Programming a modular Smart-Home-System

A practical implementation based on ESP8266 modules

Laura Hausmann

BRG Salzburg, Year 2019/20

supervised by Prof. DI. Mag. Martin Mooslechner

Abstract

The industry standard for home automation is called Smart Home. Many different commercial solutions exist for this purpose, though not all of them are good or useful, and most of them operate in a closed ecosystem.

To combat this trend, a fully open source home automation project (esh) was created. This was accomplished using ESP8266 microcontrollers in combination with barebones physical sensors running on the Arduino framework. The modules communicate through a configurable core, which is employing the ASP.Net Core platform.

The resulting product is shown to be usable by tinkerers and tech enthusiasts, but not so much for the average end user. Many commercial solutions are targeted towards this user group, trying to be as simple to use as possible, limiting the feature set and extensibility by definition. It was also compared against other open-source projects competing in a similar market, each targeting their own niche of the enthusiast market.

Contents

1 Preface

Smart Home is everywhere. The Internet of Things is the invention everyone seems to be talking about recently. But what if, instead of buying ready-made solutions from Amazon, Google, or a company with a similarly bad reputation regarding privacy, you could build a Smart Home solution which covers everything the commercial solutions do, but is open source and extensible? That question has been thrown around for quite some time now. So I set out to build a system that fulfills all of these requirements.

2 What is Smart Home?

2.1 History

A smart home is generally described as a system which can be installed in a home and allows its residents to control many devices and features of their home (for example: thermostats or lights) from their portable devices (smartphone or laptop). The main goals here are comfort, security, automation and efficiency.

The first Smart Home concepts were realized back in 1975 with the release of the X10 home automation communication protocol, which allowed users to install transmitters making use of the 120 kHz RF-frequency band in order to communicate with electrical relays and sockets. This made it possible to, for example, control an outlet's state based on time. The RF signal was transmitted through a home's internal electrical wiring. Flaws of this technology were, among others, the inability for a signal to cross circuit borders, the low reliability of the signal due to its susceptibility to interference and its rudimentary to nonexistent ability to communicate in both directions due to the design of the protocol.[1]

The next big development that caught public attention came with the incorporation of Nest Labs in 2010 and their first product, the Nest Learning Thermostat in 2011. Further lines of Nest products include smart smoke detectors and security cameras. It was acquired by Google/Alphabet in 2015.[2]

Around the same time a new movement began: The Internet of Things.[3] More commonly abbreviated as IoT, it is the term that collectively describes all conventionally non-internet "things" that are now available in an internet-connected form. Examples of this include refrigerators, microwaves, thermostats, doors, lighting and more.

Since then, many new technologies have emerged. These new technologies employ many different communication standards, leading to fragmentation. A few examples include the proprietary Z-Wave, created by Zen-Sys in 1999, which uses the 900MHz RF band to connect devices together wirelessly with a data rate of up to 100kbps,[4] which was later re-implemented with the open-source Open Z-Wave. Another commonly used standard is ZigBee, an open radio communication specification intended for use in the Smart Home market. It differs from Z-Wave in almost every way, except for its use of RF for communication.[5] Due to its restrictive license it did not really catch on in the consumer market, with a few exceptions, including the Chinese tech giant Xiaomi.

Another product with widespread adoption is the Hue protocol, developed by Philips in 2012. It uses the ZigBee wireless protocol as its link layer and adds

[1]Caitlin Batrinu. *Smart Home Automation with Linux and Raspberry Pi.* apress, 2013, Page 1-25.

[2]Margaret Rouse. *What is Smart Home or building?* URL: `internetofthingsagenda.techtarget.com/definition/smart-home-or-building` (visited on 07/12/2019).

[3]Caitlin Batrinu. *ESP8266 Home Automation Projects.* Packt Publishing, 2017, Page 7.

[4]Batrinu, *Smart Home Automation with Linux and Raspberry Pi*, Page 26.

[5]Batrinu, *Smart Home Automation with Linux and Raspberry Pi*, Page 27.

proprietary communication layers on top. Being primarily used for lighting control, its use cases for other Smart Home devices are rather limited. For integration with other systems the Hue Hub provides an open API, which can be freely integrated with other control software,[6] but its limited control scope and closed hardware implementation prohibits its use for this project.

Benefits of Smart Home technology

- Users are able to control their equipment remotely. This applies mainly to temperature and lighting control and security systems.
- Common tasks are easily automatable. For example, you can use a single button to toggle lighting in the entire house, or even fully automate it with presence detection. Thermostat and air conditioning can be controlled automatically based on outdoor and indoor temperature.
- Multiple actions can easily be grouped together to only need one click for everything. A common example of this is a ”vacation mode”, in which the temperature control is much less aggressive, and automatic lighting control is disabled.
- Safety systems like connected cameras and locks can improve security and alarm capabilities compared to conventional systems (e.g. through the ability to view camera feeds remotely or through movement detection alarms)

[6]Batrinu, *Smart Home Automation with Linux and Raspberry Pi*, Page 32.

3 The Project

3.1 Initial Idea

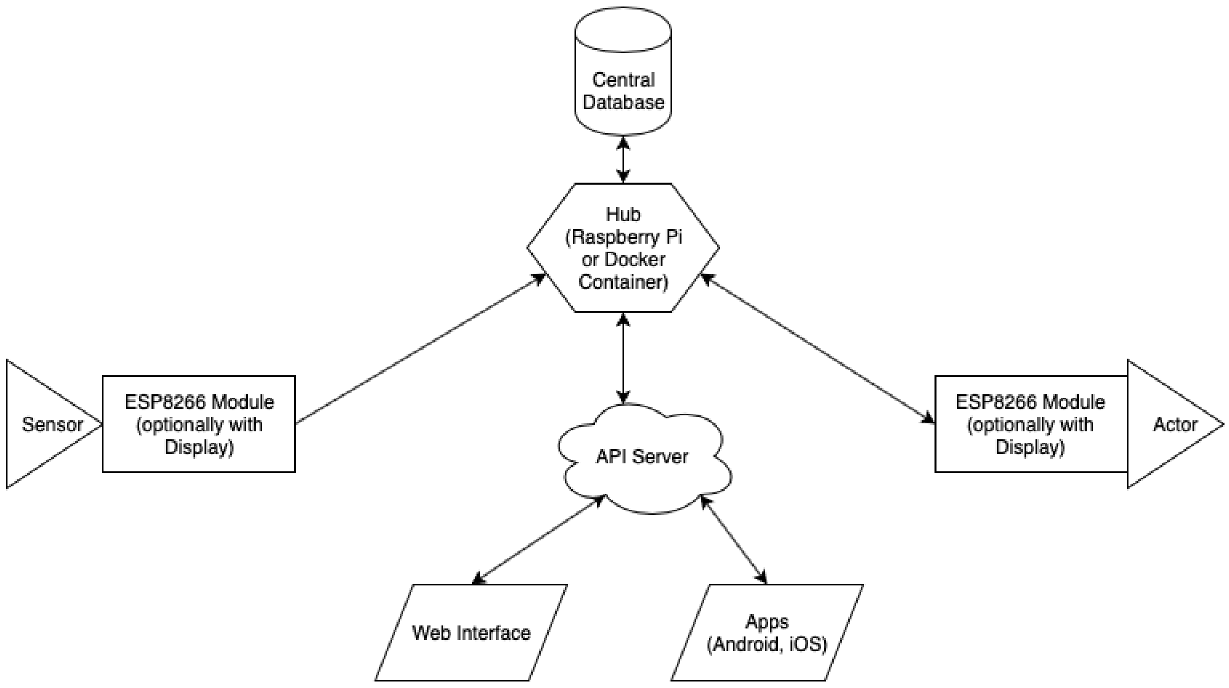
Central Database
Hub (Raspberry Pi or Docker Container)
Sensor
ESP8266 Module (optionally with Display)
API Server
ESP8266 Module (optionally with Display)
Actor
Web Interface
Apps (Android, iOS)

Figure 1: Initial Project Plan (made using draw.io)

As depicted, the initial idea is leaning on the principle that the less specifically integrated a system is, the easier it is to adapt it for many use cases.

It consists of two major hardware components: The hub, which is used to control all other components, manages and processes all events received from sensors (e.g. temperature, humidity, time). Based on this data it can execute different actions (e.g. turn up the heat). These actions can be executed based on different triggers (e.g. if temperature detected by outside temperature sensor falls below 10°C), or even more complicated ones (if outside humidity is above 80% for more than 1 hour).

The second major component are the aforementioned sensors. These consist of an ESP8266 module (an inexpensive microcontroller with built-in WiFi), as well as the physical sensor in question. This allows us to place the sensor in question anywhere in range of any WiFi Access Point connected to the same network as the hub. This setup has the major benefit over a traditional 433MHz RF setup that it can work even across an entire company building, as you can set up more APs (which will most likely be covering the entire building already) as needed.

The third major component are the actor modules. Just like sensors, they consist of the same ESP8266 we know well by now, with all the benefits of a WiFi-capable microcontroller, paired with various other modules. Examples here include a power relay (used for lighting or ”smart” outlets) or a thermostat servo motor.

The ESP8266 module’s firmware is specific to each sensor and actor type. Before flashing, it is modified to contain the hub’s IP address. When it first establishes a connection with the hub, it sends its unique MAC address which will identify the specific module from that point on. Since MAC addresses are supposed to be globally unique, it is very unlikely that duplicate addresses come up in a single network. In the web interface or an API implementation (e.g. mobile app), there is the possibility to give sensors a friendly name so they are identifiable by the end user. The sensor also sends along other important data, like sensor type, data type and measurement unit.

For the end user, there are two different main methods of configuration available: First, the web interface. The hub is running an ASP.NET Core web server (Kestrel), which manages both the API Server implementation (both for sensors/actors and configuration), the database backend, as well as the user web interface for configuration. It allows the end user to view all known sensors and actors, change their display name, view the current value and timestamp it was received, change alert ranges (e.g. temperature between -10 and +40°C, treat values outside that boundary as sensor malfunction), and most importantly set triggers.

Triggers are the core functionality of the entire system. They allow the user to connect sensors with actors. A simple example would be a light switch. When it is pressed, the module connected to it immediately sends the updated value to the hub, which processes all triggers associated with it. A reasonable example would be to turn the power relay to the room lights on. The app allows similar configuration from mobile devices, since you probably don't want to use a web interface on a mobile device, and turning on a desktop device just to, for example, disable the heater trigger because you are leaving for vacation, although advanced configuration could be limited to the web interface due to mobile interface constraints.

Statistics are presented to the end user via the open source software *Grafana*, since it is very easy to set up with an already existing database and has many clients for mobile devices already available. It also allows advanced graph analysis, and saves a lot of overhead in the Kestrel-based hub backend since it can be offloaded to a different or even external server.

The second method of configuration for the end user takes the form of an app for mobile devices, but hasn't been finished at the time of writing.

3.2 Revised Architecture & Structure

Soon after starting the project, the original structure was found to be inconvenient and limiting in some shape or form. For this reason, a few changes were made:

The Hub was renamed to Core, because it now not only manages the API server and database, but implements them directly. This goes in hand with the other major change: the Core is implemented as stateless as possible, storing only user configuration and last-known data. If statistics are required, they need to be fetched externally.

3.2.1 Modules

As mentioned before, a module consists of an ESP8266 microcontroller (with the Arduino bootloader installed) at its base, sharing many similarities with much larger and more expensive Arduino-branded microcontroller boards based on the Atmel AVR chipset, like compatibility with the Arduino IDE, the C++ programming language and libraries made for the Arduino platform. Furthermore, it has multiple GPIO (General Purpose Input Output) pins, which can be used to connect a vast amount of tiny accessories in the form of sensors, displays, buttons, RFID readers, and so on. Another huge improvement with the ESP8266 module over any Arduino board is their availability and price (depending on purchase location as low as 1$ from chinese wholesale shops like aliexpress or alibaba). Originally intended as an addon for Arduino boards, it has a much more capable CPU (32-bit RISC-based 80MHz) and most importantly an integrated Wi-Fi controller.[7] The significantly higher performance and the wireless connectivity comes with a higher power requirement though, which makes battery-powered designs somewhat more challenging, but not impossible.

[7]Espresif Systems. URL: `www.espressif.com/en/products/hardware/esp8266ex/overview` (visited on 07/13/2019).

3.2.2 Sensors

A sensor is a module which controls an external sensor. This can include temperature sensors, humidity sensors, motion sensors, potentiometers and buttons, but there are no restrictions on which sensors are supported due to the generic design of the Core and its API. It communicates with the Core via the ESP8266's integrated Wi-Fi controller. The job of the ESP8266 microcontroller in case of a sensor module is to read data from the external sensor, parse it, and send it to the Core for further processing. When it is not in use, it can fall into a power-saving deep sleep mode until it is triggered again. In this case, an hourly keepalive signal is sent to the Core to signal that the sensor is still alive, working and connected. Every sensor module has a specific firmware installed, customized with the following variables:

- The IP address of the Core (for communication), alternatively the hostname of the Core (autodiscover)
- Sensor-specific code to read and parse its values

This explicitly does not include any serial number or otherwise unique identifier, since that job is done by the ESP8266 module's MAC address, which we can assume is network-unique. This allows for multiple modules to share the same firmware, for example multiple temperature sensors.

3.2.3 Actors

An actor is a module which receives data from the Core, processes it locally and changes the state of an external module connected to it. Common use cases here would be a thermostat, an LED array or a power relay for lighting or outlets. It has all of the properties that Sensors have, including the versatility and flexibility in the range of external actors. The main differences are:

- Reversed order of operations (Process and forward received data instead of sending data)
- Deep sleep can be implemented more easily (Module checks in with Core every hour, Core pushes actions to IP address)

The actor module also sends its current state to the Core for statistical purposes.

3.2.4 Core

All modules connect to the Core. Its main purpose is to process data from all sensor modules, and send actions to the relevant actor modules. A good candidate to run it on would be the cheap Raspberry Pi 4 (around 50$), due to its 64-bit chipset and therefore .NET Core Framework compatibility. The Core processes data based on user-specified triggers, which are basic sets of instructions following the IFTTT (IF This Then That) principle. This means that every time sensor data is received, all triggers referencing said sensor are parsed and ran, and all output changes forwarded to the respective actors.

4 Practical Implementation

With the theory covered, we can now start looking into the practical implementation. The Project is split into two components: ”esh” and ”esh-modules”. ”esh” stands for Extensible Smart Home, and was chosen as a simple abbreviation for the project. ”esh” includes the core C# project, as well as the API and web interface, for interfacing with modules and end users respectively. ”esh-modules” contains the subprojects for the respective sensors and actors, using the PlatformIO IOT SDK, a bridge between the Arduino ecosystem (a popular microcontroller framework) and bare microcontrollers (like the ESP8266 we are using). The Software section in this chapter will cover ”esh”, while the Hardware section will cover ”esh-modules” with special focus on hardware challenges and implementation.

4.1 Software

”esh” is structured into three subprojects. Firstly, ”esh.core”, containing the code for the core components, as well as a few core functions to interface with modules and the database.

Secondly, ”esh.core.web”, providing the end user webinterface, employing the new ASP.Net Core architecture, which enables better and easier interfacing with the core software running in the background.

Lastly, ”esh.core.test”, a XUnit unit test project. This component is used only in the development process, to make sure all changes do not break core functionality of the project.

4.1.1 Modules

Modules have been split into two categories, Sensors and Actors, as mentioned in the initial project plan. They are uniquely identified by their MAC address, along with a 0-based ID that increases with every variable to be sent or received by the module. This allows for multiple variables under a single sensor MAC, used for example in a combined Temperature, Humidity and Air Pressure sensor package, or a multi-column display actor package.

There are a few similarities between the Sensor and Actor implementation. The main ones are:

- a unique MAC address
- type of module
- a user-definable Name, defaulting to module type
- a user-definable Description, defaulting to null
- some kind of state information
- type of value transmitted

4.1.2 Sensors

A sensor object is created in the core database object (see below) after a sensor connects to the core the first time.

```
public Sensor(string mac)
{
  Mac = mac;
  LastUpdated = DateTime.Now;
}
```

Listing 1: Constructor for the Sensor object

Complete source code: Laura Hausmann. *esh-core.* URL: `gitlab.com/lhausmann/esh/tree/master/` (visited on 11/05/2019)

The created sensor object looks like this:

```
public class Sensor
{
  public readonly string Mac; //Unique identifier of the sensor,
  public string DataType; //DataType transmitted by the sensor, e.
   g. "int", "double", "bool"
  public string SensorType; //Label for the Sensor Type, e.g. "
   Temperature"
  public string Value; //String representation of the Value
   measured, e.g. "1.0"
  public string DisplayUnit; //Unit for display purposes, e.g. "°C
   "
  public DateTime LastUpdated; //Timestamp; updated to DateTime.
   Now when the sensor changes its value
  public string CustomName; //User-definable custom name
  public string CustomDescription; //User-definable custom
   description
}
```

Listing 2: Sensor object

This skeleton structure is necessary to be able to access sensor objects from the web interface, the API and from triggers.

The actual internal code that runs when a sensor connects to the core for the first time is as follows:

1. The sensor module is powered on, reads the stored Wi-Fi SSID and credentials from the internal memory and attempts a connection to the Wi-Fi network. If it is successful, it will send a request to the stored address of the core (either in form of an IP-Address or a hostname).

2. This request takes the form of a HTTP GET request, and looks like this:

```
GET /api?mac=34:36:3b:79:98:de&datatype=int&sensortype=
    ESP8266_simple_sensor&value=18&displayunit=°C HTTP/1.1
```

```
Host: kaby.fritz.box
```

Listing 3: Sensor update message sent to the core

3. The ASP.Net Core server instance receives the request, running the following function:

```
if (isSensorQuery()) //true if "sensortype" is included in the request
        Web.Core.UpdateSensorData(Request.Query["mac"], Request.Query["datatype"],
            Request.Query["sensortype"], Request.Query["value"], Request.Query["displayunit"]); //updates the object based on the GET parameters passed above
```

Listing 4: API subpage in ASP.Net Core

This sets the above object to a state like this:

```
public class exampleSensor extends Sensor
{
  public readonly string Mac = "34:36:3b:79:98:de",
  public string DataType = "int",
  public string SensorType = "ESP8266_simple_sensor",
  public string Value = "18",
  public string DisplayUnit = "C",
  public DateTime LastUpdated = DateTime.Now, //DateTime.Now is the exact server time it is referenced at
  public string CustomName = null, //not set on initial connect
  public string CustomDescription = null //not set on initial connect
}
```

Listing 5: Completed sensor object after initial update

4. The core saves the database and the update is finished.

4.1.3 Actors

Actors are implemented similarly to Sensors (see above), and are also instances of the Module class. The main differences arise because the actor can either ask the core for an update (like sensors), or have the core push updates to it instantly. This is useful in numerous situations, for example a light switch immediately toggling one or multiple LED controller(s).

The constructor is pretty similar to that of a Sensor:

```
public Actor(string mac)
{
  Mac = mac;
  LastPing = DateTime.Now;
}
```

Listing 6: Constructor for the Actor object

The created object however is a bit different:

```
public class Actor
{
  public readonly string Mac; //Unique identifier of the actor
  public IPAddress LastKnownIP; //Last known IP address of the
   actor with the above Mac, used for pushing data directly
  public string LastKnownState; //Last known state the actor was
   in (shown in the web interface)
  public string ActorType; //Label for the actor type (e.g. "LED
   Controller")
  public string WantsDataType; //Data type the actor wants to
   receive
  public DateTime LastPing; //Last time the actor sent a keepalive
    ping to the core
  public string CustomName; //User-definable custom name
  public string CustomDescription; //User-definable custom
   description
}
```

Listing 7: Completed actor object

The key difference, as mentioned above, is the direction of communication. An actor only pings the core once every 10 minutes with a keepalive request, so that the core does not mark the actor as offline. Actual updates are sent instantly and directly by the core to minimize delay and computational effort. For this to work reliably, the core needs to keep track of the actor's IP address, which could change (for example when a DHCP lease expires) at any time.

The core also keeps track of the actors' last state for a better user experience in the web interface.

The initial handshake as well as the keepalive request looks like this:

1. In a very similar fashion to the sensor equivalent, the actor module is initialized. (see above). It then sends a similar HTTP GET request to the core:

   ```
   GET /api?mac=34:36:3a:72:98:de&datatype=int&actortype=
       ESP8266_simple_actor&state=1 HTTP/1.1
   Host: kaby.fritz.box
   ```

   Listing 8: Actor update message sent to the core

2. The ASP.Net Core server instance receives the request, running the following function:

   ```
   if (isActorQuery()) //true if "actortype" is included in the
       request
   ```

```
        Web.Core.UpdateActorData(Request.Query["mac"],
    Request.HttpContext.Connection.RemoteIpAddress,
            Request.Query["datatype"], Request.Query["actortype
    "], Request.Query["state"]); //updates the object based on
     the GET parameters passed above, also updates keepalive
    timestamp to detect offline actors
```

Listing 9: API subpage in ASP.Net Core

This sets the above object to a state like this:

```
public class exampleActor extends Actor
{
  public readonly string Mac = "34:36:3a:72:98:de",
  public string WantsDataType = "int",
  public string ActorType = "ESP8266_simple_actor",
  public string LastKnownState = "1",
  public DateTime LastPing = DateTime.Now, //DateTime.Now is
     the exact server time it is referenced at
  public IPAddress LastKnownIP = "192.168.1.155", //used to
     push data to the actor instantly
  public string CustomName = null, //not set on initial
     connect
  public string CustomDescription = null //not set on initial
      connect
}
```

Listing 10: Completed actor object after initial update

3. The core saves the database and the update/keepalive request is finished.

If the core receives an updated sensor value, it runs through all associated Triggers (see below). A request looks like this:

1. The sensor pushes an updated value to the core (see above)

2. After the update is complete, the core runs through all triggers in the database that are associated, either directly or indirectly, with the sensor that was updated. This includes all dependency (chains), but ideally runs for just 100 milliseconds, assuming the physical machine the core runs on is powerful enough. A rough overview of the process is found in the section below (Triggers).

4.1.4 Core

The core is the most ambitious part of the entire project. It should be reliable, fast, easy to use and easy to run on cheap hardware. Those requirements were a big challenge during the implementation phase, and are yet to be perfected.

At the time of writing, two out of the three subprojects included in the core are finished or in a working state (esh.core and esh.core.web), while esh.core.test is still in development, and was given a lower priority due to time constraints.

The following paragraphs outline the backend code used.

```
public readonly List<Components.Sensor> Sensors;
public readonly List<Components.Actor> Actors;
public readonly List<Components.Trigger> Triggers;

public Core()
{
  Sensors = new List<Components.Sensor>();
  Actors = new List<Components.Actor>();
  Triggers = new List<Components.Trigger>();
}
```

Listing 11: esh.core, Snippet #1

This snippet shows the initialization sequence of the core. It creates the necessary memory structures for storing Sensors, Actors and Triggers, accessed every time something interacts with said items.

```
public void UpdateSensorData(string mac, string dataType, string
   sensorType, string value, string displayUnit)
{
  // We assume the mac address is network-unique, which they /
   should/ be
  Components.Sensor sensor;

  if (Sensors.Any(p => p.Mac.Equals(mac)))
    sensor = Sensors.First(p => p.Mac.Equals(mac));
  else
  {
    sensor = new Components.Sensor(mac);
    Sensors.Add(sensor);
  }

  sensor.DataType = dataType;
  sensor.SensorType = sensorType;
  sensor.Value = value;
  sensor.DisplayUnit = displayUnit;
  sensor.LastUpdated = DateTime.Now;
}

public void UpdateActorData(string mac, IPAddress ip, string
   dataType, string actorType, string state)
{
  // We assume the mac address is network-unique, which they /
   should/ be
  Components.Actor actor;

  if (Actors.Any(p => p.Mac.Equals(mac)))
    actor = Actors.First(p => p.Mac.Equals(mac));
  else
  {
    actor = new Components.Actor(mac);
    Actors.Add(actor);
  }

  actor.WantsDataType = dataType;
  actor.ActorType = actorType;
  actor.LastKnownState = state;
  actor.LastPing = DateTime.Now;
  actor.LastKnownIP = ip;
}
```

Listing 12: esh.core, Snippet #2

This code snippet shows the functions called from the API when a request concerning sensors or actors comes in and something needs to be updated. This functionality was moved out of the web subproject and into the core project for organizational and simplicity reasons.

esh.core.components - Triggers The sensor and actor components were already elaborated extensively in their own respective sections above. The more complex components, namely Triggers and their structure, will be covered in this section.

Due to time constraints, triggers were implemented in a relatively simple manner for demonstration purposes only. Nevertheless, the implementation is sophisticated enough to warrant its own section.

```
public class Trigger{
  public string Name;
  public string id;
  public DateTime LastTrigger;
  public Condition condition;
  public Action action;
  public List<Trigger> dependencies; //triggers can depend on
   other triggers for combinations
  public Sensor source; //can be null ONLY if Type = Time
  public Actor target; //can be null (for dep triggers)
}
```

Listing 13: Trigger class, Snippet # 1

The important components of a trigger instance are its Name, id (UUID/GUID), LastTrigger timestamp (set to unix epoch of Jan 1 1970 per default, updated to DateTime.Now on trigger), Condition (see below), Action (see below), a list of other trigger instances this one depends on, the sensor it gets the data from (null if it is a time-based trigger) and a target actor (null if it is a dependency trigger).

This setup might seem a bit complicated, but will be explained further in this section.

```
public Trigger(string name, Condition condition, Action action,
   List<Trigger> dependencies = null, Sensor source = null, Actor
   target = null)
{
  if (dependencies == null)
    dependencies = new List<Trigger>();
  Name = name;
  id = Guid.NewGuid().ToString();
  LastTrigger = DateTime.UnixEpoch;
  this.condition = condition;
  this.action = action;
  this.dependencies = dependencies;
  this.source = source;
  this.target = target;
}
```

Listing 14: Trigger constructor for completeness, Snippet #2

Next up is the condition class. For this, we have several types:

```
public enum ConditionType
{
  ValueExact, //when sensor value is EXACTLY x
  ThresholdRisingEdge, //when value goes above threshold
  ThresholdFallingEdge, //when value goes below threshold
  ValueChanged, //when sensor value is different than before TODO!
  Time //TODO: intervals and stuff
}
```

Listing 15: Condition types

The rest of the condition class looks like this:

```
public class Condition{
  public ConditionType Type;
  public Value CheckValue;
  private bool thresholdCheck = false;
}
```

Listing 16: Condition class

This part is where everything gets very complicated. Checking a condition construct this complex turned out to be very difficult to implement in practice, even though the concept appeared simple in planning. At the time of writing, the implementation is not fully complete yet, though major parts are working as planned.

Firstly, a function to check the state is needed, situated in the superclass of the given condition. The first check is simple, testing whether the values are of the same type.

```
public bool IsMet(Value sourceValue){
  if (sourceValue.Type != CheckValue.Type)
    return false;
}
```

Listing 17: Condition function, snippet #1

Next, a switch statement processes the condition differently depending on the value type.

```
public bool isMet(Value sourceValue){
  switch (sourceValue.Type){
    case Value.ValueType.Integer:
    case Value.ValueType.Double:
    case Value.ValueType.Boolean:
    case Value.ValueType.String:
    default:
      throw new ArgumentOutOfRangeException();
  }
}
```

Listing 18: Condition function, snippet #2

Then it is necessary to process the condition. First up is the integer case. Here we need to check if the threshold was met already beforehand. For this we use the variable thresholdCheck, which is set to true every time the value to check goes above the reference value, to prevent duplicate trigger executions. When it drops below, we set it to false so the threshold can trigger again. This effectively leaves an ignored tolerance of one unit between checks, which can be good or bad depending on the context it is used in, and remains unfixed as implementing a customizable tolerance threshold would over-complicate this setup even more, and is therefore out of scope for this project.

```
if (int.Parse(sourceValue.StringValue) > int.Parse(CheckValue.
   StringValue) && !thresholdCheck){
  thresholdCheck = true;
  return true;
}
else if (int.Parse(sourceValue.StringValue) < int.Parse(CheckValue
   .StringValue) && thresholdCheck){
  thresholdCheck = false;
}

return false;
```

Listing 19: Condition function, snippet #3. Integer comparison

Next up is the double comparison. The method used here is similar to the integer comparison, though here we have a much smaller tolerance threshold than with integer comparisons. Again, implementing a customizable tolerance is out of scope for the project. We also make use of the same thresholdCheck here to avoid duplicate executions of the trigger action.

```
if (double.Parse(sourceValue.StringValue) > double.Parse(
   CheckValue.StringValue) && !thresholdCheck){
  thresholdCheck = true;
  return true;
}

else if (double.Parse(sourceValue.StringValue) < double.Parse(
   CheckValue.StringValue) && thresholdCheck){
```

```
  thresholdCheck = false;
}

return false;
```

Listing 20: Condition function snippet #4. Double comparison

The boolean comparison is much simpler, as there are no tolerance possibilities. ThresholdCheck is used a bit differently here, albeit still for duplicate prevention. As a boolean response value can only have two states, it always has to have the opposite state of threshold check to be functional.

```
if (bool.Parse(sourceValue.StringValue) == bool.Parse(CheckValue.
   StringValue) && !thresholdCheck){
  thresholdCheck = true;
  return true;
}

else if (bool.Parse(sourceValue.StringValue) != bool.Parse(
   CheckValue.StringValue) && thresholdCheck){
  thresholdCheck = false;
}

return false;
```

Listing 21: Condition function snippet #5. Boolean comparison

Finally, the string comparison was not finished at the time of writing, and therefore is not included in the code snippets. This is meant for things like character displays and other devices capable of displaying such data.

This section just covered the RisingEdge condition type, so a quick overview of the other types follows. FallingEdge is easy, with simply inverted less/greater than operators. ValueExact is meant for string comparisons and therefore unimplemented. ValueChanged simply triggers every time the value changes to something else (mainly string comparisons again), and Time is meant for intervals, repeated time-based triggers, and also remains unfinished at the time of writing.

4.2 Hardware

4.2.1 Modules

In hardware, a module takes the form of an ESP8266 device, as planned. However, as the module-specific code is implemented in Arduino-like C++ code, it can very easily be ported to other platforms, like ESP32 and other Arduino-like microcontroller platforms. The hardware-level code was implemented using the PlatformIO toolchain, which allows for easily configuring IDEs (in this case, Visual Studio Code and JetBrains CLion) for use with microcontrollers like the ESP8266, and automates compiling, flashing and testing updated code. It also simplifies dependency management.

4.2.2 Sensors

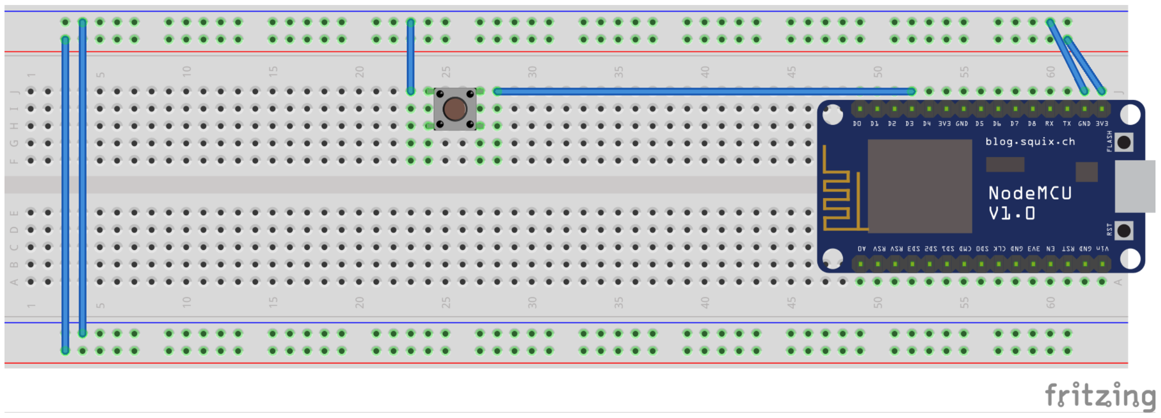

Figure 2: ESP8266_simple_sensor wiring diagram (Image source: made using the software Fritzing)

The code running on the ESP8266 module is very simple, as all the computationally intensive work is handled by the core. Therefore, this part should be a lot less complex. In this example we are looking at ESP8266_simple_sensor, which takes a digital input (from a button) and sends its state to the core.

```cpp
#include <Arduino.h>
#include <U8g2lib.h>

U8G2_SSD1306_128X32_UNIVISION_F_HW_I2C u8g2(U8G2_R0, /* reset=*/
    D0, /* clock=*/ D1, /* data=*/ D2);

#include <ESP8266WiFiMulti.h>
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
#include <WiFiClient.h>
#include <string>
```

Listing 22: Sensor C++ code, snippet #1

Here we include a few libraries. From top to bottom:

- Arduino.h, the Arduino standard library. Needed for Arduino-compatibility.
- U8g2lib.h, a monochrome display driver library. Used for debug output on the OLED display, present on some variants of ESP8266 modules available. The next line defines the OLED pinout.
- ESP8266WiFiMulti.h, ESP8266WiFi.h, ESP8266HTTPClient.h and WiFi-Client.h are required for utilizing the integrated WiFi module.
- string is the C++ string library, required for using strings.

Next up is the setup:

```cpp
ESP8266WiFiMulti WiFiMulti;

void setup(void) {
    Serial.begin(9600);
    delay(10);
    u8g2.begin();
    u8g2.setFont(u8g2_font_6x10_mf);
    u8g2.setFontMode(0);
    WiFi.mode(WIFI_STA);
    WiFiMulti.addAP("WIFI-SSID", "WIFI-PASS");
    pinMode(D3, INPUT_PULLUP);
    pinMode(D6, INPUT_PULLUP);
    pinMode(D7, INPUT_PULLUP);
    pinMode(D8, INPUT_PULLUP);
}
```

Listing 23: Sensor C++ code, initialization, snippet #2

Again, from top to bottom:

- A new instance of ESP8266WiFiMulti is initialized.
- Serial connection is initialized (for debugging)
- u8g2 is initialized (for OLED output)
- WiFi network connection is initialized
- D3, D6, D7 and D8 pins are defined as digital input, with the internal pullup resistor connected

Next up is the main loop:

```
bool state = false;
bool tmp = false;

void loop(void) {
    u8g2.firstPage();
    WiFiClient client;
    HTTPClient http;

    String payload = "";

    do {
        char macbuf[WiFi.macAddress().length()+1];
        WiFi.macAddress().toCharArray(macbuf, sizeof(macbuf));
        u8g2.drawStr(0, 7, macbuf);

        if (WiFiMulti.run() == WL_CONNECTED){
            char ipbuf[WiFi.localIP().toString().length()+1];
            WiFi.localIP().toString().toCharArray(ipbuf, sizeof(
ipbuf));
            u8g2.drawStr(0, 17, ipbuf);
            u8g2.drawStr(0, 27, "api rq...");

            if (!digitalRead(D3)){
                if (!tmp){
                    tmp = true;
                    state = !state;
                    String url = String("http://kaby.fritz.box:5000/api") + "?mac=" + WiFi.macAddress() + "&datatype=bool"
                    +  "&sensortype=ESP8266_simple_sensor" + "&value=" + state + "&displayunit=nA";
                    if(http.begin(client, url)) {
                        int httpCode = http.GET();
                        if(httpCode > 0) {
                            Serial.printf("HTTP Response Code: %d\n", httpCode);
                            if (httpCode == HTTP_CODE_OK || httpCode == HTTP_CODE_MOVED_PERMANENTLY) {
                                u8g2.drawStr(0, 27, "api http ok");
                            } else {
                                u8g2.drawStr(0, 27, "api http error");
                            }
                        }
                        http.end();
                    } else {
                        u8g2.drawStr(0, 27, "api connect fail");
                    }
                }
            }
            else if (tmp) {
                tmp = false;
            }
        }
        else{
            u8g2.drawStr(0, 17, "WiFi connecting...");
        }
        delay(50);

    } while ( u8g2.nextPage() );
}
```

Listing 24: Sensor C++ code, main loop, snippet #3

Complete source code: Laura Hausmann. *esh-sensor*. URL: `gitlab.com/lhausmann/esh-modules/blob/master/esh-sensor/src/main.cpp` (visited on 11/05/2019)

This part is a bit more complicated. At the beginning there is a bit more initialization going on. Then it checks whether the digital input on pin D3 has changed. If it has, it sends a request to the core (see chapter Software - Sensors) with its mac address, data type, sensor type, (new) value and unit. After that, there is a bit of error handling and debug output, and that's it. All further processing required is handled by the core.

4.2.3 Actors

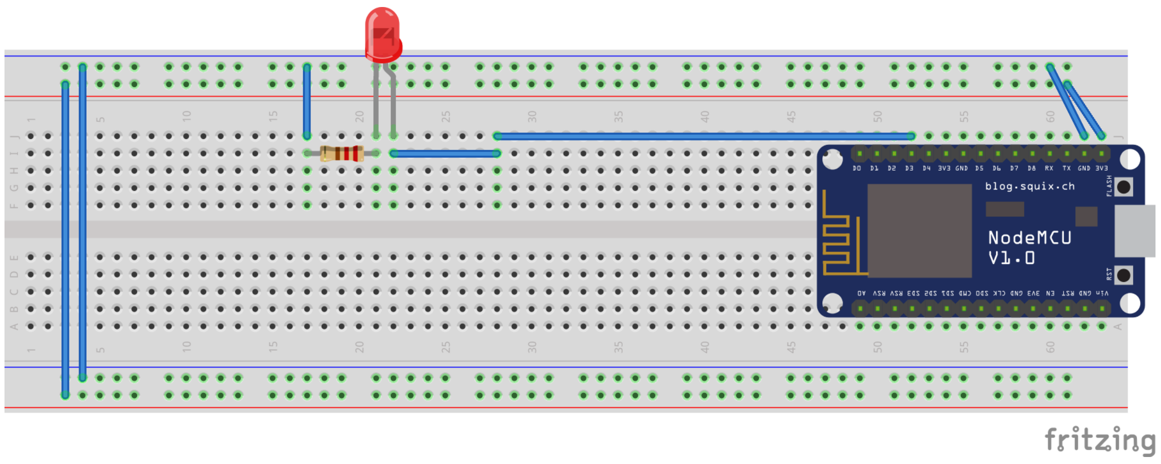

Figure 3: ESP8266_simple_actor wiring diagram (Image source: made using the software Fritzing)

The actor code, in this case of ESP8266_simple_actor, which drives a small LED, is pretty similar to the sensor code, but there are a few differences, which are outlined here.

The setup part is identical except for the pin configuration:

```
pinMode(D3, OUTPUT);
digitalWrite(D3, LOW);

String state = "Off";
```

Listing 25: Actor C++ code, initialization, snippet #1

Complete source code: Laura Hausmann. *esh-actor*. URL: `gitlab.com/lhausmann/esh-modules/blob/master/esh-actor/src/main.cpp` (visited on 11/05/2019)

Here the pin D3 is defined as an output, and set is set low (off). The state variable is now a string and set to ”Off”.

The only other difference is the api call:

```
String url = String("http://kaby.fritz.box:5000/api") + "?mac=" +
   WiFi.macAddress() + "&datatype=int"
        +  "&actortype=ESP8266_simple_actor" + "&state=" + state;

if(http.begin(client, url)) {
    int httpCode = http.GET();
    if(httpCode > 0) {
        String payload = http.getString();
        Serial.printf("HTTP Response Code: %d\n", httpCode);
        if (payload == "1"){
            Serial.println("HIGH:" + payload);
            state = "On";
            digitalWrite(D3, HIGH);
        }
        else{
            Serial.println("LOW:" + payload);
            state = "Off";
            digitalWrite(D3, LOW);
        }
        if (httpCode == HTTP_CODE_OK || httpCode ==
   HTTP_CODE_MOVED_PERMANENTLY) {
            u8g2.drawStr(0, 27, "api http ok");
        } else {
            u8g2.drawStr(0, 27, "api http error");
        }
    }
    http.end();
} else {
    u8g2.drawStr(0, 27, "api connect fail");
}
```

Listing 26: Actor C++ code, api call, snippet #2

Contrary to what the sensor does, a request is sent to the core asking for data. In this case, a boolean value which determines whether or not the LED should be lit. In addition to that, the mac address, data type, actor type and current state are sent along with the request. The response is then processed, and the LED state is set accordingly.

4.3 Tests

4.3.1 Unit tests

Every (relatively) big programming project should have unit tests. Unit tests are automated methods of testing source code for errors, by defining what small parts of the code are supposed to do, and checking if it still behaves as expected after every small change.[8] Ideally this would be implemented in esh and esh-modules, however due to time constraints this was not possible.

4.3.2 Real-world tests

The core and both the sample modules ESP8266_simple_sensor and ESP8266_simple_actor were tested on actual hardware as a Proof-of-Concept demonstration.

[8]Adam Kolawa; Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management.* Wiley-IEEE Computer Society Press, 2007, Page 75.

5 Comparisons

5.1 Commercial/proprietary systems

When compared to other commercial smart home systems, esh excels in some aspects (extensibility, security and configurability, among others), while being inferior in others (ease of use, commercial viability). It is important to note that the aspects where it is inferior were not intended to be met from the start, while commercial viability was explicitly ignored. This leaves us with the following conclusion: While a commercial solution still is the better choice for an ordinary person, for privacy-conscious people or people generally interested in tinkering with technology, willing to learn and wanting to expand on the ecosystem, esh is a very viable option.

5.2 Similar OSS/OSH projects and why MQTT is bad

There are many open source software and open source hardware projects for smart home. This begs the question - why another one? There are a few reasons for that:

- I wanted a smart home system at home, and I did not like commercial systems due to reasons stated in the very beginning of this paper.
- Existing open source solutions mainly rely on specific hardware and employ more complex protocols like MQTT (MQ Telemetry Transport). MQTT is great, but ”publish/subscribe messaging transport” as the developers describe it is complicated to implement on very low-power systems (like the ESP8266), and employs concepts that carry limitations that I did not like to have in my own system.

- I think extensible, open source software is useful and future-proof, as adding components is much easier than having to build a bridge to a closed software solution.

There are certainly use cases for MQTT and open source systems based on it or similar protocols, but for my specific case I needed something else - so I created esh.

5.3 Home Assistant

Home Assistant (hass.io) is a relatively new open source hub for smart home and home automation. It employs similar concepts as esh, but, in my opinion, adds needless UI complexity to the system, and an exhausting combination of configuration via config files and configuration via UI, that clashes very often and therefore negatively affects the user experience. For this reason it was not considered a feasible option in the planning stage.

List of Figures

Listings

References

Batrinu, Caitlin. *ESP8266 Home Automation Projects.* Packt Publishing, 2017.

– *Smart Home Automation with Linux and Raspberry Pi.* apress, 2013.

Hausmann, Laura. *esh-actor.* URL: `gitlab.com/lhausmann/esh-modules/blob/master/esh-actor/src/main.cpp` (visited on 11/05/2019).

– *esh-core.* URL: `gitlab.com/lhausmann/esh/tree/master/` (visited on 11/05/2019).

– *esh-sensor.* URL: `gitlab.com/lhausmann/esh-modules/blob/master/esh-sensor/src/main.cpp` (visited on 11/05/2019).

Huizinga, Adam Kolawa; Dorota. *Automated Defect Prevention: Best Practices in Software Management.* Wiley-IEEE Computer Society Press, 2007.

Rouse, Margaret. *What is Smart Home or building?* URL: `internetofthingsagenda.techtarget.com/definition/smart-home-or-building` (visited on 07/12/2019).

Systems, Espresif. URL: `www.espressif.com/en/products/hardware/esp8266ex/overview` (visited on 07/13/2019).