NECTARFY

The Smart Beehive

December 21, 2021

Abstract

The Nectarfy, a smart beehive which can record relevant data related to the health of the beehive and send a metric of how healthy the hive is to beekeepers. The hive will then be able to use the metrics it records to give advice to beekeepers on how to best treat the situation the hive is currently experiencing, which includes heat and weight variations. The hive will be able to assist in the beekeeping process, elevating the quality of life for beekeepers, and assist novice beekeepers in learning the craft. The beehive is able to operate both autonomously and with computer control.

The key features of the hardware are a load cell base, which can either replace existing bottom boards of beehives, or support the bottom board of the beehive itself. A beehive frame within the beehive was retrofitted to include the sensor electronics which were required to identify the situation within the beehive. On the outside, an arm mounted with an LED and servo arm was attached to give a physical notification to beekeepers when the hive reaches harvesting capacity. The software developed was a Windows Form and C# which is able to collect the data read, save them to a local file, and give curated instructions and advice on what actions to take based on the status of the hive itself.

Objective

1. **Vision:** The overall vision for this project is to create a 'Smart Beehive', which can record relevant data related to the health of the beehive and send a metric of how healthy the hive is to beekeepers or owners.

1. **What we will design:** In terms of mechanical design, we will design components around a "Deep Super"-sized beehive box, from BC Bee Beekeeping Supplies. These components include a servo/LED system which gives a visual indication of when the hive is at a harvestable weight, a mounting interface for a load cell, and an enclosure to protect other essential electronic components. The MSP430FR5739 board will be used for tracking relevant variables (temperature, weight, acceleration) and to handle communication with the beekeeper, by forwarding packets of data using UART to an accompanying C# program as the Smart Beehive's user interface.

2. **Final Product:**

Figure 1: Default State of Beehive

Figure 2: Alert State of Beehive

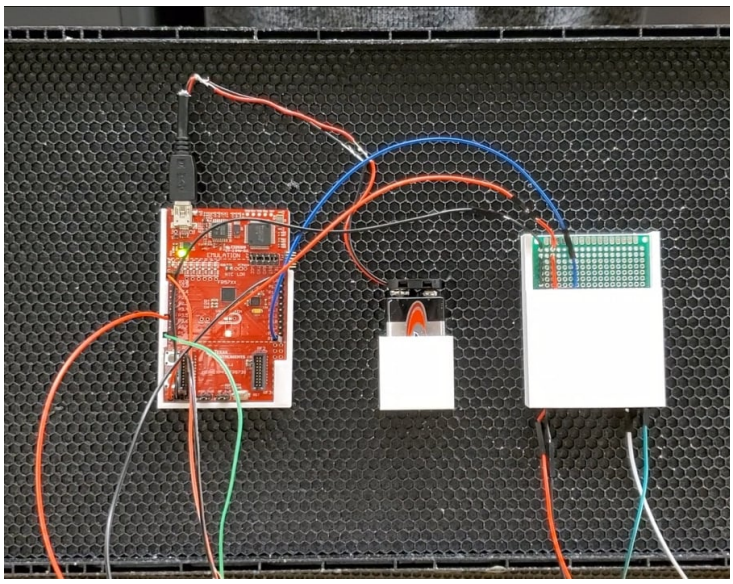


Figure 3:Top Left: Servo Attachment. Top Right: Instrumentation Frame. Bottom: Location of Instrumentation Frame

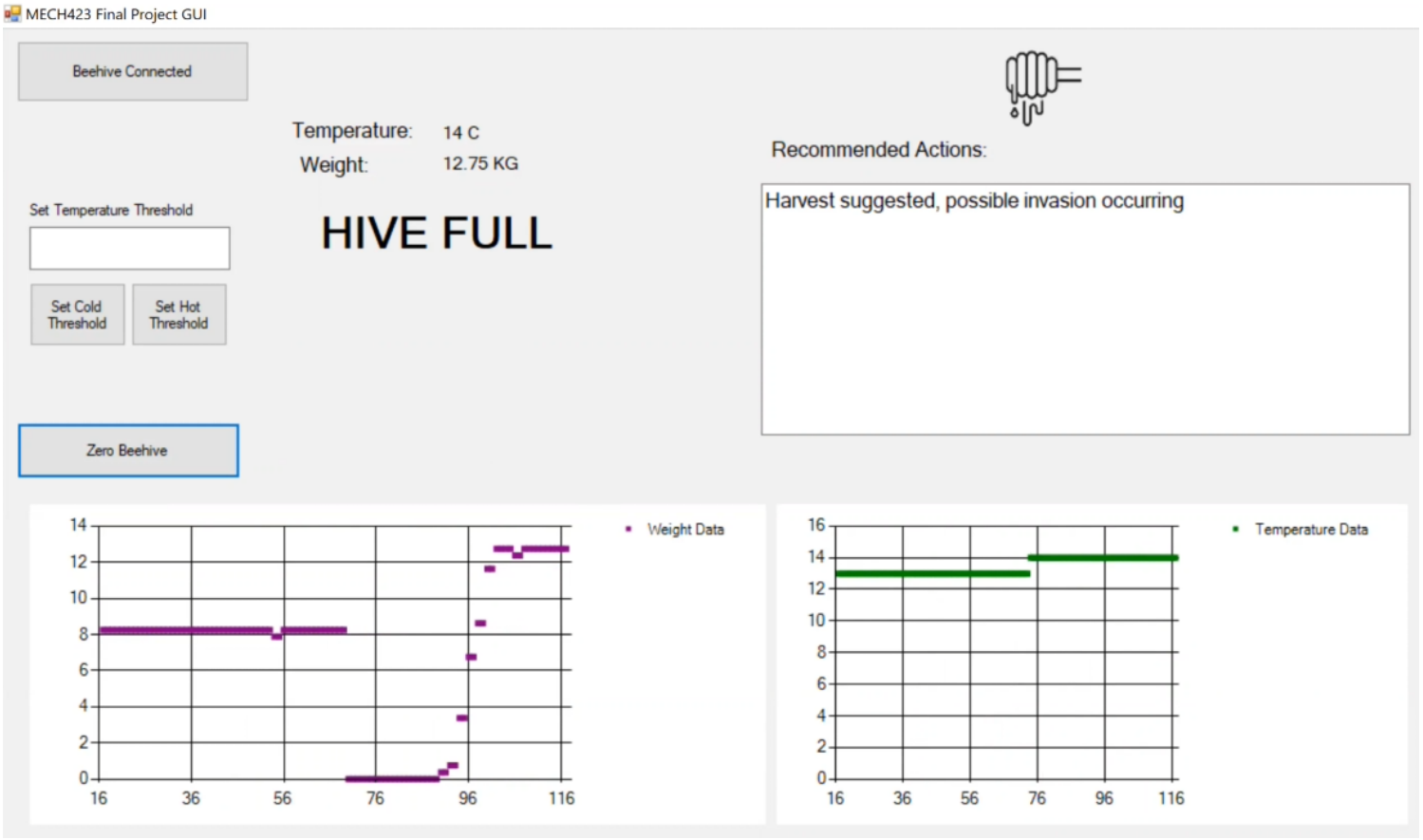


Figure 4: Windows Form GUI

We were able to accomplish almost all the goals we set out to do, including autonomous operation, which was a stretch goal. Ultimately, we were unable to send wireless messages from the hive to the user, which was one of the other stretch goals.

Rationale

Technology in the beekeeping industry has not had any major advancements in the past 150 years. We believe that technological advancements will not only improve colony health, but also combat the overall decline of beekeeping as a practice. We aim to solve certain issues revolving around beekeeping, with our implementation of an “smart beehive” aimed at urban beekeeping, which has seen growth in the last decade. Urban beekeeping allows non-professionals to

beekeep effectively without needing the resources of a dedicated apiary. We feel that our contribution through this project would be impactful to this industry as a whole, as we can provide a health benefit to the bees and improve the efficiency of existing beekeeping practices.

Often, these novice beekeepers are not educated or trained well enough to ensure their colonies will survive. Without adequate help, many colonies have reported total colony death within two years. The product we are creating aims to lessen the risk for novice beekeepers and assist them in creating thriving colonies, while also helping veteran beekeepers save time and effort when tending to their hives. It is different from similar work at UBC as there has currently been no effort at UBC to develop smart hives which we are aware of. There are existing solutions already existing in the market with sensor arrays and other data metric systems, but ours is unique in that it is able to operate fully autonomously on much lower power, none of the existing solutions have a physical attachment which shows the status of the hive (servo flag and LED), and the existing solutions are far more expensive than the solution we are able to develop.

List of Functional Requirements

Functional Requirements	**% Effort**	**Responsible Person**
The device will track the ambient temperature of the beehive frame, and record the data in a user-friendly format.	10%	Thomas
The device will track the overall weight of the beehive frame, and record the data in a user-friendly format.	50%	Thomas & Joseph
The device will send a mechanical signal when the beehive is at an acceptable harvesting weight.	15%	Thomas
The device will detect when the beehive experiences an excess amount of movement.	15%	Joseph

The device will house electronics to keep it secure from the external environment.	10%	Joseph

Functional Requirement 1: The device will track the ambient temperature of the beehive frame, and record the data in a user-friendly format

- Approach and Design
    i. The functional requirement is meant to track the ambient temperature of the beehive. This is because the temperature within hives needs to be regulated to keep bees healthy. Typical ways to assist with the heating are currently to wrap the hive with insulation to help bees be more efficient with natural heating.

    The MSP430 board's built-in temperature sensor will be used to do this, but additional attachments may be required in order to increase the sensitivity of the sensor. Refer to Appendix A for high level pseudocode of the relevant MSP430 and C# programs.

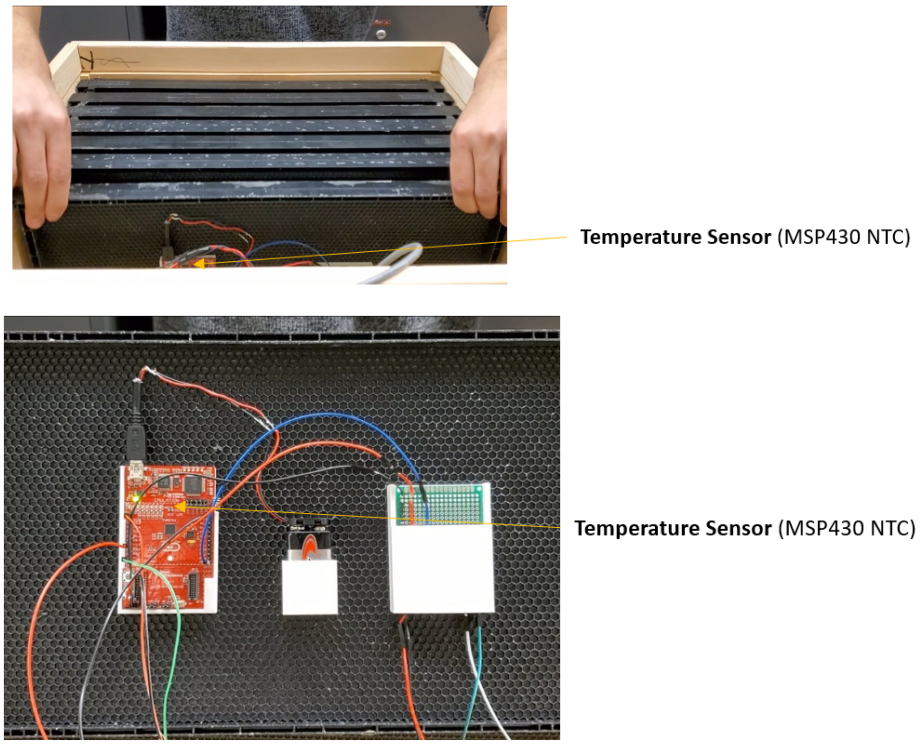
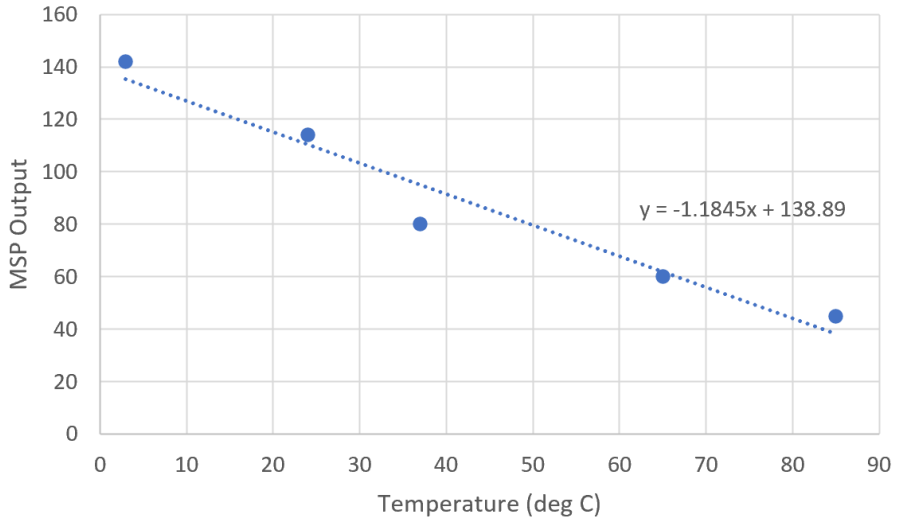


Figure 5: Temperature Sensor Location

- Inputs and Outputs
    i. The inputs for this FR will be an analog signal which comes from the temperature sensor on the board itself. The temperature within a beehive needs to generally be kept at around 35 degrees Celsius. The temperature range over an entire year for 2021 in British Columbia was between -6 to 32.4 degrees Celsius.
    ii. The output for this FR is a user-readable temperature value in degrees Celsius. The analog data input will be sent to a computer through UART (in the format of a serial message packet, similar to Lab 2), parsed by an accompanying C# program, converted to degrees Celsius, and presented to the user in the form of a CSV file.
- Parameters
    i. The software parameters are the calibration values used to translate the analog signal received from the MSP430 board into degrees Celsius.

Assuming a linear calibration curve, the slope and offset are relevant parameters. The tuning of these parameters affect the accuracy of the output. To optimize them, a soldering iron and the outdoor environment was used to manually calibrate the sensor, by mapping known temperatures to analog values, and an end-points-based linear curve was derived.

- Development Plan
  i. Use relevant information from Lab 2 to enable the MSP430 temperature sensor and send outgoing temperature readings through UART
  ii. Use relevant information from Lab 1 and 2 to create an accompanying C# program which receives, converts and saves analog temperature readings into a CSV file
  iii. Physically mount the MSP430 board to the beehive housing
- Test Plan
  i. Use the MECH423 Serial Communicator to check if the values are being transmitted through UART and are sensitive to temperature changes
  ii. Use an external temperature sensor to record the ambient outdoor temperature and record the associated analog values read from the MSP430, and use a soldering iron to slowly increase the temperature to gather more analog data points for tuning the calibration curve (within ~2-5 deg Celsius)
  iii.



iv. After the board is physically mounted, place the beehive and external temperature sensor outside to confirm that the temperature sensor is returning accurate temperature data (within ~2-5 deg Celsius), and that physical movement of the beehive does not displace/break the MSP430 mount. This test was performed by carrying the hive outside given the winter conditions, and doing so showed that the sensor array was undisturbed.

Functional Requirement 2: The device will track the overall weight of the beehive frame, and record the data in a user-friendly format.

- Approach and Design
  i. The objective of this functional requirement is to return a weight (in kg) from the beehive frame. This is to track the amount of honey being accumulated in the beehive as time goes on. A load cell will be attached to the mechanical design and will be connected to the MSP430 board, which receives the load cell data. An accompanying C# program will be developed to interpret and save the load cell data into a user-friendly format. Refer to Appendix A for high level pseudocode of the relevant MSP430 and C# programs.

Figure 6: Weight Detection SystemDesign

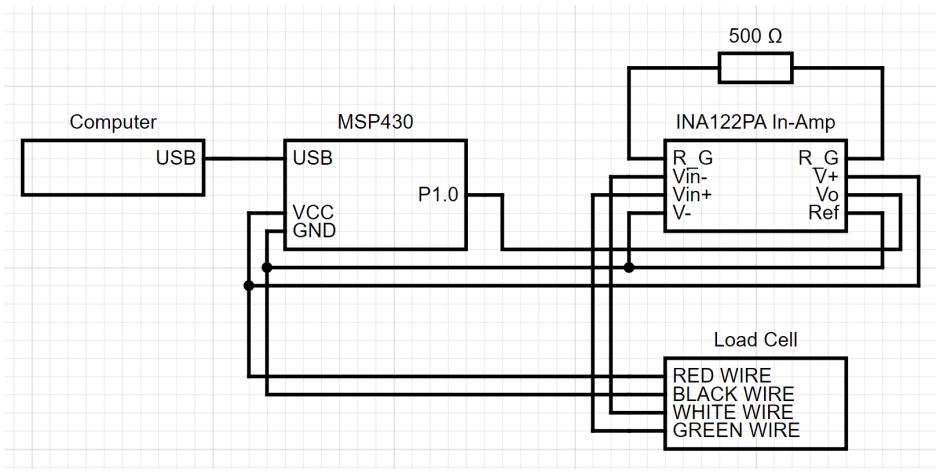
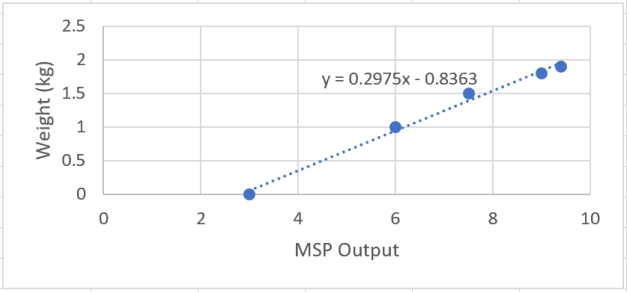


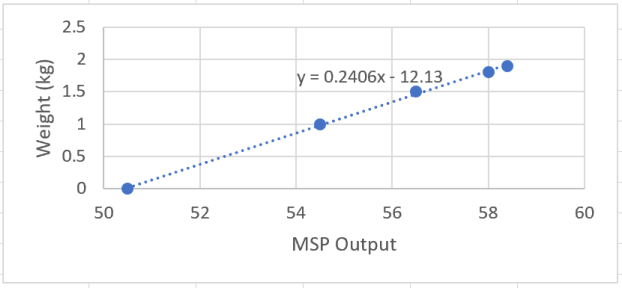
Figure 7: HIgh Level Circuit Diagram for Load Cell

- Inputs and Outputs
    i. The input for this FR is the voltage output of a load cell, corresponding to the weight of the beehive box. Our load cell is expected to handle up to 90 pounds, which is the typical weight of a “Deep Super” beehive box, the box we will be using. Therefore, we will use a 114990100 load cell from Seeed Technology Co, which has a range of 0-50kg.
    ii. The output of this FR is load cell data, presented to the user in the form of a CSV file. The load cell voltage from the MSP430 will be sent to a computer through UART (in the format of a serial message packet, similar to Lab 2), and converted into kg. While the exact transfer function from the load cell data to kg cannot be determined at the current proposal stage, we will manually calibrate the sensor using end-points-linearity calibration, using known masses to determine the range.
- Parameters
    i. The hardware parameters involve the weight range of the load cell. As mentioned above, we are using a “Deep Super”-sized beehive box, which supports up to 90 pounds. Our chosen load cell will be the 114990100 load cell from Seeed Technology Co, which has a range of 0-50kg.
    ii. The software parameters are the calibration values used to translate the load cell signals sent to an accompanying C# program into kg. Assuming a linear calibration curve, the slope and offset are relevant parameters. The tuning of these parameters affect the accuracy of the output. To optimize them, weights of known masses were used to manually calibrate the load cell and an end-points-based linear curve was derived. The weights supplied from the instrumentation lab were used.

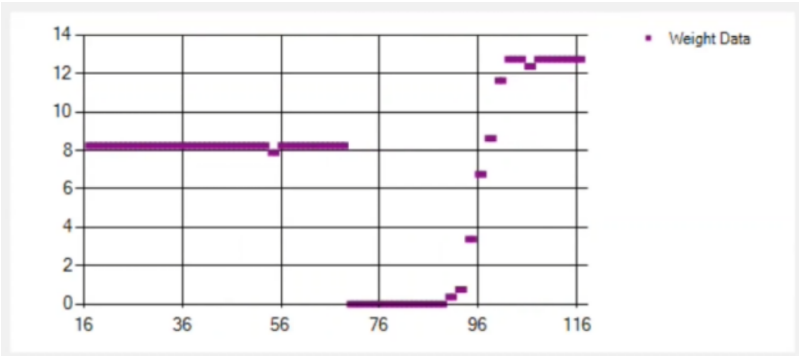
no beehive		
weight (g)	msp output	weight (kg)
0	3	0
1000	6	1
1500	7.5	1.5
1800	9	1.8
1900	9.4	1.9



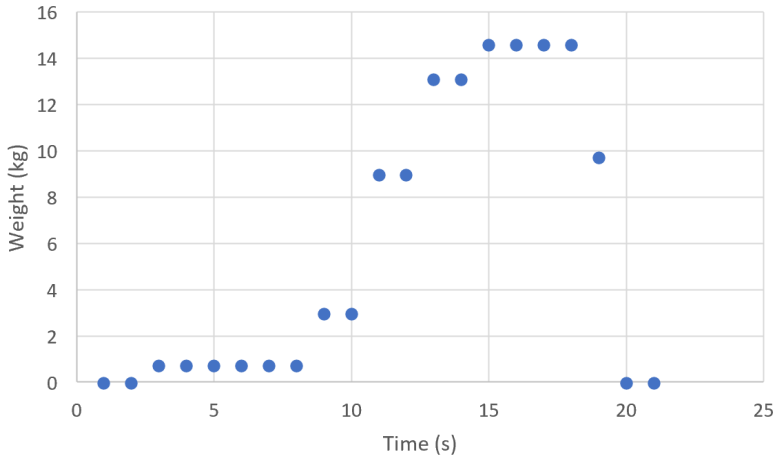
with beehive		
weight (g)	msp output	weight (kg)
0	50.5	0
1000	54.5	1
1500	56.5	1.5
1800	58	1.8
1900	58.4	1.9



- Development Plan
  i. Assemble the circuit and verify the feasibility of the load cell passing data to the MSP430 board, to an accompanying C# program using UART
  ii. Use relevant information from Lab 1 to program a C# interface that translates and stores the load cell data into a CSV file
  iii. Calibrate the load cell to accurately display the weight in kg
- Test Plan
  i. Use the MECH423 Serial Communicator to check if the MSP430 is receiving data from the load cell and sending it forward through UART, and that the values change when more/less weight is added to the load cell.

1. 

ii. Open and graph the CSV data saved by the C# program to check if the load cell data is being recorded properly over time



1.

iii. Compare the converted CSV values (after calibration) to the known values added to the load cell, check that the data is accurate to within ~3 kg (accuracy limit determined as a safety factor of 2 for the combined error of the load cell, from the datasheet). These were done and tested by using the weights available in the Mech 423 laboratory.

Functional Requirement 3: The device will send a mechanical signal when the beehive is at an acceptable harvesting weight.

- Approach and Design
  i. The objective of this functional requirement is to allow the MSP430 board to report back when the beehive has reacted critical capacity. The hive will be modified to have a small motorized flag with an LED attachment. A bright LED will be attached to the end of an actuating arm, which will be attached to a servo allowing the arm to move up and down, similar to a mailbox flag. Once the hive is full, the flag will be raised, and it will also have a blinking LED on the end along with a bright color to maximize visibility in various weather and lighting conditions. Refer to Appendix A for the code of the relevant MSP430 and C# programs.

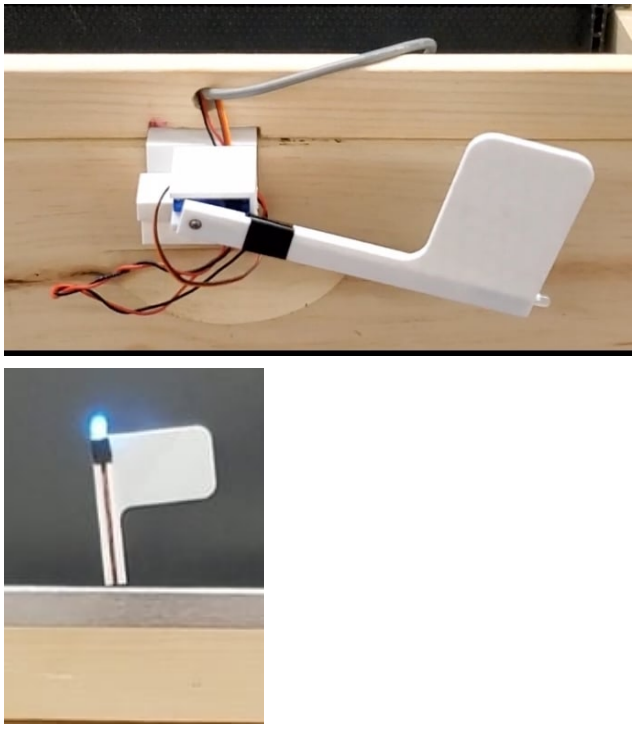


Figure 8: Mechanical Alert System with Flag

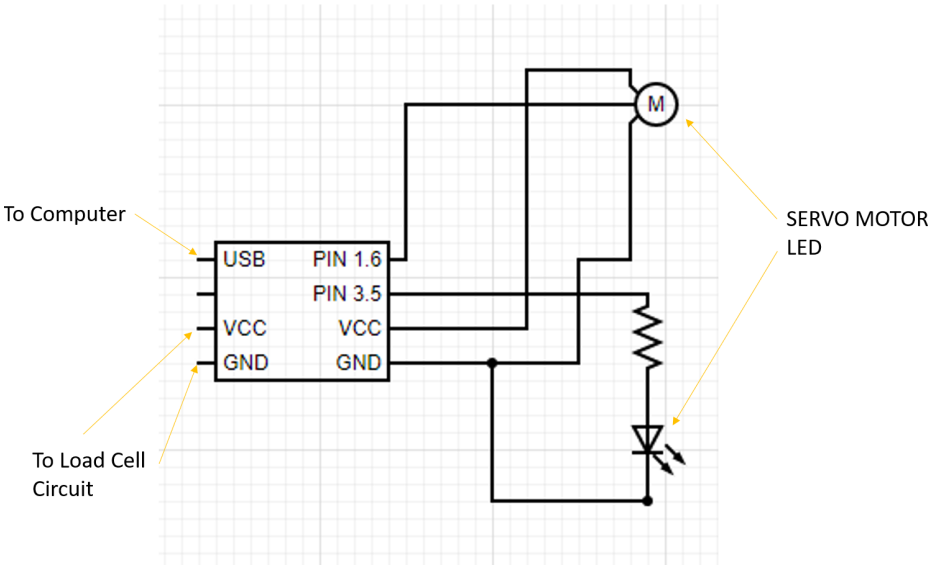


Figure 9: High Level Circuit Diagram for LED and Flag

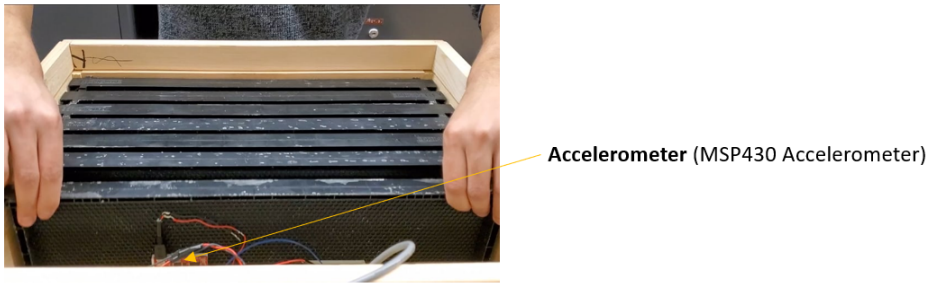
- Inputs and Outputs
    i. The hardware-related input for this FR is the raw load cell voltage output sent to the MSP430. Further detail can be found in Functional Requirement 2.
    ii. The software-related input for this FR is a serial message packet sent from the accompanying C# program back to the MSP430 board. After conversion to kg, there is an adjustable threshold limit for 'acceptable harvesting weight' in kg, and when the C# program receives a converted load cell value above that limit, it will send a serial message packet (similar to the packet used in Lab 2 with a start byte and command byte, etc) to the MSP430 board to initiate moving the servo motor and turning on the LED. Refer to Figure 8 for the circuit diagram.
    iii. The outputs for this FR are movement of an SG90 servo motor and a visible light from the LED. These visual indicators will be presented to the user to show that the beehive is ready for harvest. Refer to Figures 1 and 2 for a preliminary design of these visual indicators.
- Parameters
    i. The only software parameter to be adjusted is the threshold weight value for the C# program to send a signal to the MSP430 board, which is a user-specified parameter (implemented as a TextBox in the C# program). Other relevant parameters are already outlined in Functional Requirement 2, involving the calibration values for the load cell.
- Development Plan
    i. Assemble the connections from the MSP430 board to the servo motor and LED according to Figure 8 Circuit Diagram
    ii. Complete development of Functional Requirement 2 to ensure that the weight values are calibrated accurately
    iii. Program software functionality (for C# program and firmware on the MSP430 board) to detect when weight threshold has been passed, then send/receive serial message packets and convert them to servo motor/LED changes
- Test Plan

i. Verify that the MSP430 board can control the servo motor and LED with a simple ‘LED blink’ and ‘servo sweep’ program. Both of these were completed and validated to be functional. See video.
ii. Test for weight calibration outlined in Functional Requirement 2
iii. Verify that the LED turns on and the servo motor moves when specified weight threshold is passed, and moves back when weight goes back below the threshold. This was found to be functional. See video.

These tests can be found to be function at around 00min 45 seconds in the video.

Functional Requirement 4: The device will detect when the beehive experiences an excess amount of movement.

- Approach and Design
  i. The objective of this FR is to send a signal from the MSP430 board to the C# program when the beehive frame experiences movement. The only hardware necessary is the accelerometer present on the MSP430 board, and an accompanying C# program to interpret the signal when the board experiences a sudden acceleration. Refer to Appendix A for the code of the relevant MSP430 and C# programs.



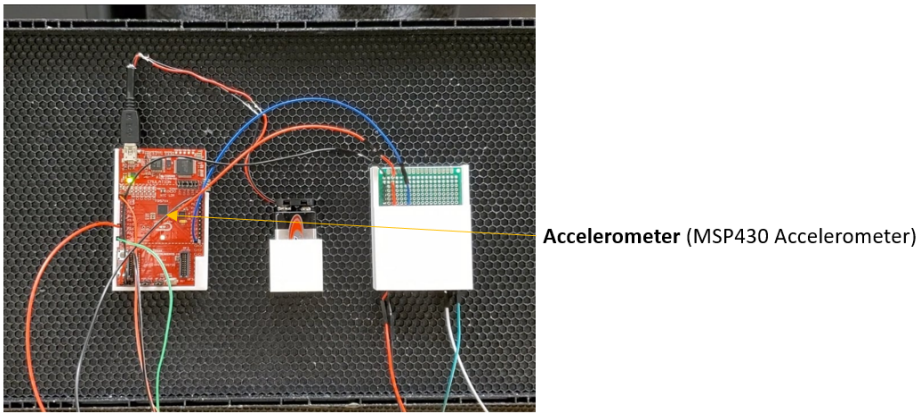


Figure 10: Accelerometer Location

- Inputs and Outputs
    i. The input for this FR is the physical movement of the MSP430 board, outputted as a voltage output from the MSP430's accelerometer. The accelerometer will detect the sudden movement and send a signal (in the format of a serial message packet, similar to Lab 2) to a C# program if an accelerometer threshold value is reached.
    ii. The C# program will receive the movement packet from the MSP430 and visually output a flashing warning sign on the screen for the user to see. This simulates an 'alarm system' as if an intruder is attempting to steal the beehive.
- Parameters
    i. The parameters for this FR rely on the 'zero' orientation of the MSP430 board when mounted to the beehive. Once the MSP430 is mounted, the stationary values in x, y, and z are recorded as the 'zero' position. Another parameter for changes in x, y, z data is recorded. This threshold will determine whether the movement is sensitive enough to trigger a pop-up on the C# interface, which will be calibrated appropriately once the prototype is created. The zero values were found to be:
        1. **X: 125**

2. **Y: 153**
3. **Z: 130**

- Development Plan
    i. Use relevant information from Lab 1 and 2 to program the MSP430 board to send raw accelerometer data to a C# program, and have the C# program send a flashing image once the accelerometer value passes some threshold (change from its zero position)
    ii. Physically mount the MSP430 board to the beehive, pick up the beehive to calibrate threshold value appropriately
- Test Plan
    i. Test if the C# program will show a flashing image once the MSP430 board is moved around over some threshold (similar to Lab 1 gesture recognition)
    ii. Test if moving the beehive (with a mounted MSP430 board) will trigger the flashing image on the C# program

    Both these tests were found to be functional, see video at 0 min 33 seconds.

Functional Requirement 5: The device will house electronics to keep it secure from the external environment.

- Approach and Design
    i. This FR outlines the mechanical structure of the design, that attaches to the beehive frame and secures the electronics.

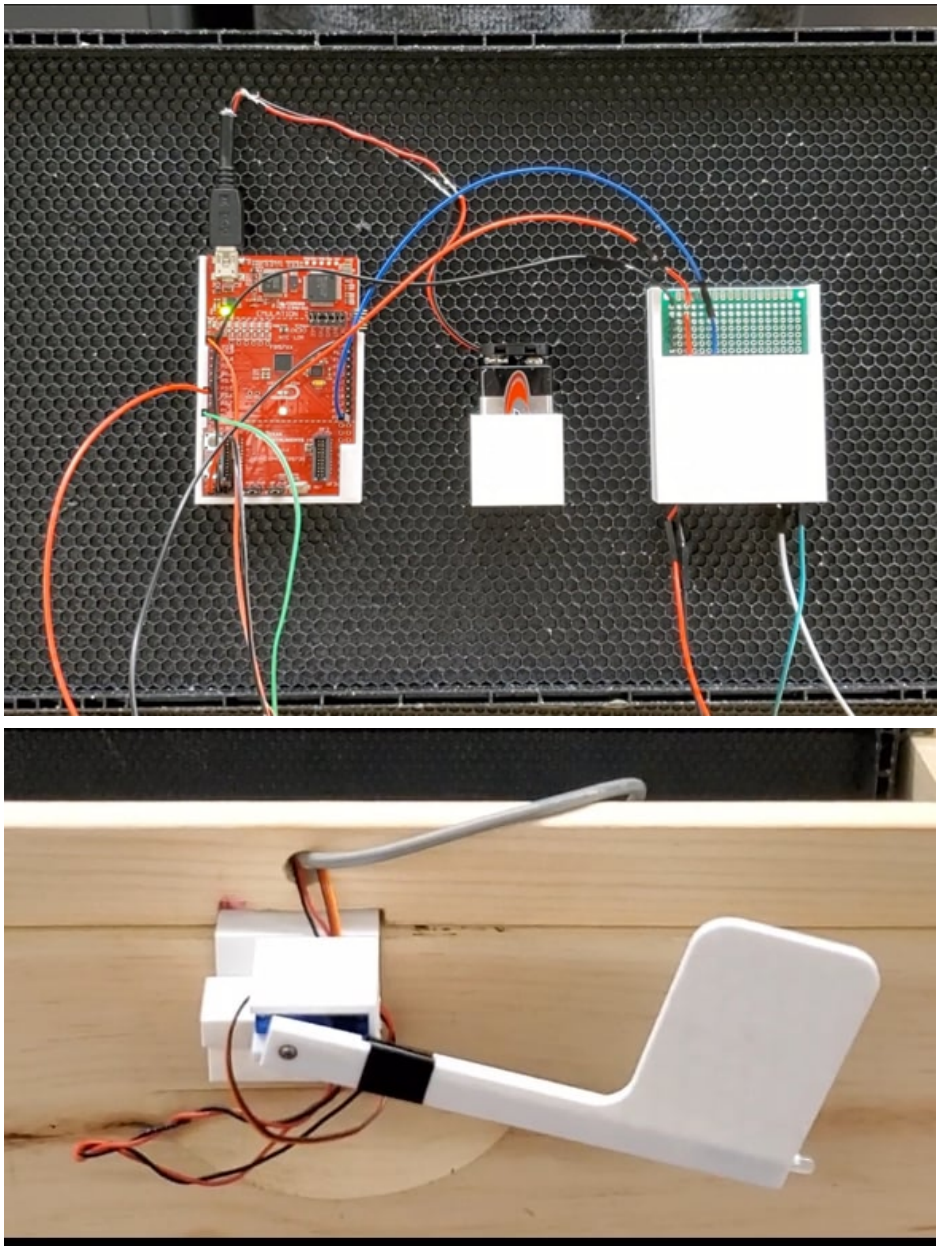


Figure 11: Enclosure Design for Various Parts

- Inputs and Outputs

i. There are no inputs for this FR, because there are no associated hardware or software modules involved. This FR only encompasses the physical design, to be done in SolidWorks and 3D-printed.

- Parameters
  i. The only parameters to adjust for the electronics enclosure involve the dimensions of the enclosure itself. The enclosures were made separately for each piece to reduce interference as much as possible. The servo enclosure had to be designed so it would be able to hook around the hive as well as maintain enough room for the top lid.
- Development Plan
  i. Design a preliminary enclosure based on approximate dimensions of the components inside it, using SolidWorks.
  ii. 3D print the enclosure as a prototype to fit the components, and determine if the dimensions need to be adjusted.
  iii. 3D print the adjusted enclosure and mount it to the beehive frame.
- Test Plan
  i. Verify that 3D representations of enclosed components fit inside the enclosure.
  ii. Verify that the actual components fit inside the prototype enclosure.
  iii. Verify that the enclosure is mounted properly, and does not fall off when the beehive box is in motion.

  These tests were all passed, as can be evidently seen from the final design being able to hold all components.

System Evaluation

The following tests were performed to test the functionality of the complete system:

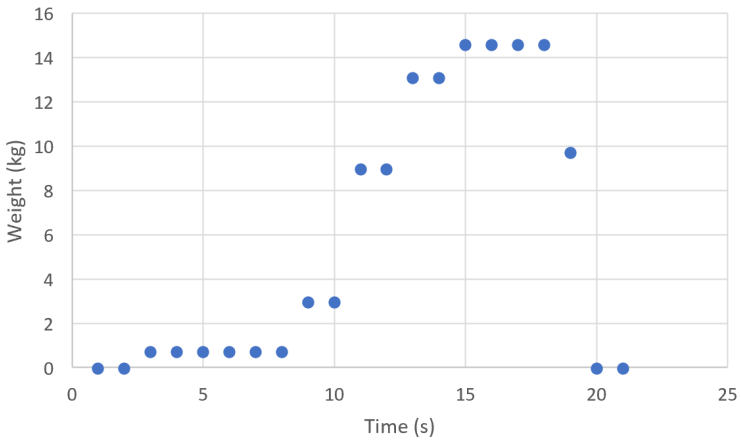
Experiment 1:

- Use the MECH423 Serial Communicator to check if the MSP430 is receiving any form of data from the load cell and sending it forward through UART, and that the values change when more/less weight is added to the load cell. This is the crux of one of the functional

requirements and was found to be working. We used various weights of known sizes and checked to see if the load cell was able to determine their weights with minimal error.

Experiment 2:

- Open and graph the CSV data saved by the C# program to check if the load cell data is being recorded properly over time. This is to confirm that we can have data flow from the load cell to the MSP430 board to the C# program used to save the data.



-

Experiment 3:

- Compare the converted CSV values (after calibration) to the known values added to the load cell, ensuring the threshold would consistently trigger at a chosen point. This was done and demonstrated during Demo Day. The threshold then was exactly two jars of Kirkland Honey, which was about 600 grams. During the demo, the hive was consistently able to perform at this weight, albeit one which is relatively light.

Experiment 4:

- Ensure various system triggers are activated. First, see if the servo will trigger when the load cell reaches a limit value chosen by the user. Check if the temperature triggers for hot and cold temperatures. Check if the accelerometer triggers when beyond a threshold based on 0 value.

These were tested and found to work. See the video for all of the results. A hairdryer was used to test the temperature NTC was functional on the board, which was then used to show the results of the temperature sensor working in the video.

Reflections

1. Most of the project was very functional, and there were relatively few hiccups in the process. The servo and sensor information went well, and generally the mechanical design was also straightforward. One of the things which didn't initially work was getting the load cell to function as intended. We initially started off using the HX711 Load Cell amplifier, which had a built-in ADC and a two-wire interface to read the data off it directly. Unfortunately, we realized that the library was made for Arduino only, and we had trouble getting Energia (a software used to translate Arduino code to MSP boards) to compile and upload the code to our MSP430 board. To get around this issue, we first attempted to reverse-engineer the Arduino library, which ended up unsuccessful due to not being able to recreate specific clock timings of the module interface. We then opted to build a protoboard with an INA122PA Instrumentation Amplifier to manually amplify the data and read it using the MSP430's ADC. This option ended up working fine for our use case.

   There were some balancing issues with regards to the load cell and the mechanical design of the actual weight detection system. If done again, two load cells would be used instead of one, and the enclosure and mechanical setup for the load cell would be designed and made out of a tougher material than wood. There was some plastic deformation of the wood which was used to support the load cell itself, which could be improved on in future versions. Furthermore, we would like to reach all our stretch goals, including exporting information wirelessly so the beehive can perform even better as an autonomous unit.

2. The first most useful thing would be learning to read datasheets and understand how to write firmware codes based on those datasheets. Given that this project used amplifier modules and other hardware which were not extensively studied in MECH 423, the ability for us to read the datasheet from the company and understand how to work with the device was very valuable in our project.

The second most useful thing would be learning about timers and their use in providing PWM signals to devices. This understanding of how the chip interacts with the world through the use of timers and voltages was very helpful in the sensory array which we employed for the project, but also the motors and data transmissions as well. These were used extensively to tune the servo motor's functionality, as well as working to ensure a robust communication between all of the sensor's incoming data and packaging them into relevant packets for the computer to interpret.

The third most useful thing would be how to control a microcontroller with the firmware itself. Given that the majority of this project relied on the MSP430 board as a controller for all of its main functions, being able to actually code the processor's behavior to how we desired. This allowed us to build the project to our desired traits and specifications.

3. The limits of our knowledge as Mechatronics engineers are based around our familiarity with various sensors and the MSP430 board itself. While we are still students, there are far more sensors and a large variety of actuators we have yet to truly use and implement.

   We are also relatively unfamiliar with control systems being applied to these smaller types of systems. We wanted to have a self-heating element within the beehive to assist with temperature variations, but due to the lack of knowledge we had implementing control systems by using the MSP board, we were unable to do so.

   Three things we would like to learn going forward:

   1. *Machine Learning/Computer Vision*
      a. Our strategy to acquire knowledge in these areas would include taking courses such as MANU 465, which teach the concepts of machine learning and computer vision. Moreover, integrating computer vision into capstone or other course projects would help facilitate our learning within these subjects. Finally, doing projects on our own through online resources would certainly help build experience in these areas.
   2. *Control Theory*
      a. Our strategy to acquire more knowledge in these areas would include taking courses such as MECH 468, Modern Control Theory, to get a better

understanding of control systems in a theoretical sense. Unfortunately, there are not many ways to get hands-on controls experience, but looking forward in our coursework, MECH 421 appears to present a unique opportunity to allow us to apply controls further with sensors and other actuators we have been learning about in our courses thus far. There could even be opportunities to work in labs throughout higher education which focus on applying the theories learned to practical problems. Moreover, taking on personal projects or guided projects on our own time would be another way to acquire knowledge.

3. *Data communication protocols*
    a. One of the most important parts of a mechatronic system is to communicate data between various devices through various interfaces. While we were able to do this partly using the UART in class, we would like to seek more ways to do it through bluetooth, or even IOT. One of the ways we can go about doing this is through self-directed learning courses such as IOT and Smart Devices courses on Coursera. Moreover, we could use resources such as instructables to help develop our skills in this area by using a hands-on approach to the problem.

Appendix A

C# GUI Code

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.IO.Ports;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Windows.Forms.DataVisualization.Charting;
```

```
namespace SmartBeeHiveInterface
{
    public partial class Form1 : Form
    {
        //Collect UART data
        ConcurrentQueue<Int32> databyte = new ConcurrentQueue<Int32>();
        int counter = 0; //Counts the number of packets
        int is255 = 0; //Switch Case conditional controller

        //Queues for data
        ConcurrentQueue<Int32> xval = new ConcurrentQueue<Int32>();
        ConcurrentQueue<Int32> yval = new ConcurrentQueue<Int32>();
        ConcurrentQueue<Int32> zval = new ConcurrentQueue<Int32>();
        ConcurrentQueue<Int32> tempval = new ConcurrentQueue<Int32>();
        ConcurrentQueue<Int32> weightval = new ConcurrentQueue<Int32>();

        //Vars for data
        double weightzero = 0;
        int avgsamplerate = 10;
        int critmass = 30; //Critical harvesting mass
        double wcalib = 0.375; //Calib value for weight
        int tcalib = 2; //Calib value for temp
        int xoperating = 125; // operating x-orientation
        int yoperating = 153; // operating y-orientation
        int zoperating = 130; // operating z-orientation
        int xyz_diff_threshold = 15; // difference between actual x/y/z-avg
and x/y/z-operating
        double tempscale = -1.1845; // calibrated temperature conversion
value
        int tempoffset = 139; // calibrated temperature conversion value //
139
        int temp_hot_threshold = 50;
        int temp_cold_threshold = 0;
        int x = 0;
        int weightavg = 0;
        int tempavg = 0;
        int xavg = 0;
        int yavg = 0;
        int zavg = 0;
        bool hive_hot = false;
```

```
        bool hive_cold = false;
        bool hive_tipped = false;
        bool hive_full = false;
        bool symbols_visible = false;

        // ----- SERIES FOR DATA PLOTTING -----
        Series weightdata = new Series();
        Series tempdata = new Series();

        // ----- FOR WRITINNG TO CSV -----
        string path = @"C:\Users\jcast\Desktop\values.csv";
        string delim = ",";
        StringBuilder csvout = new StringBuilder();

        public Form1()
        {
            InitializeComponent();
        }

        private void DataReceivedHandler(object sender,
SerialDataReceivedEventArgs e)
        {
            int bytesToRead = serialPort1.BytesToRead;
            while (bytesToRead != 0)
            {
                int newByte = serialPort1.ReadByte();
                databyte.Enqueue(newByte);
                bytesToRead = serialPort1.BytesToRead;
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            serialPort1.PortName = "COM3";
            serialPort1.DataReceived += new
SerialDataReceivedEventHandler(DataReceivedHandler);
            timer1.Tick += new EventHandler(timer1_tick);
            timer1.Interval = 1000;
            timer1.Enabled = true;
            timer1_tick(null, null);
```

```
            timer2.Tick += new EventHandler(timer2_Tick);
            timer2.Interval = 1000;
            timer2.Enabled = true;
            flashingSymbolTimer.Interval = 500;
            flashingSymbolTimer.Enabled = true;

            //Weight Chart Init
            WeightChart.Series.Add(weightdata);
            weightdata.Name = "Weight Data";
            weightdata.ChartType = SeriesChartType.Point;
            weightdata.BorderWidth = 2;
            weightdata.Color = Color.Purple;
            weightdata.XValueType = ChartValueType.Double;
            weightdata.YValueType = ChartValueType.Double;

            //Temp Chart Init
            TempChart.Series.Add(tempdata);
            tempdata.Name = "Temperature Data";
            tempdata.ChartType = SeriesChartType.Point;
            tempdata.BorderWidth = 2;
            tempdata.Color = Color.Green;
            tempdata.XValueType = ChartValueType.Double;
            tempdata.YValueType = ChartValueType.Double;

            // start with hive warnings hidden
            statlabel.Visible = false;
            temphivestat.Visible = false;
            orientationstat.Visible = false;
            tempUpdatedLabel.Visible = false;

            // start with symbols hidden
            honeyPicBox.Visible = false;
            hotPicBox.Visible = false;
            coldPicBox.Visible = false;
            oriPicBox.Visible = false;
        }

        private void timer1_tick(object sender, EventArgs e)
        {
            if (serialPort1.IsOpen)
```

```
            {
                while (databyte.TryDequeue(out int valfromq))
                {
                    // state machine
                    switch (is255)
                    {
                        case 0:
                            if (valfromq == 255) { is255 = 1; }
                            break;
                        case 1:
                            EnqueueWithQueueLimit(xval, valfromq);
                            if (xval.Count == avgsamplerate) xavg =
AverageIntFromQueue(xval);
                            is255 = 2;
                            break;
                        case 2:
                            EnqueueWithQueueLimit(yval, valfromq);
                            if (yval.Count == avgsamplerate) yavg =
AverageIntFromQueue(yval);
                            is255 = 3;
                            break;
                        case 3:
                            EnqueueWithQueueLimit(zval, valfromq);
                            if (zval.Count == avgsamplerate) zavg =
AverageIntFromQueue(zval);
                            is255 = 4;
                            break;
                        case 4:
                            EnqueueWithQueueLimit(tempval, valfromq);
                            if (tempval.Count == avgsamplerate) tempavg =
AverageIntFromQueue(tempval);
                            is255 = 5;
                            break;
                        case 5:
                            EnqueueWithQueueLimit(weightval, valfromq);
                            if (weightval.Count == avgsamplerate) weightavg
= AverageIntFromQueue(weightval);
                            is255 = 0;
                            counter++; // increments here, so 1 counter
increment == 1 full packet
```

```
                            break;

                    }
                }
            }
        }

        private void plotter(double weight, int temp)
        {
            //Store values into CSV
            csvout.AppendLine(weight.ToString() + delim + temp.ToString());
            File.AppendAllText(path, csvout.ToString());

            // only plot 100 datapoints
            if (weightdata.Points.Count() > 100)
weightdata.Points.RemoveAt(0);
            if (tempdata.Points.Count() > 100) tempdata.Points.RemoveAt(0);

            // actual plotting
            weightdata.Points.AddXY(x, weight);
            tempdata.Points.AddXY(x, temp);
            WeightChart.ResetAutoValues();
            TempChart.ResetAutoValues();
            x++;
        }

        private void ConBut_MouseClick(object sender, MouseEventArgs e)
        {
            if (!serialPort1.IsOpen)
            {
                serialPort1.Open();
                ConBut.Text = "Beehive Connected";
            }
            else
            {
                serialPort1.Close();
                ConBut.Text = "Connect Beehive";
            }
        }
```

```
        private void ZeroBut_MouseClick(object sender, MouseEventArgs e)
        {
            weightzero = weightavg * wcalib;
        }

        private void timer2_Tick(object sender, EventArgs e)
        {
            if (!serialPort1.IsOpen) return;

            double convertedweight = weightavg * wcalib - weightzero;
            int convertedtemp = (int)((tempavg - tempoffset) / tempscale);

            string sugtextboxbuffer = String.Empty;

            // WEIGHT
            weightlabel.Text = convertedweight.ToString() + " KG";
            if (weightavg > 55)
            {
                statlabel.Text = "HIVE FULL";
                statlabel.Visible = true;
                sugtextboxbuffer += "Harvest suggested, possible invasion
occurring\r\n\r\n";
                hive_full = true;
            }
            else
            {
                statlabel.Visible = false;
                hive_full = false;
            }

            // TEMPERATURE
            templabel.Text = convertedtemp.ToString() + " C";
            if (convertedtemp > temp_hot_threshold)
            {
                temphivestat.Text = "HIVE TOO HOT";
                temphivestat.Visible = true;
                sugtextboxbuffer += "Move box to shade or increase
ventillation. Possible swarming my be occuring, keep an ear out!\r\n\r\n";
                hive_hot = true;
                hive_cold = false;
```

```
            }
            else if (convertedtemp < temp_cold_threshold)
            {
                temphivestat.Text = "HIVE TOO COLD";
                temphivestat.Visible = true;
                sugtextboxbuffer += "Wrap hive with insulation\r\n\r\n";
                hive_cold = true;
                hive_hot = false;
            }
            else
            {
                temphivestat.Visible = false;
                hive_hot = false;
                hive_cold = false;
            }

            // ORIENTATION
            if ((Math.Abs(xavg - xoperating) > xyz_diff_threshold) ||
                (Math.Abs(yavg - yoperating) > xyz_diff_threshold) ||
                (Math.Abs(zavg - zoperating) > xyz_diff_threshold))
            {
                orientationstat.Text = "HIVE MOVEMENT DETECTED";
                orientationstat.Visible = true;
                sugtextboxbuffer += "Hive has experienced high amounts of
shock, check if animals have distrubed your hive, or if the hive is still
there\r\n\r\n";
                hive_tipped = true;
            }
            else
            {
                orientationstat.Visible = false;
                hive_tipped = false;
            }

            sugtextbox.Text = sugtextboxbuffer; // updated sugtextbox all
at once

            plotter(convertedweight, convertedtemp);
        }
```

```
        private void EnqueueWithQueueLimit(ConcurrentQueue<Int32> _q, int
val)
        {
            int queue_limit = avgsamplerate;
            _q.Enqueue(val);

            while (_q.Count > queue_limit)
            {
                _q.TryDequeue(out _); // dequeue if there are more elements
than the queue limit
            }
        }

        private int AverageIntFromQueue(ConcurrentQueue<Int32> _q)
        {
            int sum = 0;
            int temp;
            int queue_count = _q.Count;

            while (_q.TryDequeue(out temp))
            {
                sum += temp;
            }

            int avg = (int)((double)sum / queue_count);

            return avg;
        }

        private void tempColdButton_MouseClick(object sender,
MouseEventArgs e)
        {
            if (Int32.TryParse(tempBox.Text, out int new_cold_temp))
            {
                temp_cold_threshold = new_cold_temp;
                tempBox.Text = "";
                tempUpdatedLabel.Text = "Cold threshold updated to " +
new_cold_temp.ToString() + " C.";
                tempUpdatedLabel.Visible = true;
            }
```

```
            else
            {
                tempUpdatedLabel.Text = "Invalid input for setting new cold
threshold.";
                tempUpdatedLabel.Visible = true;
            }
        }

        private void tempHotButton_MouseClick(object sender, MouseEventArgs
e)
        {
            if (Int32.TryParse(tempBox.Text, out int new_hot_temp))
            {
                temp_hot_threshold = new_hot_temp;
                tempBox.Text = "";
                tempUpdatedLabel.Text = "Hot threshold updated to " +
new_hot_temp.ToString() + " C.";
                tempUpdatedLabel.Visible = true;
            }
            else
            {
                tempUpdatedLabel.Text = "Invalid input for setting new hot
threshold.";
                tempUpdatedLabel.Visible = true;
            }
        }

        private void flashingSymbolTimer_Tick(object sender, EventArgs e)
        {
            if (symbols_visible)
            {
                hotPicBox.Visible = false;
                coldPicBox.Visible = false;
                honeyPicBox.Visible = false;
                oriPicBox.Visible = false;
                symbols_visible = false;
            }
            else
            {
                if (hive_hot) hotPicBox.Visible = true;
```

```
                if (hive_cold) coldPicBox.Visible = true;
                if (hive_full) honeyPicBox.Visible = true;
                if (hive_tipped) oriPicBox.Visible = true;
                symbols_visible = true;
            }
        }
    }
}
```

MSP430 C Code

```
#include <msp430.h>
#include <stdio.h>

/**
 * main.c
 */
void configureCS(void);
void configureUART(void);
void configureTimer(void);
void configureMiscPins(void);
void enableInterrupts(void);
void configureSwitch(void);
void powerADC(void);
void configureADC(void);

volatile unsigned accelX;
volatile unsigned accelY;
volatile unsigned accelZ;
volatile unsigned loadCellReading;

unsigned int ADC_Result;
const unsigned int offset = 85;
int i=0;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer

    configureCS();
    configureUART();
```

```
    configureSwitch();
    powerADC();
    configureADC();
    configureTimer();
    configureMiscPins();
    enableInterrupts();

    while(1) {
    }
}

void sample_ADC_accel(void)
{
    // sample A12
    ADC10CTL0 &= ~(ADC10ENC + ADC10SC);       // disable and stop
conversion to change channel
    ADC10MCTL0 = ADC10INCH_12;                // change input channel to
A12
    ADC10CTL0 |= ADC10ENC + ADC10SC;          // enable and start
conversion
    while (ADC10CTL1 & ADC10BUSY);            // wait for conversion to
finish
    ADC10CTL0 &= ~ADC10SC;                    // stop conversion
    accelX = ADC10MEM0 >> 2;

    // sample A13
    ADC10CTL0 &= ~(ADC10ENC + ADC10SC);       // disable and stop
conversion to change channel
    ADC10MCTL0 = ADC10INCH_13;                // change input channel to
A13
    ADC10CTL0 |= ADC10ENC + ADC10SC;          // enable and start
conversion
    while (ADC10CTL1 & ADC10BUSY);            // wait for conversion to
finish
    ADC10CTL0 &= ~ADC10SC;                    // stop conversion
    accelY = ADC10MEM0 >> 2;

    // sample A14
    ADC10CTL0 &= ~(ADC10ENC + ADC10SC);       // disable and stop
conversion to change channel
```

```
    ADC10MCTL0 = ADC10INCH_14;                // change input channel to
A14
    ADC10CTL0 |= ADC10ENC + ADC10SC;          // enable and start
conversion
    while (ADC10CTL1 & ADC10BUSY);            // wait for conversion to
finish
    ADC10CTL0 &= ~ADC10SC;                    // stop conversion
    accelZ = ADC10MEM0 >> 2;

    // sample A4
    ADC10CTL0 &= ~(ADC10ENC + ADC10SC);       // disable and stop
conversion to change channel
    ADC10MCTL0 = ADC10INCH_4;                 // change input channel to A4
    ADC10CTL0 |= ADC10ENC + ADC10SC;          // enable and start
conversion
    while (ADC10CTL1 & ADC10BUSY);            // wait for conversion to
finish
    ADC10CTL0 &= ~ADC10SC;                    // stop conversion
    ADC_Result = ADC10MEM0 >> 1;

    // sample A0
    ADC10CTL0 &= ~(ADC10ENC + ADC10SC);       // disable and stop
conversion to change channel
    ADC10MCTL0 = ADC10INCH_0;                 // change input channel to A0
    ADC10CTL0 |= ADC10ENC + ADC10SC;          // enable and start
conversion
    while (ADC10CTL1 & ADC10BUSY);            // wait for conversion to
finish
    ADC10CTL0 &= ~ADC10SC;                    // stop conversion
    loadCellReading = ADC10MEM0 >> 2;

    // CALIBRATED VALUE IS loadCellReading > 55.
    if (loadCellReading > 55)
    {
        // turn on LED
        P3OUT |= BIT5;
        // change timer value
        TB1CCR1 = 1500;
    }
    else
```

```
    {
        // turn off LED
        P3OUT &= ~(BIT5);
        // change timer value
        TB1CCR1 = 500;
    }
}

void transmit_UART_accel(void)
{
    UCA0TXBUF = 255; // transmit 255 first
    while ((UCA0IFG & UCTXIFG) == 0);

    UCA0TXBUF = accelX;
    while ((UCA0IFG & UCTXIFG) == 0);

    UCA0TXBUF = accelY;
    while ((UCA0IFG & UCTXIFG) == 0);

    UCA0TXBUF = accelZ;
    while ((UCA0IFG & UCTXIFG) == 0);

    UCA0TXBUF = ADC_Result;
    while ((UCA0IFG & UCTXIFG) == 0);

    UCA0TXBUF = loadCellReading;
    while ((UCA0IFG & UCTXIFG) == 0);
}

void configureCS(void)
{
    // User Guide pg 80
    CSCTL0 = 0xA500;                          // Write password to modify
CS registers
    CSCTL1 &= ~DCORSEL;                       // Set DCORSEL to 0 for
DCOFSEL
    CSCTL1 |= DCOFSEL_3;                      // Set DCOCLKto 8 MHz
    CSCTL2 |= SELA_3 + SELS_3 + SELM_3;       // ACLK, SMCLK, and MCLK
source is DCOCLK
    CSCTL3 |= DIVA_0 + DIVS_3 + DIVM_0;       // ACLK == SMCLK == MCLK ==
```

```
DCOCLK == 8 Mhz
}

void configureUART(void)
{
    // Configure UC0
    P2SEL0 &= ~(BIT1 + BIT0);
    P2SEL1 |= BIT0 + BIT1;

    // Configure UCA0
    UCA0CTLW0 = UCSSEL0; // use ACLK as clock source, User Guide page 495
    UCA0CTLW0 &= ~(UCPEN); // disable parity (default), User Guide page 495

    // set baud rate to 9600 @ 8MHz, User Guide page 490
    UCA0BRW = 52;
    UCA0MCTLW = 0x4900 + UCOS16 + UCBRF0;
}

void powerADC(void)
{
    P2OUT |= BIT7; // power accelerometer
    P2DIR |= BIT7;
}

void configureADC(void)
{
    // Configure ADC pins
    P1OUT |= BIT4; // DOES P1OUT=1 DO ANYTHING FOR ADC?
    P1SEL1 |= BIT4;
    P1SEL0 |= BIT4;

    // A12, A13, A14 SEL1=1, SEL0=1
    P3SEL1 |= BIT0 + BIT1 + BIT2;
    P3SEL0 |= BIT0 + BIT1 + BIT2;

    // A0 SEL1=1, SEL0=1
    P1SEL1 |= BIT0;
    P1SEL0 |= BIT0;

    // Configure rest of ADC10
```

```
    ADC10CTL0 |= ADC10SHT_2 + ADC10ON;        // ADC10ON, S&H=16 ADC clks
    ADC10CTL1 |= ADC10SHP;                    // ADCCLK = MODOSC; sampling
timer
    ADC10CTL2 |= ADC10RES;                    // 10-bit conversion results
    ADC10MCTL0 |= ADC10INCH_12 + ADC10SREF_1; // ADC input select - start
with A12; Vref=1.5V I think
}

void configureTimer(void)
{
    TA0CCR0 = 40000;                          // SMCLK ~ 1MHz default, so 1
timer increment is ~1us
    TA0CTL = TASSEL_1 + MC_1 + ID_3;                // TACLK = SMCLK, Up
mode
}

void configureMiscPins(void)
{
    P1DIR |= BIT6;
    P1SEL0 |= BIT6;
    P1SEL1 &= ~(BIT6);

    P3DIR |= BIT5;
    P3SEL0 &= ~(BIT5);
    P3SEL1 &= ~(BIT5);

    TB1CCR0 = 20000; // Timer overflow value
    TB1CCR1 = 500;
    TB1CCTL1 = OUTMOD_7;
    TB1CTL = TBSSEL_2 + MC_1; // SMCLK, UP mode
}


void configureSwitch(void)
{
    // User Guide page 314
    P4DIR &= ~(BIT0); // configure P4.0 as a digital input
    P4REN |= BIT0; // enable internal pull-up/down resistors for switch
(for P4.0) (User Guide page 315)
    P4OUT |= BIT0; // enable pull-up resistor for P4.0 (User Guide page
```

```
293)
    P4IES &= ~(BIT0); // P4IFG flag is set with low-to-high transition
(User Guide page 316)
}

void enableInterrupts(void)
{
    UCA0IE |= UCRXIE; // enable Receive Interrupt, User Guide page 502
    TA0CCTL0 |= CCIE;
    TA0CTL |= TAIE;
    _EINT(); // global interrupt enable
    P4IE = BIT0; // enable P4.0 interrupt
    P4IFG &= ~(BIT0); // P4.0 IFG cleared
}

// Port 4 ISR
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=PORT4_VECTOR
__interrupt void Port_4(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(PORT4_VECTOR))) Port_4 (void)
#else
#error Compiler not supported!
#endif
{
    P4IFG &= ~(BIT0); // clear P4.0 IFG
    TB1CCR1 = 1500;
     //CCR1 PWM Duty Cycle, Minimum is around 490-500, Maximum is at around
2600
    TB1CCTL1 = OUTMOD_7;  //CCR1 selection reset-set
    TB1CTL = TBSSEL_2 | MC_1;
    PJOUT ^= (BIT0); // LED toggle for debugging purposes

    __bic_SR_register_on_exit(LPM4_bits); // exit LPM4
}

//ADC ISR
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=TIMER0_A0_VECTOR
__interrupt void TA0_ISR(void)
```

```
#elif defined(__GNUC__)
void __attribute__ ((interrupt(TIMER0_A0_VECTOR))) TA0_ISR (void)
#else
#error Compiler not supported!
#endif
{
  sample_ADC_accel();
  transmit_UART_accel();
  TA0CTL &= ~TAIFG;
}
```