An Extendable Python IoT Server and Task Handler

by

Bryce Beagle

Barrett Honors College, Arizona State University, 2018

Thesis Committee

Ruben Acuña, Director

Dr. Shawn Jordan, Committee Member

Abstract

The Internet of Things (IoT) is term used to refer to the billions of Internet connected, embedded devices that communicate with one another with the purpose of sharing data or performing actions. One of the core usages of the proverbial network is the ability for its devices and services to interact with one another to automate daily tasks and routines. For example, IoT devices are often used to automate tasks within the household, such as turning the lights on/off or starting the coffee pot. However, designing a modular system to create and schedule these routines is a difficult task.

Current IoT integration utilities attempt to help simplify this task, but most fail to satisfy one of the requirements many users want in such a system – simplified integration with third party devices. This project seeks to solve this issue through the creation of an easily extendable, modular integrating utility. It is open-source and does not require the use of a cloud-based server, with users hosting the server themselves. With a server and data controller implemented in pure Python and a library for embedded ESP8266 microcontroller-powered devices, the solution seeks to satisfy both casual users as well as those interested in developing their own integrations.

Introduction (3p)

Purpose of Project

The purpose of this project was to create a modular Internet of Things (IoT) system to create an interface between DIY devices and consumer grade products. The end product is focused on the automation of household tasks and behaviors for these devices, with functionality resembling a combination of IFTTT (*If This Then That*; used for integration between different web services) and Tasker (used for automating system actions on a cellphone). Users have the ability to integrate devices and services from off-the-shelf products and services, as well as third party or homebrewed solutions.

This project provides a design architecture as well as an implementation thereof. The architecture describes a system that addresses all of the design requirements, while the implementation makes it happen. The implementation provides the software required to run the system, as well as a set of physical modules that are able to work with it. The modules include parts lists, PCB schematics, board designs, and embedded software – everything required to create and deploy one from scratch.

Users are able to configure their system to create collections of tasks, called *routines*, that are triggered by internal or external state changes, such as the time transitioning to a predesignated time of the day, or a door being opened. These routines contain *events*, or sequences of actions to perform, where each action is an atomic operation for a device to execute. A routine can contain one or more events, and each event can contain one or more actions to perform. For example, if a user wants their Philips Hue internet-connected light bulbs to turn on automatically when they open a specific door, they can create a routine that is run when a predesignated door sensor is activated. Devices send status updates to a server which listens to the changing values and notifies routines that are subscribed to pieces of data.

The system has a *server* that handles connections and routes incoming data to device driver instances using a data control library, henceforth referred to as the *controller*. The controller manages routines and events in the system as well as the data structure that holds all of the device instances. *Devices*, both physical (*e.g.*, a Temperature Sensor) and cloud-based (*e.g.*, a Google Home), are configured for use by the controller with *driver* files that express the capabilities of the hardware or software that they represent. For example, a driver for the Philips Hue smart light bulbs would express the ability to set and retrieve attributes such as brightness, hue, saturation and on/off state, as well as behaviors such as dimming and pulsing.

Routines, events, and devices are stored in a configuration, allowing the system to start up preconfigured and ready for use. Events referenced in routines must have corresponding configuration entries describing them, and devices mentioned in events must also have drivers that the controller can access and utilize. Storing configurations in non-volatile, text-based formats allows them to be version controlled, shared, and edited with relative ease.

The software provided with this project, called *Idiotic,* is an extendable implementation of the system described above. It is written in Python 3, with the server implemented with the Flask web framework library. Routines, events, and devices are loaded from JSON configuration files. There are also a small

number of PCB devices, designed using Eagle, that function with the system as a demonstration of functionality.

Developers that wish to integrate their own devices into *Idiotic* can make use of the server's client platform agnostic websocket-based API to integrate their device within the system, using the simple, documented protocol. Creating a single driver file in a designated directory is all that is required to allow the controller to handle data from, and make commands to, a new device type. All integrated devices are treated as a first-class citizens within the system.

Furthermore, if the new device is built on an ESP8266 microcontroller platform, there is a provided library that allows for expedited embedded development and integration within the system. A developer simply needs to extend a template file to map which functions are correlated with which pieces of data and behaviors. If the new device is unable to utilize this library, it simply needs to connect to the server over a Websocket and follow a simple communication protocol.

Motivation

The motivation for this project stemmed from the lackluster support commercial smarthome products, such as the Google Home and Amazon Alexa, have for interoperability with third-party devices. It is near impossible to allow their services to communicate and control devices outside of their minimal curated list of supported devices. A method of using custom search queries to control actions on DIY devices was desired.

This system has the opportunity to save people money, as well as reduce their dependency on cloud-based products that they have no functional control over. Allowing users to integrate their own personal or third-party devices reduces their need to purchases the arguably overpriced big-box brand devices sold commercially.

A drawback of common integration systems is the requirement that all data be sent to the cloud and processed on some private remote server. Google's and Amazon's APIs are both "magical" entities where data goes in and data comes out. Many potential users avoid such services due to privacy concerns. By self-hosting a small, open source, easily verifiable, codebase, such users can be assured their data remains private, other than what they explicitly configure to be sent elsewhere.

There is a market for software services that are open source and do not *necessarily* rely on the cloud or far away, magical, black-box servers for functionality. The software created for this thesis is able to run locally on a computer, isolated from the internet if a user so desires (albeit at a loss of those specific utilities that require such a connection).

Furthermore, many integration utilities restrict users to a small, curated list of approved, commercial products. This restriction on consumer choice limits competitive solutions and increases upfront costs for the user. With the ability to integrate third party or self-made devices into the system, users are less tied to purchasing expensive commercial products, and more free to develop their own solutions are purchase independent, community designed products.

Currently, most solutions that attempt to integrate multiple IoT systems function with only off-the-shelf products that users can purchase from retailers. Users are often left without the ability to use

lesser-known products or their own creations. For example, the Google Home ecosystem only allows users to make use of a curated selection of products easily, but much effort must be exerted to attach any other device outside that small selection. This project seeks to allow users to use DIY devices and integrations alongside, or instead of, commercial products with ease. Second and third party devices are treated as first class citizens right alongside the big name products.

Real World Applications

This project is intended to satisfy the requirements of users who wish to automate tasks in their household. Such users are often, but not always, interested in extending, modifying, and developing for their systems. This project intends to provide features casual users who simply wish to utilize an existing solution, as well as those hobbyists.

Use cases

After conducting a brief poll of other students, a few desired use cases of a potential system were collected. Respondents wanted the ability …

- To have house lights turn on automatically when entering the house, except during the day when it is bright enough off outside to light the inside of the house.
- To tell Google Home to go into movies mode, and have the lights dim and the TV turn on.
- To painlessly listen for Amazon Dash button presses and start a routine.
- For a current draw meter to send a notification to a user when a device has drawn more than an acceptable amount of power
- To have the lights in the users room turn on slowly when their smart coffee pot has finished brewing its morning cup of coffee
- For a microphone in a room to turn the room lights red if the volume became too loud during a range of hours
- To make lights flash red when an issue is added to the users Github repository
- To have a camera start recording when a door was opened

All of the above use cases are possible with any implementation that follows the architecture described in § System Architecture. While not all of the functionality is directly supported in the implementation provided with the project, the implementation could easily be extended to allow a user to accomplish their goals.

To clarify, these use cases are not presented to be requirements of the implementation of the system, but guiding objectives for the design of the architecture. The desire is that with an extensive enough implementation of the architecture presented in this thesis, every one of these cases could be supported. The implementation that is provided with this project is a demonstration of functionality, not of completeness.

Scope of Project

This project was scoped to be composed of the following:

- The definition of an architecture, which, with a thorough implementation, would allow for the integration of smart home and IoT devices, both commercial and user designed.
- A demonstrative implementation of such an architecture that provides a proof of concept for the functionality of the architecture. This would be comprised of:

    - A collection of physical modules to provide useful around the house functionality, such as a door state sensor, temperature sensor, or a thermostat. These devices would include the physical board and components. These modules would be designed, printed, and assembled completely from scratch, as a demonstration of the DIY-accommodating abilities of the server.
    - A server used to manage routines and events with those devices, as well as integration with APIs for commercial smart home products.

Some aspects of this project are aimed at hobbyists who have the necessary knowledge and desire to tinker with the inner workings of the system. As it stands, in order for a user to run the software they need a basic understanding of Linux. If they wish to add functionality to the system, e.g., by adding a new device type, they also need to have an understanding of the programming language Python.

Terminology

In this report, the domain-specific terminology will be used. A collection of commonly used phrases and their meanings is provided here for the reader's convenience.

- IoT - Internet of Things; refers to the interconnection of embedded devices as a system that works together and feeds itself information
- ESP8266 - Line of Microcontrollers produced by Espressif. Each has a full TCP/IP stack allowing the microcontroller to communicate with other IoT devices using a wireless connection
- Python - Programming language that is used for the server-side and controller-side portions of the software. See § Implementation | Python Background for further information
- C++ - Programming language used for the embedded software running on the ESP8266 microcontrollers. A compiled language is required as the software runs directly on the microcontroller without an interpreter

Related Work (2p)

Software Competitors

IFTTT

IFTTT (If This Then That) is a web-based service that allows users to tie conditional statements to device actions, creating routines, called recipes or applets, using internet connected devices. While IFTTT has very simple functionality, only supporting single conditionals, relying on IFTTT is completely cloud based – users are unable to host the service themselves. IFTTT outshines its competitors with the vast amount devices it supports natively[1], as well as the simple and painless service it provides for users to integrate their own devices.

Creating an applet is very straightforward with IFTTT. A user selects a service (e.g., Youtube, Spotify), a trigger (e.g., Channel uploads video), and then creates a list of what happens when the trigger is *triggered*, again using IFTTT's list of compatible services. Because of the simplicity of the IFTTT's architecture, the platform is able to allow users to easily integrate their own devices using their Webhook service, which uses HTTP requests to trigger services or perform actions.

Tasker

Tasker is an Android application that allows users to automate actions within their phone. Users can create sequences of events, called profiles, that are run after specific triggers, such as GPS location proximity, time of day, app open/close, headphone plug, and many more. These profiles are designed using the device's powerful UI editor.

OpenHAB

OpenHAB is a utility to integrate IoT devices together in much the same fashion as this project. It has very similar goals to this project in terms of end functionality. Users can create routines that run a sequence of actions when internal or external state changes occur. Routine configurations (called *rules*) are configured in files using the following syntax:

```
rule "<RULE_NAME>"
when
    <TRIGGER_CONDITION> [or <TRIGGER_CONDITION2> [or ...]]
then
    <SCRIPT_BLOCK>
end
```

Each rule has a list of triggers; when any one of them occurs, a script block is executed. Rules are written in text files placed in a static location and are loaded into the system on startup. Each file can contain one or more rules, allowing for easy organization.

[1] IFTTT currently supports of 360 services natively

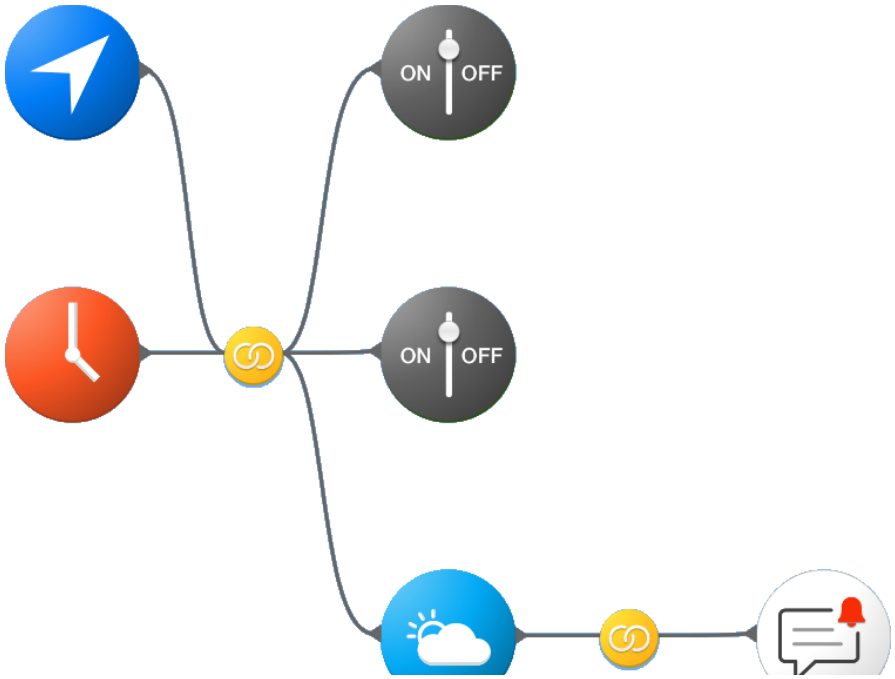
OpenHAB allows the use of both commercial and DIY devices through the creation of bindings, which are similar to the device drivers discussed in § Architecture. It is also designed to be self-hosted, relying on users to provide their own server – much like *Idiotic*. OpenHAB is written in Java and is completely open source. Its backend is based off of the Eclipse Smarthome Project. Because it is written in Java it requires the Java Runtime to be installed.

OpenHAB is very similar to the server and controller this of this project in terms of functionality and although there is a great deal of overlap between OpenHAB and this project, there are some distinctions. OpenHAB is implemented to rely on Java-based Eclipse IoT toolkit as a backend, while the implementation of this project relies on Python scripts written. Python, *subjectively*, is a more modern language, has more powerful reflection and introspection abilities, and has a larger hobbyist community and ecosystem, than Java. (More about this is discussed in § Python Background.)

Another difference is that the bindings that OpenHAB uses to add devices to a system are clunky and bloated by very nature of being written in Java. The implementation I present using Python allows developers to focus on the functionality instead of the boilerplate Java code that is required to get started. This allows for faster development and iteration within a system. Using Python for implementation keeps the code base lighter, allowing for faster software iteration.

Stringify

Stringify is a project that is meant to allow convenient chaining of actions between IoT modules. It is not open source and has limited support for self-developed modules. Stringify uses a system of routines, called *flows*, that are configured using a graphical interface. Each flow consists of a set of input conditions, and a set of output actions. An example of a flow, taken from Stringify's website, is shown below (note that the visual representation is meant to be more of a summary and many of the details, such as values, are hidden until clicked):



This flow expresses the following functionality:

```
WHEN the datetime is a specific value;
AND ONLY when the user is at a specific location:
       THEN increase the brightness of a light;
       AND increase the brightness of another light;
       AND generate a weather report;
              THEN send the weather report using a Push Notification
```

Stringify is very similar to IFTTT, using visual editor instead of the paginated selection screens that IFTTT uses. Stringify also supports multiple input conditions that must all be satisfied before a flow is run, something that IFTTT does not support with its applets. Chained operations are also supported in Stringify flows, where the results of one is sent to another.

Stringify is completely cloud-based and does not have support for allowing a user to host their own server. It is also completely closed-source.

Home Appliances

Temperature Sensor and Thermostat

Nest

The *Nest Learning Thermostat* is a WiFi connected, self-learning thermostat that optimizes heating and cooling patterns using machine learning, with the goal of saving the user energy and money. It is a digital thermostat that can also be controlled through other devices, such as a smartphone. The current-generation product retails for $259. There is an add-on external temperature sensor that retails for $39. The Nest provides a REST API to allow other devices to query it for information or to control its behavior.

While the thermostat developed for *Idiotic* does not utilize any machine learning for optimization, it is considerably cheaper, costing a fraction of the retail price of the Nest. It also does not require an internet connection to function like the Nest does.

IR Repeater

IR repeaters are fairly common devices that are usually used to extend the range of television remotes. The most common type is a device consisting of a sensor, an emitter, and a power supply. The emitter is placed on top of, or near, the television's IR sensor and the whatever the receiver picks up is mirrored immediately on the emitter. These devices can allow remotes to work even without a direct line-of-sight to the television's sensor.

Hardware (Microcontroller)

The embedded microcontrollers that are used for the hardware in the project are ESP8266-07s and ESP8266-12Es (nearly identical models of the same microcontroller). The ESP8266 microcontroller was chosen because it is a very cheap (less than $5 per, including shipping), has a full TCP/IP stack able to easily connect to a wireless network, and has a well established maker community surrounding it.

There exist a number of commercially available products that provide similar functionalities, but each is inferior to the ESP8266 in the context of this project's requirements.

Particle Photon

The Particle Photon is a development platform intended for use within IoT projects. It has a full TCP/IP stack, is fully open source, and has a thriving community. The downside with the Particle Photon is the price, costing $19 per unit.

Onion Omega2

The Onion Omega2 is a tiny microcomputer that runs a trimmed-down version of the Linux kernel. It can be thought of as a tiny server that runs Linux. Because of this, it has the ability to run any program executable on Linux, not just compile machine code targeted for the platform. Had I needed this ability, it would have been exceptionally useful, but I did not. The Onion Omega 2 retails for around $7.50 not including shipping.

ESP32

The ESP32 is a 32 bit, beefier version of the ESP8266. It is slightly more expensive, at around $10 per unit, and has a few more built in features, such as a temperature sensor, native PWM, and ethernet. However, because I did not need the extra features for every part, nor the extra processing power, I opted for the ESP8266.

System Architecture (7p)

The system is comprised of three major sections: A controller layer, a device layer, and a server that routes information between the other two layers. The controller is the backbone of the system, managing device drivers, events, and routines. Each node in the device layer is a device connected to the system such as a temperature sensor, thermostat, or Google Home. The server is the communication layer between the controller and device layers, routing information to and from devices and their corresponding drivers in the controller. The architecture for this system is presented visually in § UML.

Devices in the system are either physical, such as a temperature sensor or a smart lightbulb, or mainly software, such as a Google Home. The server and controller are agnostic of where and how the data originates or what they are controlling. Devices simply report information and/or perform action(s). For example, a thermostat device has the ability to change the active mode of an HVAC system, while the a temperature sensor simply sends its current temperature reading to the server. A Google Home can send information to the server. Devices provide a means of allowing the controller to interact with the real world, either as an input or as an output – they're not simply meant to perform computations.

Events are tasks for the system to carry out. They are collections of atomic operations, referred to as actions, for devices to perform, either setting attribute values or executing behaviors. Each event has at minimum of one action and each routine has at minimum of one event. Each event in the routine must be executed in order and each action with each event must do the same. This provides users with expectation of consistency in their routines and prevents negative effects such as race conditions where events trigger other routines in unexpected fashions.

Idiotic, an implementation created for this architecture is described in detail in § Implementation.

Controller Layer

The controller manages the functionality of the system, keeping track of the devices, events, and routines; making sure they are utilized at the proper times. The controller layer is comprised of two major sections: the actual controller instance and its collection of device drivers that enable its functionality. Because of the manner that routines are interested in data from the controller, the system follows the publish–subscribe design pattern.

Device Drivers

The controller needs to know how to interact with its devices. The device drivers are intended to abstract communication with real-world devices from the rest of the system, acting as a translator between controller and device.[2] Also note that the drivers are different than the server. The drivers tell the controller how to communicate with the device *using* the server.

Each device driver includes one or more *attributes* or *behaviors* that define its functionality.

[2] Note that each instance of a device driver corresponds to a individual device, as opposed to typical device drivers such as those within the Linux Kernel that make use of one instance for as many devices as exist

Attributes

Device attributes represent a state that the device can hold. Examples of attributes would be the reported temperature of a temperature sensor or whether or not a door sensor believes a door to be ajar. Each attribute can express the following functionalities in order for the controller to use it:

- Getter function – Called to request the attribute's value from the device.
- Setter function – Called to instruct the device to change the value of the attribute
- Updater function – Called when the device sends the server an updated value of the attribute. When this function is called, any subscribed triggers are run as well.

Note that the update function is called by the device, while the getter and setter functions are called by the driver. Two way communication is necessary with the drivers.

All of the above functions are optional for an attribute, as long as at least one is utilized. For example, a temperature sensor's *temperature* attribute would only need to define the *updater* function to periodically report its current temperature to the system. It could optionally define the *getter* function if a device in the system wanted the temperature at an exact instance (as opposed to relying on the most recently reported value or waiting for a new one). A temperature sensor would have no use for a *setter* function and would leave it undefined.

On the other hand, a thermostat could make use of the *setter* function to allow the system change its current output from AC to Heat, for example.

Behaviors

Device drivers can also include *behaviors* which are executed to instruct the device to perform a task or operation. Behaviors are very similar to the *setter* function of an attribute, but are intended to be used for operations that are more complicated than setting a single value, i.e. when the device should *do* something instead of *setting* something. Behaviors can also take more than one argument/parameter. For example, if a smart lightbulb supported the ability fade its lights from one color to another, a behavior called *pulse* that took two colors as arguments could be defined.

Routines

Routines are the most important entity in the system. Each is a collection of events that is executed on a state change in a system. A routine is are subscribed to attributes using triggers. When the attribute's value is change, the trigger is alerted. If the new value meets a set of predetermined conditions, the trigger instructs the routine to run.

An example of a routine is as follows: A user wants to create a routine to automate a set of tasks she performs every night when she says "Goodnight" to her Google Home:

- Turn off all the lights in the house
- Set an alarm to wake up in the morning
- Send a message to her significant other saying "Goodnight"

Her system already has device drivers for her Google Home, Philips Hue smart lightbulbs and her DIY smart alarm clock, as well as a driver that provides a means of sending messages on her messaging client of choice, AOL Instant Messenger. A routine is created with the following components:

- A trigger is subscribed to the Google Home driver's *current_message* attribute, with a transition of [any command] → ["Goodnight"]

- Three events are created
    - TurnOffAllLights: A sequence of actions that set each light's *On* attribute to False
    - SetAlarm: Sets the alarm clock's *AlarmTime* to 8:00 AM and its *AlarmOn* attribute to True
    - SendGoodnightMessage: Calls the messenger's *SendMessage* with a value of "Goodnight"

Her routine is automatically added to the system and is ready to execute.

Triggers

Triggers are used to alert routines of state change within the system. Each routine uses a trigger to "subscribe" to a device driver's attribute's changes. Whenever the subscribed-to attribute's *updater* function is called, the trigger is alerted so that it can execute its routine.

However, a trigger does not simply execute its routine blindly on any change with the attribute's value. A trigger is tied to a *specific* value transition. For example, a trigger for a thermostat changing routine would subscribe to a temperature sensor's *temperature* attribute, but only when the temperature transitions from less than 70°F to greater than 70°F. A trigger is instantiated with the desired attribute, its routine, and what state change the routine should be interested in.

Note that triggers are edge triggered. In the previous temperature based example, the trigger would not be executed if the temperature stay above 70°F, only if it dipped below and than again performed the upward transition.

Events

Events are sequences of actions that can be executed by a routine. For example, an event to turn off all the lights in a house would hold a list of actions, each to command a single bulb to turn off. Just as events can hold multiple actions, routines can hold multiple events. For example, if a user wanted a routine to run when they go to bed, it might have an event to turn of all the lights, an event to set an alarm on a smart alarm clock, and another event to message a loved one "Goodnight" on a messaging platform.

Persistent-State Configuration Files

Upon initialization of the system, the controller loads information about its setup from a saved configuration, most likely a database for large-scale implementations or simply a collection of formatted text files for smaller implementations. During this process, information about saved devices, events, and routines are loaded into this system for usage during runtime. *Device configurations* describe instances of specific devices, including the name and a unique identifier. *Event configurations* describe events that the system uses in configurations, such as the transition from one value to another of an attribute of a specific device found in a device configuration file. Lastly, *routine configurations* describe routines using devices and events loaded from their respective configurations.

As an example, say a user previously configured a routine that waited for the change of state in a temperature sensor to control a smart lightbulb. The routine configuration file would reference an event that would change the smart lightbulb. Then, there would be an event in the event configuration file that would reference a Hue Lightbulb. Finally, in the device configuration file would be the specific Hue Light.

Device Layer

The device layer is the section of the system that is composed of real-world devices – the end target of functionality. Devices are anything that can be connected to the server, such as a Google Home, Philips Hue lightbulbs, and especially third-party and DIY devices. Devices connect and communicate with the server using persistent-connection websockets (reasoning for this decision is provided in § Justifications for Design Choices).

Devices routinely send their current status to the system, (roughly every 30 seconds). If one of their pieces of data changes, they immediately send an update to the server. The server can also request data from the device if necessary. It can also instruct the device to perform an action.

Server

Physical device modules communicate with the server using a websocket based communication protocol. This section describes the message protocol in terms of functionality, while § Implementation describes how this protocol was implemented in *Idiotic*.

Message Protocol

The devices and the server communicate with each other using websockets. The data is sent on the socket in the form of string encoded JSON structures. Each JSON structure is an object with one or more of three allowed keys: *set*, *get*, and *do*. The value associated with the key is an array of tasks of that type for the recipient to perform. Depending on the intended recipient of the message, these instructions are slightly different.

Server Recipient

Every message sent to the server needs to include the Device's UUID so that it can be associated with its corresponding instance in the controller.

set

If the server receives a JSON object with a *set* instruction, each task in the value array is a value of one of the sender's attributes. This is used to inform the server of attribute value changes on the device.

get

If the server receives a JSON object with the *get* instruction, a device is asking it for a piece of information. The device is asking for the value of another device's attribute [unimplemented]

~~do~~

The *do* instruction is unsupported with the server as a recipient. This is to limit all device->device actions to be handled by routines. Allowing for devices to directly instruct other devices to perform a behavior only allows for shortcuts inconsistent with the system design.

Device Recipient

set

If a device receives a JSON object with a *set* instruction, the server is instructing the device to change the value of one of its attributes. For example, if a thermostat-controlling device were to receive a *set* command, the value array could contain an entry with for changing the the *output_device* attribute to *heater*.

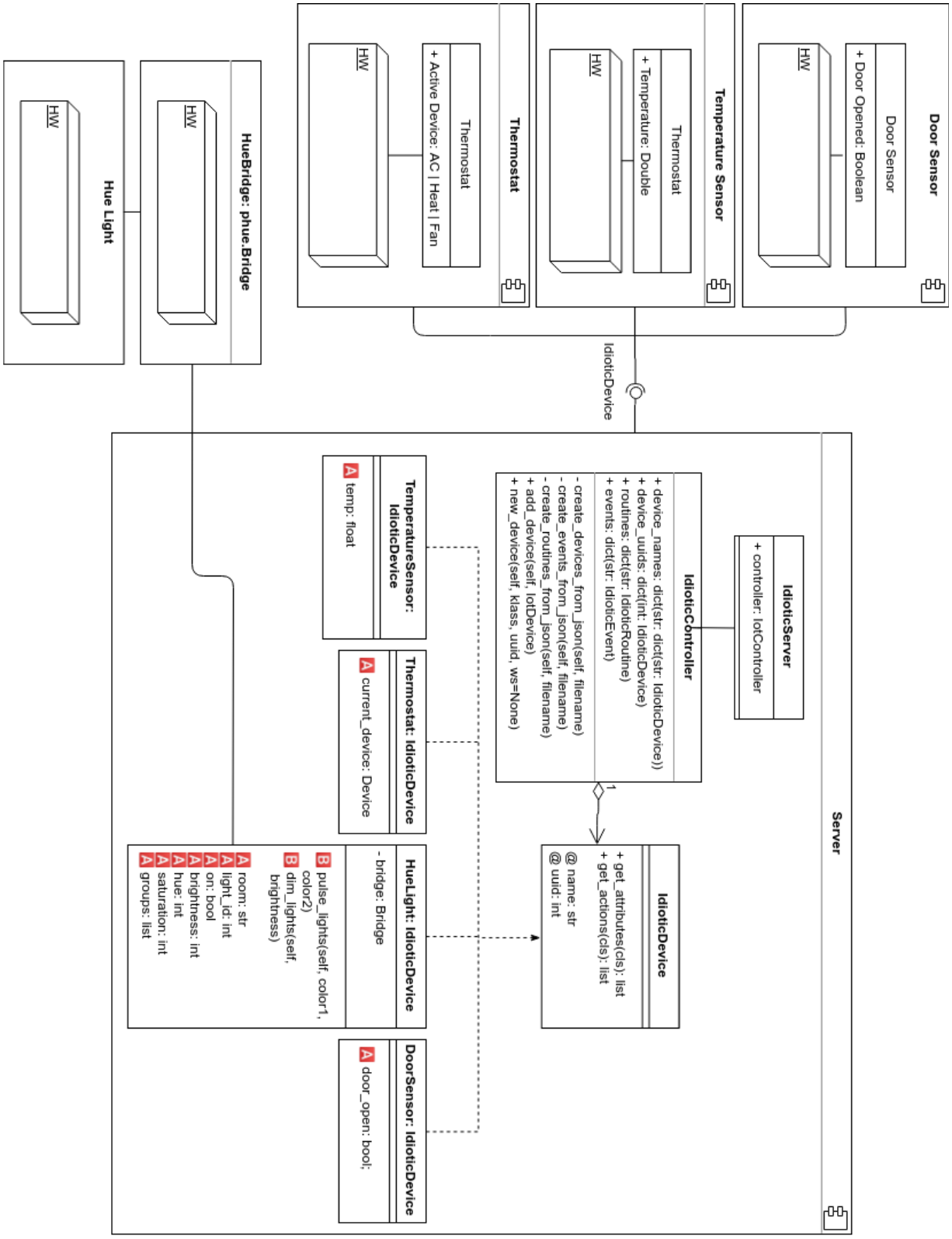
get

If a device receives a JSON object with a *get* instruction, the server is instructing the device to immediately respond with a *set* message containing the value of a requested attribute. An example use case for this would be when the server needs the value of an attribute, but the device has been configured to report the attribute's status at too slow of a rate.

do

If a device receives a JSON object with a *do* instruction, the server is instructing the device to run a behavior.

UML


Door Sensor
Door Sensor
+ Door Opened: Boolean
HW
Temperature Sensor
Thermostat
+ Temperature: Double
HW
Thermostat
Thermostat
+ Active Device: AC | Heat | Fan
HW
IdioticDevice
HueBridge: phue.Bridge
HW
Hue Light
HW
Server
IdioticServer
+ controller: IotController
IdioticController
+ device_names: dict(str: dict(str: IdioticDevice))
+ device_uuids: dict(int: IdioticDevice)
+ routines: dict(str: IdioticRoutine)
+ events: dict(str: IdioticEvent)
- create_devices_from_json(self, filename)
- create_events_from_json(self, filename)
- create_routines_from_json(self, filename)
+ add_device(self, IotDevice)
+ new_device(self, klass, uuid, ws=None)
1
IdioticDevice
+ get_attributes(cls): list
+ get_actions(cls): list
@ name: str
@ uuid: int
TemperatureSensor: IdioticDevice
A temp: float
Thermostat: IdioticDevice
A current_device: Device
HueLight: IdioticDevice
- bridge: Bridge
B pulse_lights(self, color1, color2)
B dim_lights(self, brightness)
A room: str
A light_id: int
A on: bool
A brightness: int
A hue: int
A saturation: int
A groups: list
DoorSensor: IdioticDevice
A door_open: bool;

Justifications for Design Choices

The architecture was designed with the following goals in mind:

- Modular
- Extendable
- Simple, but only necessarily for the user/hobbyist – increased backend complexity for reduced frontend complexity

Routines, Triggers, Events and Devices

Automation within an IoT system often has a strict dependency on events – `X` should occur when `Y`. Unfortunately, event driven programming is virtually infeasible without an intermediary service, which is where this and many other solutions described in § Related Work come into play. Users need to be able to configure this system, and task based solutions have become the accepted standard. Most solutions provide their users with four primitive constructs to build on:

- The device: Devices, sometimes called services, are the endpoints of the system. They alert the system when something has happened and perform actions when the system tells them to do so. They are the entity type that the user interacts with the most, the physical devices in their home or the online-services they use on a day-to-day basis
- The event: Devices need to be able to do things, otherwise they would have no purpose. Events are what the device does
- Triggers: The system needs a way to schedule the events it has been told to perform. Users don't want their lights to always be red; they want their lights to be red when something has happened. Triggers are an explicit way of allowing the user to select events that a routine is interested in.
- Routines: A collection of the devices, events, and triggers that comprise a single task for the system to perform when necessary

Communication Protocol

The communication protocol was designed to have as little bloat as possible, providing only the necessary actions, and nothing more. Devices needed to be able to perform the following abilities:

- Send data to the server
- Respond to requests for data from the server
- Carry out tasks for the server, such as setting values or performing actions

By utilizing the combination of attributes and behaviors, a device can be created that meets all the requirements.

The explicit use of attributes and behaviors in the architecture provides the system with a few important abilities with an understanding of the capabilities of each device – what it knows about; what it can do.

Websockets

Websockets are used for this project's architecture to provide the communication channels. There are a couple reasons to use websockets instead of a standard HTTP REST-based system. Firstly, if a user decides to host their server on a non-local network (e.g., an Amazon AWS instance) and their home

network is protected by a firewall (such as the those with on-campus residency), the server usually has no way of communicating with a device directly. Sockets allow the device to open a connection to the server, and then *keep* it open, allowing the server to communicate with a device through a firewall.

The second reason is aimed primarily towards the performance-minded DIY crowd. Sockets provide a significant decrease in packet overhead. Standard HTTP packets send a lot of boilerplate information with each message. The framing data for a trivial Hello World HTTP POST message is nearly 5000 bytes. On the other hand, after the initial connection start message, a websocket requires just *two* framing bytes with each message. Because each message from a device to the server are very small (often less than 20 or so bytes), the HTTP protocol increases the message size by more than two orders of magnitude. A thorough discussion of the difference in protocol sizes, as well of the sources for these numbers, can be read here[3].

Problem Definition

The system has four primitive constructs to build upon – devices, events, triggers, and routines. The system exists to satisfy the execution of an IoT use case presented/formulated by a user. A device in the system represents a heterogeneous component which is exposed to the main system as a modular component. A system may contain any number of device types as well as an unlimited number of instances of each type. Devices may be purely software based, such as one that interact with the Google Assistant framework. However, most devices will have a hardware element – a physical component that interacts with its environment or the user in the real world.

A routine is a procedure that a user wants the system to perform in response to a certain situation. The system may contain any number of routines. A trigger is, for lack of a less confusing word, an event that occurs that the user is interested in. Triggers are edge-based and signaled whenever their value changes to the desired one. An event is a set of actions that a user wants performed. When the routine is executed, a user wants these actions performed the same way every time. A user satisfies their use cases by constructing a routine, composed of a list of events controlling relevant devices, a trigger that defines when they want it to run, and a conditional that provides extra granularity for this conditional.

[3] http://blog.arungupta.me/rest-vs-websocket-comparison-benchmarks/

Implementation

This section outlines the *Idiotic* implementation for the system described above in § Architecture. This implementation is written in Python 3.

Server

The server portion of the software operates as the communication layer between the devices and the controller. Modules send HTTP PUT requests to the server to notify it of any changes in the information they measure. Modules also occasionally send the values to the server even if they have not changes as a way of informing the server that the information is still correct.

JSON Structure

The devices and the server communicate with each other using websockets. The data is sent on the socket in the form of string encoded JSON structures. Each JSON structure is an object with one or more of two currently supported keys: hello and update. The value associated with the key is an array of tasks of that type for the recipient to perform. Depending on the intended recipient of the message, these instructions are slightly different.

Server Recipient

'Hello' messages are used by a device to alert the server that it has come online. The server matches the UUID sent in the message to the websocket connection that it was transported on. This allows the server to later talk to the device, after a lookup using its UUID. An example of a hello message is shown below:

```
{
  "hello": null,
  "uuid": "13:8C:14:87:B7:AD",
  "class": "DoorSensor"
}
```

'Update' messages are sent from a device to the server to the indicate state change. The server matches the sending device to its driver instance and passes the updated value to the updater function for attribute. An example of an update message is show below:

```
{
  "uuid": "13:8C:14:87:B7:AD",
  "update": {
    "door_state": 1,
    "some_value": "some string"
  }
}
```

Device Recipient

'Set' messages are sent to a device to instruct it to change the value of one of its attributes. An example of a set message is shown below:

```
{
  "set": {
    "door_state": 0,
    "some_value": "some other string"
  }
}
```

Controller

The controller is the backend for the server. It keeps track of all instances of all modules connected to the server. The controller also manages the routines and events the system needs to know about. On startup, the controller loads the configuration for the existing routines, events, and devices from JSON files located in the `$idiotic/server/model/` directory.

Sub-Systems

Modules

Each module in the system is rooted with an ESP8266 as its backbone. Programmed in C++ the modules connect to a WiFi network. As it stands, when the ESP8266 is programmed for its board type, the WiFi is configured.

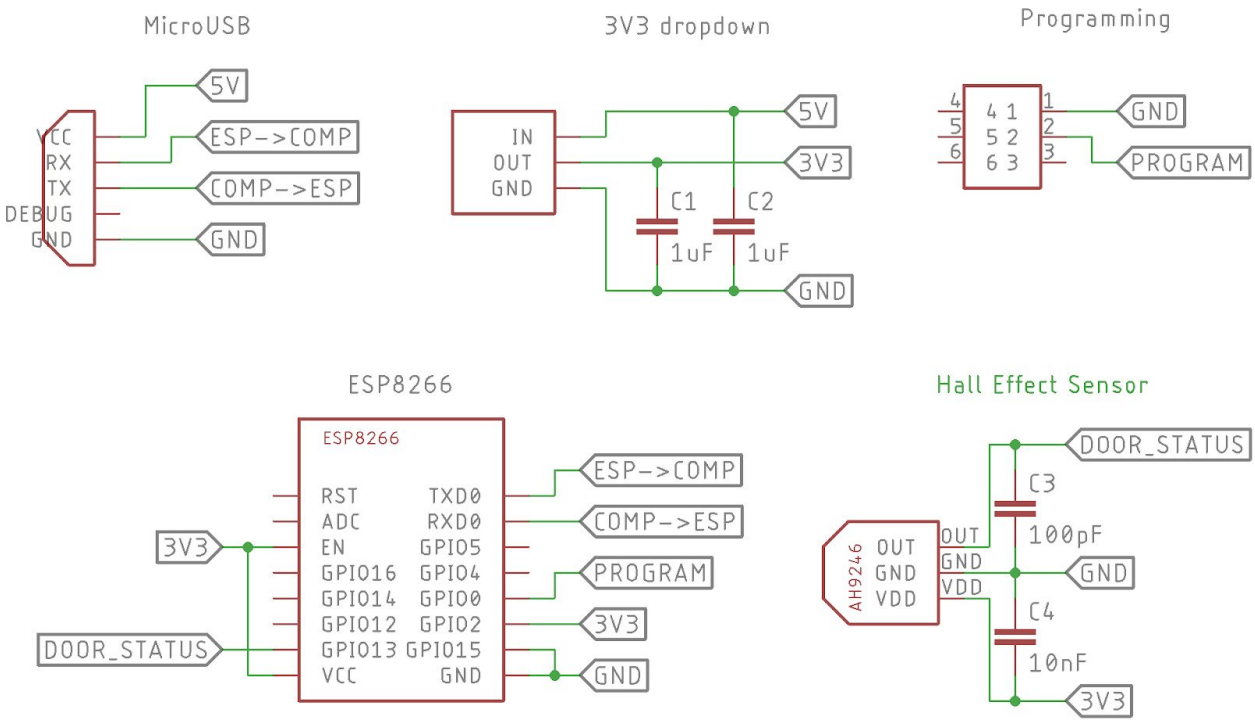
Modules communicate with the controller using HTTP *POST*, *PUT*, and *GET* requests. GET requests are used to query a module for a specific piece of information, such as its name, UUID, or the state of the door with a door sensor. PUT requests are used to set attributes to a specified value. These would be used to, for example, change the name of a device. POST requests are used to initiate an actions on the device, such as changing the desired temperature of a thermostat module.

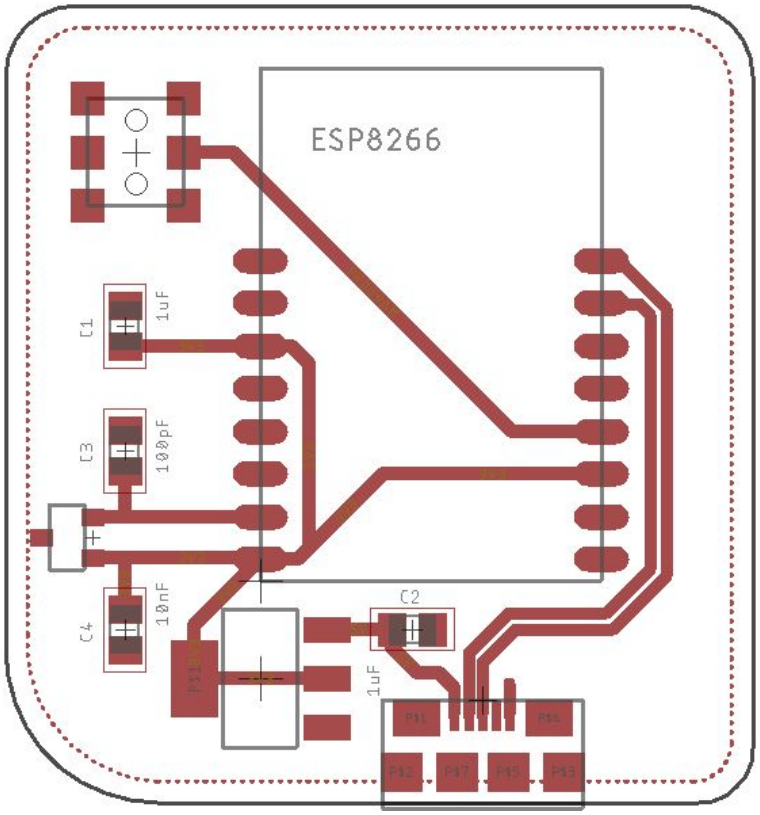
The method type is only used to notify the system of what type of data it is receiving. The actual information is sent in the body of the HTTP request as a JSON packet. The structure of the JSON packets is described above, in § JSON Structure.

Door Sensor

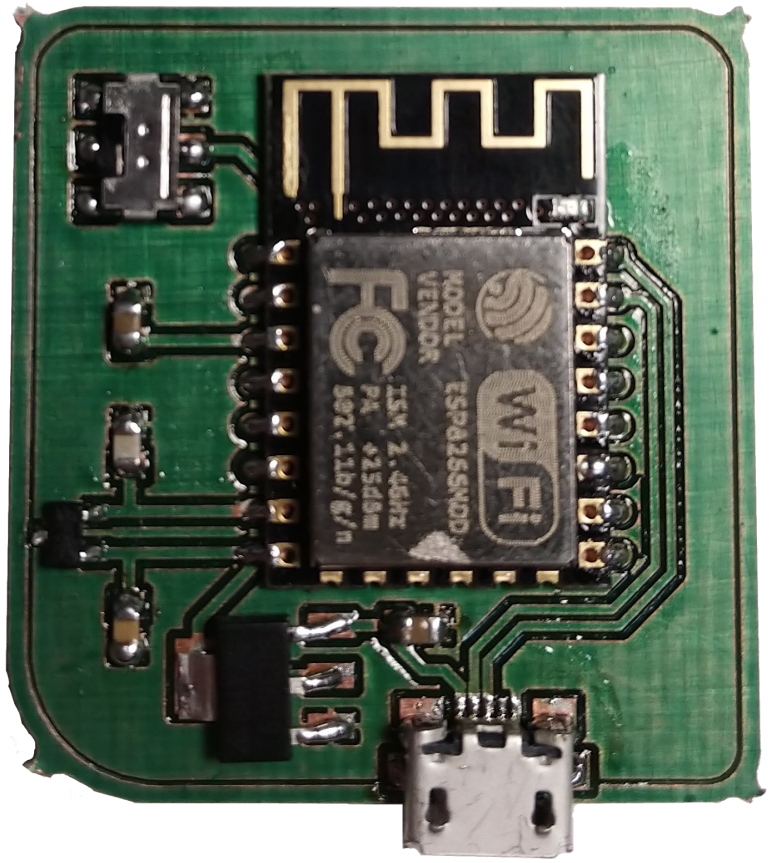
The Door Sensor module has a single purpose – to report when a door is opened or closed. Mounted to the door frame, the Door Sensor uses a Hall Effect sensor to detect the presence of a magnet attached to the door. When the door is opened, the magnet swings out of range of the Hall Effect sensor, and back in range when it's closed again. Any time the door's state is changed, the Door Sensor sends a *set* message to the server.

Eagle Schematic and Board





Assembled



Associated Software

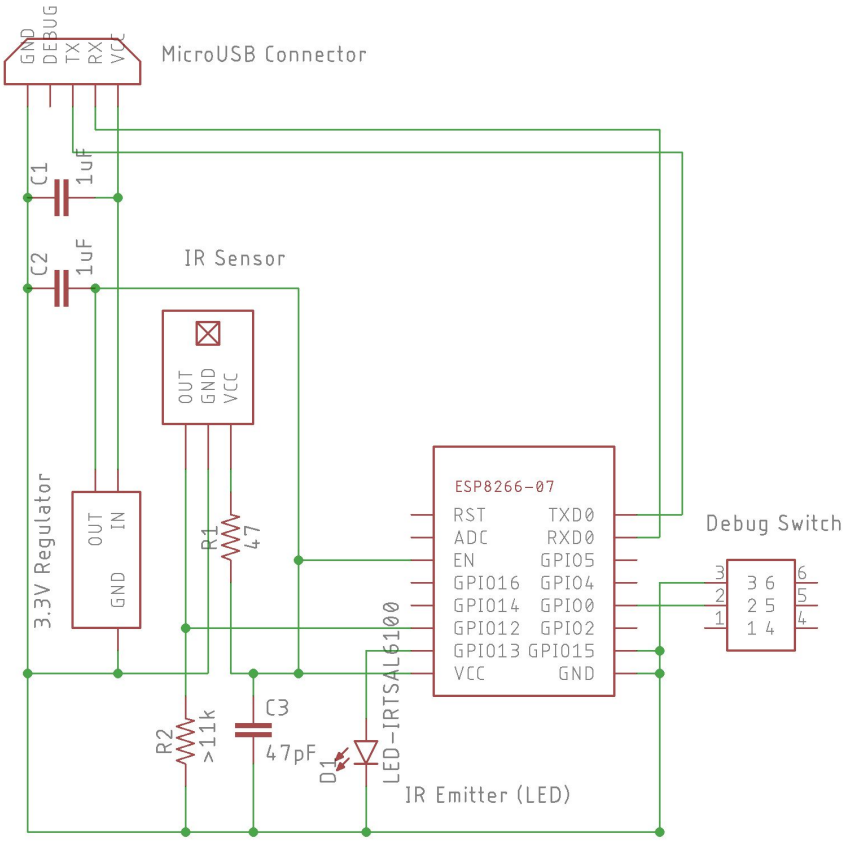
The source code for this device is available on the *Idiotic* Github:

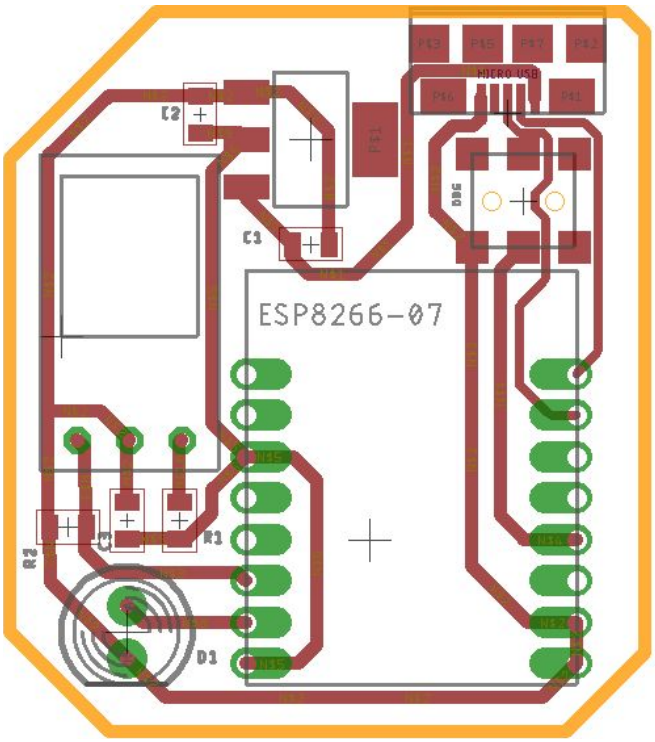
- Device Driver
- Embedded Software

IR Sensor|Emitter

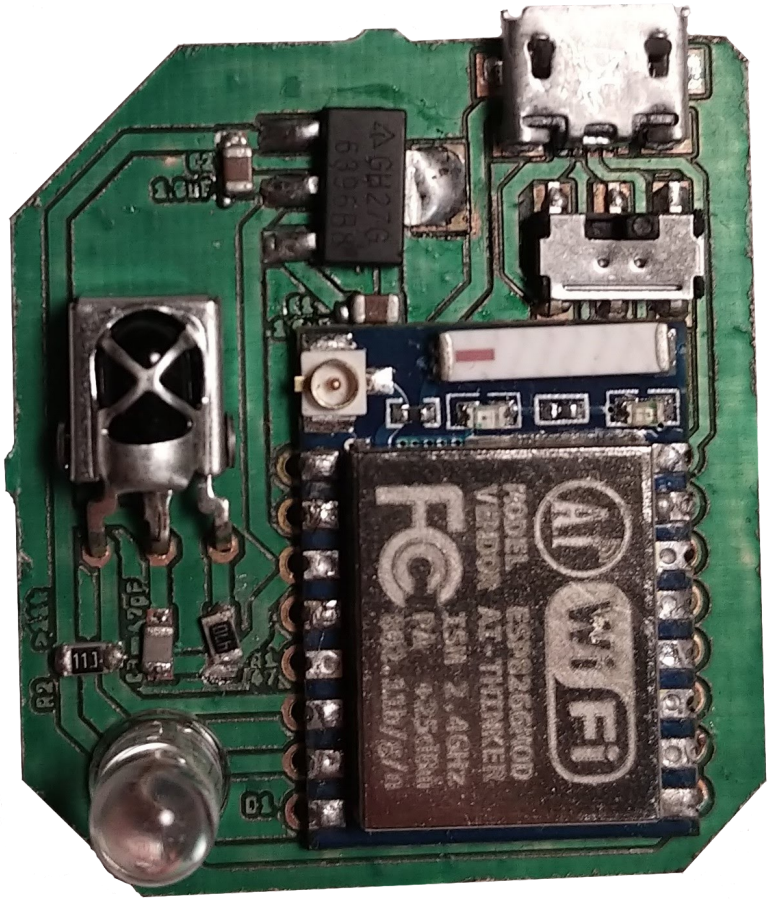
The IR Sensor|Emitter was intended to be a device which could record and play back IR signals, such as those from a TV remote. This would have been useful if the user wanted to turn their TV on and off through the Idiotic system. Work on this was scrapped after the discovery that the Google Home had a built-in feature to control the TV using an attached Chromecast and CEC. Hardware was fully designed and assembled but the software was never written for this module.

Eagle Schematic and Board





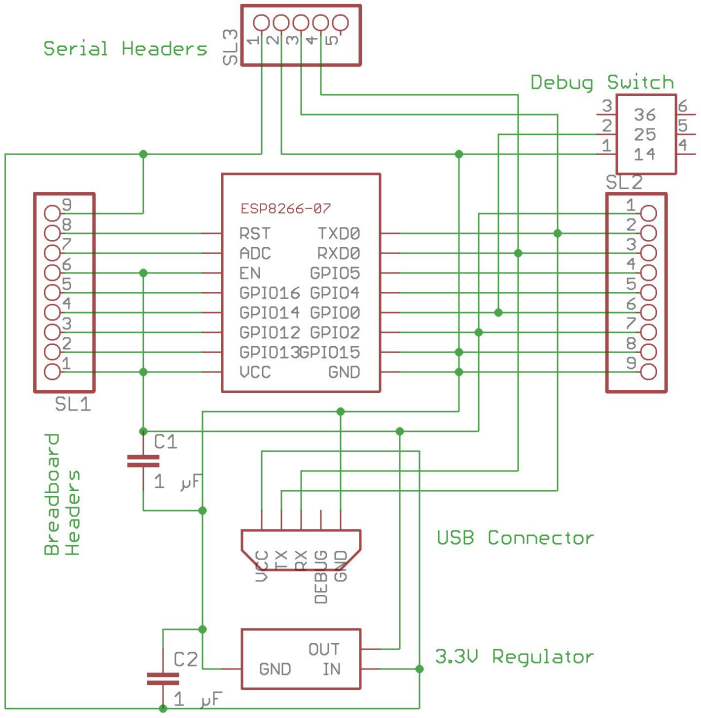
Assembled

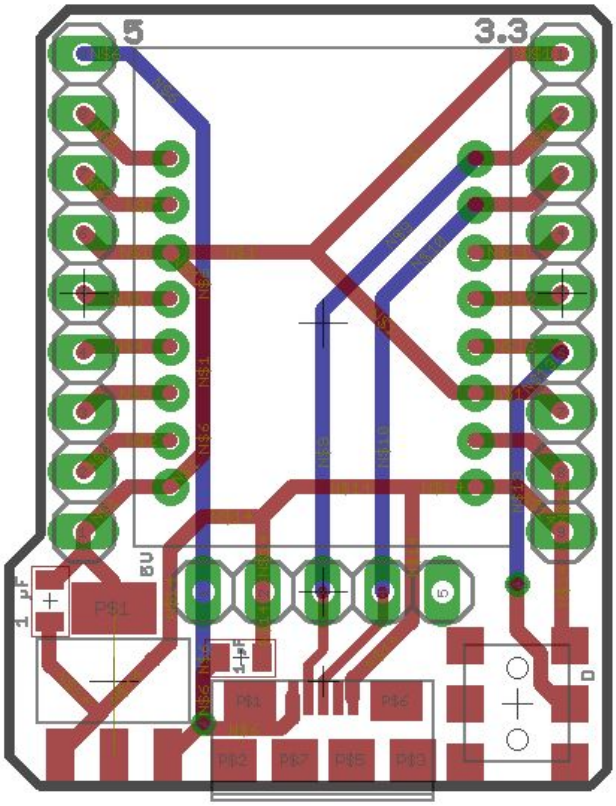


Breakout Board

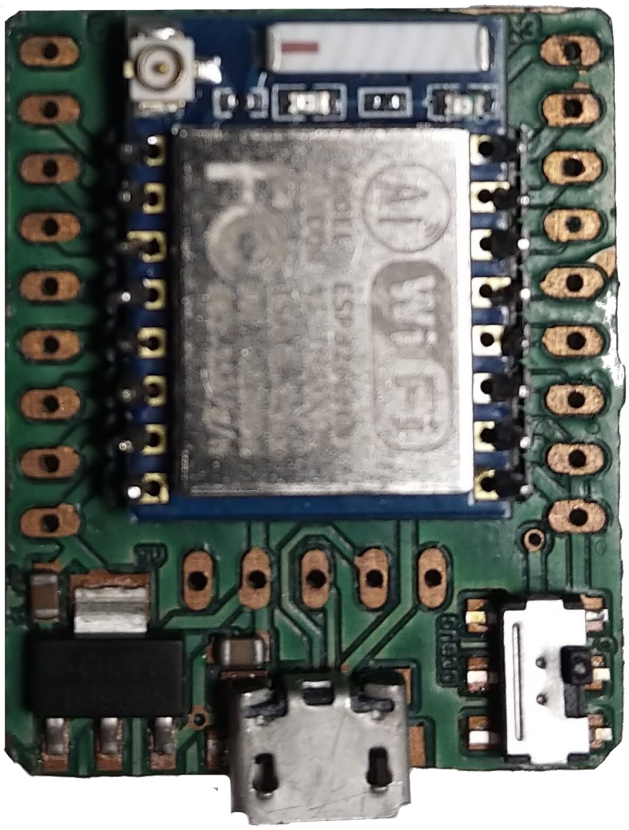
The breakout board is a component that is intended to be a development platform the the ESP8266. It has headers on the bottom that allow it to plug into a standard breadboard, as well as headers on that firmly fit an ESP8266-07. This allows a developer to use a breadboard for testing the IO on a device. It also provides a convenient method of programming a board before soldering it into a real device.

Eagle Schematic and Board



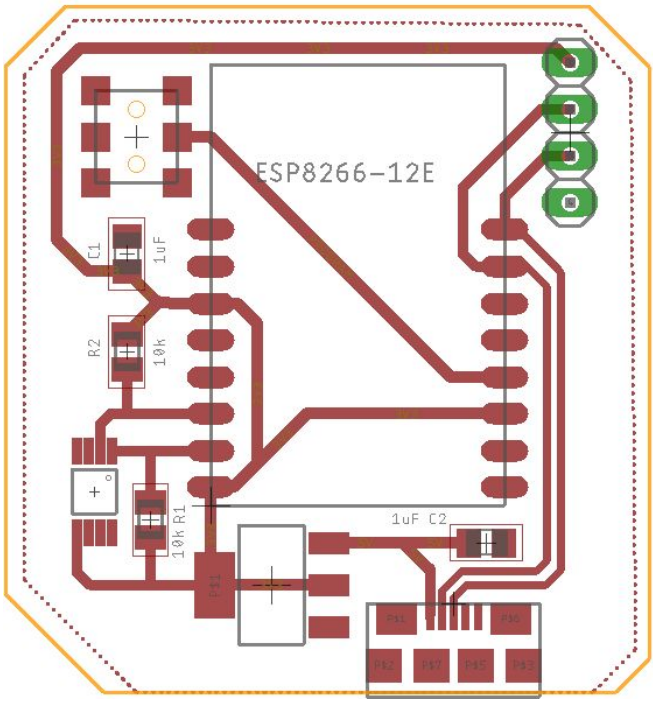


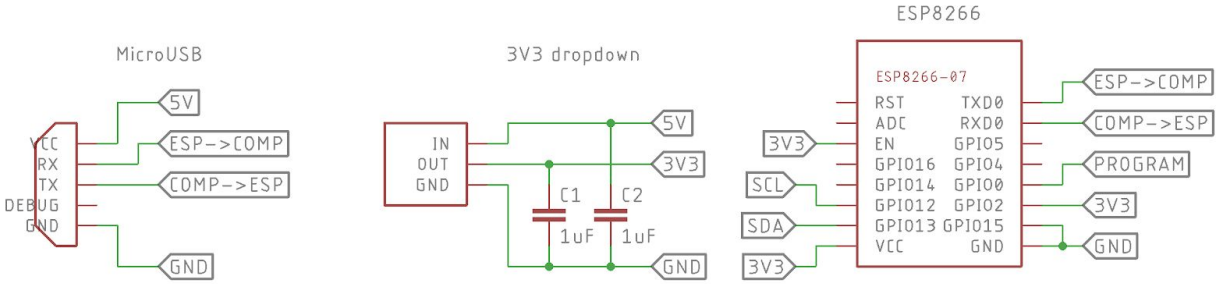
Assembled

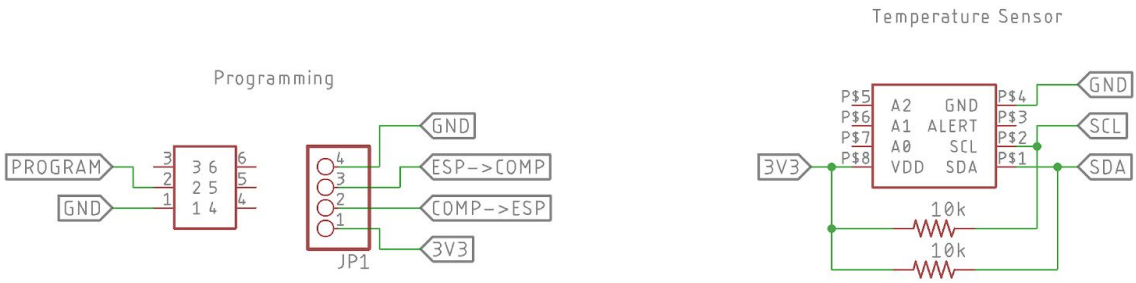


Temperature Sensor

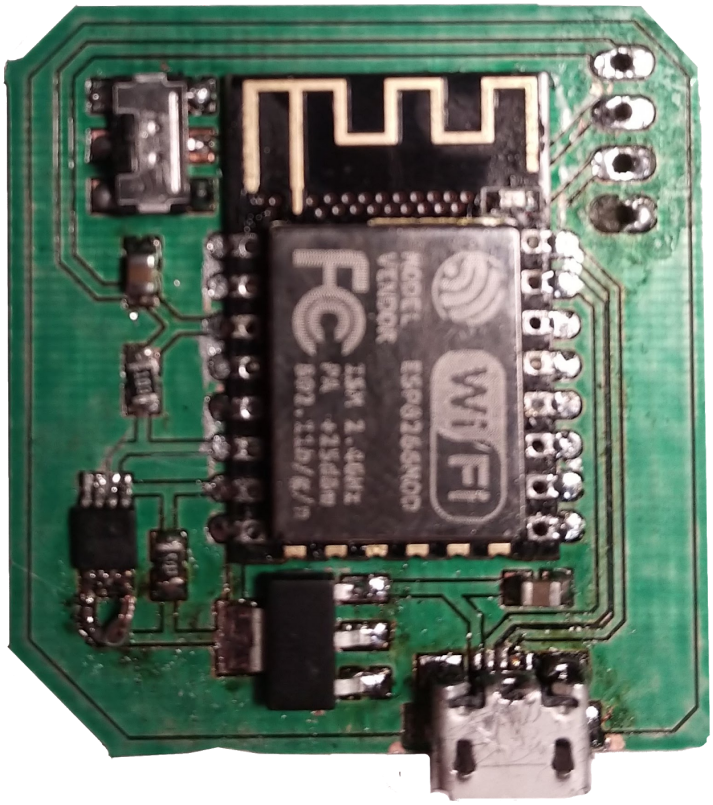
Eagle Schematic and Board







Assembled



Associated Software

The source code for this device is available on the *Idiotic* Github:

- Device Driver
- Embedded Software

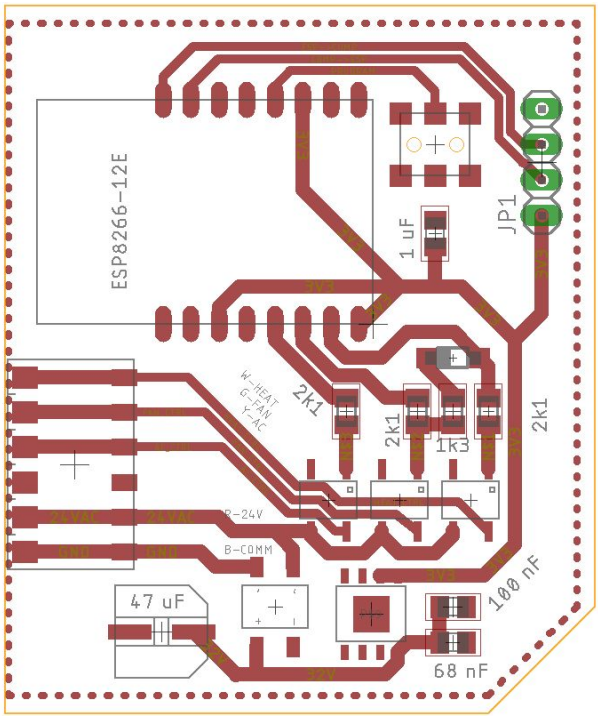
Thermostat

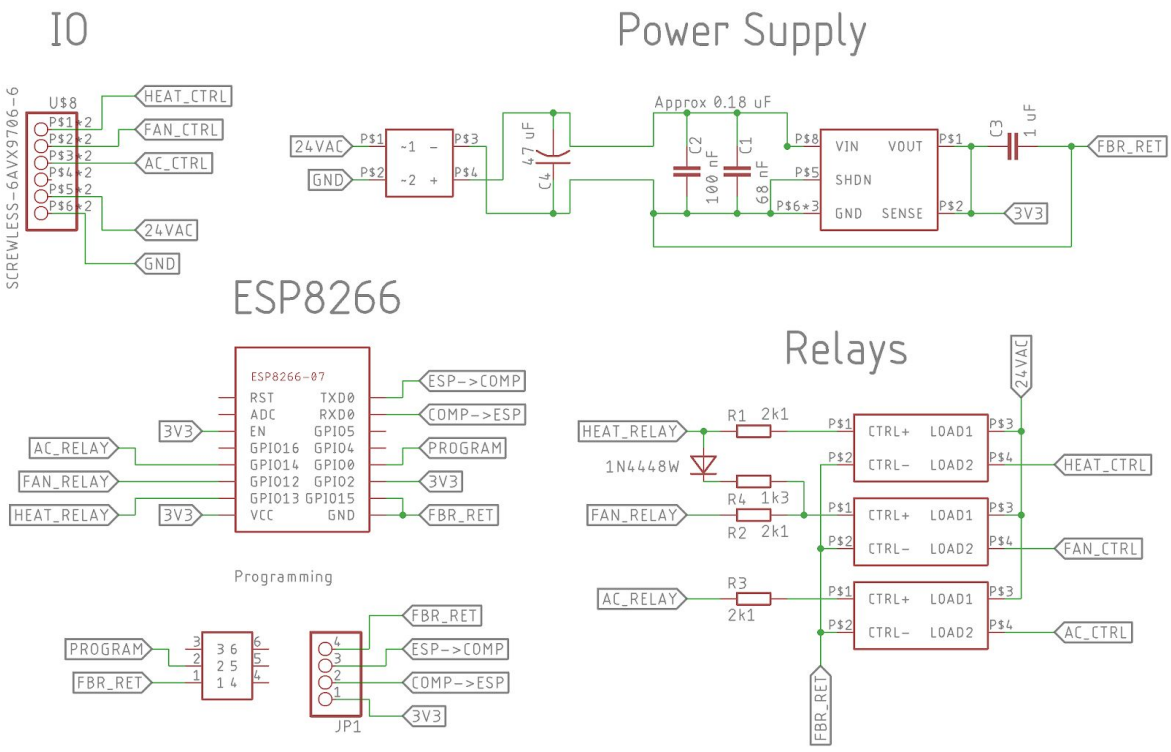
The thermostat module is the most complicated module provided with the *Idiotic* implementation. It is designed to hook into an existing HVAC system and allow the system to change the mode of the HVAC between one of four modes: AC, Heat, Fan, and Off. The system is powered using the HVAC's 24 VAC power instead of the usual Micro USB. This allows the system to replace an existing thermostat without the need of a separate power supply.

The 24 VAC input needed to be stepped down to 3.3 VDC for the ESP8266 microcontroller, requiring a full bridge rectifier (24 VAC –> 32 VDC) and a voltage regulator (32 VDC –> 3.3 VDC). Furthermore, the HVAC's inputs require the original 24 VAC to operate. This required the use of three high-voltage relays so that the microcontroller could control them using its 3.3 VDC digital outputs.

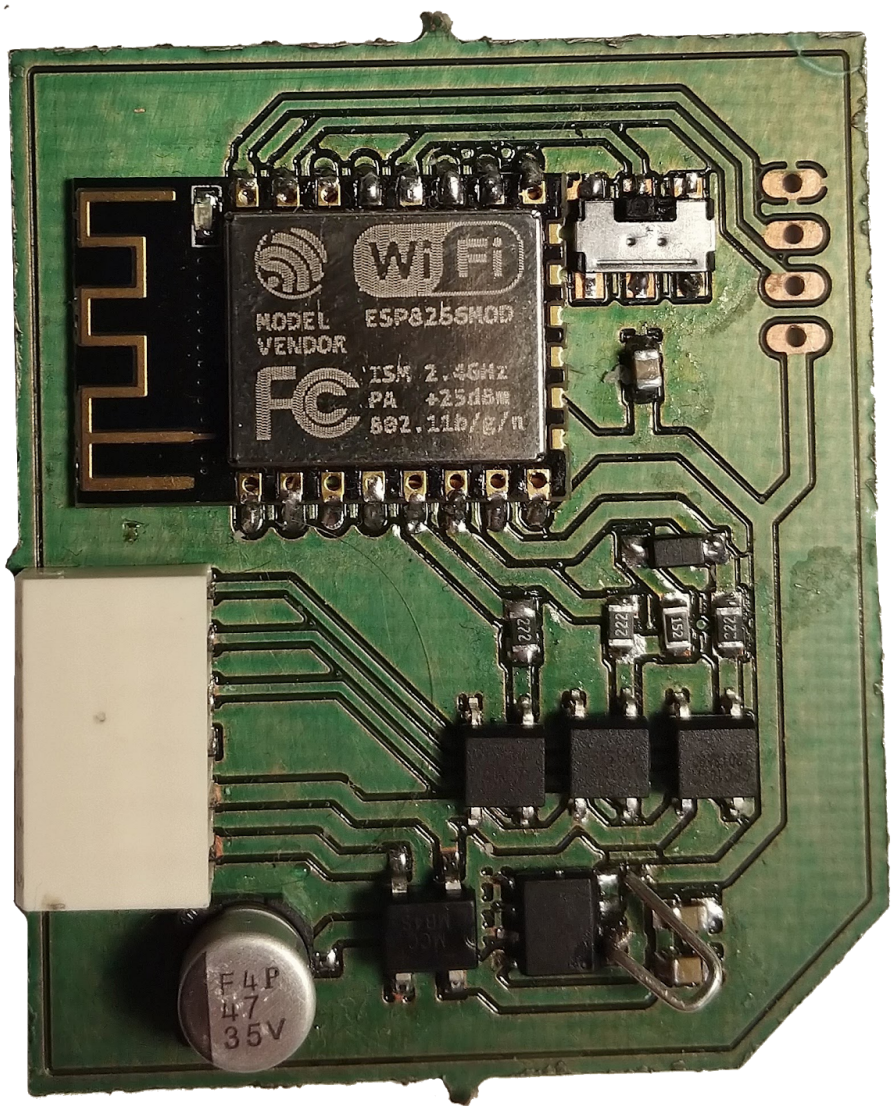
Lastly, the HVAC system in the house it was designed for allowed the heater to be on without the fan enabled, creating a potential fire hazard. The hardware designed here uses a diode between the heat relay and the fan relay to force the fan to always be on whenever the heater is on.

Eagle Schematic and Board





Assembled



Associated Software

The source code for this device is available on the *Idiotic* Github:

- Device Driver
- Embedded Software

Python Background

Python was the language of choice for the controller and server for a number of reasons. The biggest reason was familiarity. I have worked with Python in industry for nearly 3 years, providing me with a lot experience. This allowed effort to be focused on design instead of muddling with the details of the language.

The other reason Python was chosen was the reflective nature of the language. Objects can be examined, introspected, and even modified at runtime. This allows for a very modular setup where an object can determine the capabilities of another object just be looking at its attributes, and then adapt itself to fit those needs. Other languages do not provide such ease of access to these types of values.

These two reasons combined allowed tricks to be performed within the language to mold the it to the problem at task and add extra features specifically catered to the program's needs, such as the simple decorator syntax that is not so simple behind the scenes. The purpose of this section is to document some of the core Python features that I make use of in this project.

One of the main Python features that I take advantage of are decorators. Decorators are a form of syntactic sugar that wrap a function (or class) inside of another function (or class):

```
@my_decorator
def my_func():
    print("hello")
```

The above code snippet is functionally identical to the following code snippet:

```
def my_func():
    print("hello")

my_func = my_decorator(my_func)
```

Decorators are a very convenient way of tagging a function and extending its functionality. This functionality is added within the body of the definition of the decorator and is run each time the wrapped function is called. For instance, in the following example, each time my_func is run, the decorator will print the name of the function before running it.:

```
def print_name(func):
    print("Running function:", func.__name__)
    return func

@print_name
def my_func():
    print('hello')
```

In Idiotic, decorators are used to tag a method of a class in a device driver file as an attribute using a `@Attribute` decorator. `Attribute` is actually a class that takes a function as an argument during initialization and wraps it inside of a class, adding extra functionality, such as a subscribe method, or the ability to declare setter and updater functions for the function, which is now the getter method for the `Attribute` class instance. For more information on using the `@Attribute` decorator see § Adding a New Component to the Controller.

At runtime, a device driver can be introspected to determine which of its functions are actually Attributes using introspection. An example usage of this feature would be a for a user interface to show to a user the capabilities of a device, without the need for manual configuration. This can only be easily done because of how reflective Python is as a language.

Tutorials/Overview

This section serves to provide a set of introductory tutorials for using the system. Developers should read these sections while familiarizing themselves with the system. Requirements for developing are simply a computer that can run Python and an initial internet connection for downloading programs and packages.

Setting up Environment

Note
This section assumes the following:

- CLion will be used as a development environment for building and programming. It is available for free to anyone with a .edu email address and is extremely powerful
- Platformio will be used for the build environment
- You will be working with a device that runs on C or C++ code
- This section also assumes that the host computer is running Linux

You will need to adapt these settings depending on your environment

1. Clone the repository located at https://github.com/BryceBeagle/idiotic
2. Follow the instructions located here to set up the Platformio environment: http://docs.platformio.org/en/latest/ide/clion.html
    a. Make sure to initialize/generate the project within the `$idiotic/embedded` directory.
    b. Make sure to install the `File Watchers` plugin mentioned on the page.
3. If you are developing a new device, create a new directory under `$idiotic/embedded/src` with the name of your device.
4. Open `$idiotic/embedded/platformio.ini` file and change `src_dir` to the path of the device you are deploying/developing. This path is relative to `$idiotic/embedded`.

Creating the source for a new device

Note
This section assumes the following:

- CLion will be used as a development environment for building and programming. It is available for free to anyone with a .edu email address and is extremely powerful
- You will be working with a device that runs on C or C++ code
- This section also assumes that the host computer is running Linux
- This section assumes the new device uses the ESP8266 as a microcontroller. If your new device does not use the an ESP8266, make sure that it complies with the protocol defined in § Implementation

You will need to adapt these settings depending on your environment

1. If you are developing a new device, create a new directory under `$idiotic/embedded/src` with the name of your device
2. Create a new `.cpp` source file in that directory and open it for editing
3. Import `idiotic_module.hpp` to acquire the base IdioticModule class and its functionality
4. Create an instance of `IdioticModule`, passing the device type as a string.

> **Note**
> This string must exactly match the class name used in the corresponding Python file as described in § Adding a New Component to the Controller

5. In the `setup()` function:
    a. Connect the module to WiFi using `IdioticModule::connectWiFi()`
    b. Begin the connection to the Idiotic server using `IdioticModule::beginSocket()`
    c. Populate the `Idiotic::funcs` map with attribute set and get functions using `IdioticModule::function_map` structs. The key for each item in the map is the attribute name.

    > **Note**
    > - Each key must have an exact match in the device configuration file created for the controller as described in § Adding a New Component to the Controller
    > - The .get function supports lambda expressions, but for an unknown technical reason, I was unable to get this working for the .set function. As a result, only function pointers are accepted. If a class method is required, use std::bind.

    d. If serial debug output is desired, use `Serial.setDebugOutput(true)`
6. In the `loop()` function:
    a. Run the dataloop using `IdioticModule::dataLoop()`
    b. Delay the thread to for an amount of time to meet a desired period using `delay()`

An example of a functioning source file for the ESP8266 is provided below. It can be used as a template if desired

```
#include "idiotic_module.hpp"
#include "thermostat.hpp"

// Debug Serial
#define SERIAL_BAUD 115200

// WiFi
#define WIFI_SSID "your_ssid"
#define WIFI_PASSWORD "Zero0ne2numeral2"

// Idiotic
#define SERVER_HOST "192.168.1.108"
#define SERVER_PORT 5000
IdioticModule module("Thermostat");

// Thermostat
Thermostat thermostat(12, 14, 13);

// Forward declarations
void setThermostatMode(const char *);
const char *getThermostatMode();

// Setup function
void setup() {
```

```
    Serial.begin(SERIAL_BAUD);
    Serial.setDebugOutput(true);
    module.connectWiFi(WIFI_SSID, WIFI_PASSWORD);

    module.beginSocket(SERVER_HOST, SERVER_PORT);

    // Functions use to set or get thermostat mode
    module.funcs["mode"] = IdioticModule::function_map {
            .get = &getThermostatMode,
            .set = &setThermostatMode
    };
}

// Looping function
void loop() {
    //  Iterate over all functions and send result of .get functions to server
    module.dataLoop();
    delay(2000);
}

// Change which mode thermostat is set to
void setThermostatMode(const char *mode) {

    if (!strcasecmp(mode, "ac"  )) thermostat.setMode(thermostat.kAC);
    if (!strcasecmp(mode, "fan" )) thermostat.setMode(thermostat.kFan);
    if (!strcasecmp(mode, "heat")) thermostat.setMode(thermostat.kHeat);
    if (!strcasecmp(mode, "none")) thermostat.setMode(thermostat.kNone);

}

// Retrieve current thermostat mode
const char *getThermostatMode() {

    switch (thermostat.getMode()) {
        case Thermostat::kAC  : return "ac"  ;
        case Thermostat::kFan : return "fan" ;
        case Thermostat::kHeat: return "heat";
        case Thermostat::kNone: return "none";
    }
}
```

Adding a New Component to the Controller

Creating a new device driver for *Idiotic* is simple. First, create a new Python file in the `control/device_drivers/` directory. You can use the following snippet as a template for the file:

```
from control.idiotic_device import IdioticDevice
```

```
from control.idiotic_device import Attribute

class MyDeviceName(IdioticDevice):

    def __init__(self):
        self._my_attribite = None

    @Attribute
    def my_attribute():
        """Get my_attribute from device or from controller if cached

        Implicit my_attribute.getter
        """
        # In this case, the value is retrieved from cache
        return self._my_attribute

    @my_attribute.setter
    def my_attribute(value):
        """Instruct device to set my_attribute to value"""
        self.ws.send().send(f'{{"set" : {{"{active_device}"}}}}')

    @my_attribute.update
    def my_attribute(value):
        """Update value of my_attribute in controller's cache"""
        self._my_attribute = value
```

After creating the configuration file, import the class within the `__init__.py` file located in the same directory. For the above

Attributes

One of the two primary purposes of IdioticDevice drivers is to define Attributes. Attributes represent singular values and states that a device has. For example, the brightness of a Hue smart lightbulb would be an attribute, and would have a getter and a setter method. Because the Hue lights do not ping the server with status information, the HueLight class' Attributes would not have updater methods. On the other hand, a temperature sensor whose sole purpose is to report status would have updater and getter functions for its `temp` attribute.

Attributes are defined above a function use decorators. The plain `@Attribute` decorator must be used first and designates the following function as the getter function for the Attribute. To define a setter or updater function use `getter_func_name.setter` or `getter_func_name.updater`, where `getter_func_name` is the name of the function defined as the getter function.

Note
If the updater function is defined for the Attribute, the name of the getter function must exactly match the key sent to the IdioticServer that contains the status value for the Attribute.

Behaviors

Behaviors are the other primary purpose of device drivers, and are abilities that a device can perform, that are slightly more complicated that the setting of a single value that an attribute setter function provides. An example of a behavior could be a HueLight having a transition ability to slowly change between two different colors, which would be arguments for the behavior function. Behaviors are not as complicated internally as attributes and only have a single method. However, the `@Behavior` decorator must be used to tell Idiotic to enumerate the Behavior and allow Routines to call it.

Conclusion

The architecture outlined in this project attempts to provide a framework for a solution to the integration problems that currently plague Internet of Things systems. Successful implementations of the architecture will cater to both casual users, who wish to simply make use of a product, and hobbyists, who wish to extend, improve, and develop for their own systems.

The software coupled with this project, *Idiotic*, is one such implementation. It provides an extremely simple platform for developing and connecting Internet of Things devices, including both commercial and DIY. The system alleviates privacy and cost concerns by being open source and able to be self-hosted, allowing users to audit the system easily and confidently know how their data is being used.

However, while the framework is in place for expansion, the implementation is not feature complete. There is no web interface for casual users to configure their system and more modules need to be added to allow all use cases to be supported.

Results

Of the use cases presented in § Introduction, the implementation directly supports only a few. All of the remaining uses cases, including those involving products such as a Google Home, could be added to the system with ease, with the assumption that the device can be made to communicate using the desired protocol. In order to get a true idea of the successfulness of the project, real world testing would have been necessary. Without such testing, only assumptions can be made.

Future Work

Many of the objectives that were originally specified in the Prospectus for this thesis have not been implemented into the system. Some were discovered to be too labor intensive to be accomplished during the limited time frame of the project, such as the HDMI CEC components and the Google Home integration. Others were simply unfulfilled because of delays in other sections of the project. This section explores what future work could be done for the system to enhance its functionality, given more time.

Components

There are a number of components that could be implemented into the system in the future. These include both DIY PCB components and software components. Adding native support for software tools, such as the originally desired Google Home and Amazon Alexa, would be desirable.

Of the modules outlined in the Prospectus, the HDMI CEC interpreter and IR Sensor were not implemented. Because of the unforeseen complexity of the HDMI CEC protocol, it would have taken too much of my time to give it a proper implementation. The IR receiver was not finished because of a lack of time.

HDMI

The HDMI CEC handler proved to be especially difficult to work with. The main issue with HDMI CEC is that the protocol is excessively complicated[4]. Everything is implemented using a shared bus that all devices that are connected together must use. This would have added a lot of complexity to the device.

The other issue is that breakout boards are near impossible to find for HDMI devices. I was only able to find a single HDMI breakout board online, which turned out to be for mini-HDMI, incorrectly labeled for regular HDMI. In hindsight, it may have been in my best interest to design and print my own PCB, but by the time I came to this realization work had already shifted to a different component, and never returned.

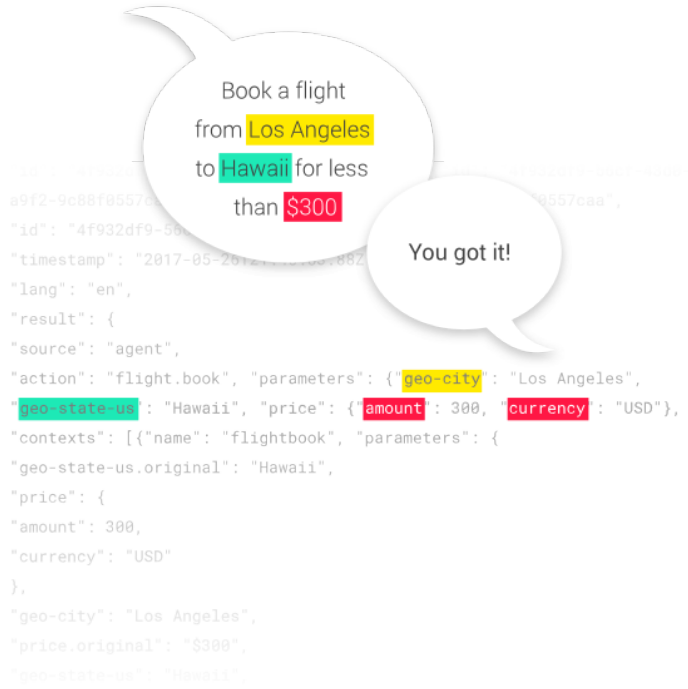
IR Sensor|Emitter

The IR sensor/emitter was not finished within the timeframe of the project. A PCB device was designed and printed, but no software beyond a couple of simple test scripts was ever written for it. The main issue with the IR sensor was the complexity of its desired tasks, compared to the other devices in the system. While the temperature sensor and door sensors merely single values, the IR sensor/emitter was planned to record and reproduce arbitrary, and oftentimes complicated, waveforms. A plan of integration into the outlined architecture was never fully realized, potentially requiring an extension of the *attribute* and *behavior* system detailed.

Google Home

A goal of this project was the ability to integrate a Google Home into the implemented system. Due to complexities with Google Home this was not completed. In order to have a Google Assistant, which powers the Google Home, interact with a third party device, a user needs to use a utility called Dialogflow to set up interactions. Dialogflow provides a means of tokenizing and parsing voice and text inputs, allowing for the of natural language queries. The service makes use of machine learning to identify key components of sentences and determine the user's intent. An example of such an operation is shown in the picture below. The user speaks a phrase, which is parsed, tokenzied, and turned into a JSON formatted string to send to a server. A visual depiction of this is show below[5].

[4] Of the 425 pages in the HDMI 1.4 specification, an ungodly 127 are devoted solely to the CEC subsystem.
[5] Taken from the main page of https://dialogflow.com



After parsing a user input for relevant data, Dialogflow allows for the data to be sent to a server for utilization using webhooks. Unfortunately, doing so is a little tricky. Dialogflow, unsurprisingly, requires that the URL is publicly accessible, but also that the service uses HTTPS for communication. In order to add HTTPS to a web server, the server needs a real domain and a certificate.

Web UI

The absence of a Web UI is one of the most glaring holes in the *Idiotic* implementation for casual users. In order to configure routines and events, a user currently needs to hardcode them into the controller or server's initialization, using Python, or needs to manually create a JSON configuration file. This severely limits the ability of the average user to utilize the system in any way, and practically forces the user to have an knowledge of the system before they can even use it.

Embedded

There are improvements that could be made to the *Idiotic* embedded library.

Currently, the library requires that the WiFi network's SSID and password be hardcoded into the device's firmware. This has a number of problems associated with it. First, plaintext passwords are never desirable, and security for IoT devices is becoming an very important topic. Secondly, in order to use a custom device, the user needs to manually edit and deploy the source beforehand. This prevents casual users from ever making use of such pieces of hardware. Furthermore, if the WiFi network's SSID or password is ever changed, every single board needs to be reprogrammed manually. This is very inconvenient.

A solution for the problem might be have the device host a WiFi network, where a user can configure these settings. After confirming, the device would save the configuration to its non-volatile EEPROM and reboot. The device would then connect to the saved network to begin serving and accepting data.

Another improvement that could be made would be the addition of OTA updates to the library. OTA updates allow devices to update their own firmware using the network, similar to how a smartphone receives updates. This would allow new versions of software to be pushed to devices without a developer needing access to the programming pins on the board.

Appendix

All software for the implementation can be found on the *Idiotic* GitHub located here. The repository also includes Autodesk Eagle schematic and boards for producing the modules mentioned in § Implementation. A direct link to each module is provided below. At the root of each module directory is a hardware specification sheet.

Door Sensor	github.com/BryceBeagle/idiotic/tree/master/modules/door-sensor
IR Sensor	github.com/BryceBeagle/idiotic/tree/master/modules/ir
Breakout Board	github.com/BryceBeagle/idiotic/tree/master/modules/breakout-board
Temp Sensor	github.com/BryceBeagle/idiotic/tree/master/modules/temp-sensor
Thermostat	github.com/BryceBeagle/idiotic/tree/master/modules/thermostat