CS498 IoT Final Report
"Smart" Heating Registers

Rick Bischoff (rdb4@illinois.edu, rdbisch@gmail.com)

April 2021

Forward

This report is meant to be accompanied with the video overview available at https://www.youtube.com/watch?v=Om3EJ3IVGAw, and the GitHub repository at https://github.com/rdbisch/cs498_final_project. The main source files are included here for posterity, but are not exhaustive.

1 Introduction

Many newer larger houses are built with a multi-stage furnace. With this design, the furnace is able to heat separate parts of the house independently. This is cost-efficient and more comfortable for the residents of the house.

Unfortunately, most houses are not built this way. The heating system is either "on" or "off", leading to potentially wasted energy heating rooms that are already hot. Retrofitting existing homes to have multiple stages is cost prohibitive, and possibly even impossible because of space constraints. Further, furnace efficiency is a real consideration–they are designed to operate at specific capacity, and running under this capacity is actually worse for the wear and tear of components.

It is questionable that manually controlling registers leads to energy savings. For instance, [1], suggests that closing vents cause added pressure and unnecessary energy-waste. While this may be true for manual controlled vents that are "set-and-forget", there may be important differences from automatic vents. This remains to be seen, and warrants further study.

Despite energy efficiency at fixed temperatures, another real concern is the comfort of individuals living in a home. Certain rooms may have drafty windows, or be too far away from the furnace. In either of these cases, those

rooms will heat at different rates, causing the occupants to force the heater to come on early by artificially raising the temperature limits. The energy loss of this cannot be ignored.

We design a prototype solution "SmartRegister" that could, one day, provide better efficiency than these multi-stage systems, and be trivially refitted into any home with forced-air heating.

My system has two components. The first is a "smart heating register", that has the ability to open or close itself. The second is a smart thermostat, similar in principle to a "Nest", but with the ability to communicate to the SmartRegisters to maximize heating efficiency. (Note: The identification of the furnace control as a critical component didn't come until later. See section **??** for more about this.)

2 SmartRegister Design

A typical room has two vents, a supply and a return. The return registers are always open to ensure adequate air is available at the blower motor– inadequate air flow can decrease blower efficiency, resulting in inadequate heat dissipation, and decreased lifetime of the furnace itself. As such, we will focus solely on the supply vents.

The supply, when it is in the form of a register, has a damper that can open or close, by use of a manual lever. These devices are used to change the amount of air allowed to through the vent.

Some vents have fixed baffles that provide turbulence to ensure a more even mixture of air.

For more information on heating vents and terminology, see [2].

The placement and size of the supply registers places some real constraints on the design. There is no power supply available, which means it needs to be battery driven. No one wants to do maintenance on their registers periodically, so time between charges has to be large. Since the whole point of a register is to control heat fluctuations, a future design could potentially incorporate a thermometric generator to provide recharging. Finally, homes may have many registers, and they aren't exactly seen as luxury items, so there is pressure to keep the overall costs low.

The electronic design considerations culminated in the choice for a low-power microcontoller. The power and range requirements also ruled out using WiFi as the communication, so I had to choose an alternate radio source. Preliminary research showed that the only realistic choice for hobbyist work was LoRa (Zigbee was considered, but licensing fees of 6 figures

quickly put that one to rest).

The range of this radio can be up to 2km in some applications, which is more than we need here, but the design of this system could find additional application for commercial buildings or high-rise in which that would be appropriate.

Based on the power considerations, radio, battery, we landed on a Arduino-Radio package provided by [3]. This contained a smaller microprocessor, and a built-in RFM9x radio chip, integrated onto the same package.

The other hardware used in initial build was a PCT2075 temperature sensor and a SG92R microsevrvo. The PCT2075 was eventually replaced with the much cheaper DS18B20. In section **??**, we describe some flaws with all of our choices.

In preliminary tests of hardware capability, I found a fully-charged unit lasted roughly 3 days sending out a temperature every second or so. Based on this and the desire to have the unit last a year, a watchdog-dog based approach of sleeping for 2 minutes, sending data, and shutting down, is anticipated.

Figure 1 shows the details schematics for the SmartRegister. Minor details not covered elsewhere: The capacitors in parallel next to the Servo are meant to smooth out power draw from the servo. The different capacitance is to reduce the possibility of a resonant frequency. The voltage divider made up of 2 100kω resistors is so the battery life can be measured. The pull-up resistor on the DS18B20 is recommended to ensure a clean signal. Finally, the antenna was a simple wire spring, similar to [4].

The software used to control this is covered in section 4.

3 Central Controller Design

The design constraints here are a little bit easier. Wall- or thermostat-provided power should be available, so energy isn't a primary concern. Ultimately, the size of the device should be roughly compatible with thermostat-sized mounts, and of course it has to be able to communicate with all of the Smart Registers.

Because of these constraints, we went with the Raspberry Pi 4, paired with a LoRA breakout board. There is also a DS18B20 sensor so that eventually the Pi could act as a whole home thermostat. Figure 2 shows the detailed schematic.

Eventually, we realized that the system needed to also control the thermostat. A future design will incorporate this, but we were unable to do so

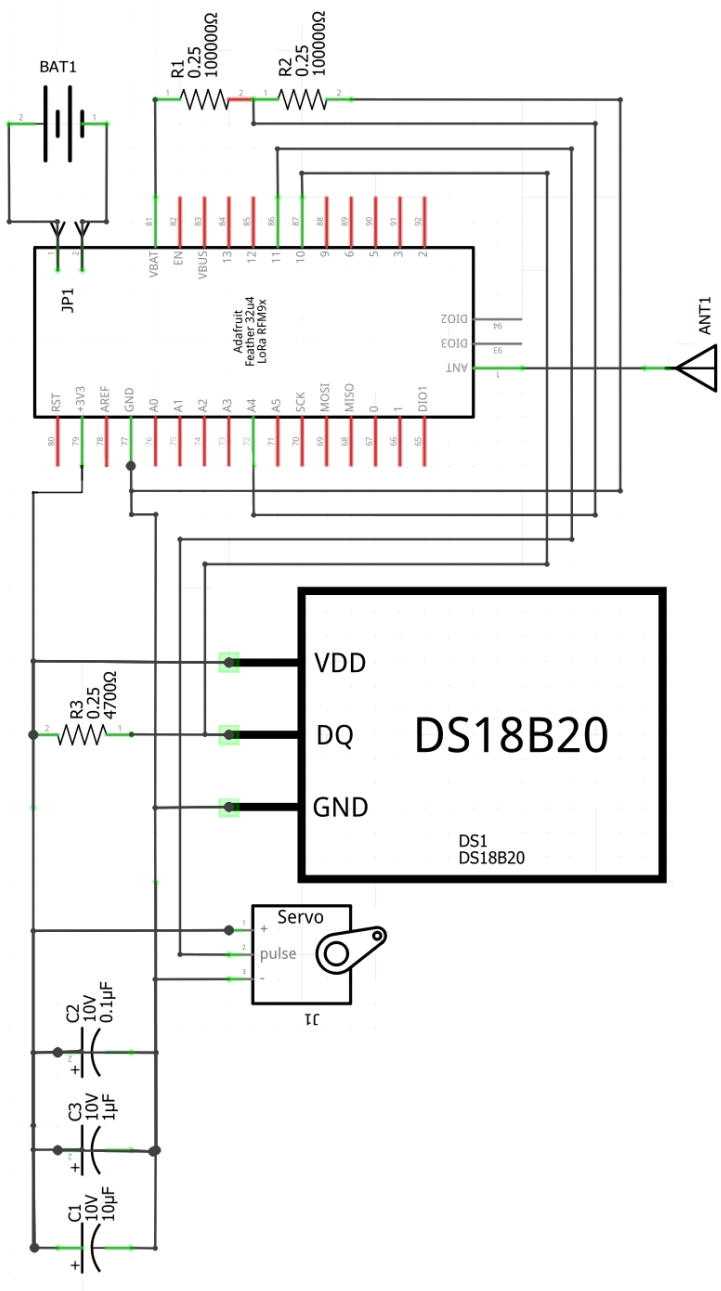

Figure 1: SmartRegister Electronic Schematic

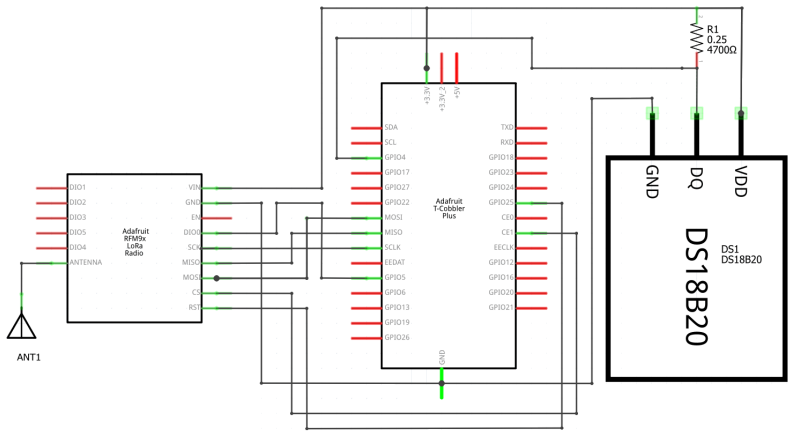

Figure 2: Electronic schematic for Raspberry Pi

in this iteration. Primarily because it requires specialized 24V hardware to interface, but also because we didn't want to accidentally fry the author's home furnace.

4 Software Design

The overall system is sketched in figure **??**. The network interface we used did not come with a complete stack, like Bluetooth or WiFi would, so we ended up having to implement a lot of basic features.

4.1 Protocol Design

We are dealing with a very low-level network interface here, without addressing or redundancy. Further, many of our devices will be sleeping often. Because of these constraints our network protocol is very lightweight, and our "Quality of Service" is basically, "good luck". This was determined to be acceptable because we are not dealing with a critical system here–if the vents miss an "open" command, it will not be the end of the world. If our thermostat misses a temperature reading, that's also not a big deal.

Each message is sent in plain text with the following structure:

addr command arg1 arg2? arg3?

4.1.1 Addresses

An address of "0" is broadcast. The main thermostat has a preassigned address of "1". The SmartRegisters, lacking a hardware defined unique ID, have to create an address as a 16-bit random number on boot. There is a small possibility of a collision, but this was deemed acceptable risk for this system.

4.1.2 Commands

`POST` This is to communicate the temperature and battery readings to the thermostat. This communication happens on a fixed timer and no acknowledgement or receipt is sent in response. On receipt, the thermostat will record this data. An example command would be something like `1 POST 25942 21.24 3.03`, which reads as "device 25942 is telling thermostat (1) that it is 21.24 degrees centigrade, and its battery is reading 3.03 volts."

`ADDR` Deprecated. This used to be part of a call-and-response allocaiton scheme, but was removed from the SmartRegisters in favor of the random number allocation scheme.

`SETFLAG` Sets the flag for a specific register. This is used by the `BAFFLE` command later on. An example command looks like `25942 SETFLAG 4`, which means SmartRegister 25942 should associate with the flag 4 going forward. This command is automatically sent when a new address is detected by the thermostat.

`BAFFLE` This is a broadcast command from the thermostat to the entire network. Each register uses its previously set flag in order to determine whether or not it should open it's baffle. This is done via a straightforward binary-AND masked on the flag, and is the reason why flags are powers of 2. An example command is `0 SETBAFFLE 49`, which means registers with flags 32, 16, and 1, should open their baffle.

The command specifies how the arguments are used.

4.2 SmartRegister

The code for the SmartRegister is shown in listing **??**. It is a striaghtforward design. The `setup` function ensures that the radio is setup correctly, the servo initialized, and sets some of the global state variables used later. The `loop` function, which runs periodically, does the following:

1. Send out current thermometer reading to the thermostat, by way of the `POST` command.
2. Listen for new radio packets with a timeout:
   (a) If we find a packet not meant for us, throw it away and restart the timeout.
   (b) Otherwise, parse the command and take action on it.
3. Flash the LED

Note that when an address `POST`s for the first time to a new network, it will immediately receive a `SETFLAG`.

There are a few external libraries used in this code.

- The "Radiohead" library to handle the RFM9x radio communication, though we are only using it as a dumb interface. Radiohead provides many functions that we did not use, because they were not implemented in exactly the same way on the Raspberry Pi side. It should also be noted that there is a hardware-level conflict with this library and the Arduino Servo library that required manual patching to resolve. [5]
- The OneWire and DallasTemperature API for reading the DS18B20 signals.
- The Servo library for interfacing with the servo that controls the baffle.

4.3 Controller

The controller code is relatively straight-forward python program.

The main loop handles radio transmissions with a timeout. Every time there is a transmission, or a timeout, a main loop is executed in the thermostat. If there is a radio message, that message is parsed and acted on.

Inside of the main loop, if a (configurable) significant amount of time has elapsed, the thermostat will also execute a "tick", in which it will output a status message to the console. Another timer, also configurable, controls how much data is retained in memory, and how it is flushed to disk for later analysis.

The thermostat itself has a temperate sensor. Depending on the heating mode, this temperature is checked against the user-setting. If the situation warrants it, the system would turn on, and an updated `BAFFLE` command

is sent out to all the registers. After the temperature returns to tolerance, another `BAFFLE` is sent out to open up all the registers.

5 Design Problems and Concerns

Though the design itself achieves the goals in hardware and software, as originally posited, it does suffer from some real design concerns that warrant more iteration. The author wishes there was another semester to work on this to put the finishing touches on it.

5.1 Physical Register

We never actually built a SmartRegister with a functional baffle, though we are not far away from the reality of this. The main issue is finding a way to mount the servo in a way that can be mounted to the device. Store-bought metal registers, such as those pictured here, are poor choices for retrofitting– they are often riveted together, and the design is such that the metal binds often, which could lead to the servo stalling and drawing too much current.

5.2 Servos

An early mistake was made in the choice of a servo to control the baffle opening. They were small, lightweight and had fine control and seemed to fit the bill for what was needed. In retrospect, this is a bad design because the servos are "always on" and they resist load. This means they draw more power than what is absolutely necessary. They are also prone to break if, say, a human tries to change the register setting manually.

A much better choice would have been a simple DC motor. This would have required separate driver hardware, or hat, to put into practice, as we are already nearing limits of what is OK on the Arduino feather itself.

5.3 Temperature Gauges

Because the temperature sensor lives inside the register, it is going to be subject to feeling the heating/cooling of the forced air furnace. This may cause the register to premature announce that the room it is in is done heating, causing some annoyance to the users. A better design would be either to fully insulate the control hardware from the forced air (difficult), or to have a wire-based temperature sensor that could be put out of the airflow.

5.4 Evolution to a Smart Thermostat

The initial sketches of this project were to only have SmartRegisters and leave the thermostat and heating system alone and separate. On testing the SmartRegisters this became a huge problem–if a room is cold, we need a way to turn the heat on, regardless of whether or not if the register is open. Sadly this realization was made too late in the project, and it would have required a specialized 24V relays for the Raspberry Pi to integrate with a central heating system.

5.5 Model of Home Heat and Machine Learning

Once an entire home is retrofitted with these devices, it should be possible to use the laws of physics to capture the underlying system. Predictions could be made, and proactive engagement of a combination of the heating system and registers could lead to even more comfort and efficiency for the home's users.

We could model the average home as a series of connected nodes, and edges corresponding to heat flow. Each room in the home to be represented by a node in a graph. Each node is connected to its (physical) adjacent neighbors, as well as connected to a furnace. Further, if the room is on the exterior of the home (in all 3 directions–the outside wall, the ground, or the attic), it is connected to a global heat-source or -sink, depending on a cooling mode. Internet access on the thermostat would allow outside information about the weather patterns to be used as well. The transfer of heat from room to room could be learned over time.

6 Testing Results

Though we could not test the SmartRegister inside of a literal heating duct, we did get over 100k sensor readings on three devices over the course of a few weeks. The results are concluded here.

1. Initial concerns around temperature calibration across device is ultimately unfounded. When plotted along time, there is almost no variance across device temperature when the devices where physically close to each other. See figure **??**.

2. The thermometer sensor takes a few minutes to get to up (or down) to temp. This could pose a problem in the future and will need to be looked into further. See figure **??**.

3. When under full power from our main computer USB, there is almost no packets dropped. However, when we switch to battery exclusively, the radio packets get increasingly jarbled as the battery runs out. Enabling debugging mode on both the arduino device and the python receiver, you can see commands like `Unknown command POCV"61138 with args 1 POCV"61138 23.43 4.0217 in SmartThermostat.recv.`, which indicates the word `POST` got garbled into `POCV"`. This is an easy enough fix by adding some checksum to radio communications. However, the fact that it happens to begin with, shows that we do not have a good understanding of the power requirements of this circuit and our power may be undersized for the application.

4. The lack of physical distinguished features make debugging these devices impossibly hard. One of the three prototypes we built doesn't publish at nearly the same frequency as the others, but without painstakingly taking one device out of circulation at a time, it is difficult to figure out. It is certain that a consumer would never put up with the headache. Perhaps a light, low cost, LED display could be added to identify which device is which.

7 Conclusion

Though we are not quite ready to pitch this system to "SharkTank," we do feel like overall significant progress has been made. The previous section comments on some considerations the next iteration should take into account for the next stage, but other than that, what have we learned?

- Soldering. The project required a lot of soldering of components, and that was something we had no experience in.
- Low-level radio. The project used a very low-level radio interface, and spent quite a bit of time learning how to work with it.
- Battery, and Low-power Concerns. Having always worked with an abundance of 120V 60Hz power, it was interesting learning to deal with these constraints.

All in all, we learned a lot and look forward to using the lessons learned in this project for future designs in both professional and hobbyist settings.

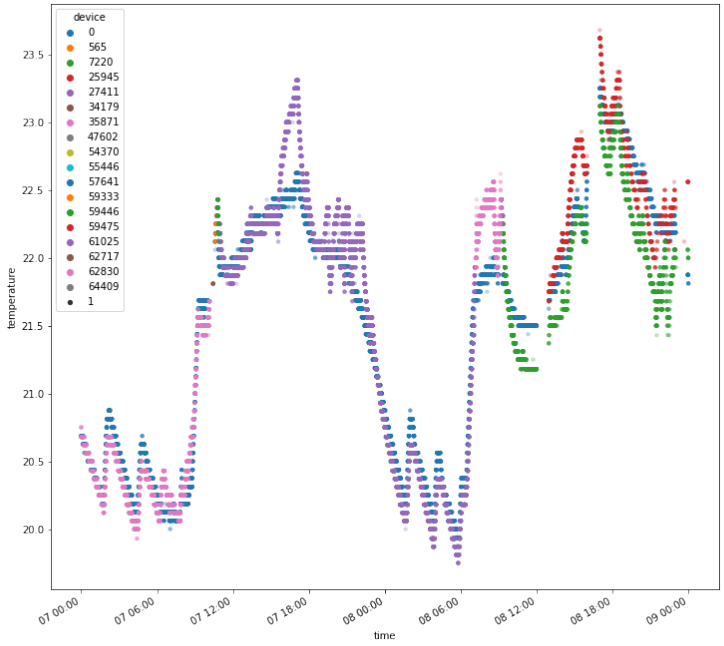

Figure 3: An example of multiple devices coalescing on the same temperature reading

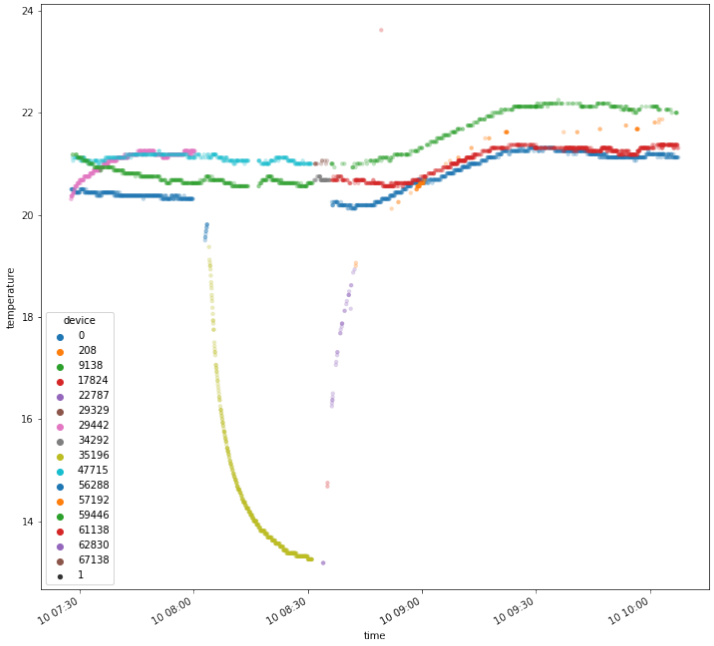

Figure 4: An example of the thermometer showing a very slow change to a temperature shock–in this case, the garage.

References

[1] "Should you close vents in unused rooms?" https://www.saveonenergy.com/learning-center/post/should-you-close-vents-in-unused-rooms, accessed: 2021-04-28.

[2] "Understanding the differences in air vents, registers, and grilles," https://adventair.com/blog/understanding-differences-air-vents-registers-grilles/, accessed: 2021-04-28.

[3] "Adafruit feather 32u4 rfm95 lora radio- 868 or 915 mhz - radiofruit," https://www.adafruit.com/product/3078, accessed: 2021-04-28.

[4] "Simple spring antenna - 915mhz," https://www.adafruit.com/product/4269, accessed: 2021-05-09.

[5] "Radiohead packet radio library for embedded microprocessors," https://www.airspayce.com/mikem/arduino/RadioHead/, accessed: 2021-04-15.

A Code Listings

A.1 SmartRegister

```
/* driver.ino
 * rdb4 4/10/2021
 *
 * This is the main driver code for the floor register hardware.
 * It is based off sample code available in
 * external_code/lora_test.ino
 * external_code/pct2075_test.ino
 * and
 * Servo demo code sweep
 */
#include <SPI.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <RH_RF95.h>
#include <Servo.h>

// Data wire is plugged into digital pin 2 on the Arduino
```

```
#define ONE_WIRE_BUS 10

// Setup a oneWire instance to communicate with any OneWire device
OneWire oneWire(ONE_WIRE_BUS);

// Pass oneWire reference to DallasTemperature library
DallasTemperature sensors(&oneWire);

/* Comment out for real run */
#define DEBUG

/* Which pin is servo on */
#define SERVO_PIN 11
Servo baffle;
int baffle_pos;
#define BAFFLE_OPEN 90
#define BAFFLE_CLOSED -90

/* For battery measurement
 *  https://learn.adafruit.com/adafruit-feather-32u4-basic-proto/power-management
 */
#define VBATPIN A4

/* for feather32u4 */
#define RFM95_CS 8
#define RFM95_RST 4
#define RFM95_INT 7

// Change to 434.0 or other frequency, must match RX's freq!
#define RF95_FREQ 915.0

// Singleton instance of the radio driver
RH_RF95 rf95(RFM95_CS, RFM95_INT);

/* SmartRegister settings */
#define SR_UNDEFINED 0xDEADBEEF
#define SR_OPEN = 0
#define SR_CLOSED = 1
unsigned int sr_addr;      // Unique ID generated at startup
int sr_flag;      // Bit mask flag set by server
```

```
int sr_state = SR_UNDEFINED;

void setup() {
  pinMode(RFM95_RST, OUTPUT);
  digitalWrite(RFM95_RST, HIGH);


#ifdef DEBUG
  Serial.begin(115200);
  while (!Serial) {
      delay(1);
  }
#endif
  delay(100);

  randomSeed(analogRead(0));
  sr_addr = random();
  while (sr_addr == 0) sr_addr = random();
#ifdef DEBUG
  Serial.println("Random integer is ");
  Serial.println(sr_addr);
#endif

#ifdef DEBUG
  Serial.println("Feather LoRa TX Test!");
#endif
  // manual reset
  digitalWrite(RFM95_RST, LOW);
  delay(10);
  digitalWrite(RFM95_RST, HIGH);
  delay(10);

  while (!rf95.init()) {
#ifdef DEBUG
    Serial.println("LoRa radio init failed");
    Serial.println("Uncomment '#define SERIAL_DEBUG' in RH_RF95.cpp for detailed debu
#endif
    while (1) {
      digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage leve
      delay(250);                        // wait for a second
```

```
      digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LO
      delay(250);
    }
  }
#ifndef DEBUG
  Serial.println("LoRa radio init OK!");
#endif
  // Defaults after init are 434.0MHz, modulation GFSK_Rb250Fd250, +13dbM
  if (!rf95.setFrequency(RF95_FREQ)) {
#ifdef DEBUG
      Serial.println("setFrequency failed");
#endif
      while (1) {
        digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage le
        delay(250);                       // wait for a second
        digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage
        delay(500);
      }
  }
#ifdef DEBUG
  Serial.print("Set Freq to: "); Serial.println(RF95_FREQ);
#endif
  // Defaults after init are 434.0MHz, 13dBm, Bw = 125 kHz, Cr = 4/5, Sf = 128chips/s

  // The default transmitter power is 13dBm, using PA_BOOST.
  // If you are using RFM95/96/97/98 modules which uses the PA_BOOST transmitter pin,
  // you can set transmitter powers from 5 to 23 dBm:
  rf95.setTxPower(23, false);


  /**** ****/
  sensors.begin();  // Start up the library

  /*** Setup Servo ***/
  baffle.attach(SERVO_PIN);
  baffle_open();
  delay(15);

#ifdef DEBUG
  Serial.println("\n\n\nMAIN LOOP STARTING\n\n\n");
```

```
#endif

}

float getTemp() {
  // Send the command to get temperatures
  sensors.requestTemperatures();
  return sensors.getTempCByIndex(0);
}

int16_t packetnum = 0;  // packet counter, we increment per xmission
int state = 0;

void loop() {
  float v = analogRead(VBATPIN);
  v *= 2;    // we divided by 2, so multiply back
  v *= 3.3;  // Multiply by 3.3V, our reference voltage
  v /= 1024; // convert to voltage

  float f = getTemp();
  char radiopacket[30];
  sprintf(radiopacket,
    "1 POST %u %d.%02d %d.02%d",
    sr_addr,
    (int)f,
    (int)(f*100)%100,
    (int)v,
    (int)(v*100));

#ifdef DEBUG
  Serial.print("Sending "); Serial.println(radiopacket);
#endif
  rf95.send((uint8_t *)radiopacket, strlen(radiopacket));
  delay(10);
  rf95.waitPacketSent();

  /**** RECEIVE LOOP ********/
  uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
  uint8_t len = sizeof(buf);
next:
```

```
  if (rf95.waitAvailableTimeout(1000)) {
    if (rf95.recv(buf, &len)) {
        unsigned int check_addr;
        int bytes;
        uint8_t* rest = 0;

        buf[len] = 0;
#ifdef DEBUG
        Serial.print("<");
        Serial.println((char*)buf);
#endif
        int ret = sscanf(buf, "%u %n", &check_addr, &bytes);
        if (ret < 1 || (check_addr > 0 && (check_addr != sr_addr))) {
#ifdef DEBUG
          Serial.println("address did not match pattern.\n");
#endif
          goto next;
        }

        rest = buf + bytes;
        if (rest[0] == 'S') setFlag(rest);
        else if (rest[0] == 'B') setBaffle(rest);
        else {
          ;
#ifdef DEBUG
          Serial.println("Unhandled command in message.");
          Serial.println((char*)buf);
          Serial.println("*************");
#endif
        }
    }
  }
  /* Blink the LED so we can see things are still working. */
  digitalWrite(LED_BUILTIN, HIGH);
  delay(500);
  digitalWrite(LED_BUILTIN, LOW);
  delay(500);
}

/** Handle SETFLAG message **/
```

```
void setFlag(uint8_t* message) {
#ifdef DEBUG
  Serial.println("in setFlag....sr_flag was...");
  Serial.println((char*)message);
  Serial.println(sr_flag);
#endif
  sscanf(message, "SETFLAG %d", &sr_flag);
#ifdef DEBUG
  Serial.println(sr_flag);
#endif
}

/** Handle BAFFLE message **/
void setBaffle(uint8_t* message) {
  int flags;
  sscanf(message, "BAFFLE %d", &flags);

  int r = (flags & sr_flag);
#ifdef DEBUG
  Serial.println("setBaffle debug  flags, sr_flags, r");
  Serial.println(flags);
  Serial.println(sr_flag);
  Serial.println(r);
#endif
  if (r) baffle_open();
  else baffle_close();
}

void baffle_open() {
  if (baffle_pos != BAFFLE_OPEN) {
#ifdef DEBUG
      Serial.println("OPENING BAFFLE");
#endif
      baffle.write(BAFFLE_OPEN);
      baffle_pos = BAFFLE_OPEN;
  }
}

void baffle_close() {
  if (baffle_pos != BAFFLE_CLOSED) {
```

```
#ifdef DEBUG
      Serial.println("CLOSING BAFFLE");
#endif
      baffle.write(BAFFLE_CLOSED);
      baffle_pos = BAFFLE_CLOSED;
  }
}
```

A.2 SmartThermostat

A.2.1 Radio.py

```
"""
Example for using the RFM9x Radio with Raspberry Pi.

Learn Guide: https://learn.adafruit.com/lora-and-lorawan-for-raspberry-pi
Author: Brent Rubell for Adafruit Industries
"""
# Import Python System Libraries
import time
# Import Blinka Libraries
import busio
from digitalio import DigitalInOut, Direction, Pull
import board
# Import RFM9x
import adafruit_rfm9x
import glob

class Radio:
    def __init__(self):
        # Configure LoRa Radio
        self.CS = DigitalInOut(board.CE1)
        self.RESET = DigitalInOut(board.D25)
        self.spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
        self.rfm9x = adafruit_rfm9x.RFM9x(self.spi, self.CS, self.RESET, 915.0)
        self.rfm9x.tx_power = 23
        self.prev_packet = None
        self.packetCount = 0
        self.badPackets = 0
```

```
    def mainLoop(self, recvCallback):
        while True:
            #if self.packetCount % 1000 == 0:
            #    print("Packets Received {} Bad {} % {}".format(self.packetCount, sel

            packet = self.rfm9x.receive(with_header=True, keep_listening=True, with_a
            if packet == None:
                time.sleep(0.1)
                continue

            self.packetCount += 1
            # Received a packet!
            # Print out the raw bytes of the packet:
            #print("Received (raw header):", [hex(x) for x in packet[0:4]])
            #print("Received (raw payload): {0}".format(packet[4:]))
            #print("RSSI: {0}".format(self.rfm9x.last_rssi))
            try:
                print("<<< {}".format(packet[4:]))
                recvCallback(str(packet[4:], "utf-8"))
            except (ValueError, UnicodeDecodeError) as err:
                #print("### Error processing packet {}\n{}".format(packet, err))
                self.badPackets += 1

    def send(self, data):
        print(">>> {}".format(data))
        self.rfm9x.send(bytearray(data, "utf-8"))
```

A.2.2 Thermostat.py

```
# Thermostat.py
# Smart Thermostat to control Smart Registers
from datetime import datetime, timedelta
import math
import pandas as pd

# Local temp
import TempSensor

# How many readings to keep local
queue_size = 10
```

```
class ThermostatException(Exception):
    pass

# This is an abstract data structure
#  to represent a (remote) physical device.
SR_UNDEFINED = 2
SR_OPEN = 0
SR_CLOSED = 1
class SmartRegister:
    def __init__(self, addr, flag):
        self.temps = []
        self.dates = []
        self.flags = []
        self.state = SR_OPEN
        self.addr = addr
        self.flag = flag

    def registerReading(self, temp, flag):
        self.temps.append(temp)
        self.dates.append(datetime.now())
        self.flags.append(int((flag & self.flag) > 0))

    def __str__(self):
        t = datetime.now()
        temp = self.temps[-1]
        date = self.dates[-1]
        flag = self.flags[-1]

        d = t - date
        if d < timedelta(days = 1): time_str = "{:02d}:{:02d}".format(date.hour, date
        else: time_str = "{:02d}/{:02d}".format(date.month, date.days)

        return "r={0} @{1} t={2:6.2f} flag={3}".format(self.addr, time_str, temp, fla

    # This is meant to be called periodically
    #  to keep memory footprint low.
    # In the curren design of an update every 8s, a 2-week update
    #  should have about 2.4k rows, so hopefully 10k memory is ok (per register)
    def archiveData(self, filename, asof, delta):
```

```
        # Otherwise, just dump the data out to a CSV
        #  Another ocmponent will read this in for analytics later
        df = pd.DataFrame({
            "temperature": self.temps,
            "datetime": self.dates,
            "flags": self.flags
        })
        df.to_csv(filename)

        # Now we truncate the data to be fresher than "delta"

        # Find the first date in dates that is older than days old.
        #  (remember these lists are all stored in increasing order of time)
        # so as long as "asof" is newer than anything in the list,
        # all we have to do is find the first element younger than our delta.
        found = None
        for idx, t in enumerate(self.dates):
            if asof - t < delta:
                found = idx
                break

        # If we didn't actually find anything,
        #  no truncation has to happen.
        if found == None: return

        # Truncate internal structure
        self.temps = self.temps[found:]
        self.dates = self.dates[found:]
        self.flags = self.flags[found:]

    def openRegister(self):
        assert(self.state == SR_CLOSED)
        self.state = SR_OPEN

    def closeRegister(self):
        assert(self.state == SR_OPEN)
        self.state = SR_CLOSED

    def flipState(self):
        if self.state == SR_CLOSED: self.openRegister()
```

```
        elif self.state == SR_OPEN: self.closeRegister()
        else:
            raise ThermostatException("Register in unknown state")


class SmartThermostat:
    def __init__(self, radio):
        self.radio = radio
        # This houses all of our discovered smart registers.  The key is their
        #  pseudo-MAC.   The initial value 0 represent this device, i.e. the thermost
        #  own temperature reading.
        self.registers = { 0: SmartRegister(0,0) }
        self.flags = 0
        self.lastTick = datetime.now()
        self.lastSave = self.lastTick
        self.TICK_FREQUENCY = timedelta(seconds = 4)
        self.SAVE_FREQUENCY = timedelta(seconds = 30)
        self.demo = 0

    def archiveData(self, filename):
        with open(filename, "w") as f:
            f.write("registers\n")
            for addr in self.registers.keys():
                f.write(str(addr) + "\n")

    def readTemp(self):
        t = TempSensor.read_temp()
        self.registers[0].registerReading(t, 0)  #TODO Change this to system ON or OF

    # This is called by radio
    #  to process events.
    def mainLoop(self, mesg):
        self.recv(mesg)

        # Figure out if we should do a self-update
        t = datetime.now()
        if t - self.lastTick > self.TICK_FREQUENCY:
            self.tick()
            self.lastTick = t
```

```
        # Store off data
        if t - self.lastSave > self.SAVE_FREQUENCY:
            print("\n\n============= DATA FLUSH ==============\n\n")
            # isoformat() returns "2021-05-06T14:01:13.313976"
            # [0:13] truncates 2 digits after T
            suffix = t.isoformat()[0:13]
            self.archiveData("data/smart_{}.csv".format(suffix))
            for r in self.registers.values():
                r.archiveData(
                    "data/register_{}_{}.csv".format(r.addr, suffix),
                    t,
                    timedelta(days=14))
            self.lastSave = t

    # Ordinary clock tick with no radio activity
    def tick(self):
        # Read ambient temperature
        self.readTemp()
        # Update log
        outstr = "{0:4d}/{1:7d} (bad/total)pckt   ".format(self.radio.badPackets, sel
        for r in self.registers.values():
            outstr += r.__str__() + "   "
        print(outstr)

        ambientTemp = self.registers[0].temps[-1]
        # Decide if we should turn on thermostat?
        if ambientTemp < 23:
            # Turn on thermostat
            # Switch relay if we have one...

            # Figure out what rooms to open or close.
            if self.demo % 4 == 0: threshold = 21
            else: threshold = 23
            self.demo += 1

            temp = 0
            for r in self.registers.values():
                if r.temps[-1] < threshold: temp += r.flag
            if temp != self.flags:
                self.flags = temp
```

```
                self.radio.send("0 BAFFLE {} ".format(self.flags));

        # Decide if we've lost sensors

    def recv(self, mesg):
        commands = { "ADDR": self.addr,
                     "POST": self.post,
                     "MAN": self.manual }
        mesgs = mesg.split(' ')
        # Our protocol is <addr> <command> <args>
        # Since we are the server, our valid address is 0 (broadcast) or 1.
        #  So if its not here, ignore the message

        addr = int(mesgs[0])
        if (addr != 0 and addr != 1): return

        if (mesgs[1] not in commands):
            print("Unknown command {} with args {} in SmartThermostat.recv.".format(m
        else:
            commands[mesgs[1]](mesgs[2:])

    def addr(self, args):
        # Message is of form
        # 1 ADDR abcde
        #
        # where abcde is the register's unique address.

        # Find first unused flag.
        if len(self.registers) == 0: flag = 1
        else:
            max_flag = max([ r.flag for r in self.registers.values()])
            # == 0 is our own flag, so we hardcode a bypass
            if max_flag == 0: flag = 1
            else: flag = int(math.log2(max_flag)) + 1
        if flag > 31:
            raise ThermostatException("Out of flags in SmartThermostat.")

        addr = int(args[0])
        flag = 2**flag
        reg = SmartRegister(addr, flag)
```

```
        assert(addr not in self.registers)
        self.registers[addr] = reg
        self.radio.send("{} SETFLAG {} ".format(addr, flag))

    def post(self, args):
        addr = int(args[0])

        # This can happen if we reboot the thermostat
        # while registers are still running
        if addr not in self.registers:
            self.addr(args)

        try:
            temp = float(args[1])
            self.registers[addr].registerReading(temp, self.flags)
        except ValueError as err:
            pass

    # A human changed the register setting manually.
    def manual(self, args):
        addr = int(args[0])
        assert(addr in self.registers)
        self.registers[addr].flipState()
```

A.2.3 TempSensor.py

```
import glob

# From https://learn.adafruit.com/adafruits-raspberry-pi-lesson-11-ds18b20-temperatur
base_dir = '/sys/bus/w1/devices/'
device_folder = glob.glob(base_dir + '28*')[0]
device_file = device_folder + '/w1_slave'

def read_temp_raw():
    f = open(device_file, 'r')
    lines = f.readlines()
    #print(lines)
    f.close()
    return lines
```

```
def read_temp():
    lines = read_temp_raw()
    while lines[0].strip()[-3:] != 'YES':
        time.sleep(0.2)
        lines = read_temp_raw()
    equals_pos = lines[1].find('t=')
    if equals_pos != -1:
        temp_string = lines[1][equals_pos+2:]
        temp_c = float(temp_string) / 1000.0
        return temp_c
```