\_WriteColon(){

LCD\_DisplayLargeInt(13);

}

*//Draws Alarm Indicator Bell*

void LCD\_DrawBell(int x, int y){

**for**(int i=0; i<9; i++){

lcd\_buffer[x][y+i] = pgm\_read\_byte(&(bell[i]));

}

LCD\_Update();

}

void LCD\_DrawNote(int x, int y){

**for**(int i=0; i<9; i++){

lcd\_buffer[x][y+i] = pgm\_read\_byte(&(note[i]));

}

LCD\_Update();

}

void LCD\_DrawBar(int y, int x, uint8\_t height){

lcd\_buffer[x][y] = height;

lcd\_buffer[x][y+1] = height;

LCD\_Update();

}

*//Sets one individual pixel*

void LCD\_SetPixel(uint8\_t x, uint8\_t y){

uint8\_t row = y/8;

uint8\_t value;

value = lcd\_buffer[row][x];

value |= (1 << (y%8));

lcd\_buffer[row][x] = value;

LCD\_GoTo(x,row);

LCD\_Update();

}

*//Clears one individual pixel*

void LCD\_ClearPixel( uint8\_t x, uint8\_t y )

{

unsigned char value;

unsigned char row;

row = y / 8;

value = lcd\_buffer[row][x];

value &= ~(1 << (y % 8));

lcd\_buffer[row][x] = value;

LCD\_Update();

LCD\_GoTo(x,row);

}

*//Writes character to screen*

void LCD\_WriteChar (unsigned char ch)

{

unsigned char j;

*//clear a small space before every writing a character to ensure no overlap*

LCD\_ClearSmallSpace();

lcd\_buffer[current\_row][current\_col] = 0x00;

**for**(j=0; j<5; j++)

lcd\_buffer[current\_row][current\_col + j] |= pgm\_read\_byte(&(smallFont [(ch-32)\*5 + j] ));

lcd\_buffer[current\_row][current\_col + 6] = 0x00;

*//Write data in buffer to screen*

**for**(j=0; j<7; j++)

LCD\_WriteData(lcd\_buffer[current\_row][current\_col++]);

}

*//Write data type Integer to screen. Used for displaying Time/date*

void LCD\_WriteInt (uint8\_t i, bool large){

int first\_digit = digitAtPos(i,2);

int second\_digit = digitAtPos(i,1);

**if** (i<10){

**if**(large){

*//LCD\_ClearLargeSpace();*

LCD\_DisplayLargeInt(i);

}

**else**{

LCD\_ClearSmallSpace();

LCD\_DisplaySmallInt(i);

}

}

**else**{

**if**(large){

**if**(second\_digit == 0)

LCD\_ClearLargeSpace();

LCD\_DisplayLargeInt(first\_digit);

LCD\_GoTo(current\_col+13, current\_row);

LCD\_ClearLargeSpace();

LCD\_DisplayLargeInt(second\_digit);

}

**else**{

**if**(second\_digit == 0)

LCD\_ClearSmallSpace();

LCD\_DisplaySmallInt(first\_digit);

LCD\_ClearSmallSpace();

LCD\_DisplaySmallInt(second\_digit);

}

}

}

*//Write a small integer to screen*

void LCD\_DisplaySmallInt(uint8\_t i){

int j;

lcd\_buffer[current\_row][current\_col] = 0x00;

**for**(j=0; j<5; j++)

lcd\_buffer[current\_row][current\_col + j] |= pgm\_read\_byte(&(smallFont [(i+16)\*5 + j] ));

lcd\_buffer[current\_row][current\_col + 6] = 0x00;

**for**(j=0; j<7; j++)

LCD\_WriteData(lcd\_buffer[current\_row][current\_col++]);

}

*//Actually draws the large integer to the screen...*

*//The reason for this function is to factor out this mildly complex algorithm*

void LCD\_DisplayLargeInt(uint8\_t num){

int i,j;

**for**(i=0;i<3;i++)

{

LCD\_GoTo(current\_col, i);

**for**(j=0; j<16; j++) {

lcd\_buffer[current\_row][current\_col + j] = pgm\_read\_byte(&(number[num][i][j]));

LCD\_WriteData(lcd\_buffer[current\_row][current\_col + j]);

}

}

}

int digitAtPos(int number, int pos)

{

**return** ( number / (int)pow(10.0, pos-1) ) % 10;

}

void LCD\_WriteString(**const** char \*string)

{

**while** ( \*string )

LCD\_WriteChar( \*string++ );

}

*//Highlights a row on the screen*

void LCD\_InvertRow(int row){

**for**(int i=0; i<84; i++){

lcd\_buffer[row][i] ^= 0xFF;

}

LCD\_Update();

}

*/\**

*\* Header definitions for RTC*

*\**

*\* Created: 9/26/2014 2:00:04 PM*

*\* Author: Nick Schrock, Kevin Sager*

*\*/*

#define SECONDS\_REGISTER 0x00

#define MINUTES\_REGISTER 0x01

#define HOURS\_REGISTER 0x02

#define DAYOFWK\_REGISTER 0x03

#define DAYS\_REGISTER 0x04

#define MONTHS\_REGISTER 0x05

#define YEARS\_REGISTER 0x06

#include <stdbool.h>

uint8\_t DecimalToBCD (uint8\_t decimalByte);

uint8\_t BCDToDecimal (uint8\_t bcd);

void get\_Time(void);

void set\_Time(uint8\_t hours, uint8\_t minutes, uint8\_t seconds);

void set\_Date(uint8\_t months, uint8\_t days, uint8\_t years);

uint8\_t get\_Hours();

uint8\_t get\_Minutes();

uint8\_t get\_Seconds();

uint8\_t get\_Month();

uint8\_t get\_Year();

uint8\_t get\_Day();

void RTC\_DisplayDate(int xpos, int ypos);

void RTC\_DisplayLargeClock(int xpos, int ypos);

void RTC\_DisplaySmallClock(int xpos, int ypos);

void RTC\_ClearVariables();

int RTC\_GetPM();

void RTC\_TogglePM();

*//Code to Drive Real time clock*

*//Author: @NickSchrock*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*This module is dedicated to controlling the functions of the DS1407 Real-Time Clock IC.*

*The primary functions used in the main state are the function that displays the large*

*clock. This is the primary display for the idle state so it must be accurate.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <stdio.h>

#include <avr/io.h>

#include "RTC\_Driver.h"

#include "TWI\_Driver.h"

#include <math.h>

#include <stdbool.h>

#include "LCD\_Driver.h"

#include "Alarm\_Driver.h"

#define F\_CPU 16000000UL

#define BAUD 9600

#define MYUBRR F\_CPU/16/BAUD-1

#define LARGE\_SPACE 13

#define SMALL\_SPACE 6

int displayed\_hours=100;

int displayed\_minutes=100;

int displayed\_seconds=100;

int displayed\_month=100;

int displayed\_day=100;

int displayed\_year=100;

bool pm = false;

uint8\_t BCDToDecimal (uint8\_t bcd)

{

**return** ((bcd >> 4)\*10+(bcd & 0x0F)); *//Converts BCD formatted bit to a decimal formatted bit*

}

uint8\_t DecimalToBCD (uint8\_t decimalByte)

{

**return** (((decimalByte / 10) << 4) | (decimalByte % 10));

}

*//Subroutine to read hour, minute, second*

void get\_Time(void){

*//Set 8 bit variables to value of register in RTC*

uint8\_t hours\_byte = I2C\_ReadRegister(DS1307,HOURS\_REGISTER);

uint8\_t minutes\_byte = I2C\_ReadRegister(DS1307,MINUTES\_REGISTER);

uint8\_t seconds\_byte = I2C\_ReadRegister(DS1307,SECONDS\_REGISTER);

**if** (hours\_byte & 0x40) *// 12hr mode:*

hours\_byte &= 0x1F; *// use bottom 5 bits (pm bit = temp & 0x20)*

**else**

hours\_byte &= 0x3F; *// 24hr mode: use bottom 6 bits*

*//Convert to Decimal format*

hours\_byte = BCDToDecimal (hours\_byte);

minutes\_byte = BCDToDecimal (minutes\_byte);

seconds\_byte = BCDToDecimal (seconds\_byte);

}

int RTC\_GetPM(){

**return** pm;

}

void RTC\_TogglePM(){

printf("Before toggle: %i **\n**", pm);

pm ^= true;

printf("After toggle %i**\n**", pm);

}

uint8\_t get\_Hours(){

int hours;

uint8\_t hours\_byte = BCDToDecimal(I2C\_ReadRegister(DS1307,HOURS\_REGISTER));

hours = hours\_byte;

**if**(hours\_byte > 12){

hours -=12;

}

**if**(hours\_byte == 0){

hours = 12;

}

**if**(hours\_byte >= 12){

pm = true;

}

**else**{

pm = false;

}

**return** hours;

}

uint8\_t get\_Minutes()

{

**return** BCDToDecimal(I2C\_ReadRegister(DS1307,MINUTES\_REGISTER));

}

uint8\_t get\_Seconds()

{

uint8\_t seconds = I2C\_ReadRegister(DS1307,SECONDS\_REGISTER);

uint8\_t seconds\_dec = BCDToDecimal(seconds);

**return** seconds\_dec;

}

uint8\_t get\_Month(){

**return** BCDToDecimal(I2C\_ReadRegister(DS1307, MONTHS\_REGISTER));

}

uint8\_t get\_Day(){

**return** BCDToDecimal(I2C\_ReadRegister(DS1307, DAYS\_REGISTER));

}

uint8\_t get\_Year(){

**return** BCDToDecimal(I2C\_ReadRegister(DS1307, YEARS\_REGISTER));

}

void get\_Date(void){

*//Set 8 bit variables to value of register in RTC*

uint8\_t months\_byte = I2C\_ReadRegister(DS1307,MONTHS\_REGISTER);

uint8\_t days\_byte = I2C\_ReadRegister(DS1307,DAYS\_REGISTER);

uint8\_t years\_byte = I2C\_ReadRegister(DS1307,YEARS\_REGISTER);

*//Convert to Decimal format*

months\_byte = BCDToDecimal (months\_byte);

days\_byte = BCDToDecimal (days\_byte);

years\_byte = BCDToDecimal (years\_byte);

}

void set\_Time(uint8\_t hours, uint8\_t minutes, uint8\_t seconds)

{

I2C\_WriteRegister(DS1307,HOURS\_REGISTER, DecimalToBCD(hours));

I2C\_WriteRegister(DS1307,MINUTES\_REGISTER, DecimalToBCD(minutes));

I2C\_WriteRegister(DS1307,SECONDS\_REGISTER, DecimalToBCD(seconds));

}

*//Use this function to hard-code in an initial date to the RTC*

void set\_Date(uint8\_t months, uint8\_t days, uint8\_t years)

{

I2C\_WriteRegister(DS1307,MONTHS\_REGISTER, DecimalToBCD(months));

I2C\_WriteRegister(DS1307,DAYS\_REGISTER, DecimalToBCD(days));

I2C\_WriteRegister(DS1307,YEARS\_REGISTER, DecimalToBCD(years));

}

void RTC\_DisplaySmallClock(int xpos, int ypos)

{

**while**(1)

{

**if**(get\_Hours()>9)

xpos -=7; *// xpos = xpos - 7*

LCD\_Clear();

LCD\_GoTo(xpos, ypos);

LCD\_WriteInt(get\_Hours(),false);

LCD\_WriteChar(':');

**for**(int i=get\_Minutes(); i<60; i++)

{

LCD\_GoTo(xpos+12,ypos);

**if**(get\_Minutes()<10){

LCD\_WriteInt(0,false);

}

LCD\_WriteInt(get\_Minutes(),false);

LCD\_WriteChar(':');

**for**(int j=get\_Seconds(); j<60; j++){

LCD\_GoTo(xpos+30,ypos);

**if**(get\_Seconds()<10){

LCD\_WriteInt(0,false);

}

LCD\_WriteInt(get\_Seconds(),false);

one\_sec\_delay();

}

}

}

}

void RTC\_DisplayDate(int xpos, int ypos){

char update;

int month\_pos1 = xpos;

int month\_pos2 = month\_pos1 + SMALL\_SPACE;

int slash\_pos1 = month\_pos2 + SMALL\_SPACE;

int day\_pos1 = slash\_pos1 +SMALL\_SPACE;

int day\_pos2 = day\_pos1 + SMALL\_SPACE;

int slash\_pos2 = day\_pos2 + SMALL\_SPACE;

int year\_pos1 = slash\_pos2 + SMALL\_SPACE;

int year\_pos2 = year\_pos1 + SMALL\_SPACE;

**if**(get\_Year() != displayed\_year){

update = 'Y';

displayed\_year = get\_Year();

displayed\_month = get\_Month();

displayed\_day = get\_Day();

}

**else** **if**(get\_Month() != displayed\_month){

update = 'M';

displayed\_month = get\_Month();

displayed\_day = get\_Day();

}

**else** **if**(get\_Day() != displayed\_day){

update = 'D';

displayed\_day = get\_Day();

}

**else** {

update = 'N';

}

**switch**(update)

{

**case** 'Y':

*//Write year to screen*

*//LCD\_Clear();*

LCD\_GoTo(year\_pos1, ypos);

**if**(displayed\_year<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(year\_pos2,ypos);

}

LCD\_WriteInt(displayed\_year,false);

*//Write month to Screen*

**if**(get\_Month()>9){

LCD\_GoTo(month\_pos1, ypos);

}

**else**{

LCD\_GoTo(month\_pos1, ypos);

LCD\_WriteChar(' ');

LCD\_GoTo(month\_pos2,ypos);

}

LCD\_WriteInt(displayed\_month,false);

LCD\_GoTo(slash\_pos1, ypos);

LCD\_WriteChar('/');

*//Write day to Screen*

LCD\_GoTo(day\_pos1, ypos);

**if**(displayed\_day<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(day\_pos2, ypos);

}

LCD\_WriteInt(displayed\_day,false);

LCD\_GoTo(slash\_pos2, ypos);

LCD\_WriteChar('/');

**break**;

**case** 'M':

*//Write month to Screen*

**if**(get\_Month()>9){

LCD\_GoTo(month\_pos1, ypos);

}

**else**{

LCD\_GoTo(month\_pos1, ypos);

LCD\_WriteChar(' ');

LCD\_GoTo(month\_pos2,ypos);

}

LCD\_WriteInt(displayed\_month,false);

LCD\_GoTo(slash\_pos1, ypos);

LCD\_WriteChar('/');

*//Write day to Screen*

LCD\_GoTo(day\_pos1, ypos);

**if**(displayed\_day<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(day\_pos2, ypos);

}

LCD\_WriteInt(displayed\_day,false);

LCD\_GoTo(slash\_pos2, ypos);

LCD\_WriteChar('/');

**break**;

**case** 'D':

*//Write day to Screen*

LCD\_GoTo(day\_pos1, ypos);

**if**(displayed\_day<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(day\_pos2, ypos);

}

LCD\_WriteInt(displayed\_day,false);

LCD\_GoTo(slash\_pos2, ypos);

LCD\_WriteChar('/');

**break**;

**case** 'N':

**break**;

}

}

void RTC\_DisplayLargeClock(int xpos, int ypos){

char update;

int hours\_pos1 = xpos - LARGE\_SPACE;

int hours\_pos2 = xpos;

int colon\_pos = hours\_pos2 + LARGE\_SPACE;

int minutes\_pos1 = colon\_pos + LARGE\_SPACE;

int minutes\_pos2 = minutes\_pos1 + LARGE\_SPACE;

LCD\_GoTo(70, ypos);

**if**(pm == true){

LCD\_WriteString("PM");

}

**else**{

LCD\_WriteString("AM");

}

**if**(get\_Hours() != displayed\_hours){

update = 'H';

displayed\_hours = get\_Hours();

displayed\_minutes = get\_Minutes();

displayed\_seconds = get\_Seconds();

}

**else** **if**(get\_Minutes() != displayed\_minutes){

update = 'M';

displayed\_minutes = get\_Minutes();

displayed\_seconds = get\_Seconds();

}

**else** **if**(get\_Seconds() != displayed\_seconds){

update = 'S';

displayed\_seconds = get\_Seconds();

}

**else** {

update = 'N';

}

**switch**(update)

{

**case** 'H':

*//Write Hours to screen*

**if**(displayed\_hours>9){

hours\_pos2 = hours\_pos1;

}

*//LCD\_Clear();*

LCD\_GoTo(hours\_pos1, ypos-1);

LCD\_WriteString(" ");

LCD\_GoTo(hours\_pos1, ypos);

LCD\_WriteString(" ");

LCD\_GoTo(hours\_pos1, ypos-2);

LCD\_WriteString(" ");

LCD\_GoTo(hours\_pos2, ypos);

LCD\_WriteInt(displayed\_hours,true);

LCD\_GoTo(colon\_pos, ypos);

LCD\_WriteColon();

*//Write Minutes to Screen*

LCD\_GoTo(minutes\_pos1,ypos);

**if**(displayed\_minutes<10){

LCD\_WriteInt(0,true);

LCD\_GoTo(minutes\_pos2,ypos);

}

LCD\_WriteInt(displayed\_minutes,true);

*//Write Seconds to Screen*

LCD\_GoTo(70,1);

**if**(displayed\_seconds<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(76,1);

}

LCD\_WriteInt(displayed\_seconds,false);

**break**;

**case** 'M':

*//Write Minutes to Screen*

LCD\_GoTo(minutes\_pos1,ypos);

**if**(displayed\_minutes<10){

LCD\_WriteInt(0,true);

LCD\_GoTo(minutes\_pos2,ypos);

}

LCD\_WriteInt(displayed\_minutes,true);

*//Write Seconds to Screen*

LCD\_GoTo(70,1);

**if**(displayed\_seconds<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(76,1);

}

LCD\_WriteInt(displayed\_seconds,false);

**break**;

**case** 'S':

*//Write Seconds to Screen*

LCD\_GoTo(70,1);

**if**(displayed\_seconds<10){

LCD\_WriteInt(0,false);

LCD\_GoTo(76,1);

}

LCD\_WriteInt(displayed\_seconds,false);

**case** 'N':

**break**;

}

}

void RTC\_ClearVariables(void){

displayed\_seconds=100;

displayed\_minutes=100;

displayed\_hours=100;

displayed\_month=100;

displayed\_day=100;

displayed\_year=100;

}

*/\**

*\* SPI\_Driver.h*

*\**

*\* Created: 9/24/2014 3:59:18 PM*

*\* Author: Nick Schrock, Kevin Sager*

*\*/*

void SPI\_Master\_Init(void);

void SPI\_Master\_Send(unsigned char data);

*//SPI driver for the ATMega328P*

*//Author: Nick Schrock, Kevin Sager*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*This is a short module that is called to initialize the SPI protocol and send data.*

*It is relatively short because a lot of the work is done by the only SPI device in the*

*system (the LCD Display).*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*//I would like to eventually modify this to be a multi-functional SPI driver much*

*//like TWI\_Driver.c*

#include <avr/io.h>

void SPI\_Master\_Init(void){

*//Set PB2(~SS), PB3(MOSI), PB5(SCK) as outputs*

DDRB |= \_BV(2) | \_BV(3) |\_BV(5) |\_BV(4) |\_BV(0);

*//Make SS high*

PORTB |= \_BV(2);

*//Enable SPI in Master Mode (Prescalar Fck/16)*

SPCR |= \_BV(SPE) | \_BV(MSTR) | \_BV(SPR0);

}

*/\*void SPI\_Master\_Init\_Interrupt(void){*

*DDRB |= \_BV(4) | \_BV(5) | \_BV(7);*

*SPCR |= \_BV(SPIE) | \_BV(SPE) | \_BV(MSTR) | \_BV(SPR0);*

*}\*/*

void SPI\_Master\_Send(unsigned char data){

*//Select Slave*

*//PORTB &= ~(\_BV(2));*

*//Set data register to send*

SPDR = data;

*//wait for trasmission to complete*

**while**(!(SPSR & \_BV(SPIF)));

*//Make SS high*

*//PORTB |= \_BV(2);*

}

*//TWI master control header definitions*

*//Written with reference to http://w8bh.net/avr/AvrDS1307.pdf*

*//Author: Nick Schrock, Kevin Sager*

#define TW\_START 0xA4

#define TW\_READY (TWCR & 0x80) *//Check if TWINT bit in TWCR control register is high*

#define TW\_STATUS (TWSR & 0xF8) *//returns value of status register*

#define DS1307 0xD0 *//references address of RTC (DS1307)*

#define TW\_SEND 0x84 *//sets TWINT and TWEN (basically start sending data)*

#define TW\_STOP 0x94 *//sets TWINT, TWSTO, TWEN (sets stop condition)*

#define TW\_NACK 0x84

#define READ 1

void I2C\_WriteRegister(uint8\_t device\_address, uint8\_t deviceRegister, uint8\_t data);

uint8\_t I2C\_ReadRegister(uint8\_t device\_address, uint8\_t deviceRegister);

void I2C\_Init();

uint8\_t I2C\_Start();

uint8\_t I2C\_SendAddress(uint8\_t address);

uint8\_t I2C\_Write (uint8\_t data);

void I2C\_Stop();

uint8\_t I2C\_ReadNACK();

uint16\_t I2C\_ReadRegister\_16bit();

void I2C\_WriteRegister\_16bit(uint16\_t data);

void I2C\_StartWait(unsigned char address);

uint8\_t I2C\_ReadACK();

*///TWI\_Driver.c*

*// Nick Schrock, Kevin Sager*

*// Written with reference to http://w8bh.net/avr/AvrDS1307.pdf*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*This is the driver module for the two wire I2C communication protocol. The functions in*

*this module are used by the microcontroller to send and receive data from the SI4703 FM*

*receiver and the DS1407 Real-Time clock. This module is vital for the overall system*

*because it enables communication between the microcontroller and the various peripheral*

*devices.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <stdio.h>

*//#include "uart.h"*

#include <avr/io.h> *//Library for use with AVR*

#include <avr/interrupt.h> *//Library for AVR Interrupts*

#define F\_CPU 16000000UL

#define BAUD 9600

#define MYUBRR F\_CPU/16/BAUD-1

#define F\_SCL 100000 *//I2C clock speed*

*//#include <util/twi.h> //Header for built in functions 328P (delays)*

#include <stdlib.h> *//Header for studio library*

#include "TWI\_Driver.h" *//'Home made' header for Two Wire Interface Drivers to be called*

#include <util/delay.h>

*//Initialize Bit register to proper frequency*

void I2C\_Init(){

TWSR = 0; *//set prescalar to 0*

TWBR = ((F\_CPU/F\_SCL)-16)/2; *//Set Bit register to SCL frequency*

}

uint8\_t I2C\_Start(){

TWCR = TW\_START; *//send start condition (TWINT, TWISTA, TWEN)*

**while**(!TW\_READY); *//wait for TWINT bit in TWCR to be logic 0*

**return** (TW\_STATUS==0x08); *//return 1 if found returns value of status register 0000 1000*

}

uint8\_t I2C\_SendAddress(uint8\_t address){

TWDR = address; *//load slave address*

TWCR = TW\_SEND; *//send the address (TWINT, TWEN)*

**while**(!TW\_READY); *//wait to be ready TWINT in TWCR to be logic 0*

**return**(TW\_STATUS==0x18); *//return 1 if found returns value of status register 0001 1000*

}

uint8\_t I2C\_Write (uint8\_t data){

TWDR = data; *//TWDR = Data register*

TWCR = TW\_SEND; *//Conrol Register (TWINT and TWEN)*

**while**(!TW\_READY); *//wait for TWINT low*

**return** (TW\_STATUS!=0x28); *//return 1 if found Returns Ack SLave addr;register 1111 1000*

}

void I2C\_Stop(){

TWCR = TW\_STOP; *//send stop condition*

}

uint8\_t I2C\_ReadNACK(){*// reads a data byte from slave*

*//printf("Made it to Read\_Nack function\n");*

TWCR = TW\_NACK; *// nack last byte = not reading more data after*

**while**(!TW\_READY); *//wait TWINT not logic 1*

*//printf("TWINT is pulled low\n");*

**return** TWDR; *//return whats in the Data Register*

}

uint16\_t I2C\_Read(){

**while**(!TW\_READY);

**return** TWDR;

}

uint8\_t I2C\_ReadACK(){

TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);

**while**(!(TWCR & (1<<TWINT)));

**return** TWDR;

}

void I2C\_WriteRegister(uint8\_t device\_address, uint8\_t deviceRegister, uint8\_t data){

I2C\_Start(); *//Start Condition (TWINT,TWSTA,TWEN)*

printf("I2C started**\n**");

I2C\_SendAddress(device\_address);*//Send Master DS1307 Addr with 'Ack' code*

printf("Device address sent**\n**");

I2C\_Write(deviceRegister); *//Load, send TWCR, and wait TWINT cleared*

I2C\_Write(data); *//Interrupt, enable, wait, Ack*

I2C\_Stop(); *//Stop Condition: (TWINT(set),TWSTO,TWEN)*

}

void I2C\_WriteRegister\_16bit(uint16\_t data){

uint8\_t low\_byte = data & 0x00FF;

*//printf("Wrote low byte: %i\n", low\_byte);*

uint8\_t high\_byte = data >> 8;

*//printf("Wrote high byte: %i\n", high\_byte);*

I2C\_Write(high\_byte);

\_delay\_ms(1);

I2C\_Write(low\_byte);

}

uint8\_t I2C\_ReadRegister(uint8\_t device\_address, uint8\_t deviceRegister){

uint8\_t data = 0; *//Initialize what will be sent so no confusion*

*//printf("Reading Register\n");*

I2C\_Start(); *//set\_up-->TWINT, TWSTA, TWEN*

*//printf("I2C started\n");*

I2C\_SendAddress(device\_address); *//Master RTC Addr with ACK*

*//printf("Address Sent\n");*

I2C\_Write(deviceRegister); *//Load, send BID, wait, returns Ack SLave*

*//printf("sent register address\n");*

I2C\_Start(); *//set\_up-->TWINT, TWSTA, TWEN*

*//printf("i2c started again\n");*

I2C\_SendAddress(device\_address+READ); *//Send Master RTC Addr with Ack, and Read Bit (0xd1)*

*//printf("Sent address with read bit\n");*

data = I2C\_ReadNACK(); *//Master Requests byte with Nack(0x58)*

*//printf("read nack\n");*

I2C\_Stop(); *//TWINT set, TWSTO and TWEN*

*//printf("I2C stopped\n");*

**return** data; *//Read from SLave's info from TWI Ack Status Code*

}

uint16\_t I2C\_ReadRegister\_16bit(){

uint16\_t data = 0;

uint8\_t high\_byte = I2C\_ReadACK();

\_delay\_ms(1);

uint8\_t low\_byte = I2C\_ReadACK(); *//Master Requests byte with Nack(0x58)*

**while**(!TW\_READY);

*//printf("Read High Byte: %i\n", high\_byte);*

*//printf("Read Low Byte: %i\n", low\_byte);*

data = high\_byte << 8;

data |= low\_byte;

**return** data; *//Read from SLave's info from TWI Ack Status Code*

}