UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

# Conception and realisation of a resilient Smart Home solution

*Konzeption und Realisierung einer resilienten Smart-Home-Lösung*

**Bachelorarbeit**

im Rahmen des Studiengangs
**Informatik**
der Universität zu Lübeck

vorgelegt von
**Hannes Preiß**

ausgegeben und betreut von
**Prof. Martin Leucker**

Lübeck, den 15. Januar 2020

## Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne die Benutzung anderer als der angegebenen Hilfsmittel selbständig verfasst habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet.

(Hannes Preiß)
Lübeck, den 15. Januar 2020

**Abstract** There exist a number of modern smart home solutions that provide the user with a convenient method of remote device control and home automation. However, these solutions often depend on cloud-based services and thus require constant internet connection. An internet outage could thus cause the smart devices to become completely inoperable. They also frequently raise privacy and security concerns. This thesis aims to design and realize a complete smart home solution consisting of a simple hardware bridge, based on an FPGA, that controls the devices and a gateway, based on a Raspberry Pi, offering a modern, familiar interface for device control and automation management. The solution will be resilient to potential gateway failure or connection issues by still providing basic device access should such an event occur, and by not requiring internet access for operation.

**Kurzfassung** Es existieren eine Reihe an modernen Smart-Home-Lösungen, die dem Benutzer bequeme Möglichkeiten von entfernter Gerätesteuerung und Heimautomation bieten. Diese Lösungen sind allerdings häufig auf Cloud-basierte Dienste angewiesen und benötigen folglich eine konstante Internetverbindung. Ein Ausfall dieser könnte somit die Smart-Geräte komplett inoperabel machen. Darüber hinaus verursachen sie häufig Bedenken hinsichtlich Privatsphäre und Sicherheit. Diese Arbeit strebt das Konzept und die Realisierung einer Komplett-Smart-Home-Lösung an, bestehend aus einer einfachen Hardware-Bridge, basierend auf einem FPGA, welche die Geräte steuert als auch aus einem Gateway, basierend auf einem Raspberry Pi, welches eine moderne, vertraute Schnittsttelle für Gerätekonttrolle und Automationsmanagement bietet. Die Lösung wird eventuellen Gatewayausfällen sowie Konnektivitätsproblemen gegenüber ausfallsicher, indem sie in solchen Fällen einfachen Gerätezugang ermöglicht und nicht auf einen Internetzugang angewiesen ist.

# Contents

# 1 Introduction

Undoubtedly, smart home technology has become a popular topic over the recent years. With the general growth of the IoT – the *Internet of Things* – market, smart devices can now be found in more and more homes. In 2015, around 19 percent of all US households were equipped with smart technology, and come 2025 this rate is expected to grow to about 53 percent. [Koz16]

In fact, the ever-increasing desire for smart home systems has led to the surfacing of so many self-contained ecosystems that several companies including Google, Apple and Amazon, who are actually competing with each other on the smart home market, have recently formed a new working group that aims to provide users a unified experience to smart home technology, creating a layer of interoperability for all those devices which are currently still incompatible with each other. [Con19]

Current existing smart home solutions mainly focus on presenting the user with a convenient, feature-rich experience by expanding the possibilities of the local home network with ubiquitous cloud-based services like voice recognition and more. This approach, however, aside from the implications for security and privacy, often also sacrifices another critical factor due to the constant need for an available internet connection: resilience.

While these factors might be acceptable for private use of such a solution, it is not quite suitable for professional use, e.g. inside an office complex.

## 1.1 Goals of this Thesis

The primary goal of this thesis is to create a complete, modern smart home proof-of-concept solution, including all of the necessary parts that comprise it, by using readily available, low-cost hardware.

As the title of this work states, we're especially looking to design a *resilient* solution. Let us briefly consider the motivation behind this aspiration. Imagine a modern smart home system that is controlled by a single, very modern, very complicated computer-based device. Now, imagine this computer locking up due to software problems, being compromised due to security issues or even because it is currently automatically installing the newest vendor update, which happens to be a process not controllable or interruptible by the user. Since all of the system's wireless smart devices directly depend on this controller, the whole smart home system suddenly becomes completely unresponsive. This would pose a problem anywhere between inconvenient to potentially even dangerous, depending on the type of smart devices involved.

This scenario – albeit, admittedly, somewhat exaggerated – served as the primary motivation behind this project. Thus, we look to design a smart home solution that mitigates this issue. We accomplish this by splitting the system's controller into two parts:

- a *bridge* running on very simple hardware logic – most importantly not being computer-based – to reduce the risk of failure, responsible for controlling the system's devices, and
- a *gateway* based on a computer, allowing more sophisticated and modern methods of control and automation to be used.

Together, those two devices strive to present a modern smart home experience with all of its benefits, including convenient automation capabilities and elegant user interfaces. The gateway is still relatively prone to failure, being a modern computer-based device with an operating system of some kind – but should it come to this, the bridge can still work on its own, providing at least basic, "non-smart" control over the system's devices.

A compromise has to be made between allowing the bridge part of this system to remain flexible enough to serve different device configurations, while still being simple enough to not be prone to failure. We will settle on a solution that requires manual reprogramming of the bridge's hardware logic, accommodating for a certain configuration of devices connected to it. However, the intended audience for this solution consists mainly of electricians and end-users, who can not be expected to be versed in the relatively exotic topic of hardware programming. Therefore, the project will be supplemented by a tool that allows for easy reconfiguration of the bridge hardware.

## 1.2 Limitations to the Thesis Scope

The main focus of this work is the software and hardware design of the smart home system components, especially of the controller hardware.

In particular, it will not deal with any subjects from the fields of power engineering; including topics regarding serving particular voltages to appliances or providing appropriate cabling inside a building.

On a related note, it should be stated that the finished product will merely be a proof-of-concept and, on its own, will not yet be ready for production. It shall serve as a base for potential future work, which integrates it into a system ready for home or office use.

## 1.3 Outline

**Chapter 2 (pg. 5)** will first provide a brief analysis of smart home systems and their components to provide some context for the work done later in this thesis.

The main goal of this thesis, the concept and design of a resilient smart home solution, will be outlined in detail within **Chapter 3 (pg. 15)**. Details regarding its concrete implementation will be described in **Chapter 4 (pg. 61)**.

During evaluation in **Chapter 5 (pg. 79)**, we will see if our completed solution achieves the goals we set for it, and **Chapter 6 (pg. 87)** will provide a summary of the work that has been accomplished.

# 2 Analysis of Smart Home Architecture

To understand in what ways a smart household differs from a non-smart one, we first need to understand what exactly defines it. In fact, it is not trivial to pinpoint the term *smart home* to a singular definition, despite – or perhaps even because of – its increasing popularity. In [ARA12], Alam et al. gather a few attempts to define the term that have appeared over the years, and eventually summarize a smart home as "an application of ubiquitous computing that is able to provide user context-aware automated or assistive services in the form of ambient intelligence, remote home control or home automation."

Following this definition, in this chapter we will explore the question of what is needed to make a home smart by analyzing the different **devices** involved in a usual smart home setup, including their roles and responsibilities. Since device interoperability plays a big part in smart home networks, several different means of **communication and protocols** will be outlined next. Then, we shall examine a few different examples of **existing smart home solutions**, and how they compare to each other and to our eventual goal of a resilient smart home solution.

## 2.1 Device Hierarchy and Roles

A smart home is comprised of several different groups of devices. **Appliances** define the subject of home automation, as they are the devices one wishes to control, while **Sensors** instead supply the system with input data. They are managed by a **gateway**, which controls the appliances through automation rules based on the sensor input data as well as user interaction through a **user interface**. Finally, some devices cannot directly communicate with the gateway and instead need a mediating device called a **bridge** in between.

Figure 2.1 on the next page gives a simple overview of the architecture.

### 2.1.1 Appliances

In broad terms, an *appliance* is some sort of electrical device with a certain purpose for use in a household. Within the context of a smart home, however, only the

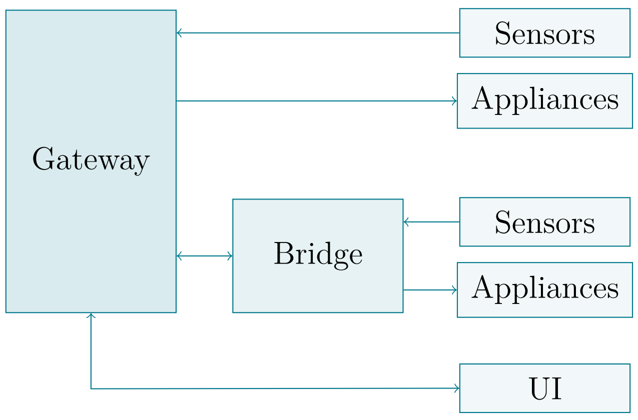


**Figure 2.1:** Device hierarchy and communication flow in a standard Smart Home architecture. Devices are either connected directly to the gateway, or through a mediating bridge.

subset of devices possessing the ability to be remotely and automatically controlled is of particular interest.

Examples for common appliances in smart homes thus include lights, radiators, air conditioners, window covers and shutters, among others.

Devices capable of connecting to and communicating with a network on their own are also called *smart devices*. Many modern appliances, especially in the entertainment area (televisions, gaming consoles, media players, etc.) but also an increasing number of other household appliances (refrigerators, washing machines, robotic vacuum cleaners, etc.) provide network connectivity features, making them a viable and easy target for home automation.

### 2.1.2 Sensors

*Sensors* are, generally speaking, devices with the purpose of collecting data from the surrounding environment, providing it to other devices and notifying them when significant changes or events in the environment occur.

A sensor may be as trivial as a simple switch or button that detects when it has been flipped or pressed. These types of sensors are highly commonplace in most, if not all modern households in the form of light switches and doorbells, among many others. Examples for slightly more sophisticated types of sensors with common usage in homes include hygrometers, thermometers and infrared motion detectors.

Inside a smart home, sensors play an essential part of home automation. Aside from their typical role also found in non-smart systems (such as a standard light switch), environmental changes and events are often used to trigger certain automation rules.

Additionally, the advent of smart homes also introduced more abstract kinds of sensors that may not even require any specialized sensor hardware. A common example is a system being able to recognize when a member of the household arrives or leaves home, accomplished through surveilling GPS data from the members' smartphones or other personal devices with similar functionality. In these cases, the smart home system takes advantage of already existing IoT hardware as a source of environmental data.

### 2.1.3 Bridges

Certain devices intended for smart home usage may not be meant to communicate directly with the gateway. Rather, they are to be connected to and controlled by another device called a *bridge*, which in turn then serves as the interface exposing and representing its devices towards the gateway.

For example, certain smart devices might communicate using a protocol that is not supported by the smart home system. The bridge then serves as a means of literally bridging the two communication systems together, so that they can be made to work with one another.

Designing such a bridge will be a major component of the smart home solution that will be outlined starting in Chapter 3 (pg. 15).

### 2.1.4 Gateways

A *gateway*, also sometimes called a *hub*, defines the centerpiece of a smart home installation. Its task is to manage and control all devices of the smart home system as well as to keep track of the current states of all appliances and sensors. To that end, it maintains a connection to all devices, either directly or indirectly via bridges.

Another responsibility of the gateway is to provide a means of home automation – automatically controlling appliances via a set of user-definable rules, based on triggers and conditions that take data from certain sensors as input. For example, a user could create a rule that automatically turns off all lights at a certain time of day, or another that regulates the air conditioning based on a room temperature sensor.

There are a number of freely available gateway software solutions, including *openHAB*[1] and *Home Assistant*[2], which will be talked about in more detail throughout the following chapters.

Commercially sold gateway implementations often come in the form of a small device the user installs in their home that either runs the gateway software itself or only acts as a thin client, which instead just serves as an interface to a cloud service running the software. A few examples of these will be mentioned in Section 2.3 (pg. 12).

Open source gateway software is freely available for the user to download and deploy on a server of their choice; though open source cloud solutions also exist [ope19b].

### 2.1.5 User Interfaces

The gateway serves one or several user interfaces as part of its frontend. Inside, it reports the state of all appliances and sensors, it allows the user to directly control, manage or add new devices and it provides means to adjust the whole system's behavior, including the creation of automation rules.

Commonly, the interface is offered through a website-based or a dedicated smartphone application. Certain software solutions like openHAB even provide multiple interfaces specifically created for different types of screen media, such as wall-mounted touchscreen displays [ope19c].

Commercially available gateway devices usually provide a voice command and text-to-speech interface, while also serving as a smart media playback appliance. For this reason, they are also called *smart speakers*. Section 2.3 (pg. 12) will introduce some popular devices of this kind.

## 2.2 Communication and Protocols

Allowing two or more devices to communicate with each other always has to involve a certain medium for the messages to be transmitted. As the interaction between gateway and bridge will be a central part of our smart home solution, it seems appropriate to discuss some standards of communication, both with potential use in our project as well as what is commonly found among existing smart home systems.

[1]`http://www.openhab.org`
[2]`http://www.home-assistant.io`

We can generally distinguish between two different methods of linking devices together: **wired** and **wireless**.

### 2.2.1 Wired

In the context of creating a smart home solution, the wired bus standards **SPI** and **I²C** are the most significant, as both our future gateway and bridge offer native support for each, and we are eventually going to have to choose one for our solution.

#### SPI

*SPI* – short for *Serial Peripheral Interface* – is a communication bus with common use inside embedded systems. It connects two devices together through four wires: a clock signal, a slave select signal as well as one wire for each data direction.

SPI follows the *master/slave model*, where one device, serving as the bus master, controls the flow of communication as well as the clock speed. The other device(s) cannot initiate transmissions; they can merely reply to a message coming from the master.

The purpose of the slave select line is to allow multiple slaves to connect to the same bus, reducing the overall amount of wires needed to connect several devices together. In such a multi-slave setup, when the master wishes to transmit to a certain slave, it enables the slave select signal for that particular device only, allowing it to reply via the MISO *(master in, slave out)* line. All devices not addressed in this manner must not respond.

Implementing SPI in both hard- and software is rather simple; as the protocol does not contain any error-checking capabilities or similar nuances. This also allows for relatively high transmission rates. However, this also marks a significant disadvantage as it becomes difficult to determine faulty connections. Due to the absence of an acknowledgment signal from a slave, the master could potentially not notice connection failures. Reliability measures could be implemented on top of the SPI standard, but this obviously requires more effort. [Wik19b]

#### I²C

*I²C*, or *Inter-Integrated Circuit*, is a bus standard very similar to SPI in that it is also offers a serial communication interface adhering to the master/slave architecture.

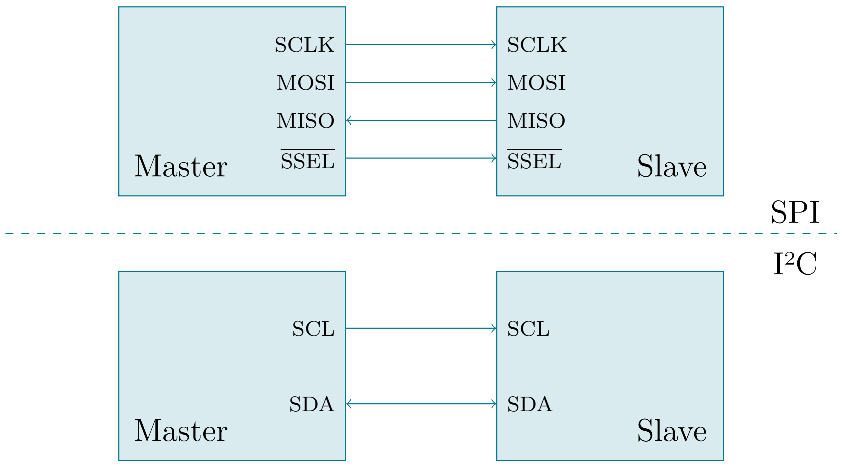


**Figure 2.2:** Comparison of SPI and I²C bus wiring.

Instead of SPI, it only uses two wires, clock and data, for transmissions between devices.

Instead of selecting slaves via a separate wire, the I²C protocol instead uses a 7-bit address space for all connected slave devices. At the beginning of a transmission, the I²C master transmits the address of the particular slave onto the data line. Afterwards, a message is sent, byte per byte, with the slave sending an acknowledgment bit after each successful reception.

I²C transmissions are either read or write only, signified by a single bit the master sets immediately after the slave address. However, again, only the master can initiate a conversation. To request data from a slave, the master first sends a command or similar, then begins a read transfer during which the targeted slave takes control of the data line to write an appropriate response.

The digital addressing mode makes I²C inherently more scalable than SPI, as the number of wires remain constant regardless of the number of slaves connected. The presence of the acknowledgment bit makes this I²C transmissions more reliable, but at the same time the standard is slightly harder to implement than SPI. [NXP14]

### 2.2.2 Wireless

Although not quite as relevant for our solution, which will use a wired network medium, wireless protocols are very prevalent in smart home systems because they enable convenient setup of devices throughout large areas. We will compare a few prominent standards in regards to some properties relevant to smart home usage, including communication range, power requirements and maximum network size.

### Wi-Fi

*Wi-Fi* is the canonical standard for wireless local area networks among computers. One of its primary applications is to provide internet access to devices inside a local area. [Wik20c]

It is generally not widely used in the smart home domain, as it focuses more on range and high transmit rates. This causes it to require too much power to be feasible for smaller smart devices. [LSS07] However, it is nonetheless significant as smart gateways or bridges often make use of Wi-Fi, for example to be controlled by a user's smart phone.

### Bluetooth

*Bluetooth* is a wireless standard intended for replacing short-ranged cabled connections. Similar to the wired protocols described above, it uses the master-slave architecture, and a network consists of one master communicating with up to seven slaves. [Blu19]

Its short range of up to 10 m and limited network size make Bluetooth not a particularly useful standard for use in a smart home system of a larger scale. However, it has certain applications especially for private use, as Bluetooth support is relatively widespread across common devices that can be used for remote control, such as smart phones.

### ZigBee

Compared to Bluetooth, *ZigBee* was designed primarily for use in the IoT field. [Zig20] It supports communication ranges of up to 100 m in ideal conditions and several different network topologies, including ad-hoc, mesh and star networks. This allows it to theoretically support tens of thousands of devices on a single network. [Bak05]

While its power consumption during data transmission approximately equals that of Bluetooth [LSS07], the ZigBee hardware supports a standby mode that causes it to draw next to no power during idle phases, whereas the Bluetooth standard does not have such a feature. As such, ZigBee powered devices can potentially have a battery life of several years prior to recharging. [Bak05]

#### Z-Wave

*Z-Wave* is a competing standard to ZigBee, as it was also specifically developed with IoT in mind. It supports ranges of up to 30 m, and a single Z-Wave network can consist of up to 232 devices. Its power requirements compare to those of ZigBee.

One of its primary features is that it operates on a lower-frequency band than Bluetooth, Wi-Fi and ZigBee devices, which all transmit on the 2.4 GHz band, thus avoiding interference on crowded radio networks. [Wik20d]

## 2.3 Popular Services and Solutions

The rapid development of smart home technology and popularity has caused numerous solutions to surface over the past few years, each offering different advantages and disadvantages over another. In particular, after this section it should become clear how the solution we are going to design differs from the concepts behind these products, justifying the motivation behind the project.

### 2.3.1 Amazon Echo

*Amazon Echo* is a series of smart speakers that primarily serve as an interface for Amazon's *Alexa*, a personal assistant service, and can optionally be used as a smart home gateway, connecting to smart devices via Bluetooth or Wi-Fi. The Echo Plus also supports the ZigBee protocol. [Ama20]

Through the use of an Echo speaker, users may, among other activities, play music, take phone calls, order products from the Amazon store, query weather and news data, or control connected smart devices. This is done either through voice commands or a smartphone app. In any case, because both the voice recognition and command handling as well as most logic driving the device's behavior reside on Amazon cloud servers, it requires internet connectivity at all times even though the smart devices are connected to the Echo inside the local network.

This puts certain limits on the service's usefulness as a resilient smart home system, as a failing internet connection causes all appliances to become unresponsive. Also, as with all cloud-based services, but for Alexa in particular, the speaker, which is constantly recording sound from its environment, raises several privacy and security concerns. [JO18] [HSWW17]

### 2.3.2 Google Home

In response to Amazon's Echo product line, Google developed their own brand of smart speakers, *Google Home*. The devices feature roughly the same functionality as their Amazon counterparts, listed above. It also features the Google-brand voice-controlled personal assistant service, *Google Assistant*. [Wik20a] Like the Echo, it uses Bluetooth and Wi-Fi for connection to smart devices.

Google Home speakers also require a constant internet connection for all tasks, due to the same reasons as the Echo speakers. However, at the time of writing, Google is currently working on a new version of its Google Assistant service that is able to completely run on local devices, and so perhaps this restriction might be lifted in the near future. [Goo20]

### 2.3.3 Apple HomeKit

Apple's *HomeKit* framework is another smart home solution designed especially for Apple devices. It is available through an app found inside all devices running Apple's proprietary operating system, iOS, starting version 8. [Wik20b]

HomeKit is notable because unlike its competitors, it does not necessarily require a cloud platform, and therefore access to the internet. The Apple device running the Home app itself serves as the system's gateway, directly connecting to and controlling appliances through Bluetooth or Wi-Fi. However, certain features like support for automation rules and voice commands through Siri, Apple's personal assistant service, do require internet connectivity.

To include support for HomeKit into a smart device, it needs to be officially licensed by Apple [App20a]. As a result, the HomeKit ecosystem contains fewer supported devices overall than the other solutions [Wik20b]. However, it is possible to integrate non-HomeKit-enabled devices through certain bridges or even freely available solutions, like *homebridge*[3].

To allow remote management of the smart home, certain Apple products including iPods and Apple TVs can be used as a remote gateway server. This is a convenient solution for users who already possess such a device, not having to buy a specialized gateway product. [App20b]

[3] `https://github.com/nfarina/homebridge`

### 2.3.4 Philips Hue

Specifically tailored towards providing a smart lighting solution, *Philips Hue* is a series of smart lights coming in several forms, including light bulbs, strip lights and lamps. [Phi20c]

It is mentioned along the other solutions here because the Hue devices can not only be connected to a range of other smart home solutions due to using the ZigBee protocol [Phi20a], but also comprise their own, independent smart home solution with the addition of a Philips-brand gateway.

More recent device versions also ship with additional Bluetooth support. Those devices are then able to be controlled via a dedicated gateway smartphone app, removing the need for a hardware bridge entirely. Due to the limitations of the Bluetooth protocol, this only supports up to ten devices per system, however. [Phi20b]

Like HomeKit, a Hue-based smart system does not require internet connectivity because it only uses the local network. However, the solution will be limited to smart lights only, which might limits its applications.

# 3 Designing a Smart Home Solution

After describing the components that comprise a complete smart home system, it is now time to start working on constructing our own. First, the **goals** we are attempting to achieve in building this smart home solution will be outlined, after which we will take a look at the **overall architecture**, including a description of each component's roles and responsibilities in the bigger picture.

This chapter will then go into in-depth detail about designing the individual parts of the smart home solution, starting with the **bridge hardware logic**, which will be created completely from scratch. In the next part about the **communication protocol**, we will design a means for the bridge and the gateway to communicate with each other. Conception of said **gateway** follows shortly after, where we will pick both a suitable operating system and gateway software for use in our project. To enable easier interfacing with the bridge, we will also design an API and finally an extension for the gateway software that makes use of it.

After concluding design work on the solution parts themselves, in the final section of this chapter a **device management UI** will be created to help making the solution more accessible for users not versed in hardware design.

## 3.1 Defining Solution Goals

We're starting this chapter by defining a clear set of goals for our project. This will provide us with sensible boundaries and expectations while working on the design process.

Chapter 1 (pg. 1) already mentioned the motivation behind this project. From within, we can already extrapolate a number of significant key points. In part inspired by similar characteristics by which software quality is usually measured, they are as follows:

**Resilience.** One of the solution's key features should be to prevent the worst-case scenario – all appliances becoming unresponsive due to controller failure – from occurring as much as possible. This is often caused by said controller appearing as a single point of failure, and sufficient measures have to be taken into consideration to avoid such a design.

**Maintainability.** To maximize the system's usability, being able to reconfigure it without much hassle should be a priority.

**Accessibility.** As an addendum to the last point, the steps needed for system maintenance should be able to be performed even without too much in-depth knowledge of the system's inner workings, and certainly should not require proficiency of a specific computer language, if at all possible.

**Scalability.** As the solution is also intended for use inside sizable buildings, it should support an accordingly appropriate number of connected devices.

**Conformance.** To avoid creating merely another proprietary, self-contained ecosystem, the solution should allow integration or at least interoperability with other smart home products or even solutions. On the same note, it should strive to present users with a modern presentation that feels familiar to similar solutions.

**Cost-efficiency.** Though a slightly minor point, this work will show that the aspired solution can be achieved using efficient, yet low-cost hardware, thus making it a viable option not only for professional use, but also for hobbyists and smart home enthusiasts. In fact, the simplicity of the provided hardware might even be considered a benefit to the project's goals, as will be explained when discussing the hardware's suitability for the task.

**Improvement over prototype.** This project can be viewed as being an evolution of a prototype – to be introduced shortly – that originally had the same goals in mind, yet showed several flaws. Improving upon this first version should be another of this solution's priorities.

To achieve these goals, let us first consider the **hardware** that has been provided for this task and contemplate their features and resulting suitability for the solution. Afterwards, it seems only appropriate to talk about the aforementioned **prototype** that this project will be based upon. We will examine its design and capabilities as well as its shortcomings and use the gathered information as a starting point for creating an improved product.

### 3.1.1 Provided Tools and Hardware

The design of this project revolves mostly around the two hardware components that were provided for this task. To make efficient use of them, it is important to understand their features and capabilities, which will therefore be outlined in this section. Afterwards, even though the hardware choice was fixed to those presented shortly, some other alternative hardware choices that have been examined with potential use in this project will be discussed as well.

### MachXO2-7000 FPGA

The *MachXO2* is an FPGA – a *field-programmable gate array* – developed and manufactured by Lattice Semiconductor[1].

FPGAs are integrated circuits with the capability to be configured and reconfigured multiple times by designers and end-users [Wik19a]. This, together with the fact that they are usually outfitted with many I/O ports, makes them especially versatile and suited not only for quick hardware prototyping, but also for flexible reconfiguration of already deployed hardware.

Programming an FPGA is, for the most part, accomplished by describing the hardware logic in an abstract manner using a *hardware description language* (HDL) such as Verilog, the process of which will be detailed more in Section 4.1 (pg. 61).

The MachXO2-7000 in particular contains up to 334 usable I/O ports and is shipped with several embedded function blocks for common features found in integrated hardware, like I$^2$C and SPI as well as a configurable, integrated oscillator [Lat19].

This makes the MachXO2 an excellent target for use as a smart home bridge, as it allows for many different electronics to be wired to and controlled by it, while also already providing several common interfaces for communication purposes. Finally, the ability to be reconfigured allows its behavior to be well suited to the devices you connect to it.

### Raspberry Pi 3

The Raspberry Pi is a fully-featured single-board computer, able to run a variety of operating systems including a number of Linux distributions, Windows 10 IoT Core [Ras19c] and Android [And19]. Due to its low cost and efficient energy usage, it is a popular choice for many do-it-yourself projects.

Used in this design is the device's third revision, the Raspberry Pi 3 Model B. It possesses an 1.2 GHz Broadcom ARM CPU as well as 1 gigabyte of RAM, along with a number of USB and general purpose I/O ports, Ethernet, Bluetooth and Wi-Fi support [Ras19b]. The Broadcom SoC includes hardware support for SPI and I$^2$C, and some of the board's I/O pins can be designated for their use [Bro12].

These qualities easily justify its use in this project as a smart home gateway. Being able to run Linux, it can host a number of server applications including some smart home gateway software for controlling and managing the bridge, which it can connect to using one of its supported bus interfaces. Of course, with all the benefits of

[1]`http://www.latticesemi.com`

running a modern operating system also come its drawbacks, as it is more prone to security and instability issues than a dedicated environment specifically tailored to the task. However, with sufficient work put towards ensuring security and stability, these issues are outweighed by the hardware's capabilities.

#### Alternatives

- Instead of splitting the responsibilities between the FPGA and the Raspberry Pi, it could also be considered to **only use a Raspberry Pi** and to have it manage any appliances by itself. In fact, openHAB for example even includes native support for controlling the device's I/O pins as smart devices [ope19a]. However, even if none of the pins are used for their designated functionality like SPI, the Raspberry Pi 3 merely has 26 I/O pins available for use [Ras19a], which was seen as too limiting for this solution. Moreover, if the whole system was controlled by only a single controller, that device would serve as a single point of failure, decreasing the system's overall resilience and thus going against the project's original goal.
- Usage of a **microcontroller** like an *Arduino*[2] as a replacement for either device has also been considered. The Arduino Mega, for example, contains 54 general-purpose I/O ports, 128 KB of flash program memory and supports SPI and I$^2$C as well [Ard19a]. Yet, it has limited use as a gateway since it does not have network support and is incapable of running anything but the simplest kinds of server software. Using certain extension hardware like the *Arduino Ethernet Shield 2* [Ard19b], it is possible to add this functionality to the microcontroller to some extent, but this would also occupy a fair number of I/O pins while also still not quite comparing to what the Raspberry Pi could already do on its own.

  Replacing the MachXO2 as a smart bridge would be slightly more suitable for a microcontroller like the Arduino, but since it is basically a CPU running a program, there is always a risk of it getting stuck in an endless loop due to programming errors or other problems. As integrated circuits or FPGAs only run on (relatively) simple digital logic, their usage gives a better outlook on long-term reliability.

### 3.1.2 Starting Point: SPI Prototype

Previously, a prototype project (hereafter referred to as the *SPI Prototype*) had been created by a small group of students at the University of Lübeck, including myself,

[2]`http://www.arduino.cc`

using the same hardware that will also be used in the final solution.

The SPI Prototype was created with the same goals in mind; creating a reliable smart home proof-of-concept solution using low-cost hardware. Since it serves as an important starting point of the work done in this thesis, the following few sections will briefly describe how the prototype was designed and outline some of its problems and limitations that the final solution strives to improve upon.

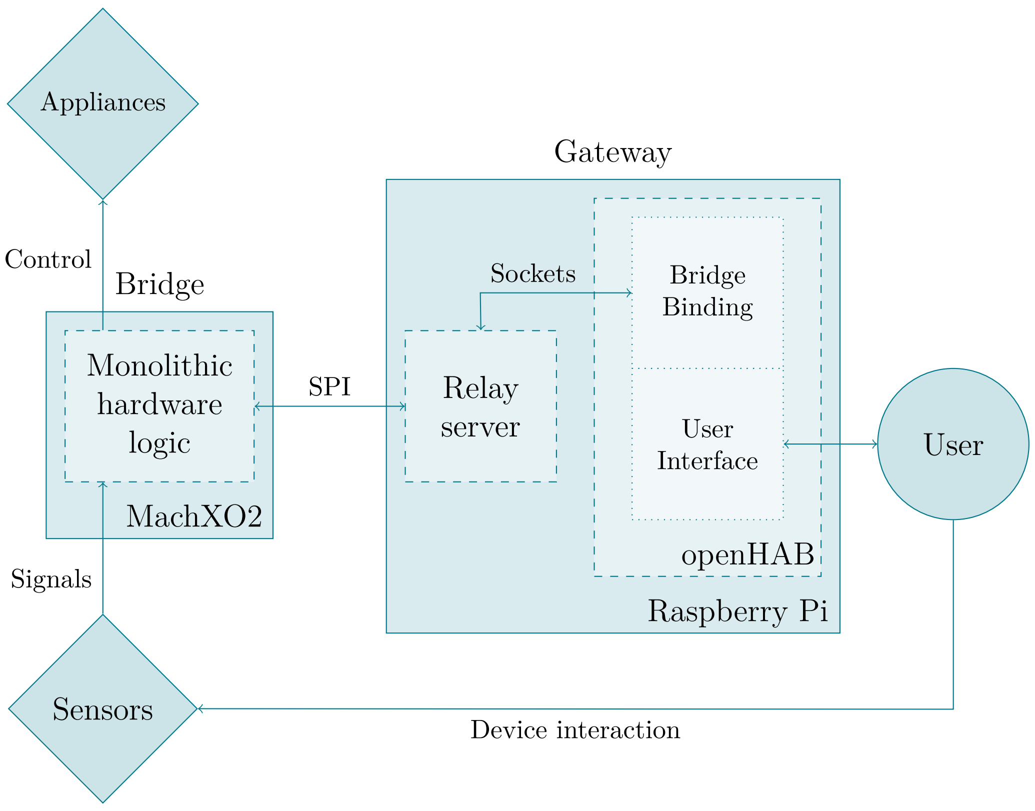


**Figure 3.1:** Overview of the architecture used in the SPI prototype, including communication flow between each component. Appliances and sensors (also referred to as *output* and *input devices*) were directly connected to and controlled by the MachXO2 FPGA, which acted as the *smart bridge* in this setup. The Raspberry Pi served as the *gateway*, running management software that provided several user interfaces for controlling and automating the appliances.

## Bridge

The MachXO2 was given some basic logic for handling incoming commands sent by the gateway and for managing the state of a few appliances – depicted through a few LEDs – connected to it. An exemplary sensor in the form of a contact switch was also installed.

Internally, both input and output devices were (separately) assigned single-byte addresses or IDs, which were hard-coded inside the FPGA hardware logic depending on which I/O pin the devices were connected to. This effectively enforced a hard limit of 254 ($2^8$ minus two non-assignable bytes) devices for both appliances and sensors.

The bridge supported two output modes; one for simply turning an appliance on or off, and another for analog control via pulse-width modulation. A device's state was also represented as a single byte, where `0x00` meant *off*, and anything between `0x01` and `0xFD` was interpreted as *on* for binary devices and set the pulse-width modulation frequency accordingly for analog control.

### Gateway and Relay Server

The gateway consisted of the Raspberry Pi running a distribution of Linux which in turn hosted a smart home gateway software of some kind. Arch Linux[3] was chosen as the operating system because, at its base, it offers a very lightweight and flexible environment especially suitable for headless servers that can easily be extended by many available software packages. It offers a rolling release system, meaning updates to individual packages are released soon after they are available. This helps to prevent security issues due to outdated packages.

For the gateway software, openHAB was picked because it is a popular open-source smart home software solution, already supporting many different kinds of smart devices while also providing documentation for developing custom extensions. Additionally, it is written in Java, which the whole team had already been familiar with.

Since there are no APIs for the Linux SPI or I$^2$C drivers available for Java, a middleware was required, written in a language that has support for those drivers. For this purpose, we developed a *relay server* that served as a bridge between openHAB and the FPGA communication channels. The server was written in C, for which both SPI and I$^2$C interfaces exist, and relayed communication between SPI (towards the FPGA) and TCP sockets (towards clients connecting to it.)

To allow openHAB to interface with the relay server, a Java API implementing the communication protocol was written, which was then used by an openHAB extension (called a *binding*) specifically made for the prototype.

[3] `http://www.archlinux.org`

### Protocol

As the name implies, the SPI Prototype makes use of the Serial Peripheral Interface bus for inter-device communication. It was originally planned to use I$^2$C, but this was later changed due to diverse difficulties during development.

Messages sent via the protocol consisted of an opcode determining the type of command, some command-specific parameters and a special *termination byte* (see Figure 3.2.)

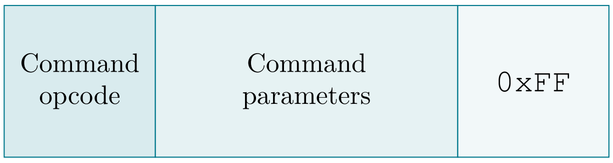


**Figure 3.2:** Format of a command message from the SPI Prototype protocol. All messages start with the command opcode and end with a termination byte (`0xFF`.) Command parameters could theoretically be of any length, but are typically between 0 and 2 bytes long.

The termination byte was employed to allow for the transmission of arbitrarily sized messages. This also meant that messages had to be sent one byte at a time, stopping only when the termination byte had been received. Since both communication parties had to await the full message before processing but still had to send something back for each byte they received, another special *filler byte* (`0xFE`) was introduced. This byte effectively meant nothing, it was merely sent whenever one device was listening to the other and was to be ignored.

The protocol supported commands for querying and setting both the state and type of devices, a handshake authentication that compares each participant's protocol version and a few other meta-commands like shutting down the relay server (see Table 3.1 on the following page for a full listing.)

Despite SPI using the master/slave model, it was found that both parties needed to be able to spontaneously notify the other at any given time about updates, such as when the FPGA needs to tell the Raspberry Pi that a button has been pressed, or when the Raspberry Pi sends a state update to the FPGA. To solve this problem, the Raspberry Pi employed polling to frequently query new messages from the FPGA, which, when necessary, it created and stored internally inside a buffer until the next polling request happened. As such, the Raspberry Pi was decided to be the SPI master while the FPGA acts as a SPI slave.

| **Command** | **Opcode** | **Parameters** | **Length** |
|---|---|---|---|
| Device state update | `0x00` | Device ID | 3 |
| Device type update | `0x01` | Device ID | 3 |
| Query device state | `0x10` | Device ID | 3 |
| Query device type | `0x11` | Device ID | 3 |
| Query device state and type | `0x1F` | Device ID | 3 |
| Set device state | `0x20` | Device ID, New state | 4 |
| Set device type | `0x21` | Device ID, New type | 4 |
| Query all devices | `0x30` | - | 2 |
| Handshake | `0x40` | Protocol version | 3 |
| Disconnect from server | `0x41` | - | 2 |
| Poll FPGA commands | `0x42` | - | 2 |
| Shutdown relay server | `0x90` | - | 2 |
| Response: OK | `0xF0` | - | 2 |
| Response: Unknown command | `0xF1` | Command opcode | 3 |
| Response: Version mismatch | `0xF2` | Own version | 3 |
| Response: No such device | `0xF3` | Device ID | 3 |
| Response: Not authenticated | `0xF4` | - | 2 |
| Response: Unknown error | `0xF9` | - | 2 |

**Table 3.1:** SPI Prototype protocol commands with their respective opcodes, parameters and length (including opcode and termination byte.)

### Issues and Limitations

The approach used in creating the SPI prototype resulted in several problems. For one, the contrived architecture resulting from needing to create a middleware relay server made the entire project error-prone as well as hard to maintain and debug. It is especially hard to engineer software written in pure C that is both stable and maintainable, and while the relay server has worked well enough in the limited time it was tested, it was very difficult to ensure its reliability in the long run.

The bridge's hardware logic was nearly completely monolithic and thus very hard to maintain. Moreover, logic for each input and output port was hard-coded, so any potential change of device configuration required major changes in the hardware description code.

openHAB is a very extensive software suite that ran extremely slow on the limited Raspberry Pi hardware. Starting up openHAB until it was ready to use took anywhere between two and five minutes, and there was a noticeable delay between using the UI to change a device's state and seeing the actual change taking place. Furthermore, openHAB's internal architecture is complicated and, at the time of development, the provided documentation was confusing at best and incomplete at worst, causing severe problems in developing the necessary binding. As a result, while the prototype did have preliminary support for controlling devices remotely, it was unstable and missing several key features, such as automatically updating the UI when a device has changed its state by itself.

The design of the communication protocol also introduced its own share of problems. For each participant in the communication chain (FPGA, relay server and end-user API), the protocol was implemented symmetrically, meaning that each participant had to support handling *all* of the commands as defined in the protocol, even if certain commands were never meant to be received by any specific entity. For example, the *Shutdown relay server* command was only meant to be sent to the relay server, but could theoretically be sent to any of the three participants. This resulted in a lot of redundant programming in all three of the protocol's implementations.

Furthermore, the protocol was not secured against bit errors as induced by noise or faulty connections. As the SPI standard does not include any error-checking means by itself, both the FPGA and SPI software drivers would not be able to tell connection problems due to loose connectors[4]. Messages corrupted due to noise would also be interpreted as they were.

[4]Being that the SPI data signal is active low, all "incoming" bytes would thus be read as `0xFF` which the protocol would in turn interpret as the end of a message.

Several of these problems will be addressed in the design of the solution worked towards in this thesis, and both the architecture and protocol will be reworked to mitigate them.

## 3.2 Solution Architecture Overview

Before starting to work on the individual components, we shall take a look at the complete architecture we're going to construct in this chapter, and in what ways it will differ from the architecture of the SPI Prototype.

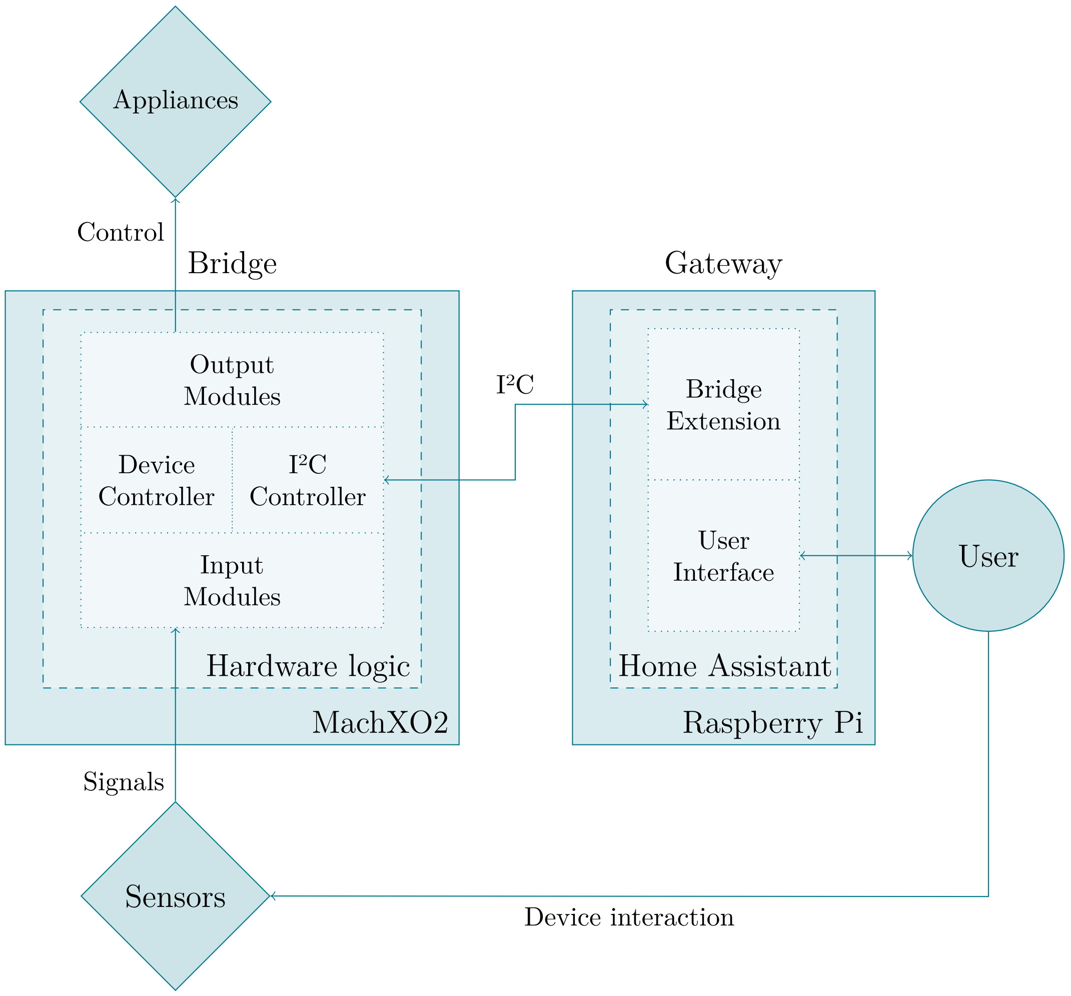


**Figure 3.3:** Overview of the revised architecture of the smart home solution, displaying an overall simplified architecture due to the removal of the relay server and a slightly more sophisticated bridge hardware logic.

As pictured in Figure 3.3 on the preceding page, the overall hierarchy will be very similar to the prototype. The appliances and sensors will remain connected to the MachXO2, which is forming the bridge. The Raspberry Pi will still serve as the gateway.

The general idea of the device control flow will also remain the same. Appliances and sensors will be connected to the MachXO2, and it will control the appliances based on input coming from the sensors. In normal operation mode, the sensor data will be relayed to the Raspberry Pi and processed by the gateway software. By means of those input signals, automation rules and the user interface, the gateway will send appliance control commands back to the MachXO2, which then changes the outgoing control signals accordingly.

Should the gateway become unresponsive, the bridge will automatically change operation and directly send the sensor input signals to the appropriate appliances, ensuring that the whole system remains operational even in absence of the gateway.

In order to resolve some of the issues encountered in the previous design, significant changes will occur to the internal structure of both bridge and gateway.

Regarding the bridge, its hardware logic will be designed in a much more structured manner. Abolishing the monolithic design approach, the logic will be modularized instead, allowing for easier maintenance and, eventually, streamlines the process of changing device configurations considerably.

On the gateway side, the exchange of openHAB for a different solution, Home Assistant, will remove the need for a relay server, as the software will be able to use the Raspberry Pi's communication interface on its own. This simplifies several aspects of the solution, including its overall architecture and the inter-device communication protocol.

As for the protocol itself, aside from the aforementioned change in architecture causing the number of required commands to be reduced, the underlying bus standard will also be changed from SPI to I$^2$C.

### 3.2.1 Ensuring Resilience

Several of the design choices outlined above were intentionally made to improve the system's overall reliability. Since this is one of the project's primary goals, we shall summarize these measures once more.

**Preventing a single point of failure.** The "standalone mode" of the bridge is a key feature for increasing the general resilience of the system. Being that the gateway is a computer, its chance of software-caused failure is much higher than it is the case with the relatively simple FPGA bridge. The particular behavior of the bridge that allows this will be outlined during the section on the design of its logic, in Subsection 3.3.1 on the facing page.

**Stabilizing the protocol.** The change to the I²C standard slightly improves the stability of device-to-device communication due to some of the nuances of its base protocol. In particular, the presence of the acknowledge bit already introduces a certain amount of hardware-side error checking. The protocol will be even further stabilized against noise-induced bit errors by appending CRC error-checking data. This will be explained in more detail in Subsection 3.4.5 (pg. 44).

**Simplifying the architecture.** Compared to the SPI Prototype, the new design approach does away with the need for a relay server, a choice that will allow for a more straightforward communication flow. At the same time, it removes what had arguably been the weakest link in the chain in terms of stability. The reason for this change in design will be the reconsideration of the gateway software, explained in Subsection 3.5.2 (pg. 46).

**Using a stable operating system.** Even though we cannot completely prevent a computer from becoming unresponsive, we can at least take measures to attempt maximizing its stability. Creating an appropriate environment suitable for a server that is both stable and secure is one of these measures. This will be talked about more in Subsection 3.5.1 (pg. 46).

## 3.3 Designing the Bridge Hardware Logic

This section deals with the design of the smart bridge, later to be implemented on the MachXO2 FPGA using Verilog in Section 4.1 (pg. 61).

The central idea of the design approach used here is to split the bridge's responsibilities – appliance control, sensor monitoring, I²C handling and so on – into smaller, independent modules that communicate with each other as needed. Much like in modular software design, this separation of concerns will allow for easier development and testing of the individual modules.

This design includes four major components, each handling a different area of responsibility inside the bridge. Centrally important are the **Device Controller**, which manages the state of all appliances and handles I²C messages, and the **I²C Controller**, an interface for the MachXO2's internal I²C hardware.

The modular design approach will also allow us to improve on the previously hard-coded, specialized hardware logic of the SPI Prototype. Two modules called the **Output** and **Input Device Managers** will be created, which in turn contain several **output** and **input modules** each. Designed as an interface towards each type of appliance and sensor, respectively, these can be instantiated as many times as needed, allowing the FPGA to be easily reconfigured to any desired device configuration. If using the specialized management UI that will be designed in Section 3.6 (pg. 56), this can even be accomplished without the need to manually adjust any hardware description code.

Furthermore, both output and input modules are each connected to a **bus** residing along them, which is needed for internal communication with the Device Controller.

A simplified, schematic overview of the modules designed for the hardware logic is presented in Figure 3.4 on the following page.

Each of the modules will now be described in greater detail, including a schematic of their relevant inputs and outputs. Note that these schematics are only meant to clarify the relationships between each of the modules, and thus some I/O signals have been omitted for the sake of clarity. For example, most modules also contain an input for the clock signal generated by the FPGA itself, which is what drives the module's internal logic.

### 3.3.1 Device Controller

The *Device Controller* is the centerpiece of the FPGA hardware logic. Its primary objective is to manage the state of all devices connected to the bridge. It also handles and responds to I²C messages the I²C controller receives, during which it communicates with the output modules to set or read the states of corresponding devices accordingly. While reading and writing those messages, it also performs checksum calculation and verification, as detailed in Subsection 3.4.5 (pg. 44).

It is also responsible to handle updates coming from both input and output modules by monitoring the State Update and Input buses, respectively. The handling of those events differs depending on whether the Raspberry Pi is currently communicating with the FPGA. To that end, the Controller keeps track of the Raspberry Pi's perceived state by declaring it dead if no I²C messages have been received within the last five seconds. The Controller behavior then changes in the following ways:

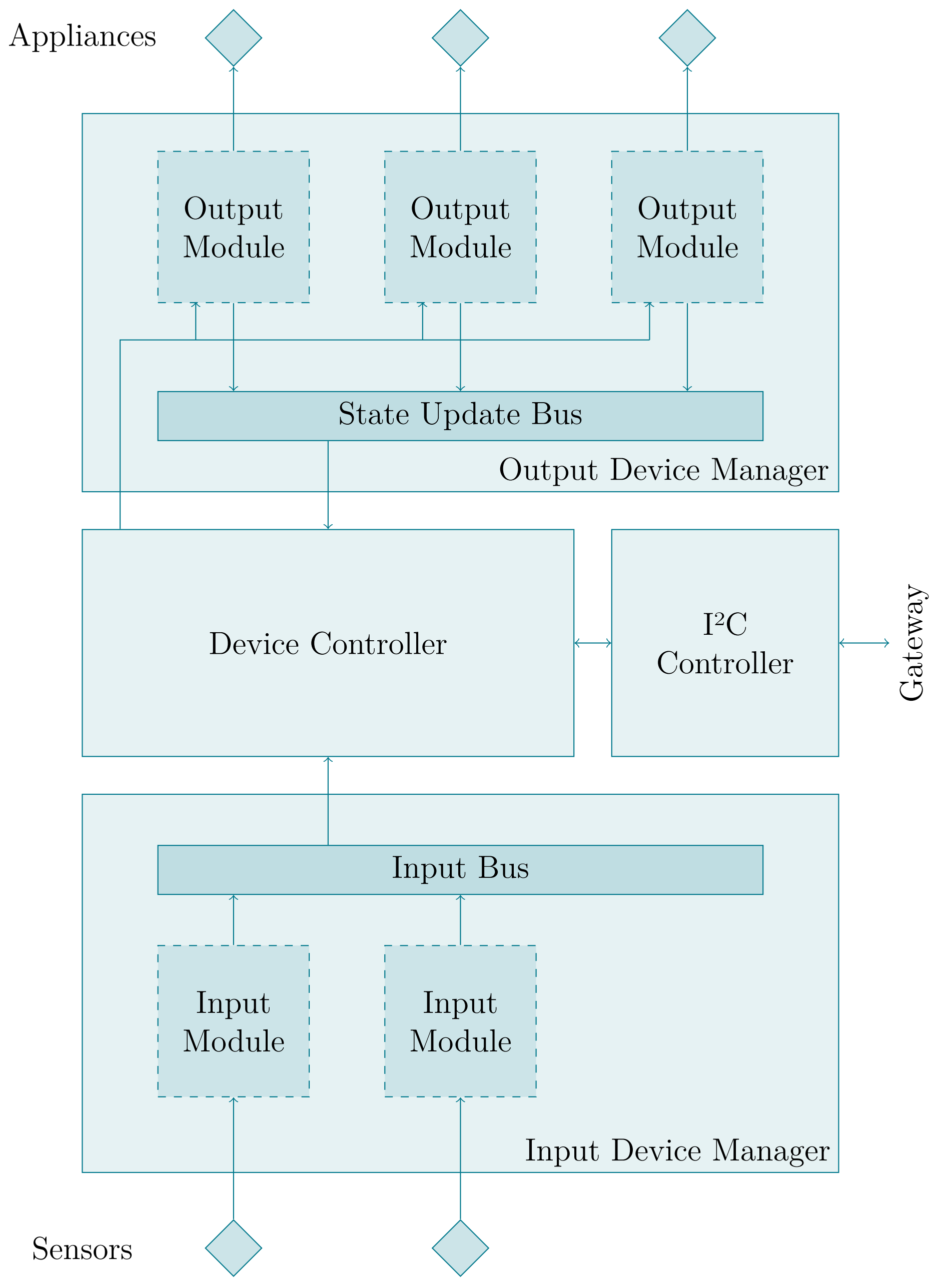


**Figure 3.4:** Schematic overview of the FPGA hardware logic blocks and their communication flow. The variably instantiated Input Modules send signals coming from sensors to the Device Controller, which in turn controls appliances through instantiated Output Modules. The I²C Controller handles communication with the gateway.

- If the Raspberry Pi is *alive*, the Device Controller does not handle the State Update and Input events itself. Instead, on the next polling query it receives, it relays those events to the Raspberry Pi for processing, then discards them. It reports the contents of the State Update queue first and Input queue second, and expects the Raspberry Pi to keep polling until no more events are available.
- Should the Controller consider the Raspberry Pi to be *dead*, it ignores all State Update events – since those are only meant for updating the Raspberry Pi's device representations – and handles all Input events itself, by directly sending the requested state to the output module that is assigned to the input module which fired the event.

Communication with the I²C Controller will be detailed in the section about that module, Subsection 3.3.4 (pg. 38).

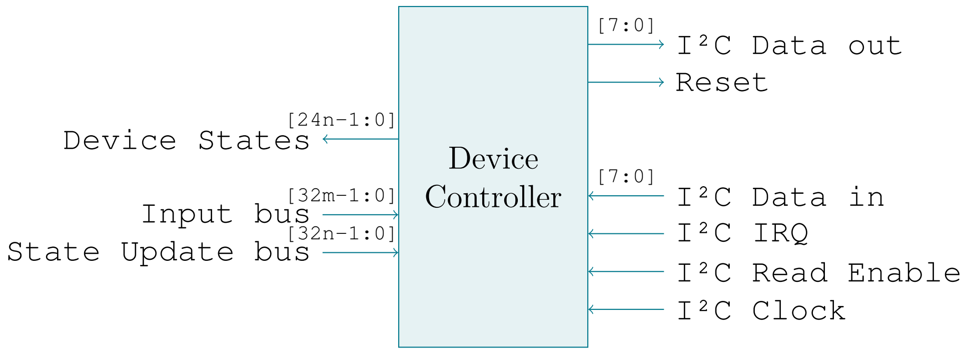


**Figure 3.5:** Device Controller schematic. $n$ and $m$ are the number of appliances and sensors in the current configuration, respectively.

### 3.3.2 Output Modules and Manager

Functionality for each type of appliance is consolidated within *output modules*. They contain certain logic that directly controls a device's control signal(s), and for every appliance connected to the bridge, a corresponding module is to be instantiated. As such, they could also be seen as a kind of *device driver*.

The control signals emitted from a module depend on a three-byte (or 24-bit) value called a *state*, which it takes as an input from the Device Controller. Compared to the SPI Prototype, where a device's state was represented by only a single byte, the larger data width allows for more fine-grained control over more sophisticated appliances. A RGB lamp, containing three LEDs that are each dimmable individually, was chosen as a measure for this value. Using the SPI Prototype, such a device had to be split into three separate entities; one for each LED to be controlled. The new data width allows it to be represented as a single device with sufficient granularity in its color control capabilities.

At initialization, the state of all output modules is zero.

Since the state of an output module is exclusively controlled by the Device Controller, it follows that the module cannot control its own state. In certain cases, however, this might be a necessity; e.g. if an output module stops a device on its own after a short while. To make it possible for a device to spontaneously and autonomously update its own state, the *State Update bus* is introduced, which serves as an interface for state update events towards the Device Controller. Should an output module require to update its own state, it can write an event, containing its device ID and the new desired state value, onto the Bus, later to be handled by the Controller, as described in Subsection 3.3.1 (pg. 27).

Thus, not every output module actually requires access to the bus; in fact, only one of the modules designed for this solution makes use of the ability to update itself.

The logic of certain output modules may also be customized through configurable parameters.

Serving merely as a container for all output modules used in the current device configuration, the *Output Device Manager* barely contains any important logic by itself. However, it does provide a *bus arbiter* that controls the output modules' access to the bus to prevent two modules from writing to it at the same time; a scenario that would usually result in undefined behavior.

As a first proof of concept, four output modules have been designed for this solution, meant for controlling some appliances that are commonly found in smart home installations: simple devices with an on/off switch, dimmable lights, RGB color lights and shutters. These modules will now be described in greater detail, including a brief description of their logic, output signals and parameters, if applicable.

#### Generic Binary Output

The simplest example of an output module, the *Binary* module manages a single control signal which it can turn either on or off. It is most suitable for simple appliances that possess no further fine-tuning in their operative modes.

The module activates the control signal when the input state is non-zero. Otherwise, the signal is cut.

The Binary module does not have any configurable parameters.

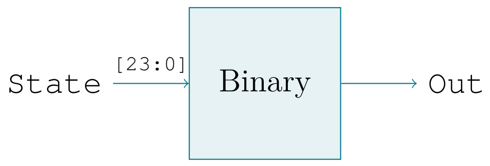


**Figure 3.6:** Binary output module schematic.

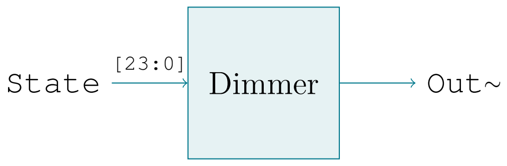


**Figure 3.7:** Dimmer output module schematic.

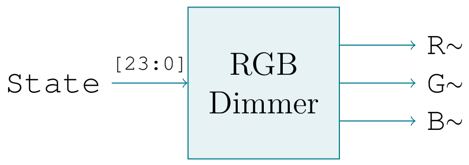


**Figure 3.8:** RGB Dimmer output module schematic.

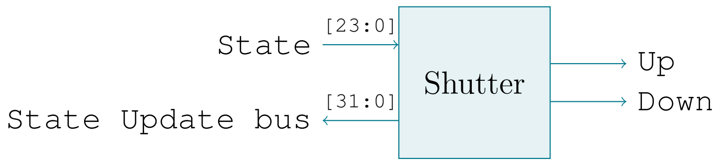


**Figure 3.9:** Shutter output module schematic.

### Dimmer Output

The *Dimmer* module also manages only a single control signal. Compared to the Binary module, however, that signal is controlled via PWM. As its name implies, this module is primarily meant for dimmable LEDs or other devices supporting PWM input.

The frequency of the PWM output is dependent on the lower eight bits of the input state; the other bits are disregarded. As such, a state of `0x000000` turns the signal off completely, `0x0000FF` represents a constant high signal, and the frequency gets adjusted in a linear manner for any values in between those two.

When the input state is changed, instead of assuming the new value immediately, the Dimmer module slowly fades towards the desired target state, at a rate of one unit every $t_{\text{fade}}$ milliseconds, providing a slightly more aesthetic presentation when adjusting a light's brightness, for example.

The value of $t_{\text{fade}}$ can be adjusted through the module's `DELAY` parameter. For example, at the default value of 10 ms, a change from state `0x30` $= 48$ to `0x60` $= 96$ would take approximately $(96 - 48) \cdot 10$ ms $= 480$ ms.

### RGB Dimmer Output

An output module specifically designed for color lamps, the *RGB Dimmer* module works exactly like the Dimmer, except that it contains three control signals that can each be controlled separately from each other.

The PWM frequencies of the red, green and blue components are controlled by the first, second and third byte of the input state, respectively. In that way, the state value closely resembles a RGB hex color triplet notation, as it is commonly used e.g. in HTML. For example, the state `0xFF7700` produces an orange color.

Like the Dimmer module, the RGB Dimmer fades smoothly towards a newly set state, independently for each color component; and the update frequency $t_{\text{fade}}$ can be set with the `DELAY` parameter as well.

### Shutter Output

Finally, the *Shutter* output module is designed for controlling retractable, electronic shutters, window shades or the like. It controls two output signals, one for moving the shutter down and another one for moving it up; each to be connected to the appropriate inputs on the shutter's motor.

Typical for common electronic shutters is a two-button interface, where one button moves the shutter up and another moves it down. For convenience, holding one of the buttons causes the shutter to move all the way in the specified direction either until the shutter has reached the end, or when a button has been pressed again.

The logic of the Shutter module is inspired by this behavior. It internally distinguishes between five possible states: one idle state, two states for moving the shutter up or down just for a short while, and two more for moving the shutter up or down fully. The output signals directly depend on this state, as seen in Table 3.2. This also ensures that both up and down signals cannot be sent at the same time, preventing behavior that could potentially damage a shutter motor.

| **State name** | **Value** | **`Up` signal** | **`Down` signal** |
|---|---|---|---|
| Idle | `0x00` | 0 | 0 |
| Move up once | `0x01` | 1 | 0 |
| Move down once | `0x02` | 0 | 1 |
| Move up fully | `0x03` | 1 | 0 |
| Move down fully | `0x04` | 0 | 1 |

**Table 3.2:** Shutter output signals, depending on the module's state.

Internally, the module's logic works similar to a state machine. When in any non-idle state, the module requests, via the State Update bus, to be set back to the idle state after the appropriate amount of time has passed (see Figure 3.10 on the next page.) The Device Controller has the capability to set the module's current state to any of the defined states, at any time. On any state change, induced externally or internally, the timer counting towards the state change request will be reset.

An appropriate Input Module will later be designed to complement this behavior.

The Shutter module supports two parameters: the `TICK_MS` parameter defines the duration of a single movement cycle ($t_{\text{once}}$), i.e. for how long the up or down signal will be sent to the motor. The other parameter, `FULL_MOVEMENT_TICKS`, defines the amount of movement cycles for moving the shutter all the way (equivalent to $t_{\text{full}}/t_{\text{once}}$). Since there is no way for the module to determine the current position of the shutter, an appropriate value should be chosen generously to give the shutter enough time to move from one end to the other. By default, a single cycle is 500 milliseconds long, and a full movement consists of 30 cycles, translating to a full movement time of $500\text{ ms} \cdot 30 = 15$ seconds.

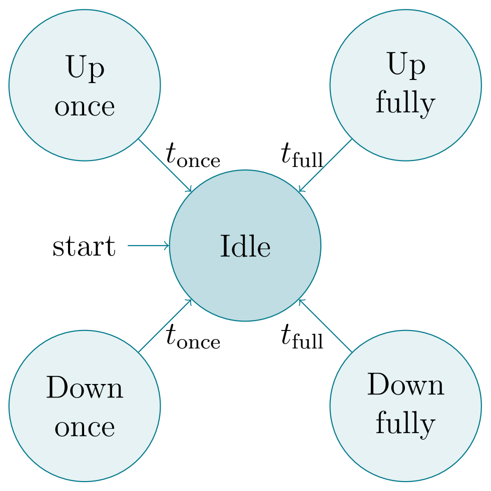


**Figure 3.10:** State transitions as able to be induced by the Shutter module itself, where $t_{\text{once}}$, $t_{\text{full}}$ are the times spent moving the shutter once and fully, respectively. Additionally, the Device Controller can put the module in any state at any time (not pictured here.)

### 3.3.3 Input Modules and Manager

As the output modules are like device drivers for the system's appliances, so too are its sensors governed by certain *input modules*.

Their responsibility is to monitor the input signals generated by the sensors and, on any change, fire an appropriate event onto the *Input bus*. Like the state update events, the input events contain the ID of the input module that emitted it alongside the new state value it calculated from the sensor's signals.

When creating a device configuration, each input module is assigned to a single output module which it is supposed to control. The Device Controller contains a lookup table that maps every input module to an output module, which it uses to route the input events to the correct appliance when the FPGA is working in standalone mode.

Similar to the Output Device Manager, the *Input Device Manager* contains no logic of its own, except for another bus arbiter managing the Input bus.

In many cases, the state value generated from the input modules can be configured through their respective parameters. Should any of those modules initialize to a non-zero state value, it fires an event onto the Input bus immediately to ensure that the connected output module also initializes to that value.

Five input modules have been designed, mostly to complement the behavior of the output modules.

### Generic Button Input

The *Button* input module simply notifies the bus whenever the state of the connected input signal has changed. Once the signal changes to low, it fires an event with the state $s_{\texttt{off}}$ =`0x000000`. Should it become high, the state to send is $s_{\texttt{on}}$ =`0x000001`. In other words, this module simply passes through the state of the input signal.

This module does not have any parameters.

### Generic Toggle Input

Very similar to the Button input module, the *Toggle* module also reports either one of two configurable states. However, this module toggles between the two on every rising edge of the connected input signal.

### Dimmer Cycle Input

The *Dimmer Cycle* module works similar to the Toggle module, except that it cycles between five different output states on every rising edge of the input signal. It is primarily meant for toggling the brightness of a dimmable lamp between five preset states.

If the states are enumerated from $s_0$ through $s_4$, it initializes to state $s_0$ and advances to $s_1$ on the first rising edge, to $s_2$ on the next, and so on until $s_4$, after which it resets to $s_0$ again.

The five states' values can be modified through the `BRIGHTNESS_`$n$ (for $n = 1 \ldots 5$) parameters, defaulting to `0x00`, `0x40`, `0x7F`, `0xBF` and finally `0xFF`, corresponding to approximate brightness increments of 25% each.

### RGB Cycle Input

An extension of the Dimmer Cycle module, the *RGB Cycle* module allows to cycle a RGB color lamp through eight different color settings. Otherwise, it works in the exact same way.

This module is primarily meant to complement the RGB Dimmer output module, and to exemplify the potential flexibility of input modules in general. In a production setting, which will allow for much more fine-grained control of the color lamp through

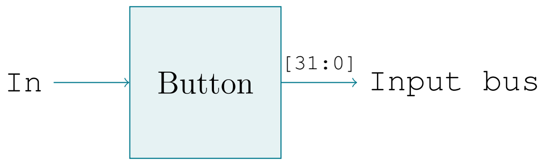


**Figure 3.11:** Button input module schematic.

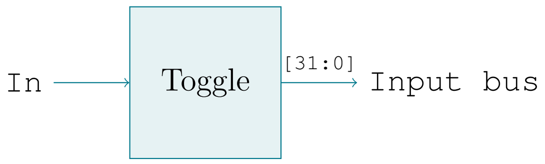


**Figure 3.12:** Toggle input module schematic.

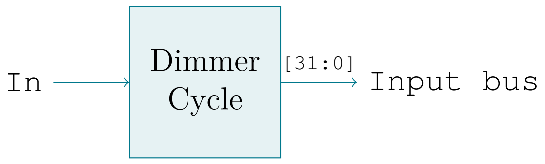


**Figure 3.13:** Dimmer Cycle input module schematic.

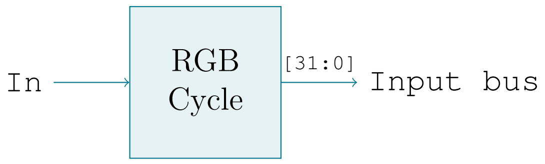


**Figure 3.14:** RGB Cycle input module schematic.

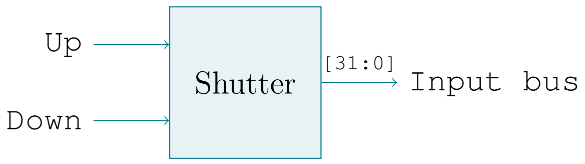


**Figure 3.15:** Shutter input module schematic.

appropriate controls contained in the gateway interface, it should not be seen as much more than a novelty.

The eight states $s_0$ through $s_7$ are configurable via the `COLOR_`$n$ (for $n = 1 \dots 8$) parameters and default to the values `0x000000`, `0xFF0000`, `0xFF7700`, `0xFFFF00`, `0x00FF00`, `0x00FFFF`, `0x0000FF` and `0xFF00FF`, representing the seven colors of a rainbow with an additional *off* setting at initialization.

### Shutter Input

This module is the counterpart to the output module of the same name, as described in Subsection 3.3.2 (pg. 32). It is meant to provide the common two-button interface for electronic shutters outlined before; where one button moves the shutter up, the other moves it down, and a long press of either button makes the shutter move all the way in the specified direction.

A state machine is used internally to realize this behavior. The idle state monitors both `Up` and `Down` input signals and changes to the appropriate state if either one of them becomes active. Should the signal still be active (i.e. the button is still being pressed) after $t_{\text{long}}$ time, then a *long press* occurred and the Shutter module writes the $s_{\texttt{up\_full}}$ or $s_{\texttt{down\_full}}$ state onto the bus, while switching onto a return state waiting for the signal to deactivate, after which the module finally returns to the idle state.

Should the input signal go low *before* the timer reaches $t_{\text{long}}$, the module recognizes this as a *short press*, firing either the $s_{\texttt{up\_once}}$ or $s_{\texttt{down\_once}}$ event as it returns to the idle state.

There is no button or button combination for stopping a shutter movement, but since the output module resets itself to the idle state after it has completed its current operation, a full movement cycle can be cancelled through a short press in any direction.

Figure 3.16 on the next page pictures the state machine used in the Shutter module.

The time needed for a long press $t_{\text{long}}$ can be configured through the module's `FULL_PRESS_MS` parameter, expressed in milliseconds and defaulting to 2 seconds.

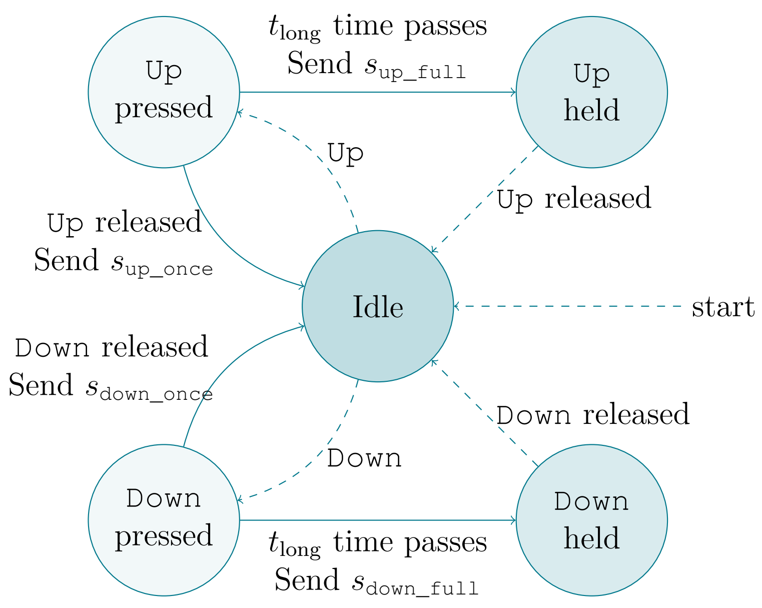

$t_{\text{long}}$ time passes
Send $s_{\texttt{up\_full}}$
Up pressed
Up held
Up
Up released
Up released
Send $s_{\texttt{up\_once}}$
Idle
start
Down released
Send $s_{\texttt{down\_once}}$
Down released
Down
Down pressed
Down held
$t_{\text{long}}$ time passes
Send $s_{\texttt{down\_full}}$

**Figure 3.16:** State machine of the Shutter input module. The dashed transitions do not send any state update to the bus.

### 3.3.4 I²C Controller

The single purpose of the *I²C Controller* is to serve as an interface for the MachXO2's internal I²C function block. Making use of its functionality involves accessing certain memory-mapped registers through the device's Wishbone bus, which is a common communication interface standard for embedded components. [Wik19c]

The complete process of sending and receiving I²C messages on the MachXO2 is, comparatively, rather extensive and would certainly complicate the Device Controller's internal logic, were it to be directly implemented within. Therefore, it seemed appropriate to create an abstraction layer the Device Controller can interface with.

The I²C Controller's logic and signals were designed according to the necessary signals and steps to be taken for communicating with the MachXO2's Wishbone bus as well as the I²C function block. This information can be found in the device's documentation [Lat16], and as such this will not be described in further detail here.

Instead follows a brief outline of the I²C Controller's usage and how it communicates with the Device Controller.

**Initializing.** The I²C Controller requires to be initialized from the Device Controller by activating the `Reset` signal for one cycle. This performs the initialization routine of the MachXO2's I²C module, as per its documentation. Once this process is finished, the Device Controller is notified by pulling only the `I²C Clock` signal high for one cycle, at which point both modules become ready for use.

**Relaying incoming I²C bytes.** At the beginning of an incoming transmission from the Raspberry Pi, the Device Controller is notified by activating the `I²C IRQ` signal, which is kept high until the transmission is over. Whenever a byte is received, it is latched onto the `I²C Data out` register and the `I²C Clock` is activated for a cycle to signify this. When the transmission has concluded, the `I²C IRQ` signal is deactivated, notifying the Device Controller that it can process the message and prepare a response.

**Requesting I²C response bytes.** Immediately after sending a message, the Raspberry Pi can restart the communication to request a response from the FPGA. As before, the Device Controller is notified of this by activating the `I²C IRQ` signal. In addition, the `I²C Read Enable` signal is pulled high, which causes the Device Controller to write the next byte of the response message to the `I²C Data in` register on every pulse of the `I²C Clock` signal. After requesting exactly eight bytes from the Device Controller, both `I²C IRQ` and `I²C Read Enable` signals are deactivated, signifying the Device Controller that the outgoing transmission is over.

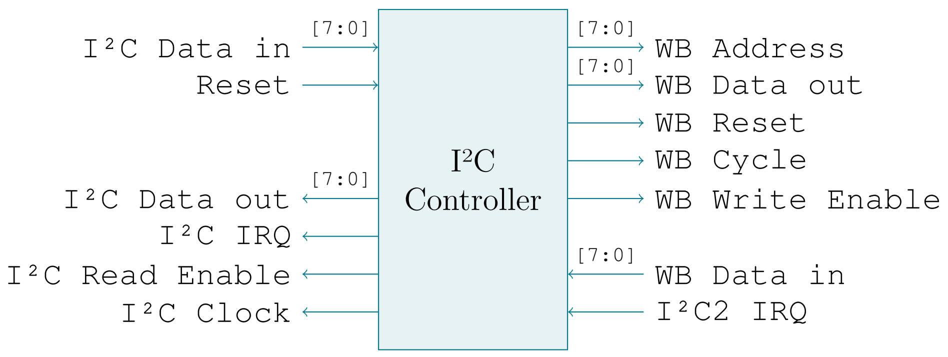


**Figure 3.17:** I²C Controller schematic.

## 3.4 Designing the Protocol

Providing a stable and efficient means of communication between the bridge and gateway is an essential part of this solution. The protocol used in the SPI Prototype unfortunately showed several flaws, yet remains as a good foundation for a new, improved design that will be described in this section.

First, we will briefly revisit the choice of the underlying **bus system**. It has already been mentioned several times, but the bus standard used in the SPI Prototype, will be abandoned in favor of I$^2$C, and the reasoning behind this decision will follow shortly. Another big change lies in the approach to the **protocol symmetry**, which had caused some issues before. Next, the protocol messages themselves, together with how they are now divided between **commands** and **responses**, will be explained, and finally we will briefly introduce **CRC** as a means of further securing the protocol against faulty transmissions due to potential noise.

### 3.4.1 Choosing a Bus System

Before going into more detail about the design of the protocol itself, an appropriate bus system standard has to be chosen first.

Both the MachXO2 and the Raspberry Pi support SPI and I$^2$C natively in hardware, making those two standards the best options for this solution. It could also be considered to design a proprietary bus from scratch, but, at least on the Raspberry Pi, its implementation would have to be accomplished purely in software ("bit-banging"), which would be several magnitudes slower than using the interfaces already present in the hardware. Also, the use of an open and well-documented standard like SPI or I$^2$C makes it easier for potential future developers to pick up work on this project, if the need ever arises.

The SPI Prototype, of course, used SPI as the communication bus between the FPGA and the Raspberry Pi. This had worked well enough, but even back then, I$^2$C appeared as the more elegant solution in regards to its scalability as well as the added error-checking capabilities that SPI doesn't provide.

Moreover, the fact that one of the major obstacles to using I$^2$C in the SPI Prototype was, in hindsight, merely a limited understanding on how to correctly interface with the MachXO2's internal I$^2$C hardware was not seen as a valid hindrance for this project. Enough time has been spent on studying the device's I$^2$C support as well as the design of hardware logic to complement it (as documented in Subsection 3.3.4 (pg. 38)), making this a non-issue going forward.

The Raspberry Pi remains as the I$^2$C master in this design, and thus keeps the responsibilities of initiating messages and setting the length of each transmission. The MachXO2 keeps serving as an I$^2$C slave. Polling remains as the method of choice to allow the slave to notify the master about updates.

### 3.4.2 Protocol symmetry

A significant change in the protocol is a different approach to its implementations across the whole architecture. Before, each participant (bridge, relay server and gateway) had to implement the protocol symmetrically to each other, needing to support every single one of the protocol's commands and handling thereof.

Since there is no need for a relay server between bridge and gateway in this design, these two devices can now communicate directly with each other, allowing the protocol to be implemented asymmetrically. Now, while both devices obviously still have to understand the full protocol, only the Raspberry Pi needs the capability to send full command messages including opcodes, while the FPGA merely has to reply to those with simple responses consisting of a status code and some data, if applicable. This significantly simplifies the protocol and its implementations, while also reducing the overall number of different commands.

### 3.4.3 Commands from the Raspberry Pi

The general format of a command message as to be sent from the Raspberry Pi can be seen in Figure 3.18. It largely remains the design of the messages used in the SPI Prototype, the major difference being the addition of two bytes of CRC error correction data.

Also, the termination byte marking the end of a message has been retired, since the ability of the MachXO2's I²C implementation to reliably tell the end of a transmission makes it obsolete.

| | | |
|---|---|---|
| Opcode<br>1 byte | Parameters<br>0 - 4 bytes | CRC<br>2 bytes |

**Figure 3.18:** Protocol message format.

A few of the commands from the previous protocol were deemed redundant and removed, in part by the aforementioned change to protocol symmetry. The new complete list of commands can be found in Table 3.3 on the following page.

### 3.4.4 Responses from the MachXO2

The general format of response messages is very similar to that of the command messages, and can be seen in Figure 3.19 on the next page. In place of a command

| Command | Opcode | Parameters | Length |
|---|---|---|---|
| Get appliance state | `0x00` | Appliance ID | 4 |
| Get appliance type | `0x01` | Appliance ID | 4 |
| Get sensor type | `0x02` | Sensor ID | 4 |
| Set appliance state | `0x10` | Appliance ID, New state | 7 |
| Get FPGA version | `0x20` | - | 3 |
| Reset FPGA | `0x2F` | - | 3 |
| Poll events | `0x30` | - | 3 |
| Repeat last message | `0x40` | - | 3 |

**Table 3.3:** Commands used in the new Raspberry Pi to FPGA communication protocol, including opcodes, parameters and message length including CRC data.

opcode, responses start with a status code determining the command result. The possible status codes are listed in Table 3.4.

Of note is that, to simplify the implementation of the protocol in hardware logic, response messages always have a fixed length of eight bytes. This number was chosen to accommodate for the longest possible response data, which is five bytes in length. CRC data is always sent last, and any unused bytes in between are filled with zeroes.

The fixed message size also eliminates the need for a termination byte here.

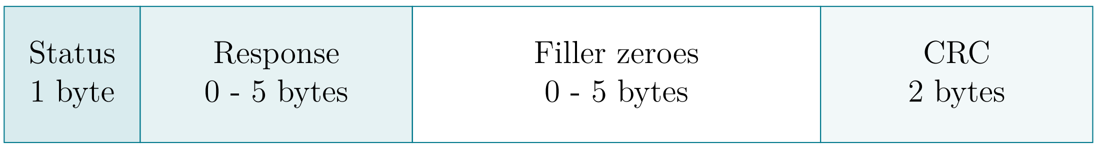


**Figure 3.19:** Format of responses to protocol commands.

| Status | Code | Description |
|---|---|---|
| OK | `0xF0` | The command was successful. |
| Error | `0xF1` | The command was not successful. |
| No data | `0xF2` | No data is available for a polling request. |

**Table 3.4:** Protocol response status codes.

Following the status code is some response data that depends on the command that was sent. No further metadata about the nature of the response data is included in the message, and as such the Raspberry Pi needs to deduce the meaning of the response from the current context. However, since most commands sent to the

| Command | Expected response data |
|---|---|
| Get appliance state | Appliance ID, Appliance state (3 bytes) |
| Get appliance type | Appliance ID, Appliance type |
| Get sensor type | Sensor ID, Sensor type |
| Set appliance state | None |
| Get FPGA version | FPGA version (2 bytes), Highest output ID, Highest input ID (see notes) |
| Reset FPGA | None |
| Poll events | Update event (5 bytes) (see notes) |
| Repeat last message | Varies (see notes) |

**Table 3.5:** Protocol commands and their expected responses.

MachXO2 expect a certain, fixed response, this does not pose a problem. A full list of expected response data to each of the commands is listed in Table 3.5.

Should a command result in an error, the response data includes the error code and some relevant parameters. The possible error codes are listed in Table 3.6.

| Error | Code | Parameters |
|---|---|---|
| Unknown command | `0x10` | Opcode of unknown command |
| Unknown device | `0x20` | ID of unknown device |
| CRC failure | `0x30` | Erroneous CRC data |
| Unknown error | `0xFF` | - |

**Table 3.6:** Protocol response error codes.

Following are a few notes on certain responses that require slightly more explanation.

### Get FPGA version

When this command is sent, the MachXO2 responds with a two-byte version constant, followed by the highest device ID of all appliances and sensors each. This information is meant to aid the Raspberry Pi during the beginning of a connection, when it is querying the FPGA for all of its devices.

### Poll events

The appropriate response to this command depends on the contents of the Device Controller's internal update event buffers.

Input Events are prefixed with `0x00` and contain the ID of the input module that emitted the event as well as its three-byte state value.

State Update Events are prefixed with `0x01` and contain the ID of the output module that has updated itself and its new three-byte state value.

If both buffers do not contain any events, the MachXO2 responds with the No data status code (`0xF2`) instead.

#### Repeat last message

This command is sent to the MachXO2 in case the Raspberry Pi detects a CRC error in an incoming transmission, asking the MachXO2 to repeat the last valid message it sent.

### 3.4.5 Using CRC to Secure the Protocol

To further increase the reliability of device-to-device communication across potentially noisy cabling, the protocol was secured by adding 16 bits of CRC data to each message. CRC was chosen as a means to secure the protocol because it is very easy to implement in both hardware and software while at the same time being very efficient and effective at detecting noise-induced errors in network transmissions. [Cha01]

CRC consists of adding a hash value or digest, also called the *check value*, generated from the message data that is being appended to the message when transmitting. Both sender and receiver calculate this value independently from one another; this way, the receiver can safely tell whether the incoming data has been corrupted by noise.

In particular; when transmitting a message, the sender calculates the message data's CRC value by interpreting the message's bits as a single, long polynomial, where each bit of the message corresponds to the polynomial's coefficients. This polynomial is then divided by a special, fixed *generator polynomial*, which results in a quotient and a remainder. The former is discarded, while the latter is converted back into bits using the same approach as before. These bits are then appended to the message payload and both are sent to the receiver as a single message.

On the receiving end, the CRC data for the incoming message is calculated in the exact same way, using the same generator polynomial as well. Since the remainder of the polynomial division has been appended by the sender to the original payload, the resulting message polynomial is, by definition, always guaranteed to be divisible by the generator polynomial without leaving a remainder. In case one or more bits of the message have been corrupted by noise, it is extremely unlikely that the division still results in a remainder of zero. Therefore, the receiver can verify any message by simply checking whether its own CRC calculation results in a remainder of zero. If it is not zero, the receiver assumes that the message is not valid, and can take appropriate measures from there. [PB61]

CRC calculations are characterized by two important factors: the *generator polynomial*, and the resulting *length of the CRC data*.

The generator polynomial is a polynomial $P$ of degree $n$, where $n$ is the desired length of the check value. Additionally, $P$ belongs to the finite field $GF(2)$, meaning each of its coefficients can only either be a 0 or 1. This allows for a direct translation from binary digits to the polynomial's coefficients. Depending on the notation, either the highest or lowest degree term has a fixed coefficient of 1.

Being the divisor in the CRC calculation, the generator polynomial determines the number of remainders that can be generated by the division, and thus indirectly the types of errors that can be detected. Therefore, many different polynomials for various applications have already been analyzed and used [Koo15].

The degree of the generator polynomial directly translates to the length of the resulting CRC data. A larger degree increases the amount of errors that can be detected while reducing chances of possible collisions due to the greater amount of bits available. On the other side, this leads to a slightly larger overhead for each message and in certain cases also might increase the computational complexity.

For this protocol, where message data is up to 6 bytes = 48 bits in length, a CRC data size of **16 bits** has been chosen. Depending on the length of the data to be transmitted, this increases the message overhead by 33 to 200 percent. However, since transmission bandwidth and speed are not an issue in this application, this was found to be a reasonable trade-off for practically removing the possibility of uncaught bit errors occurring. Even though a 8-bit wide CRC would probably be sufficient for relatively short payloads as they are used in this protocol, it could only produce one of $2^8 = 256$ different values for each message, resulting in a relatively high chance of collision. With the $2^{16} = 65\,536$ possible values resulting from using a 16-bit code, this chance is much smaller.

Based on the work done in [Cha01], the polynomial $x^{16} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^4 + x^2 + 1$ (encoded as `0x2F15` when omitting MSB) is an optimal generator polynomial of degree 16 for message payloads of up to 64 bits. As such, it was deemed appropriate for use in this protocol.

## 3.5 Designing the Gateway

Finally, we will have a look at the other significant part of the smart home system, the gateway. Compared to the bridge, which was designed almost completely from scratch, the gateway will make use of several third party software components, and as such the amount of work needed to be done is comparatively smaller. Nevertheless, it is an important part of the whole project.

The whole gateway stack contains four major components. First, an **operating system** for the Raspberry Pi has to be chosen that, in turn, runs a suitable **open-source gateway software solution**. Next, to allow the Raspberry Pi to interface with the bridge, an **API** for the communication protocol designed in Section 3.4 (pg. 39) has to be created, which finally gets integrated into a custom-made **extension** for the gateway software.

### 3.5.1 Setting Up a Host Environment

*Arch Linux* remains a suitable choice for the operating system of the Raspberry Pi. Its qualities were already discussed in Subsection 3.1.2 (pg. 20). Between the choices of operating systems available for the Raspberry Pi's ARM architecture, it was seen as offering a good ratio of usability and suitability as a server environment.

It also happens to be my operating system of choice for personal use, and several years worth of experience setting up a familiar environment significantly help in maintaining a healthy system.

Finally, it has already worked very well as a server environment for the SPI Prototype.

### 3.5.2 Choosing a Gateway Software

In the SPI Prototype, openHAB was chosen as a gateway software because of its popularity and its main development language, Java, already being familiar to the team.

As noted before, however, many problems were encountered in its usage including slow response times on the Raspberry Pi, a limited understanding of its development resources as well as the fact that I$^2$C support had to be added through some kind of middleware.

Considering this, Home Assistant[5] appears an attractive alternative to openHAB.

[5]`http://www.home-assistant.io`

The two software projects are very similar, in that Home Assistant also provides support for numerous devices through extensions and offers sufficient documentation for enabling developers to write their own. The major difference between the two is that Home Assistant and its extensions are written in Python, solving the problem of API availability for SPI and I$^2$C, because Python libraries exist for both[6].

### Weighing Python against Java

The choice of the gateway software Home Assistant also obviously infers the use of Python as the programming language for the API (and, of course, the Home Assistant extension that will use it) instead of Java or even other languages.

While Java is a strongly typed programming language with static type checking, Python instead makes use of duck types and supports run-time type checking only. From a software engineering standpoint, the type safety options arguably make Java a better choice, especially for a system designed with reliability in mind. Using Python, the risk of application failure due to typing errors is higher.

The absence of compile-time type-checking can be somewhat compensated for by the use of a modern development environment capable of performing static type checking. Even though Python uses dynamic typing, it optionally supports type annotations, which can then be used for static analysis.

In terms of run-time performance, even though openHAB struggled to do well on the Raspberry Pi in the SPI Prototype, it is not possible to determine beforehand whether Home Assistant will perform better, as those analytics depend on a number of factors other than just the choice of language, and heavily vary on a case-by-case basis. Both Python and Java are first compiled to byte code and then executed on a virtual machine; though Java has the capability to produce native machine code through its just-in-time compiler, a feature that Python lacks. Thus, generally speaking, Java code might perform better than Python code. In any case, once the finished solution will be evaluated in Section 5.1 (pg. 79), this question will be revisited.

As for the Bridge API, it will be lightweight enough that the choice of language will probably not make a difference regarding its performance.

Personal experience and preference also each play a significant part in the potential efficiency of a programming language, as it is arguably the same with any tool. While my experience of both Java and Python amount to approximately the same at the time of writing, between the two I personally strongly prefer Python for a number of reasons, including its much more elegant syntax and approach to deployment.

[6]Packages `spidev` and `smbus2`, available from `http://pypi.org`.

Overall, it is expected that developing using Python should result in an API that is nearly as stable and efficient as if done in a language like Java, should enough attention be given to good programming practices.

### 3.5.3 Designing the Bridge API

The gateway software Home Assistant needs some way to use the protocol so that it can communicate with the bridge. Theoretically, the extension that will be created for Home Assistant could make direct use of the low-level I$^2$C bindings. However, this would result in an extension with a lot of complicated, hard to maintain, low-level hardware-specific code. This approach is also recommended against by the Home Assistant developer documentation [Hom20b], most likely for this very reason.

Rather than directly including such functionality within the extension itself, a much cleaner solution is to design an external *API* – an *application programming interface* – that can later be used by the extension. In the interest of separation of concerns, the API will serve as a layer of abstraction from the comparatively complicated process of using the Raspberry Pi's I$^2$C interface by taking the responsibility of handling the details of I$^2$C communication, while exposing a simplified interface.

To make it accessible for developers, the API should therefore make extensive use of information hiding and have simple and self-explanatory method and class names. [Blo06]

#### Layers of Abstraction: API Overview

The API contains three layers of abstraction, each serving as an interface towards the next. At the top level, the Home Assistant extension will be able interface with representations of the bridge itself as well as the appliances connected to it. The next level consists of an instance representing the I$^2$C bus, which provides an interface for the communication protocol described in Section 3.4 (pg. 39). It does so by abstracting from the Python I$^2$C bindings. Finally, the bindings and, by extension, the protocol, can be interpreted as an abstraction layer for interfacing with the bridge hardware itself.

An overview of this architecture is pictured in Figure 3.20 on the next page.

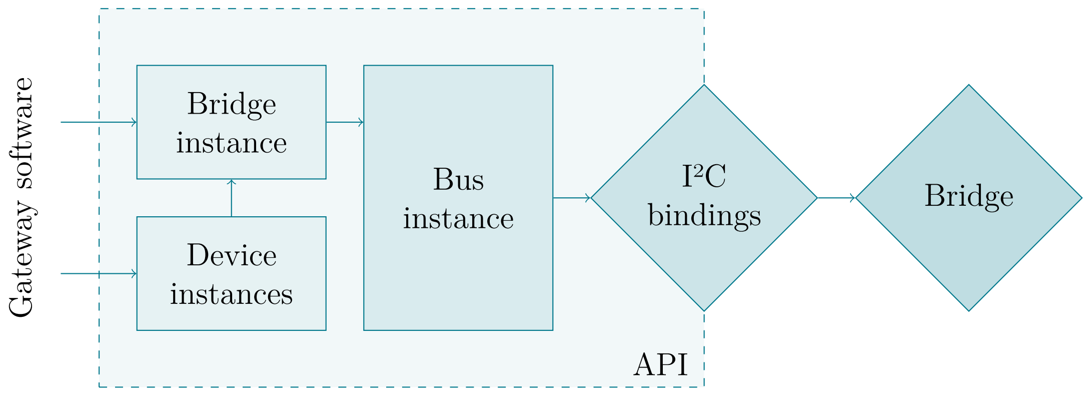


**Figure 3.20:** Bridge API abstraction layers and interface access. Each layer provides abstraction to the next. Arrows indicate interface access.

### Representing the Bridge

An object of the `I2CBridge` class is the first to be instantiated when using the API. On initialization, it uses the `I2CBus` layer to connect to the bridge hardware and query it for its status and all connected devices. It creates corresponding `I2CAppliance` and `I2CSensor` instances and provides them inside publicly accessible dictionaries, indexed by their respective device IDs, for the gateway software to access.

This class is also responsible for the automatic polling of State Update and Input events from the bridge, as previously explained in Subsection 3.3.1 (pg. 27). Incoming events will be forwarded to the corresponding device objects, allowing them to update their local state and to notify the gateway about the update.

### Representing Appliances

Each appliance that is currently connected to the bridge is represented by an appropriate object that has two main responsibilities: keeping track of the device's state, and exposing an interface towards the gateway software for controlling it. As each type of appliance mostly contains a different set of controls and control states, so contains each device class a different interface suited to the type of appliance it represents.

Internally, most functionality related to device control that is executed via the communication protocol, including setting and querying a device's state, is universal to all different types of devices. As such, it seems fitting to gather common functionality within a superclass, so that later, the implementation of specialized classes for each type of appliance can be kept as simple as possible.

Figure 3.21 on the next page shows a UML class diagram drafting such an approach. The abstract super class `I2CAppliance` contains the method `request_state()` that is responsible for sending device state updates to the bridge. Its static factory method, `create()`, instantiates the correct object based on the supplied device type, and will simplify the initialization process.

Again, it is important to consider providing an appropriate layer of abstraction. Rather than forcing developers to work with the raw 24-bit state values of the bridge's output modules, each of the specialized classes represent their respective device's internal state through a data type that is more intuitive in comparison. Binary switches can only be turned on or off, so their state is represented by a single Boolean value. The states of Dimmers and RGB Dimmers are represented by one or three floats, respectively, ranging from 0.0 to 1.0, expressing the range of possible PWM frequencies from none to full. Finally, the five distinct states that can be assumed by a Shutter are expressed through an enumerated value.

For translating between these abstract values and the raw 24-bit states, each subclass requires an implementation of the virtual `decode_state()` and `encode_state()` methods that translate from and to the raw state values, respectively; they are used automatically by the superclass logic during communication with the bridge.

Regarding the device control interface; each class provides few methods with a straightforward naming scheme, making it easy to see at a glance the capabilities of each device type. The device state is publicly exposed in each class's `state` field, its data type depending on the device class as described above.

Additionally, the device objects will follow the observer pattern, receiving the ability to be subscribed to by the gateway software. The objects will then send a notification to the subscriber whenever its state was changed through a state update event.

### Representing Sensors

Representation of the sensors used in this solution is slightly different, as they are not meant to maintain a certain state. Instead, they only have to notify the system whenever they have received an input. The observer pattern will also be applied here to allow the gateway to be informed whenever an input event has been received.

Overall, the representation of sensors will be split into three distinct classes:

- The `I2CPassthroughSensor`, as its name states, directly passes through incoming input events, and is meant for the Button and Toggle modules. Two different notifications will be sent; one for the state changing to on, and another for changing to off.

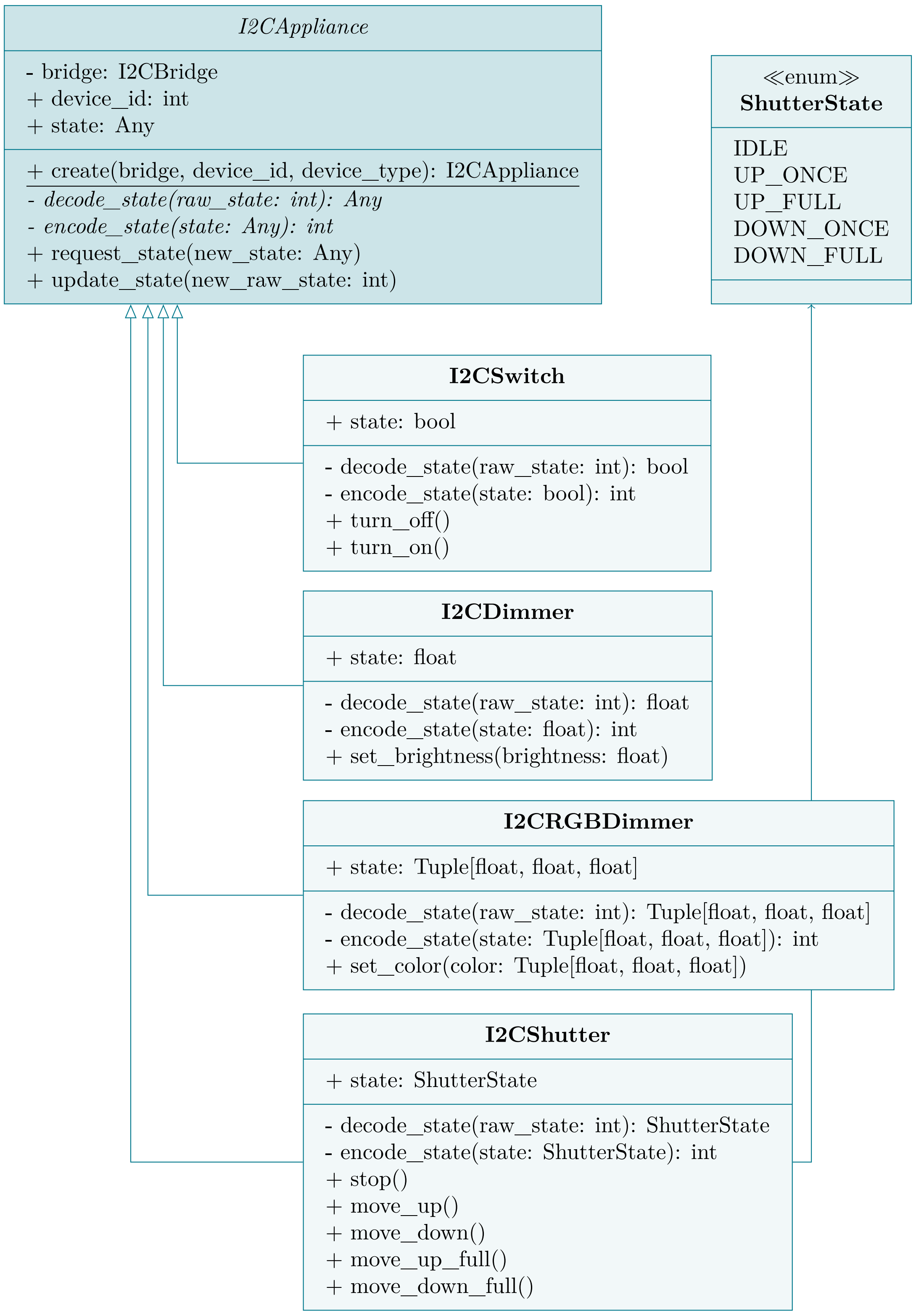

*I2CAppliance*
- bridge: I2CBridge
+ device_id: int
+ state: Any
+ create(bridge, device_id, device_type): I2CAppliance
- *decode_state(raw_state: int): Any*
- *encode_state(state: Any): int*
+ request_state(new_state: Any)
+ update_state(new_raw_state: int)
≪enum≫
**ShutterState**
IDLE
UP_ONCE
UP_FULL
DOWN_ONCE
DOWN_FULL
**I2CSwitch**
+ state: bool
- decode_state(raw_state: int): bool
- encode_state(state: bool): int
+ turn_off()
+ turn_on()
**I2CDimmer**
+ state: float
- decode_state(raw_state: int): float
- encode_state(state: float): int
+ set_brightness(brightness: float)
**I2CRGBDimmer**
+ state: Tuple[float, float, float]
- decode_state(raw_state: int): Tuple[float, float, float]
- encode_state(state: Tuple[float, float, float]): int
+ set_color(color: Tuple[float, float, float])
**I2CShutter**
+ state: ShutterState
- decode_state(raw_state: int): ShutterState
- encode_state(state: ShutterState): int
+ stop()
+ move_up()
+ move_down()
+ move_up_full()
+ move_down_full()

**Figure 3.21:** UML for class representations of appliances in the Bridge API.

- The `I2CCycleButtonSensor` for the two Cycle modules will provide a notification without any data, merely signaling that the button has been pushed.
- The `I2CShutterControlSensor` will contain four different notifications, depending on which button – up or down – was pressed and for how long – long or short.

### Representing the Bus and Protocol

The `I2CBus` class is responsible for abstracting from the communication protocol. It provides methods for each of the protocol's commands and returns appropriate response objects.

The class internally represents those by `I2CCommand` and `I2CResponse` objects. One exists for each possible command and response, containing the appropriate data.

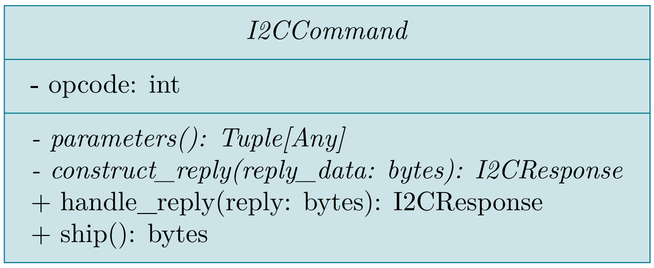

*I2CCommand*
- opcode: int
*- parameters(): Tuple[Any]*
*- construct_reply(reply_data: bytes): I2CResponse*
+ handle_reply(reply: bytes): I2CResponse
+ ship(): bytes

**Figure 3.22:** UML for class representations of protocol commands. A subclass will be created for every command.

When a command is to be transmitted, its corresponding object first gets transformed into raw bytes via its `ship()` method. The bytes then get supplied to one of the `raw_read()` and `raw_write()` methods. Realization of those methods is deliberately deferred to subclasses, allowing for different I$^2$C implementations to be used. The API is designed with two of those; the `I2CBusReal` class uses the `smbus2` package and is intended for production, while the `I2CBusDummy` implementation merely simulates communication with a real bridge, meant for testing purposes.

A class diagram for the Bus layer is presented in Figure 3.23 on the facing page.

### 3.5.4 Creating a Home Assistant Integration

The final step towards integrating support for our bridge into Home Assistant is to write an extension, or, as it is called here, an *Integration* (or sometimes also

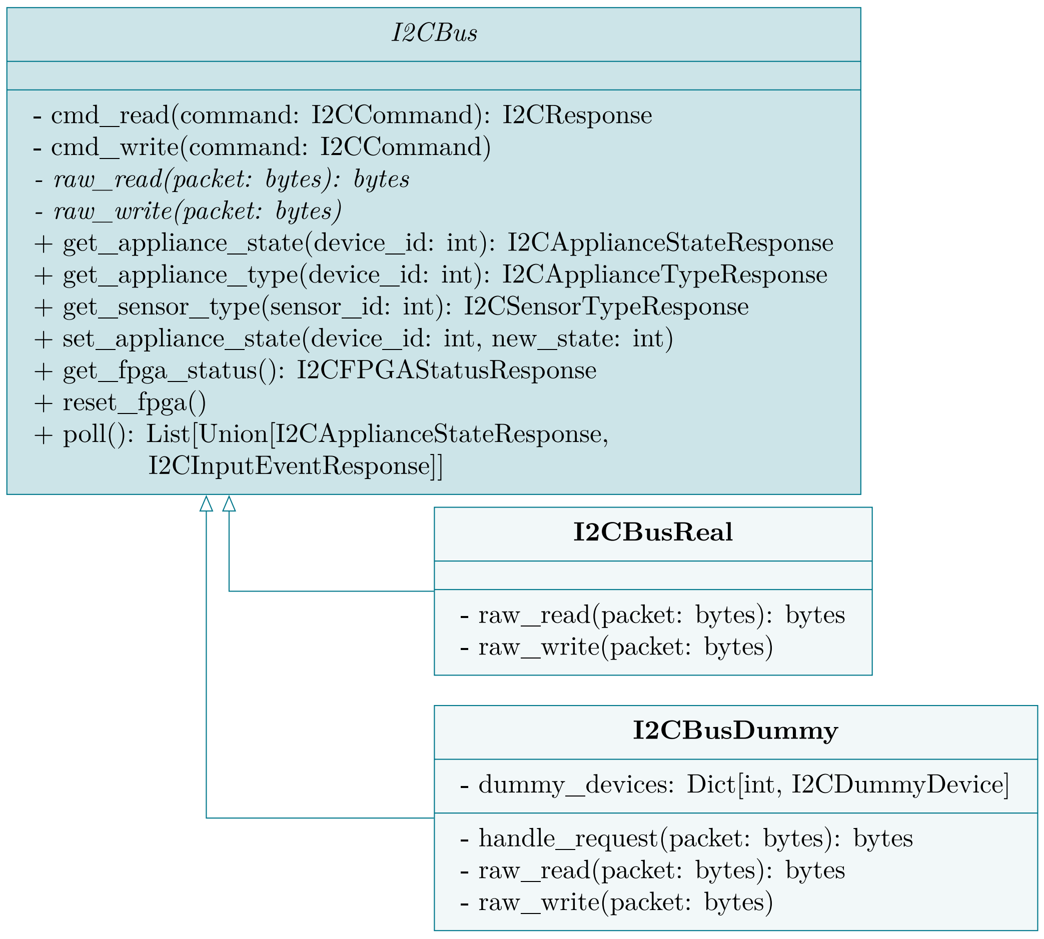

I2CBus
- cmd_read(command: I2CCommand): I2CResponse
- cmd_write(command: I2CCommand)
- raw_read(packet: bytes): bytes
- raw_write(packet: bytes)
+ get_appliance_state(device_id: int): I2CApplianceStateResponse
+ get_appliance_type(device_id: int): I2CApplianceTypeResponse
+ get_sensor_type(sensor_id: int): I2CSensorTypeResponse
+ set_appliance_state(device_id: int, new_state: int)
+ get_fpga_status(): I2CFPGAStatusResponse
+ reset_fpga()
+ poll(): List[Union[I2CApplianceStateResponse, I2CInputEventResponse]]
I2CBusReal
- raw_read(packet: bytes): bytes
- raw_write(packet: bytes)
I2CBusDummy
- dummy_devices: Dict[int, I2CDummyDevice]
- handle_request(packet: bytes): bytes
- raw_read(packet: bytes): bytes
- raw_write(packet: bytes)

**Figure 3.23:** UML for class representations of the I²C bus.

*Component*), for it. This integration will make use of the API written in the previous section to access the bridge and its devices and provides them to the gateway software, so that they can then be managed and controlled via its user interface.

To be able to write such an extension, it is first important to understand the **internal architecture** of the Home Assistant software.

### A Brief Overview of Home Assistant's Architecture

Pictured in Figure 3.24 on the next page is an overview of the major parts that comprise the Home Assistant internal architecture [Hom20a].

Device control is presented to the user by means of either the user interface or automation configuration; both being frontends for the system core.

Depending on the type of device that needs to be controlled, the core then calls on an appropriate internal component. Home Assistant includes a component for each general type of device it supports (i.e. lights, switches, blinds and so on), each containing specific functionality for the appropriate device class. It provides several objects called *entities* which represent each device type's capabilities. For example, the Light component contains all code necessary for managing a lamp from the user interface as well as a generic Light entity with virtual methods for turning on and off a lamp, controlling its brightness or color.

Support for the management of specific vendor devices is the responsibility of what Home Assistant calls a *platform*, which hierarchically lies between it and the API containing the vendor-specific code. It serves as another layer of abstraction towards the more general functionality contained within the device component.

This is where our integration is going to come into play.

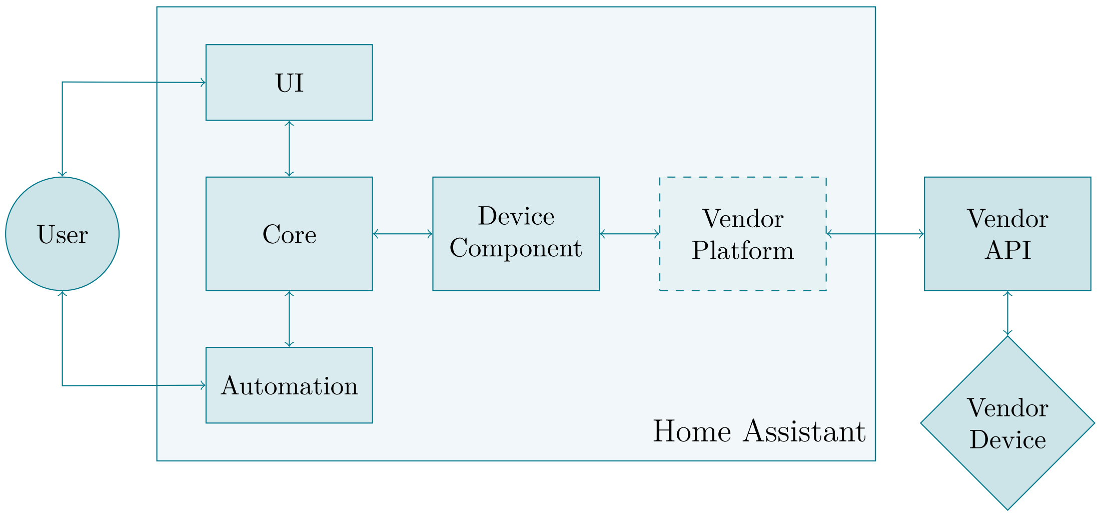


**Figure 3.24:** An overview of Home Assistant's internal architecture and communication flow for controlling one particular device. The FPGA bridge integration will take the part of the vendor platform, communicating with the API to provide specific control for the bridge hardware.

## Extending the Architecture

To integrate support for our bridge, we need to design an integration that includes a platform for each type of device we wish to have support for from inside Home Assistant. Specifically, that means we have to create platforms for lamps, dimmable lambs, color lamps and shutters as well as buttons and switches.

The main work needed in creating a platform consists mainly of extending a device component's generic entity classes to provide Home Assistant with an object serving as a representation of a real device. At the time of writing, the software only provides 15 different entities, but each one is generic enough that this selection covers the most common types of appliances [Hom20c]. For example, the Light entity can be extended in different ways to include support for all three types of lamp modules designed for our bridge hardware.

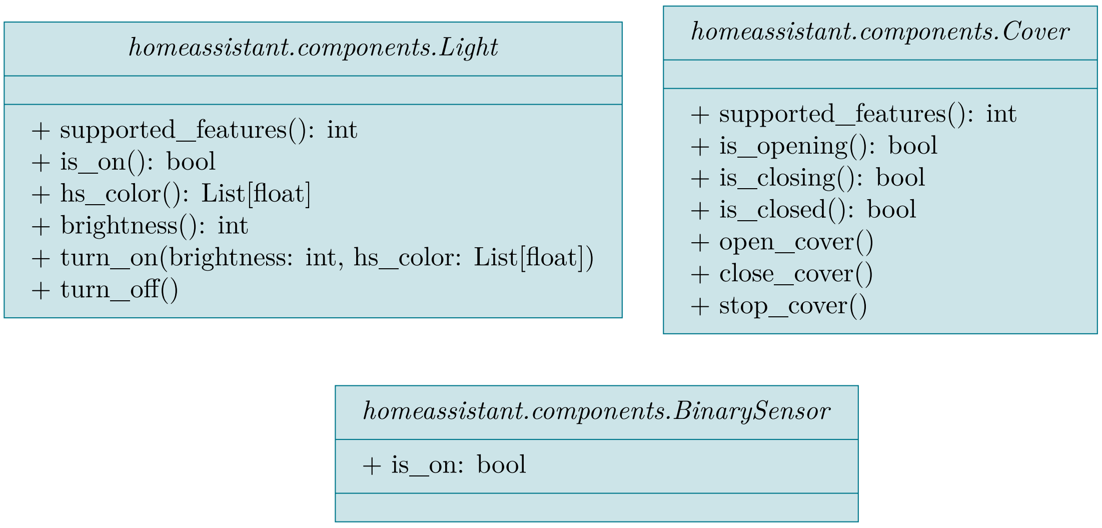

*homeassistant.components.Light*
+ supported_features(): int
+ is_on(): bool
+ hs_color(): List[float]
+ brightness(): int
+ turn_on(brightness: int, hs_color: List[float])
+ turn_off()
*homeassistant.components.Cover*
+ supported_features(): int
+ is_opening(): bool
+ is_closing(): bool
+ is_closed(): bool
+ open_cover()
+ close_cover()
+ stop_cover()
*homeassistant.components.BinarySensor*
+ is_on: bool

**Figure 3.25:** UML for Home Assistant's base entities with all methods and fields relevant for the bridge component. The capabilities of each subclass depend on the methods that it chooses to implement as well as the bit field supplied by the `supported_features()` method.

By inheriting from these base entities, we create four distinct classes, each a representation of the types of appliances our bridge supports – the `FPGABinary`, `FPGADimmer` and `FPGARGBDimmer` classes each inherit from the **Light** entity and implement its methods according to their capabilities. A `FPGAShutter` class based on the **Cover** entity will represent the Shutter module.

Sensors will not receive one entity per device, but per signal instead. A single `FPGABinarySensor` based on the **Binary Sensor** platform will be created for each Button, Dimmer and Cycle module, as they can each only fire one signal. A Shutter module will be represented by *four* of those objects – one for each possible notification fired by the API object, i.e. long press down, long press up, short press down and short press up.

## 3.6 Designing a Management UI

The bridge hardware has been designed in such a way that changing the device configuration requires reprogramming of the MachXO2, a task that usually encompasses manually rewriting certain parts of the hardware description logic and then uploading the resulting code onto the FPGA. The target audience for this project primarily includes electricians and hobbyists, i.e. people who are not expected to be able to speak a relatively esoteric computer language such as Verilog.

To increase the accessibility of this solution, the final piece of this project therefore consists of a graphical interface intended to make device configuration easier for engineers and end-users. This tool will be designed in such a way that new device configurations can be created without having to write a single line of Verilog code, as it produces all of it on its own. The user then merely has to supply the generated code to the Verilog compiler and programmer to upload it to the MachXO2.

### 3.6.1 Outlining the Use Cases

Performing a use case analysis is commonly used to identify the required features of a software project. Since this application has only a single purpose and is thus going to be relatively simple, drafting a short list of planned features should be appropriate enough here.

Considering this, the management tool will be designed with the following use cases in mind:

- The user can manage a device configuration.
  - The user can add new appliances and sensors to the configuration and give them recognizable names.
  - The user can manage appliances, including changing their name, device ID, assigned pins and device parameters.
  - The user can manage sensors, including changing their name, device ID, assigned pins, assigned appliance and device parameters.
  - The user can configure each pin to be active low instead of high.
- The user can generate Verilog code from the current configuration.
- The user can save the current configuration.
- The user can load a saved configuration.

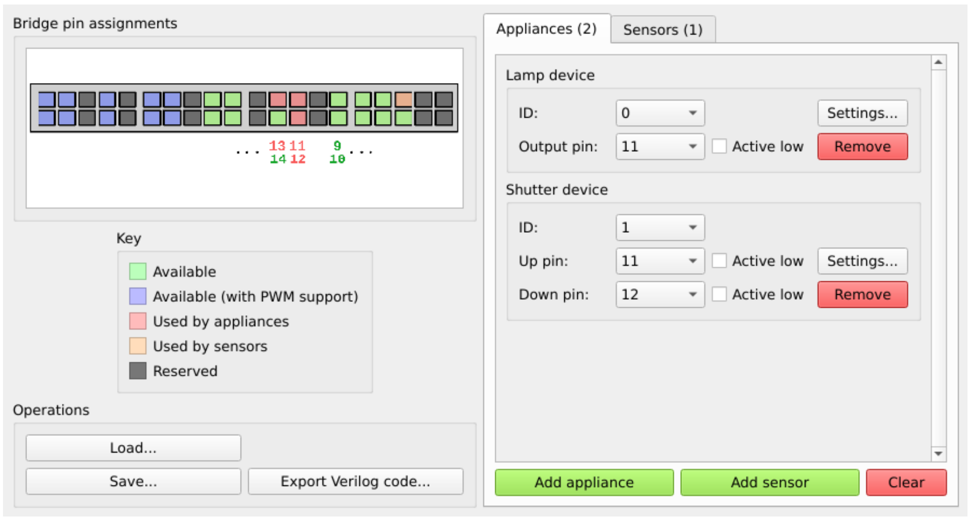
## 3.6.2 Designing the User Interface

Next, the tool's main dialog window will be designed.

Going into all the intricate details of proper dialog design should not be the focus of this work, and as such this topic will be kept relatively short, but it should be stated that the work is loosely based on the seven principles of dialog design laid out in the ISO 9421 standard [ISO06].

Based on those principles and the list of use cases, a mockup will be designed that can be used as a point of reference for establishing the working dialog proper in Section 4.4 (pg. 73). Such a mockup is presented in Figure 3.26. It was designed so that most of the features are recognizable immediately instead of being hidden behind obscure menus, and focus is placed on the most important buttons via color highlights. To support the user in visualizing the configuration result, a preview of the bridge's pins is presented on the left side of the dialog that closely resembles the hardware's pin banks while providing clear information about which pins are still available for assignment. Should a bridge pin not support PWM output, it will be marked as such.

Additionally, to prevent users from creating invalid configurations, the selection widgets for device IDs as well as pin and appliance assignments only show valid selections at all times. To help identifying mistakes, devices with erroneous settings are highlighted, as can be seen in Figure 3.27 on the following page.


Bridge pin assignments

... 13 11 9 ...
14 12 10

Key

🟩 Available
🟦 Available (with PWM support)
🟥 Used by appliances
🟧 Used by sensors
⬛ Reserved

Operations

Load...
Save... | Export Verilog code...

Appliances (2) | Sensors (1)

Lamp device

| | | | |
|---|---|---|---|
| ID: | 0 | | Settings... |
| Output pin: | 11 | ☐ Active low | Remove |

Shutter device

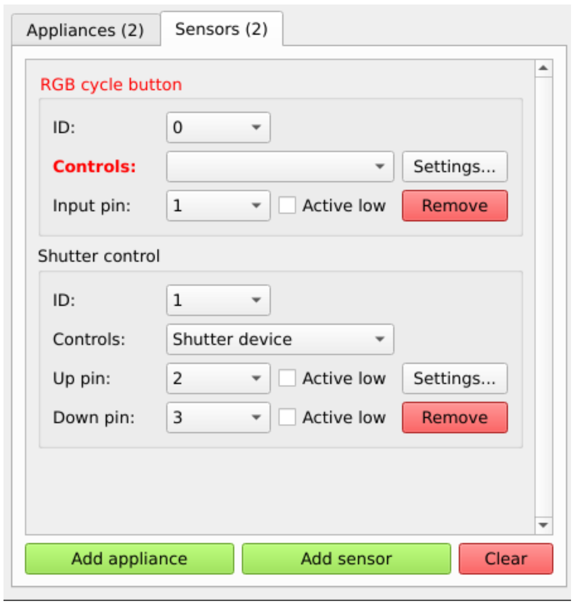
| | | | |
|---|---|---|---|
| ID: | 1 | | |
| Up pin: | 11 | ☐ Active low | Settings... |
| Down pin: | 12 | ☐ Active low | Remove |

Add appliance | Add sensor | Clear

**Figure 3.26:** Mockup of the Management UI's main dialog. A clearly labeled preview visualizes the pin assignment. The device settings are clear and simple. Important buttons are highlighted with green and red.

| **Principle** | **Effect on design** |
|---|---|
| Task suitability | The main dialog will be kept simple, and all features will be recognizable at a glance. |
| Self-descriptiveness | All components will be labeled clearly and outfitted with tool tips. Important widgets will be highlighted. Devices with invalid configurations will be marked. |
| Controllability | The application will not present the user with unexpected prompts. |
| Expectation conformity | Familiar widgets will be used, and a preview will keep the user informed about the expected result. |
| Error tolerance | The user will be asked to confirm irreversible actions and given a chance to rectify mistakes. No invalid configuration can be produced. |
| Individualization suitability | The user will be able to give recognizable names to devices and adjust device IDs freely. |
| Learning suitability | Not applicable, since the tool is simple enough that it doesn't require customization facilities. |

**Table 3.7:** Effects of the seven principles of dialog design on the device management UI.



RGB cycle button

ID: 0
**Controls:**
Input pin: 1 ☐ Active low

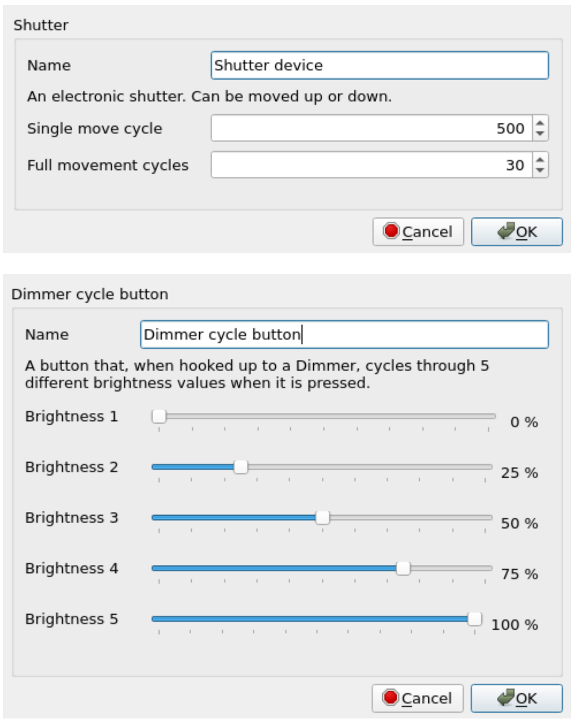
Shutter control

ID: 1
Controls: Shutter device
Up pin: 2 ☐ Active low
Down pin: 3 ☐ Active low

**Figure 3.27:** Mockup of the Sensors tab, displaying appliance assignment options and highlighting of invalid configurations.

Loading, saving and generating code is accomplished through three simple buttons, and each of them invokes the operating system's standard file or directory selection dialog.

Other than the main window, the application will contain only one more dialog; meant for configuring a device's name and its module parameters, the latter of which are adjusted via appropriate, self-describing widgets. Figure 3.28 shows a mockup of this dialog.


Shutter

Name: Shutter device

An electronic shutter. Can be moved up or down.

Single move cycle: 500

Full movement cycles: 30

Dimmer cycle button

Name: Dimmer cycle button

A button that, when hooked up to a Dimmer, cycles through 5 different brightness values when it is pressed.

Brightness 1: 0 %

Brightness 2: 25 %

Brightness 3: 50 %

Brightness 4: 75 %

Brightness 5: 100 %

**Figure 3.28:** Mockups of the device settings dialog for output and input modules respectively, containing name and module parameter settings. Parameters are adjusted through appropriate widgets fitting the type of value.

# 4 Implementation of the Design

Designing the smart home solution in great detail was a rather extensive process, but it allows us to keep this chapter's notes on the implementation of each component – *relatively* – short.

We start work on the design's realization by first **implementing the bridge logic**, during which we will touch upon using Verilog as a hardware description language and some of the nuances of hardware logic design in general.

Afterwards, it is time to work on the gateway by first **creating the Bridge API** and, following shortly after, the **Home Assistant integration** that uses it. Topics worth mentioning here include the setup of a suitable environment and the development of the individual integration components.

Work on the solution concludes by creating the **management UI**, which will deal with using Qt as a UI toolkit and the generation of Verilog code through a template system.

## 4.1 Implementing the Bridge Hardware Logic

Designing the bridge's logic appeared as the most comprehensive task of the previous chapter, and its realization also takes a significant amount of effort.

We are going to briefly introduce the **Verilog** language used to describe and realize the hardware's behavior, followed by a few notes on the **module development** itself. Since the hardware brand prescribes the use of a certain **development suite**, it seems appropriate to briefly talk about it as well. We close this section by discussing **simulation** as a means of testing the behavior of the produced code.

### 4.1.1 Working with Verilog

*Verilog* is a *HDL* – a *hardware description language* – used for the abstract modeling of digital logic and behavior [Ver06].

Verilog possesses several facilities that are analogous to elements more commonly found in programming languages in order to abstract from the concept of concrete logic gates and circuits that define the behavior of the hardware to be designed. Individual modules can be instantiated like objects, and can contain registers of different sizes that are able to be loaded with data, like variables. Conditional logic allows for branches and loops, and repeated calculations and routines can be summarized within functions and tasks, so that cleaner code can be produced.

This allows hardware behavior to be modeled in a relatively high-level manner, as it takes the responsibility of laying out concrete logic gates and wiring on a circuit board away from the designer, who can instead focus on describing the abstract behavior using a syntax that closely resembles a high-level programming language. This arguably makes hardware design a lot more accessible, especially to designers already experienced in software development.

#### Process of Design Realization

After the hardware logic has been described using the HDL, it has to be processed for use on the FPGA, much like source code has to be compiled into machine code to be executed as a program. This process consists of several individual steps [BV04], the most significant ones being:

**Synthesis.** The Verilog code gets compiled into a *netlist*, which is a compilation of circuit gates that are equivalent to the described hardware behavior. After generation, the netlist is first optimized and then mapped to the resources available within the FPGA.

**Place and route.** The logic blocks contained in the netlist get placed onto concrete physical locations on the FPGA, and the wiring between them is determined.

**Upload.** The final resulting logic data is uploaded to the flash memory of the FPGA.

With the aid of appropriate tools, all of these steps are mostly automated, so we can primarily focus on writing the Verilog code.

### 4.1.2 Developing the Bridge's Logic Modules

In general, the basic workflow when translating the modules designed in Section 3.3 (pg. 26) to Verilog code consists of defining the module's input and output ports, any module parameters and internal registers, if applicable, and finally the module's logic itself.

Instead of describing the whole process, we will only talk about some general aspects of the hardware development. Aside from all the modules we previously designed, there are a few additional ones needed to be created, most notably a **top module** housing the complete design. It is also important to mention in what ways the process of hardware programming **differs to software development**. Finally, the implementation of one particular component, the **CRC calculation**, also deserves some special mention.

### Creating a Top Module and Auxiliary Module Instances

Much like software applications have a class or function that serves as a main entry point, the bridge logic requires a certain module that contains all other modules we created, alongside the wiring that connects them as well as some auxiliary components. This module is generally called a *top module*.

Here, next to the components we previously designed, we are going to create an instance of the MachXO2's internal components that we require for this project. One of these is the *embedded I$^2$C function block*, which we previously talked about during the design of the I$^2$C Controller in Subsection 3.3.4 (pg. 38). It is adjoined to an instance of the device's Wishbone bus, which we can now connect to the I$^2$C Controller via the appropriate wires.

We also need to define an instance of the MachXO2's *internal oscillator*, which will serve as the device's global clock source. Its frequency can be configured from a range of different values, ranging between 2.08 and 133 MHz [Lat17].

The default value of 2.08 MHz is sufficient for our bridge. The only timing constraint we need to fulfill is that the bridge be fast enough to accommodate for I$^2$C transmissions, which the Raspberry Pi, by default, clocks at a frequency of 100 kHz [Bro12]. According to the MachXO2's documentation [Lat16], the clock driving the Wishbone bus needs to be at least seven times as fast as the I$^2$C clock during a transmission, which the default frequency provides easily enough.

### Differences to Software Development

A big fundamental difference to software design is that, since the hardware is not run by a CPU, it at first has no concept of sequentialism at all. When defining a block of code in Verilog, all of the register assignments contained within will be executed at the same time. However, some of the hardware behavior that we designed previously does require to be performed in a certain sequence; especially initialization routines and I$^2$C communication.

To enable a module performing tasks in a certain order, it can be outfitted with a state machine. Driven by the device's global clock, it will define the module's behavior at each cycle based on the value of an internal state register, and it can modify this value based on the operation that is to be performed next in sequence. Conditional branches can be used to adjust the behavior further based on the current module inputs or internal values and counters. In Verilog, this is realized through a switch-case-like statement.

The most prominent state machine in the bridge hardware logic is featured in the design of the I$^2$C Controller, as it requires to repeatedly perform read and write operations on the MachXO2's internal Wishbone bus. This process involves sending a certain sequence of input signals while also waiting for a response signal from the bus intermediately. This effectively makes each of those bus operations a small state machine in and of itself, causing the resulting outer state machine to become quite complicated; however, through the use of Verilog pre-compiler macros, the module's code can fortunately be kept somewhat readable.

### Calculating CRC in Hardware

The general algorithm for computing the CRC error checking data, as described in Subsection 3.4.5 (pg. 44), basically amounts to a series of polynomial divisions. Calculating the remainder of a polynomial division can be achieved through a series of bit shifting and XOR operations. As a result, it is very easily implemented in hardware. The message polynomial is fed into a $n$-bit shift register, one bit per cycle. If this causes the most significant bit of the register to be set, the whole register is then XORed against the generator polynomial, which effectively subtracts it from the current intermediate result, just like it would be done in polynomial long division. Once the last message bit has been shifted in, the register holds the CRC check value, which can then be appended to the message when sending, or checked against zero when receiving.

However, since messages are transmitted byte-per-byte instead of bit-by-bit, this approach would require eight cycles for each byte to compute. This may not be fast enough for the MachXO2's embedded I$^2$C function block, which may require the next byte to be ready within seven clock cycles [Lat16].

The CRC calculation can be optimized by employing a lookup table, where part of the intermediate result is used to lookup the resulting CRC check value, as described in [Sar88]. This reduces the time required to process one byte to a single clock cycle. A lookup table for eight bits of a CRC-16 calculation contains 256 entries of 16 bits, or 512 bytes in total. Because of its small size, it can be easily generated by software and then statically included in the hardware ROM.

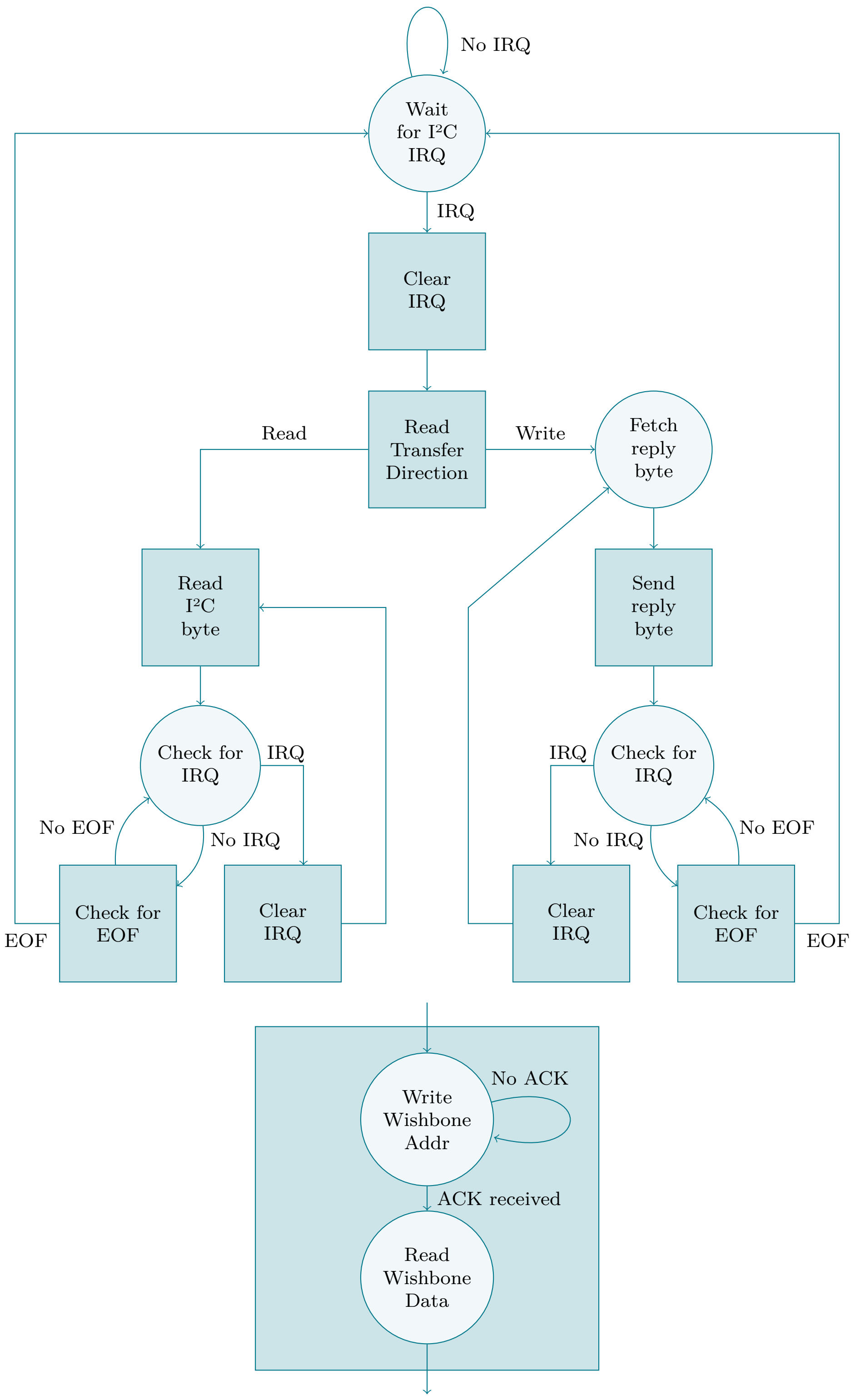


**Figure 4.1:** Excerpt from the I²C Controller's internal state machine. Each rectangle state is a Wishbone read or write operation that, in itself, consists of sub-states. Pictured on the bottom is a generic Wishbone read operation.

### 4.1.3 Working with Lattice Diamond

The FPGA that is used in this project, the MachXO2-7000, requires use of *Lattice Diamond*[1], a proprietary, full-fledged FPGA development suite. It requires a license to be used, which is available for free after registration with a valid e-mail address on the Lattice website.

Lattice Diamond features all the tools necessary for the entire development process, as it comes with an IDE, synthesis and place and route tools as well as a programmer for the MachXO2.

#### Choosing a Synthesis Tool

The Lattice Diamond IDE ships with two different synthesis tools: the default *Lattice Synthesis Engine* as well as *Synplify Pro*[2], a synthesis tool from another development suite of the same name, created by Synplify.

During development of the bridge logic, in particular while implementing the I$^2$C Controller module, it came to attention that the Lattice synthesis tool appeared to produce incomprehensible bugs, observed through strange behavior of the hardware after synthesis that was irreproducible in the simulation of the same code, which had produced the expected results. This phenomenon caused a significant delay in development, as it happened without any apparent reason, but the problem was eventually able to be narrowed down to the choice of synthesis tool, since after switching to Synplify Pro, the developed logic started to work as expected.

Research on this problem unfortunately brought up no further information regarding this issue. It is suspected that the problem lies with the tool's internal translation of state machines described within the logic, since during testing, the MachXO2 displayed several values for states that had not even been defined in the code, but this is merely conjecture. In any case, the Synplify synthesis tool has produced expected results more consistently.

### 4.1.4 Verifying Functionality through Simulation

A big hurdle in designing the hardware logic, especially when comparing the process to software development, is caused by the MachXO2 not providing any interface

[1] `http://www.latticesemi.com/latticediamond`

[2] `https://www.synopsys.com/implementation-and-signoff/fpga-based-design/synplify-pro.html`

like output devices or serial consoles[3] for debugging the coded hardware behavior, so verifying that any logic behaves as expected is actually not quite trivial. The FPGA does contain eight on-board LEDs that can be controlled from the digital logic, which at least allows for some limited observation of its internal state. For example, this proved to be very useful in displaying the current state of the I$^2$C Controller's state machine. However, this method of verification is obviously quite limited.

One way to test the Verilog code is through the use of a simulator, which effectively emulates the module logic and logs its output signals for a determined number of cycles. To provide the module with various input signals, it can be driven by what is called a *test bench*, which in itself is another Verilog module containing an environment for testing the logic.

A test bench mainly consists of an instance of the module to be tested, registers and wires that can be linked to the module's input connections as well as a block containing register assignment operations, through which the module's inputs can be manipulated.

The resulting output can then be visualized through a *logic analyzer*, allowing one to inspect the module's simulated output signals. This information is quite helpful to determine whether the logic works as intended.

While the functionality of individual modules can be verified through this method well enough, it becomes increasingly difficult to test the behavior of interconnected modules or even the whole system. This is hindered even more through the fact that some of the designed logic depend on internal components of the MachXO2, like the Wishbone bus or the embedded I$^2$C function block, which cannot be simulated as they are essentially black boxes. The only point of reference here are the components' respective documentation, from which their predicted output signals can be inferred to aid in the simulation.

Another point of difficulty is present in a discrepancy existing between simulated and real hardware behavior. For example, the simulation tool that was used for this project, *Icarus Verilog*[4], supports initializing registers to certain pre-loaded values. This is not the case with the FPGA synthesis tools, which ignore the corresponding statements, so every register is initialized to zero. Some of these intricacies are unfortunately not clearly documented anywhere, and thus overall a lot of guesswork was involved in the development of the hardware logic.

[3]Of course, such an interface could be designed by oneself, but it would have to be verified as well before being of any meaningful use.

[4]`http://iverilog.icarus.com/`

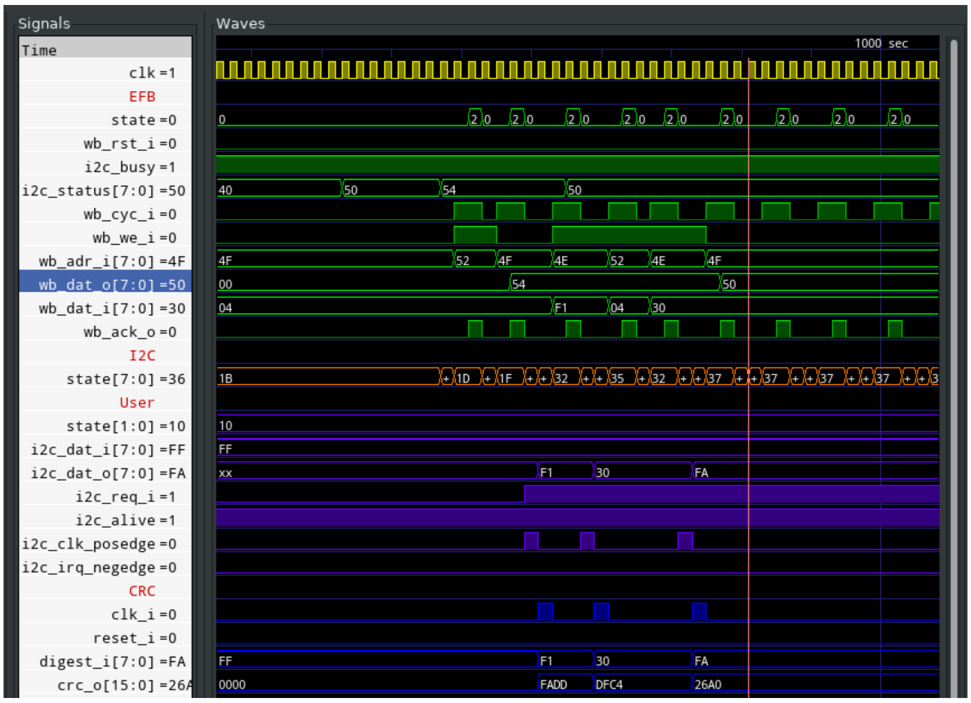


**Figure 4.2:** Using a logic analyzer to aid in verifying the intended behavior of I²C Controller and Device Controller, with the help of a simulated Wishbone bus. Pictured is the Device Controller reacting to a CRC checksum failure from an incoming I²C message by responding with `F1 30 FA`.

## 4.2 Developing the Bridge API using Python

The design for the Bridge API, created in Subsection 3.5.3 (pg. 48), was implemented in a rather straightforward manner. We shall only discuss two brief topics of importance, which is **testing** and **deployment** of the resulting code.

### 4.2.1 Testing the API

Ensuring that the functionality of the API works as intended was done by help of four different tools that worked together.

#### Dummy I²C Implementation

As was mentioned during the creation of the API design, the class representing the I²C Bus can make use of different bus implementations. Aside from the proper I²C bindings, a dummy implementation was created that simulates a real I$^2$C bus and bridge, together with some dummy appliances connected to it.

This helped the testing of many of the API's components, especially regarding I$^2$C communication; as the I$^2$C hardware bindings are only properly usable on systems with real I$^2$C hardware support. Since the API was mainly developed on a standard desktop computer which lacked such functionality, the dummy implementation served as a useful substitute.

### Unit Tests

Using the dummy implementation, a few test cases have been written that automatically test basic functionality like successful connection to the I$^2$C bus, reading device types and valid states for certain device types. These could be quickly executed after every iteration of the development to make sure the API still functioned.

### Debugging Console

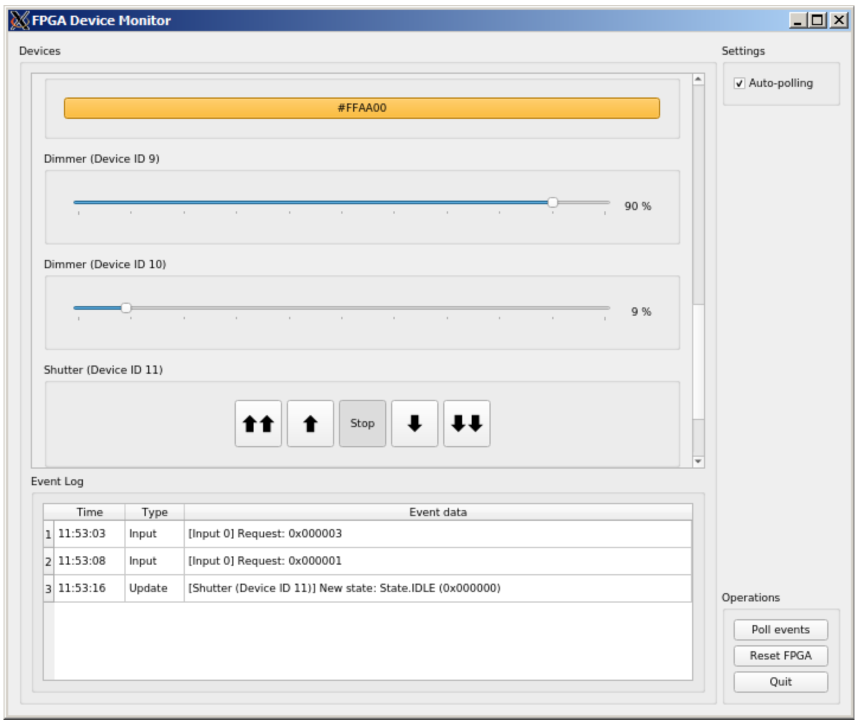
The API has been supplemented with a debugging console that can be run by executing the Python API module via command-line. It can be used to send I$^2$C messages to the bridge by manually inputting the bytes, and the tool will show the retrieved response. CRC data will optionally be calculated automatically from the input bytes.

The console served as an invaluable tool during debugging of I$^2$C connectivity and CRC verification.

### Device Monitor

An additional Python module, the *device monitor*, was created alongside the API. It serves as an example implementation of a GUI tool making use of the API to list and manage the bridge's devices, and therefore appears as a very simplistic gateway software.

It helped to ensure that the API functioned as intended, before including it in the much more complex Home Assistant. Figure 4.3 on the following page shows the Device Monitor running on the gateway.



✓ Auto-polling

| | Time | Type | Event data |
|---|---|---|---|
| 1 | 11:53:03 | Input | [Input 0] Request: 0x000003 |
| 2 | 11:53:08 | Input | [Input 0] Request: 0x000001 |
| 3 | 11:53:16 | Update | [Shutter (Device ID 11)] New state: State.IDLE (0x000000) |

**Figure 4.3:** Device monitor used for verifying the API. It also serves as an example for a graphical application that makes use of the API. By running the monitor on the gateway, the bridge's appliances can be controlled.

### 4.2.2 Creating a PyPI Package

*PyPI*[5], the *Python Package Index*, is a public, community-driven repository of Python packages and serves as one of the primary distribution methods of Python projects. It defines a standard for preparing redistributable software packages, which can be installed into a Python environment through a package manager, which also automatically installs all necessary dependencies. This offers a slightly more user-friendly approach to distributing code than installing from source or a repository.

Furthermore, this is also how the Home Assistant documentation prescribes how third-party code should be added to the software [Hom20b]. As such, the API was packaged in a PyPI-compatible format. Even though it has not been made public at the time of writing, the created software package can already be used for local deployment, and installed to a Home Assistant environment via command-line.

The process of packaging involves following a certain directory structure as well as the creation of a `setup.py` file containing metadata about the package, as de-

[5]`http://pypi.org`

scribed in the official documentation [Pyt20]. Based on this file, the redistributable packages are automatically created through supplied tools.

## 4.3 Developing the Home Assistant Integration

After the Bridge API is finished and working, it is now time to manifest the final part of the gateway by including support for our bridge with a Home Assistant extension.

Work on the integration begins by **setting up a development environment** for Home Assistant, after which we introduce our bridge to the system by **connecting to our Bridge API**. Lastly, the different **device platforms** have to be realized to allow representations of our appliances and sensors.

### 4.3.1 Setting Up a Development Environment

In order to start working on a Home Assistant extension, it is first necessary to set up a proper environment which contains all of the necessary tools and libraries. As per the documentation [Hom20d], this task merely involves cloning the Home Assistant source repository and installing a few dependencies.

The Bridge API package that was created during the last section is also installed to this environment, so that it can be used by the extension.

#### Creating an Integration

The Home Assistant development environment fortunately supplies several tools that aid in the creation of a new integration. In particular, the *scaffold* script creates a basic skeleton integration from a template with the correct format, serving as a convenient starting point.

### 4.3.2 Integrating the API

In Home Assistant, when an integration is initialized, a special setup method is called. Here, it gets the opportunity to perform initialization and to forward setup calls to its device platforms.

This is where we can integrate our Bridge API. Upon instantiation, it will automatically connect to the bridge and scan it for its devices, which it will store for when the platforms initialize momentarily.

Much like the bridge's appliances will be represented by entity objects, we also create and register an entity for our bridge itself, so that it will show up on Home Assistant's list of connected devices later.

#### Enabling Customization through Config Flow

The API can be configured to adjust the I$^2$C connection parameters, notably slave address and bus identifier. It can also be set to use the dummy I$^2$C implementation mentioned earlier, providing Home Assistant with simulated devices. However, we first need a way for the end user to configure the API through Home Assistant.

While it is possible to manually configure an integration via a text file, this approach is not particularly user friendly. Instead, Home Assistant provides a standardized way to allow integrations to be configured through its graphical interface, a process it calls *Config Flow*. To add support for this, the integration has to conform to this standard by defining a configuration schema, which is an abstract description of all its configurable settings and their allowed values. (Home Assistant uses `voloptuous`[6], a data validation library, for this purpose.)

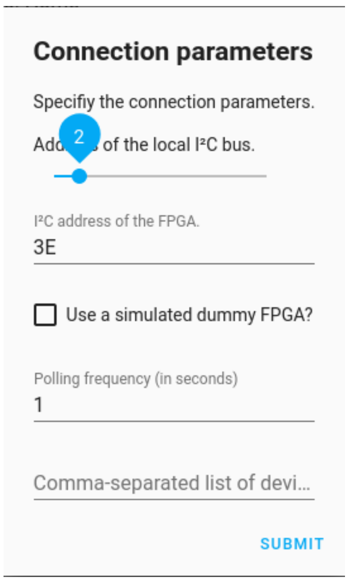
All integrations supporting Config Flow are listed in Home Assistant's menu for adding a new integration to the configuration. When a user selects one of them, its configuration schema is parsed by the software, generating a standardized dialog containing all of the options defined in the schema. Figure 4.4 on the next page shows an example of this. After the user confirms his settings, the configuration routine automatically verifies the input values per the rules set in the schema and, if valid, supplies them to the aforementioned setup method of the integration, so that it can finally pass these values to the API.

The bridge integration received support for a Config Flow setup, as it is a relatively effortless process. A schema was created containing the API parameters, together with a routine that performs some additional sanity checking on the user-supplied configuration values.

### 4.3.3 Setting up the Device Platforms

The three supported platforms – *Light*, *Cover* and *Binary Sensor* – have been created according to our designs in Subsection 3.5.4 (pg. 52). During setup of the

[6]`https://github.com/alecthomas/voluptuous`



Connection parameters

Specifiy the connection parameters.

Address of the local I²C bus.

2

I²C address of the FPGA.

3E

☐ Use a simulated dummy FPGA?

Polling frequency (in seconds)

1

Comma-separated list of devi...

**Figure 4.4:** Configuration dialog automatically generated for the bridge integration through the use of Config Flow.

integration, each platform also gets initialized via a call to its own setup method. Here, it looks up the list of bridge devices from the API, and creates their entity representations accordingly.

Alongside the methods listed in our designs, each entity also got outfitted with some meta information including a reference to the bridge entity. This is used to link the two entities together, which is required for Home Assistant to display a list of all devices connected to the bridge inside its management menu.

## 4.4 Developing the Management UI

Lastly, the device configuration management UI tool will be completed. As will be explained shortly, by creating the mockup dialogs in Section 3.6 (pg. 56), we already have completed a significant amount of work towards the application, since the **use of Qt as a toolkit** greatly simplifies the dialog implementation step. Other than implementing the necessary logic, we still have to **devise file formats** for save and data files and device data and choose an efficient way to **generate Verilog code** from a user-made device configuration.

### 4.4.1 Using Qt as a UI Toolkit

There are several toolkits available that help in rapid development of cross-platform user interfaces. Among the more popular choices with support for Python are GTK[7], TkInter[8] and Qt[9]. The three projects are very similar to each other, as they all base the creation of layouts around the arrangement of UI widgets and offer roughly the same functionality. *Qt* was chosen for this project due to personal experience.

Creating window layouts using Qt can be done in several ways. The layout can either be described purely via code, or a graphical design tool can be used, in a WYSIWYG fashion, to create and save layouts into XML files that the Qt engine can then use to generate the dialog at application runtime. This latter approach allows for rapid and effective UI design, especially since the canonical tool *Qt Designer* shipped with the toolkit itself is efficient and straightforward to use.

This also makes it an excellent tool for prototyping user interfaces – in fact, the mockups showed in Section 3.6 (pg. 56) were designed in this way. This has an added bonus of allowing the drafted prototypes to be translated to the final, usable dialogs with relatively few adjustments.

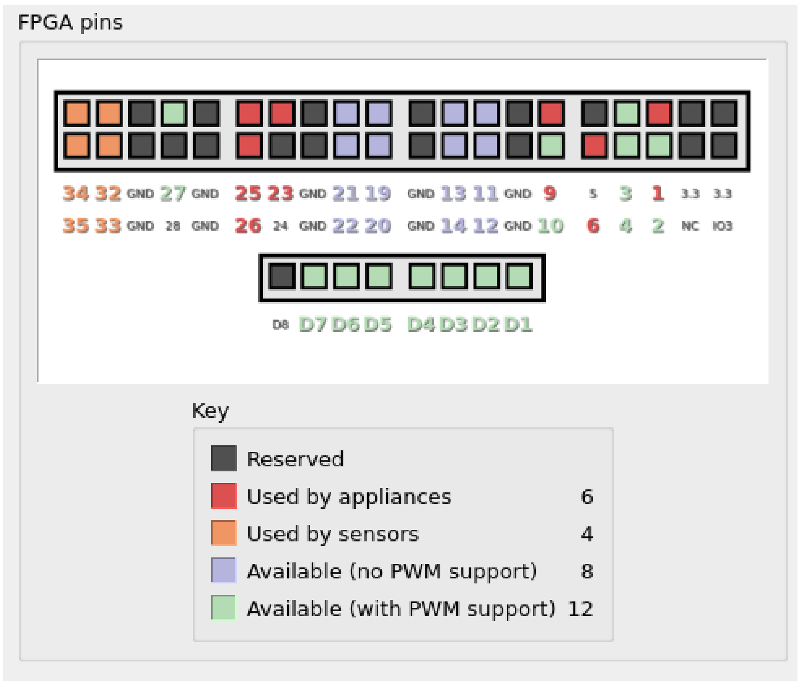
After creating all necessary window layouts, the resulting XML files can be loaded and inflated from within the Python code, which then merely has the responsibility to fill the widgets with the appropriate data, and to update their states according to user interaction.

#### Drawing the Preview Image

Using Qt has another benefit, as the toolkit also comes with image drawing functionality. The management UI can make use of this to easily create the preview image showing the current pin assignments, without the need to resort to external drawing libraries.

Although the image is generated dynamically, the information containing the layout of the pins remains hard-coded. It was considered to develop a more general structure for pin layouts to allow for easier adjusting to different devices other than the MachXO2, but this was ultimately decided against, seeing as this project only serves as a proof-of-concept for the time being.

[7]`http://www.gtk.org`
[8]`https://wiki.python.org/moin/TkInter`
[9]`http://www.qt.io`


FPGA pins
34 32 GND 27 GND 25 23 GND 21 19 GND 13 11 GND 9 5 3 1 3.3 3.3
35 33 GND 28 GND 26 24 GND 22 20 GND 14 12 GND 10 6 4 2 NC IO3
D8 D7 D6 D5 D4 D3 D2 D1
Key

| | | |
|---|---|---|
| ■ | Reserved | |
| 🟥 | Used by appliances | 6 |
| 🟧 | Used by sensors | 4 |
| 🟪 | Available (no PWM support) | 8 |
| 🟩 | Available (with PWM support) | 12 |

**Figure 4.5:** Final version of the preview image, generated by Qt from the application's state.

### 4.4.2 Defining JSON Data Structures

The management UI needs to understand two types of external data: its **save files** and a **device type database**, so we need to devise a certain format for each.

#### Save Files

Since the application is going to support loading and saving of the current device configuration, a file format has to be defined for this data. JSON has been chosen for this purpose, as it is a format that is widely used, humanly readable and supported by Python out-of-the-box.

The data structure contains all information about a device configuration, and the application receives support for exporting and importing its current state into or from this format.

### Device Type Database

The manager also needs some type of record containing all of its supported output and input modules, including their names, descriptions and properties. Again, JSON appears as an appropriate choice. A database format like SQLite could also be considered, but since the amount of data rows is relatively small and there is no need for sophisticated queries, this solution seemed to be too elaborate for this purpose.

```
{
  "1": {
    "name": "Dimmer",
    "description": "A device whose state can be regulated via
                    pulse-width modulation (PWM).",
    "driver": "Out_PWM_Single",
    "pins": {
      "signal": {
        "active_low": false
      }
    },
    "parameters": {
      "delay": {
        "name": "Step delay",
        "description": "When updating the lamp's brightness,
                        the delay (in ms) between each step.",
        "type": "int",
        "default": 10,
        "min": 1
      }
    }
  }
}
```

**Listing 1:** Example for an output device type database file containing one type. Types are indexed by a unique ID number and require at least a name, driver and one pin.

Following the *Prototype pattern*, the application reads this database into several read-only template objects. When the user wishes to add a new appliance or sensor, new device objects will be created from those templates accordingly.

## 4.4.3 Generating Verilog Code through Templates

After the user has created a device configuration and the application has checked its validity, it will allow generation of the equivalent Verilog code for importing into the Lattice project.

It should be noted that, to keep this process simple and clean, this application is not generating a complete Verilog project. Instead, only the files containing information pertaining to the custom device configuration will be generated, and have to manually be put alongside the remaining, static Verilog files. They will then be automatically included through compiler directives at the appropriate locations.

Because the basic structure of the generated code remains the same across all configurations, the application uses a *template engine* to generate the code. Instead of writing an algorithm that produces each file line by line, templates are written instead; one for each file to be generated. During processing, a template will generally be kept as-is, except for special template tags that are evaluated to produce the final output.

Commonly, these tags include variables that can be filled in with the supplied values, but depending on the template engine, there is also support for basic conditional logic or even loops, making these templates very versatile. `moody-templates`[10] is a template engine for Python with such functionality.

Having written templates for the required files, generation of Verilog code becomes quite simple. The templates merely have to be supplied with the appropriate data, and are then filled in with the proper values.

Unfortunately, it is very hard to write clean, readable templates that also produce clean results at the same time. Adding to that, Lattice's Verilog parser seems to be particularly picky about its syntax, even failing to read certain lines if they merely contain a single surplus whitespace at the end. This is why, in this particular case, the focus is shifted towards producing pretty Verilog output files, while the template code readability regrettably suffers as a result. Listing 2 on the next page shows, as an example, one of the template files written for this process.

Aside from Verilog code, another file has to be generated specifically for Lattice Diamond that tells it which of the source files to include when compiling the project. This is dependent on the types of devices in the configuration. When a module does not get instantiated inside a project, Lattice Diamond assumes that this module belongs to the top level, which causes compilation errors. A similar problem happens should a module be referenced that does not belong to the source files.

[10]`https://github.com/etianen/moody-templates`

```
{% extends "base.tpl" %}
{% block title %}Output device modules file{% endblock %}
{% block body %}

{% for i, dev in enumerate(devices) %}
/*
    Output device {{dev.dev_id}} ({{dev.name}})
    Type: {{dev.type}}
    Pin(s) used: {{", ".join(pin.assigned_pin.name for pin in dev.pins.values())}}
*/
// Device outputs{% for pin_name, pin in dev.pins.items() %}
wire dev_{{ dev.dev_id }}_{{ pin_name }}_o;
assign pin_signals_o[{{pin_ids[pin.assigned_pin.name]}}] = \\
{% if pin.active_low %}~{% endif %}dev_{{ dev.dev_id }}_{{ pin_name }}_o;{% endfor %}

// Device driver
{{dev.template.driver}} #(
    .DEV_ADDR(8'd{{dev.dev_id}}){% for param_name, param in dev.parameters.items() %},
    .{{param_name.upper()}}({{param.export()}}){% endfor %}
) output_{{dev.dev_id}} ({% if dev.uses_clk %}
    .clk_i        (clk_i),{% endif %}{% if dev.uses_bus %}
    .bus_access_i (bus_access_i[{{i}}]),
    .bus_req_o    (bus_requests_o[{{i}}]),
    .bus_cmd_o    (bus_cmd_o),{% endif %}
    .state_i      (device_states_i[{{dev.dev_id * 24 + 23}}:{{dev.dev_id * 24}}])\\
{% for pin_name, pin in dev.pins.items() %},
    .{{pin_name}}_o(dev_{{dev.dev_id}}_{{pin_name}}_o){% endfor %}
);
{% endfor %}
{% endblock %}
```

**Listing 2:** Verilog code template for the file containing output modules. Tags are identified by curly braces. Some long lines had to be wrapped here; this is denoted by a double backslash (\\).

# 5 Evaluation

By comparing the behavior of the solution to the **goals** we set before we started working on it, we will see if we can consider our work to be successful. We will especially take considerable care to evaluate the system's **reliability**, including resilience against connectivity issues. Finally, we shall test the **usability of the management UI** by performing a small-scale user evaluation.

Whenever applicable, significant improvements over the SPI Prototype shall be noted.

## 5.1 Evaluating the Goals

In this section, we will revisit the goals we set at the beginning of Chapter 3 (pg. 15) and decide, by examining our solution's properties and behavior, whether we have fulfilled them to an appropriate extent.

### 5.1.1 Maintainability and Accessibility

Although the bridge hardware still has to be reprogrammed through Lattice Diamond, the approach of offering the management UI made device reconfiguration easier to perform, especially for non-developers. The comparably simple graphical tool produces ready-to-use Verilog code that, after integrating into the Lattice project, automatically compiles to a usable result. Compared to the SPI Prototype, which required manual reworking of the Verilog code, this is a big improvement in terms of accessibility.

After re-initializing the bridge integration inside Home Assistant after performing changes in a device configuration, the gateway automatically adds all new devices without any required intervention. However, should device IDs have been changed in the process, then old device entities sometimes point to wrong devices, and any appliances or sensors that were removed from the configuration still appear inside Home Assistant. Therefore, it is advised to manually remove the bridge integration before adjusting the device configuration.

The accessibility still has room for improvement. For example, the reconfiguration process could be further streamlined by having the management UI making direct use of Lattice's build tool chain, avoiding the need of manually copying files and executing the build process inside Lattice Diamond. Further research has to be done on the integrability of the Lattice toolset in this regard.

### 5.1.2 Scalability

The bridge logic and protocol enforce a hard limit of 256 devices for each group of appliances and sensors. The MachXO2 only contains 110 usable I/O ports for devices, however. Depending on the types of devices used, this should allow for approximately 30 to 50 appliances inside a single environment, perhaps more if the number of sensors is reduced.

For a private home environment, this number should be more than sufficient. Professional or office use might drive the solution to its limits, however.

Although this was not tested due to lacking the necessary equipment, one could theoretically increase the number of devices by adding more bridges onto the very scalable I$^2$C bus, each configured with a different slave address. Home Assistant readily supports this, as several instances of the bridge integration can be added, each with a different configuration. Quick testing also revealed that the bindings used to access the Raspberry Pi's I$^2$C interface do not appear to require exclusive access to a single bus, making this a potentially feasible idea; but this warrants more research as well. At the very least, the Raspberry Pi's second hardware I$^2$C bus should be usable by another FPGA bridge without any conflicts.

### 5.1.3 Conformance and User Experience

Through use of the Config Flow feature mentioned in Subsection 4.3.2 (pg. 71), the bridge integration can be added and configured effortlessly to an existing Home Assistant setup. All devices are automatically recognized and made available for control.

Home Assistant only provides default names for each device based on its type, but this can be changed via the device's management menu to allow for more user-friendly names.

Automation rules are, for the most part, easily created using Home Assistant's management interface, allowing sensor-appliance interactions to be customized freely. All input modules have been shown to successfully trigger automation rules.

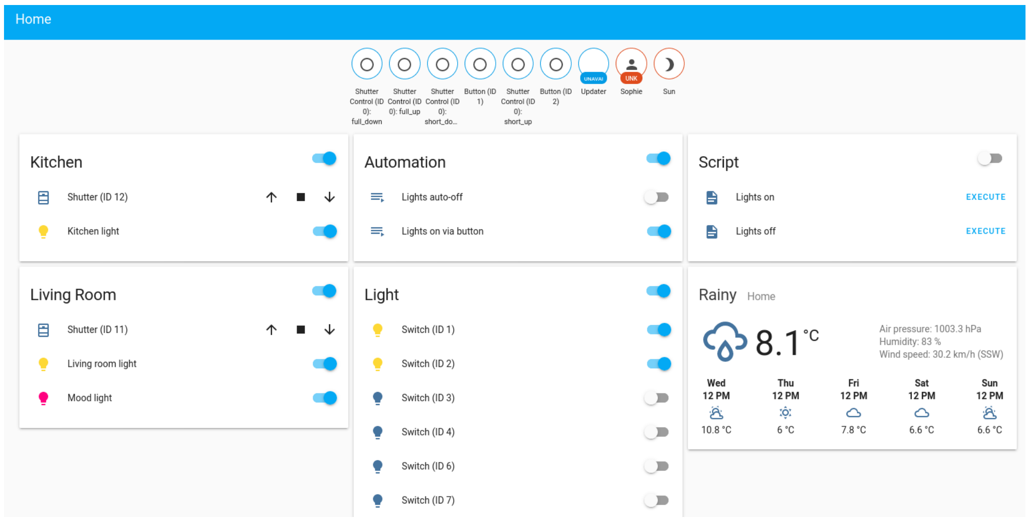


**Figure 5.1:** Home Assistant device control interface, showing all connected appliances and their current state. Devices can be categorized by area and renamed freely.

Unfortunately, the software has a few quirks that stand in the way of its usability. For instance, automation of Cover devices like our Shutter is less straightforward than with the other devices; as it involves manually inputting the name of the internal entity belonging to the shutter, whereas other devices can simply be selected from a list. This is unfortunate, but there appears no obvious way around it. Home Assistant's management interface is still relatively recent, and perhaps a future version improves on this issue.

To further improve the user experience, the device management UI could be augmented to also generate a configuration file for Home Assistant that automatically sets up appropriate automation rules based on the device configuration. The software stores this information in the human-readable YAML file format, which is similar to JSON and thus very easy to generate. However, as devices in automation rules are identified by their internal entity names, these would have to be generated as well. Also, this method would require some way of automatically transferring these files to the gateway and then restarting Home Assistant afterwards.

### 5.1.4 Performance

Home Assistant, surprisingly, performs magnitudes better on the Raspberry Pi than openHAB did. The required time from startup to full availability of the UI averages at around ten seconds; although the UI sometimes did take a few more seconds to initialize. Considering this only happens when the system is starting up, however, this becomes a moot point.

Slightly more groundbreaking is the interface's responsiveness to user input. Whereas with the SPI Prototype, there used to be a significant delay of up to two seconds between toggling a lamp's state via openHAB's UI and seeing the result, in Home Assistant every change takes place nearly instantly, even if toggling several lamps at once. This is a big improvement for the overall user experience.

## 5.2 Evaluating the Reliability

As this thesis specifically set out to create a smart home solution that is resilient, its reliability under various circumstances should be examined in greater detail.

### 5.2.1 Response to Connection Problems

A key feature of the solution is its ability to handle connection problems between the bridge and the gateway. To evaluate this, three tests have been performed during standard operation: removing the cabling between the two devices, a spontaneous reboot of the Raspberry Pi and reprogramming of the bridge hardware. Both have led to similar results, described below.

#### From the bridge's perspective

After five seconds of no contact from the Raspberry Pi, the bridge automatically switches to its standalone operation mode. The device states are kept as they were before the connection failure, and sensor commands are automatically sent to the assigned appliances, allowing them to be controlled manually.

To the user, this mode is currently expressed through activation of a LED onboard the MachXO2, but this could potentially be changed to send a signal to any device. For example, this signal could be used to activate a watchdog timer that, after a certain amount of time, automatically reboots the Raspberry Pi, removing the need for manual intervention and thus further improving the system's overall availability.

#### From the gateway's perspective

Failure during an I$^2$C communication causes the API to temporarily pause automatic polling, with an continuously increasing amount of timeout up to 30 seconds. During this time, inside Home Assistant's user interface, the device controls become unresponsive. After the connection is restored, polling automatically resumes and normal operation is resumed.

### 5.2.2 Long-Term Uptime and Stress Tests

The system has been observed to remain fully functional for at least four days of constant operation under normal circumstances, with no connection failure between the bridge and gateway logged.

Furthermore, the gateway has underwent a short stress test to examine its reliability during times when the system is in high load. This should usually not be the case during production, as the Raspberry Pi is not supposed to run any other services besides the gateway software. However, maintenance may sometimes require system updates, which can cause a significant impact on performance due to disk I/O.

Putting the gateway under severe CPU and memory load for three minutes does not appear to cause any apparent problems, as the Home Assistant UI has remained responsive throughout. Only when high disk activity was introduced to the stress test, however, the bridge eventually signaled a connection failure. This is to be expected from a Raspberry Pi, as personal experience shows that the storage controller is by far its biggest bottleneck.

After the stress test had finished, the connection was reestablished almost immediately, showing that the solution behaves exactly as intended under this circumstance.

### 5.2.3 State Update Event Redirection Bug

Although the hardware logic has been carefully tested as much as possible, the bridge unfortunately still contains a critical bug.

In certain cases, the events of the State Update bus, which are meant for giving an output module the ability to update its own state, do not seem to get sent by the Device Manager to the correct output module. This was noticed because sometimes, a shutter gets stuck inside a movement cycle, failing to stop itself. Instead, the update request either gets lost completely or sent to another output module, turning it off instead. This bug only appears during the bridge's standalone mode. When connected to the gateway, the state update succeeds at all observed times.

Regrettably, this flaw significantly reduces the solution's overall reliability when used without the Raspberry Pi. Although much time has been spent in attempting to track the bug down, the limited set of testing tools available as well as a lack of personal experience in the field of hardware development caused the bug's origin to remain in obscurity.

## 5.3 Usability of the Management UI

A very brief evaluation of the management tool designed in Section 3.6 (pg. 56) was performed to receive feedback on the dialog design and general usability.

The test group consisted of four people with varying degree of expertise in the smart home domain, ranging from completely inexperienced to competent. They were briefly introduced to the basic concepts of the tool and its purpose. Afterwards, they were given a fabricated scenario of a smart home setting with some appliances and sensors, which they were then supposed to translate to a device configuration for this solution, using the management application. The whole process was closely monitored and a brief interview was held afterwards, asking the subjects for feedback on the design of the application and their experience while using it.

Each test subject successfully managed to autonomously recreate the fictional scenario using the application. The somewhat less experienced users took a bit longer than the ones more versed in the field, which was to be expected.

Following is a brief summary of the most significant points from the user feedback.

Whenever a new device is added, its settings dialog pops up before any pin assignments can be done (as can be seen in Figure 3.28 (pg. 59).) This has caused some confusion among one of the users, as they expected to be able to assign pins right away, not finding the option for it among the device's settings. It could be considered to omit the configuration dialog upon adding a device, only showing it when a user presses the appropriate button, but it would have to be tested again whether this method meets with more acceptance.

The device parameter settings were also confusing to some users, as the tool tip hints mostly went unnoticed by them, but this can also be explained by having failed to provide proper context beforehand.

The preview image was generally seen as helpful, but has earned some criticism in regards to its color choice. Pins occupied by appliances display as bright red, which was interpreted by most test subjects as a warning or sign of misconfiguration. Admittedly, since warnings are displayed as red in other parts of the application, this conclusion is not far-fetched at all, and as a result the color scheme will be reconsidered in an eventual future update.

On a side note, although the application was intended to work the same across all platforms, it has turned out during testing that highlighting certain buttons with green and red colors did not work under certain versions of Windows. Instead of using colors, it might be more favorable to use recognizable icons instead. This will also aid vision-impaired users of the software.

Overall, the evaluation of the management UI showed that while it has some minor flaws, its usability is generally acceptable. Since the resources allotted for this project allowed for only a test group of four people, which is rather tiny, the results are obviously not very accurate. The tests would have to be resumed with a bigger group to yield more useful information.

# 6 Conclusion

At the end of this thesis, we will **summarize** the results of our work once again and give a brief **outlook** on future work that can still be done on the project.

## 6.1 Summary

In this thesis, we have analyzed the basic concepts of a smart home system including its features, device components and connections. We have designed and realized a complete proof-of-concept solution comprised of individual components. It contains a bridge that manages appliances and sensors of varying numbers and types, with reconfigurable FPGA hardware logic built from the ground up specifically for this project. We created a gateway using a Raspberry Pi that can communicate with the bridge and provide a convenient user interface for controlling the appliances and automation rules. For communication between the two, a lightweight, low-level $I^2C$ protocol was invented that has been secured against noise-related errors. To ease the process of reconfiguring the bridge hardware, a management UI tool was designed allowing users to create device configurations and export the corresponding Verilog hardware description language code.

We have discussed some details of the design's implementation, including working with Verilog and creating a Python API for interfacing with the bridge. A Home Assistant integration was built that makes use of the API to allow integration of the bridge into the gateway software.

The finished solution was evaluated based on various criteria, showing that a smart home solution can be created using low-cost hardware that is resilient to connection failure, providing basic device control through the bridge when the gateway is unavailable. We have seen that the solution responsive, accessible and provides a user experience similar to commercially available smart home solutions. The solution works purely in the local network and is thus independent from potentially occurring internet connection problems.

## 6.2 Outlook

As was discussed during the evaluation in Chapter 5 (pg. 79), the solution still has room for improvement. The user experience and accessibility can still be furthered by, for example, integrating the management tool more tightly into the hardware programming process and gateway software configuration. The bridge hardware logic can also be further hardened against potentially undiscovered bugs, perhaps by someone slightly more experienced in hardware design.

It was stated during the introduction that there will be no focus placed on the actual integration of the system into a building's wiring, or to make it usable with different kinds of appliances that require certain voltages or other power requirements. Instead, a basis for further work on this field to be done has been provided. The same way it itself was based on a prototype, the solution we have now created should serve as an effective starting point for a concrete implementation of a resilient smart home system suitable for use in both personal and professional environments.

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| API | Application programming interface. A software library written to provide easier programmatic access to other software or hardware. See Subsection 3.5.3 (pg. 48). |
| CRC | Cyclic redundancy check. An error detecting code used primarily in networks. See Subsection 3.4.5 (pg. 44). |
| FPGA | Field-programmable gate array. An integrated circuit with the ability to be configured after manufacturing. See Subsection 3.1.1 (pg. 17). |
| HDL | Hardware description language. A computer language used for abstract descriptions of hardware logic. See Subsection 3.1.1 (pg. 17) and Section 4.1 (pg. 61). |
| $I^2C$ | Inter-Integrated Circuit. A serial bus used especially in communication within integrated circuits. See Subsection 2.2.1 (pg. 9). |
| I/O | Input/output. |
| IoT | Internet of Things. |
| JSON | JavaScript Object Notation. A human-readable data format. See Section 4.4 (pg. 73). |
| PWM | Pulse-width modulation. A technique used to simulate analog voltage control by sending pulses of digital signals at certain frequencies. See Subsection 3.3.2 (pg. 29). |
| SPI | Serial Peripheral Interface. An interface specification used for short-distance communication. See Subsection 2.2.1 (pg. 9). |
| YAML | YAML Ain't Markup Language. A human-readable data format. |
| WB | Wishbone. A standardized communication bus found in certain integrated devices. See Subsection 3.3.4 (pg. 38). |
| WYSIWYG | What You See Is What You Get. A design principle for editor software, where editing is performed on a visual preview of the final result. See Section 4.4 (pg. 73). |

# Bibliography

[Ama20] AMAZON: *Amazon Echo (2nd Generation)*. https://www.amazon.com/-/de/gp/product/B0749WVS7J/ref=ods_ac_dp_dr_ps?th=1. Version: 2020. – Accessed on 2020-01-12

[And19] ANDROID OPEN SOURCE PROJECT: *Raspberry Pi 3 - Android Things*. https://developer.android.com/things/hardware/raspberrypi. Version: 2019. – Accessed on 2019-12-19

[App20a] APPLE: *HomeKit - Apple Developer*. https://developer.apple.com/homekit/. Version: 2020. – Accessed on 2020-01-13

[App20b] APPLE: *iOS - Home*. https://www.apple.com/ios/home/. Version: 2020. – Accessed on 2020-01-13

[ARA12] ALAM, M. R. ; REAZ, M. B. I. ; ALI, M. A. M.: A Review of Smart Homes—Past, Present, and Future. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42 (2012), Nov, Nr. 6, S. 1190–1203. http://dx.doi.org/10.1109/TSMCC.2012.2189204. – DOI 10.1109/TSMCC.2012.2189204. – ISSN 1558–2442

[Ard19a] ARDUINO AG: *Arduino - ArduinoBoardMega*. https://www.arduino.cc/en/Main/arduinoBoardMega/. Version: 2019. – Accessed on 2019-12-19

[Ard19b] ARDUINO AG: *Arduino Ethernet Shield 2 - Arduino Official Store*. https://store.arduino.cc/arduino-ethernet-shield-2. Version: 2019. – Accessed on 2019-12-19

[Bak05] BAKER, N.: ZigBee and Bluetooth strengths and weaknesses for industrial applications. In: *Computing Control Engineering Journal* 16 (2005), April, Nr. 2, S. 20–25. http://dx.doi.org/10.1049/cce:20050204. – DOI 10.1049/cce:20050204. – ISSN 0956–3385

[Blo06] BLOCH, Joshua: How to Design a Good API and Why It Matters. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. New York, NY, USA : Association for Computing Machinery, 2006 (OOPSLA '06). – ISBN 159593491X, 506–507

[Blu19] BLUETOOTH SPECIAL INTEREST GROUP (Hrsg.): *Bluetooth Core Specification*. 5.2. Bluetooth Special Interest Group, 12 2019

[Bro12] BROADCOM CORPORATION (Hrsg.): *BCM2835 ARM Peripherals*. Broadcom Corporation, 2012. `https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf`. – Accessed on 2019-12-19

[BV04] BROWN, Stephen ; VRANESIC, Zvonko: *Fundamentals of Digital Logic with VHDL Design*. 3. USA : McGraw-Hill, Inc., 2004. – ISBN 0072499389

[Cha01] CHAKRAVARTY, Tridib: *Performance of Cyclic Redundancy Codes for Embedded Networks*. Version: 2001. `http://www.ece.cmu.edu/~koopman/thesis/chakravarty.pdf` Accessed on 2019-12-02

[Con19] CONNECTED HOME OVER IP WORKING GROUP: *Project Connected Home over IP*. `https://www.connectedhomeip.com/`. Version: 2019. – Accessed on 2019-12-20

[Goo20] GOOGLE: *Bringing you the next-generation Google Assistant*. `https://blog.google/products/assistant/next-generation-google-assistant-io`. Version: 2020. – Accessed on 2020-01-12

[Hom20a] HOME ASSISTANT COMMUNITY: *Components Architecture - Home Assistant dev docs*. `https://developers.home-assistant.io/docs/en/architecture_components.html`. Version: 2020. – Accessed on 2020-01-08

[Hom20b] HOME ASSISTANT COMMUNITY: *Development Checklist - Home Assistant dev docs*. `https://developers.home-assistant.io/docs/en/development_checklist.html`. Version: 2020. – Accessed on 2020-01-07

[Hom20c] HOME ASSISTANT COMMUNITY: *Entity - Home Assistant dev docs*. `https://developers.home-assistant.io/docs/en/entity_index.html`. Version: 2020. – Accessed on 2020-01-08

[Hom20d] HOME ASSISTANT COMMUNITY: *Set up Development Environment - Home Assistant dev docs*. `https://developers.home-assistant.io/docs/en/development_environment.html`. Version: 2020. – Accessed on 2020-01-10

[HSWW17] HAACK, William ; SEVERANCE, Madeleine ; WALLACE, Michael ; WOHLWEND, Jeremy: *Security Analysis of the Amazon Echo*. 5 2017

[ISO06] ISO Central Secretary: Ergonomic requirements for office work with visual display terminals, part 110: Dialogue principles / International Organization for Standardization. Version: 2006. https://www.iso.org/standard/38009.html. 2006 (ISO 9241-1110:2006). – Forschungsbericht

[JO18] Jackson, Catherine ; Orebaugh, Angela: A study of security and privacy issues associated with the Amazon Echo. In: *Int. J. Internet of Things and Cyber-Assurance* 1 (2018), Nr. 1, S. 91 – 100

[Koo15] Koopman, Philip: *Best CRC Polynomials*. http://users.ece.cmu.edu/~koopman/crc/index.html. Version: 2015. – Accessed on 2019-12-06

[Koz16] Kozak, Blake: *Household Penetration of Smart Home Subscribers and Devices*. https://technology.ihs.com/584229/household-penetration-of-smart-home-subscribers-and-devices. Version: 2016. – Accessed on 2019-12-25

[Lat16] Lattice Semiconductor (Hrsg.): *Using User Flash Memory and Hardened Control Functions in MachXO2 Devices Reference Guide*. 2.4. Oregon, US: Lattice Semiconductor, 11 2016

[Lat17] Lattice Semiconductor (Hrsg.): *MachXO2 sysCLOCK PLL Design and Usage Guide*. 2.7. Oregon, US: Lattice Semiconductor, 03 2017

[Lat19] Lattice Semiconductor: *MachXO2*. https://www.latticesemi.com/Products/FPGAandCPLD/MachXO2. Version: 2019. – Accessed on 2019-12-17

[LSS07] Lee, J. ; Su, Y. ; Shen, C.: A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi. In: *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*, 2007. – ISSN 1553–572X, S. 46–51

[NXP14] NXP Semiconductors (Hrsg.): *I$^2$C bus specification and user manual*. 6. NXP Semiconductors, 4 2014

[ope19a] openHAB Foundation e.V.: *GPIO - Bindings*. https://www.openhab.org/addons/bindings/gpio1/. Version: 2019. – Accessed on 2019-12-19

[ope19b] openHAB Foundation e.V.: *myopenHAB Website*. https://www.myopenhab.org/. Version: 2019. – Accessed on 2019-11-25

[ope19c] openHAB Foundation e.V.: *openHAB Documentation - HABPanel*. https://www.openhab.org/docs/configuration/habpanel.html. Version: 2019. – Accessed on 2019-11-25

[PB61] Peterson, W. W. ; Brown, D. T.: Cyclic Codes for Error Detection. In: *Proceedings of the IRE* 49 (1961), Jan, Nr. 1, S. 228–235. http://dx.doi.org/10.1109/JRPROC.1961.287814. – DOI 10.1109/JRPROC.1961.287814. – ISSN 2162–6634

[Phi20a] Philips: *Friends of Hue*. https://www2.meethue.com/en-us/works-with. Version: 2020. – Accessed on 2020-01-13

[Phi20b] Philips: *New to Hue: Bluetooth smart LED lights*. https://www2.meethue.com/en-us/blog/bluetooth-led-lights. Version: 2020. – Accessed on 2020-01-13

[Phi20c] Philips: *The official site of Philips Hue*. https://www2.meethue.com/en-us/. Version: 2020. – Accessed on 2020-01-13

[Pyt20] Python Software Foundation, The: *Packaging Python Projects*. https://packaging.python.org/tutorials/packaging-projects/. Version: 2020. – Accessed on 2020-01-10

[Ras19a] Raspberry Pi Foundation: *GPIO - Raspberry Pi Documentation*. https://www.raspberrypi.org/documentation/usage/gpio/. Version: 2019. – Accessed on 2019-12-19

[Ras19b] Raspberry Pi Foundation: *Raspberry Pi 3 Model B*. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Version: 2019. – Accessed on 2019-12-17

[Ras19c] Raspberry Pi Foundation: *Raspberry Pi Downloads - Software for the Raspberry Pi*. https://www.raspberrypi.org/downloads/. Version: 2019. – Accessed on 2019-12-19

[Sar88] Sarwate, D. V.: Computation of cyclic redundancy checks via table look-up. In: *Communications of the ACM* 31 (1988), aug, Nr. 8, 1008–1013. http://dx.doi.org/10.1145/63030.63037. – DOI 10.1145/63030.63037

[Ver06] IEEE Standard for Verilog Hardware Description Language. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), April. http://dx.doi.org/10.1109/IEEESTD.2006.99495. – DOI 10.1109/IEEESTD.2006.99495

[Wik19a] WIKIPEDIA: *Field-programmable gate array*. `https://en.wikipedia.org/wiki/Field-programmable_gate_array`. Version: 2019. – Accessed on 2019-11-14

[Wik19b] WIKIPEDIA: *Serial Peripheral Interface*. `https://en.wikipedia.org/wiki/Serial_peripheral_interface`. Version: 2019. – Accessed on 2019-11-14

[Wik19c] WIKIPEDIA: *Wishbone (computer bus)*. `https://en.wikipedia.org/wiki/Wishbone_(computer_bus)`. Version: 2019. – Accessed on 2019-11-14

[Wik20a] WIKIPEDIA: *Google Home*. `https://en.wikipedia.org/wiki/Google_Home`. Version: 2020. – Accessed on 2020-01-12

[Wik20b] WIKIPEDIA: *HomeKit*. `https://en.wikipedia.org/wiki/HomeKit`. Version: 2020. – Accessed on 2020-01-13

[Wik20c] WIKIPEDIA: *Wi-Fi*. `https://en.wikipedia.org/wiki/Wi-Fi`. Version: 2020. – Accessed on 2020-01-04

[Wik20d] WIKIPEDIA: *Z-Wave*. `https://en.wikipedia.org/wiki/Z-Wave`. Version: 2020. – Accessed on 2020-01-04

[Zig20] ZIGBEE ALLIANCE: *Home - Zigbee Alliance*. `https://zigbeealliance.org/`. Version: 2020. – Accessed on 2020-01-05