IoT Enabled Smart Inventory Design Document

`

Adil Saldanha | Ammar Rehan | Melika Salehi | Wency Go

April 8, 2020

Group 110

Contents

List of Figures

List of Tables

Change log

Date	Author	Summary of change	Section pointer
Jan. 7, 2020	M.S.	Validation replaced by bi-directional communication. Request-response protocol.	Edge Device section
Jan. 15, 2020	A.S.	Weight requirement scaled down to 780g	Load cell section
Jan. 22, 2020	M.S.	Power Bi information added	Edge Device section
Jan. 29, 2020	A.S.	Load cell calibration method added	Load cell section
Feb. 04, 2020	A.S.	Load Cell Setup design added	Load cell section
Feb. 04, 2020	A.S.	Update Arduino code to calibrate and measure weight using load cell setup	Load cell section
Feb. 07, 2020	A.R.	Edge Device configuration code added	Appendix
Feb. 08, 2020	A.R.	Edge Device configuration- testing and validation cases added	Appendix
Feb 09, 2020	A.R.	Edge-to-End Communication section added	Edge-to-End Communication section
April 7, 2020	A.S.	Changes made to Load cell dock design and pressure plate	Load Cell
April 7, 2020	A.S.	Calibration code updated and algorithm added	Load Cell
April 7, 2020	A.S.	Container design and operation added	Container Design
April 7, 2020	A.S.	Sizing documentation added	Container Design
April 8, 2020	A.S.	References added for Xbee and Load Cell	

1.0 Overview

The IoT-enabled Smart Inventory tracks inventory by collecting data in the form of product weight, transmitting this data from end devices to an edge device, which finally sends this data to the cloud, where the user may then access their inventory data. The requirements, constraints, and goals (RCGs) for the project are available in Appendix A and can be referenced with respect to our design rationale.

Two key terms used in this document are the 'end device' and 'edge device'. In Internet of Things (IoT) infrastructure, an **end device** is primarily used to collect data and transmit this data through some form of low bandwidth communication protocol for low power consumption and size constraints. The **edge device** essentially acts as a middleman, or router, to receive data from the end device and then send this data to the cloud, where developers and users may access the data.

The three major components of our system are as follows:

1. End Device - Arduino Uno connected with load sensor(s) and a Zigbee Transceiver.
2. Edge Device - Raspberry Pi 3+ with a Zigbee Transceiver and WiFi adapter.
3. Cloud Computing - Microsoft Azure IoT Hub, Microsoft Power Bi

Each of these components will be explained in more detail in their own section of this document as well as the design rationale for choosing these components.

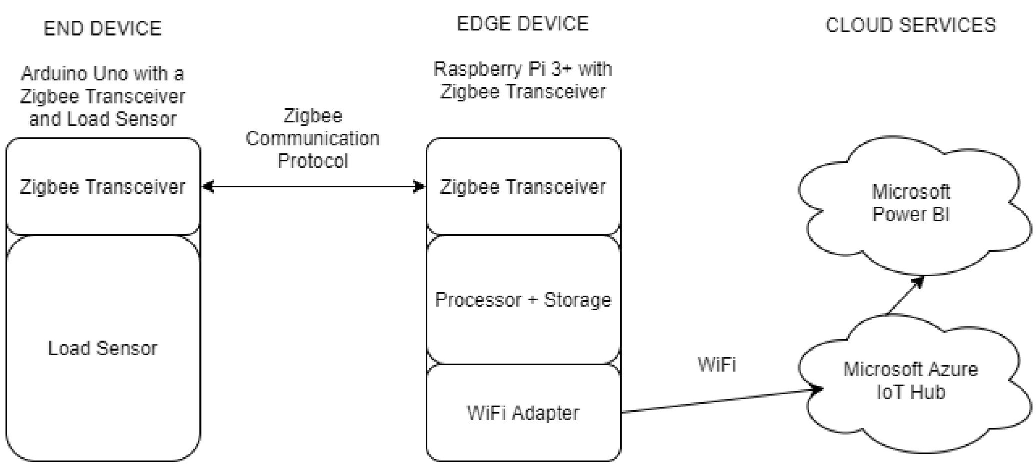


Figure 1 System Architecture diagram outlining the major components and communication protocol between these components.

2.0 End Device

The main processing unit for our end device was chosen to be an Arduino Uno, connected with a micro load cell and an Xbee transceiver. The Arduino family of electronics conveniently provides a microprocessor, on-board memory, I/O pins, and an IDE, making it an easy choice for the design. Another important reason is that there is a Zigbee communication library available. Without software libraries, we would have to develop our own library for processing and transmitting data between end and edge devices which would significantly increase the time required for this project. In addition, most of our team is familiar with coding in Arduino so using the Arduino IDE will allow us to write and debug the code more easily than writing in other low-level programming languages, as well as code readability for future developers and stakeholders. We have also chosen the Arduino Uno for scalability purposes. The board also contains numerous pins where we can conveniently attach components such as the load sensor and the Xbee transceiver. If, in the future, it is required to add more components, it is easily accomplished.

3.0 Load Cell

For our choice of weight sensor, we decided to go with a 780g micro load cell. The reasoning behind this was that it was the most accurate device for what it costs and has an appropriate range for our purpose. Other options such as strain gauges were more expensive or measured much higher masses which would induce more error. Although our original requirement was met be able to measure up to 2.25 kg, after a meeting with our client they stated that they will be satisfied with the 780g weight limit, so our current load cell is satisfactory. The earlier idea of installing multiple load cells would be mechanically very complicated, so we will not go down this path. One concern with this load cell is that the connection wires are not very strong so this will also have to be considered when doing the final design.

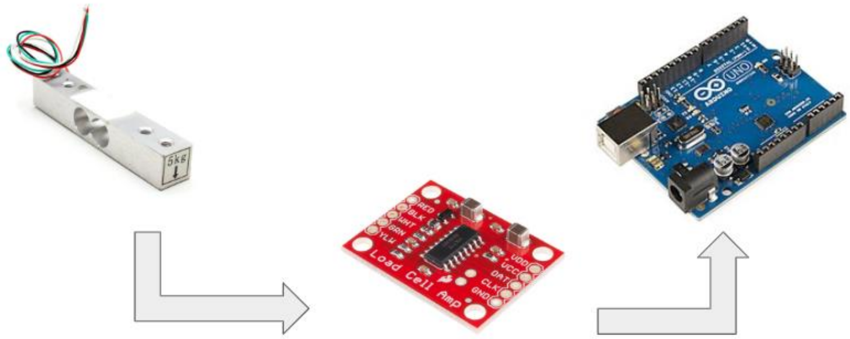


Figure 2 Load cell connection to HX711 board to Arduino Uno

For our setup, we decided to go with connecting the load cell to a HX711 board which is then connected to the Arduino. The HX711 board is an amplifier which reads the changes in resistance, amplifies them and sends the readings to the Arduino. One of the reasons we picked the HX711 is that there is a lot of existing support for the board such as Arduino libraries for reading and calibrating the load cell. Another reason is that the board is relatively inexpensive.

For our load cell setup, we took apart an existing weighing scale to examine how a load cell is setup so that we're not 'reinventing the wheel' and so that our setup is a as simple as possible. From our break down of an existing load cell we learnt 2 things.

1) The load cell is mounted on one end to a stationary point, and to a pressure plate which takes the weight of the items on the other end.
2) The load cell has spacers at either end where it's mounted as shown in Figure 3 so that all the pressure is on the mount.

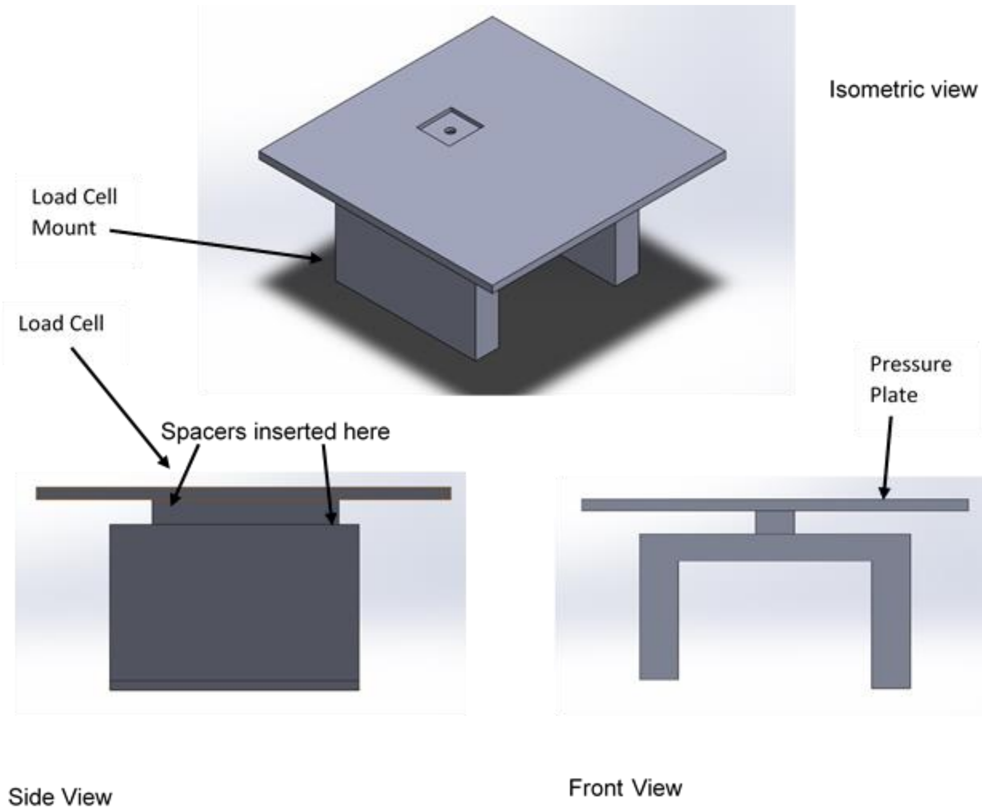


Figure 3 Pressure Plate and mount setup with load cell

Following the design of the existing load cell we attached the mount and pressure plate to either end of the load cell to gain the largest moment across the load cell. This would induce the largest voltage possible from the load cell which will improve our accuracy. Having spacers to make sure all the pressure is on the mounting point of the load cell ensures that wherever the weight is placed on the pressure plate the load cell measure the same weight, i.e. the load cell isn't measuring different weights at the center compared to one of the corners for the same item.

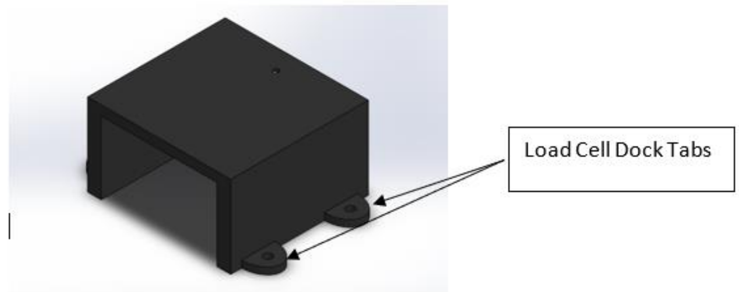


Figure 4 Addition of Load Cell Dock tabs

Tabs were added to the dock to mount to the base of the container using screws. There are 2 tabs on either side so that the dock does not shift around when the container is in use or being transported. This is vital as it will alter the calibration factor is the load cell dock moves position in the container when in use. The size of the pressure plate was chosen to be 15x15 cm, as this would provide enough area so that the container around the load cell wouldn't be too tall. The plate was sized using 10g smarties packets as a sample product. As a result the height of the container was sized to be 22 cm tall to fit 750g worth of smarties packets (75 packets). The complete dimensions for the load cell dock and pressure plate can be found in Appendix G.

3.1 Load cell calibration algorithm

The container calibrates its load cell once it is setup and in use. The process for the automated calibration is shown in figure 5. The algorithm for calculating the calibration factor (Figure 6) uses the product that is being stored in the container as a reference to calibrate the device. The user uses the pushbutton on the keypad to go through a series of steps to calibrate the load cell. Simultaneously when the user enters the weight and product ID for the item used to calibrate the load cell the container will record this information to send to the end device.

The following outlines an example of how the calibration steps would work, in this scenario user wants to store product A weighing 10g in the container.

1) The user presses a button to go to configuration mode where they place 1 of product A on the pressure plate. The microprocessor records this and stores it.

2) They push the button again and enter the product ID, product A.

3) They push the button again to go to the next stage and enter the product weight, 10g.

4) They push the button to go to calibration mode where the microprocessor uses the item of product A placed on the pressure plate to calculate a calibration value.

5) Once this value is calculated the container returns to normal operation mode, and now multiple items of product can be stored in the container.

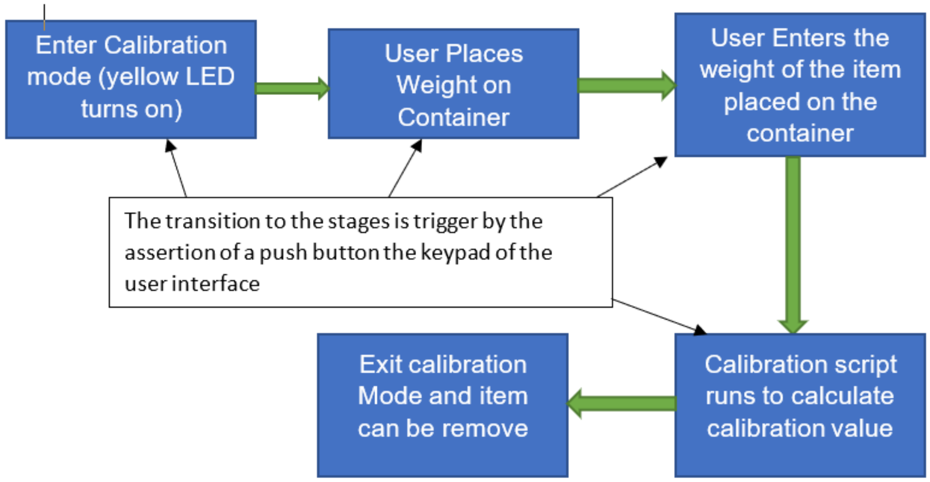


Figure 5 Process for automated calibration

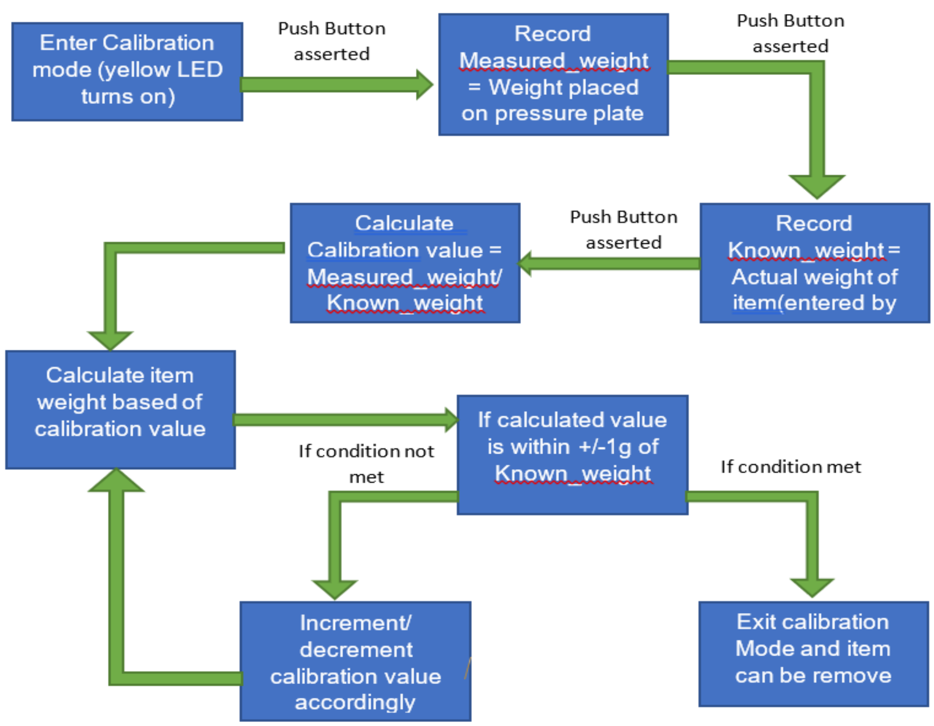


Figure 6 Algorithm for calculating the calibration factor

4.0 Smart inventory container (stretch goal)

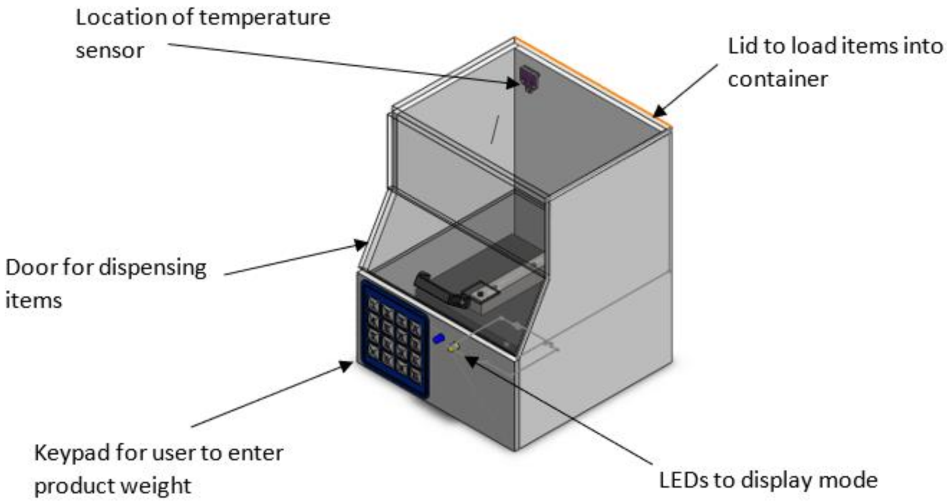
4.1 Container design

A simple gravity dispenser (Figure 7) was chosen for the container design. See appendix H for details and dimensions.

Items are loaded into the container using the lid on the top of the container. When in use, i.e. an item is purchased they can be removed from the container using the door at the front of the container. The keypad is used during configuration to enter product information such as the product ID and weight. The LEDs are used to display the mode of operation of the container:

- Blue is for normal use of operation; weighing the number of items inside and transmitting the information to the edge device.
- Yellow is for when the container is in configuration

The components of the end device are mounted at the base of the container, which causes the generated heat to impact the conditions within the container. Given retail application of the smart inventory, the container is likely to be used to store foodstuffs. In accordance with Food Safety Laws and Requirements, a temperature sensor was added to monitor conditions within.



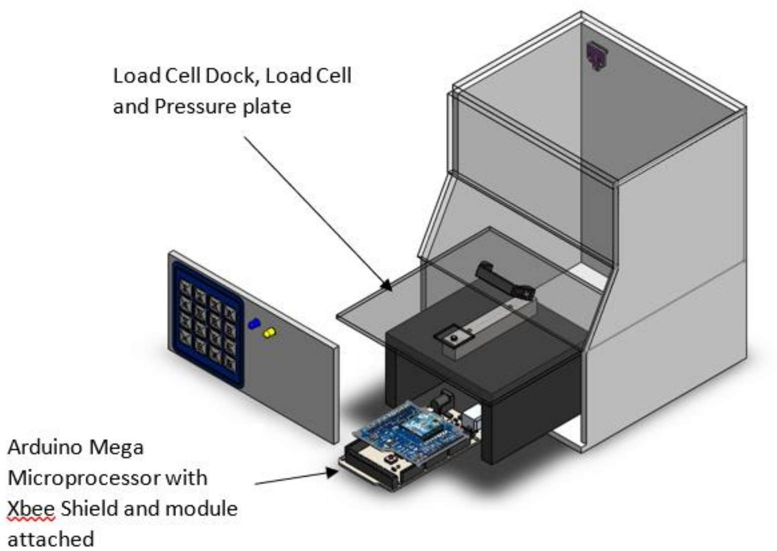


Figure 7 Storage container unit for smart inventory

4.2 Container Operation

The operation of the container can be described as a simple state machine (figure 8). Operation of States:

OFF: If the container is not plugged into power then the container will be int this state

NORMAL OPERATION: As soon as the container is plugged in it enters this mode, where is begins taking measurements from its sensors and transmitting this information to the edge device.

CONFIGURATION: In this mode the product information is entered on the container using the keypad. The container load cell is simultaneously calibrated during this process using the input product weight as a reference. After this the container switched back to normal operation where it starts sending the number of items in the container, product weight, and product ID.

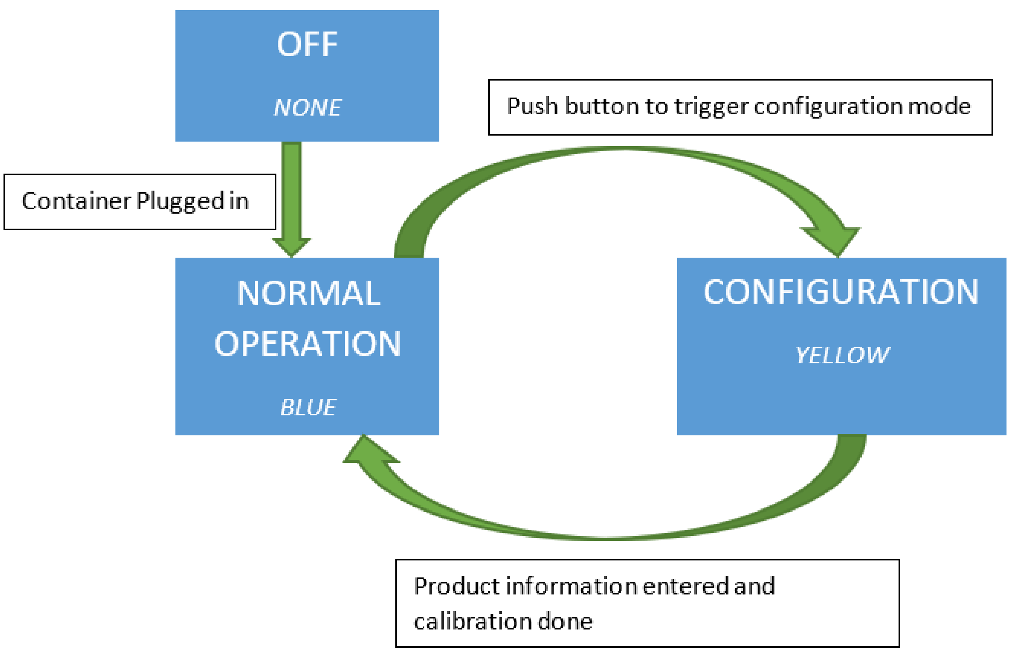


Figure 8 Container modes of operation flow diagram. Output is the color of the LED on the container

5.0 Edge Device

The edge device system architecture consists of 3 major modules: the Raspberry Pi 3 B+, XBee-Pro ZigBee communication module, and XBee serial interface. The parts, their purpose, and the design rationale are summarized in Table 1. The cloud supported tools Azure IoT Hub and Power Pi details can be seen in Table 1 & 2.

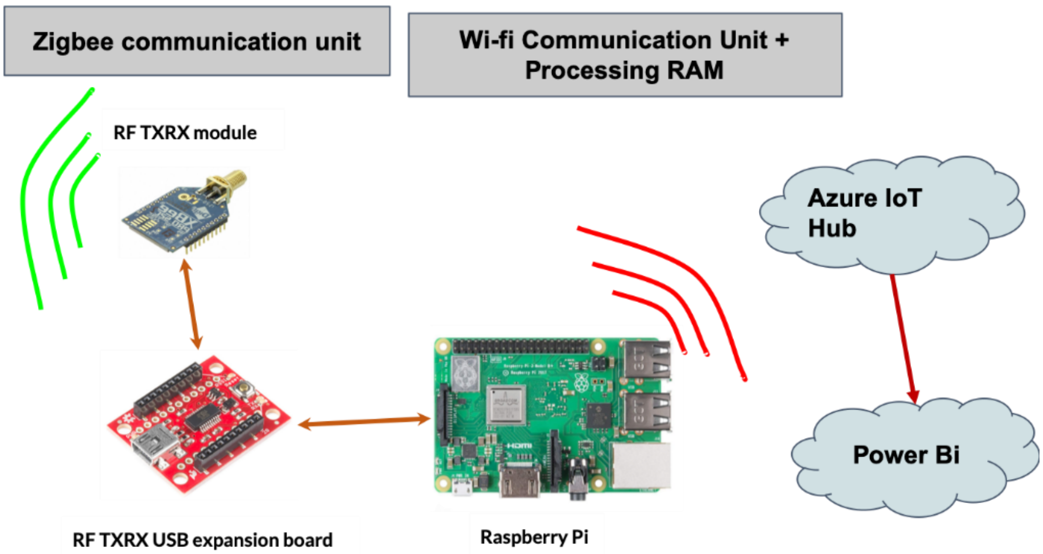


Figure 9 Edge device connectivity and communications

Table 1 Edge device system component details

COMPONENT	PURPOSE	DESIGN RATIONALE
Raspberry Pi 3 B+	Main processing unit. Acts as a host gateway for connecting to the Azure cloud	WiFi compatibility Recommended by Microsoft for IoT prototyping Official azure compatible package and documentation provided by Microsoft Available on-board ports Available Azure libraries Economical price
Xbee-Pro ZigBee communication module	RX/TX for Zigbee protocol, used for communicating with end devices.	The same model is being used on end device(s). Having the same model at both ends allows for a simpler, hassle free implementation.
Xbee Explorer USB	Serial interface between the XBee-Pro ZigBee module and the Raspberry Pi 3 B+	Available hardware interface for Raspberry Pi.

Table 2 Cloud system details

PLATFORM	PURPOSE	DESIGN RATIONALE
Azure IoT Hub	Cloud supported central message hub for bi-directional communication between edge device and cloud	Client uses Azure services for their POS system. They have enterprise contract with Microsoft
Power Bi	Live data stream and visualization. Real time report generation	Is a Microsoft supported platform. Seamlessly integrates Azure IoT Hub and reads from desired IoT Hub node

5.1 Edge System Software Architecture

The edge device will act as the host gateway between the end device and the cloud. The Rapsberry Pi, equipped with both the Xbee ZigBee communication module and a WiFi adapter, allows for receiving the end device data, pre-processing the data with its on-board processor, and finally submitting the pre-processed data to the cloud for further use.

Once the edge device sends the data to the cloud, we use Microsoft Azure IoT Hub to access the data for further use. The Microsoft Azure IoT Hub is an online, cloud database that contains information regarding each edge device connected to it. Each edge device is registered with a unique identifier. Once it is registered, the IoT Hub and the device can then transfer data between each other. The Microsoft Azure IoT Hub was chosen because of client requirements – their current system uses Microsoft Azure for their cloud computing, so it was chosen for scalability and integration purposes. In order to visualize data for analysis, the data is accessed real time by Microsoft Power Bi. Power Bi actively checks the IoT Hub node and pulls data. The end user can then visualize in various forms and edit if needed.

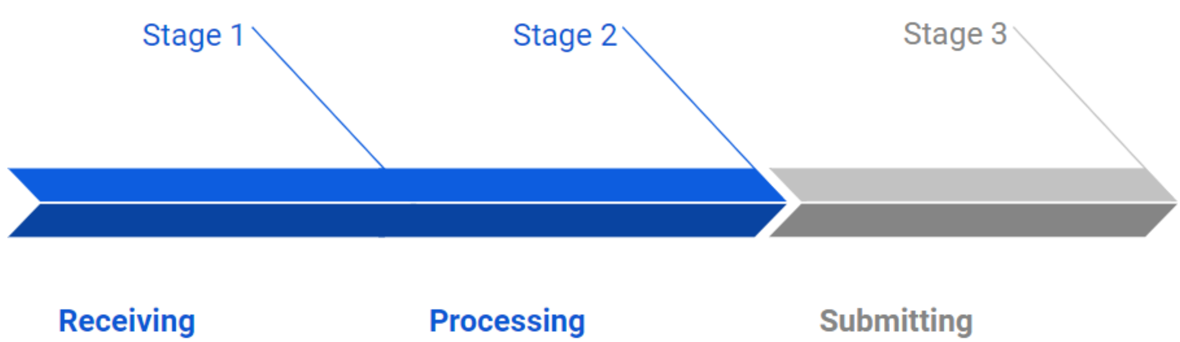


Figure 10 Edge device process flow

Table 3 Edge device software system architecture details

STAGE	PURPOSE	RATIONALE
ISR	The interrupt service routine prioritizes accepting incoming data.	Data submission to cloud can be done at any time, as the raspberry pi can hold the data in memory.
RECEPTION	Uses request-response protocol. Establish bi-directional communication with each end device.	The nature of data is not sensitive, and the received data is not encrypted. This allows more control over which end device you want to receive data from. Can also allow for prioritization and higher check frequency of specific end devices
PREPARATION	Cleans up the data string received and converts to json string	The data received will contain extra information due to the protocol being utilized. This extracts the information we care about. Json allows for seamless insertion.
SUBMISSION	Starts an API request to the cloud. Proceeds to submit data.	The Raspberry Pi is not in constant active connection with the Microsoft Azure IoT Hub. This is due to the limited number and duration of API requests that are given to cloud users, depending on the contract.

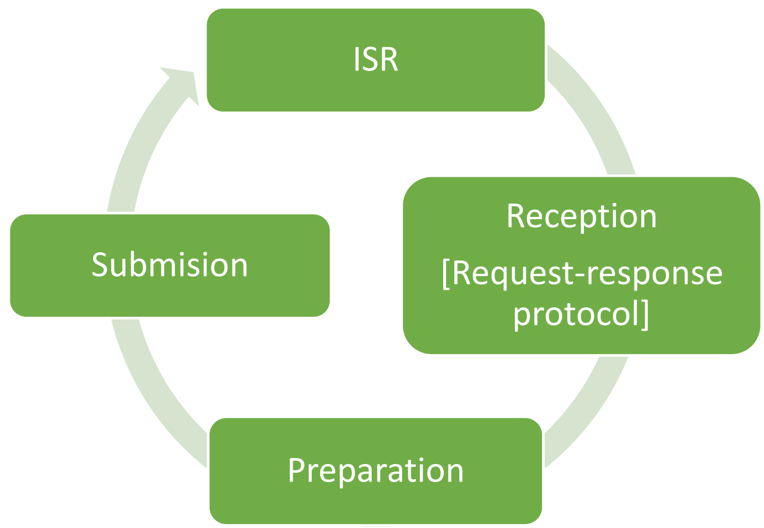


Figure 11 Data management cycle in the edge device

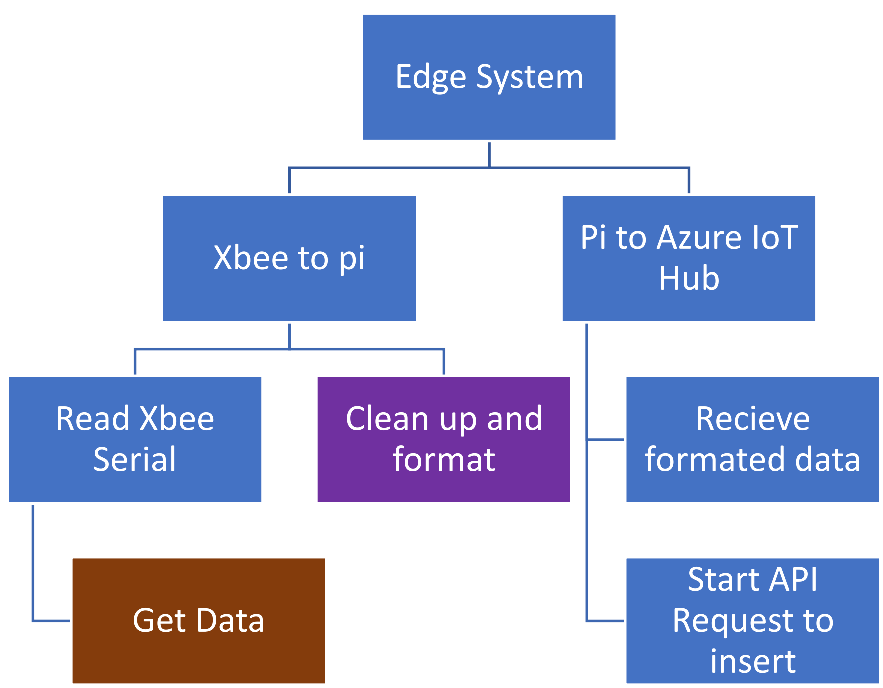


Figure 12 Top Level edge Software System

7.0 Edge-to-End communication

Zigbee was chosen over the alternatives due to its ideal range and its ability to create a Zigbee 'mesh network' so that a multitude of Zigbee-enabled devices can communicate with one another. Compared to Bluetooth, which has a much smaller range at around 10 meters, Zigbee can range up to 100 meters and would be able to operate in small- to medium-sized warehouses or stores and track multiple end devices at a time. Another alternative was LoRa technology, but its range was far too large, up to 2-3-kilometer coverage, which has the potential to create interference with other IoT-enabled Smart Inventory systems in that area.

We use the request-response protocol for communication between the edge and end devices (Figure 10). An edge device will have a list of connected end devices. It will ping an address, wait for the response, return confirmation, decode received data and upload the data along with other important parameters such as device ID to the cloud, before moving to the next connected device. See appendix for detailed architecture of this process.

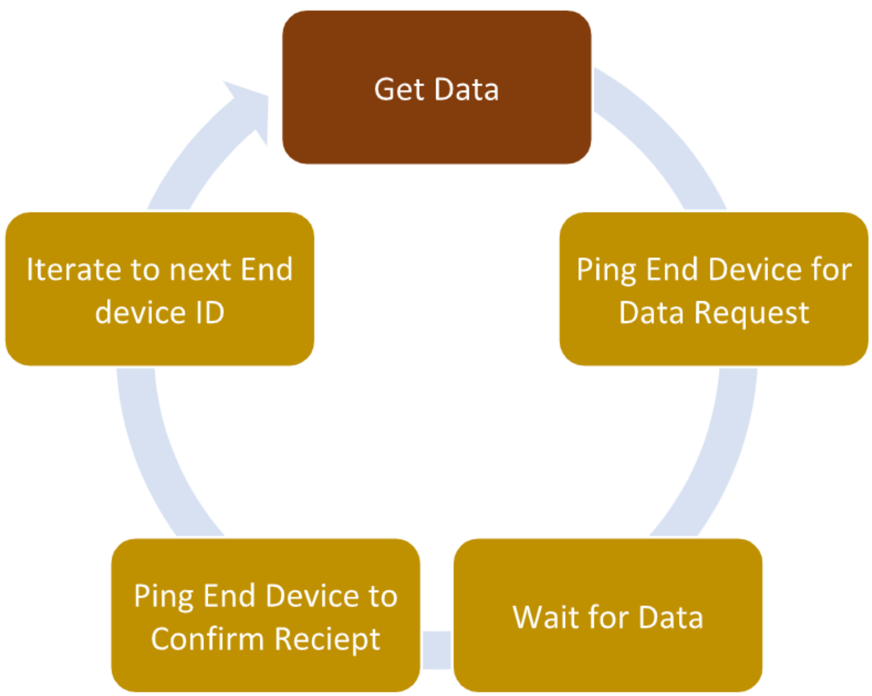


Figure 13 Edge to End Data exchange process

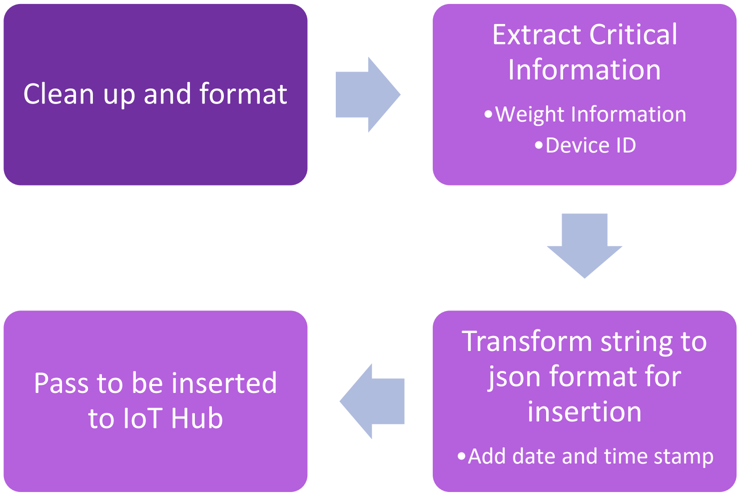


Figure 14 Data preparation process

As an edge device handles multiple end devices, we have a star topology with bi-directional communication. There are multiple benefits of this approach discussed below in further detail.

Table 4 Benefits of start topology for edge device

BENEFIT	RATIONALE
Reduced signal interference	End devices are not broadcasting continuously; there is low interference in the 2.4 GHz band. This makes communication between the devices more reliable and secure against loss of data packets. It has minimal impact on other wireless services like Wi-Fi that operates in the same frequency band.
Greater control	Edge device can select which end device to communicate to. The bi-directional communication allows for confirming the correct end device has been accessed. This can also help determine if an end device is malfunctioning Can protect against attempts of unauthorized access to end devices as they would only respond if a specific request is received.
Scalability	There will be an upper limit to how many end devices that can be added to the system. The system however can be easily transformed into a tree topology with multiple edge devices that would allow for virtually infinite scaling.
Low power	Broadcasts are only a result of requests, end devices can operate in a lower powered mode when not broadcasting, hence saving power. Additionally, Xbees have a sleep mode for power saving

Table 5 Drawbacks of star topology for edge device

DRAWBACK	RATIONALE
High dependency on the edge device	The edge device not only stores the address of all connected end devices but is also responsible for initiating the request-response protocol. If the edge device fails, live data updating would stop.
Limited capacity of the edge device	The request-response protocol requires at least three seconds to complete. Hence each device is only communicating for three seconds and then waits for a request. Time between requests will determine how frequently values are updated. Hence adding each end device to the network would increase time between requests by three seconds. Must move to tree topology if the number of devices introduces significant delay.

Appendix

Appendix A: Sample Code for Arduino ZigBee Communications

A sample base code written by our team for receiving data with an Xbee-connected Arduino with an Xbee module connected to a computer and sending data via XCTU software.

```
int analogPin = A0;
int val = 0;

void setup() {
  // initialize serial communication
  Serial.begin(9600);
}

void loop() {
  // read analog value at given pin
  val = analogRead(analogPin);
  //send read value at pin (calibration required)
  Serial.println(val);
  delay(1000);
}
```

Appendix B: Microsoft Sample Code for IoT Hub and Raspberry Pi

Sample code provided by Microsoft to test Rapsberry Pi communication with the Microsoft Azure IoT Hub with a small change to the device name to 'blueberry'.

```
/*
* IoT Hub Raspberry Pi C - Microsoft Sample Code - Copyright (c) 2017 -
Licensed MIT
*/
#include "./wiring.h"

static unsigned int BMEInitMark = 0;

#if SIMULATED_DATA
float random(int min, int max)
{
    int range = (int)(rand()) % (100 * (max - min));
    return min + (float)range / 100;
}

int readMessage(int messageId, char *payload)
{
    float temperature = random(20, 30);
    snprintf(payload,
             BUFFER_SIZE,
             "{ \"deviceId\": \"blueberry\", \"messageId\": %d,
\"temperature\": %f, \"humidity\": %f }",
             messageId,
             temperature,
             random(60, 80));
    return (temperature > TEMPERATURE_ALERT) ? 1 : 0;
}

#else
int mask_check(int check, int mask)
{
    return (check & mask) == mask;
}

// check whether the BMEInitMark's corresponding mark bit is set, if not, try
to invoke corresponding init()
int check_bme_init()
{
    // wiringPiSetup == 0 is successful
    if (mask_check(BMEInitMark, WIRINGPI_SETUP) != 1 && wiringPiSetup() != 0)
    {
        return -1;
    }
    BMEInitMark |= WIRINGPI_SETUP;

    // wiringPiSetup < 0 means error
    if (mask_check(BMEInitMark, SPI_SETUP) != 1 &&
wiringPiSPISetup(SPI_CHANNEL, SPI_CLOCK) < 0)
```

```
    {
        return -1;
    }
    BMEInitMark |= SPI_SETUP;

    // bme280_init == 1 is successful
    if (mask_check(BMEInitMark, BME_INIT) != 1 && bme280_init(SPI_CHANNEL) !=
1)
    {
        return -1;
    }
    BMEInitMark |= BME_INIT;
    return 1;
}

// check the BMEInitMark value is equal to the (WIRINGPI_SETUP | SPI_SETUP |
BME_INIT)

int readMessage(int messageId, char *payload)
{
    if (check_bme_init() != 1)
    {
        // setup failed
        return -1;
    }

    float temperature, humidity, pressure;
    if (bme280_read_sensors(&temperature, &pressure, &humidity) != 1)
    {
        return -1;
    }

    snprintf(payload,
             BUFFER_SIZE,
             "{ \"deviceId\": \"blueberry\", \"messageId\": %d,
\"temperature\": %f, \"humidity\": %f }",
             messageId,
             temperature,
             humidity);
    return temperature > TEMPERATURE_ALERT ? 1 : 0;
}
#endif

void blinkLED()
{
    digitalWrite(LED_PIN, HIGH);
    delay(100);
    digitalWrite(LED_PIN, LOW);
}

void setupWiring()
{
    if (wiringPiSetup() == 0)
    {
        BMEInitMark |= WIRINGPI_SETUP;
    }
    pinMode(LED_PIN, OUTPUT);)
```

Appendix C: Load cell test and calibration sample code

Load cell Source Code

```
Basic Load cell test:
Source: https://github.com/sparkfun/HX711-Load-Cell-Amplifier/tree/master/firmware


#include "HX711.h"


// HX711 circuit wiring
const int LOADCELL_DOUT_PIN = A0;
const int LOADCELL_SCK_PIN = A1;


HX711 scale;


void setup() {
  Serial.begin(57600);
  scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);
}


void loop() {


  if (scale.is_ready()) {
    long reading = scale.read();
    Serial.print("HX711 reading: ");
    Serial.println(reading);
  } else {
    Serial.println("HX711 not found.");
  }


  delay(1000);

}
```

Load cell calibration

```
* HX711 library for Arduino - example file
 * https://github.com/bogde/HX711
 *
 * MIT License
 * (c) 2018 Bogdan Necula
#include "HX711.h"


// HX711 circuit wiring
const int LOADCELL_DOUT_PIN = A5;
const int LOADCELL_SCK_PIN = A4;


HX711 scale;

void setup() {
  Serial.begin(38400);
  Serial.println("HX711 Demo");

  Serial.println("Initializing the scale");

  // Initialize library with data output pin, clock input pin and gain
factor.
  // Channel selection is made by passing the appropriate gain:
  // - With a gain factor of 64 or 128, channel A is selected
  // - With a gain factor of 32, channel B is selected
  // By omitting the gain factor parameter, the library
  // default "128" (Channel A) is used here.
  scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);

  Serial.println("Before setting up the scale:");
  Serial.print("read: \t\t");
  Serial.println(scale.read());                 // print a raw reading from
the ADC

  Serial.print("read average: \t\t");
  Serial.println(scale.read_average(20));       // print the average of 20
readings from the ADC

  Serial.print("get value: \t\t");
  Serial.println(scale.get_value(5));           // print the average of 5
readings from the ADC minus the tare weight (not set yet)

  Serial.print("get units: \t\t");
  Serial.println(scale.get_units(5), 1);  // print the average of 5 readings
from the ADC minus tare weight (not set) divided
                                    // by the SCALE parameter (not set yet)

  scale.set_scale(2280.f);                      // this value is obtained by
calibrating the scale with known weights; see the README for details --
original value = 2280
  scale.tare();                             // reset the scale to 0
```

```
  Serial.println("After setting up the scale:");

  Serial.print("read: \t\t");
  Serial.println(scale.read());                 // print a raw reading from
the ADC

  Serial.print("read average: \t\t");
  Serial.println(scale.read_average(20));       // print the average of 20
readings from the ADC

  Serial.print("get value: \t\t");
  Serial.println(scale.get_value(5));           // print the average of 5
readings from the ADC minus the tare weight, set with tare()

  Serial.print("get units: \t\t");
  Serial.println(scale.get_units(5), 1);        // print the average of 5
readings from the ADC minus tare weight, divided
                                    // by the SCALE parameter set with
set_scale

  Serial.println("Readings:");
}

void loop() {
  Serial.print("one reading:\t");
  Serial.print(scale.get_units(), 1);
  Serial.print("\t| average:\t");
  Serial.println(scale.get_units(10), 1);

  scale.power_down();                       // put the ADC in sleep mode
  delay(5000);
  scale.power_up();
}
```

Automated Calibration

```
#include <HX711.h>
// HX711 circuit wiring
const int LOADCELL_DOUT_PIN = A5;
const int LOADCELL_SCK_PIN = A4;
HX711 scale;
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial.println("HX711 Calibration");
  Serial.println("Calibrating");
}
   char temp = 0;
   float measured_weight = 0;
   int known_weight = 0;
   float calibration_value = 0;

void loop() {
  Serial.println("Set Scale:");
  scale.set_scale();  // Call `set_scale()` with no parameter.
  //void tare();       //Call `tare()` with no parameter.

//3. Place a known weight on the scale and call `get_units(10)`.
  Serial.println("Place known weight:");
  while ( temp != 'a' ){
    temp =  Serial.read();
  }
  Serial.println("Enter known weight:");
    known_weight  =  Serial.read();
  Serial.println(known_weight);

  measured_weight= (scale.get_units(10));
  Serial.println(measured_weight);
    temp = 0;
//4. Divide the result in step 3 to your known weight. You should
//   get about the parameter you need to pass to `set_scale()`
  calibration_value = measured_weight/known_weight;
  Serial.println(calibration_value);

    while (!((calibration_value*measured_weight-1) < measured_weight &&
(calibration_value*measured_weight+1) > measured_weight)) {
        if (  measured_weight < (calibration_value*measured_weight-1)  )
            calibration_value = calibration_value - 10;
        else if (measured_weight > (calibration_value*measured_weight+1)  )
          calibration_value = calibration_value + 10;    }  }
```

Appendix D: Microsoft Azure and Power Bi Web Interface

Microsoft Azure is a cloud computing service provided by Microsoft. One of the services we are particularly interested in is the IoT Hub, which is a cloud database that communicates with and tracks IoT devices. It has a developed web interface that allows the user to generate identifier keys for registering new IoT devices and metrics to display usage statistics and data being transferred.

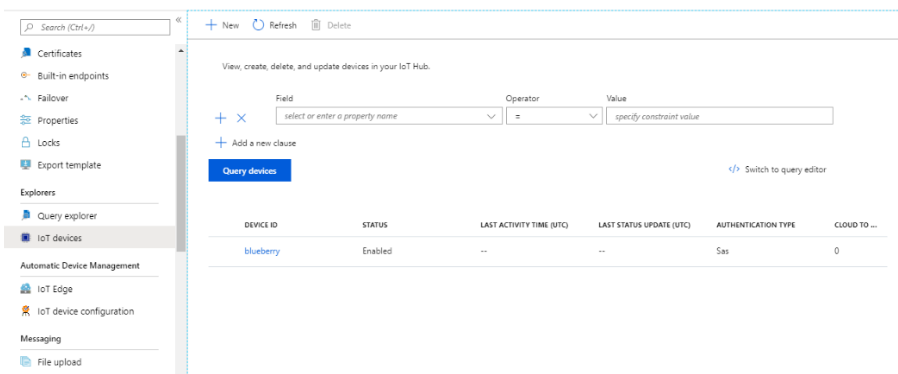


Figure 1: Microsoft Azure IoT Hub device list.

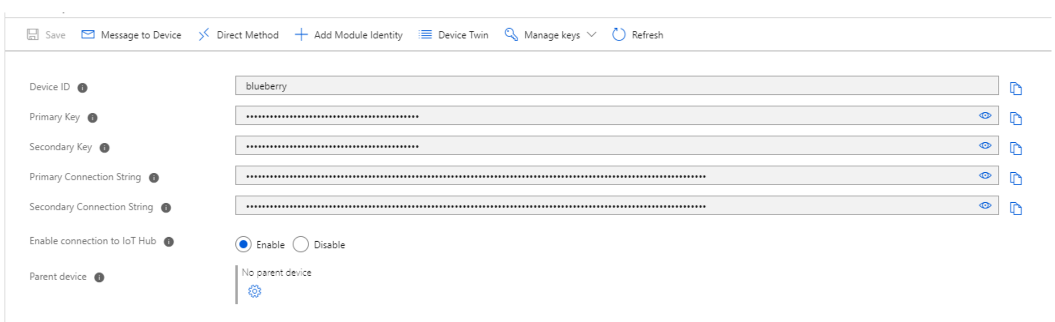


Figure 2: Microsoft Azure IoT Hub device configuration.

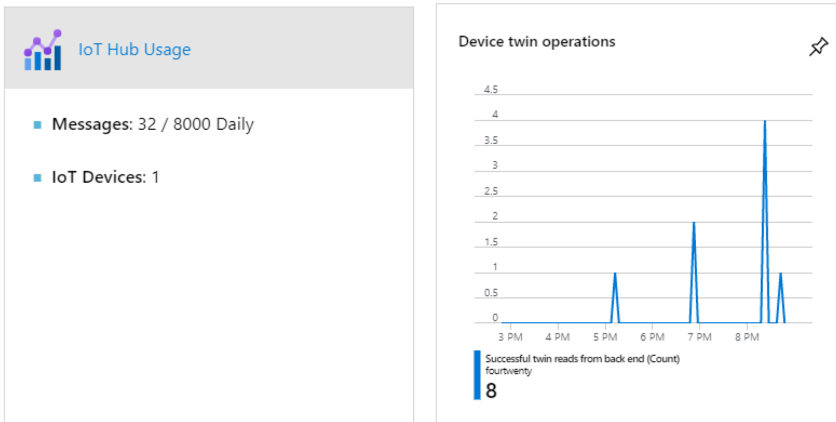


Figure 3: Microsoft Azure IoT Hub usage metrics.

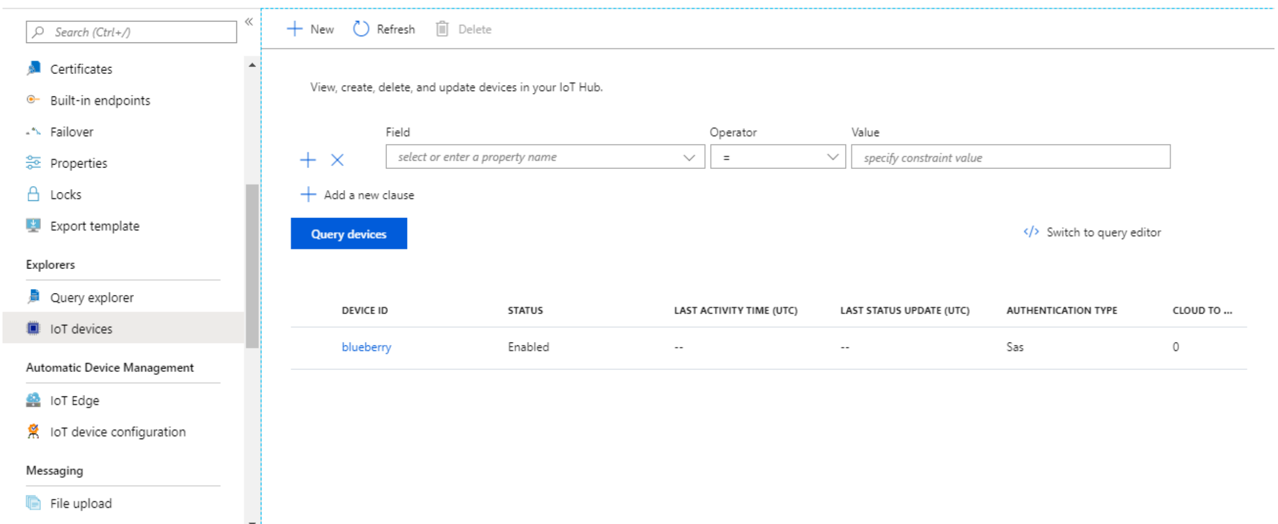


Figure 4 Microsoft Azure IoT Hub web interface with a sample device registered as ‘blueberry’

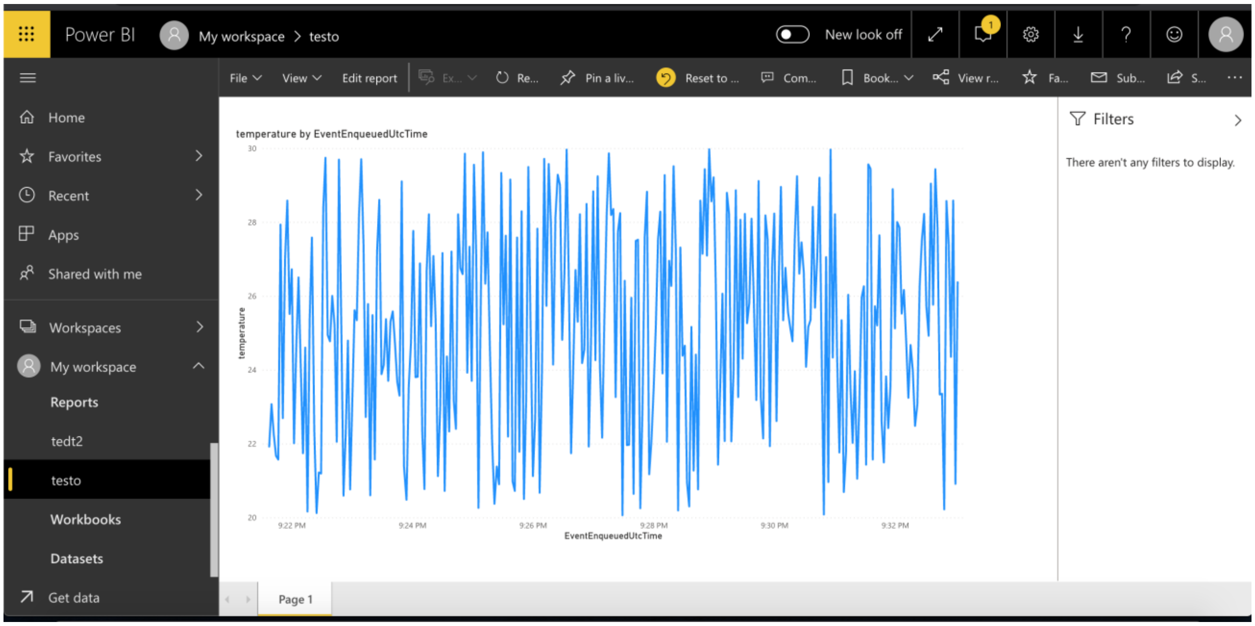


Figure 5 Microsoft Power Bi web interface with sample live data

Appendix E: End-to-edge communication code and architecture

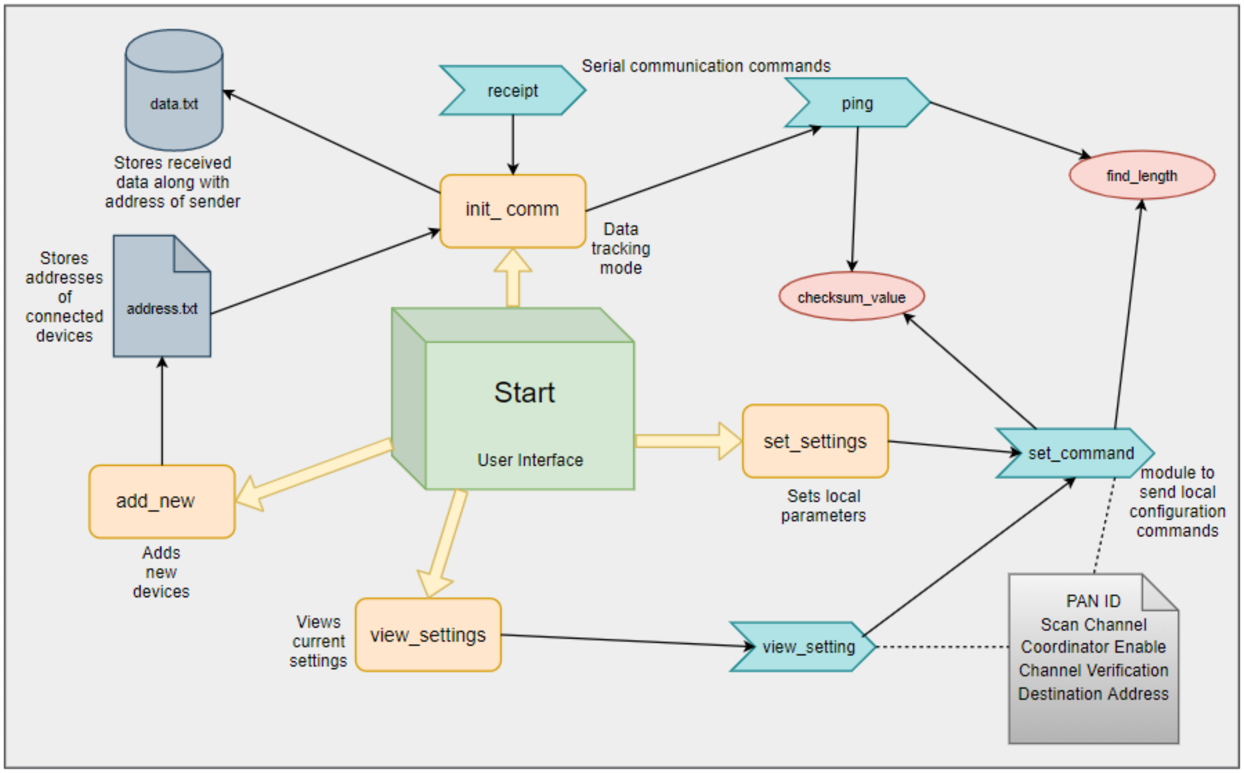
Operation

This code can do the following:

1) Initialize serial communication with XBee.
2) Send clean (or decoded) messages to and receive them from Arduino (received messages must start with !!! and end with ???)
3) Stores connected device addresses to text file and loops through them (multi-device communication) until stopped by interrupting kernel
4) Able to add new devices by adding their addresses to text file.
5) Can view and set local XBee parameters including PAN ID, scan channel, channel verification, coordinator enable and destination address.
6) Copies read data along with address of device data was received from onto a text file.
7) Includes UI to select what to do

For proper operation, code must be modified to set baud rate, serial port and paths of address text file and data text file.

Need to run start() only to enter UI. Here you can enter data tracking mode, view settings, change settings and add new devices. To remove devices, address text file must be modified. Running once will let you run any of these operations once, however we will stay in data tracking mode until kernel is interrupted, using KeyboardInterrupt or other. This diagram visualizes how code on the edge device achieves this.



The accompanying Arduino serial transceiver code provides a template to ping the end device and receive a response. A specific response is sent on receiving a specific request. This is used to test the code's functionality.

Arduino_serial_recieve

```
String starts, ends, initial, msg, check, sending;

void setup() {
  Serial.begin(9600);

  starts = String("!!!");
  ends = String("???");
  initial = String("incorrect code");
  msg = String("DATA");
  check = String("code");
}



void loop() {
  if(Serial.available() > 0) {
  //Serial.write(Serial.read());
   String str = Serial.readString();
   if (str == check) {
    sending = starts + msg + ends;
    }
    else {
     sending = starts + initial + ends;
     }
    Serial.print(sending);
    //Serial.print(str.length());
    }
}
```

Messages start and end with identifiers

Received code is checked and an appropriate response is given

COM5

```
??!!!DATA???!!!incorrect code???!!!i
```

Autoscroll Show timestamp

Edge Device - Python script

```
################################################################################
###
#import relevant libraries
import serial
import codecs
import copy
import time
import binascii as ba

#this initializes the port and serial communication, both parameters must be
strings
ser = serial.Serial() #initializing
ser.baudrate = 9600 #default baud rate
ser.port = 'COM4' #depends on port of RPi/computer
ser.timeout = 0 #sets timeout to 1 second
ser.open() #opens port

#------------------------------------------------------------------
# below are used modules for local commands
# baud rate and API mode must be pre-set using XCTU
#------------------------------------------------------------------------
def find_length(s): #finds hex length of string as byte string
      length_int = int(len(s)/2)
      if length_int < 16:
      hex1 = hex(length_int)
      hex1 = hex1[-1:]
      length = '000' + hex1
      elif length_int >= 16 and length_int < 256:
      hex1 = hex(length_int)
      hex1 = hex1[-2:]
      length = '00' + hex1
      elif length_int >= 256 and length_int < 4096:
      hex1 = hex(length_int)
      hex1 = hex1[-3:]
      length = '0' + hex1
      else:
      hex1 = hex(length_int)
      length = hex1[-4:]
      return length

#------------------------------------------------------------------------
def checksum_value(s): #finds checksum byte of instruction
      base = 0
      f = ""
      j = ""
      l = [s[k:k+2] for k in range (0, len(s), 2)] #this splits list into
blocks of 2
      for x in l: #this does interger addition of all bytes in list
      a = int(x, 16)
```

```
      base += a
      b = bin(base) #this converts sum into binary of type string
      for c in range(2, len(b)): #this removes 0b from start of string that
shows it is binary string
      f += b[c]
      x = list(f) #this splits string into list
      for i in range(0 , len(x)): #this takes 1's complement of binary list
      if x[i] == '1':
            x[i] = '0'
      else:
            x[i] = '1'
      y = "".join(x) #this joins the list into a string
      z = hex(int(y, 2)) #this converts the base-2 string to a hexadecimal
string
      for c in range(2, len(z)): #this removes 0x from hexamdecimal string
      j += z[c]
      return j[-2:] #this outputs only last 2 characters that give checksum
byte

#---------------------------------------------------------------------------
def set_command(msb, lsb, param): #forms command to set Channel Verification
as on or off
      initial = '7e'
      frame_type = '08'
      frame_id = '01'
      string = frame_type + frame_id + msb + lsb + param
      length = find_length(string)
      checksum = checksum_value(string)
      command = initial + length + string + checksum
      command = codecs.decode(command, "hex") #converts into command
      return command
#---------------------------------------------------------------------------
#below are modules for sending and receiving data
#---------------------------------------------------------------------------
def ping(addr, msg): #forms command to ping device of addr(str) with msg(str)
      s = msg.encode('utf-8')
      msg = s.hex()
      initial = '7e'
      frame_type = '10'
      frame_id = '00'
      addr16 = 'fffe'
      broadcast_radius = '00'
      options = '00'
      string = frame_type + frame_id + addr + addr16 + broadcast_radius +
options + msg
      length = find_length(string)
      checksum = checksum_value(string)
      command = initial + length + string + checksum
      command = codecs.decode(command, "hex") #converts into command
      return command

#----------------------------------------------------------------------------
def receipt(): #this returns address of sender and message sent
```

```
      mask = 0
      start_index = 1010
      end_index = 1000
      string = copy.copy(ser.readline())
      a = ba.hexlify(string)
      b = a.decode('utf-8')
      c = [b[k:k+2] for k in range (0, len(b), 2)] #this splits list into
blocks of 2
      for i in range(0, len(c) - 2):
      if c[i] == '7e' and mask == 0: #this identifies first start bit
            addr_start_index = i + 4
            addr_end_index = i + 11
            mask = 1;
      if c[i] == '21' and c[i+1] == '21' and c[i+2] == '21': #extracts
starting index of message
            start_index = i + 3
      if c[i] == '3f' and c[i+1] == '3f' and c[i+2] == '3f': #extracts ending
index of message
            end_index = i - 1
      if start_index >= end_index or start_index == 1010 or end_index ==
1000:
      return "Serial Read Failed"
      else:
      d = [c[i] for i in range(start_index, end_index + 1)]
      e = ''.join(d)
      f = [c[i] for i in range(addr_start_index, addr_end_index + 1)]
      addr = ''.join(f)
      msg = codecs.decode(codecs.decode(e,'hex'),'ascii')
      return [addr, msg]

#-------------------------------------------------------------------------
# below are modules for different modes of operation
#-------------------------------------------------------------------------
def init_comm(): #this initiates communication
      mask = 0;
      addr_file = open("address.txt", "r")
      addr_list = addr_file.readlines() #we read the entire file and place
information in a list
      addr_file.close()
      for i in range(0, len(addr_list)):
      temp = addr_list[i]
      temp = temp[:-1]
      addr_l = []
        addr_l.append(temp)
      data_file = open("data.txt", "a")
      print("Start of data tracking! Interrupt kernel to stop")
      try:
      while True:
            for i in range(0, len(addr_l)):
                  addr = addr_l[i]
                  msg = "code"
                ser.write(ping(addr,msg)) #ping command sent to end device
```

```
                time.sleep(3) #wait for data to be sent, processed and then
returned
                  temp = receipt()
                rec_addr = temp[0]
                  if len(rec_addr) != 16:
                    print("Unable to ping: Device-" + str(i))
                    break
                  else:
                    print("Ping Successful for Device-" + str(i))
                rec_msg = temp[1]
                  data = rec_addr + " , " + rec_msg + "\n"
                data_file.write(data)
      except KeyboardInterrupt:
        data_file.close() #finish writing to data file
      return print("End of communication")

#-------------------------------------------------------------------------
def add_new(): #This adds new devices to connected list
      add_more = 'y'
      addr_file = open("address.txt", "a")
      while (add_more == 'y'):
      mac = input("Add new device MAC address: ")
      if len(mac) != 16:
            print("Invalid MAC address")
      else:
            addr_file.write(mac + "\n")
            add_more = input("Do you want to add new devices? press y/n: ")
    print("Finished adding devices")
      addr_file.close()
      return

#--------------------------------------------------------------------------
# below are modules for viewing and changing settings
#--------------------------------------------------------------------------
def set_settings():
    print("\nChoose from one of the following options: ")
      print("1: Change PAN ID")
      print("2: Change Scan channel")
      print("3: Set Coordinator Enable")
      print("4: Set Channel Verification")
      print("5: Set Destination Address- High")
      print("6: Set Destination Address- Low")
      mode = input("Enter your selection: ")
      print("For the following parameters, enter an even number of hex digits
or otherwise specified.")
      if mode == "1":
      param = input("Enter new PAN ID (2-16 hex): ")
      if len(param) % 2 == 0:
            cmd = set_command('49', '44', param)
            ser.write(cmd)
      else:
            print("Invalid! Enter even number of hex digits")
      elif mode == '2':
```

```
      param = input("Enter new Scan Channel (1- 4 hex): ")
      if len(param) % 2 == 0:
            cmd = set_command('53', '43', param)
            ser.write(cmd)
      else:
            print("Invalid! Enter even number of hex digits")
      elif mode == '3':
      param = input("Enter 00 to disable and 01 to enable: ")
      if len(param) % 2 == 0:
            cmd = set_command('43', '45', param)
            ser.write(cmd)
      else:
            print("Invalid! Enter even number of hex digits")
      elif mode == '4':
      param = input("Enter 00 to disable and 01 to enable: ")
      if len(param) % 2 == 0:
            cmd = set_command('4a', '56', param)
            ser.write(cmd)
      else:
            print("Invalid! Enter even number of hex digits")
      elif mode == '5':
      param = input("Enter new Destination High (8 hex): ")
      if len(param) % 2 == 0:
            cmd = set_command('44', '48', param)
            ser.write(cmd)
      else:
            print("Invalid! Enter even number of hex digits")
      elif mode == '6':
      param = input("Enter new Destination Low (8 hex): ")
      if len(param) % 2 == 0:
            cmd = set_command('44', '4c', param)
            ser.write(cmd)
      else:
            print("Invalid! Enter even number of hex digits")
      else:
        print("Invalid Selection")
      return
 #------------------------------------------------------------------------
def view_setting(msb, lsb):
      param = ''
      d = ''
      cmd = set_command(msb, lsb, param) #send empty command
      ser.write(cmd)
      time.sleep(0.2)
      string = copy.copy(ser.readline())
      a = ba.hexlify(string)
      b = a.decode('utf-8')
      c = [b[k:k+2] for k in range (0, len(b), 2)] #this splits list into
blocks of 2
      if c[0] == '7e' and c[3] == '88' and c[7] == '00':
      for i in range (8 , len(c) - 1):
            d += c[i]
      return d
```

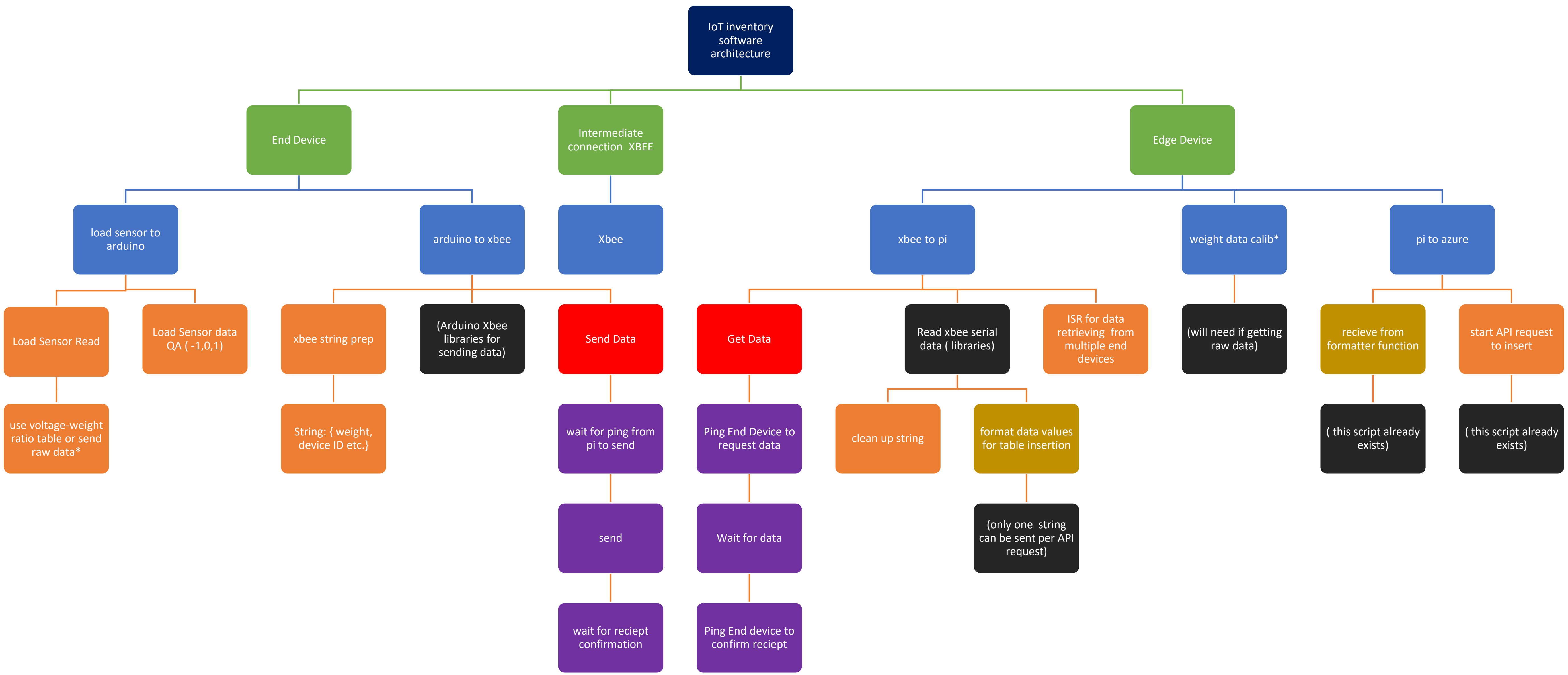
```
      else:
        print("Error in reading parameters. Check connection.")
      return param
 #---------------------------------------------------------------------------
def view_settings():
      pan_id = view_setting('49', '44')
      sc = view_setting('53', '43')
      jv = view_setting('4a', '56')
      ce = view_setting('43', '45')
      dh = view_setting('44', '48')
      dl = view_setting('44', '4c')
      if jv == '00':
      jv = 'Disabled'
      elif jv == '01':
      jv = 'Enabled'
      if ce == '00':
      ce = 'Disabled'
      elif ce == '01':
      ce = 'Enabled'
      print("\nPAN ID: "+ pan_id)
      print("Scan Channel: " + sc)
    print("Channel Verification: " + jv)
    print("Coordinator Enable: " + ce)
    print("Destination High: " + dh)
    print("Destination Low: " + dl)
      return
#----------------------------------------------------------------------------
 def start(): #this is the main program executable
      addr_file = open("address.txt", "r") #open address.txt and print added
devices and prints list
    print("Connected devices are: ")
      for addr in addr_file: #this shows existing devices
      print(addr)
      addr_file.close()
    print("\nChoose from one of the following options:")
      print("1: Enter data tracking mode")
      print("2: View local settings")
      print("3: Change local settings")
      print("4: Add new devices")
      mode = input("Enter your selection: ")
      if mode == '1':
      init_comm()
      elif mode == '2':
        view_settings()
      elif mode == '3':
      set_settings()
      elif mode == '4':
      add_new()
      else:
        print("Invalid selection!")
      ser.close()
      return print("Terminated")
```

End Device - Arduino Script

```
#include <HX711.h>


// HX711 circuit wiring
const int LOADCELL_DOUT_PIN = A5;
const int LOADCELL_SCK_PIN = A4;

HX711 scale;

String starts, ends, initial, msg, check, sending;
String mode;
char temp = 0;
float num = 0;
float weight = 0;
float calibration_value = 0;

void setup() {
  Serial.begin(9600);
  scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);

  starts = String("!!!");
  ends = String("???");
  initial = String("incorrect code");
  msg = String("DATA");
  check = String("code");
}


void loop() {

  if ( mode == 'running' ){
    //Serial.print("one reading:\t");
    //Serial.print(scale.get_units(), 1);
    Serial.print("\t| average:\t");
    msg = Serial.println(scale.get_units(10), 1);

    scale.power_down();              // put the ADC in sleep mode


  if(Serial.available() > 0) {
  //Serial.write(Serial.read());
   String str = Serial.readString();
   if (str == check) {
    sending = starts + msg + ends;
    }
    else {
     sending = starts + initial + ends;
      }
    Serial.print(sending);
    //Serial.print(str.length());
    delay(3000);
    scale.power_up();
    }
```

```
  }
  else if ( mode == 'calibrating') {
    calibration();
  }
}


void calibration() {


//1. Call `set_scale()` with no parameter.
  Serial.println("Set Scale:");
  scale.set_scale();
//2. Call `tare()` with no parameter.
 // Serial.println("Tare:");
 // scale.tare();
//3. Place a known weight on the scale and call `get_units(10)`.
  Serial.println("Place known weight:");
  while ( temp != 'a' ){
    temp =  Serial.read();
  }
  Serial.println("Enter known weight:");
    weight  =  Serial.read();

  num = Serial.println(scale.get_units(10));
  temp = 0;
//4. Divide the result in step 3 to your known weight. You should
//   get about the parameter you need to pass to `set_scale()`
  calibration_value = num/weight;

  //if (
  Serial.println(calibration_value);


//delay(10000);
}
```

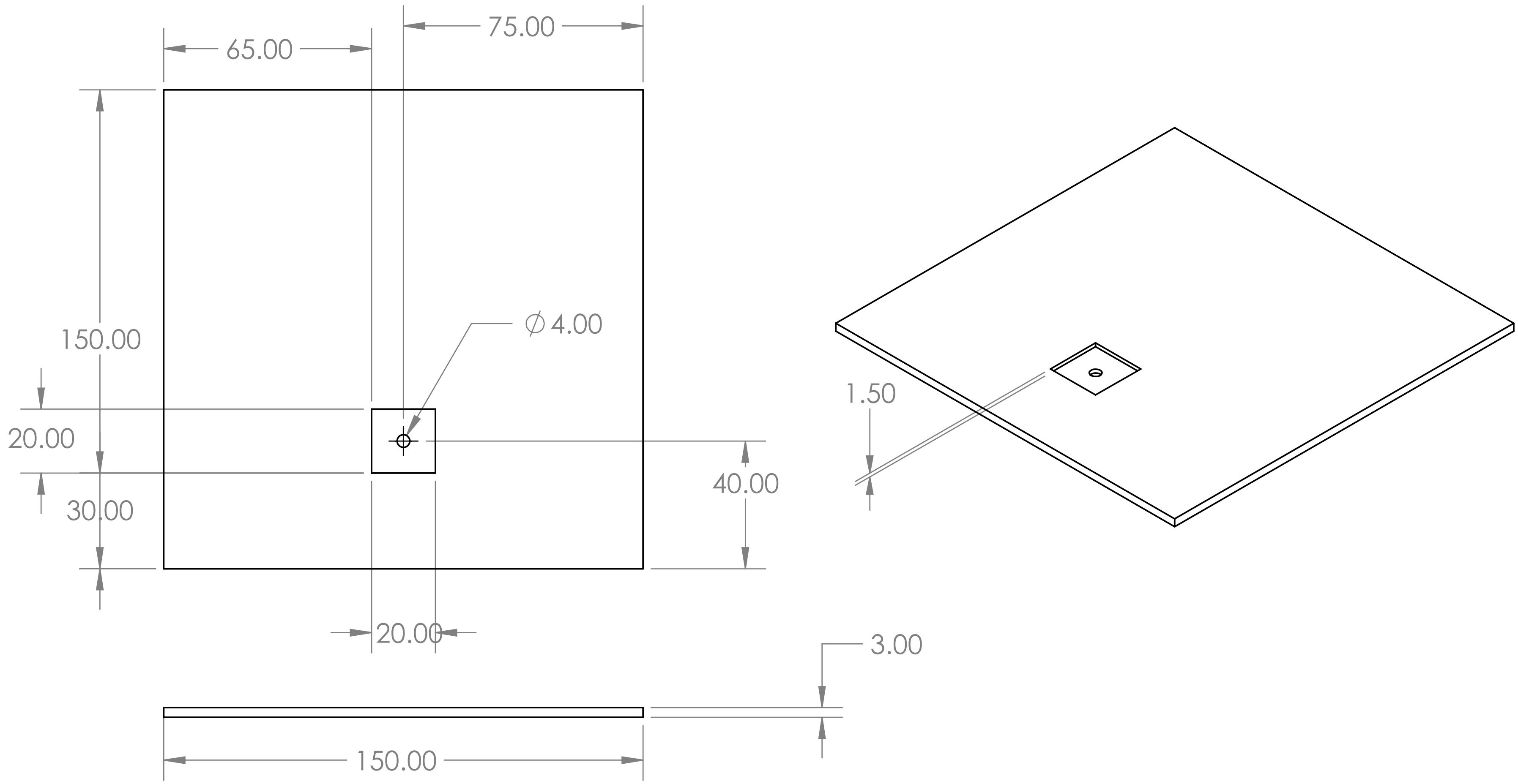
Appendix F: Full system top-level architecture diagram



Appendix G: Reference links

Section	References
Load Cell Setup	https://learn.sparkfun.com/tutorials/load-cell-amplifier-hx711-breakout-hookup-guide/all
Load Cell Datasheet	https://cdn.sparkfun.com/assets/b/f/5/a/e/hx711F_EN.pdf
HX711 Code	https://github.com/sparkfun/HX711-Load-Cell-Amplifier/tree/V_1.1
Xbee Datasheet	https://www.digi.com/pdf/ds_xbee_zigbee.pdf
Xbee Code	https://github.com/andrewrapp/xbee-api
Xbee WRL-11812 pi breakout board Datasheet	https://media.digikey.com/pdf/Data%20Sheets/Sparkfun%20PDFs/WRL-11812_Web.pdf
Xbee breakout board arduino DFR0015	https://wiki.dfrobot.com/Xbee_Shield_For_Arduino__no_Xbee___SKU_DFR0015_
Edge Device	https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-raspberry-pi-kit-c-get-started
Cloud	https://docs.microsoft.com/en-us/power-bi/service-real-time-streaming

Appendix H: Storage Container CAD files



NEXT ASSY	USED ON
APPLICATION	

UNLESS OTHERWISE SPECIFIED:

DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL ±
ANGULAR: MACH ± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±

INTERPRET GEOMETRIC TOLERANCING PER:

MATERIAL

FINISH

DO NOT SCALE DRAWING

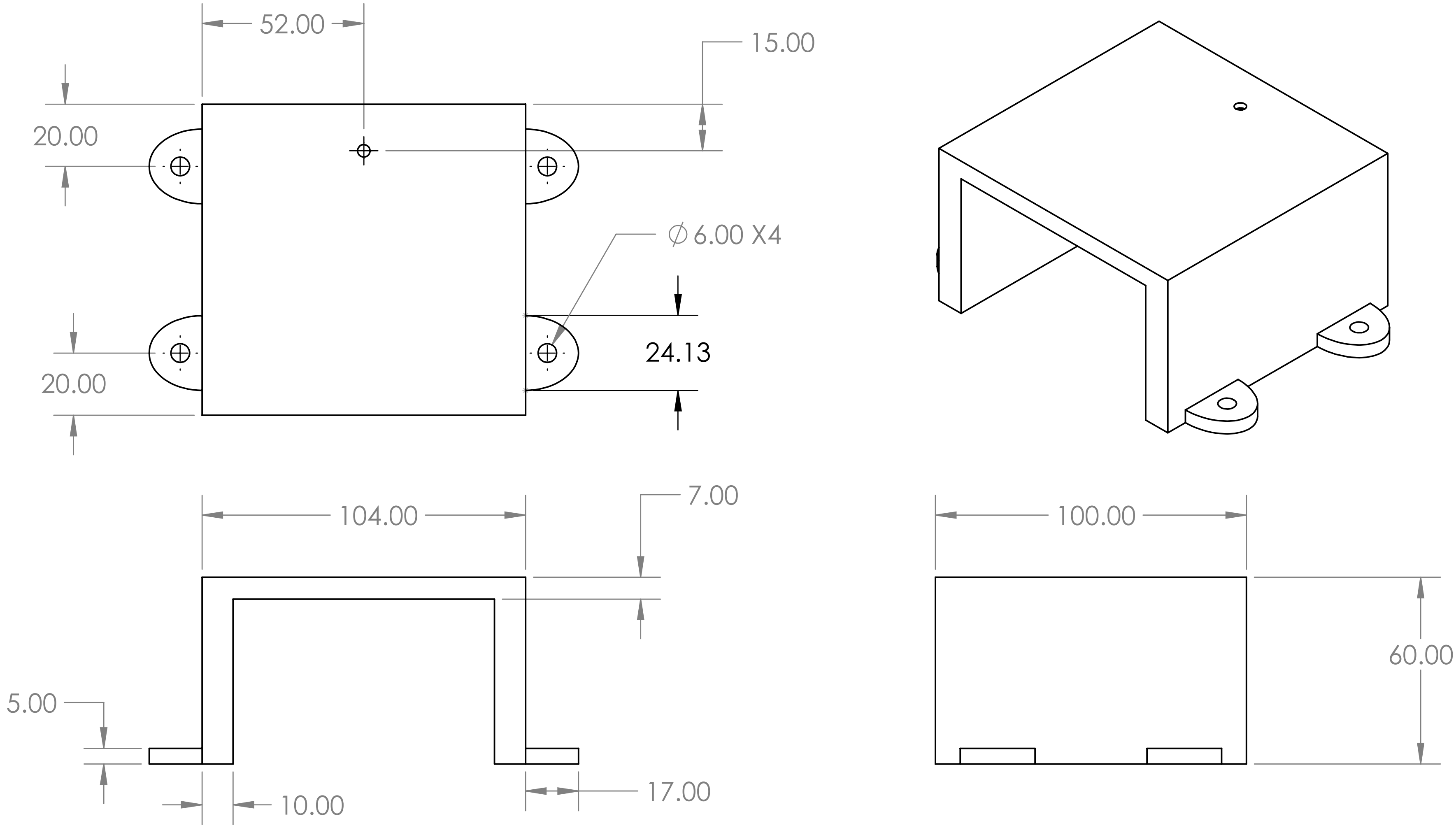
	NAME	DATE
DRAWN		
CHECKED		
ENG APPR.		
MFG APPR.		
Q.A.		

COMMENTS:

TITLE:

SIZE	DWG. NO.	REV
A	Pressure plate v2	

SCALE: 1:2 WEIGHT: SHEET 1 OF 1



		UNLESS OTHERWISE SPECIFIED:		NAME	DATE
		DIMENSIONS ARE IN INCHES TOLERANCES: FRACTIONAL ± ANGULAR: MACH ± BEND ± TWO PLACE DECIMAL ± THREE PLACE DECIMAL ±	DRAWN		
			CHECKED		
			ENG APPR.		
			MFG APPR.		
		INTERPRET GEOMETRIC TOLERANCING PER:	Q.A.		
		MATERIAL	COMMENTS:		
NEXT ASSY	USED ON	FINISH			
APPLICATION		DO NOT SCALE DRAWING			

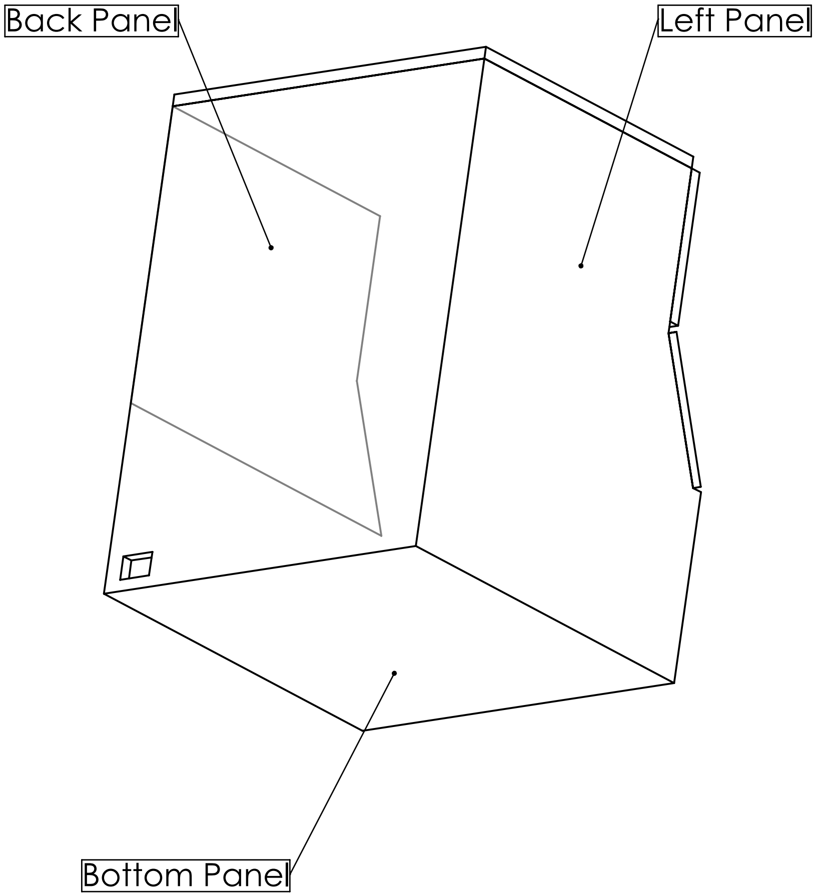
PROPRIETARY AND CONFIDENTIAL

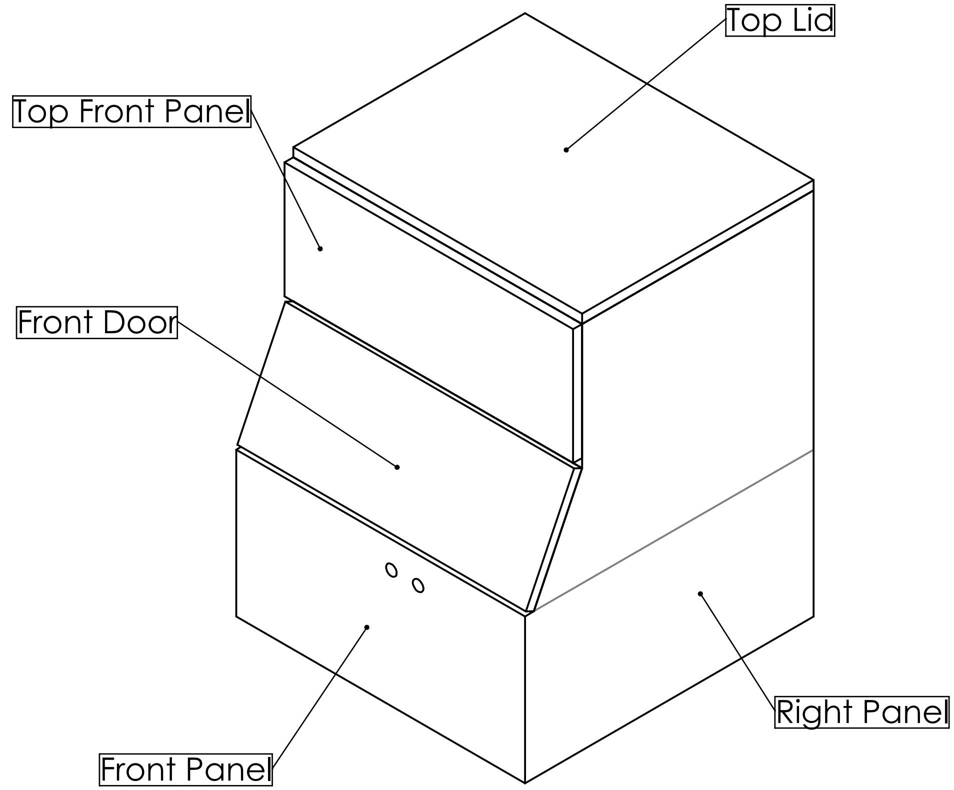
TITLE:

SIZE	DWG. NO.	REV
A	Load cell dock v2	
SCALE: 1:2	WEIGHT:	SHEET 1 OF 1

2 1

B B





A A

PROPRIETARY AND CONFIDENTIAL

THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF <INSERT COMPANY NAME HERE>. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF <INSERT COMPANY NAME HERE> IS PROHIBITED.

NEXT ASSY	USED ON
APPLICATION	

UNLESS OTHERWISE SPECIFIED:

DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL ±
ANGULAR: MACH ± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±

INTERPRET GEOMETRIC TOLERANCING PER:

MATERIAL

FINISH

DO NOT SCALE DRAWING

	NAME	DATE
DRAWN		
CHECKED		
ENG APPR.		
MFG APPR.		
Q.A.		

COMMENTS:

TITLE:

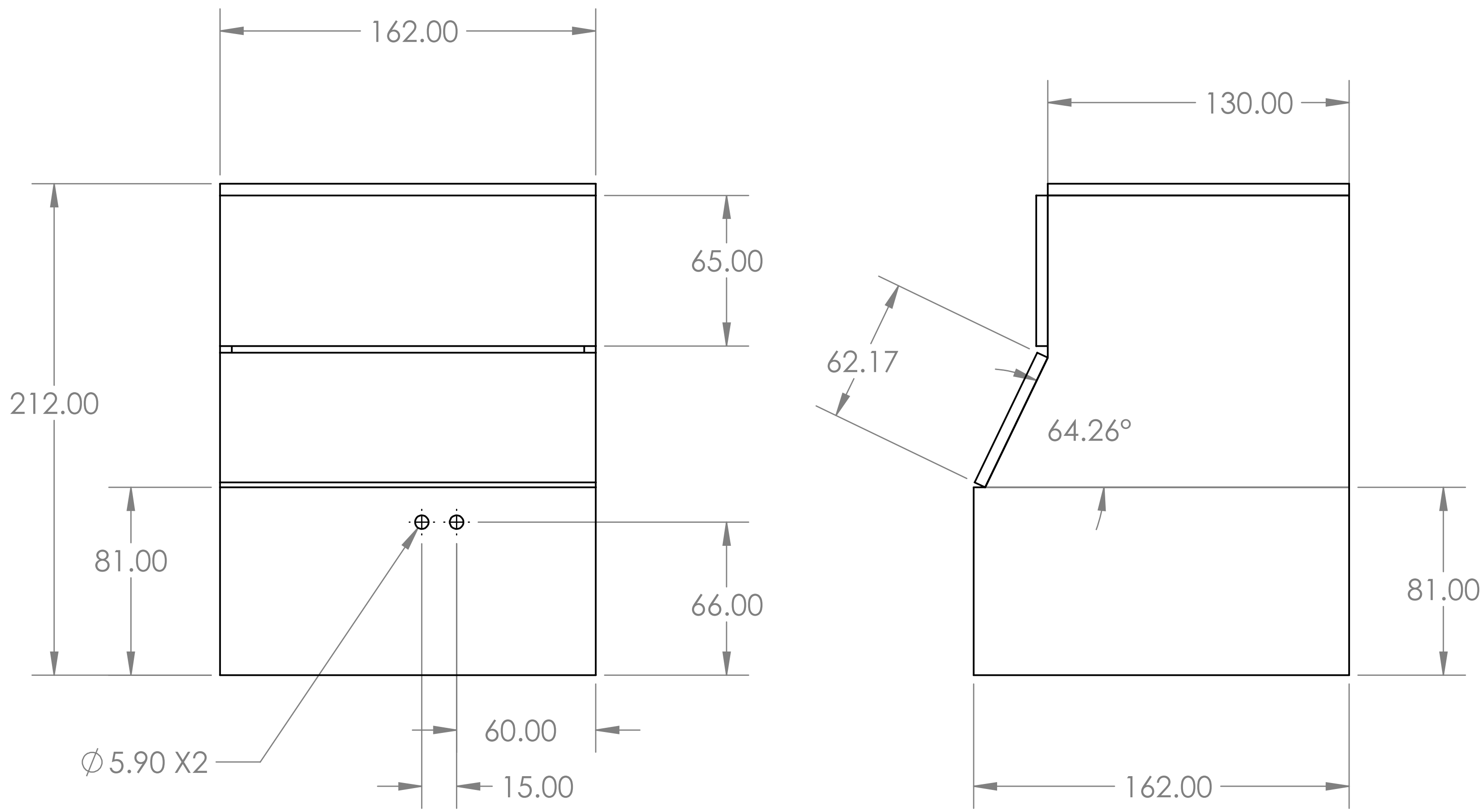
SIZE	DWG. NO.	REV
A	Dispenser	

SCALE: 1:3 WEIGHT: SHEET 1 OF 3

2 1

2 1

B B



A A

NEXT ASSY	USED ON
APPLICATION	

UNLESS OTHERWISE SPECIFIED:

DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL ±
ANGULAR: MACH ± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±

INTERPRET GEOMETRIC TOLERANCING PER:

MATERIAL

FINISH

DO NOT SCALE DRAWING

	NAME	DATE
DRAWN		
CHECKED		
ENG APPR.		
MFG APPR.		
Q.A.		

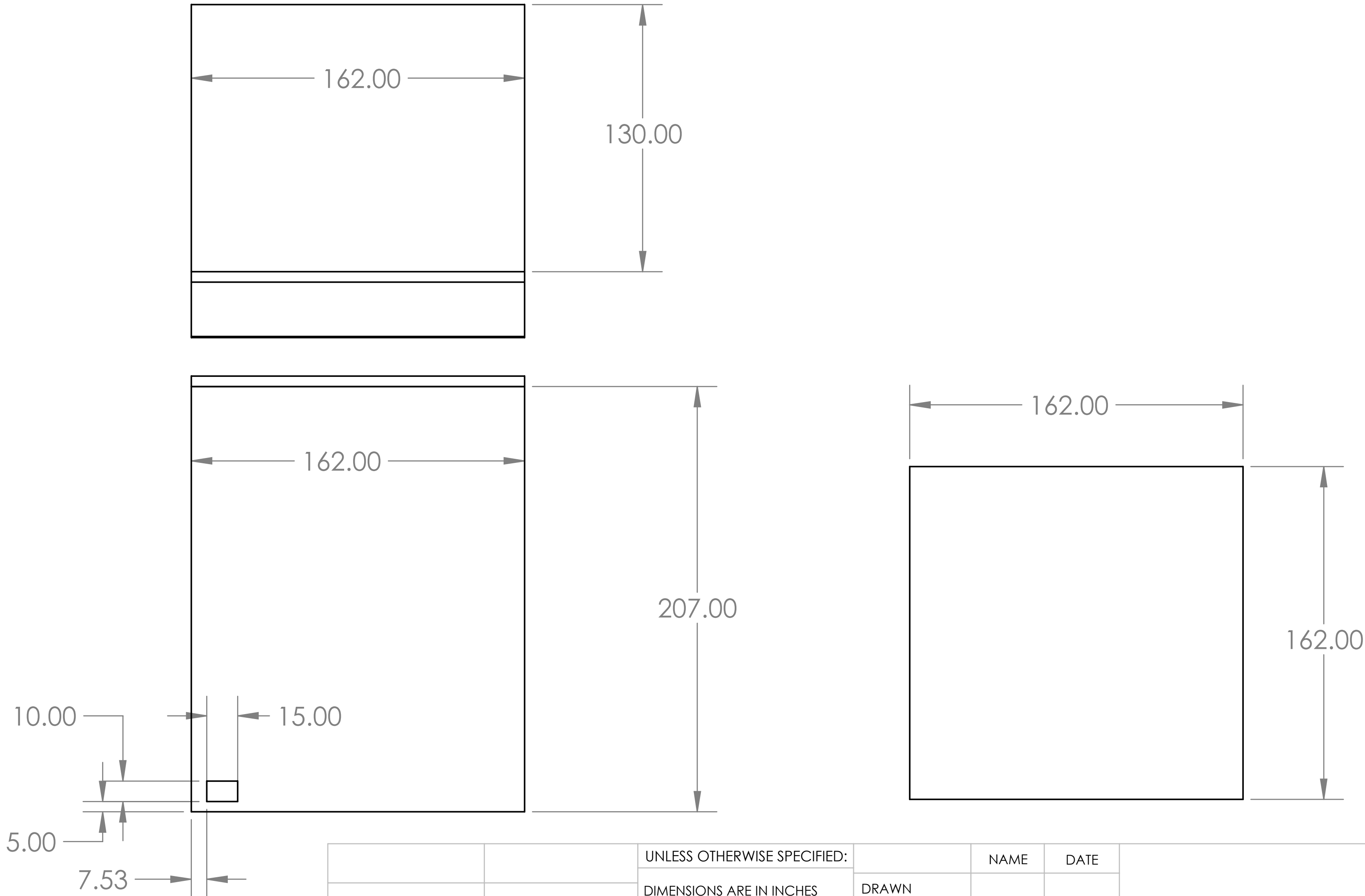
COMMENTS:

TITLE:

SIZE	DWG. NO.	REV
A	Dispenser	

SCALE: 1:3 WEIGHT: SHEET 2 OF 3

2 1



NEXT ASSY	USED ON
APPLICATION	

UNLESS OTHERWISE SPECIFIED:

DIMENSIONS ARE IN INCHES
TOLERANCES:
FRACTIONAL ±
ANGULAR: MACH ± BEND ±
TWO PLACE DECIMAL ±
THREE PLACE DECIMAL ±

INTERPRET GEOMETRIC TOLERANCING PER:

MATERIAL

FINISH

DO NOT SCALE DRAWING

	NAME	DATE
DRAWN		
CHECKED		
ENG APPR.		
MFG APPR.		
Q.A.		

COMMENTS:

TITLE:

SIZE **A** DWG. NO. **Dispenser** REV

SCALE: 1:3 WEIGHT: SHEET 3 OF 3