Apple

iSnooze™

"wake easier—one song at a time"

EE459, Spring 2008
Professor Weber/Professor Redekopp
Senior Design Project
Team 1A: Tony Chen, Nate Houk, Ashwin Sathe

1. Introduction

Our project team designed, prototyped, and debugged a working iPod integrated alarm clock named the Apple iSnooze™. With the help of a marketing team, the iSnooze was designed to adhere to the simple, stylish and innovative ideals for which the Apple brand has become famous. We enjoyed using our creativity to select features and components for our design and gained valuable experience in debugging and building a working embedded system. Having completed this alarm clock, our group has learned many hard and soft skills we hope will enable us to be more successful in the real world.

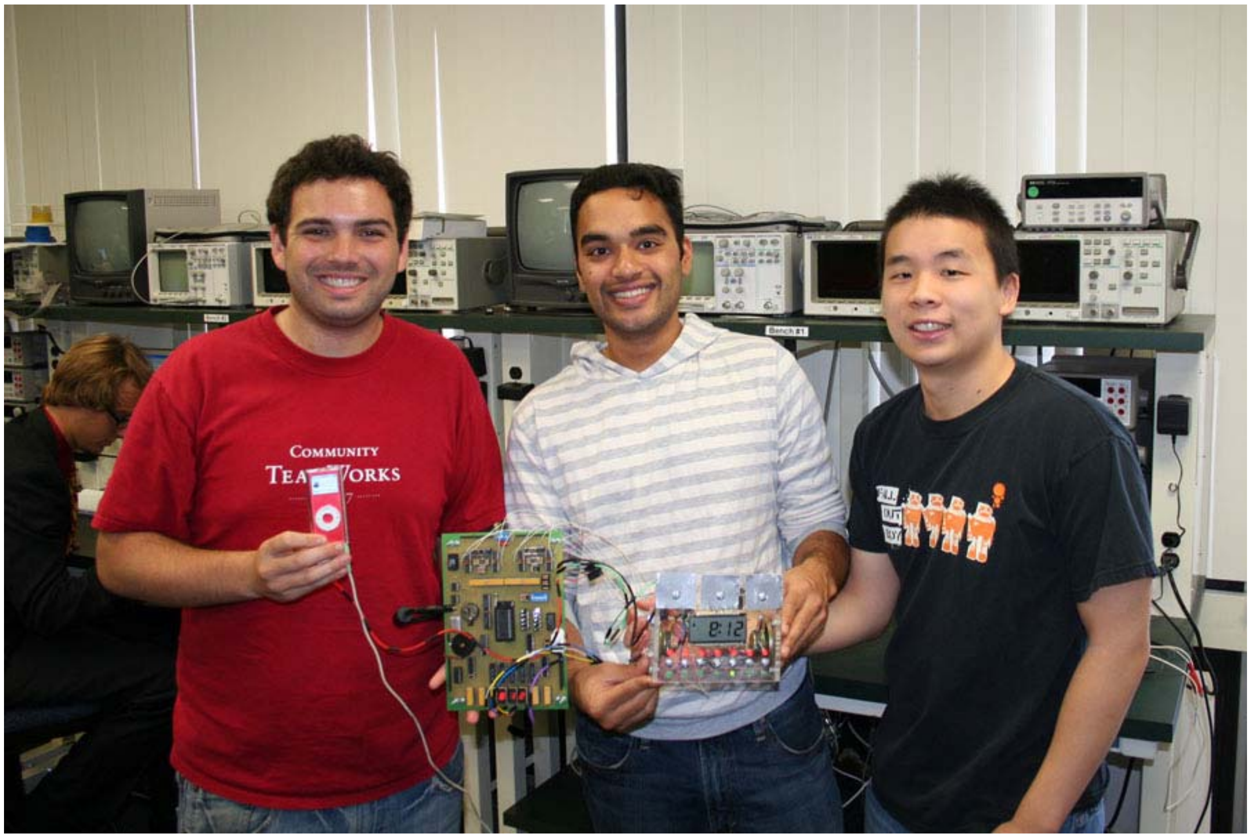


Photo 1 - Members of Team 1A (from left to right): Nate Houk, Ashwin Sathe and Tony Chen.

2. Objective

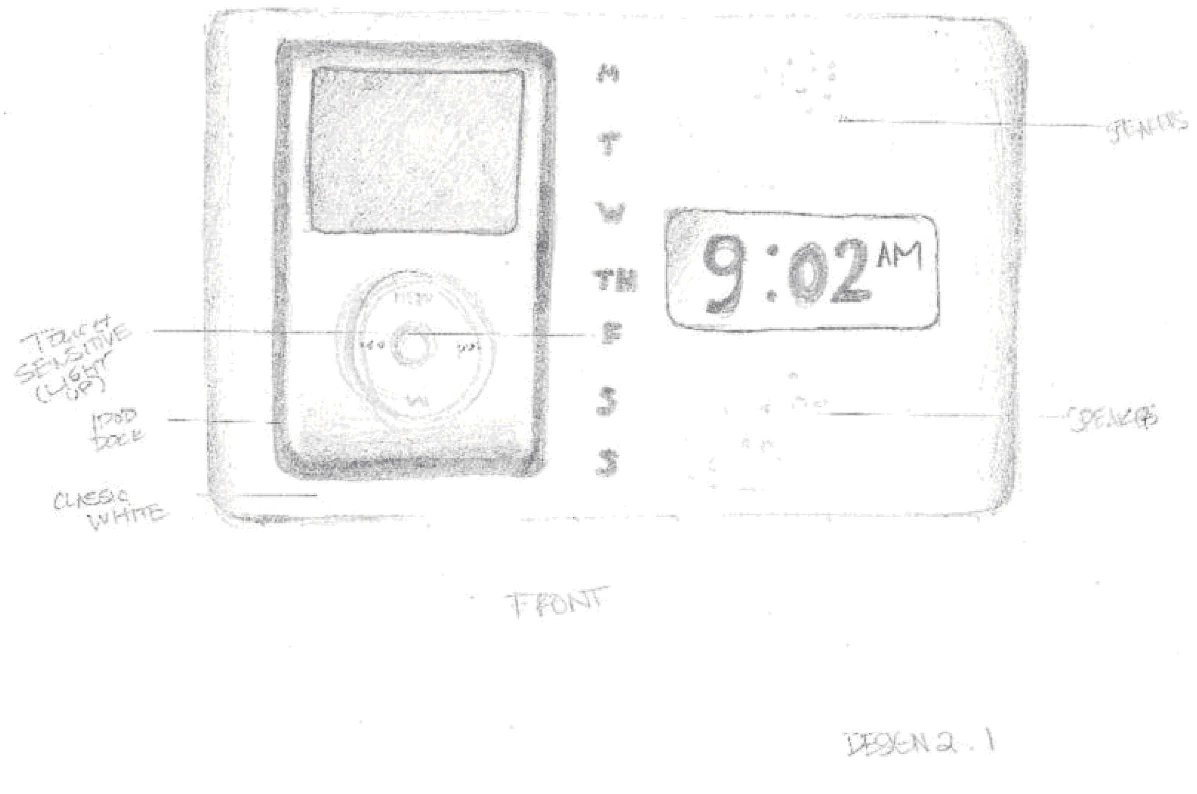
The main goal in designing the clock was to adhere as closely as possible to the Apple product design template. The Apple brand blends innovation and simplicity into a sleek, aesthetically pleasing, user-friendly and intuitive package. Appearance and ease of use were two of the most important aspects considered while designing the clock. Furthermore, the clock should avoid "feature creep" and the resulting complicated and tedious menus that clutter an interface. The interface should allow the user to access the features they want at the push of a button. Lastly, the clock should have an ergonomic design and the input scheme should be straightforward yet elegant.

3. Features

In our quest to create the ideal alarm clock, we had a chance to be innovative and inventive to create a product that possessed all the functions of a standard alarm clock, yet stands out as a unique product with novel features.

3.1. Touch Sensitive Buttons

Integral to Apple products is their sleek stylish design. It was important to design the interface and features to accommodate a stylish alarm casing. Apple is a pioneer in the area of touch sensitivity as seen in many popular Apple products (iPod, iPhone, iMac, etc). Touch buttons can be uniquely integrated into the product casing allowing for a sleek and stylish design. It was therefore decided to use touch buttons to allow for a case that had zero visible buttons—instead it would have simple touch regions. At first glance, it would look like there were no buttons at all similar to how the iPhone has a button-less appearance. Drawing 1 shows the final artist rendition of the Apple iSnooze.



Drawing 1 - Artist rendition of the iSnooze.

3.2. iPod Integration

Market research showed that one of the main consumer requests was iPod integration. The iSnooze "music for your alarm" feature makes the iSnooze one of the most exciting, functional and versatile alarm clocks on the market. The user interface is simple to understand (Appendix E) and offers full control of an iPod as well as enables consumers to wake up to music as opposed to an annoying buzzer. It also allows the clock to be used as a fully functional iPod speaker system, allowing the user to play music at their discretion. Furthermore, the system automatically charges the iPod any time it is connected to the clock.

3.3. 7-Day Alarms

The alarm clock features a 7-day alarm system, allowing for one alarm for each day of the week. Each alarm can be independently set and enabled at the touch of a button. The alarms status is indicated on the front panel with a single LED for each alarm.

3.4. Standard Features

The iSnooze also incorporates all the standard features found on most alarm clocks on the market: accurate time keeping, snooze, and a battery backup which ensure the time and alarms are retained even when power is lost.

4. Cost Analysis

On average, the Apple Corporation records a 15% profit margin on its business[1]. Using this as a goal, we set out to make an affordable and profitable clock for Apple. We asked our marketing team to take surveys to figure out what consumers might be willing to pay for this product. The marketing team's research showed that people would be willing to pay between $49.99 $59.99 for a quality alarm clock with iPod integration. We were trying to be as realistic as possible and decided to use the $49.99 value as a comparison point for our profit margin.

To save cost, we designed a system that implemented a simple parallel 7-segment liquid crystal display along with generic LEDs. The screen gave us a quality, sleek look, at a reduced price. Likewise, the simple LEDs used to display the day of the week and alarm settings could be stylishly enclosed in a case and reduced software complexity. Together, the combination of the simple 7-segment LCD and LEDs helped avoid us the monstrous cost and complex coding associated with expensive graphic screens.

[1] http://finance.yahoo.com/q/ks?s=Aapl

We were therefore quite surprised, when after compiling all the costs for our parts, the cost of making our finished clock was $45.96 (Appendix D). Unfortunately, an unforeseen consequence of the design we used was the introduction of *numerous* additional hardware pieces to drive the parallel outputs. We had to use many different logic gates and hardware parts to implement our design. In a final design, we would definitely do more research to find the right design to reduce hardware and therefore the price.

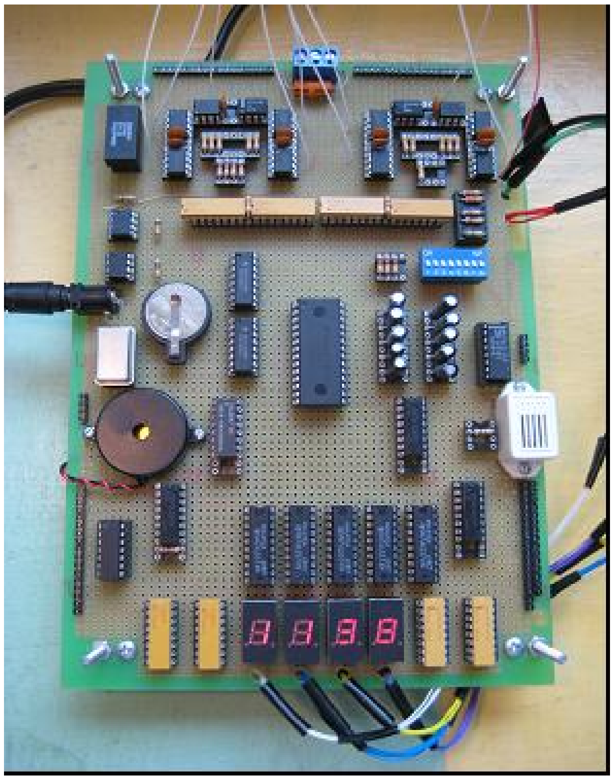


Photo 2 - Numerous miscellaneous hardware components were required, increasing the price.

Citing Apples ability to buy in bulk, the simplicity of many of the parts used, as well as Apple's ability to manufacture certain parts themselves, we settled on a 40% volume discount cost reduction, bringing our total cost down to $27.58. We then introduced a $15.00 expenditure for a quality audio amplification and speaker system, leading to a final cost of $42.58. With a selling price of $49.99, this provided a $7.41 profit and 14.8% profit margin, in line Apple's overall profit margin.

Although most of these prices are completely blind estimates, there are many similar and successful products on the market already such as the iHome alarm clock; therefore we are confident Apple would be successful in making this clock into a profitable and competitive product in the marketplace.

5. Hardware Documentation

5.1. System Design

5.1.1. Microcontroller

- Freescale MC908JL16 Microcontroller
- Oscillator (10 KHz)

5.1.2. Overview

The alarm clock is powered by a Freescale MC908JL16 microcontroller. It includes various built-in functionalities including RS232 and I^{2}C serial communication, as well as 22 general-purpose I/O pins. The system is clocked using a 10 KHz oscillator and driven using code written in C (Appendix B). Outside the microcontroller, the hardware is broken down into 4 major sections: Input, Output, Storage, and Audio. Each section is connected to the microcontroller and configured to either send or receive signals as shown in Figure 1.

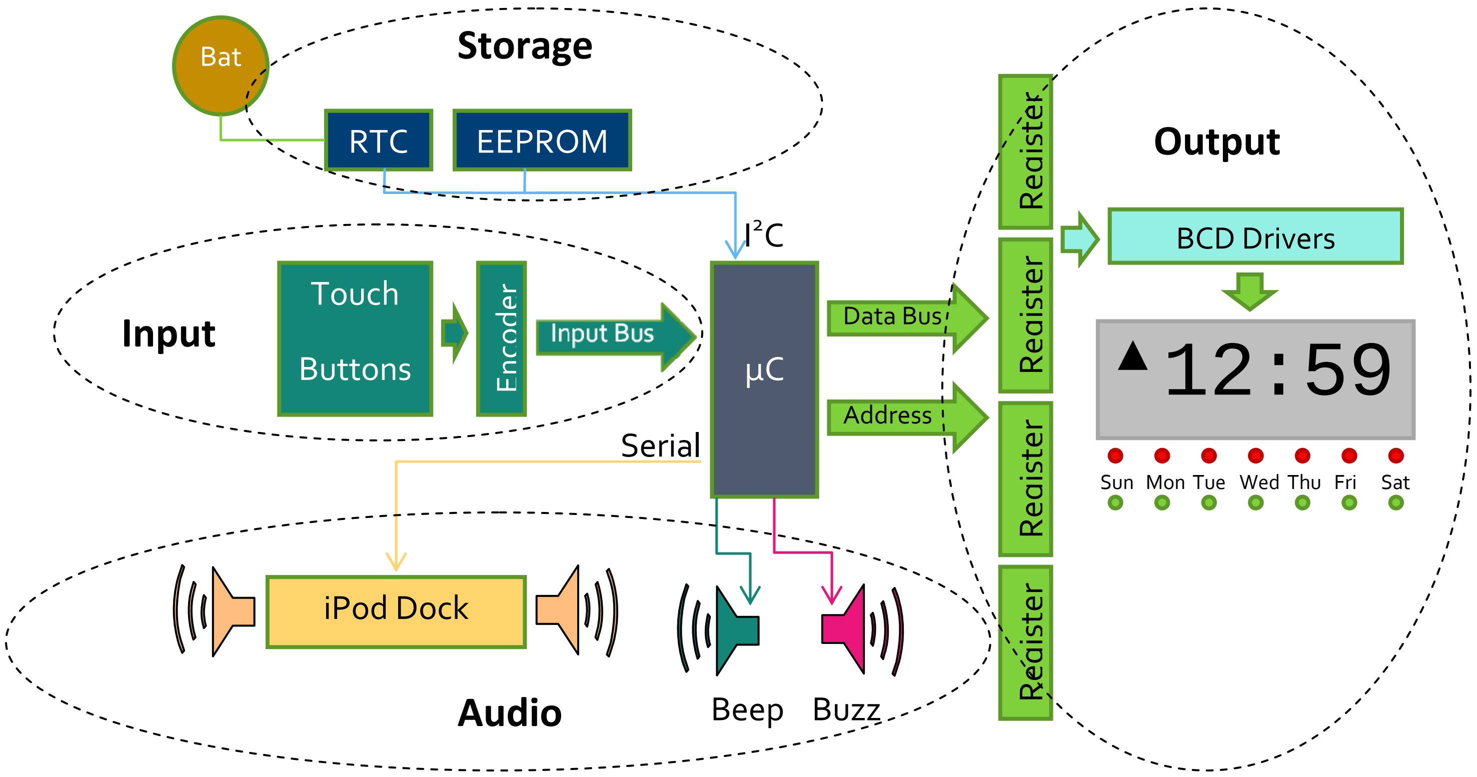


Figure 1 – Block diagram of the alarm clock system design.

The inputs allow the user to set the time and alarms as well as control the iPod. The outputs display the status of the alarm clock including the current time, day of the week and alarm status for each day of the week. The real time clock constantly provides seconds, minutes, hours, and day information to the microprocessor to ensure the accuracy of the clock. The EEPROM is the non-volatile storage where the alarm times are stored. Lastly, speakers and an iPod dock connector are implemented to allow the user to play music when the alarm goes off.

5.2. Input

5.2.1. Parts Used

- Touch-Sensitive Buttons (QT110)
- 8-to-3 Encoder (74LS148N)
- AND-gate Chip (74LS08N)
- Generic Toggle Switch

5.2.2. Overview

The inputs consist of ten touch-sensitive buttons and two toggle switches. Three of the touch buttons are designated as the general control buttons: set, increment and decrement. These 3 buttons can be used to either set the time and the alarms or control the iPod. When controlling the iPod, the increment button will act as the skip forward or the volume up button, the decrement button will act as the skip back or the volume down button and the set button will act as the play/pause button. The other 7 touch buttons are for enabling/disabling alarms and allow alarm times to be set for each day of the week. Lastly the two toggle switches are for selecting between buzzer/iPod mode, and debug/normal mode.

5.2.3. Touch Buttons

To implement the touch sensitive buttons, we chose Atmel's QTouch™ technology charge-transfer chips[2]. The chips work by projecting a sense fields through any dielectric surface such as glass, wood, or plastic which become sensitive to human touch. By continuously charging the sense electrode to a known potential, and then measuring the resulting charge, it is possible to detect a change in capacitance in the sense field. Because the human body has a capacitance, contacting the electrode with a finger introduces an increased capacitance and registers as a touch. The sensitivity of the touch chips can be configured by modifying the 100pF capacitor shown in Figure 2.

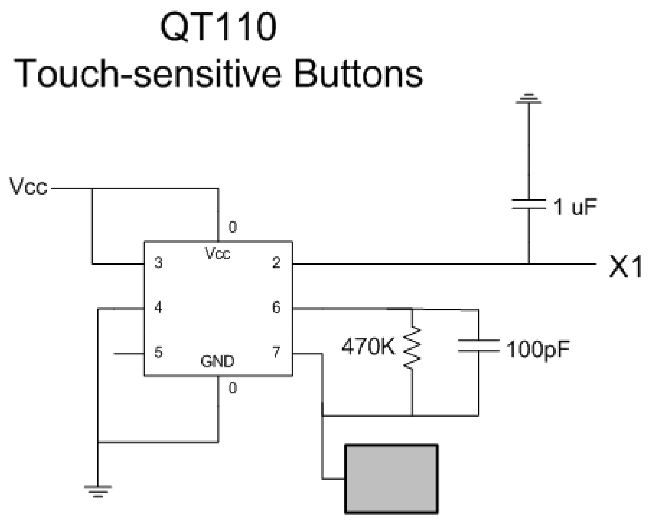


Figure 2 – Schematic of the QT110 touch-sensitive button chip.

All 10 outputs from the QT110's go into a 16-to-4 priority encoder, which was built from two cascaded 8-to-3 encoders and several AND gates (Appendix A). The 4 outputs of the

[2] http://www.qprox.com/technologies/qtouch.html

encoder are connected to pins PTA0, PTA1, PTA4 and PTA 5. By reading these 4 input pins, it is easily determined which of the 10 buttons is selected. Because an encoded button value is used, only one button at a time can be pressed; therefore the user interface of the clock was designed such that the user is never required to press more than one button at a time (Appendix E). By reordering the inputs of the encoder, the priority of the buttons can be determined; in practice, however, the priority of the buttons is immaterial. Input 0 of the encoder is left disconnected and acts as a default to distinguish when no button is selected.

It is important to note that the QT110 chip includes a “Heartbeat” feature that causes the signal to pulse for 350ns at every charge cycle. These Heartbeat pulses are unsynchronized and therefore cause the encoded output to be unstable. To circumvent this feature, a 1μF was added to damp out the voltage pulse, as shown in Figure 2.

5.2.4. Toggle Switches

Two simple on/off toggle switches were used to allow the user to switch between different modes. One of the switches allows the user to select between iPod mode and buzzer mode. In iPod mode, when the alarm goes off, the iPod will turn on and play music through the speakers. In buzzer mode, it will simply sound the buzzer when an alarm goes off. The other switch was added only to aid the engineers; it allows the clock to go into debug mode, where the minutes run much faster so it is more convenient for the engineer to test the alarm clock.

5.3. Output

5.3.1. Parts Used

- 8-bit Registers (74LS374N)
- 7-Segment LCD Display (VI-319-DP-FC-S)
- BCD to 7-segment decoder (74LS247)
- 3-to-8 Decoder (74LS138N)
- AND-gate Chip (74LS08N)
- Generic LEDs

5.3.2. Overview

There are two major parts to the outputs: the 7-segment LCD display and the LEDs. The outputs are driven from registers, which can be updated as needed by the software. 4 bits were required for each of the three 7-segment digit drivers, 1 bit for the tens digit in the hour, 1 bit for the colon, 1 bit for the AM/PM indicator, 14 bits for the LEDs, and 2 bits for the buzzer and beep speakers. In total this is 31 bits, requiring 4 8-bit registers to drive the output.

The first register holds the minute in two BCD encoded numbers. The next register holds the hour; however only one BCD encoded number is needed, as the tens digits of the hour will only be a “1” digit or else disabled and therefore only requires a single bit. Therefore, only 5 bits in total are needed for the hour, and the other three bits can be used for driving the colon and the AM/PM indicator. The next register holds the 7 bits needed to indicate the day of the week, along with the buzzer bit. The last register holds the 7 bits needed to indicate whether the alarm is enabled for each day, along with the beep bit. Together, these registers continuously drive the display output and can be updated as needed by the software.

5.3.3. Registers

The registers act like memory banks, sharing an 8-bit data bus and are enabled using a 3-bit address bus and simple chip select logic (Appendix A). When a register needs to be written, the 8-bit value is output on Pins PTB 0-7, which acts as the data bus. Next, a register is selected by outputting a 3 bit address onto pins PTD 0, PTD, 1, and PTD 2 which connect to a 3-to-8 decoder. The 3-to-8 decoder acts like the chip select in a memory system, by bringing the clock pin low on the given register. Because the registers are flip flops, and therefore edge triggered, the value is not actually latched in however, until the clock pin is brought back high creating an upward edge. This is done by outputting a 0 on the address bus in software, causing the decoder to output high signals to all chips.

5.3.4. LEDs

A total number of 14 LEDs were used: 7 green ones to indicate the day of the week and 7 red ones to show if the alarm is enabled for each day of the week. The LEDs are connected directly to the registers through a 330Ω resistor, and the given bit for each register simply needs to be written as a 0 to sink current through the LED and enable it.

5.3.5. 7-Segment LCD

The time was displayed using a Varitronix 7-Segment monochrome liquid crystal display[3]. Unlike a graphic LCD, this screen is limited to displaying a “1” digit and three 7-segment digits, a colon, an AM/PM indicator and a few other various images. The LCD has 40 pins, with each image being controlled by a specific pin. For the 7-segment digits, the pins were connected to BCD drivers that output the 7-segment bit combination needed to display the given digit. The BCD driver inputs were connected to the registers, allowing the software to write a number 0-9 to the register in order to display the given digit. The other LCD images (AM/PM, colon, and “1” digit) were connected directly to register pins and could be enabled by outputting a 0 to the corresponding bits.

[3] http://www.varitronix.us.com/?pid=2

5.4. Storage

5.4.1. Parts Used

- Real Time Clock (DS1307)
- EEPROM (24LC256)
- Oscillator (DS32KHZ)
- Battery (CR2032)

5.4.2. Overview

In order to ensure accurate time keeping and non-volatile storage of the alarms, a real time clock chip and EEPROM are used. Both chips communicate data to the microprocessor serially through an I^2C bus. The DS1307 real time clock chip uses an external oscillator and a 3.3v battery to continuously keep the time even when the wall power is disconnected.

5.4.3. I^2C

Both the RTC and the EEPROM communicate with the microcontroller via I^2C. A 10KΩ pull-up resistor is required on both the SCL and SDA line. The I^2C address for the RTC chip is 0xD0; the EEPROM chip address is 0xA0.

5.4.4. Real Time Clock

The DS1307 chip was used for this project because it includes a built-in power-sense circuit to detect power failures and automatically switch to a backup power supply when the main power is lost. The chip is specifically designed to utilize very little power and is easily accessible via the simple I^2C bus. The accuracy of the clock is determined by the accuracy of the attached 32.768 KHz oscillator. The RTC can keep time in both 12 hour and 24 hour format; for simplicity, the time is kept in 24 hour format and converted later via software to 12 hour format.

5.4.5. Backup Battery

A 3.3v backup battery is connected to both the RTC and the attached oscillator. When the RTC detects that the main power has dipped below the battery level (~3.3v) it automatically switches to the battery power. Considering the miniscule amps drawn, the battery is rated to last 10 years when connected to the RTC chip and oscillator.

5.4.6. EEPROM

A 256K EEPROM is used to store alarms in a non-volatile location as a backup when power is lost. When the clock first initializes, the alarms are retrieved from the EEPROM; any time an alarm is updated afterwards, it is immediately written back to the EEPROM. This ensures the user alarms always persist through a power failure and never need to be reentered. The EEPROM is accessible via I^2C and can withstand more than a million erase/write cycles.

5.5. Audio

5.5.1. Parts Used

- iPod Dock Connector
- Main Speakers/Audio Amp
- Buzzer
- Piezoelectric Speaker

5.5.2. Overview

There are two ways a user can choose to be awoken when an alarm is activated: a standard alarm buzzer or music from an Apple iPod. The user can choose by toggling a switch on the clock to either buzzer mode or iPod mode. If the user has selected iPod mode and connected an iPod via the dock connector, then music is played from the iPod through two main speakers when an alarm goes off. Otherwise, a standard alarm buzzer is sounded to wake up the user. Additionally, a piezoelectric speaker is used to create tactile feedback for the touch buttons, sounding a quick “beep” to indicate a touch was registered. Note that if the user selects iPod mode, but no iPod is connected, no sound will be emitted when an alarm is activated.

5.5.3. iPod Connector

The iPod is connected to the alarm clock via a 30-pin dock connector. This dock connector allowed full integration with the iPod including audio line outs and a serial communication link. Further details of the iPod integration can be found in Section 7.

5.5.4. Main Speakers/Audio Amp

Via the iPod dock, there is a stereo audio line out that allows us to play music from an iPod directly to speakers. Attached to the alarm clock casing were two 25Ω speakers envisioned for playing the iPod music. Unfortunately, the iPod was not powerful enough to drive these speakers and an audio amplification circuit was required. Due to a lack of analog circuit expertise and time, this circuit was not implemented. A final product design would require this portion of the project to be completed.

5.5.5. Buzzer

A simple alarm buzzer was included if the user preferred to be awoken to a standard pulsing buzz. The buzzer could be sounded by driving 5v across the terminals. It was connected directly to a register pin and could be enabled by outputting a 0 to the corresponding bit.

5.5.6. Piezoelectric Speaker

A piezoelectric speaker was used to provide tactile feedback to the user when a button press was registered. The speaker requires a frequency to create sound; this signal was provided as a feature from the DS1307 clock chip. The wave signal was sent through an AND gate and then connected to the speaker. The other input of the AND gate was connected directly to a register pin. By outputting a 1, the digital wave signal would be passed through the AND gate and emitted by the speaker.

6. Software Documentation

6.1. Overview

The alarm clock was modeled into a simple Moore state machine that is implemented in software and shown in Figure 3. The software controlling the embedded system is written in C for the Freescale MC908JL16 microcontroller. The chip carries a limited 16Kb of flash memory restricting the size of allowable code. Furthermore, only 512 bytes of ram are available for both the stack and heap making it essential to write efficient and resourceful code. In total the code is less than 1200 lines.

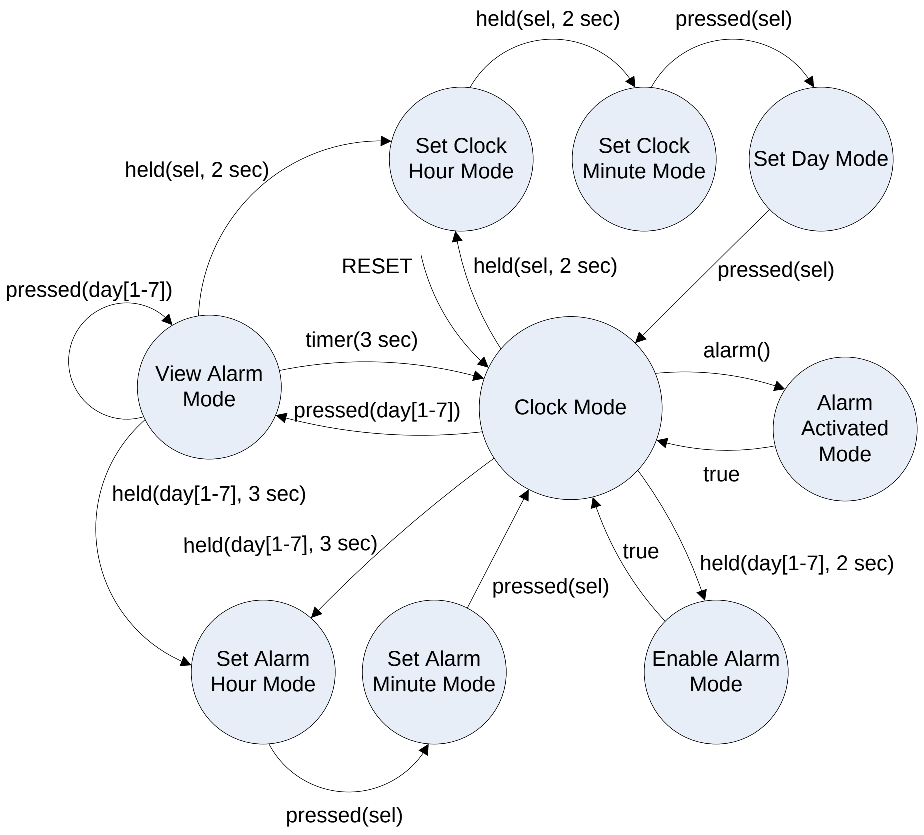


Figure 3 – Alarm clock state machine.

6.2. Initialization

The initialization routine is responsible for setting up the various microcontroller functional units and assigning data directions to the various input/output pins. The Serial Communications Interface is enabled on pins 13 and 14 at a baud rate of 19,200 and the I^2C Interface is enabled on pins 8 and 9. Pins 0, 1, 4, and 5 on Port A are configured for input from the touch button decoder. Pins 0-7 on Port B are configured for output and used as the data bus. Pins 0-2 on Port D are configured for output and used as the address bus. Pins 3-4 on Port D are configured as input from the toggle switches.

To determine if a battery backup was in use while the power was off the code checks for a checksum previously stored in an unused register in the real time clock chip. If there was a battery backup, this checksum would have persisted; else it is assumed there was no battery backup and the `volatile` variable is set and the clock blinks until the time is set by the user. The blinking is to indicate to the user that power was lost and that the time may not be correct.

Next hardware initialization begins: the RTC chip is initialized with the necessary configuration bits; the saved alarms and current time is loaded from the EEPROM and RTC over I^2C; and the iPod is forced into charge mode via serial. Following this, the state machine is initialized by setting the initial state to *clock* mode, and setting the internal time and alarm variables. Lastly the programmable interrupt timer is configured to expire every 125,000 clock cycles (~5 milliseconds).

6.3. Time Keeping

In order to ensure accurate time keeping, the software polls the RTC chip every loop cycle, retrieving the current time and storing it in an array called `time[]`. This polling is done via I^2C using the function `time_read()`. Any time the clock time is set by the user, the time is immediately written back to the RTC via I^2C using the function `time_write()`. Because the RTC is always kept up-to-date and because it is backed up by the battery, an accurate time can always be gained by polling the RTC.

If the clock is in a state that does not require displaying the current time then the RTC is not polled and the `time[]` array is simply injected with the time that should be displayed. For example, if the clock is in *set alarm minute* mode, then the `time[]` array is loaded with the alarm time being set because that is the time that should be displayed to the user.

6.4. Alarm Storage

The seven alarms are stored in an array called `alarms`[]. Every time the alarms are updated, this array is immediately written to the EEPROM via I^2C using a function called `alarm_write()`. In this way, the alarms are always saved and can be reloaded the next time the clock initializes in case of power failure.

6.5. Programmable Interrupt Timer

A programmable interrupt timer is set to expire approximately every 5 milliseconds. This single timer is responsible for a number of things, which are individually timed using pre-scalars. Duties include synchronizing the button state machines, inverting the `flash` variable used to determine the blink rate of flashing digits, and emitting the beep sound when a button is pressed.

6.6. State Machine

The main state machine is contained in an infinite loop that continuously runs after initialization. While the input and output state variables are determined in hardware, it is both time-consuming and complex to access the hardware every time a state variable changes. For simplicity, a duplicate working copy of the state variables are kept in software and updated to/from hardware once per loop cycle. This minimizes the amount of hardware interaction and negates the need for complex synchronization between the hardware and software. In order to accomplish this separation, the main state machine cycle is broken into three parts: updating state variables, checking for state transitions, and flushing the output states back to the hardware

6.6.1. Updating State Variables

6.6.1.1. Scanning Inputs

In order to ensure a responsive alarm clock, it is necessary to update the input state variables as often as possible. This is done by scanning the touch button inputs once per loop cycle and determining which of the ten buttons the user is currently touching (if any). Because the QT110 touch button chips have built in auto-debouncing, we simply need to read the encoded value of our input to determine what button is pressed at any given time.

Knowing the status of each button is insufficient, however, in determining state transitions. Because the alarm clock user input involves holding buttons for a given period of time, it is also necessary to model a state machine for each button. The button state machine is shown in Figure 4.

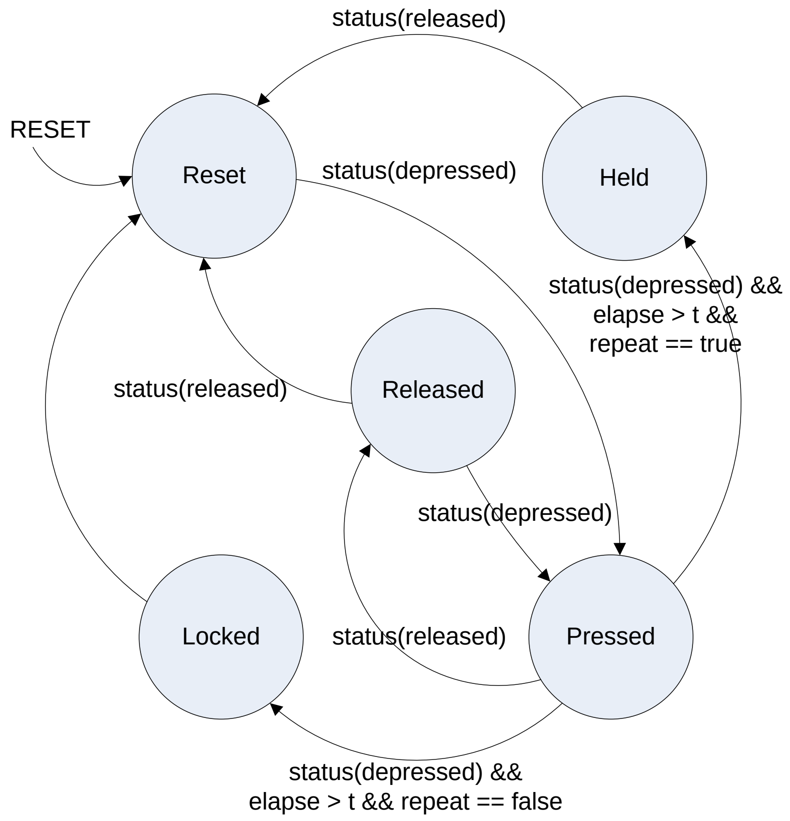


Figure 4 – Button state machine.

It is necessary to update the state machine at a known time interval in order to accurately track how long a button has been held or released. To achieve this, the programmable interrupt timer is used to call the function `scan()`, every 5 milliseconds. In this function, the current state of each button is updated depending on the buttons current status (depressed/released) and how much time has elapsed since the last status change.

The button states are used to determine the transitions in the main alarm clock state machine. For example, if the user has held the *select* button for 1 second, then the clock enters *set time* mode. The *locked* button state is necessary because in some cases, we might not want to the button to repeatedly fire if held. For example, if the *Monday alarm* button is held for 1 second we enter *set alarm hour* mode. At this point, if the user continues holding the *select* button, the alarm clock should not transition to *set alarm minute* mode. Instead, nothing should occur until the user first releases the button and then presses it again.

6.6.1.2. Control Variable

The `control` variable is used to set various clock states as shown in Figure 5. This variable is later used by the `flush()` function to determine how the output should be displayed.

Control Variable							
Bit 7	6	5	4	3	2	1	0
╳	╳	Buzzer On	Flash Alarm Day	Flash Day	Flash Hour	Flash Minute	Flash Second

Figure 5 – Control variable

At the start of each state, the `control` variable is set using bit masks. For example, when the clock is in the *alarm activated* mode, bit 5, bit 2 and bit 1 are enabled to indicate that the buzzer should be sounding, and the hour and minute should be flashing.

6.6.2. State Transitions

In order to determine state transitions, two functions are used as transition guards:

```
char released(char n, signed char t, char b);
char held(char n, signed char t, signed char r, char b);
```

The `released()` function returns true if button *n* is in the *released* state and was previous held for at least *t*5 milliseconds*. If argument *b* is set then it will produce a short beeping sound to indicate the button press registered.

The `held()` function returns true if button *n* is in the *pressed* state and at least *t*5 milliseconds* has elapsed or the button is in the *held* state and at least *r*5 milliseconds* has elapsed. Again, argument *b* determines if the tactile beep is sounded. The reason both a *t* time and an *r* time are needed is best explained by comparing the function with how a keyboard works; the amount of time it takes before a key registers the first time is often different between the elapsed time between button fires when you hold the key down.

By utilizing these two functions, it is easy to check if state transitions guards have been met. For example, when the user holds and releases the *Wednesday alarm* button for 1 second, the alarm should enter *set alarm hour* mode. To check for this, we use the following function call:

```
held(WED, NO_START, 20, BEEP);
```

`WED` is defined as the Wednesday button. `NO_START` means that the button should not register as held until it has already registered true at least once already. `20` indicates the amount of time (in units of 5 milliseconds) that should elapse until it registers true again. `BEEP` means a beep should be emitted to indicate the button was pressed.

If this function were to return true, then we know that the state transition guard has been satisfied and it's a simple matter of updating the alarm clock state variable, `mode[CLOCK_MODE]` to the next state. This is also the appropriate time for running any one-time processes, such as starting the music if we are entering the *alarm acitivated* mode or loading the `time[]` array with the correct alarm time if we entering *set alarm hour* mode.

6.6.3. Flushing Output

The last step in a state machine cycle is flushing the internal state variables held in software, to the external hardware. This is done using the function `flush()`. This function updates the output registers, using the variables `data` and `addr`, which have been mapped to the corresponding data bus and address bus pins. Before doing this, the code uses the `control` variable and simple logic to format the output. For example, if the hour digits are blinking, then the function should alternate between writing the hour and writing a blank digit. Likewise, the output must be converted from the internal decimal format, to the BCD encoded format required by the BCD drivers.

In total, the `flush()` function is responsible for loading the registers with the `time[]` to be displayed, the current day of the week, the enable status for each `alarm[]`, the AM/PM bit, the colon bit, the "1" digit bit, the buzzer bit and the beep bit. After this function completes, the displayed output is up-to-date and can be seen by the user. Note that the `time[]` array might not contain the current time; it simply holds the time that should be displayed to the user according to the mode the alarm clock is in.

6.7. Helper Functions

The code includes numerous helper functions which are reused throughout the code to accomplish various tasks. These functions range from the very simple, such as converting an 8-bit number from decimal to the equivalent BCD encoding, to the more complex such as writing an `n` byte `buffer[]` array to a chip over I^2C. These functions are mostly self explanatory, however, one function should be noted: `sn()`.

The `sn()` function is used to swap the upper and lower nibbles of an 8 bit number before outputting it to the register. This is done because we accidentally wired the most significant register bits to the least significant digit on the LCD and vice-versa for both the hour and minute digits.

7. iPod Integration

7.1. Overview

There were three goals for the iPod integration: charging, song control, and audio output. Apple provides a dock port at the bottom of most iPods and a protocol for controlling the iPod, which combined provide all the features desired. This dock and protocol are proprietary to Apple, but have been reverse engineered by enthusiasts. Through an internet search, we discovered a description of the 30 pin port[4] along with a description of the accessory protocol[5]. An iPod dock connector was purchased via an iPod hobbyist in Sweden and was hand soldered for the project[6].

7.2. Dock Connector

The port offers many different communication options including USB, FireWire and RS232, as well as access to the audio and video line outs of the iPod. The pins listed in Table 1 were identified as useful for the purposes of the project. Specifically, the project uses the USB pins for charging the iPod, the audio out pins for playing music, and the serial RxD pin for controlling the iPod over RS232. USB was chosen for charging since it conveniently runs off of 5v allowing us to charge directly from the main power rails. Likewise, serial communication was chosen for controlling the iPod since the microcontroller has built in RS232 and the serial protocol was the only documented accessory protocol on the internet.

Table 1 - Summary of dock connector pins used.

Pin	Signal	Note
2	Audio GND	
3	Audio (R)	Audio line out, right stereo channel
4	Audio (L)	Audio line out, left stereo channel
11	Serial GND	
12	iPod Serial TxD	For receiving *from* the iPod via serial
13	iPod Serial RxD	For transmitting *to* the iPod via serial
16	USB GND	
21	Accessory Indicator	A 500kΩ resistor between this pin and ground enables serial communications
23	USB Power	
25	USB Data (-)	A 10KΩ resistor between this pin and 5v forces iPod into charging mode
27	USB Data (+)	

7.3. iPod Interface Circuit

The iPod interface circuit (Figure 6) implemented in the iSnooze alarm clock was designed, configured and tested with a 2nd Generation iPod Nano. The specific circuit requirements are unique to each iPod generation and very poorly documented. An incorrectly designed circuit runs the risk of damaging the iPod. *Proceed with caution…*

One such consideration is that the iPod Nano runs on 3.3v, while the microcontroller on 5v. Therefore it is necessary step down the serial TxD voltage driven from the microcontroller

[4] http://pinouts.ru/Devices/ipod_pinout.shtml

[5] http://ipodlinux.org/Apple_Accessory_Protocol

[6] http://home.swipnet.se/ridax/connector.htm

using a simple voltage divider. It’s also noted that it’s possible to force most iPod’s into charging mode by connecting pin 25 to ground through a 10KΩ resistor and pin 27 to 5v through a 10KΩ resistor. However, the iPod Nano requires pin 27 to be left open, a distinction not noted in most documentations.

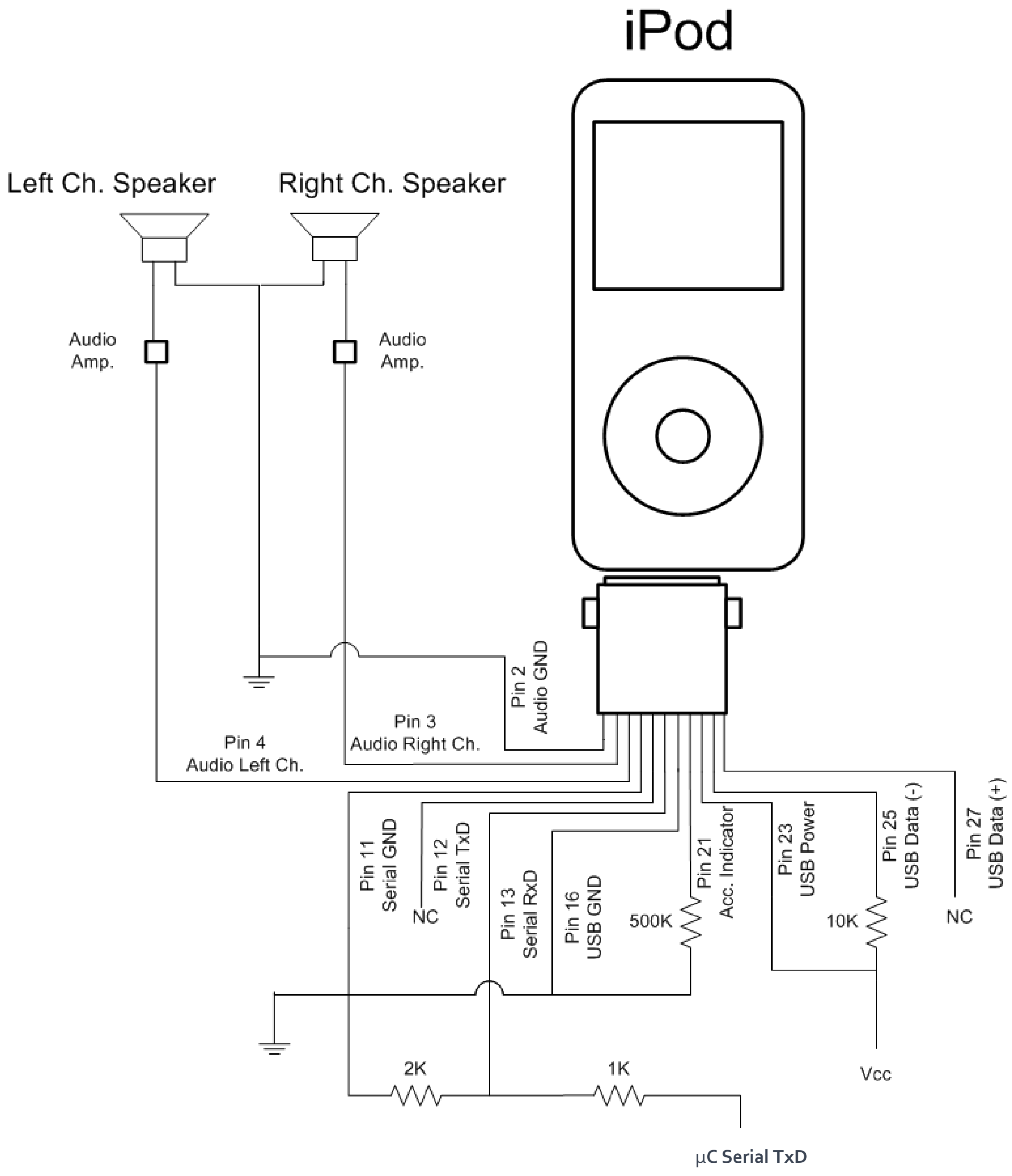


Figure 6 – Schematic of the iPod dock connector circuit.

The dock connector also provides a left and right audio channels line out on pins 3 and 4. By theses pins to an oscilloscope, it was possible to see the audio signal being driven by the iPod. The iPod is not powerful enough to drive most speakers, and therefore an audio amplification circuit needs to be built for these lines out pins to be utilized. Careful consideration should be given to ensure the iPod is not inadvertently overloaded. Due to time constraints, this part of the project was not completed.

Lastly, we note that it is necessary to connect a 500kΩ resistor from pin 21 to ground to force the iPod into serial communication mode for use of the Apple Accessory Protocol.

7.4. Soldering

The most difficult part of the iPod integration was soldering. It required a steady hand and multiple tries. The pins were very fragile and often came unsoldered as seen in Photo 3.

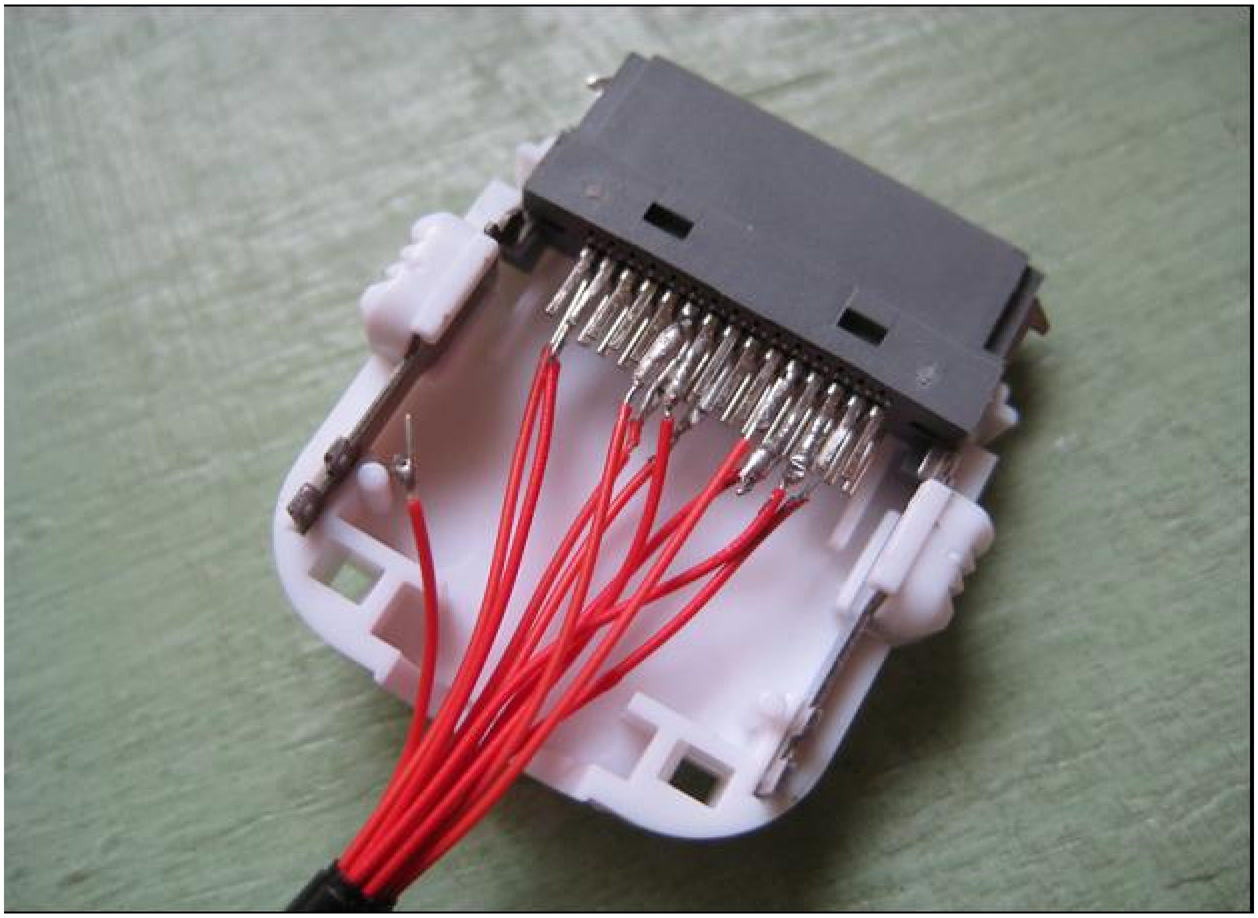


Photo 3 - Soldering the iPod dock connector was very tedious.

7.5. Apple Accessory Protocol

Controlling the iPod is done using a protocol known as the Apple Accessory Protocol. This is an asynchronous protocol operated over RS232. It is proprietary to Apple and only released to Apple accessory partner vendors. It was necessary, therefore, to use documentation obtained via reverse engineering. Such documentation[7] is available on the internet, abet very incomplete and poorly written.

[7] http://ipodlinux.org/Apple_Accessory_Protocol

Many different features are provided by the protocol including normal song control, photo transferring, Nike+ communication and more. For the purposes of this project, only the basic song control features were needed.

7.5.1. Overview

There are various modes that can be used to control different features of the iPod. Mode 2 is the basic iPod remote control mode which allows for simple song control features such as pause/play, skip/back, volume, etc. If the iPod is properly connected via the dock connector, including the correct accessory indicator resistor, then communicating with the iPod is a simple matter of sending asynchronous command packets to the iPod. When the iPod recognizes a proper command, it will respond immediately. Malformed packets appear to be ignored.

7.5.2. Baud Rate

The protocol can operate at various RS232 baud rates, but is appears to be most reliable at 19,200, which is the speed used for this project.

7.5.3. Packets

Packets are sent to the iPod asynchronously using a defined structure as shown in Table 2. The header is the constant value 0xFF 0x55. The length is equal to the number of bytes in the mode, command and parameter portion of the packet. The mode is the mode number corresponding to the command. The command value is the byte combination corresponding to the command. The command parameter is an optional parameter associated with the command. The checksum is a value calculated with the following equation: 0x100 - ([length + mode + command + parameter bytes] & 0xFF).

Table 2 – Apple Accessory Protocol packet structure.

Field	Size (in bytes)	Value
Header	2	0xFF 0x55
Length	1	Size of mode + command + parameter
Mode	1	Command mode
Command	2	Command value
Parameter	n >= 0	Optional command parameter
Checksum	1	0x100 - ([length + mode + command + parameter bytes] & 0xFF)

7.5.4. Command Sequence

The protocol is designed such that commands sent emulate button presses. For example, if you want to skip to the next song, you would send the “Skip Forward” command followed by the “Release Button” command to simulate a user pressing and releasing the skip button. For most commands, the iPod will not respond until the “Release Button” command is sent. Because the documentation is reverse engineered, it took some creative debugging to get the iPod to always respond as intended. Often it didn't work as documented, and again, different iPod generations may respond differently to the same command sequence.

7.5.5. Commands

The commands shown in Table 3 were identified as useful for the project, all of which are mode 2 commands.

Table 3 –Useful Apple Accessory Protocol mode 2 commands.

Value	Parameter	Command
0x00 0x00		Button Release
0x00 0x00	0x01	Play (no pause)
0x00 0x01		Pause
0x00 0x80		Stop
0x00 0x08		Skip Forward
0x00 0x10		Skip Back
0x00 0x02		Increase Volume
0x00 0x04		Decrease Volume

7.5.6. Example Code

The following example code shows how the programmer would increase the volume on the iPod by sending bytes via serial.

```
void ipod_increase_volume(void) {
  /* Volume up */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x02);
  sci_write(0xF9);              // Checksum

  /* Button release */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x00);
  sci_write(0xFB);              // Checksum
}
```

8. Debugging

One of the key components of any successful project is debugging. We decided to debug the project in phases to help alleviate the complexity of final testing in the end. After each feature was completed, we would test for both software and hardware errors along with user interface issues. If any were found, we would fix them and re-test before moving on.

8.1. Wiring

The single greatest challenge was ensuring the wiring was accurate and secure. Due to the large amount of hardware components, the final project was completed with a *mess* of wiring (Photo 4), making it very difficult to debug hardware issues near the end. The large amount of wiring also introduced other issues with the touch buttons as described below.

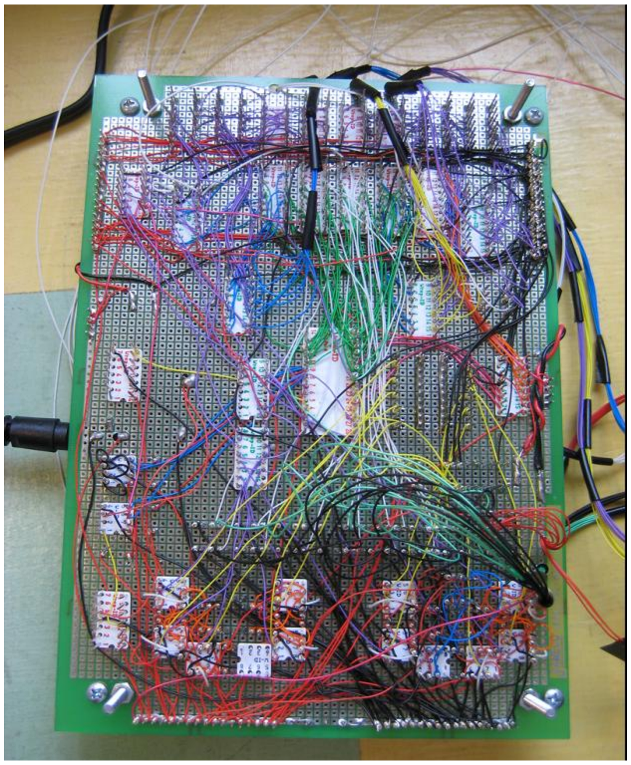


Photo 4 - Lots of wiring causes major headaches!

8.2. Touch Button Issues

One of the major problems that we ran into was that the touch sensitive chips were extremely sensitive to small changes in capacitance and often would detect phantom touches. Because parallel wires act as a capacitor, the large number of wires on the board created fluctuating capacitance when brushed against each other. This would cause the buttons chips to go haywire and induced problems that we often spent hours debugging in software when they were instead unpredictable hardware issues.

8.3. iPod Integration

The iPod integration was especially difficult to debug, simply because of the lack of documentation. The protocol is entirely reverse engineered and many websites offered conflicting or lacking information. One example of a problem encountered was in getting the iPod to enter forced charging mode. The main documentation we were using said that pin 27 should be tied to ground through a 10KΩ resistor—it wasn't until much later that we found another document that claimed pin 27 should be specifically left unconnected if you are using a 2nd Generation iPod Nano.

8.4. Software Bugs

There were many small software bugs although some were actual issues, while others just constituted what we thought were deficiencies in the user-interface. For example, at one point only the odd numbered buttons were being recognized. This turned out to be because we forgot to increase the size of an array and our pointer arithmetic was incorrect—a true bug. Conversely, we made an improvement that made the button debouncing code more reliable and the touch buttons more responsive. Because the original code was working as expected (just not well), this fix was more an improvement in the user interface and experience than the squashing of a bug.

8.5. Hardware Failures

We encountered multiple hardware failures that were both difficult to debug and excruciating frustrating. The first iPod connector we received had a bridged pin, leading to hours of confusion. Also, late in the project cycle our main microcontroller oscillator became temporal—it would occasionally stop operating if physically bumped. This took many hours to discover and was an odd problem to encounter.

9. Casing

We decided to make a casing for our alarm clock to abstract the hardware circuitry from the user interface. Our housing features three main large touch buttons: set, increment and decrement; as well as seven smaller touch buttons, one for each day of the week. Additionally, there were two LEDs for every day of the week; green told the user which day of the week it was and red indicated whether the alarm for that day was enabled or disabled. The LCD screen was mounted in between the main buttons and the day buttons; a triangle in upper left part of the LCD indicated AM/PM.

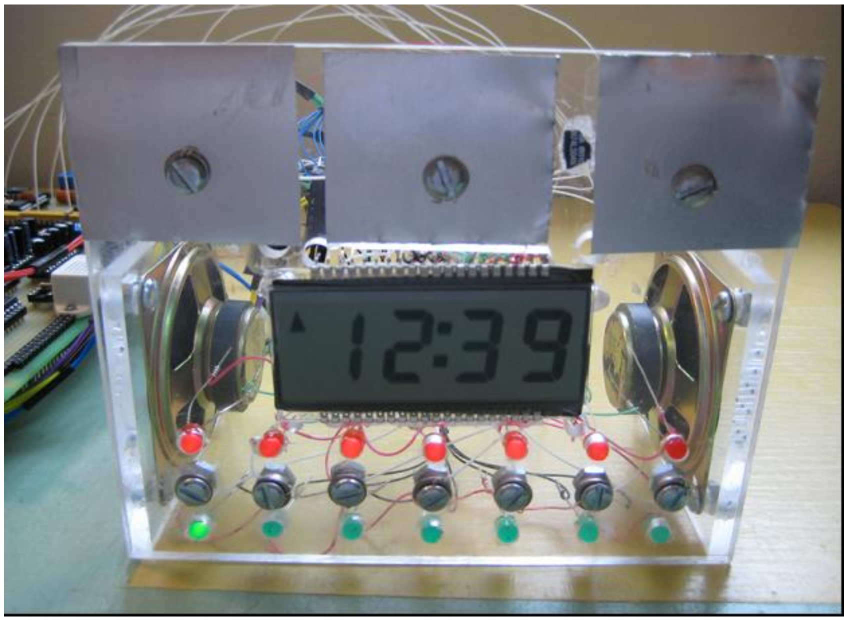


Photo 5 - Alarm clock casing.

Lastly, we attached two speakers, one for the left audio channel and one for the right audio channel. However, due to a lack of time we were unable to figure out how to correctly design an audio amplification circuit and did not actually integrate the attached speakers. Therefore, it

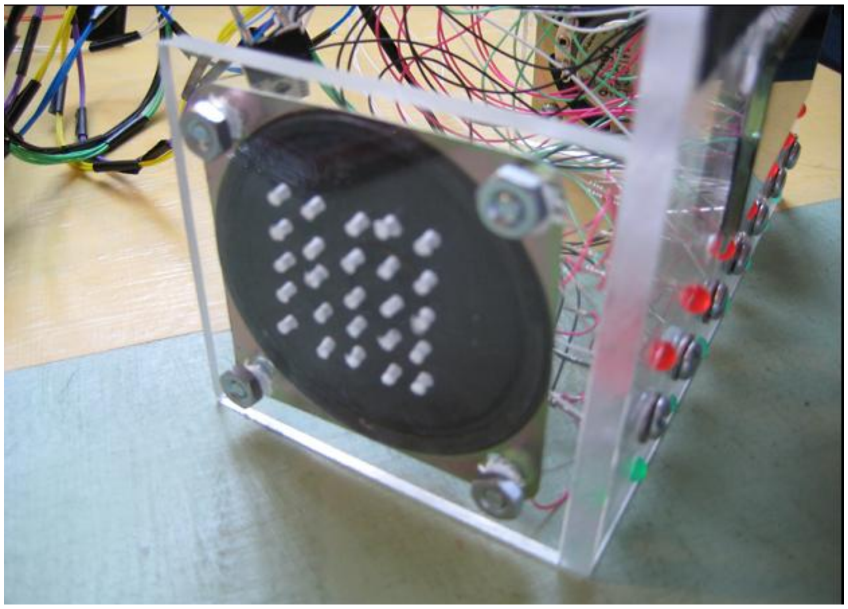


Photo 6 - Integrated speakers.

is necessary to attach external speakers to the iPod when demoing the project. In a final design, however, the Apple iSnooze would have a quality amplifier and stereo speakers integrated.

10. Conclusion

This course was enjoyable because we used our creativity to design and build a project from real components, encountering and solving real problems on the way. It is impossible to gain the practical and wide-ranging engineering experience this course offered through basic classroom lectures. In addition, our group gained excellent experience working as a team, meeting deadlines, and making compromises; all skills that are useful both in engineering and beyond. Coordinating with the marketing students was a bit awkward at first; however, their input and research eventually helped to streamline our design and create a superior final product. Overall we believe the project was a smashing success and someday we hope the Apple iSnooze will help people wake easier—one song at a time.

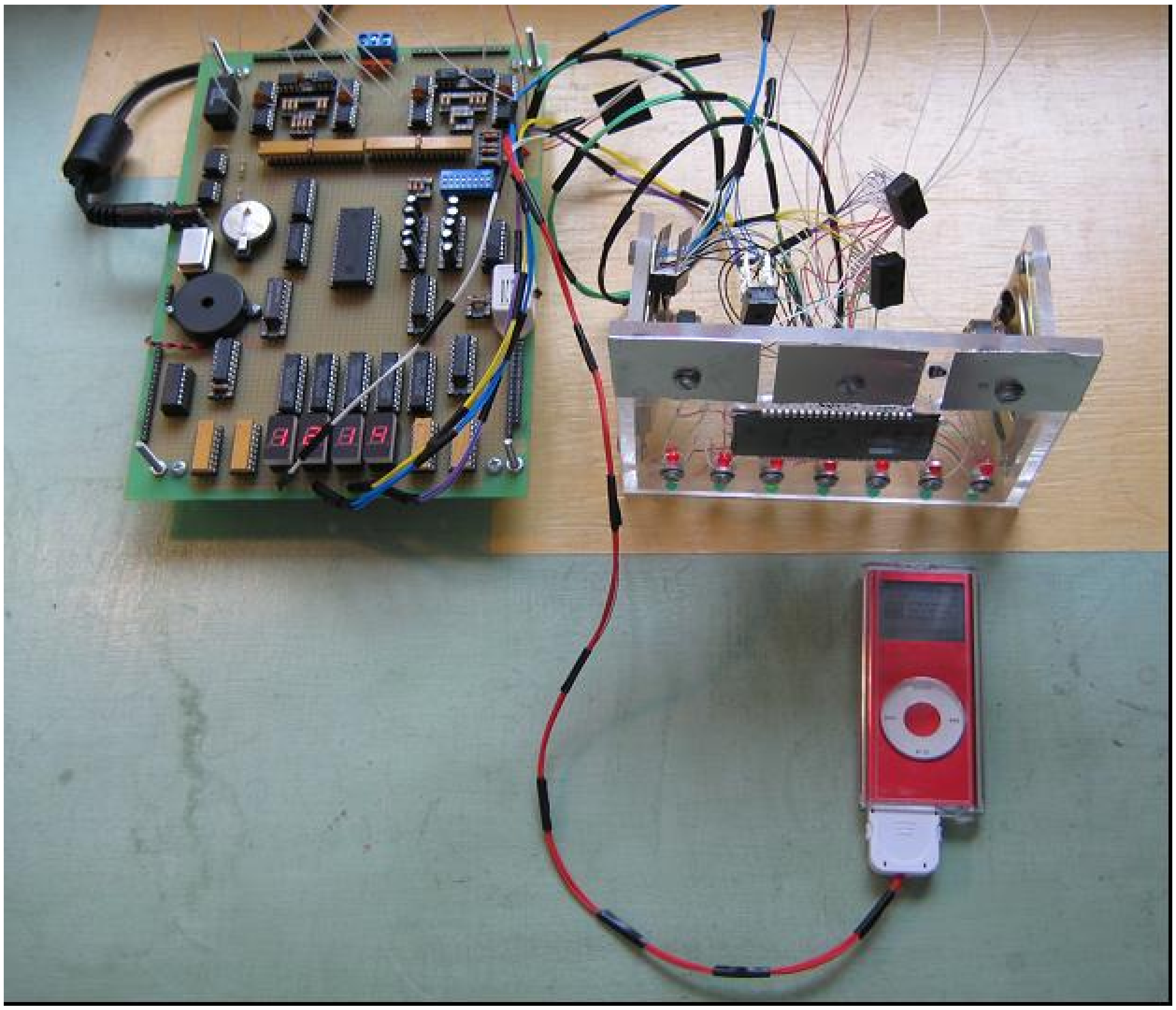
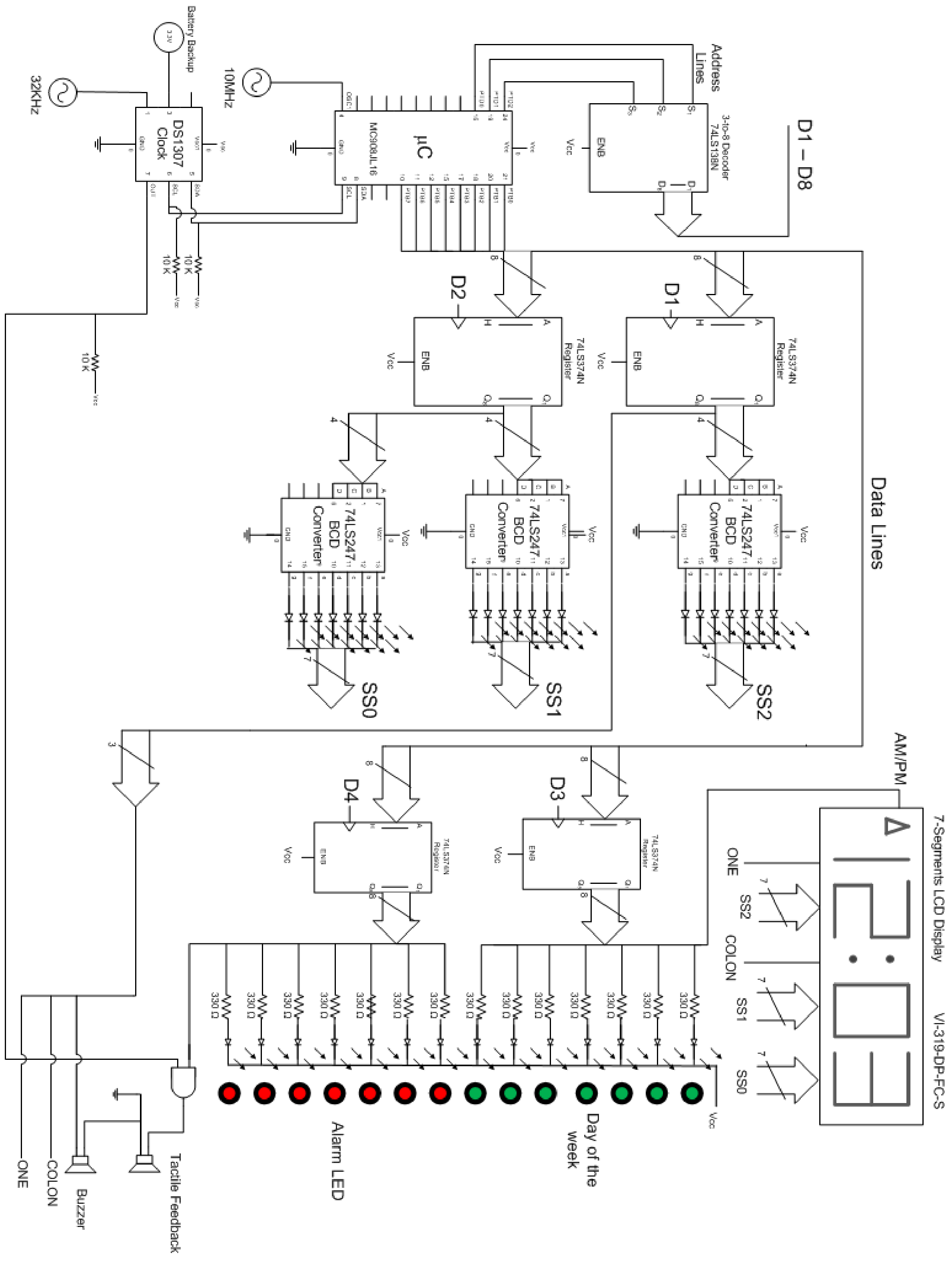
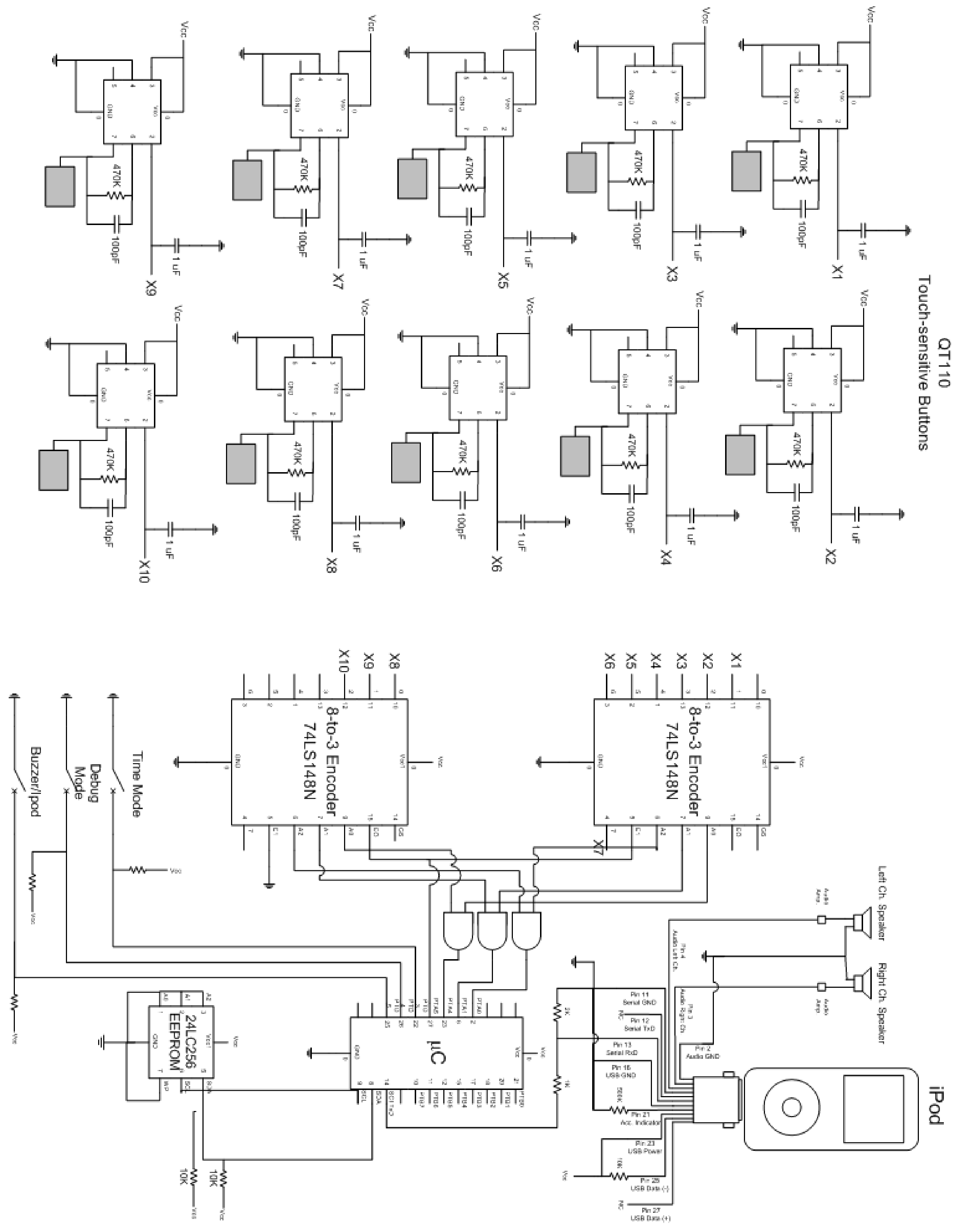


Photo 7 - The Apple iSnooze was a smashing success!

Appendix A – Hardware Schematic





QT110
Touch-sensitive Buttons
X1
X2
X3
X4
X5
X6
X7
X8
X9
X10
Vcc
470K
100pF
1 uF
8-to-3 Encoder
74LS148N
µC
24LC256
EEPROM
10K
Time Mode
Debug Mode
Buzzer/Ipod
iPod
Left Ch. Speaker
Right Ch. Speaker
Audio Amp
Pin 2 Audio GND
Pin 3 Audio Right Ch.
Pin 4 Audio Left Ch.
Pin 11 Serial GND
Pin 12 Serial TxD
Pin 13 Serial RxD
Pin 16 USB GND
Pin 21 Acc. Indicator
Pin 23 USB Power
Pin 25 USB Data (-)
Pin 27 USB Data (+)

Appendix B – Software Code (main.c)

```
/**************************************************************
*
*   EE459 Alarm Clock Project
*   Team 1A, Spring 2008
*
*   Author: Nate Houk
*   Description: This code implements a 7-day alarm clock
*   on a MC908JL16 microcontroller utilizing the Apple
*   Accessory Protocol to control an iPod via RS232.
*
*   Copyright 2008, all rights reserved.
*
*************************************************************/

#include <hidef.h>      /* For EnableInterrupts macro */
#include "derivative.h" /* Include peripheral declarations */

/* The following puts the dummy interrupt service routine at
   location MY_ISR_ROM which is defined in the PRM file as
   the start of the FLASH ROM */
#pragma CODE_SEG MY_ISR_ROM
#pragma TRAP_PROC
void dummyISR(void) {}

/* This pragma sets the code storage back to default area of ROM as defined
in
   the PRM file.*/
#pragma CODE_SEG DEFAULT

/* Implements the ASSERT macro */
#define assert(x)      if(!(x)) {\
                          DisableInterrupts;\
                          for(;;) {}\
                       }

/* Defines the variable input as PTA bits 0-1, 4-5 */
volatile struct {
    char a:1;
    char b:1;
    char :2;
    char c:1;
    char d:1;
    char :2;
} MyPTA @0x0000;
#define input ((MyPTA.d << 3) | (!MyPTA.c << 2) | (!MyPTA.b << 1) | !MyPTA.a)

/* Defines the variable data as PTB bits 0-7 */
volatile struct {
    char data:8;
} MyPTB @0x0001;
#define data MyPTB.data

/* Defines the variable addr as PTD bits 0-2, the variable mode_switch as PTD
bit 3,
   the variable debug_switch as PTD bit 4, the variable buzzer_switch as PTD
bit 5 */
volatile struct {
    char addr:3;
    char mode_switch:1;
    char debug_switch:1;
    char alarm_switch:1;
    char :2;
} MyPTD @0x0003;
#define addr MyPTD.addr
#define mode_switch (!MyPTD.mode_switch)
#define debug_switch (!MyPTD.debug_switch)
#define alarm_switch (!MyPTD.alarm_switch)

/* Defines the device addresses on the I2C bus */
#define CLOCK_ADDR      0xD0    // DS1307 Clock
#define EEPROM_ADDR     0xA0    // 24LC256 EEPROM

/* Defines the device addresses on the data bus */
#define SEND            0x00
#define HOUR_ADDR       0x01
#define MIN_ADDR        0x02
#define OUTPUT0_ADDR    0x03
#define OUTPUT1_ADDR    0x04
#define OUTPUT2_ADDR    0x05

/* Defines the internal clock addresses */
#define CLOCK_SEC_ADDR  0x00
#define CLOCK_CTR_ADDR  0x07
#define CLOCK_MODE_ADDR 0x06

/* Defines the internal clock control bits */
#define CLOCK_CTR_BITS  0x11
#define CLOCK_SET       0xD9
#define CLOCK_NOT_SET   0x00

/* Defines the internal eeprom addresses */
#define EEPROM_MSB_ADDR 0x00
#define EEPROM_ALM_ADDR 0x00

/* Defines common bit masks */
#define BIT0_MASK       1
#define BIT1_MASK       2
#define BIT2_MASK       4
#define BIT3_MASK       8
#define BIT4_MASK       16
#define BIT5_MASK       32
#define BIT6_MASK       64
#define BIT7_MASK       128
#define CH_MASK         128
#define TIME_MODE_MASK  64
#define SEC_MASK        0x7F
#define MIN_MASK        0x7F
#define HOUR_MASK       0x3F
#define DAY_MASK        0x07
#define ALARM_MASK      1
#define BEEP_MASK       1
#define AM_PM_MASK      1
#define COLON_MASK      2
#define ONE_DIGIT_MASK  4
#define DAYS_MASK       0xFE
#define FLASH_MASK      0x1F

/* Global data and constants*/
```

```
#define HOUR            0
#define MIN             1
#define SEC             2
#define DAY             3
#define ALM_ENABLE      2
#define BLANK           0xFF
#define TIME_SIZE       4
#define ALARM_COUNT     8
#define ALARM_SIZE      3
#define INPUT_COUNT     10
#define INPUT_SIZE      5
#define BUFFER_SIZE     128
#define OUTPUT_SIZE     3

char debug;
signed char time[TIME_SIZE], debug_time[TIME_SIZE];
char alarms[ALARM_COUNT][ALARM_SIZE];
char output[OUTPUT_SIZE];
char buttons[12][7];
char buffer[BUFFER_SIZE];

/* State variables and constants */
#define CLOCK_MODE      0
#define CLOCK           0
#define SET_CLOCK_HOUR  1
#define SET_CLOCK_MIN   2
#define SET_DAY         3
#define SET_ALARM_HOUR  4
#define SET_ALARM_MIN   5
#define VIEW_ALARM      6
#define ENABLE_ALARM    7
#define ACTIVATE_ALARM  8
#define TIME_MODE       1
#define NORMAL          0
#define MILITARY        1
#define ALARM_MODE      2
#define BUZZER          TRUE
#define IPOD            FALSE
#define NONE            0
#define FLASH_SEC       1
#define FLASH_MIN       2
#define FLASH_HOUR      4
#define FLASH_DAY       8
#define FLASH_ALARM_DAY 16
#define ALARM_ON        32
#define FLASH_INTERVAL  5       // Flash interval in 1/20's of a second
#define BUZZ_INTERVAL   5       // Buzz interval in 1/20's of a second
#define VIEW_TIME       80      // View time in 1/20's of a second
#define BEEP_TIME       2       // Beep time in 1/20's of a second
#define OFF_TIME        80      // iPod off button hold time in 1/20's of a
second
#define DEBUG_SPEED     6       // 1/( 20 * Debug_Speed) = length of a second
in debug mode
#define BEEP            TRUE
#define NO_BEEP         FALSE
#define SNOOZE          0
#define SNOOZE_TIME     10      // Snooze interval in minutes
#define SUN             1
#define MON             2
#define TUE             3
```

```
#define WED             4
#define THU             5
#define FRI             6
#define SAT             7
#define DOWN            8
#define SEL             9
#define UP              10
#define STATUS          0
#define RELEASED        0
#define PRESSED         1
#define TOTAL_ELAPSE    1
#define KEYFIRE_ELAPSE  2
#define PREV_ELAPSE     3
#define STATE           4
#define STATE_RESET     0
#define STATE_PRESSED   1
#define STATE_HELD      2
#define STATE_RELEASED  3
#define STATE_LOCKED    4
#define NO_START        -1
#define NO_REPEAT       -1
#define LOCK_BUTTON     -2

char mode[3];
char control;
char flash;
char flash_ds;
char beep;
char beep_ds;
char buzz;
char buzz_ds;
char view;
char view_ds;
char off;
char off_ds;
char alarm_day;
char clock_set;
char i2c_timeout;

/* Initialize function prototypes */
void init(void);

/* I/O function prototypes */
void flush(void);
void scan(void);;
char released(char, signed char, char);
char held(char, signed char, signed char, char);

/* Helper function prototpes */
char bcd2dec(char);
char dec2bcd(char);
char sn(char);

/* Alarm function prototypes */
void alarm_write(void);
void alarm_read(void);
char alarm_check(void);

/* Time function prototypes */
void time_write(void);
void time_read(void);
char time_format(char);
void time_volatile(char);

/* iPod function prototypes */
void ipod_play(void);
void ipod_pause(void);
void ipod_off(void);
void ipod_skip_forward(void);
void ipod_skip_back(void);
void ipod_volume_up(void);
void ipod_volume_down(void);
void ipod_cmd_play(void);
void ipod_cmd_pause(void);
void ipod_cmd_skip_forward(void);
void ipod_cmd_skip_back(void);
void ipod_cmd_volume_up(void);
void ipod_cmd_volume_down(void);
void ipod_cmd_stop(void);
void ipod_cmd_off(void);
void ipod_cmd_button_release(void);

/* SCI bus function prototypes */
void sci_write(unsigned char);

/* I2C bus function prototypes */
void i2c_write(char, char *, char);
void i2c_read(char, char *, char);
void i2c_start_timeout(void);

void main(void) {
  char i;

  EnableInterrupts;             // Enable interrupts
  CONFIG1_COPD = 1;             // Disable COP reset

  /* Configure SCI */
  SCC1_ENSCI = 1;               // Enable SCI

  /* Set output for 19,200 baud assuming a 9.8304MHz clock */
  SCBR_SCP = 0;                 // Baud rate prescaler = 1
  SCBR_SCR = 1;                 // Baud rate divisor = 2
  SCC2_TE = 1;                  // Enable transmitter

  /* Configure I2C */
  CONFIG2_IICSEL = 1;           // Set PTA bits 2-3 for I2C
  MIMCR_MMBR = 2;               // Set baud rate divisor
  MMCR_MMEN = 1;                // Enable MMII

  /* Set PTA bits 0-1, 4-5 for input */
  DDRA = DDRA & ~BIT0_MASK & ~BIT1_MASK & ~BIT4_MASK & ~BIT5_MASK;

  /* Set PTB bits 0-7 for output */
  DDRB = DDRB | BIT0_MASK | BIT1_MASK | BIT2_MASK | BIT3_MASK
              | BIT4_MASK | BIT5_MASK | BIT6_MASK | BIT7_MASK;

  /* Set PTD bits 0-2 for output, 3-5 for input */
  DDRD = DDRD | BIT0_MASK | BIT1_MASK | BIT2_MASK & ~BIT3_MASK & ~BIT4_MASK &
~BIT5_MASK;
```

```
/* Initialize the clock */
init();

/* Configure Timer 2 */
T2SC_TRST = 1;              // Reset timer
T2SC_PS = 2;                // Set prescalar for divide by 4
T2SC_TOIE = 0;              // Disable timer interrupt
T2MOD = 31250;              // Store modulo value in T1MODH:T1MODL
T2SC_TSTOP = 0;             // Start timer running

for(;;) {
  /* Update status of buttons */
  for (i=0; i<INPUT_COUNT; i++) {
    buttons[i][STATUS]=RELEASED;
  }
  if (input>0 && input<=INPUT_COUNT) {
    buttons[input-1][STATUS]=PRESSED;
  }

  /* Are we debugging? */
  if (debug != debug_switch) {
    debug = debug_switch;
    time_write();
  }

  /* Are we in 12/24 mode? */
  if (mode[TIME_MODE] != mode_switch) {
    mode[TIME_MODE] = mode_switch;
  }

  /* Are we in buzzer or iPod mode? */
  if (mode[ALARM_MODE] != alarm_switch) {
    mode[ALARM_MODE] = alarm_switch;
  }

  /* Did Timer 2 expire? */
  if (T2SC_TOF == 1) {
    T2SC_TOF = 0;             // Reenable timer
    if (!(control & FLASH_MASK)) {
      flash=TRUE;
      flash_ds=1;
    } else if (flash_ds++ % FLASH_INTERVAL == 0) {
      flash=!flash;
      flash_ds=1;
    }
    if (!(control & ALARM_ON)) {
      buzz=TRUE;
      buzz_ds=1;
    } else if (buzz_ds++ % BUZZ_INTERVAL == 0) {
      buzz=!buzz;
      buzz_ds=1;
    }
    if (!view) view_ds=1;
    else if (view_ds++ % VIEW_TIME == 0) {
      view=FALSE;
      view_ds=1;
    }
    if (!beep) beep_ds=1;
    else if (beep_ds++ % BEEP_TIME == 0) {
      beep=FALSE;
      beep_ds=1;
    }
    if (!off) off_ds=1;
    else if (off_ds++ % OFF_TIME == 0) {
      off=FALSE;
      off_ds=1;
      ipod_cmd_button_release();
    }

    /* Debug clock? */
    if (debug) {
      /* Speed up time for debugging */
      debug_time[SEC]+=DEBUG_SPEED;
      if (debug_time[SEC]>59) {
        debug_time[SEC]=0;
        debug_time[MIN]++;
      }
      if (debug_time[MIN]>59) {
        debug_time[MIN]=0;
        debug_time[HOUR]++;
      }
      if (debug_time[HOUR]>23) {
        debug_time[HOUR]=0;
        debug_time[DAY]++;
      }
      if (debug_time[DAY]>7) {
        debug_time[DAY]=1;
      }
    }

    scan();                   // Scan inputs
  }

  /* CLOCK mode */
  if (mode[CLOCK_MODE] == CLOCK) {
    control = NONE;

    if (clock_set == FALSE) control = FLASH_HOUR | FLASH_MIN;

    /* Update time */
    time_read();

    /* Change mode? */
    if (alarm_check()) {
      mode[CLOCK_MODE] = ACTIVATE_ALARM;
      if (mode[ALARM_MODE]==IPOD) {
        ipod_play();
      }
    } else if (released(SEL, 0, NO_BEEP)) {
      ipod_pause();
    } else if (released(DOWN, 0, NO_BEEP)) {
      ipod_skip_back();
    } else if (released(UP, 0, NO_BEEP)) {
      ipod_skip_forward();
    } else if (held(SEL, 20, LOCK_BUTTON, BEEP)) {
      time_read();
      mode[CLOCK_MODE] = SET_CLOCK_HOUR;
      time_volatile(FALSE);
    } else if (held(UP, 10, 4, NO_BEEP)) {
      ipod_volume_up();
    } else if (held(DOWN, 10, 4, NO_BEEP)) {
      ipod_volume_down();
    } else {
      for (i=SUN; i<=SAT; i++) {
        if (released(i, 0, NO_BEEP)) {
          alarm_day = i;
          view = TRUE;
          mode[CLOCK_MODE] = VIEW_ALARM;
        }
        else if (held(i, 10, NO_REPEAT, BEEP)) {
          alarm_day = i;
          mode[CLOCK_MODE] = ENABLE_ALARM;
        } else if (held(i, NO_START, 20, BEEP)) {
          alarm_day = i;

          /* Load alarm to set */
          time[HOUR]= alarms[alarm_day][HOUR];
          time[MIN] = alarms[alarm_day][MIN];

          mode[CLOCK_MODE] = SET_ALARM_HOUR;
        }
      }
    }
  }

  /* SET_CLOCK_HOUR mode */
  else if (mode[CLOCK_MODE] == SET_CLOCK_HOUR) {
    control = FLASH_HOUR;

    /* Decrease hour? */
    if (released(DOWN, 0, NO_BEEP)) time[HOUR]--;
    else if (held(DOWN, 20, 20, NO_BEEP)) {
      if (mode[TIME_MODE] == NORMAL) time[HOUR]-=12;
      else time[HOUR]--;
    }
    if (time[HOUR]<0) time[HOUR]+=24;

    /* Increase hour? */
    if (released(UP, 0, NO_BEEP)) time[HOUR]++;
    else if (held(UP, 20, 20, NO_BEEP)) {
      if (mode[TIME_MODE] == NORMAL) time[HOUR]+=12;
      else time[HOUR]++;
    }
    if (time[HOUR]>23) time[HOUR]-=24;

    /* Change mode? */
    if (held(SEL, 0, LOCK_BUTTON, BEEP)) mode[CLOCK_MODE]=SET_CLOCK_MIN;
  }

  /* SET_CLOCK_MIN mode */
  else if (mode[CLOCK_MODE] == SET_CLOCK_MIN) {
    control = FLASH_MIN;

    /* Decrease minute? */
    if (released(DOWN, 0, NO_BEEP)) time[MIN]--;
    else if (held(DOWN, 10, 2, NO_BEEP)) time[MIN]--;
    if (time[MIN]<0) time[MIN]+=60;

    /* Increase minute? */
    if (released(UP, 0, NO_BEEP)) time[MIN]++;
    else if (held(UP, 10, 2, NO_BEEP)) time[MIN]++;
    if (time[MIN]>59) time[MIN]-=60;

    /* Change mode? */
    if (held(SEL, 0, LOCK_BUTTON, BEEP)) mode[CLOCK_MODE]=SET_DAY;
  }

  /* SET_DAY mode */
  else if (mode[CLOCK_MODE] == SET_DAY) {
    control = FLASH_DAY;

    /* Decrease day? */
    if(released(DOWN, 0, NO_BEEP)) time[DAY]--;
    else if (held(DOWN, 10, 10, NO_BEEP)) time[DAY]--;
    if (time[DAY]<1) time[DAY]+=7;

    /* Increase day? */
    if(released(UP, 0, NO_BEEP)) time[DAY]++;
    else if (held(UP, 10, 10, NO_BEEP)) time[DAY]++;
    if (time[DAY]>7) time[DAY]-=7;

    /* Change mode? */
    if (held(SEL, 0, LOCK_BUTTON, BEEP)) {
      mode[CLOCK_MODE]=CLOCK;
      time[SEC] = 0;
      time_write();
      time_volatile(TRUE);
    }
  }

  /* SET_ALARM_HOUR mode */
  else if (mode[CLOCK_MODE] == SET_ALARM_HOUR) {
    control = FLASH_HOUR | FLASH_ALARM_DAY;

    /* Decrease hour? */
    if (released(DOWN, 0, NO_BEEP)) time[HOUR]--;
    else if (held(DOWN, 10, 20, NO_BEEP)) {
      if (mode[TIME_MODE] == NORMAL) time[HOUR]-=12;
      else time[HOUR]--;
    }
    if (time[HOUR]<0) time[HOUR]+=24;

    /* Increase hour? */
    if (released(UP, 0, NO_BEEP)) time[HOUR]++;
    else if (held(UP, 10, 20, NO_BEEP)) {
      if (mode[TIME_MODE] == NORMAL) time[HOUR]+=12;
      else time[HOUR]++;
    }
    if (time[HOUR]>23) time[HOUR]-=24;

    /* Change mode? */
    if (held(SEL, 0, LOCK_BUTTON, BEEP)) mode[CLOCK_MODE]=SET_ALARM_MIN;
  }

  /* SET_ALARM_MIN mode */
  else if (mode[CLOCK_MODE] == SET_ALARM_MIN) {
    control = FLASH_MIN | FLASH_ALARM_DAY;

    /* Decrease minute? */
    if (released(DOWN, 0, NO_BEEP)) time[MIN]--;
```

```
    else if (held(DOWN, 10, 2, NO_BEEP)) time[MIN]--;
    if (time[MIN]<0) time[MIN]+=60;

    /* Increase minute? */
    if (released(UP, 0, NO_BEEP)) time[MIN]++;
    else if (held(UP, 10, 2, NO_BEEP)) time[MIN]++;
    if (time[MIN]>59) time[MIN]-=60;

    /* Change mode? */
    if (held(SEL, 0, LOCK_BUTTON, BEEP)) {
      mode[CLOCK_MODE]=CLOCK;

      /* Enable and save the alarm */
      alarms[alarm_day][HOUR]=time[HOUR];
      alarms[alarm_day][MIN]=time[MIN];
      alarms[alarm_day][ALM_ENABLE] = TRUE;
      alarm_write();
      alarm_day = NONE;
    }
  }

  /* VIEW_ALARM mode */
  else if (mode[CLOCK_MODE] == VIEW_ALARM) {
    control = FLASH_ALARM_DAY;

    /* Update time */
    time_read();

    /* Change mode? */
    if (alarm_check()) {
      mode[CLOCK_MODE] = ACTIVATE_ALARM;
      if (mode[ALARM_MODE]==IPOD) {
        ipod_play();
      }
    } else if (held(SEL, 20, LOCK_BUTTON, BEEP)) {
      time_read();
      mode[CLOCK_MODE] = SET_CLOCK_HOUR;
      time_volatile(FALSE);
    } else if (held(UP, 0, LOCK_BUTTON, NO_BEEP)) {
      ipod_volume_up();
    } else if (held(DOWN, 0, LOCK_BUTTON, NO_BEEP)) {
      ipod_volume_down();
    } else {
      for (i=SUN; i<=SAT; i++) {
        if (released(i, 0, NO_BEEP)) {
          alarm_day = i;
          view = TRUE;
          view_ds=1;
          mode[CLOCK_MODE] = VIEW_ALARM;
        }
        else if (held(i, 10, NO_REPEAT, BEEP)) {
          alarm_day = i;
          mode[CLOCK_MODE] = ENABLE_ALARM;
        } else if (held(i, NO_START, 20, BEEP)) {
          alarm_day = i;

          /* Load alarm to set */
          time[HOUR]= alarms[alarm_day][HOUR];
          time[MIN] = alarms[alarm_day][MIN];
```

```
            mode[CLOCK_MODE] = SET_ALARM_HOUR;
          }
        }
      }

      /* Did the view time expire? */
      if (view == FALSE) {
        alarm_day = NONE;
        mode[CLOCK_MODE]=CLOCK;
      }

      /* Load alarm to view */
      time[HOUR]= alarms[alarm_day][HOUR];
      time[MIN] = alarms[alarm_day][MIN];
    }

    /* ENABLE_ALARM mode */
    else if (mode[CLOCK_MODE] == ENABLE_ALARM) {
      control = NONE;

      /* Toggle alarm enable */
      alarms[alarm_day][ALM_ENABLE] = !alarms[alarm_day][ALM_ENABLE];
      alarm_write();
      alarm_day = NONE;

      mode[CLOCK_MODE]=CLOCK;
    }

    /* ACTIVATE_ALARM mode */
    else if (mode[CLOCK_MODE] == ACTIVATE_ALARM) {
      control = FLASH_HOUR | FLASH_MIN;
      if (mode[ALARM_MODE]==BUZZER) {
        control |= ALARM_ON;
      }

      /* Update time */
      time_read();

      /* Change mode? */
      if (released(DOWN, 0, NO_BEEP) | released(SEL, 0, NO_BEEP) |
released(UP, 0, NO_BEEP)) {
        mode[CLOCK_MODE]=CLOCK;
        alarms[SNOOZE][ALM_ENABLE]=TRUE;
        alarms[SNOOZE][MIN]=time[MIN]+SNOOZE_TIME;
        alarms[SNOOZE][HOUR]=time[HOUR];
        if (alarms[SNOOZE][MIN]>59) {
          alarms[SNOOZE][MIN]-=60;
          alarms[SNOOZE][HOUR]++;
        }
        if (alarms[SNOOZE][HOUR]>23) {
          alarms[SNOOZE][HOUR]-=24;
        }
        if (mode[ALARM_MODE]==IPOD) {
          ipod_off();
        }
      } else if (held(DOWN, 20, LOCK_BUTTON, NO_BEEP) | held(SEL, 20,
LOCK_BUTTON, NO_BEEP) | held(UP, 20, LOCK_BUTTON, NO_BEEP)) {
        mode[CLOCK_MODE]=CLOCK;
        alarms[SNOOZE][ALM_ENABLE]=FALSE;
        if (mode[ALARM_MODE]==IPOD) {
          ipod_off();
        }
      }
    }

    /* Flush output */
    flush();
  }
}

/* Initalizes the clock upon bootup */
void init(void) {
  int i;

  /* Check for power loss */
  buffer[0] = CLOCK_MODE_ADDR;
  i2c_write(CLOCK_ADDR, buffer, 1);
  i2c_read(CLOCK_ADDR, buffer, 1);
  if (buffer[0] != CLOCK_SET) time_volatile(FALSE);
  else time_volatile(TRUE);

  /* Set modes */
  mode[CLOCK_MODE]=NORMAL;
  mode[TIME_MODE]=NORMAL;

  /* Initialize timers */
  flash_ds=1;
  beep_ds=1;
  buzz_ds=1;
  view_ds=1;
  off_ds=1;

  /* Initalize clock chip */
  buffer[0] = CLOCK_SEC_ADDR;
  i2c_write(CLOCK_ADDR, buffer, 1);
  i2c_read(CLOCK_ADDR, buffer, 3);
  buffer[3] = buffer[2] & ~TIME_MODE_MASK; // Set 12/24 = 0
  buffer[2] = buffer[1];
  buffer[1] = buffer[0] & ~CH_MASK;        // Set CH = 0
  buffer[0] = CLOCK_SEC_ADDR;
  i2c_write(CLOCK_ADDR, buffer, 4);
  buffer[0] = CLOCK_CTR_ADDR;
  buffer[1] = CLOCK_CTR_BITS;
  i2c_write(CLOCK_ADDR, buffer, 2);

  /* Load time and alarms */
  time_read();
  alarm_read();
  alarms[SNOOZE][ALM_ENABLE]=FALSE;

  /* Wake iPod up */
  ipod_skip_back();

  /* Wait for iPod to wake up */
  for (i=0; i<1000; i++) {}

  /* Turn the iPod off */
  ipod_off();
}
```

```
/* Flushes the output to the board */
void flush() {
  char i;

  /* Output hour */
  if (control & FLASH_HOUR && !flash) data = BLANK;
  else {
    /* Format the hour, convert to BCD and swap the nibbles! */
    data = sn(dec2bcd(time_format(time[HOUR])));

    /* Display AM/PM? */
    if (mode[TIME_MODE] == NORMAL && (data & 0x0F) == 0) data = data | 0x0F;
  }
  addr = HOUR_ADDR;
  addr = SEND;

  /* Output minute */
  if (control & FLASH_MIN && !flash) data = BLANK;
  else {
    /* Convert to BCD and swap the nibbles! */
    data = sn(dec2bcd(time[MIN]));
  }
  addr = MIN_ADDR;
  addr = SEND;

  /* Write output bank 0 */
  if ((control & FLASH_DAY) && !flash) output[0] = output[0] | DAYS_MASK;
  else output[0] = (output[0] | DAYS_MASK) & ~(1 << time[DAY]);
  data = output[0];
  addr = OUTPUT0_ADDR;
  addr = SEND;

  /* Write output bank 1 */
  if (beep) output[1] = (output[1] | BEEP_MASK);
  else output[1] = output[1] & ~BEEP_MASK;
  for (i=SUN; i<=SAT; i++) {
    if ( (alarms[i][ALM_ENABLE] && i!=alarm_day) ||
          (alarms[i][ALM_ENABLE] && !(control & FLASH_ALARM_DAY) && i ==
alarm_day) ||
          ((control & FLASH_ALARM_DAY) && flash && i == alarm_day) ) {
      output[1] = output[1] & ~(1 << i);
    }
    else output[1] = output[1] | (1 << i);
  }
  data = output[1];
  addr = OUTPUT1_ADDR;
  addr = SEND;

  /* Write output bank 2 */
  if ((control & ALARM_ON) && buzz) output[2] = (output[2] | ALARM_MASK);
  else output[2] = output[2] & ~ALARM_MASK;
  if ((control & FLASH_HOUR) || (control & FLASH_MIN) || (control &
FLASH_DAY) || (control & FLASH_ALARM_DAY)
           || (!debug && (time[SEC] % 2 == 0)) || (debug && (time[MIN] % 2
== 0))) output[2] = (output[2] & ~COLON_MASK);
  else output[2] = (output[2] | COLON_MASK);
  if ((control & FLASH_HOUR) && !flash) output[2] = (output[2] |
ONE_DIGIT_MASK);

  data = output[2];
  addr = OUTPUT2_ADDR;
  addr = SEND;
}

/* Scan the buttons and update the states */
void scan() {
  char i;
  for (i=0; i<INPUT_COUNT; i++) {
    if (buttons[i][STATUS]==PRESSED && buttons[i][STATE]!=STATE_LOCKED) {
      if (buttons[i][STATE]!=STATE_HELD) buttons[i][STATE]=STATE_PRESSED;
      else buttons[i][KEYFIRE_ELAPSE]++;
      if (buttons[i][TOTAL_ELAPSE]<255) buttons[i][TOTAL_ELAPSE]++;
    } else if (buttons[i][STATUS]==RELEASED &&
buttons[i][STATE]==STATE_PRESSED) {
      buttons[i][STATE]=STATE_RELEASED;
      buttons[i][PREV_ELAPSE]=buttons[i][TOTAL_ELAPSE];
      buttons[i][TOTAL_ELAPSE]=0;
      buttons[i][KEYFIRE_ELAPSE]=0;
    } else if (buttons[i][STATUS]==RELEASED &&
buttons[i][STATE]==STATE_RELEASED) {
      buttons[i][STATE]=STATE_RESET;
    } else if (buttons[i][STATUS]==RELEASED && (buttons[i][STATE]==STATE_HELD
|| buttons[i][STATE]==STATE_LOCKED)) {
      buttons[i][STATE]=STATE_RESET;
      buttons[i][PREV_ELAPSE]=0;
      buttons[i][TOTAL_ELAPSE]=0;
      buttons[i][KEYFIRE_ELAPSE]=0;
    }
  }
}

/* Check if button n has been held for t 1/20's of a second; r determines
repeat setting; b determines if tactile feedback is given */
char held(char n, signed char t, signed char r, char b) {
    assert(n>0 && n<=INPUT_COUNT);
    n--;
    if (buttons[n][STATUS]==PRESSED && buttons[n][STATE]!=STATE_LOCKED &&
buttons[n][STATE]!=STATE_HELD) {
      buttons[n][STATE]=STATE_PRESSED;
    }
    if (buttons[n][TOTAL_ELAPSE]>=t && t != NO_START &&
buttons[n][KEYFIRE_ELAPSE]==0 && buttons[n][STATE]==STATE_PRESSED) {
      if (r == LOCK_BUTTON) buttons[n][STATE]=STATE_LOCKED;
      else buttons[n][STATE]=STATE_HELD;
      if (b) beep=TRUE;
      return TRUE;
    } else if (buttons[n][KEYFIRE_ELAPSE]>=r && r != NO_REPEAT &&
buttons[n][STATE]==STATE_HELD) {
      buttons[n][KEYFIRE_ELAPSE]=0;
      if (b) beep=TRUE;
      return TRUE;
    }
    else return FALSE;
}

/* Check if button n has been held for t 1/20's of a second and then
released; b determines if tactile feedback is given */
char released(char n, signed char t, char b) {
    assert(n>0 && n<=INPUT_COUNT);
    n--;
    if (buttons[n][STATUS]==RELEASED && buttons[n][STATE]==STATE_PRESSED) {
      buttons[n][STATE]=STATE_RELEASED;
      buttons[n][PREV_ELAPSE]=buttons[n][TOTAL_ELAPSE];
      buttons[n][TOTAL_ELAPSE]=0;
      buttons[n][KEYFIRE_ELAPSE]=0;
    }
    if (buttons[n][STATE]==STATE_RELEASED && buttons[n][PREV_ELAPSE]>=t) {
      buttons[n][PREV_ELAPSE]=0;
      buttons[n][STATE]=STATE_RESET;
      if (b) beep=TRUE;
      return TRUE;
    }
    else return FALSE;
}

/* Helper function to convert a decimal value to BCD */
char dec2bcd(char n) { assert(n<100); return ((n / 10) << 4) | (n % 10); }

/* Helper fuction to convert a BCD value to decimal */
char bcd2dec(char n) { return ( ((n >> 4) * 10) + (n & 0x0F) ); }

/* Helper function to swap nibbles in a byte. This hack is because we
   wired the most significant digit to the the least significant seven
   segment digit and vice versa. Oops! */
char sn(char n) { return (n << 4) | (n >> 4); }

/* Saves the alarm for day d and writes the alarms to the EEPROM over the I2C
bus */
void alarm_write(void) {
  buffer[0] = EEPROM_MSB_ADDR;
  buffer[1] = EEPROM_ALM_ADDR + ALARM_SIZE*alarm_day;
  buffer[2] = alarms[alarm_day][HOUR];
  buffer[3] = alarms[alarm_day][MIN];
  buffer[4] = alarms[alarm_day][ALM_ENABLE];
  i2c_write(EEPROM_ADDR, buffer, 5);
}

/* Reads the saved alarms from the EEPROM over the I2C bus */
void alarm_read(void) {
  char i, j;
  buffer[0] = EEPROM_MSB_ADDR;
  buffer[1] = EEPROM_ALM_ADDR;
  i2c_write(EEPROM_ADDR, buffer, 2);
  i2c_read(EEPROM_ADDR, buffer, (ALARM_SIZE * ALARM_COUNT));
  for (i=0; i<ALARM_COUNT; i++) {
    for (j=0; j<ALARM_SIZE; j++) {
      alarms[i][j] = buffer[(ALARM_SIZE*i + j)];
    }
  }
}

/* Checks the alarms for a match */
char alarm_check(void) {
  if (((alarms[time[DAY]][ALM_ENABLE] && alarms[time[DAY]][HOUR]==time[HOUR]
&& alarms[time[DAY]][MIN]==time[MIN])
        || (alarms[SNOOZE][ALM_ENABLE] && alarms[SNOOZE][HOUR]==time[HOUR] &&
alarms[SNOOZE][MIN]==time[MIN]))
        && time[SEC]==0) return TRUE;
  else return FALSE;
}
```

```
/* Writes the time to the clock over the I2C bus */
void time_write(void) {
  if (debug) {
    debug_time[SEC] = time[SEC];
    debug_time[MIN] = time[MIN];
    debug_time[HOUR]= time[HOUR];
    debug_time[DAY] = time[DAY];
  }
  buffer[0] = CLOCK_SEC_ADDR;
  buffer[1] = (dec2bcd(time[SEC]) & SEC_MASK);
  buffer[2] = dec2bcd(time[MIN]) & MIN_MASK;
  buffer[3] = dec2bcd(time[HOUR]) & HOUR_MASK;
  buffer[4] = time[DAY] & DAY_MASK;
  i2c_write(CLOCK_ADDR, buffer, 5);
}

/* Reads the time from the clock over the I2C bus */
void time_read(void) {
  if (debug) {
    time[SEC] = debug_time[SEC];
    time[MIN] = debug_time[MIN];
    time[HOUR]=debug_time[HOUR];
    time[DAY] =debug_time[DAY];
  } else {
    buffer[0] = CLOCK_SEC_ADDR;
    i2c_write(CLOCK_ADDR, buffer, 1);
    i2c_read(CLOCK_ADDR, buffer, 4);
    time[SEC] = bcd2dec(buffer[0] & SEC_MASK);
    time[MIN] = bcd2dec(buffer[1] & MIN_MASK);
    time[HOUR]= bcd2dec(buffer[2] & HOUR_MASK);
    time[DAY] = buffer[3] & DAY_MASK;
  }
}

/* Formats the time to either normal or military time for displaying */
char time_format(char hour) {
  if (mode[TIME_MODE] == NORMAL ) {
    if (hour>=12) output[0] = output[0] & ~AM_PM_MASK;
    else output[0] = output[0] | AM_PM_MASK;
    if (hour==0 | (hour>=10 && hour <=12) | (hour>=22 && hour <=23))
output[2] = output[2] & ~ONE_DIGIT_MASK;
    else output[2] = output[2] | ONE_DIGIT_MASK;
    if (hour==0) return 12;
    else if (hour>12) return hour-12;
    else return hour;
  } else if (mode[TIME_MODE] == MILITARY) {
    if (hour>=10 && hour <=19) output[2] = output[2] & ~ONE_DIGIT_MASK;
    else output[2] = output[2] | ONE_DIGIT_MASK;
    output[0] = output[0] | AM_PM_MASK;
    return hour;
  }
}

/* Saves the time volatility to the clock chip */
void time_volatile(char b) {
  if (b == TRUE) {
    clock_set = TRUE;
    buffer[0] = CLOCK_MODE_ADDR;
    buffer[1] = CLOCK_SET;
    i2c_write(CLOCK_ADDR, buffer, 2);
  } else {
    clock_set = FALSE;
    buffer[0] = CLOCK_MODE_ADDR;
    buffer[1] = CLOCK_NOT_SET;
    i2c_write(CLOCK_ADDR, buffer, 2);
  }
};

/* Starts the iPod playing */
void ipod_play(void) {
  ipod_cmd_play();
  ipod_cmd_button_release();
  ipod_cmd_skip_back();
  ipod_cmd_button_release();
}

/* Pauses the iPod playing */
void ipod_pause(void) {
  ipod_cmd_pause();
  ipod_cmd_button_release();
}

/* Stops the iPod playing */
void ipod_off (void) {
  ipod_cmd_stop();
  ipod_cmd_button_release();
  ipod_cmd_pause();
  off = TRUE;
}

/* Skips forward on the iPod */
void ipod_skip_forward(void) {
  ipod_cmd_skip_forward();
  ipod_cmd_button_release();
}

/* Skips back on the iPod */
void ipod_skip_back(void) {
  ipod_cmd_skip_back();
  ipod_cmd_button_release();
}

/* Increases the iPod volume */
void ipod_volume_up(void) {
  ipod_cmd_volume_up();
  ipod_cmd_button_release();
}

/* Decreases the iPod volume */
void ipod_volume_down(void) {
  ipod_cmd_volume_down();
  ipod_cmd_button_release();
}

/* Writes the play command to the iPod */
void ipod_cmd_play(void) {
  /* Play */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x04);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x00);
  sci_write(0x01);
  sci_write(0xF9);              // Checksum
}

/* Writes the pause command to the iPod */
void ipod_cmd_pause(void) {
  /* Pause */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x01);
  sci_write(0xFA);              // Checksum
}

/* Writes the stop command to the iPod */
void ipod_cmd_stop(void) {
  /* Stop */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x80);
  sci_write(0x7B);              // Checksum
}

/* Writes the skip forward command to the iPod */
void ipod_cmd_skip_forward(void) {
  /* Skip back */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x08);
  sci_write(0xF3);              // Checksum
}

/* Writes the skip back command to the iPod */
void ipod_cmd_skip_back(void) {
  /* Skip back */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x10);
  sci_write(0xEB);              // Checksum
}

/* Writes the volume up command to the iPod */
void ipod_cmd_volume_up(void) {
  /* Volume up */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x02);
  sci_write(0xF9);              // Checksum
}

/* Writes the volume down command to the iPod */
void ipod_cmd_volume_down(void) {
  /* Volume down */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x04);
  sci_write(0xF7);              // Checksum
}

/* Writes the button release command to the iPod */
void ipod_cmd_button_release(void) {
  /* Button release */
  sci_write(0xFF);              // Header
  sci_write(0x55);
  sci_write(0x03);              // Length
  sci_write(0x02);              // Mode
  sci_write(0x00);              // Command
  sci_write(0x00);
  sci_write(0xFB);              // Checksum
}

/* Writes the byte ch to SCI port */
void sci_write(unsigned char ch) {
    while (SCS1_SCTE == 0);
    SCDR = ch;
}

/* Writes num_bytes bytes to an I2C device at address addr */
void i2c_write(char device_addr, char *p, char num_bytes) {
  i2c_start_timeout();          // Start I2C watchdog timer
  assert(num_bytes >= 1);
  while (MIMCR_MMBB)  {         // Wait for bus not busy
    if (i2c_timeout) return;
  }
  if (i2c_timeout) return;
  MMSR_MMTXIF = 0;              // Set MMDRR writable
  MIMCR_MMRW = 0;               // Set for transmit
  MMADR = device_addr;          // Device address -> address reg
  MMDTR = *p++;                 // First byte of data to write
  MIMCR_MMAST = 1;              // Start transmission
  while (MMSR_MMRXAK)  {
    if (i2c_timeout) return;
  }
  if (i2c_timeout) return;
  while (num_bytes-- > 0) {
    i2c_start_timeout();        // Start I2C watchdog timer
    while (!MMSR_MMTXBE) {      // Wait for TX buffer
      if (i2c_timeout) return;
    }
    if (i2c_timeout) return;
    if (num_bytes == 0)         // Is this the last byte?
      MMDTR = 0xFF;             // Yes. dummy data -> DTR
    else
      MMDTR = *p++;             // No. next data -> DTR
    while (MMSR_MMRXAK) {       // Wait for ACK from slave
      if (i2c_timeout) return;
    }
    if (i2c_timeout) return;
  }
  MIMCR_MMAST = 0;              // Generate STOP bit
  T1SC_TSTOP = 1;               // Stop I2C watchdog timer
}

/* Read num_bytes bytes from an I2C device at address addr */
void i2c_read(char device_addr, char *p, char num_bytes) {
    i2c_start_timeout();        // Start I2C watchdog timer
    assert(num_bytes >= 1);
    while (MIMCR_MMBB) {
      if (i2c_timeout) return;
    }
    if (i2c_timeout) return;
    MMSR_MMRXIF = 0;
    MIMCR_MMRW = 1;             // Set for receive
    if (num_bytes == 1)
      MMCR_MMTXAK = 1;
    else
      MMCR_MMTXAK = 0;
    MMADR = device_addr;        // Device address -> address reg
    MMDTR = 0xFF;               // Dummy data to get ACK clock
    MIMCR_MMAST = 1;            // Initiate transfer
    while (MMSR_MMRXAK) {       // Wait for ACK from slave
      if (i2c_timeout) return;
    }
    if (i2c_timeout) return;
    while (num_bytes-- > 0) {   // Up to last byte
        i2c_start_timeout();    // Start I2C watchdog timer
        while (!MMSR_MMRXBF) {  // Wait for RX buffer full
          if (i2c_timeout) return;
        }
        if (i2c_timeout) return;
        if (num_bytes == 1)
          MMCR_MMTXAK = 1;
        else
          MMCR_MMTXAK = 0;
        *p++ = MMDRR;           // Get data
    }
    MIMCR_MMAST = 0;            // Generate STOP bit
    T1SC_TSTOP = 1;             // Start I2C watchdog timer
}

/* Reset the I2C bus if the I2C watchdog timer expired */
void i2c_reset(void) {
    MMCR_MMEN = 0;
    MMCR_MMEN = 1;
    i2c_timeout = TRUE;
}

/* Start the I2C watchdog timer */
void i2c_start_timeout(void) {
    i2c_timeout = FALSE;
    T1SC_TRST = 1;              // Reset timer
    T1SC_PS = 6;                // Set prescalar for divide by 64
    T1SC_TOIE = 1;              // enable timer interrupt
    T1MOD = 65535;              // store modulo value in T1MODH:T1MODL
    T1SC_TSTOP = 0;             // start timer running
}

/* The ISR for Timer 1 */
#pragma TRAP_PROC
void i2c_watchdog(void) {
    T1SC_TOF = 0;               // Reenable timer
    i2c_reset();
}
```

Appendix C – Software Code (project.prm)

```
/* This is a linker parameter file for the mc68hc908jl16 */

NAMES END /* CodeWarrior will pass all the needed files to the linker by
command line. But here you may add your own files too. */

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in
PLACEMENT below. */
    Z_RAM                    =  READ_WRITE   0x0060 TO 0x00FF;
    RAM                      =  READ_WRITE   0x0100 TO 0x025F;
    ROM0                     =  READ_ONLY    0xBC00 TO 0xBC0F;
    ROM                      =  READ_ONLY    0xBC10 TO 0xFBFF;
 /* INTVECTS                 =  READ_ONLY    0xFFDE TO 0xFFFF; Reserved for
Interrupt Vectors */
END

PLACEMENT /* Here all predefined and user segments are placed into the
SEGMENTS defined above. */
    DEFAULT_RAM                        /* non-zero page variables */
                                       INTO  RAM;

    _PRESTART,                         /* startup code */
    STARTUP,                           /* startup data structures */
    ROM_VAR,                           /* constant variables */
    STRINGS,                           /* string literals */
    VIRTUAL_TABLE_SEGMENT,             /* C++ virtual table segment */
    DEFAULT_ROM,
    COPY                               /* copy down information: how to
initialize variables */
                                       INTO  ROM;

    MY_ISR_ROM                         INTO  ROM0;

    _DATA_ZEROPAGE,                    /* zero page variables */
    MY_ZEROPAGE                        INTO  Z_RAM;
END


STACKSIZE 0x50

VECTOR 0 _Startup /* Reset vector: this is the default entry point for an
application. */
VECTOR ADDRESS 0xFFF2 i2c_watchdog
VECTOR ADDRESS 0xFFF6 dummyISR
VECTOR ADDRESS 0xFFF8 dummyISR
VECTOR ADDRESS 0xFFFA dummyISR
VECTOR ADDRESS 0xFFFC dummyISR
```

Appendix D – Cost Analysis

Part #	Description	Qty	Price	Total
MC908JL16CPE	iC	1	$2.19	$2.19
VI-319-DP-FC-S	7-Segment LCD	1	$6.80	$6.80
KIT-LED-41	LCD Backlight	1	$3.25	$3.25
LEDs		14	$0.27	$3.78
74LS374N	Register	4	$0.45	$1.80
74LS247	BCD Converter	4	$0.96	$3.84
74LS08N	Hex AND Gate	1	$0.32	$0.32
74LS138N	3-8 Decoder	1	$0.41	$0.41
74LS148N	8-3 Encoder	2	$0.80	$0.80
DS1307	RTC	1	$1.69	$1.69
24LC256	EEPROM 256k	1	$1.35	$1.35
DS32KHZ	Oscillator	1	$2.55	$2.55
10Mhz	Oscillator	1	$1.27	$1.27
QT1103		1	$2.17	$2.17
Resistors Pack	330Ù, 1Ù , 2Ù, 10KÙ , 500KÙ	6	$0.31	$1.86
Capacitors	100pF, .1uF	20	$0.10	$2.00
3.3v Battery		1	$0.85	$0.85
Speakers		2	$1.59	$3.18
Buzzer		1	$0.50	$0.50
Piezoelectric Speaker		1	$0.50	$0.50
iPod Dock Adaptor		1	$0.50	$0.50
AC Adaptor		1	$4.35	$4.35
			Total	$45.96
			40% Volume Discount	($18.38)
			Subtotal	$27.58
			Audio System	$15.00
			Cost Basis	$42.58
			Price Point	$49.99
			Profit	$7.41
			Profit Margin	14.8%

Appendix E – User Manual

Setting the Time and Day

- Hold the SET button for 1 second and the “Hour” digits will start blinking.
  - *The alarm will beep to indicate you have entered set hour mode.*
- Tap the INCREMENT or DECREMENT buttons until the correct hour is shown. Holding the INCREMENT or DECREMENT buttons will quickly switch between AM and PM.
  - *Note: A triangle icon lights up in the upper left of the clock display to indicate PM time.*
- Tap the SET button and the “Minute” digits will start blinking.
  - *The alarm will beep to indicate you have entered set minute mode.*
- Tap the INCREMENT or DECREMENT buttons until the correct minute is shown. Holding the INCREMENT or DECREMENT buttons allows for cycling through the minutes quicker.
- Tap the SET button and the “Day” LED will start blinking.
  - *The alarm will beep to indicate you have entered set day mode.*
- Tap the INCREMENT or DECREMENT buttons until the correct day is shown. Holding the INCREMENT or DECREMENT buttons allows for cycling through the days quicker.
- Tap the SET button to finish setting the time and return to normal clock mode.
  - *The alarm will beep to indicate the time has been saved.*

Displaying an Alarm

- Tap one of the 7 alarm buttons and the display will show the alarm for that day. The alarm LED will blink indicating which alarm is being displayed.
  - *The alarm will automatically return to normal clock mode after 4 seconds.*

Enabling/Disabling an Alarm

- Hold one of the 7 alarm buttons for a ½ second and the alarm will either be enabled or disabled as indicated by the alarm LED.
  - *When the alarm LED is ON, that days alarm is enabled..*
  - *When the alarm LED is OFF, that days alarm is disabled.*

Setting an Alarm Time

- Hold one of the 7 alarm buttons for 1 second and the “Hour” digits will start blinking for that alarm.
  - *The alarm will beep to indicate you have entered set alarm hour mode.*
- Tap the INCREMENT or DECREMENT buttons until the correct hour is shown. Holding the INCREMENT or DECREMENT buttons will quickly switch between AM and PM.
  - *Note: A triangle icon lights up in the upper left of the clock display to indicate PM time.*
- Tap the SET button and the “Minute” digits will start blinking.
  - *The alarm will beep to indicate you have entered set minute mode.*
- Tap the INCREMENT or DECREMENT buttons until the correct minute is shown. Holding the INCREMENT or DECREMENT buttons allows for cycling through the minutes quicker.
- Tap the SET button to finish setting the alarm and return to normal clock mdoe.
  - *The alarm will beep to indicate the alarm has been saved.*

Snooze

- When an alarm is activated, tap the SET, INCREMENT, or DECREMENT button to snooze the alarm and set an alarm to be activated in 10 minutes.

Acknowledging Alarm

- When an alarm is activated, hold the SET, INCREMENT, or DECREMENT button for 1 second to acknowledge the alarm.

iPod/Buzzer Mode

- To switch between iPod/Buzzer mode, flip the alarm mode toggle switch.
  - In buzzer mode, the alarm will be sounded with a pulsing buzzing sound.
  - In iPod mode, the alarm will be sounded by playing a song from the iPod.
    - *Note: If the alarm is in iPod mode, and no iPod is connected then no sound will be emitted when the alarm activates.*

Appendix F – Proportional Work

Item	Tony Chen	Nate Houk	Ashwin Sathe
System Design	33%	33%	33%
Hardware Design	40%	40%	20%
Hardware Assembly	40%	40%	20%
Hardware Debugging	25%	25%	50%
Software Design	0%	100%	0%
Software Coding	0%	100%	0%
Software Debugging	0%	100%	0%
Alarm Clock Casing	0%	20%	80%
Documentation/Schematics	90%	10%	0%
Project report (oral)	40%	40%	20%
Project report (written)	30%	40%	30%

Tony Chen ____________________________

Nate Houk_____________________________

Ashwin Sathe __________________________