THE HIVE

EE459 Team 12 Spring 2022

Matt Liberson || Anna Pendleton || Joshua Solomon

INTRODUCTION

Looking for a simple way to help the environment? Wanting to find an easy side hustle and make some extra cash? Hoping to pick up a new hobby? Look no further than beekeeping with our amazing new product: The Hive. The Hive is a smart beehive which automatically adjusts and tracks the hive conditions to improve beehive health and increase honey production. It helps eliminate most of the maintenance and monitoring of the hive, helping the user create a healthy, happy, and stress-free hive.

There are many advantages for humans to pick up beekeeping. First, beekeeping is great for the environment. Bees are pollinators who transfer pollen from flower to flower and thus are necessary for agriculture and food production. Bees are currently in critical condition and without them, we face major disturbances to our food supply chains and global food production. Beekeeping ensures that the bees are taken care of and can continue to pollinate their ecosystems.

Second, beekeeping is great for the economy. Today, roughly 500 million pounds of honey are produced and consumed annually across the globe which makes the honey and beeswax industry worth hundreds of millions of dollars. However, that is not all bees can do. It is estimated that bees pollinate roughly 19 billion dollars worth of crops annually. While commercial beekeeping is vital to the world economy, there is also a strong demand and interest in small-scale, individual beekeeping, producing honey and bee-related sustainable products.

While beekeeping is a great idea for these and many other reasons, we have identified a problem: Many people are interested in beekeeping, however are not willing to set aside the required amount of time and energy to take care of their hives. Because of this, it is no surprise that vast sums of money have been spent on creating better beekeeping tools to address this type of consumer. Modern technology has allowed us to maintain and monitor hives better and easier than ever before. For example, apiarists, people who have beehives, have developed innovative ways to maximize honey and beeswax production while minimizing the danger to humans. Other technologies have been developed including various hive morphologies and protective devices like beekeeping suits and smokers. Academics have also researched and published a plethora of information on bee colonies and their optimal conditions. Using this technology, commercial farms and amateurs alike will be able to increase output, reduce accidents, and keep up in an industry poised to balloon in the coming decades. We intend to leverage that information to make the next great advance in apiary technology.

PRODUCT

Our product “The Hive” autonomously gathers and processes data to optimize the beehive health and the beehive honey production. Bees operate best at temperatures around 95 degrees, 50-60% humidity, and low to medium UV indices. Under such conditions, bees are expected to produce the most amount of honey. Consequently, our product senses UV Light, Temperature, and Humidity and detects honey production using a weight sensor. Additionally, it has fans and heaters to actuate according to the needs of the hive. This product provides the user with warnings regarding the hive, if detected, and prompts the user to adjust beehive settings, specifically temperature. The hive can store data regarding the best conditions for the hive to automatically adjust its settings accordingly. Our hive design was modeled on the popular modular box-shaped Langstroth hive design, in which different beehives are stacked on top of each other, as shown in **Figure 1**. Inside each modular stack, there is one outer shell of the hive, an inner shell of the hive within the outer shell, and then individual layers of honeycomb inside the inner shell. Our final product sought to replicate this design, shown in **Figure 2**. The outer shell, labeled **(1)**, will hold our user interface components such as rotary encoder, button, switch, and LCD screen externally, which help the user alter the operating conditions of the hive by scrolling with the rotary encoder, selecting with the button, and viewing with the LCD display. A UV light sensor and temperature and humidity sensor are placed externally on the outer shell, as well. The inner shell, labeled **(2)**, will house the individual layers of honeycombs, labeled **(3)**. In between the inner shell and the outer shell, we placed all of our electronics, including our ATMEGA328p, EEPROM, fans, heaters, another temperature and humidity sensor, and more. The open side of the hive will also feature a transparent plastic window pane allowing the beekeeper to view the colony. The final product of The Hive is shown below, following the same design as above, in **Figure 3**. On the picture on the left, (1) is showing the outer shell, (2) is showing the inner shell which houses the honeycomb layers, and (3) is showing the electronics. On the right, the external side of the outer shell is showing with the UI and sensors installed.

Figure 1: Langstroth Hives

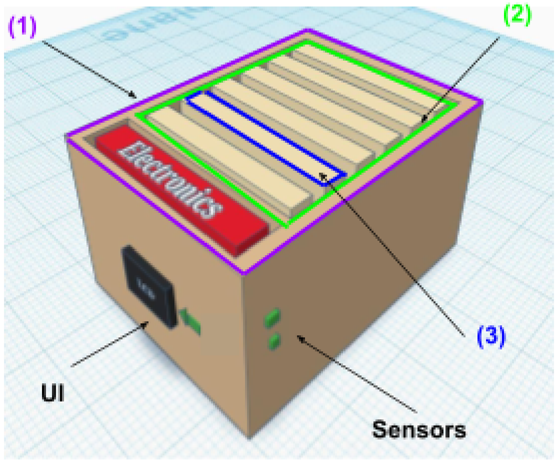

Figure 2: Hive Design

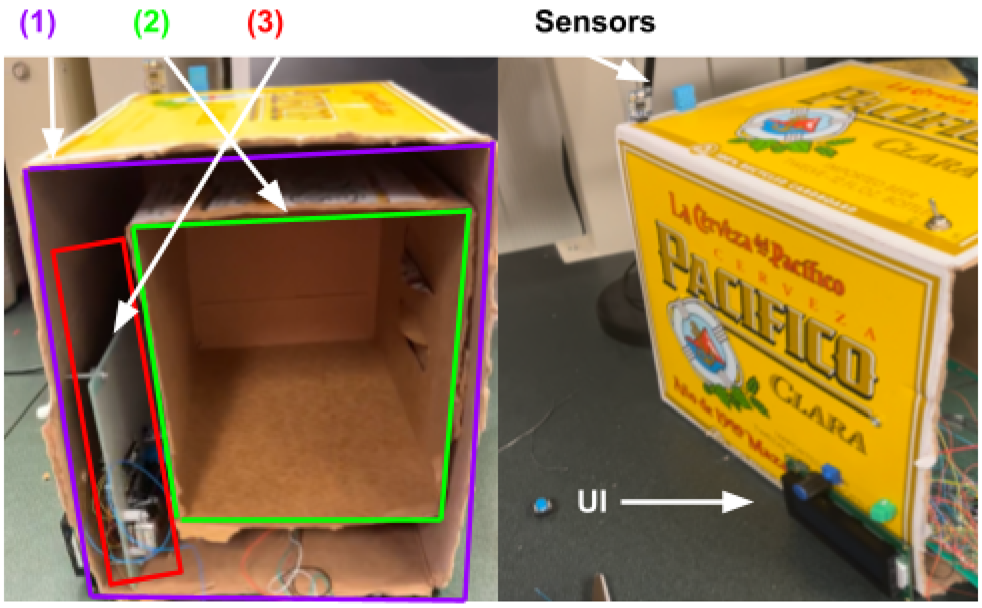

Figure 3: Final Product

HARDWARE

Schematics

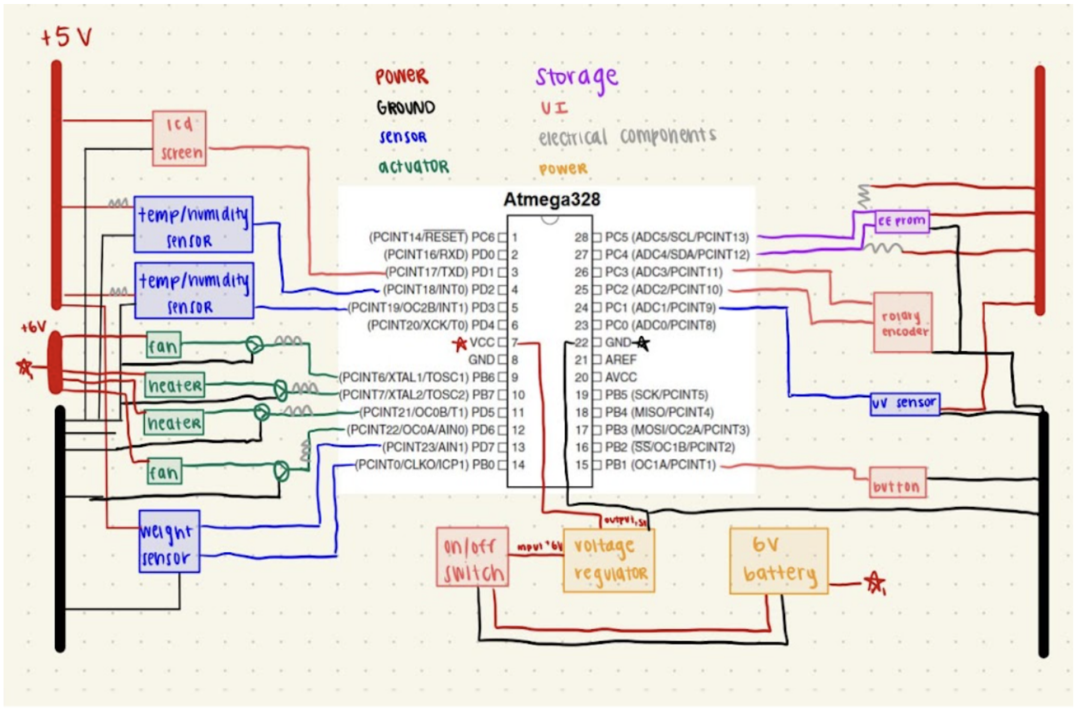

Figure 4: Pinout Schematic

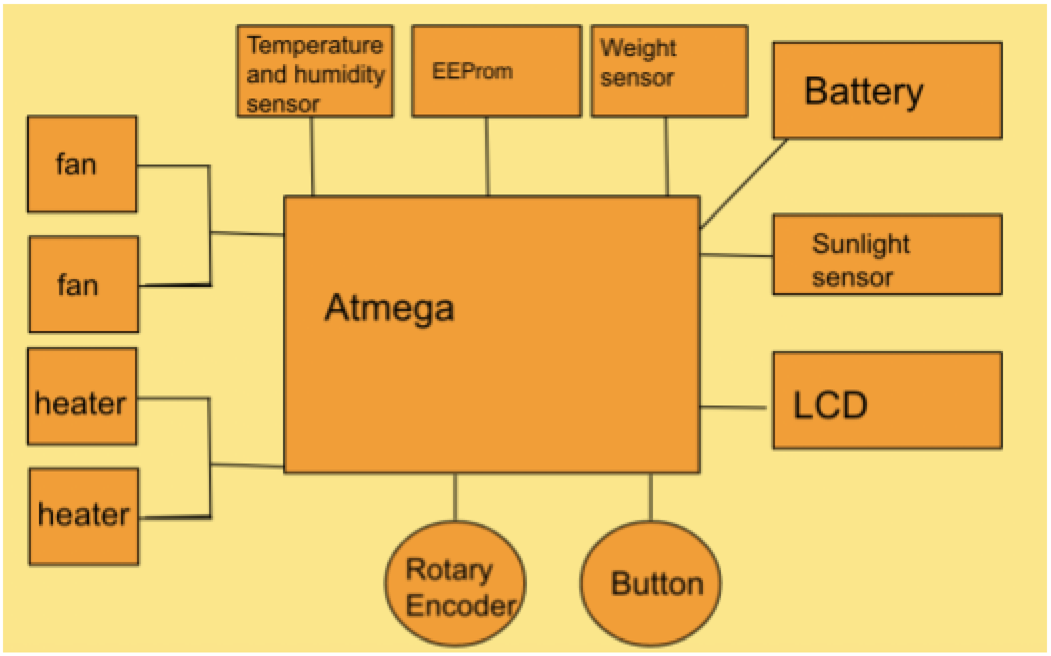

Figure 5: Block Diagram

Components

The Hive required many different electrical components. They are divided into categories of sensors, actuators, user interface, power, and other electronics. Each component is referenced in the pinout schematic shown in **Figure 4** above and our block diagram of the major components is shown in **Figure 5** above. The Parts List with all of the components, quantity, part number, and cost can be found in **Appendix B: Parts List** and **Appendix C: Cost.**

Both of the figures above represent the overall design for The Hive. The Atmega is powered using 4 AAA batteries in a 6V battery house which is connected to an on/off switch which connects to a 5V voltage regulator which connects to the Atmega, providing it the necessary 5V. For the User Interface, the button was connected to one Digital I/O pin and the rotary encoder was connected to 2 Analog pins and ground. The LCD Display utilized TTL communication protocol and needed to be connected to the TX pin of the Atmega and power and ground. For the temperature and humidity sensors, each is connected to one Digital I/O pin with pull up resistors and the weight sensor had to be connected to 2 Digital I/O pins plus power and ground. The UV light sensor was connected to one Analog pin and power and ground. The fans and heaters each were connected to one Digital I/O pin with transistors for current amplification. The EEprom is connected to the SDA and SCL pins with pull up resistors and uses I2C communication protocol. How each of these components interfaced with the software is described in more depth later in the report.

Voltage and Current Requirements

Most of the components operated well at the current and voltage supplied by the Atmega's digital I/O pins. Unfortunately, the fans and heaters required more input current than the Atmega could supply. The 5V fans required 200 mA to function properly whereas the heaters worked with anywhere between 200 mA and 1 A. To satisfy these requirements, we incorporated a 6V battery pack into the design. Unfortunately, this caused a new issue: Most of the components need 5V to operate properly –including the microcontroller. So, we also added a voltage regulator to reduce the 6V battery supply to 5V where necessary. This still left the problem of controlling the fans and heaters though.

We could not connect the fans and heaters directly to the 6V power supply, because that would leave them on indefinitely. Not only would that be an enormous power drain, it would also prohibit us from controlling the temperature of the hive. To gain control of these components, we introduced NPN transistors into our design as shown in the *Block Diagram* section above. Each transistor was controlled by an Atmega digital I/O pin connected through a resistor to its base. The 6V power supply was run through either the fan or the heater and connected to the collector of each transistor as shown to the right in **Figure 6**. The emitter of each transistor was then connected to ground. This allowed us to control each fan or heater individually while also supplying it with an appropriate current. While this setup worked when a laboratory power supply was used in place of the 6V battery pack, we unfortunately had large voltage drops when the 6V battery pack was substituted back in meaning the fans and heaters struggled to operate effectively at some times. The reasons for this have not been entirely deduced, but we believe the problem could be mitigated by introducing a larger power supply such as a 9V battery or using two separate power supplies – one for the fans and heaters and one for the other components.

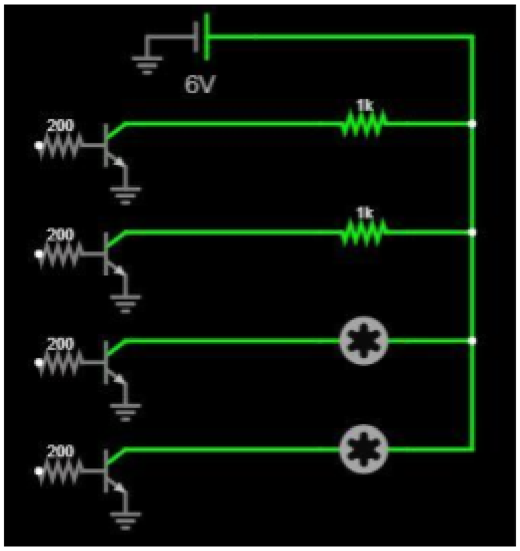

Figure 6: Heater/Fan Setup

SOFTWARE

Some of the code included in our codebase was derived from Introduction to Embedded System (EE 109) at USC and/or from Professor Allan Weber. Wherever code was derived, it is marked accordingly with a comment.

Structure

We compartmentalized our codebase to more easily delegate tasks and make functionality modular. Our main user interface file *top.c* contained the state machine and functionality to interact with the user. It utilized ten other C files to fetch data and – in the case of *optimize.c* – do work on that data. Each of these "helper" files was stored in its own directory, and some even included their own "helper" files as is the case *lcd.c* and *serial.c*. Because each of these files was separated from the user interface, all of the data fetching on the user interface side of the codebase is interacting with black boxes. So, if we wanted to use a different (hopefully better) weight sensor in the future, we would simply need to change the implementation of the *get_weight_sample* function in *HX711.c* rather than changing the top design and the underlying helper files.

Beyond the directory structure, the *top.c* file is the most complex file and contains its own structure as well. Our top file begins by initializing each of the underlying "helper" files such as *HX711.c* and *uv.c*. Then, it runs an infinite while loop to simulate an operating system. Within the while loop, the top file continuously updates the state, interacts with the user, and updates output function logic to maintain the environment within the beehive. All functionality is highly compartmentalized and modularized. For example, the scrolling functionality is the same for each of the various screens regardless of what text actually appears on the screen or how many lines of text there are. This modularity makes future updates to the project easier to implement and more reliable.

Finite State Machine

As shown below in **Figure 7**, we used a finite state machine (FSM) with five states. The FSM is designed as a tree graph to reduce the number of buttons necessary to control the product and to simplify the interface for the user. As a result, one button can be used to control all state transitions, and no transition takes the user more than one step away from the HOME state. Each state represents a unique feature of our product and a similarly unique screen on the LCD.

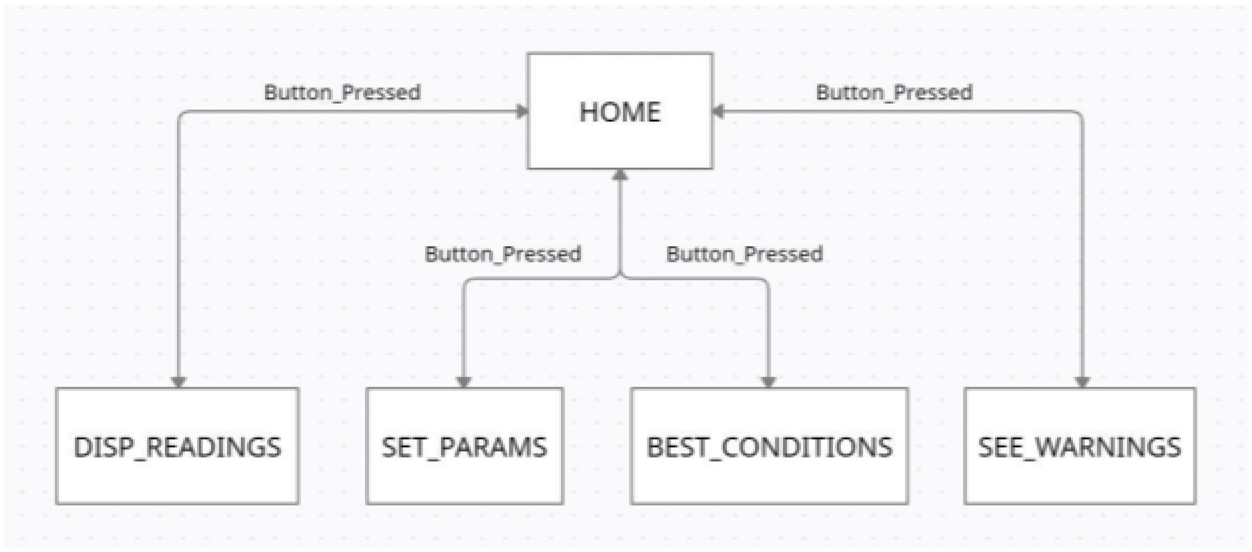

Figure 7: Finite State Machine Design

The HOME state is where the user can see what functionality is available and select which functionality they would like to use. This is also what the design is initialized to on restart. From the HOME state, the user can press the button to go to one of the other states within the FSM.

The DISP_READINGS state displays the most recent sensor readings in a scrollable list for the user. The readings will not update in real time, so the user must exit and reenter this state to see updated values. To return the HOME state, the user can press the button.

The SET_PARAMS state allows the user to adjust controllable parameters. The only value that we are currently able to control is temperature, so that is the only value the user can adjust in this iteration of the product. This state is unique in that it contains two separate displays: the first screen allows the user to select which parameter they would like to update while the second allows the user to update that value with the rotary encoder. When they are happy with the value, they can press the button to set it. Unfortunately, this means that *some* value must be updated when the user enters the SET_PARAMS state, even if that means setting the temperature to its default value.

The BEST_CONDITIONS state displays the optimal conditions for the hive. The algorithm used to find these conditions is described in the Best Conditions Algorithm section.

Finally, the SEE_WARNINGS state shows the user any warnings that might have accumulated while they were away. These warnings might include excessively high or low temperatures or rapid changes in weight. As expected, to go back to the HOME state the user must press the button.

Best Conditions Algorithm

The best conditions algorithm determines which hive conditions result in the highest honey production. Once the algorithm completes, the result will be displayed to the user through the LCD. These results will include the change in weight and all associated conditions that likely produced this change. Since maximum hive control is given to the user, these conditions are not automatically sought by the hive actuators. Rather the optimal conditions are left to the user to request through the SET_PARAMS state.

The algorithm uses the 100 most recent samples that have been stored in an off-chip EEProm to calculate the optimal conditions.. These samples are stored into EEProm every hour. Since the algorithm requires 100 samples, if the user selects the best conditions option before 100 EEProm samples have been stored, then the LCD will tell the user they need to wait.

Optimizing functions that make inferences from past data often require many past samples. While the EEProm can store a large number of samples, the Atmega itself only has 2KB of RAM. Given that each sample requires 40B, the Atmega could at most store 50 samples

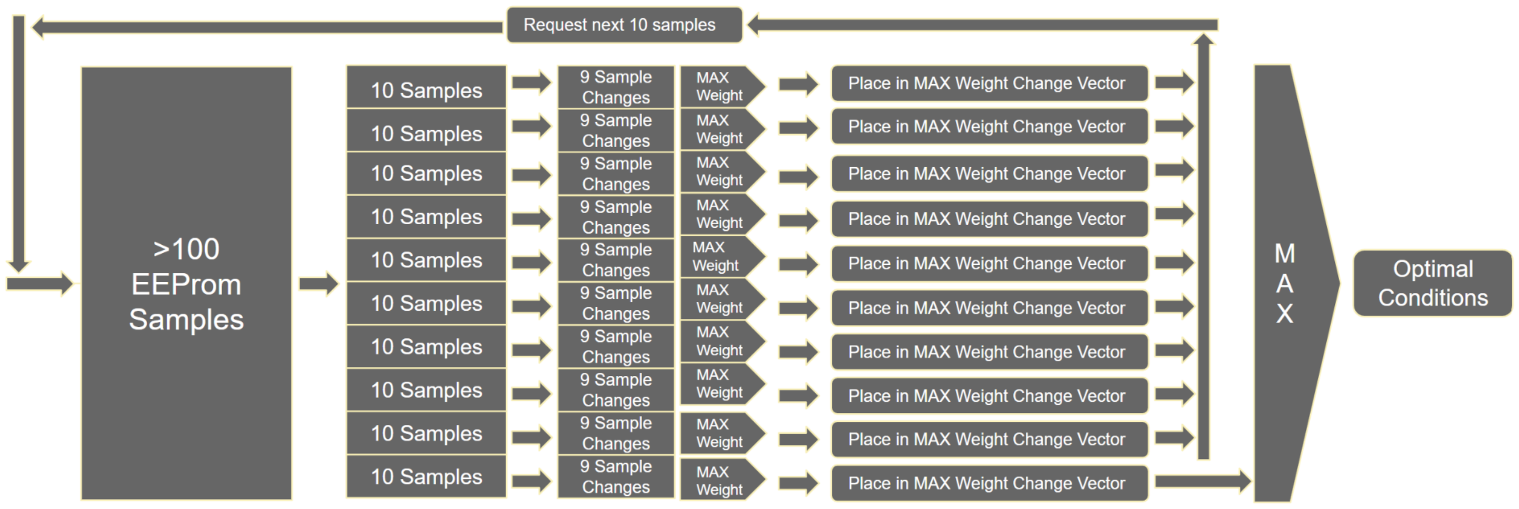

Figure 8: Best Conditions Flow Diagram

at a time. However, additional RAM space should be reserved for variables maintained by the top program. The solution was to take out 10 samples at a time, totalling 400B of RAM space and find an intermediary optimal conditions from among those 10 samples. These intermediary optimal conditions are then placed in a 10-element max sample array. This 10-element max sample array takes an additional 400B of RAM. The program will repeat these steps nine more times until the 10-element max sample array is filled. The sample in the max sample array with the highest weight value will then be returned to the top program. When the returned sample is displayed to the user, he or she will know that its conditions allowed the bees to produce more honey than any other of the last 100 recorded conditions.

Since the program optimizes for highest honey production, the program must return the sample with the highest change in weight rather than just the sample with the highest weight value. Before the intermediary max weight value is determined, the 10 samples from EEProm are converted into nine samples whose weight represents the change in weight between two adjacent samples. The conditions throughout this period are approximated by taking an average of the two adjacent samples from which the weight change was determined.

Pseudocode for the optimize.c functions:

```
eepromSamples       := 10 element array of type data
maxWeightSamples    := 10 element array of type data
bestConditions      := data type variable to be returned
eepromIdx           := EEProm byte index to start reading the first of 10 samples from
maxIdx              := index of the data element with the maximum weight value

For 10 iterations while increasing i from 0 do:

    For 10 iterations while increasing j from 0 do:
        eepromSamples[j] = 40 eeprom bytes starting at eepromIdx
        Increase eepromIdx by 40
    Endfor

    For 9 iterations while increasing j from 0 do:
        For all measurements in data type variable other than weight do:
            Average eepromSample[j]'s and eepromSample[j+1]'s
            measurement and place the result in eepromSample[j]'s
            measurement location
        subtract eepromSample[j] from eepromSample[j+1]'s weight and place the
        result in eepromSample[j]'s weight location
    Endfor

    maxIdx = location of the element with the highest weight value from only the first
            9 eepromSample elements

    maxWeightSamples[i] = eepromSamples[maxIdx]

Endfor

maxIdx = location of the element with the highest weight value in maxWeightSamples
bestConditions = maxWeightSamples[maxIdx]
return bestConditions to top program
```

Inputs

The hive contains a combined 8 inputs to collect data from 6 sensors and the EEPROM chip. These inputs are each connected to their own Atmega chip pins. These pins are enabled for a variety of signal types and protocols, including serial, analog, and digital. Below is a list of the connected peripheral devices and the inputs they use.

Button

The button is connected to pin PB1 and enabled as an input with a pullup resistor. If the button outputs a 0, then it has been pressed. Otherwise, it is not being pressed. Debouncing is also used to ensure a single user press is not read multiple times.

Rotary Encoder

The rotary encoder is attached to PC2 and PC3, which are both configured as inputs with pullup resistors. An interrupt is also configured on PORTC to monitor any changes in PC2 and PC3. When the program enters the interrupt, it checks the values of PC2 and PC3. The encoder state will either be incremented or decremented depending on how the two pin values have changed.

DHT11 Temperature and Humidity Sensor 1 and 2

The first DHT11 sensor 1 is connected to PD2 and DHT11 sensor 2 is connected to PD3. The pin functions as both an input and output for the Atmega to communicate back and forth with the sensor. Since this is the sensor section, we will only consider its functionality in that capacity. The PD2 pin is digital and operates at HIGH and LOW voltage levels.

Once the user sends the request signals (which will be described in the output section), the pin will go HIGH, then LOW, and then HIGH again before the DHT11 chip begins transmitting its values.

The sensor will then output a HIGH signal for more than 30 microseconds before returning to low if it intends to send a one and will stay high for less than 30 microseconds if it intends to send a zero. It will repeat this process 8 times, or until it has sent 1B of data.

That process then repeats five times, to provide the integer temperature value, the decimal temperature value, the integer humidity value, the decimal humidity value, and the checksum, resulting in a total of 5 sent bytes. The checksum is compared to an 1 byte sum of the previous signals to ensure signal integrity.

DX711 Weight Sensor

The weight sensor output was attached to the Atmega PD7 pin. The pin is configured as a digital output. Once the user begins oscillating the output clock to the weight sensor, the sensor will start to output a total of 24 bits in serial. At every clock cycle, the binary output will represent a new input bit. The *i*'th bit value should be placed *i* bits from the LSB.of the 32 bit integer that will store the final weight value.

UV Sensor

The UV sensor input PC1 because this pin can be configured as an analog input and fed into the ADC. The ADC is enabled with a voltage reference of VCC, 8 bit resolution, and prescaler of 128. The conversion is performed at the request of the top program, rather than through an ISR.

EEPROM

The off-chip EEProm uses I2C communication protocols on pins PC4 and PC5. These pins function as SDA and SCL respectively. The I2C address for the EEProm chip is 0xA0.

Outputs

There are 7 output pins for the Atmega to communicate with peripheral devices. All of these pins are digital and some also use common serial communication protocols. Below is a list of the peripheral devices and a description of how they use the pins to which they are connected.

Fans

The fans are not directly connected to the Atmega, but rather each go through one NPN transistor and a resistor before reaching the Atmega PB6 and PD6 pins. For more information on the use of NPN transistors, see the Voltage and Current Requirements section. To turn the fans on PB6 and PB5 output HIGH. To turn them off, the pins are set to LOW.

Heaters

The heaters are connected to PD5 and PD6, which are configured as digital output pins. The heaters are turned on when PD5 and PD6 are set to HIGH and turned off when PD5 and PD6 are set to LOW.

Weight Sensor

The weight sensor uses PB0 as a digital clock input. This means that it must be configured as a digital output on the Atmega. Whenever the Atmega starts oscillating the digital clock, the weight sensor will begin outputting values bit by bit. Therefore the Atmega only needed to change these pin values when it wanted to receive weight data.

EEProm

The off-chip EEProm uses I2C communication protocols on pins PC4 and PC5. These pins function as SDA and SCL respectively. The I2C address for the EEProm chip is 0xA0.

LCD Screen

The LCD screen is connected to the TX pin on the Atmega so it can send commands and ASCII characters through a serial link. The data is transmitted through the TTL protocol with a 9600 BAUD rate.

Communication

The Hive uses standard serial communication protocols to communicate with some of its peripherals. Using standard protocols is engineering best practice because it makes it easier for the user to identify any issues that might arise or repair the Hive if needed. It will make this easier because there is ample documentation on these protocols and the common issues that come up when using them. The two standard communication protocols used in the Hive are : RS-232 and I2C.

RS-232 is used by the Atmega to send serial commands to the LCD screen through the TX link. The bits are sent character by character in the UDR0 register. The lower-level implementation of the communication protocol is handled in the background.

I2C is used to communicate with the EEProm device. The EEProm comes with an address set to 0xA0 by the manufacturer. I2C serial communications is handled by source libraries opaque to the programmer. This follows engineering best practice because it ensures that the programmer will not cause any errors in the serial communication steps themselves.

User Interface and User Interaction

As stated above in the *Structure* section, the *top.c* file contains the state machine and user interface functionality of the code. Within the infinite while loop simulating the product's operating system, the first the software does is check which state it is currently in. Then, it checks for user input like a button press or a change in the rotary encoder position. If either of these has occurred, the software makes the corresponding update to the information on the screen and displays it to the user. While the button is polled, the rotary encoder makes use of a pin change Interrupt Service Routine to ensure the code responds quickly to user input. The button polling also makes use of debouncing to improve reliability.

Along with updating the state, the top code also updates output function logic. This includes turning the fans and heaters on and off. It does this if the temperature within the hive varies from the user-defined temperature value by more than the margin – which we have set to 3 degrees Celsius. This is also where the code records any warnings that need to be seen by the user like an excessively high temperature or sudden weight change. This data will then be displayed to the user at a later time.

Finally, the top file gathers new measurements. It does so every five seconds using an internal timer and Interrupt Service Routine. This service routine is also used to store a measurement in the EEPROM every hour. This is the only time-related functionality within the project, but future iterations could make use of a real time clock module for some of these operations.

EEPROM

The Hive utilizes an I2C bus to store data to and read data from an EEPROM. It stores ten values for each data point: internal temperature (integer and decimal), external temperature (integer and decimal), internal humidity (integer and decimal), external humidity (integer and decimal), UV index, and weight. The integer and decimal are separated to ensure no floating point arithmetic is done, since that would severely increase program memory size. We assume that each of these values will be 3 digits or less, so each value takes up 4 bytes (3 digits plus a null byte). Because we know that each data point will take up 40 bytes and the order in which data is stored, we can easily recover that data as well. Along with all of these data points, we also store the address of the next slot to write to and how many data points we have recorded thus far. We use these values in the event of a system restart or loss of power to ensure that we do not write over previous data that is still useful.

These EEPROM values are read to our program in batches and either displayed to the user as the most recent data (if the system was just restarted) or operated on within the *optimize.c* file. Since the EEPROM is read to once an hour and it can store about 32 kB, we will be able to store about a month of data before we overwrite old values. In the future more EEPROM memory could be added to increase the amount of data we read and operate on.

CONCLUSION

Ultimately, “The Hive” is a great product for both individual and commercial beekeepers alike, allowing each to guarantee the health of their hive and produce the most amount of honey with the best optimal conditions. With The Hive, we hope that we can enable anyone to start beekeeping. By doing the hard stuff for you, beekeeping can be a simple and pleasant experience for both you and your bees, living in perfect harmony.

Appendix A: Schematics

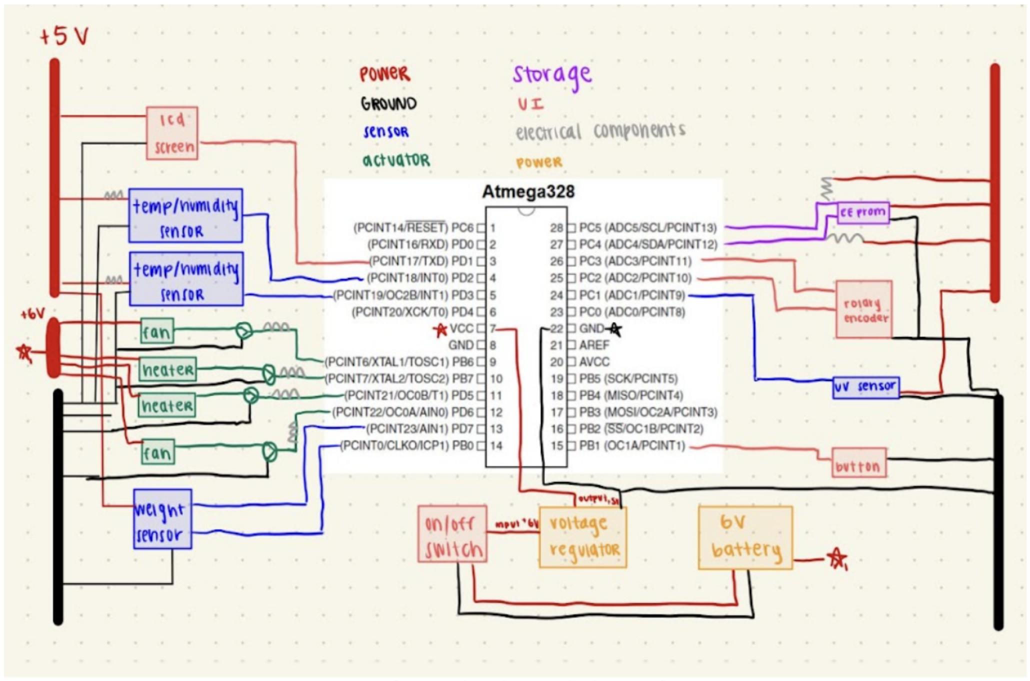

Figure 4: Pinout Schematic

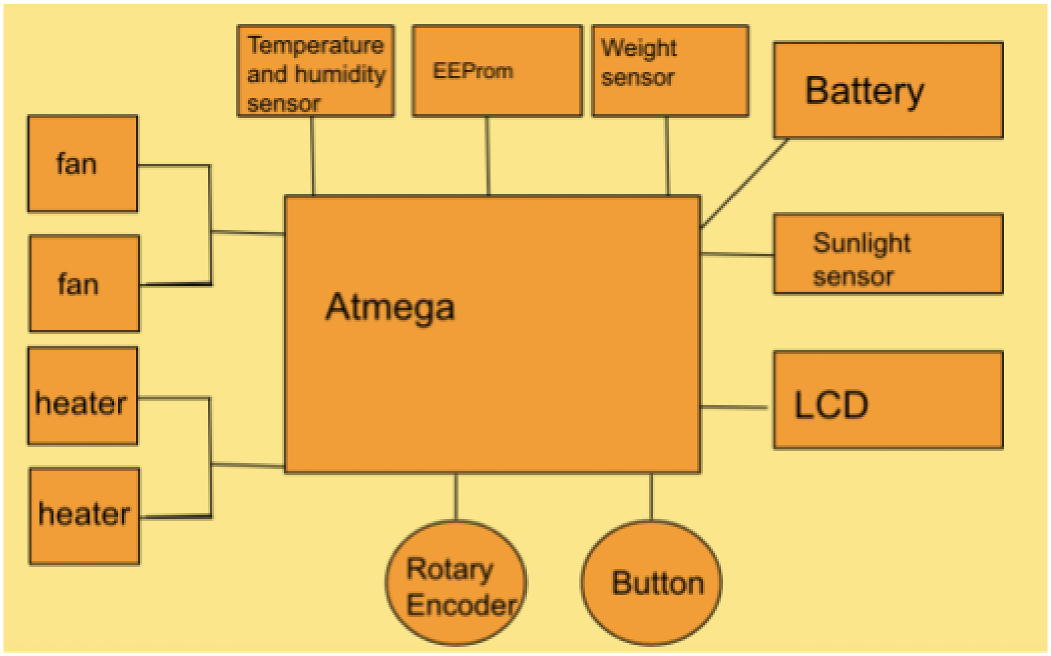

Figure 5: Block Diagram

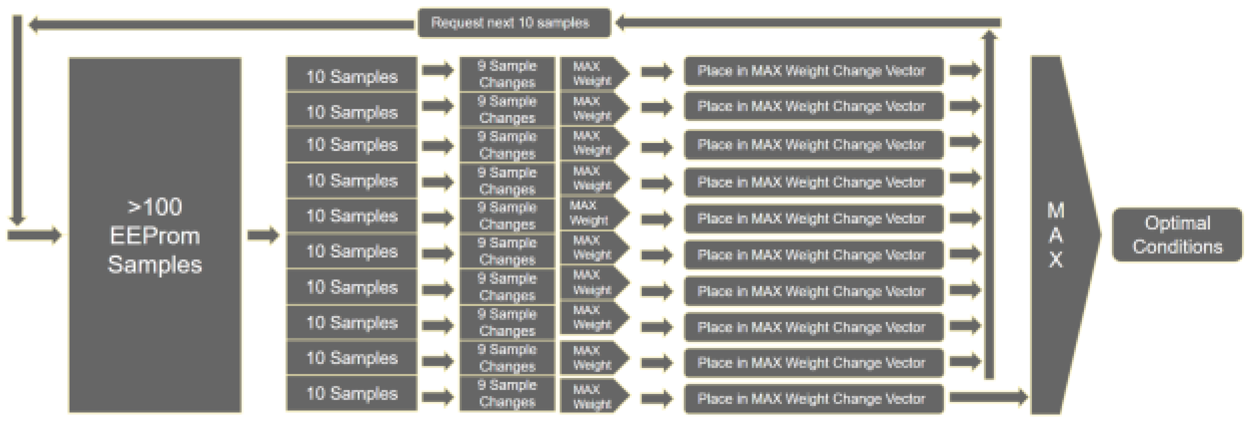

Figure 8: Best Conditions Flow Diagram

Appendix B: Parts List

Sensors	Quantity	Part #	Cost
Temperature and humidity sensor	2	DHT11, Adafruit 386	$5.00
UV Light sensor	1	Parallax 28091, DigiKey 149-28091-ND	$10.49
Weight Sensor	1	HX711 Amazon B078KS1NBB	$8.49

Actuators	Quantity	Part #	Cost
Fans	2	Adafruit 4468	$2.95
Heaters	2	Adafruit 1481	$5.95

User Interface	Quantity	Part #	Cost
LCD Screen	1	NHD-0420D3Z-FL-GBW -V3	$28.03
Button	1	Given by Professor	NA
Rotary Encoder	1	Given by Professor	NA
Switch	1	Given by Professor	NA

Power	Quantity	Part #	Cost
6V Battery Housing	1	Given by Professor	~$1.25
AAA Batteries	4	Given by Professor	~$2.50
5V Voltage Regulator	1	Given by Professor	~$0.70
Voltage Regulator Heat Sink	1	Given by Professor	~$0.20
NPN Transistors	4	Given by Professor	NA

Other Electronics	**Quantity**	**Part #**	**Cost**
EEProm	1	Given by Professor	NA
Resistors	8	Given by Professor	NA
ATMEGA328P	1	Given by Professor	$3.00

Appendix C: Cost

The cost of each item we used to produce The Hive is listed above in **Appendix B: Part's List**. Because some of the items were given to us by the Professor, we were not able to quantify everything, which is why some component's cost was listed as either "NA" or "~" in the Cost column. In total, we estimate that the material cost of this product is $90. If the components were to have been purchased all in bulk, we anticipate that the cost would significantly decrease. All generic electrical components, such as the resistors, transistors, wires, buttons, rotary encoders, switches, and more would be negligible. The most expensive components were the LCD Display, UV light sensor, and Weight sensor making up over ½ of our total cost. If we would be able to get these components for less, through different parts or buying in bulk the cost of our product would significantly decrease.

Appendix D: Signature Sheet

	Anna	Josh	Matt
System design	60%	20%	20%
Component Selection	20%	20%	60%
Hardware Design	60%	20%	20%
Software Design	5%	60%	35%
Documentation	20%	60%	20%
Project Report (oral)	33.3%	33.3%	33.3%
Project Report (written)	33.3%	33.3%	33.3%
Total	33%	35%	32%

Team Member 1:	Joshua Solomon	[illegible]
Team Member 2:	Anna Pendleton	[illegible]
Team Member 3:	Matthew Liberson	