Apple Home

Smart Home Embedded System Design

Ethan Chan, Leo Linsky, Rob Puncel, Jessie Qiu, David Tan, Adi Yogev

I. Abstract

The modern home is teeming with consumer electronics; smart home automation systems capitalize on the increasing number of electronically controllable appliances to make the lives of their patrons more convenient. Air conditioning units, space heaters, fans, lights, security systems, and entertainment systems are amongst the plethora of easily automated devices that have become ubiquitous in the typical American household. It is no surprise, then, that various service providers have begun offering complex home control systems to automate certain tasks or products, sometimes even remotely through smart-phone apps or websites. However, a homeowner must often rely on disparate control systems for their heater, air conditioner, lights, security, and other systems. Additionally, homeowners are usually dependent on the vendors of the particular appliances in their homes to provide means of control - there is no "standard interface." Until now. The Apple Home seeks to centralize various household controls with an accessible, easy-to-use interface that eliminates its client's need to control her various devices separately. The Apple Home provides smart climate regulation and light controls, using remote sensors and controllers to completely automate the homeowners central systems. The Apple Home easily interfaces with pre-existing devices, limiting set-up to a one-time installation. The Apple Home will also be able to provide energy efficiency through its control algorithms, and will allow a user to disable home appliances remotely when they are not home through a seamless iOS interface. The Apple Home aims to bring the highly integrated, familiar Apple design paradigm into the home.

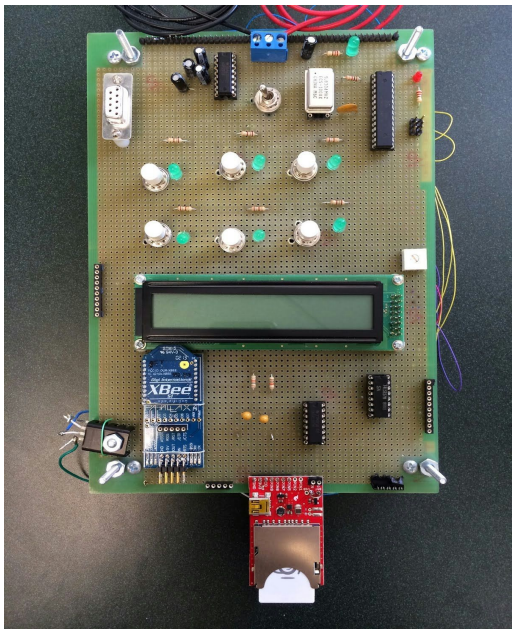
Designer Concept

II. System Overview

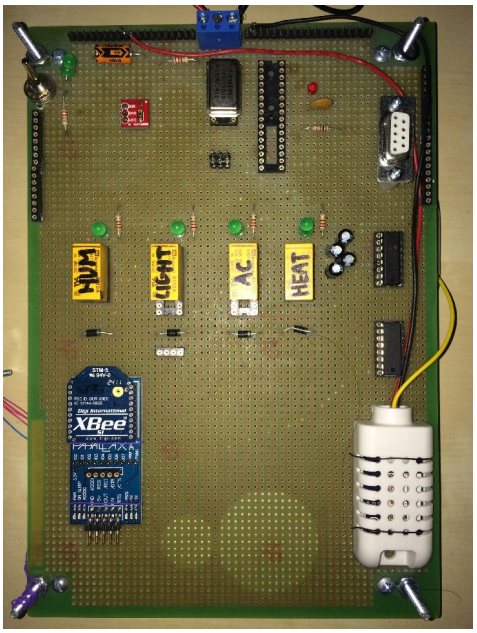
The Apple Home is a configurable and automated control system that uses temperature, humidity, and lighting in its environment to make decisions to manipulate a home's heater, air conditioner, humidifier, and lights. The Apple Home consists of three separate physical components: a main unit with a user interface consisting of an LCD screen and buttons, a remote unit that consists of the system's sensors as well as solid state relays to control home appliances, and a companion iOS application to provide a user interface on a user's phone or tablet.

The main unit communicates with both the iPhone App and the remote sensor modules, which communicate only through the main unit. The main unit receives sensor readings from the remote unit, displays the values on its LCD, and forwards sensor values to the iOS application. The main unit also relays commands from both the LCD and the application to the remote unit to influence its outputs.

The Apple Home controls home appliances by interfacing with their direct Alternating Current power lines to control the on/off state of the appliances. The Apple Home implements control decisions through an internal state machine that supports always on, always off, and automatic modes. The state machine, when in always on or always off modes, defers to a user's choice as set through either of the two user interfaces. In auto mode, the Apple Home uses internally configured hysteresis in order to decide actions to take. In general, the Home avoids oscillatory actions by "overdoing" a certain action so that minor fluctuations in the environment do not trigger immediately after turning off an appliance that was recently on.



Main/Control Unit Board



Remote Sensor/Outputs Board

III. Main User Interface

User Guide

In order to make the Apple Home user-friendly, we utilize a Crystalfontz LCD screen to display the modules associated with the current state of a room in the household. The information associated with a room's state includes three things: 1) The current temperature, humidity , and lighting of a room, 2) The desired temperature, humidity, and lighting of a room, and 3) The state of the devices that control temperature, humidity, and lighting such as an A/C unit, humidifier, or light switch. The display is only half of the interface, as any user would like to have the ability to modify their desired settings for a particular room. The user interface is controlled through the use of four standard push-buttons. For the scope of our prototype, we use push-buttons, but a commercial Apple product would transition to a touch-screen compatible with iOS.

LCD

On the main board, all user settings and sensor inputs are displayed via a 24x2 Crystalfontz LCD screen. For this display to work, the main board formats relevant data into 24 character wide arrays of characters, which are then output to the LCD one character at a time. To compensate for the slow string formatting time, all I/O on the main board is delayed by a small amount of time in addition to time delays for the XBee and electric imp. The eight data lines on the LCD (DB0-DB7) and a few other pins (E, R/W, RS, VO) are connected to the microprocessor, the A and K pins are disconnected, and the remaining pins are connected to power and ground (VCC and GND). Our LCD is connected in parallel to the ATMega328 as to save the serial connections for the serial connections of the XBee and Electric imp. To save our serial connections we connected DB0-DB1 to bits 0-1 on PortB and DB2-DB7 on bits 2-7 on PortD of the microprocessor. Sending data over two separate ports required additional timing delays so the transferred data could register and be concatenated correctly on the LCD. In addition, we have connected a potentiometer to the VO pin where the variable voltage level corresponds with the contrast level of the screen.

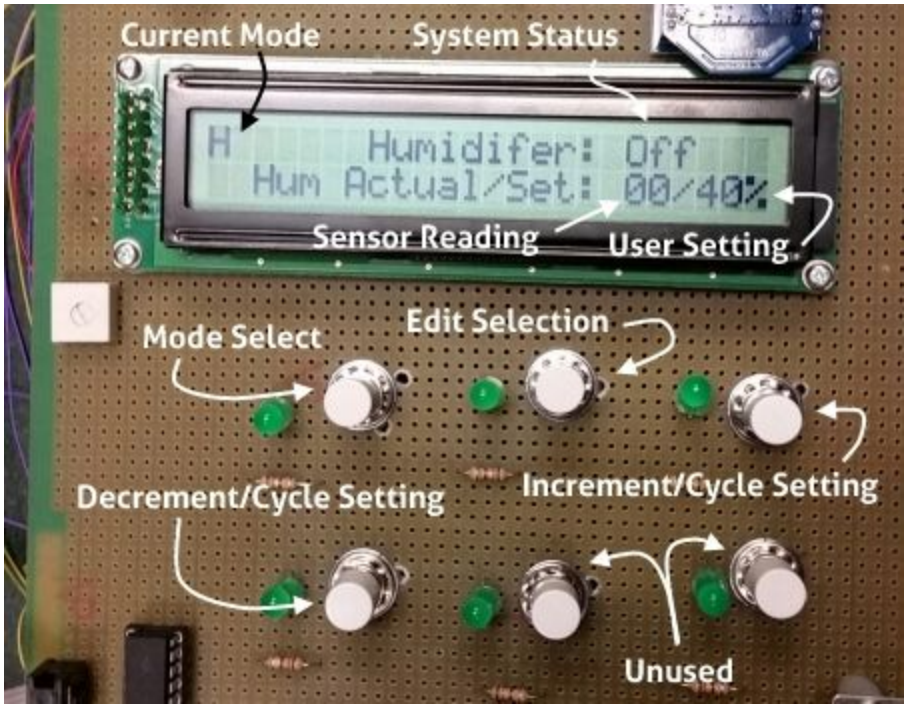
Crystalfontz CFAH2402A-TFH-JT LCD

Buttons

To input user settings, there are four buttons that the user may press to change user-defined targets for temperature, humidity, and light. This number was reduced from a total of six buttons, though ideally the use of mechanical buttons would be entirely replaced by a touch screen interface. Each button is debounced to reduce the chance of the user setting an input or setting unintentionally and to force the user to release the button for any effects to take place. On the main board, buttons are connected to LEDs to signify that a particular button is active and are mainly in place for debugging purposes. The six physical buttons (two unused) are wired to PB7 and PC1-PC5. The software section of the user interface will go into further detail on how PB7 and PC1-PC5 detects when a button is pushed and what necessary procedures are required when a particular button is pressed.

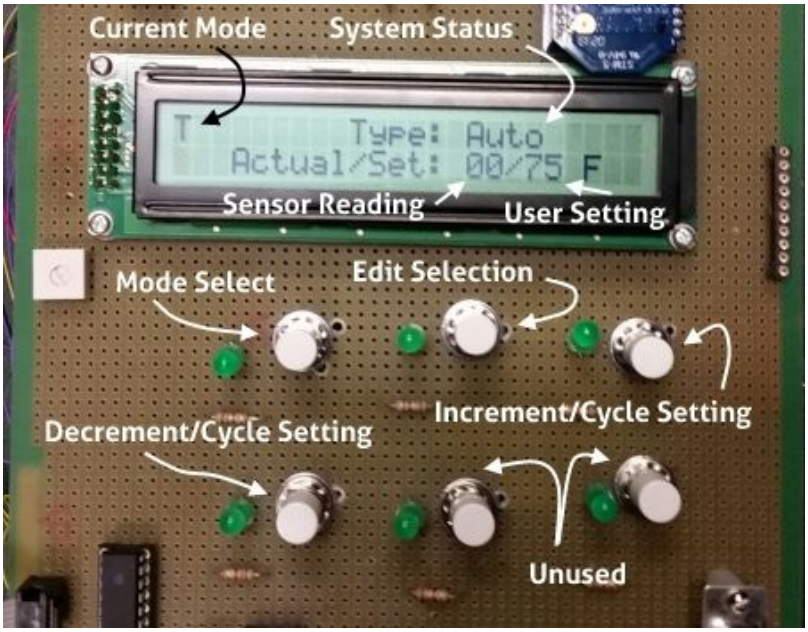
Jameco 121304 Push Button

Shown below is the main unit displaying the Humidifier Control module. Pressing the “Mode Select” in this setting will cause the display to switch to a different Control setting. Pressing the “Edit Selection” will cause the display to blink on and off indicating that you can edit your settings by clicking the either the “Increment/Cycle Setting” or “Decrement/Cycle Setting” will increment/decrement a number or change the system’s setting to On, Off, or Auto. Clicking “Mode Select” in the Edit Selection mode will toggle between information fields for editing. Finally clicking “Edit Selection” once more will cause the display to stop blinking and save the following settings in non-volatile memory.



Humidifier Control

Temperature Control module below shares the same format as the Humidifier module, with the exception that there are more options available for climate control in this module, including settings for Heat, Cold, Fan, and A/C Auto. One thing not mentioned previously in the Humidifier Control module, is the option to set the system status to “Auto”. This functionality enables the Apple Home system to automatically control appliances in order to bring the room to the user’s desired settings. Apple Home will compare the “Sensor Reading” of the actual temperature of the room with the “User Setting” of the room and will turn on the cooling unit if the desired temperature is lower than the actual temperature and vice versa, with additional details described in the software section.. This Auto selection works similarly in the lighting and humidifier control settings



Temperature Control

The Lighting Control module of the main unit is the simplest; the buttons interact with the LCD in the same manner as in the Temperature and Humidifier Control settings.

Note: the white square on the bottom left of the LCD is the potentiometer that controls the variable voltage, controlling the contrast of the LCD screen.

Lighting Control

User Interface Software

The software for the user interface is essentially a refresh-loop for the LCD display that reacts to button events. When the software is run on the board, the program starts at the top of main and checks for the validity of the EEPROM space that it will use for saving user settings. If the EEPROM is invalid, then it is likely that this is the first time that the user has started the device, and all values are initialized to default ones for the user's convenience, as shown below.

```
// Initialize default temperature to 75 F and default humidity to 40%.
if (eeprom_read_byte((uint8_t *) TEMPR_0) == 0xFF) {
    eeprom_write_byte((uint8_t *) TEMPR_0, 117); //75 in BCD
    eeprom_write_byte((uint8_t *) TEMPR_1, 0);
    eeprom_write_byte((uint8_t *) HUMID_0, 64); // 40 in BCD
    eeprom_write_byte((uint8_t *) HUMID_1, 0);
    eeprom_write_byte((uint8_t *) LIGHT_0, 0);
    eeprom_write_byte((uint8_t *) LIGHT_1, 0);
    eeprom_write_byte((uint8_t *) PACKET0, 0);
    eeprom_write_byte((uint8_t *) PACKET1, 0);
    eeprom_write_byte((uint8_t *) PACKET2, 0);
}
```

Once this procedure is complete, the program enters the main loop, where it stays until the device is turned off. The first segment of the main loop concerns user interactivity, while the second segment concerns inter-board communication.

First, the software reads the relevant data out of the EEPROM for display. By default, the data initially read on startup is temperature settings and data. For user settings modification, the program then enters an inner loop, which handles configuration of temperature, humidity, and light settings in relation to buttons and EEPROM. At the top of the inner loop, the software determines if the user wanted to edit setting for another mode; if this is true, then the loop immediately breaks and begins the execution from the outer loop in order to read more relevant data.

The core of the user interface refresh cycle is the following set of three functions.

```
switch (current) {
    case 0: tempr_config(&local_data_0, &local_data_1); break;
    case 1: humid_config(&local_data_0, &local_data_1); break;
    case 2: light_config(&local_data_0, &local_data_1); break;
    default: break;
}
strout(0x00, (unsigned char *) str_0); // Print first line of text to LCD.
```

```
strout(0x40, (unsigned char *) str_1); // Print second line of text to LCD.
// Run the internal clock - divides clock by 4
clk();
```

After determining which module the user is currently interacting with, the software will enter a subroutine to edit the respective settings information. If there is any update to the controls or sensor data via the physical user interface (buttons,) network, or remote sensor module, the changes will be reflected in the next pass of the main loop. There are three main subroutines for editing settings, each one corresponding to one mode that the user may select, and each subroutine is similar.

At the start of the function call, the function takes in the data read from EEPROM as a parameter, and breaks down the input bytes as shown below. All of the state data used in our Apple Home prototype is stored in global variables according to this format.

Data Encoding - Bit-wise Breakdown

Assume each byte is organised into eight bits, as in `[ B7 B6 B5 B4 B3 B2 B1 B0 ]`.

Temperature - 2 bytes
Byte 1: `[B7:B4]` - Temperature most significant digit (MSD) in binary coded decimal (BCD)
`[B3:B0]` - Temperature least significant digit (LSD) in BCD
Byte 2: `[B7:B6]` - Device mode in binary format
`[B5:B0]` - Unused

Humidity - 2 bytes
Byte 1: `[B7:B4]` - Humidity MSD in BCD
`[B3:B0]` - Humidity LSD in BCD
Byte 2: `[B7]` - Humidifier enable
`[B6:B0]` - Unused

Light - 2 bytes, 1 unused
Byte 1: `[B7:B6]` - Device mode
`[B5:B0]` - Unused
Byte 2: `[B7-B0]` - Unused; only read, never written

All user settings, whether they are read in from the buttons or set via electric imp, are stored within the ATMega328P EEPROM space so that the settings are not lost after loss of power or a board reset. Being that EEPROM has a limited number of writes before the memory would fail, the software logic attempts to minimise EEPROM writes as much as possible; EEPROM is typically not altered by button presses unless the user has finished with an editing phase or has moved to another screen.

```
if (!editing && changed) {
        eeprom_update_byte(addr, local_data_0);
        eeprom_update_byte(addr+1, local_data_1);
    }
}
```

Following the reading of data from EEPROM, the software determines whether the user wanted to change a setting and responds accordingly. If a setting was changed, then the software will prepare some strings to display to the LCD. Because of the slow functions required to dynamically create the relevant text, the main clock was slowed to allow these functions to complete within a clock period. Upon termination of the subroutine, the return to the inner loop will then call the `strout()` function to serially send character bytes to the LCD.

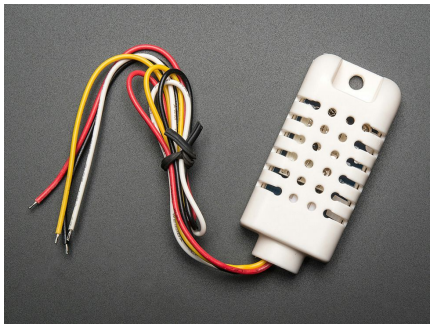
Once the basic user interactivity is completed, interboard interactions begin; following that, the inner loop completes and returns to the top.

IV. Sensors/Outputs Module

The Apple Home derives its usefulness from its output modules and remote sensors. To demonstrate utilization of stimuli, we built a standalone sensor and output module to communicate wirelessly with the main unit. This module has its own logic to enable outputs intelligently based on its current command settings, and it transmits temperature and humidity sensor data back to the main unit.

Temperature / Humidity Combined Sensors

The Apple Home prototype uses the DHT22, a joint humidity and temperature sensor. The two-in-one functionality of the DHT22 made it a primary candidate for filling the role of two separate sensors that would have made the prototype more complex and possibly more expensive. The DHT22 has an extremely simple pinout scheme, and only needs one data line to the microcontroller. The DHT22 was selected for both the efficiency of board design as well as its adaptability in only needing one microcontroller pin. The DHT22 also has a large sensing range, and is capable of detecting humidity values between 0 and 100%, and temperature between -40 to 80 degrees celsius. The Apple Home converts celsius to Fahrenheit for its American product.



DHT22 Temperature/Humidity Sensor

The Sensor/Output Unit Schematic in Appendix A shows the DHT22's pinout, with its data line connecting to the Atmel ATmega328P's PD7 pin, its power pin Vcc connecting to a 5V supply, and its GND pin connected to 0V. The DHT22 communicates serially with the microcontroller over the PD7 line. It uses a simple, waveform-based protocol documented in its datasheet. The microcontroller's software must include code that interfaces with the DHT22 according to this waveform. After the microcontroller signals the DHT22 to start sending data, the DHT22 will send 40 encoded bits, with the first two bytes containing the humidity reading, the second two bytes containing the temperature reading, and the last byte containing an XOR checksum of the first four.

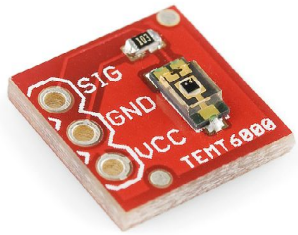
The data line from the microcontroller is driven using an internal pullup, so that when the microcontroller is ready to read the sensor values, it drives the data line low for at least a

millisecond. After pulling the line high again, the microcontroller waits the maximum 40 microsecond wait period for the DHT22 to begin its transmission. The DHT22 starts its transmission bit by pulling the data line low for 80 micro seconds, and then high for 80 microseconds. The first data bit immediately follows, consisting of a low voltage signal and a high voltage signal, with a '1' bit indicated by the high signal lasting for 70 microseconds or more, and a '0' bit indicated by the high signal lasting under 70 microseconds. After the last bit, the signal is driven low one more time, and then returns to the idle state (high voltage).

The microcontroller polls the value of the data line, and uses a counter to indicate how long it has been since a transition. The counter roughly corresponds to time; the threshold the counter needs to be compared to to determine a '1' or a '0' was determined through trial and error. The microcontroller calculates the value of bit i on the '0' level of the bit i+1, and is accommodated with an extra high-to-low transition after the 40th bit by the DHT22.

Ambient Light Sensor

The prototype also features an ambient light sensor to enable automation of light controls in the home. For this, we used the TEMT6000 Ambient Light Sensor and breakout board by Vishay Semiconductors. Similar to the combined temperature and humidity sensor, the light sensor was selected for its simple pinout and minimal load on the microcontroller. Furthermore, it is adapted for human eye responsivity, which means that it detects changes in light levels that are noticeable by the average person, and is only sensitive to the visible spectrum. It comes already connected to the breakout board so that it can be wired to a prototype board, because otherwise the chip itself is only suited to mounting onto a printed circuit board.



TEMT6000 Ambient Light Sensor with Breakout Board

The TEMT6000 is wired to the microcontroller and to power and ground as shown in the schematic in Appendix A. The GND pin is connected to ground, VCC is connected to power, and SIG is connected to the microcontroller through its PC1 pin.

Because it uses a phototransistor, this sensor acts like a transistor, sending a greater analog voltage on the SIG pin to PC1 the more ambient light is sensed. Since it sends an analog signal, we had to use our microcontroller's build-in ADC to convert this to a digital signal so

that it could be used in the general system logic. The analog signal is converted to an 8bit value, meaning 0 (no light) to 255 (most light) and read by the microcontroller.

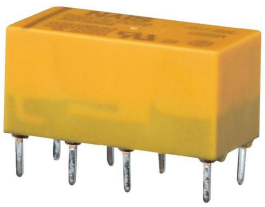
Relays

The relays are used in order to prove that our prototype could be used to power regular home fixtures as opposed to simply turning on 5V LEDs. Our board features 4 Panasonic DS2Y relays to demonstrate turning on and off the four different systems that our product is designed to control: heating unit, AC unit, humidifier, and lights. These relays are wired to both a male 6-conductor Clinch-Jones plug and to their own LED. The Clinch-Jones plug can be connected to the EE459 AC Load Simulator, which is just a set of four 12V AC light bulbs which can all be powered individually. When a signal is sent from the microcontroller to turn on one of the four systems, it goes through the relay and turns on both a light bulb on the load simulator and the LED (in case the simulator isn't plugged in).

The relays are essentially an electrically powered switch. When given the control signal from the microcontroller, the coil in the relay is energized and throws the switch to turn the pole to *on*. Because of the coil in the relay, we are able to use it to amplify the digital signal from to microcontroller to control a higher-voltage circuit. In our case, we are using these relays to control the load simulator which turns on the light bulbs.

The coils in the relays are furthermore wired through transistors so that the relays would not burn out the pins on the microcontroller. See Appendix A for the complete schematic which shows how the relays are wired to the rest of the system.

Relays were chosen as the method of conveying the output of the system because of the lack of a unified control interface for climate control systems and lighting. Vendors have many disparate interfaces, many of which are proprietary; therefore, the Apple Home would only be able to interface with some units at the exclusion of others. Many vendors do not publish the interface for their systems. The Apple Home thus uses relays to control appliances at the voltage level.



Panasonic DS2Y Relay

6-pin Clinch-Jones Plug, Male

SMART Home

By managing a user's home, the Apple Home provides energy consumption management. The core of this feature, the sensing and handling of a user's preferences, is implemented on the remote sensor board.

Because of the nature of the sensors on the board, the Apple Home keeps power consumption to a minimum. The light sensor and the temperature/humidity sensor are both low power; the light sensor is a passive circuit, and the DHT22 does not to be constantly operating. When turned on, the DHT22 can take up to two seconds for an entire transaction, during which it is actively switching voltages on the pins quickly. During the collecting period, the DHT22 can require 1 to 1.5 mA, which it needs for a two second collection period. In contrast, it only requires a maximum of 50 μA while idle.

To limit the frequency of sensing, the Apple Home employs interrupts to monitor the recency of the last temperature and humidity reading. While the delay remains highly configurable in software for testing purposes, the final version of the Apple Home can use interrupts to delay for any reasonable amount of time. Currently, the prototype measures about every eight and a half minutes.

The interrupt corresponding to the control of the DHT22 is set up in the timer_init() function of sensormain.c (see code in the appendix). The DHT22 currently uses the 16-bit timer on board the Atmel 328p, with the prescaler set at 256, and the interrupt triggering when the timer hits the value 65535. This value is somewhat arbitrary, but uses the full range of the timer. Because the other timers are not used, any timer could be used, but this timer will be used as a demonstration.

The clock runs at 9.8304 MHz, and the timer increments a counter that enables the system to read turn on the DHT22 as follows:

```
ISR(TIMER1_COMPA_vect)
{
    dht_ready_counter++;
    if (dht_ready_counter == 300) {
        dht_ready_counter = 0;
        dht_do_read = 1;
    }
}
```

The value 300 is the threshold amount of times the timer needs to interrupt the microcontroller before dht_do_read, which controls whether the DHT22 is triggered for a new reading, is set to true. Thus, the total delay comes to:

*t = 256 * 300 * 65535 / 9,830,400 = 511 seconds*

Due to the highly configurable nature of the counter, the delay in between sensor readings can be set to many different values. In future iterations of the Apple Home, this could also be a user preference. It should be noted that the system works well even with an infrequent delay on the order of tens of minutes because the temperature and humidity are both values that do not change quickly, even when climate control outputs are on.

The user also has the option to manipulate their home from abroad if any of the appliances the Apple Home controls are not needed. For example, if a user goes on vacation and realizes the air conditioner does not need to be on, the user can use the iOS application to change the mode on the air conditioning or any of the other output controls the Apple Home regulates. The modes will be explained in more depth in the net section.

The nature of the lighting control in the Apple Home is meant to give users the option to keep lights off as long as natural light is present. In the case a homeowner wants to leave a light on for a pet, for example, the solution is often to leave a light on for hours, even if it is daylight when the owner leaves the home. The Apple Home can be set to turn lights only when it detects there is no lighting present. Similar to the climate control, a user can also view the status of his or her lights after leaving the home, and can switch the lights off remotely.

Control Logic

The outputs of the Apple Home system will affect the sensor values used to drive those same outputs. For example, turning on a heater will increase the temperature value being read by the temperature sensor. This can lead to complications if not handled correctly. For example, if the controller were trying to aim at a precise temperature of 70 degrees Fahrenheit, then heating slightly past 70 degrees could cause the air conditioner to kick in. The internal control logic of the system was built with this feedback loop in mind.

In another case, turning on an output when the sensor value deviates slightly from the target value can cause frequent and extraneous turn-ons that can waste energy and put undue stress on mechanical components such as fans. For example, if the target temperature is 70 degrees, and the temperature in the room is 69 degrees, the heater will be turned on for a short amount of time and will turn off when the room reaches 70 degrees. The heater may be turned on again after a short interval if the room's temperature sinks one degree.

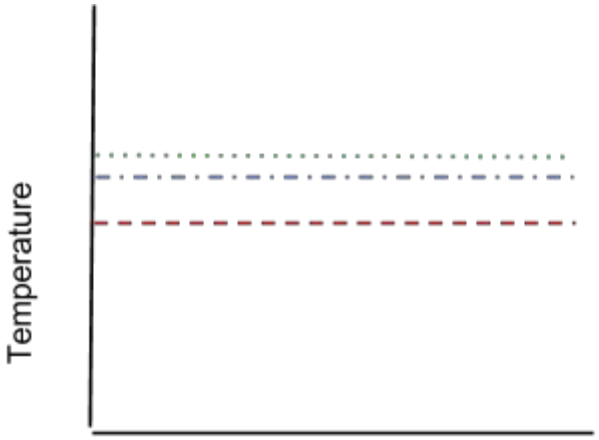


Illustration of the hysteresis used in output logic. The target temperature is shown in blue, and the red line shows the actual heater turn-on threshold a couple of degrees below that. The green line slightly above the green line shows the temeprature to which the heater will remain on until.

For this reason, the automatic control logic for the heating, cooling, and humidity all use hysteresis to mitigate the oscillatory nature of instantaneous reactions to the environment. The software defines intervals of temperature and humidity readings, centered at the target value, for which the system will not react. The system will also keep an output on until slightly past the actual target value, but less than the end of the hysteresis interval. This is a tradeoff between precision of the system's execution and power conservation. The lighting controls are a special case because turning lights on and off is an instantaneous change from the current state to the desired state.

The logic described above is encapsulated within an XXX_MODE_AUTO state in the controller's internal state machine, where the state machine is a three state Mealy machine, where the inputs affecting outputs are the sensor values and the user's settings. There are also states for the XXX_MODE_ON and XXX_MODE_OFF settings that a user can select in order to override the controller's automatic control logic. These modes are intended to allow the user to maintain a feeling of control over their home to put them at ease with the system.

Software Performance

The program loaded onto the remote unit of the Apple Home is relatively simple in its structure. The program does not need to be realtime. The climate control is relatively tolerant with respect to delays in part due to the infrequency of measurement and manipulation, and the slow response time of connected appliances. Manipulation of the lights only needs to be fast enough for the delay to not be irritating to a user. The

propagation of settings from the main board and return of data from sensors should likewise be “good enough” to satisfy a user.

Most tasks in the program do not keep the processor busy for long. The most time-intensive task, the reading of temperature and humidity sensor, is performed infrequently due to delay as described previously. As a result, the program follows a very simple, linear set of tasks, performing each as it is able.

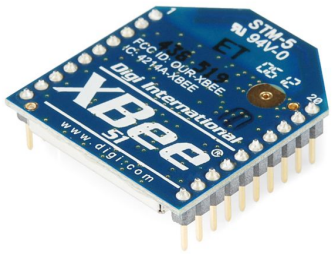
The wireless communication, described in further detail in the following section, is connectionless and thus the microcontroller does not need to wait to send or receive messages. This allows the microcontroller to assume that updates of its settings will be sent enough that it will eventually find a message waiting for it, and can move on if the XBee does not have any data received. The remote unit likewise continues sending updated sensor data back to the main board, both as a heartbeat signal and to avoid tying up the microcontroller if the main unit is not ready to talk.

As a result of these choices, the microcontroller’s software is able to rotate between tasks very effectively. It continually reads data from sensors, processes that data, reads from the XBee, sends to the XBee, and repeats.

To test both functionality and responsiveness, the engineering team manipulated the lighting reaching the sensor, sent different temperature and humidity target values from the main board, and sent different mode commands from main board. The board functioned properly and was able to respond in a satisfactory manner. Covering the light sensor, for example, causes the corresponding relay to turn on immediately.

V. Wireless Communication

To emulate the versatility of an advanced smart home system, we developed two communication systems: one system allows wireless inter-board communication to relay information between the environmental sensors/device-control module and the primary system control module; the second system enables the system control module to receive commands from the Internet.



Xbee Radio without breakout adapter

Inter-modular Communication Using Xbee Radios:

Today's commercial systems avoid wires whenever possible – for the purposes of our smart home module, which requires numerous sensors to be in different locations, wireless communication between our boards is imperative. Due to the simplicity of the data communication required between our two modules and the superfluous expense of including Wi-fi capable technology on every single remote sensor, we used Xbee Radios to emulate a direct serial communication line.

The primary requirements for wireless communication between the main unit and the remote sensor unit are ease of use and scalability. Ease of use is a hallmark of Apple products, and the final product should require a one-time setup for any user. Apple should also be able to leverage the existing Apple Home product to expand in-home presence and features by allowing users to buy and install other modules that have yet to be created and/or purchased. The user should be able to use the existing units in their house in order to extend the functionality provided to their home. Scaling up the presence of Apple Home components in a household should also not introduce problems for the user. A user should also be able to buy multiple of the same module in order to have their Apple Home manage sensor readings in multiple rooms. Also, the wireless communication should be intelligent enough to not conflict with any RF devices owned by neighbors or located elsewhere in the home.

The Xbee Series 1 radio transceiver satisfies these requirements. It allows plug-and-play communication via UART serial communication, and automatically communicates with other Xbees in the area. With a range of 300 feet, the Xbees are able to communicated over the entire footprint of most properties. They also are configured out of the box to communicate with a particular subnet, which will prevent their interference with other devices nearby. Because they implement the Zigbee mesh protocol, new Xbee-connected modules of the Apple Home can easily be added into an existing Apple Home network. The Xbees do not require Wi-fi connectivity within a user's existing network, so the user can avoid any effort to configure the system as would be required if the Apple Home used the household LAN.

Internet Integration and Remote Control via iPhone App

Modern smart control systems are typically able to connect to Ethernet or IEEE 802.11 (Wi-fi) networks due to the relatively simple, inexpensive hardware required and the numerous advantages of network-capable technologies. Because our system was designed to be an Apple Product, we implemented an iOS App to serve as a remote control panel for the Apple Home system.

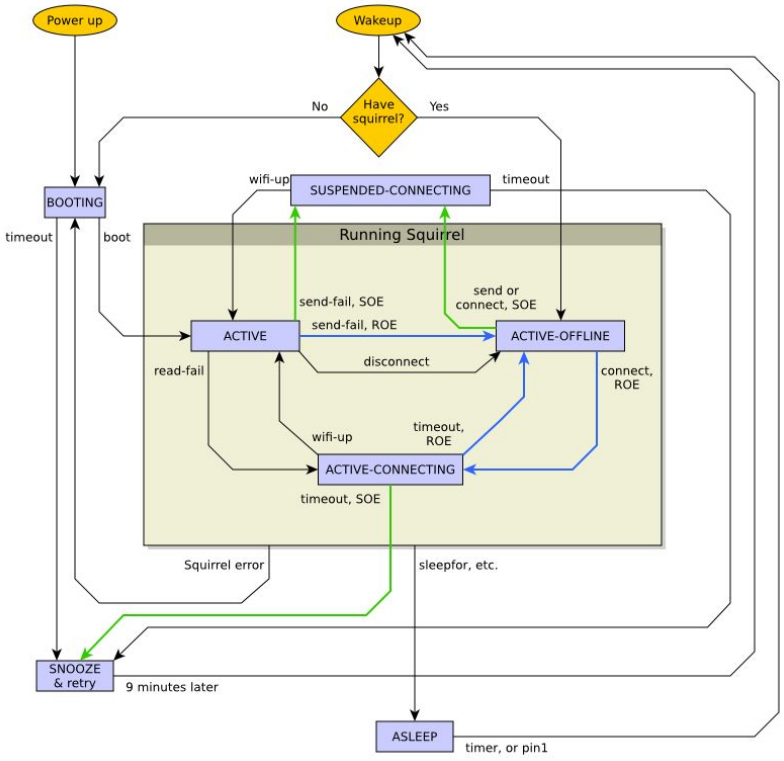
Connecting to the Internet:

To connect our primary control module to the Internet, we chose to use the Electric Imp (imp001 breakout board) to provide a gateway to the Internet via the local Wi-fi network. The Electric Imp is a modular IEEE 802.11 compatible radio transceiver and processor system that can be configured wirelessly through the Electric Imp website after initialization on a Wi-fi network. Its radio features include 802.11 b/g/n WiFi, 20MHz 11n channels, and +16.75dBm max output power (802.11b,) making the Imp suitable and robust for use on a wireless LAN. Even with its Wi-fi capabilities, the Electric Imp is a non-invasive, low power device, that is perfect for modular consumer electronics. After connecting the Electric Imp to the Internet, we are able to re-configure the Electric Imp with software written in the Squirrel programming language. Because of its Cortex microprocessor and available input and output ports, the Electric Imp is capable of running useful software; we limited our use of the Imp to essentially function as a router: relaying information between the iPhone App and the primary control module.

The Electric Imp requires a "Blink-up" sequence that uses a phototransistor circuit and a separate, Electric Imp App (designed by Electric Imp exclusively for configuring new devices,) to encode data visually from a smart-phone screen in a sequence that contains information about the Wi-fi network to be "learned" by the Electric Imp (Wi-fi details/security and account credentials.) After programming via the Blink-up procedure, the Electric Imp will automatically connect to the best available, recognized Wi-fi network on startup.

Electric Imp with Imp Circuit Board

Once connected to the Internet, the Electric Imp can be accessed online through the Electric Imp online IDE, which maps your account to the serial number of your Electric Imp, and allows you to download Squirrel code wirelessly onto your Electric Imp. The same website also provides an environment for running server-side code (in Squirrel) with functions specialized for communicating with the Electric Imp. We used this server-hosting functionality to facilitate communication with the iPhone App; i.e. the server acts as an intermediary between the Electric Imp and the remote user device. This model is beneficial, because it allows us to communicate through HTTP POST requests to the server, which are then processed and forwarded to the Electric Imp on the main system board. Furthermore, the Electric Imp connects through a secure SSL connection, allowing for more robust security implementations in the future.



Electric Imp Wi-fi State Machine (From Imp Documentation)

Integrating the Electric Imp on the System Board

The Electric Imp connects to the Internet and communicates with the online IDE without additional circuitry (aside from power and appropriate jumpers.) However, to interface the Electric Imp to the board's Atmega328P, we designed a circuit to enable communication using a serial communication protocol.

The Electric Imp, programmed in UART57 mode, uses pin 5 on the breakout board as its Tx line, and pin 7 as its Rx line. Similarly, the Atmega328P has a port designated as a UART line. However, the Electric Imp is a 3.3V device, whereas the Atmel microprocessor and the rest of the module's circuitry run on 5V. Additionally, Atmega328P only has one UART port, but the board must communicate serially with two devices: the Electric Imp and the Xbee Radio. Therefore, additional circuitry was constructed to meet these requirements.

Pin Number	uart1289	uart57	uart12
1	CTS		TX
2	RTS		RX
5		TX	
7		RX	
8	TX		
9	RX		

Electric Imp Pin Mux and Available UART Modes

To allow the two serial devices to share the Atmega328P's UART lines, we multiplexed the Xbee and Imp communication lines using a SN74LS153 4-bit wide 2-to-1 multiplexer (see Appendix A for schematic.) The circuit connects either the Xbees Rx and Tx lines or the Imp's, depending on the value of the select line, which is controlled by the Atmel using Port C. Because the Atmega328P configures the multiplexer, we use a polling mechanism in the Atmel software. For this reason, both serial devices transmit repetitive broadcast-style messages on the UART buses, to avoid lost packets.

The Xbee side of the multiplexer is simple; the serial lines connect directly to the multiplexer, and there is a 5V pull-up resistance on both lines to ensure the default state of the lines is high (inactive.) The Electric Imp, however, requires an open-collector buffer to enable both a 5V pull-up on the multiplexer side, and a 3.3V pull-up on the Electric Imp side. As an aside, the level shifter is probably not necessary for the multiplexer because TTL technology ideally

still recognizes 3.3V as high, but we shifted all bus lines to the designated voltage levels for good practice.

In summary, the Atmega328P interfaces its serial lines to a multiplexer, which the Atmega controls in order to communicate serially with either the Electric Imp or the Xbee Radio. A polling mechanism is used to prevent missing packets, since only one side (the Atmel microprocessor) controls the multiplexer's select line.

Communication Protocol

Data is meaningless without a protocol; in order to encode meaningful data, we developed a four-byte long packet to be sent from the iPhone to the control module, and then retransmitted to any relay modules.

The data to be transmitted is either a control message (4 bytes) or a sensor message (3 bytes). Included in the packet size for both messages is a header packet, to distinguish good data from noise and ensure reliable data reception.

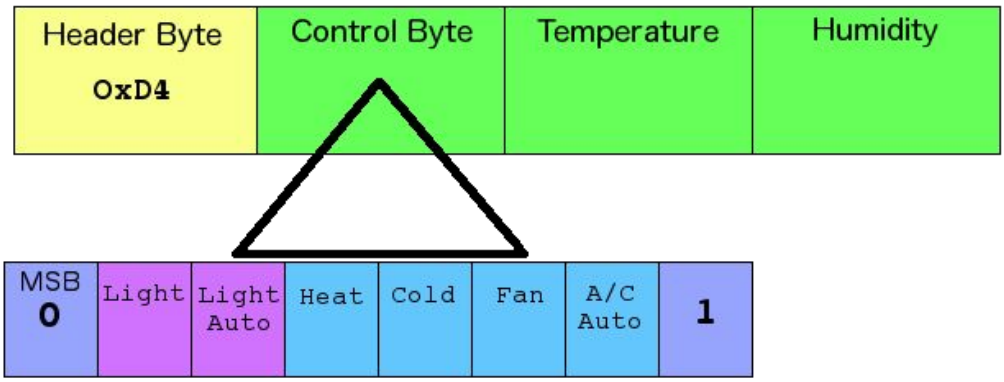
A control message is initially sent from the iPhone App, or independently from the control board via the LCD user interface. The control message describes the desired state of your home system, such that the relays can effect such changes. For example, there is one bit in the control word that corresponds to the light settings: on or off. To extract that Boolean, you simply perform the following operation in software:

```
bool lights = ((Byte0 & 0x40) != 0x00);
```

The above code is the basis for decoding all of our data; in the light example, the relevant bit is the second MSB (0x40.) A similar procedure is utilized to create a packet, given the Boolean values.

A sensor message is created by an external module and sent to the main control board and the iPhone App via Xbee Communication. The message is only three bytes for our system, with one byte representing the current temperature as measured from our sensor, and another byte representing the current humidity. This packet is used to update the interface on the LCD of the control board, and may also be displayed on the iPhone App, although we did not use this functionality.

Our message protocol takes the following format:



Control Message:

Byte 1:	0xD4	Header Packet
Byte 2:	0x--	Control Packet (see below)
Byte 3:	0x--	Temperature Packet
Byte 4:	0x--	Humidity Packet

Format of Byte 2:

Bit 1 = 0
Bit 2 = Lights (1 = On, 0 = Off)
Bit 3 = Lights Auto Mode (1 = Auto, 0 = Manual)
Bit 4 = Heater (1 = On, 0 = Off)
Bit 5 = Cooler (1 = On, 0 = Off)
Bit 6 = Fan (1 = On, 0 = Off)
Bit 7 = A/C Auto Mode (1 = Auto, 0 = Manual)
Bit 8 = 1

Format of Byte 3:

Bit 1 = 0
Bits 2-8 = Temperature to be set (when in A/C auto mode.) Valid between 60 and 90 degrees Fahrenheit.

Format of Byte 4:

Bit 1 = Humidifier (1 = On, 0 = Off)
Bits 2-8 = Humidity to be set. Valid between 0 and 100% humidity.

Sensor Message:

Byte 1:	0xE3	Header Packet
Byte 2:	0x--	Temperature Packet (see below)
Byte 3:	0x--	Humidity Packet

Format of Byte 2:

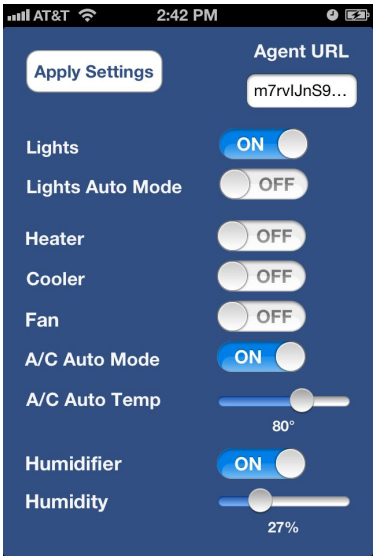
Bit 1 = 0
Bits 2-8 = Current temperature value (from DHT sensor)

Format of Byte 3:

Bit 1 = 0
Bits 2-8 = Current humidity value (from DHT sensor)

The header bytes and constant bits in these packets establish reasonable certainty that our data is valid and not corrupted. The values are arbitrarily chosen to be0xD4 and 0xE3.

Sending and Receiving Information from the Palm of Your Hand (iPhone App)



We developed an iPhone App, as is standard Apple procedure, using the Xcode development environment and the Objective-C programming language. The App is a simple remote control that communicates with the Imp through the Electric Imp server. It uses the HTTP protocol to send and receive data, because of the built-in support for the protocol on both sides of the communication.

To send a control packet from your iPhone to your Apple Home, we first create the encoded message to be sent. It takes the format described in our protocol above, and the complete software method is shown in Appendix C.

Using Booleans that represent the various control states of the Apple Home system such as Lights, Heater, Fan, Humidifier, etc, we encode the first byte of our message as a control byte. Encoding the temperature to be set produces the second byte, and the third byte contains humidity control data. We append a constant header byte of 0xd4 to maintain protocol, and have our complete control message.

```
//Create command string
      char cmdByte0 = 0x00;
      if (self.status) {
      cmdByte0 |= 0x80;
      } if (self.lights) {
      cmdByte0 |= 0x40;
      }
      ...
      char cmdByte1 = (char) self.temp;
      char cmdByte2 = (char) self.humidValue;
      ...

      NSString *cmdString1 = @"?command=";
      NSString *cmdString2 = [NSString
stringWithFormat:@"%c%c%c",cmdByte0,cmdByte1,cmdByte2];
```

The Electric Imp server is designed to receive data via HTTP POST requests that use URL-encoding. As such, you can send a message to the server by simply typing in a URL to your browser. Our server has a static URL that corresponds to this Electric Imp. We hard-coded this value onto the iPhone App to create the encoded URL.

For example, the following URL string turns off the lights, and sets the temperature to 78° Fahrenheit, using our control message protocol:

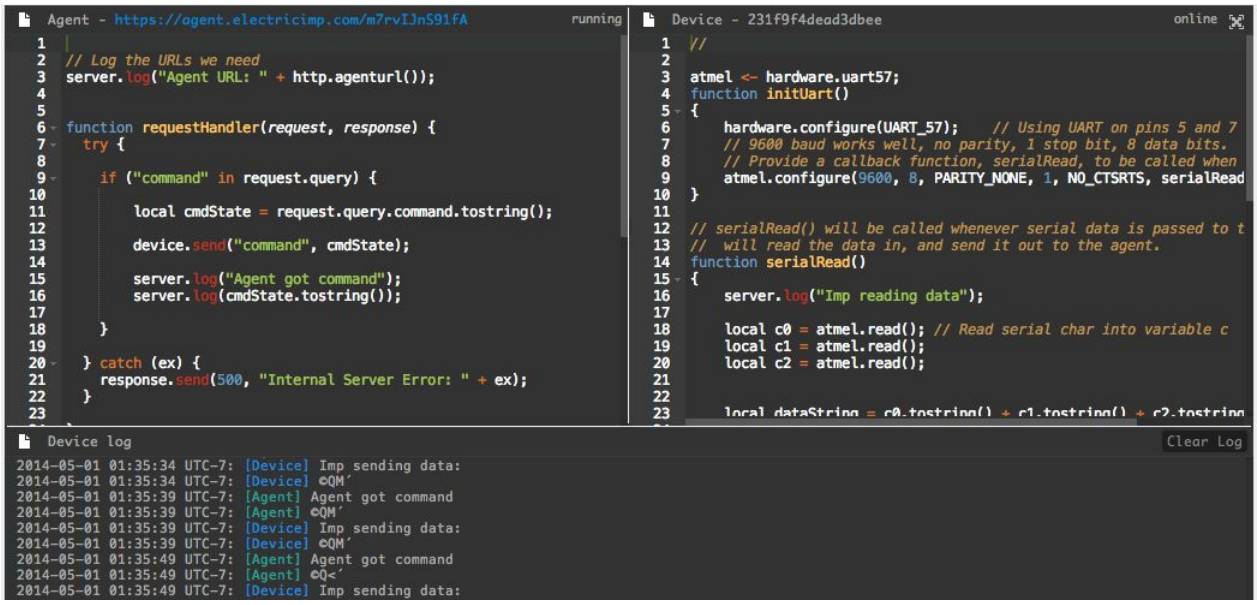
https://agent.electricimp.com/m7rvlJnS91fA?command=A<b

After encoding our control word into a URL and properly formatting the characters (URL's require that certain special characters be represented by a % followed by a 2 digit code,) we send an HTTP POST request to the new URL with no data. This is done using the NSURL library, and implementing the NSURLConnection delegates (see Appendix C for code details.) All of the necessary data is encoded in the URL.

On the server side, the "agent" (the software variable representing the server in Squirrel) responds to incoming HTTP requests to its URL.

```
if ("command" in request.query) {
      local cmdState = request.query.command.tostring();
    device.send("command", cmdState);
}
```

If the URL is formatted properly and encodes its data as “command,” then the server extracts the message as a string and sends it directly to the Electric Imp through the device.send() command, built into the Electric Imp Squirrel libraries.



Electric Imp IDE: Left - Server Code, Right - Imp Code, Bottom - Log

The Electric Imp is initialized to listen to a message labeled “command,” and forwards any such message as a string to the UART bus via the atmel.write() command. The atmel variable is previously declared to be an output UART port configured with a 9600-baud rate, 8-bit-long bytes, and no parity bit. Upon receiving a command, it outputs the corresponding data to its Tx line, port 5 on the imp001 module, and send a confirmation log to the server.

```
function sendCommand(command) {
    lastString = command.tostring();
    atmel.write(command.tostring());
    server.log("Imp sending data:");
    server.log(command.tostring());
}
```

Note: The data is converted to a string by the server and transmitted as such at the Imp. Therefore, the Imp sends an extra character—the null character—at the end of the packet, which is dropped by the Atmega328P.

At this point, the command sent from the user’s iPhone App has reached the UART bus between the Electric Imp and the microcontroller on our main system module. Next, the system board needs to read this command, update its interface, and perform the required tasks.

Managing Communication on the Main Control Unit

The Apple Home's primary control module relies on one Atmega328P to refresh the user interface (CrystalFontz LCD), respond to commands from buttons, configure the external sensor/relay modules, respond to sensor data, and respond to commands from the network. In order to maintain predictable refresh times for our physical user interface, which is the highest priority for user experience, we manage our various communication channels with a polling/TDMA mechanism that relies on short time-outs and repeated messages, instead of more efficient, but less predictable, interrupt driven software.

The user interface software (described in a previous section) performs LCD writes and EEPROM reads and writes, and takes an estimated average of 100 milliseconds per loop. This time depends on the current UI module being written to the LCD, whether the user is making changes, amongst other factors. In order to establish reliable communication without degrading interface performance, we use time-outs that take approximately the same time to read data from the Imp and the Xbee via UART.

After each user interface refresh cycle, the system microcontroller begins a polling cycle to look for a new control message from the Electric Imp. It also performs a similar polling cycle to update its sensor data from the Xbee.

```
char tempDataByte = usart_in_imp();

if (tempDataByte == 0xD4) {
        tempDataByte = usart_in_imp();
        if (tempDataByte != 0xFF) {
        io_char = tempDataByte;
        tempDataByte = usart_in_imp();
        if (tempDataByte != 0xFF) {
             temp_char = tempDataByte;
             tempDataByte = usart_in_imp();
                if (tempDataByte != 0xFF) {
                 humid_char = tempDataByte; //At this point we have presumably valid data

                 //update our local data and then forward to xbee
                 eeprom_update_byte(((uint8_t *) PACKET0), io_char);
                 eeprom_update_byte(((uint8_t *) PACKET1), (temp_char&0x7F));
                 eeprom_update_byte(((uint8_t *) PACKET2), humid_char);

                 //var_config: given packets in memory, modify global control variables
appropriately
                 var_config();
```

```
                //forward command to Xbee periodically
                counter++;
                if (counter >= 10) {
                    counter = 0;
                    usart_out_xbee(0xD4);
                    usart_out_xbee(io_char);
                    usart_out_xbee(temp_char);
                    usart_out_xbee(humid_char);
                }
            }
        }
}}
```

The above code is the part of our primary Atmega328P's main loop responsible for reading in a control message from the Electric Imp.

It performs up to four USART read operations (if it is receiving valid data); the first "if-statement" ensures that the correct packet-header is received first. Next, the three control characters are read—a return value of 0xFF signifies a time-out and cancels the read operation. 0xFF would never be encountered in a valid control byte.

If the control message is read successfully, the corresponding instance variables are updated (in the var_config function,) and state data is saved to EEPROM to ensure data protection upon a reboot. The var_config function takes the raw control byte, and extracts the bits into the relevant Boolean variables to be easily manipulated by the rest of the software, including the user interface refresh cycle. The temperature configuration section of the var_config function is shown below to demonstrate how information is extracted from the control message:

```
void var_config()
{
        int byte_bools = eeprom_read_byte((uint8_t *) PACKET0);
        int byte_tempr = eeprom_read_byte((uint8_t *) PACKET1);
        int byte_humid = eeprom_read_byte((uint8_t *) PACKET2);

        uint8_t tempr_MSD = (byte_tempr / 10) << 4; //convert to BCD for LCD
        uint8_t tempr_LSD = byte_tempr % 10;
        uint8_t tempr_BCD = tempr_MSD | tempr_LSD;

        uint8_t tempr_set = (byte_bools & 0x02) ? 0 :(byte_bools & 0x04) ? 1 : (byte_bools &
0x08) ? 2 : 3;
        tempr_set <<= 6;

        eeprom_update_byte((uint8_t *) TEMPR_0, tempr_BCD); //save BCD digits
        eeprom_update_byte((uint8_t *) TEMPR_1, tempr_set);
```

```
        // write humidity data and settings
                /* ... */
        //write light settings
                /* ... */
}
```

Next, the control word is forwarded on to the Xbee via the UART bus, by changing the select bit on the bus. We customize the basic usart_in and usart_out functions that are typically used in simple applications to include time-outs and delays. The sample usart_in function for the Electric Imp and usart_out function for the Xbee radio are shown below, complete with time-outs to prevent performance degradation:

```
#define time_const1 10000

void usart_out_xbee(char ch)
{
        PORTC |= 1 << PC0; //Select Xbee on mux
        _delay_ms(5);
        unsigned int timeOut = 0;
        while ((UCSR0A & (1 <<UDRE0)) == 0) {
                timeOut++;
                if (timeOut >= (time_const1)) {
                        return;
                }
        }
        UDR0 = ch;
        _delay_ms(5);
        PORTC &= ~(1 << PC0); //Set mux select to default (imp)
}

unsigned char usart_in_imp(void)
{
        _delay_ms(5);
        PORTC &= ~(1 << PC0); //Set select line to 0 to select Imp on UART mux
        _delay_ms(5);
        unsigned int timeOut = 0, timeOut2 = 0;
        while (!(UCSR0A & (1 << RXC0)))
        {
                timeOut++;
                if (timeOut >= (4*time_const1)) {
                        timeOut2++;
                        timeOut=0;
                }
                if (timeOut2 >= (24)) {
                        return 0xFF;
                }
```

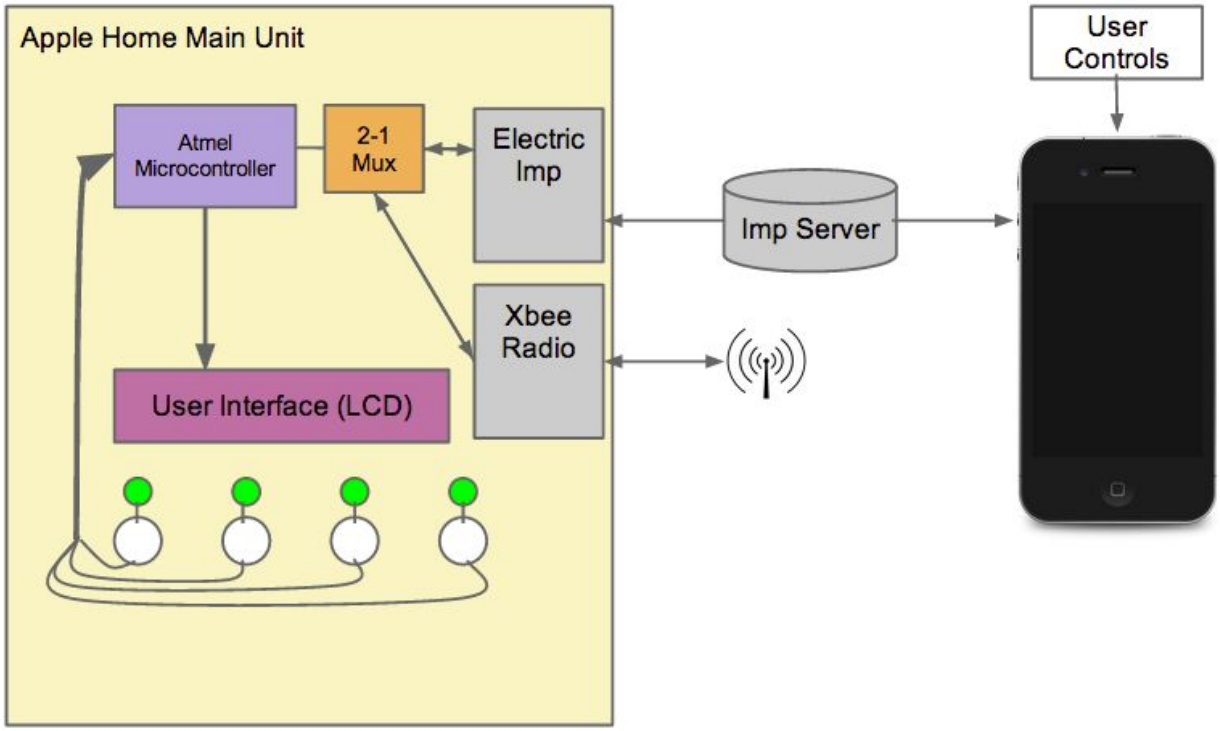
```
        }
        return UDR0;
}
```

Note: The best time-out values were determined experimentally; in the case of the UART read for the imp, we used an embedded loop to increase time-out, because the 16 bit integers on the Atmega328P overflowed before reaching a suitable constant for the time-out counter.

Delays are incorporated to satisfy hardware latency of the multiplexer and/or serial devices, which operate at a relatively slow baud rate compared to the microprocessor's clock. After the Electric Imp is polled for data, or a time-out, the same procedure is done for the Xbee, except using the sensor message format (see above.) The polling mechanism works because both the Xbee and the Electric Imp repeat the last new message sent. That way, if the microprocessor misses an Imp command, it will attempt to read the latest data every 200 ms or so, which has proven to be reliable. We used rates of 4 packets/second from both the Xbee and the Electric Imp for the prototype demonstration.

In the command packet polling cycle, note that the packet is forwarded every 10 times it sees the message. The sensor and relay modules would likely be battery driven in a real Apple product, and so we limit the amount of messages sent to our remote systems for decreased power consumption. This allows for increased sleep-time, and hence the power life cycle, which is important in remote, battery-powered subsystems.

The iPhone App is also capable of receiving commands from the Electric Imp. The iPhone can read either a control word, if the user wanted to see the current settings (this would be done on App bootup,) or a sensor message, which shows the current temperature and humidity detected by the remote module. The mechanics of this process are exceptionally similar to the procedure that transmits a packet from the iPhone to the Electric Imp illustrated above, so it will not be described in detail here (see Appendix C for source code.) However, in a finished Apple Home Product, this functionality would be essential to integrating the user with his or her home from anywhere in the world.



Wireless Communication System Diagram

VI. Conclusion

Creating a Real-World Product

Our Apple Home prototype demonstrates the core functionalities of a modern smart home system. However, there are significant differences between the prototype and what you could expect on a commercial Apple Home product.

Some of the shortcomings of the prototype are obvious and expected, such as packaging—a real Apple product would be carefully enclosed in a symmetric case, whereas we have raw circuits overflowing a board. Similarly, the commercial product would likely use a more advanced microprocessor to allow the Apple Home to run the iOS operating system, and it would have an LCD touch screen to complement the touch-based OS. Aside from the various other design changes, the Apple Home would of course have hardware to do what we simply emulated with electronic relays—actually controlling home appliances.

Instead of one sensor and relay board, the software on this project is designed to use multiple remote sensor and/or relay control modules, which Apple would certainly provide in the finished product. These remote modules offer plug-and-play functionality, allowing a homeowner to tailor her Apple Home specifically to her household. Modular sensors and relays could be used to control anything, not just limited to the major home appliances we chose for our prototype demonstration. Due to the extensive control Apple Home has of a user's home, there would also be an emphasis on security, which was more or less ignored for the purposes of our design.

All in all, a marketable Apple product would look significantly different than our prototype, for obvious reasons. Such a product requires a cutting edge manufacturing facility, the best parts on the market, years of planning and perfecting, and is, frankly, beyond the scope of this course. Regardless, our Apple Home prototype demonstrates all the core functionalities that you would find on the real deal.

Challenges and Takeaways

As with any large project, our team encountered difficulties and learned important lessons as we constructed the Apple Home prototype.

Integrating the Electric Imp hardware onto the system board proved to be quite a challenge. There were numerous setbacks, most of which involved bad wiring. Eventually, with the Oscilloscope handy and a little help from Professor Weber, we were able to diagnose a wiring problem that had been preventing Electric Imp UART data from reaching the main unit's Atmega328P.

One of our most significant design challenges was communication on the main board; we needed to communicate wirelessly with the sensor board via the Xbee radio, and simultaneously to the network through the Electric Imp. The catch: we only have one UART bus on the Atmega328P. Eventually, we came up with our scheme for multiplexing the signals, and using a polling procedure to receive new packets, which worked perfectly in our demonstration.

The DHT sensor proved to be another challenge. As described previously, the DHT22 has a proprietary communication protocol over one data line that essentially requires the microcontroller's software to implement one side of a timing diagram. Because visibility of the operation of the sensor is limited, it was difficult to understand where read operations were malfunctioning. To solve this, we kept the MAX232 UART serial model connected to the sensor board in addition to the XBee to help with debugging. At first, we found the read operations would time out completely, which was because of a bug in which we read from a PORT rather than a PIN. After pinpointing that issue, it appeared that the microcontroller was essentially missing a bit; this resulted from a miscounting of the number of periods of the waveform required for the entire 40 bit transmission. Finally, a parameter being used as the time threshold to signify a "1", originally based off of a sample Arduino program, needed to be set to nearly three times its original value; this was determined experimentally after the checksum continuously failed. While building a functional driver for the DHT22 proved challenging, it was provided valuable experience with timing issues as well as in-depth reading of product data sheets.

Designing and building our Apple Home prototype encompassed circuits, networking, power systems, software, and digital logic—the project provided a valuable learning experience across a multitude of technical fields. However, some of the lessons stuck more than others. One particular peculiarity is the limited capability of your microcontroller. When programming in an ordinary IDE, it is difficult to think that refreshing a 24x2 LCD while maintaining two communication channels would prove to be a significant hardware challenge for your processor. Timing became a very important part of our software as we adjusted to satisfy the latencies of slower devices on our board. In the end, we managed to juggle a host of processing tasks on a cheap Atmel microcontroller without sacrificing performance.

An early challenge in the development of our wireless communication system was dirty packets and noise. We knew we were receiving the correct packets across the Xbee channel some of the time, but in other cases, the message bytes were transmitted in the wrong order, or we received completely corrupt data. The solution was as simple as developing a protocol—adding a header byte and error detection prevented out-of-order transmission or unwelcome packets from corrupting our boards' state variables.

As the prototype demonstration date approached, we were still making plenty of last minute changes. We cycled through plenty of IC parts until we finally managed to create the shared

communication channel with a multiplexer circuit. It's like the saying: "There's an App for that," except the same holds true with IC parts!

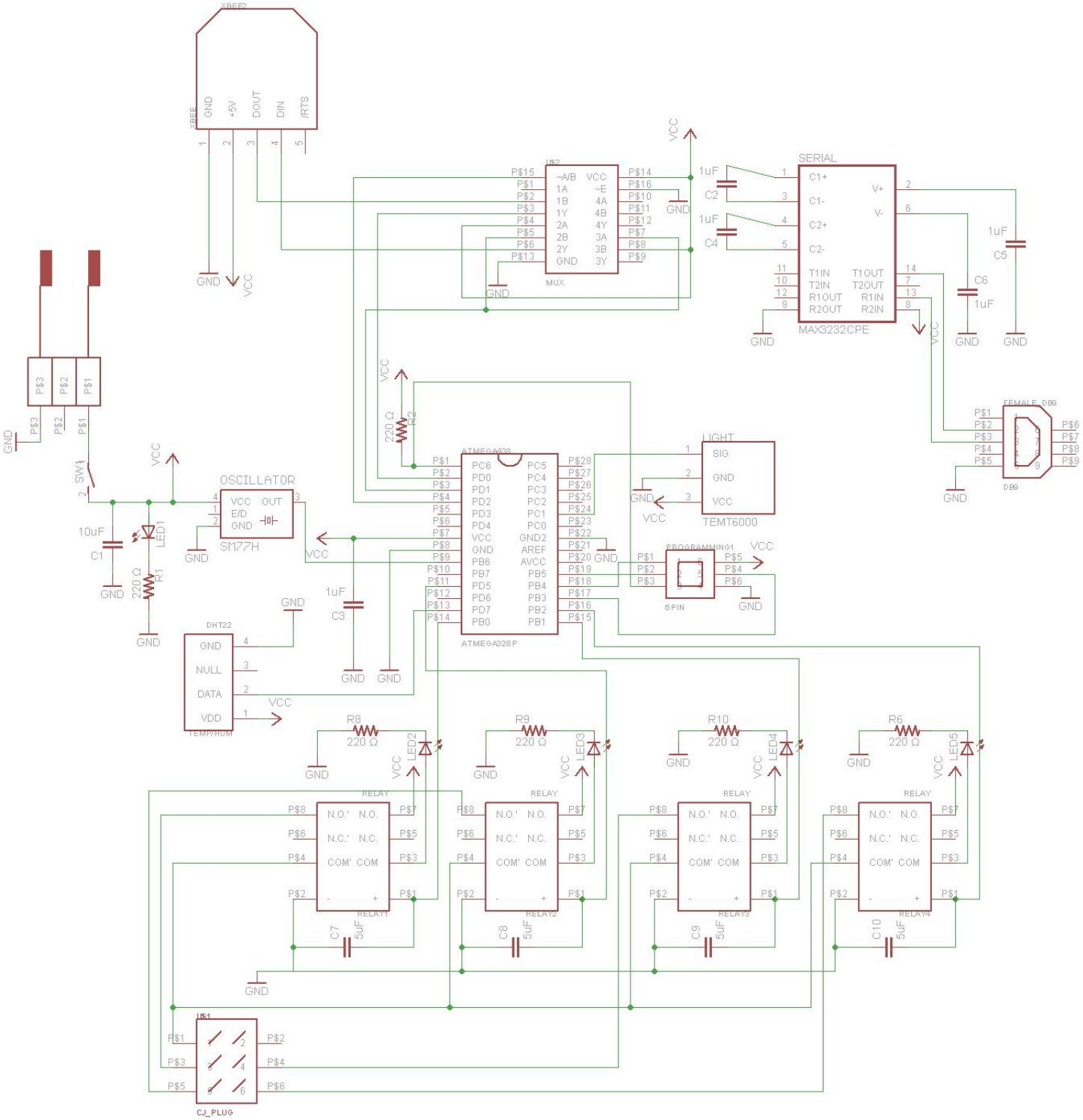
The UART components included in the projects, including the Electric Imp and the Xbees, proved to be invaluable in reducing the complexity of communication between components. Because the connections are relatively simple and the wireless communication is abstracted away, it was relatively easy to get test programs working with these components. We expected the communication between these chips and the microcontroller to be complex, but the UART interface was implemented in all of the involved chips made the low-level communication between local ICs the easiest part of the wireless communication. Contrasted to the aforementioned trouble communicating via a proprietary serial protocol with the DHT22 sensor, these parts highlighted how valuable protocol standards can be. UART allowed the team to mainly focus on the high level communication protocols between the three main Apple Home modules.

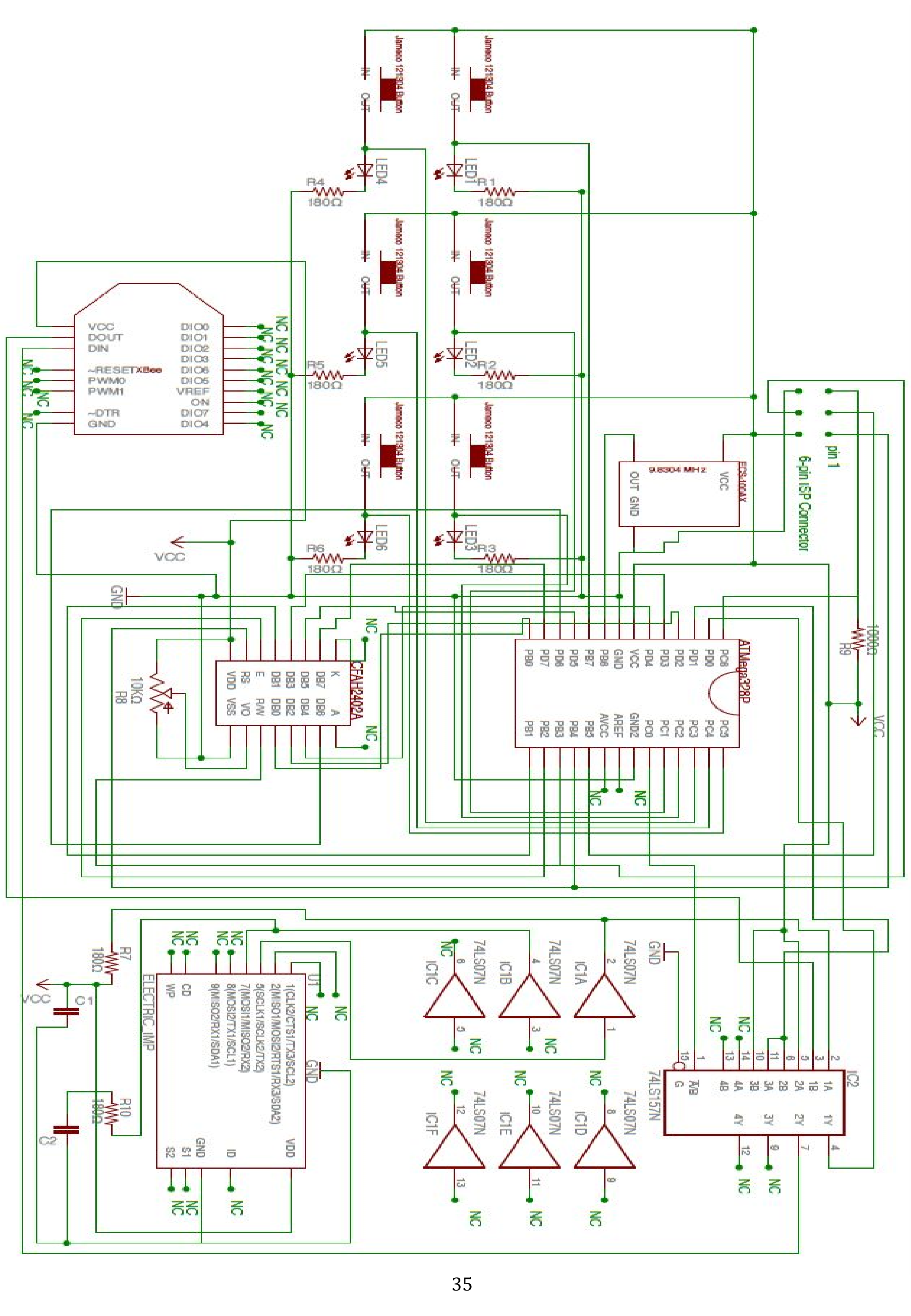
Summary

Apple Home is the smart home system of the future. Our prototype demonstrates the important functionalities that should come standard in any modern smart home system. Using a 10 week time-line and standard parts, our Apple Home system provides a physical user interface, reads and responds to sensor data, and enables remote control from the convenience of your iPhone. While not without its technical challenges or aesthetic issues, the Apple Home prototype exemplifies seamless remote control of any connected appliance. Maybe Team 5 will even qualify for royalties if Apple ends up making a smart home system!

Appendix A: Schematics

Sensors/Outputs Unit Schematic




Jameco 121934 Button
IN
OUT
LED1
LED2
LED3
LED4
LED5
LED6
R1
R2
R3
R4
R5
R6
180Ω
XBee
VCC
DOUT
DIN
~RESET
PWM0
PWM1
~DTR
GND
DIO0
DIO1
DIO2
DIO3
DIO6
DIO5
VREF
ON
DIO7
DIO4
NC
VCC
GND
CFAH2402A
R8
10KΩ
ATMega328P
9.8304 MHz
VCC
OUT GND
6-pin ISP Connector
pin 1
R9
1000Ω
74LS07N
IC1A
IC1B
IC1C
IC1D
IC1E
IC1F
74LS157N
IC2
ELECTRIC IMP
U1
R7
R10
C1
C2

Appendix B: Summary of Important Components

Main Board
Crystalfontz CFAH2402A-TFH-JT parallel LCD
Electric Imp imp001 module
Imp circuit board
Atmel ATSHA204 authentication device
Jameco 121304 Push button (6)

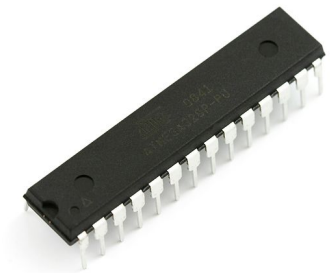
Remote Sensor/Outputs Board
Vishay Semiconductors TEMT600 ambient light sensor and breakout board
Aosong Electronics AM2302 temperature and humidity sensor
Panasonic DS2Y relays (4)

Common Components
Atmel ATmega328p microcontroller (2)
TI MAX232N IC driver/ receiver (2)
Xbee series 1 with breakout board (2)
Jameco 21936 Toggle switch (2)
9.8304MHz oscillator (2)
Jameco 21936 Toggle switch (2)
[Resistors, capacitors, transistors, LEDs, etc.]

--

Microcontroller

The Atmel ATmega328P is a cheap, highly functional microcontroller. It provides diverse functionality out of the box beyond just the capability of running custom C and assembly code, including on-board ADC circuits and timers. It also support UART serial communication and GPIO.



Atmel ATmega238P Microcontroller

Appendix C: Source Code

Main Unit Microcontroller Programming

Program (.text + .data + .bootloader): 5806 bytes (*17.7% Full*)
Data (.data + .bss + .noinit): 297 bytes (*14.6% Full*)

Sensors/Outputs Module Microcontroller Programming

Program (.text + .data + .bootloader): 4092 bytes (*12.5% Full*)
Data (.data + .bss + .noinit): 114 bytes (*5.6% Full*)

Appendix D: Cost Analysis

Component	Quantity	Cost Per Unit	Total Cost
CFAH2402A-TFH-JT LCD Screen	1	$15.95	$15.95
AM2302 Temp/Humidity Sensor	1	$15.00	$15.00
Ambient light sensor and break-out board	1	$1.50	$1.50
ATSHA204 authentication break-out board	1	$2.95	$2.95
Electric IMP 001	1	$25.00	$25.00
Electric IMP circuit board	1	$12.95	$12.95
Xbee module and SIP board	2	$22.00	$44.00
9.8304MHz oscillator	2	$0.63	$1.26
Green 5mm LED	11	$0.09	$0.99
Jameco 121304 push button	4	$0.75	$3.00
Panasonic DS2Y Relay	4	$1.95	$7.80
74LS153 2-to-1 mux	2	$0.75	$1.50
Transistor	4	$0.65	$2.60
.1 uF capacitor	2	$0.55	$1.10
5 uF capacitor	8	$0.69	$5.52
10 uF capacitor	2	$0.79	$1.58
100 uF capacitor	2	$0.99	$1.98
180 Ω resistor	8	$0.06	$0.48
220 Ω resistor	6	$0.06	$0.36
74LS07 open-collector buffer	1	$0.50	$0.50
LM3940 regulator	1	$0.50	$0.50
ATmega238P Microcontroller	2	$3.24	$6.48
Total Cost			**$153.00**

Appendix E: Contributions

Contribution categories are chosen to be about equally weighted:

Task	Ethan	Leo	Rob	Jessie	David	Adi
LCD Board standalone Hardware	50%	-	-	-	50%	-
Sensor Board Hardware	-	-	30%	-	-	70%
Electric Imp Hardware + Server Integration	-	100%	-	-	-	-
Main Board Hardware Integration	-	45%	45%	-	-	10%
Xbee Communications Hardware	-	-	100%	-	-	-
Xbee Communications Software	-	60%	30%	-	-	10%
Sensor Board Software	-	-	50%	-	-	50%
LCD Board standalone Software	-	-	-	-	100%	-
iPhone App + Squirrel Software	-	100%	-	-	-	-
Main Unit Software Integration	-	60%	40%	-	-	-
Presentations	-	33%	33%	-	-	33%
Final Report	12%	27%	17%	-	17%	27%
Miscellaneous Contributions (Design/Leadership/Availability)	2%	30%	30%	1%	7%	30%

Signatures

Ethan Chan

Leo Linsky

Rob Puncel

Jessie Qiu

David Tan

Adi Yogev