Smart home report

Baptiste Jacquemot, Thomas Berkane, Thomas Bienaimé
baptiste.jacquemot@epfl.ch, thomas.berkane@epfl.ch, thomas.bienaime@epfl.ch
EPFL

Advised by Prof. George Candea and Solal Pirelli from DSLAB

June 2021

Outline

1 Smart home design

Our goal in this project was to create a smart home that can be proved correct and that respects given requirements. The code for the smart home can be found here:
`https://github.com/BajacDev/doge-home`

Here is a general definition of a smart home to give a first overview of our goal [35]:

A smart home is a home in which a communications network connects appliances and services and allows them to be remotely controlled, monitored or accessed.

So a home which is smart must contain 3 elements:

1. Home automation, which means devices within the home and links to services and systems outside the home.
2. Intelligent control, which is going to be the gateways to manage the system.
3. And an internal network, which is the basis of the smart home and can be wire, cable, or wireless.

1.1 Requirements

The smart home must respect the requirements given by the following scenario.

An elderly couple in their 80s, living alone in a house in a semi-rural area (think e.g. of grandparents). One of them is disabled and in a wheelchair, while the other spouse is still mobile. The objective of transforming this home into a smart-home using domotics is, in decreasing order of priority:

1. watch over and improve the health and safety of the couple
2. maximize their independence and psychological comfort
3. provide pleasant living conditions

1.1.1 Basic architecture

It has been agreed that for security purposes the architecture would be composed of a single smart hub and thus that communication between devices would be centralized. This design allows a better monitoring of connections between devices. This smart hub runs on a Raspberry Pi. In our model we include the Raspberry Pi and the devices connected to it in our trusted computing base, as well as the OS running on the Raspberry Pi event though it has not been verified yet. Thus, only the smart hub will be verified for now.

1.1.2 Selection of the smallest feature set

In order to make verification of the smart home easier, we determined a minimal set of components to include in it. This set has been determined from a selected list of smart home devices (A.2) and user stories (A.1) that can be found in the appendix.

The smallest set in priority order that was chosen is:

- Smart watch (detect falls & analyzes health data), Connected door lock (independence & safety).
- Camera and front door speaker (safety). This part has not been implemented and is left for potential future work.

In our design, the smart watch and the smart hub are connected via TCP on the Internet (more information in the app section).

For the communication between the smart lock and the raspberry pi, we decided to look at common smart home protocols.

1.1.3 Smart home and device protocols

Smart homes and home automation do not have a general standard for communication. Here is a small list of well-known standards and their uses:

- MQTT: this is a publish/subscribe protocol. It runs over TCP/IP and can use TLS for security. Its specification is open. It is used as an interconnection between other protocols. End devices usually do not use MQTT, thus an interface is needed between MQTT and the device's protocol.
- Zigbee: very popular, it uses a wireless mesh network topology so devices are connected to each other. However, it does require a central hub to operate, so that it can coordinate the devices with each other. It's built on the physical layer and MAC sublayer for low-rate wireless personal area networks, on top of which it adds its own network layer and application layer.
- Z-wave: similar to Zigbee and also very popular. Differences are at the level of individual metrics. Z-Wave is a bit more compatible with other devices, but Zigbee is faster and has a longer range.
- CoAP: also a network protocol but uses a client-server architecture and is one-to-one, uses UDP, and DTLS for security. Its specification is open.
- KNX: for large-scale building automation. It can manage lighting, blinds and shutters, ventilation, etc.
- DALI: more specialized, only for controlling lighting.

For further investigation we selected only a small fraction of those protocols: MQTT, Zigbee and KNX. We also investigated wired connection between a Raspberry Pi and a door lock.

- KNX is not a viable option because of lack of KNX door locks and of a verified security library.
- Zigbee does not use a security library which has been verified, and we would have to add a USB stick to connect Zigbee to the Raspberry Pi (which is another element which cannot be verified)
- MQTT we did not find a door lock, we found a wifi door lock but the vendors do not specify what they used and are not very informative about it.
- The wired connection is the solution we chose, but it has scalability problems: if we add more doors or connected windows we will have to think of how to handle them, and we will need to treat each of them

So to begin with, it has been decided to use a simple custom protocol for the communication between the pins of the Raspberry Pi and the door lock. The smart home having to be verified, it is an important criteria that we should have a small code base. The internal architecture of the smart home has been designed in such a way, that adding a new communication channel (i.e. a binding) for any protocol can be done easily without changing the code structure too much.

1.2 Smart home internal design

To avoid issues with the Rust ownership system, which states that we cannot have two pointers that point and write to the same object. We decided to create a simple hierarchical design (see figure 1).

At the top of the figure is the smart home core, composed of an infinite loop which calls the bindings' methods. We modeled our notion of bindings on OpenHAB bindings [1]:

> A binding is an extension to OpenHAB that integrates an external system like a software service or a hardware device.

Each binding has a fetch function that returns an event. The smart home core fetches the events from all bindings, processes them, modifies the smart home's state and triggers the bindings' actions.

1.3 The electronic circuit

The circuit for the smart home is composed of the following components:

- a Raspberry Pi 4 [2] to serve as the smart hub
- one SD card [3] for the Raspberry Pi
- a USB C cable to supply power to the Raspberry Pi

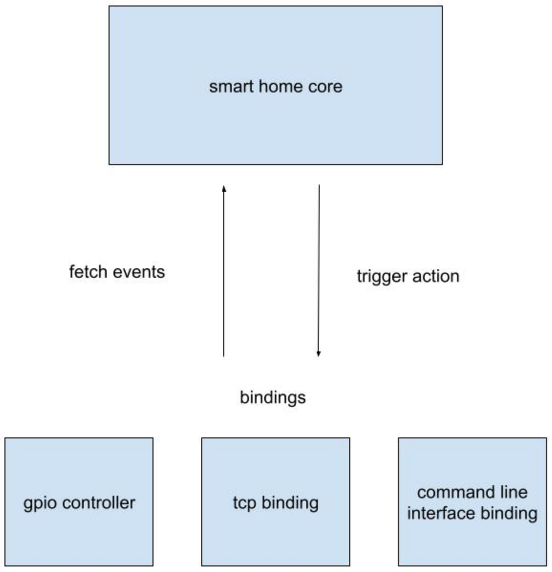

Figure 1: Smart home software design

- an electric strike [4]
- a relay module [5] to control the electric strike from the Raspberry Pi
- three jumper cables [6] to connect the Raspberry Pi pins to the pins of the relay module
- a 12V power supply [7] for the electric strike

In addition to these, and for future reference, some other components have been purchased for working on the smart home:

- two Raspberry Pi Zero [8] for testing purposes
- two SD cards [3] for use with the two Raspberry Pi Zero

2 The mobile application

2.1 Introduction

A mobile application [9] has been created to interact with the smart home. It is an equivalent solution to replace the smart watch in our design in respect to fall

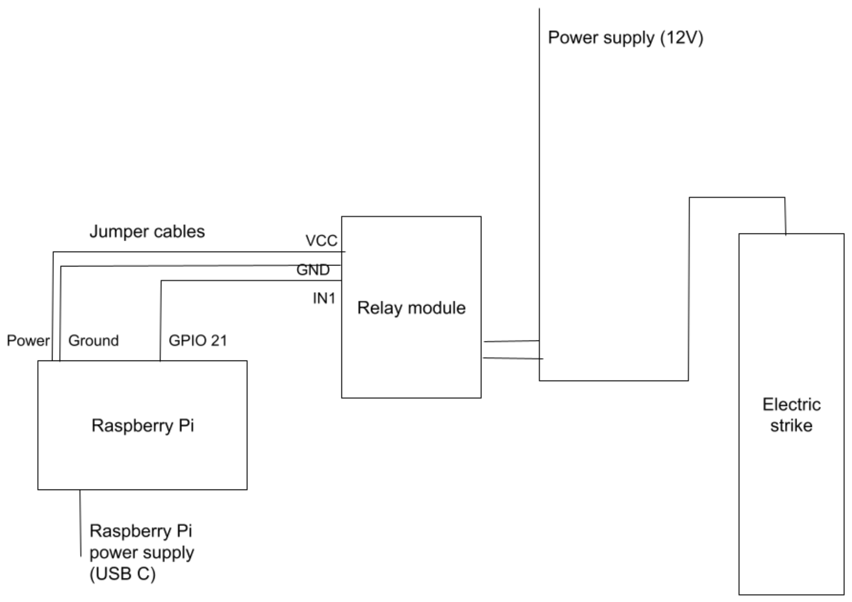

Figure 2: A diagram of the circuit for the smart home.

detection. The app has two main functionalities. The first one is to remotely toggle the door-lock by pressing a button in the app. The second is to detect if the user suffers a fall and in that case launch a phone call to a previously saved number, and open the door so that a health assistant can get in.

For example, if the user is receiving visit and wants to open the door, then they can do so simply by pressing the button on the app, which saves them from having to move to the door. If later they have an accident and suffer a fall, the automatically launched phone call will allow them to quickly warn a health assistant. The door also unlocks itself automatically, which will let the health assistant get in and help the user.

2.2 Technical choices

We chose to develop the app on Android for several reasons. Android is the most used mobile operating system [10], it has a robust emulator, and there are many Android smart watches available (see Further work). Furthermore, we chose to develop it in Java (instead of Kotlin) because that was the language used by the repository on which the fall detection functionality is based.

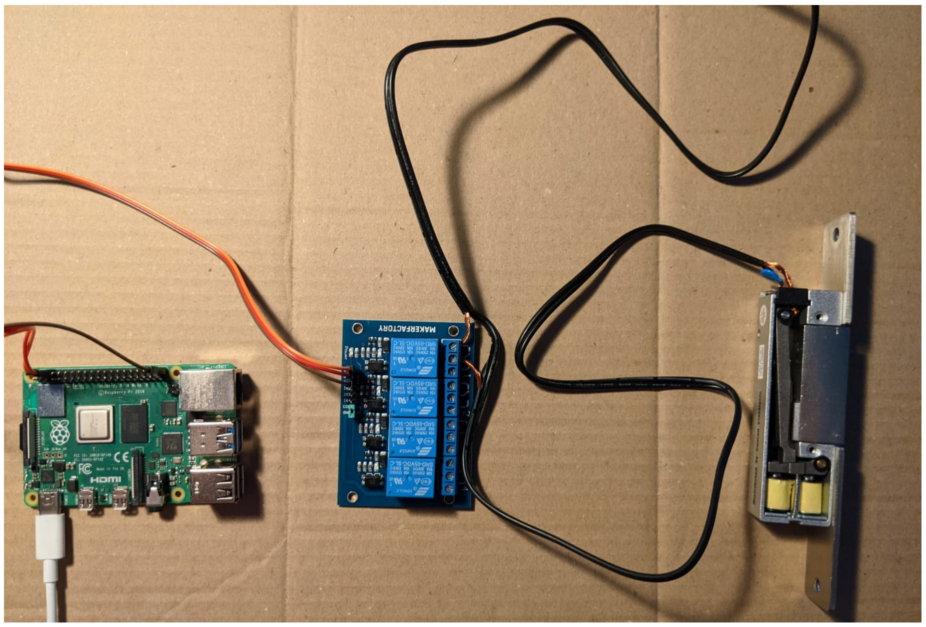

Figure 3: A photo of the circuit for the smart home. From left to right: Raspberry Pi 4, relay module and electric strike.

2.3 Quick start

Below are some quick steps to get the application up and running.

- Launch smart hub on raspberry pi
- Ssh into raspberry pi and launch ngrok on it using "ngrok tcp 8080 –region eu"
- Copy and paste hostname and port given by ngrok over to application's MainActivity
- Download app onto Android phone
- Configure health assistant's phone number inside of app

2.4 Fall detection

To avoid writing a fall detector from scratch, we looked for existing projects which implement this functionality. After considering the existing repositories, we decided to use [11] for several reasons.

First, it was the most minimal that we found as it provides almost only the functionality that we need (we just had to remove a part of the code which plots

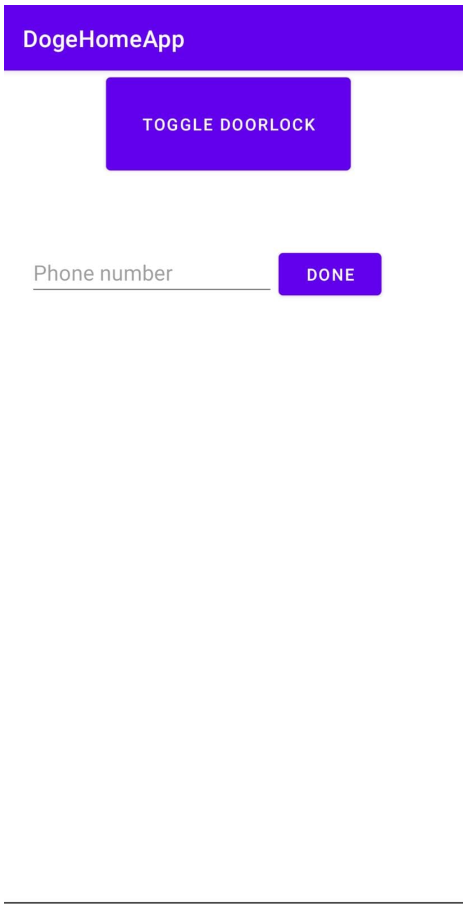

Figure 4: The design of the app was kept simple to make it as easy to use as possible by an elderly user.

the values of the accelerometer). Whereas projects such as [12] also do other things such as keeping a history of falls in a local database.

Second, it uses few libraries and those which it uses are widely known, such as the Java standard library and the Android hardware and OS libraries.

Finally, after multiple tests, the fall detection has been shown to be accurate and easily adjustable as there is a single parameter to tweak to adjust the sensitivity of the fall detection.

In case a fall is detected, we needed two things to happen: to warn a heath assistant, and to enable that health assistant to enter the house so that they can help the user.

2.4.1 Sending an emergency alert to a health assistant

The considered alternatives were:

1. Launching a telephone call. The advantage of this approach is that in addition to warning the assistant it actually allows the user to communicate with them. But it requires us to add the telephone lines to our trusted computing base (TCB). However, those are almost surely already being used by the user so they are already trusted.

2. Sending a UDP message to the assistant's phone, which would act as a notification that the user needs help. An advantage of this method is that the UDP message could be first sent to the smart hub and then the smart hub would send a UDP message to the health assistant. Indeed, since the app is not in the TCB, this would help to reinforce security and the single entry-point principle, since communication would not be occurring between the two phones directly but rather between the smart hub and the assistant's phone.

   However, since the two devices are on separate networks, we have found that it is not possible to directly send a message from one to the other. Possible solutions would be to create a UDP tunnel or to use port forwarding. The former would require to trust an external piece of code (such as ngrok) to establish the tunnel, and the latter would require gateway configuration, which is impossible in our situation.

In the end the first alternative has been chosen as it was found to better minimize the TCB and to be more convenient for the users.

2.4.2 Allowing the health assistant to enter the house when their help is needed

Three alternatives were considered:

1. Having the health assistant keep a key to the house. This is simple and convenient, but it has the downsides that the assistant has to be trusted with the key and that it might be inconvenient for them to keep many keys in case they are in charge of many users.

2. Opening the door once a fall is detected. This has the advantage of placing less trust in the health assistant. However, it does leave the door open for a short while when the health assistant is on their way to the house.

3. Having the assistant use another application, which would allow them to open the door remotely when they get to it. However, since the assistant's phone and the smart hub would be in different networks, we run into the same problem as above for warning the health assistant.

The second alternative was chosen here as it better minimizes the TCB compared to both the other options.

2.5 Communication between the app and the smart hub

To communicate with the smart hub, we chose to use TCP. The main alternative considered was UDP. The reason for our choice is that we want to maximize the reliability of the smart home, and so we want to be sure that the message will get to its destination, which TCP can guarantee but not UDP.

To be able to send messages from the app to the smart hub, we had to use ngrok [13], which establishes a TCP tunnel. We found this to be required because even though both the devices are connected to the same residential network, the network has a firewall which prevents direct communication between the devices. Thus ngrok is added to the TCB here, while keeping in mind that in a real setting, the network would be configurable at will which would mean that we would not need a TCP tunnel anymore.

Communication from the app to the smart hub consists of two types of messages: OPEN_DOOR (character '0') and TOGGLE_DOOR (character '1'). Once the TCP connection is established, the smart hub simply listens for these messages and reacts accordingly to them.

2.6 Possible further work

Below are some ideas for further work on the app.

The phone could be replaced by a smart watch. This would be more convenient for the user as it will always be attached to them so there are fewer chances of losing it or forgetting to put it on (which is necessary for the fall detection). Moreover, this would allow for the collection and processing of health signals such as heart rate and sleeping patterns perhaps. This data could then be used to make sure that the user is staying healthy and to detect certain health problems which could arise. Preferably, the data would be sent from the smart watch to the smart hub, and the latter would take care of processing it since it is trusted (and not the application).

Moreover, the communication protocol could be improved by adding authentication to it, with the use of TLS (for which verified libraries exist [14]).

3 Testing and formal verification

3.1 State of Rust safety

The goal of Rust is to provide a memory safe system with Zero Cost Abstractions. Memory safety ensure that there cannot be memory vulnerability in our program such as buffer overflow or dangling pointers. Zero Cost Abstractions ensures that all memory safety requirement and other abstractions does not have an impact on performance (zero cost) at runtime.

Rust provides the "unsafe" keyword to separate safe and unsafe code. If no unsafe is used in the code, then we can be sure that the code is memory safe thanks to its ownership system.

Despite all this, Rust libraries are built with unsafe code, in order to manipulate pointers. For instance the "Box" object uses unsafe code. Nevertheless, there is a team called Rustbelt working to formally verify unsafe parts [15]:

> Unfortunately, none of Rust's safety claims have been formally investigated, and it is not at all clear that they hold. To rule out data races and other common programming errors, Rust's core type system prohibits the aliasing of mutable state, but this is too restrictive for implementing some low-level data structures. Consequently, Rust's standard libraries make widespread internal use of "unsafe" blocks, which enable them to opt out of the type system when necessary. The hope is that such "unsafe" code is properly encapsulated, so that Rust's language-level safety guarantees are preserved. But due to Rust's reliance on a weak memory model of concurrency, along with its bleeding-edge type system, verifying that Rust and its libraries are actually safe will require fundamental advances to the state of the art.

From this point on we will assume that safe Rust (rust that does not use unsafe code) is memory safe. We will try to implement tests in order to ensure that the code does not panic. Panic is a way for Rust programmer to indicate that the program reached an unsafe state. If the code does not panic, it means it is always in a safe state.

3.2 Testing and fuzzing

In order to make a 100% secure smart home, we set up the following verification and testing tools:

- Rust basic unit testing
- fuzzing with Proptest
- formal verification with KLEE

Unlike C, Rust contains an integrated support for writing and executing unit tests [16].

Proptest is a fuzzing library based on Hypotesis, a testing framework for Python. Fuzzing consists of executing a test multiple times with different inputs generated automatically in order to trigger an assertion. Another well-known Rust fuzzing library is Quickcheck. We decided to use Proptest for two reasons [17]:

- Quickcheck does not support constraints on valuees, thus generation can lead to a lot of input rejections
- Proptest has an easier syntax than Quickcheck

But fuzzing cannot prove that the tests pass for all input values, because only a subset of all possible input are tested. Thus, in order to verify our smart home, we decided to use formal verification.

3.3 Formal verification

Rust is already memory safe, so we decided to use formal verification to verify that the "no panic" property holds.

There are many formal verification tools for Rust:

- crux-mir [18]: this is based on the MIR language used internally by the Rust compiler. Crux-mir is still actively under development.
- seer [19]: symbolic execution engine for Rust. There are still some very important features missing. For instance, it does not handle basic things such as overflow checking on symbolic arithmetic yet.
- MIRAI [20]: it is based on MIR.
- seahorn [21]: it is an automated analysis framework for LLVM-based languages. But it only supports LLVM 5.
- KLEE [22]: a symbolic execution engine. It supports up until LLVM 11. This is a well-known execution engine used with C.

We decided to use KLEE because it has a good documentation, is very stable and is the most popular formal verification tool in C. Furthermore many other formal verifiers based on MIR cannot verify unsafe code.

3.3.1 KLEE and LLVM

The LLVM Project is "a collection of modular and reusable compiler and toolchain technologies" [23] . LLVM-IR is an intermediate language close to low level languages. Rust is compiled to LLVM-IR in order to benefit from the already existing compiler from LLVM-IR to executable (elf or EXE).

Lots of projects are based on LLVM, for instance KLEE, is a formal verifier that executes symbolically LLVM code.

KLEE was not made for Rust in the first place: it was made for C. In C, one should use the clang compiler that compiles C into LLVM-IR. On the KLEE website, we can see that there is a page for using KLEE with Rust. Unfortunately, at this date, this link redirects us to a Rust Verification Tools page [24]. Rust Verification Tools is unstable, and the documentation of this tool is often out of date. Moreover to simplify the verification task, they provide a script that works in a docker, but this docker is highly bloated as it contains almost every Rust verification tool and fuzzer listed above. However reading Rust verification tools was useful in order to find other tools that make it easy to write KLEE tests with Rust [25]. I highly recommend reading their webpage for anyone wishing to continue the project.

3.3.2 The klee-rust docker

In order to have easily usable tools for writing and executing KLEE tests on any machine, we have decided to create a docker image containing KLEE with Rust and the right version of LLVM (currently LLVM 10) [26]

This docker also contains cargo KLEE [27], a simple cargo extension that allows to perform compilation and KLEE execution in one command. The compilation result can be found in `path/to/doge-home/target/debug/examples`.

In order to write tests, we use *klee-sys*, a low level binding to the klee api by the same author as *cargo-klee* [28]. Nevertheless, this interface is not enough in some cases to access KLEE features. That is why, some tests have a manual binding to the KLEE interface.

In order to make the KLEE test abort on crash there are two solutions. The first one consists in using *panic_klee* [29]. *panic_klee* overwrites Rust panics with KLEE aborts. However, in order to overwite panics, Rust requires the file and all its dependencies to use no_std:

```rust
#![no_std]
```

This can be bothersome because most of the Rust library uses std. Thus, the second option is to use a panic hook. In Rust you can attach a hook to the std implementation of panic in this way:

```rust
#![no_main]

use klee_sys::*;
use std::panic;

// with no mangle, klee will be able to find the entrypoint
#[no_mangle]
fn main() {

    //attach a hook to make klee abort on panic
    panic::set_hook(Box::new(|_| {
        klee_abort!();
    }));

    // initialization is compulsory in Rust
    let mut a: i32 = 1;
    klee_make_symbolic!(&mut a, "a");

    if a == 0 {
        // klee will panic at this point for input a = 0
        panic!("{:?}", a);
    }
}
```

In order to write KLEE tests, create a Rust file with the same format as the one above in `/path/to/doge-home/examples`. Instructions to execute it are provided in the *README.md*.

3.4 Current state of the verification

For simplicity we test the smart home core and the bindings separately. To this day only the smart home core is tested.

The smart home execution can be seen as a state diagram. At each iteration of the smart home loop, the smart home is in a state A, fetches an event E, processes E and goes to a state B. In our case, the set of states the smart home can be in is the set of states the door lock can be in, that is to say: open or closes. Bindings have been removed to only test the core.

In order to test at 100% the smart home core, we need to test that for all valid state A, after processing any event E, the smart home goes to a state B that is valid, i.e. a state that did not trigger a panic.

We propose two implementations of the smart home core test:

- a fuzzing implementation in `/tests/smarthome.rs`. This test is for show purpose only and does not test all events. It gives a good overview of Proptest syntax for future use of Proptest. But it cannot prove the smart home core to be secure because fuzzing does not cover all the input space.
- a KLEE implementation in `/examples/smarthome.rs`. This test tests all states of the smart home and all events that the smart home can receive. Only one event is not tested fully: *TcpRead*. This event contains a byte array representing the read from a tcp client. With klee, only a fixed size array can be made symbolic. Thus, the symbolic executer does not have access to all possible events. Nevertheless, in the smart home core, only the first byte of the *TcpRead* event is used for now, so it should not be an issue.

At this point only the core of the smart home is being tested. That is to say the part that processes events.

3.5 Possible further work

Right now, the tests deactivate bindings by making all bindings optional. This can surely be improved to make the code more readable. A possible solution we see is to make a better separation between the processing event functions and bindings' action calls. For example by creating another type of event which goes from the smart home core to the bindings. *process_event* would return such events, and they will be processed after. In this way, *process_event* can be tested without binding involvement. As future work, one could find a way to test the bindings. For instance, to test the GPIO binding, one can replace the physical memory map for the GPIO controller by a memory space allocated on the stack or heap. This way, we can test the GPIO controller outside of a Raspberry Pi at a fine grain level and check that the bits have been written at the right place.

4 Controlling the General Purpose Input /Output (GPIO) of the Raspberry Pi

4.1 Ways to control the Raspberry Pi GPIO

The Raspberry Pi GPIO are digital signal pins which can be used both as inputs and outputs. We have found three ways to control them:

1. Through the system file at `/sys/class/gpio` [39]. It has been the primary way of doing it in user-mode (legacy way) , however it has been deprecated since version 4.8 of the Linux kernel. Furthermore, it relies on abstractions and functions of the kernel that will have to be added to our trusted computing base. Therefore, we will need to either assume it to be correct or prove it formally. We think that proving it formally is more complex than required, since a system file is a very general abstraction. Moreover, we will also need to prove the implementation of the GPIO access in respect to system file and to the given way of interact with it. In addition to that using a "complex" abstraction would have been more prone to implementation error or even use error. Finally, a minimal OS is less likely to offer this abstraction.

2. Through the character device at `/dev/gpiochip` [39]. Like the system file it is an abstraction provided by the Linux kernel (since version 4.8). It has the same drawback in respect to the abstractions and functions given by the OS we talked about for the system file.

3. Through GPIO memory mapped registers [34, Chapter 6]. If we use the program in an OS, where there are no virtual addresses that map to the physical addresses where the registers controlling the GPIO are. In that case, we will have to use abstractions provided by the kernel to map those GPIO memory mapped registers to the virtual address space, or implement a driver that offers an API for us to interact with those registers.

   If the virtual address space includes virtual addresses that map to the physical addresses where the registers controlling the GPIO are. Then, we are capable of interacting directly with those GPIO memory mapped registers, without any abstractions/functions/codes from the OS that we have to put in our trusted computing base.

   Concerning the first case of virtual address space, we will discuss later that this solution is less complex to interact with, making it less prone to errors (errors of use from us and errors of implementation, so from codes that we did not write). Thus, the problem linked with abstractions discussed before is reduced compared to the other way discussed. Moreover, we can hope, later on, to execute the program in an OS where the GPIO memory mapped registers will be directly accessible from the virtual address space of the program.

4.2 Controlling the GPIO through memory mapped registers

4.2.1 Controlling the GPIO through memory mapped registers

To begin, controlling the GPIO through the memory mapped registers is straightforward once we can write to those registers. Indeed, to give instructions to the GPIO pins, we need to write the proper bits to the right registers. To know what to write and to which registers, we just need to read the description of the peripheral documentation[31].

4.2.2 Mapping GPIO registers to virtual address space

Writing to the GPIO registers is less straightforward in a virtual address space where no virtual memory address maps to the physical memory address of the registers.

In the case where there is such addresses, we can simply access the registers by accessing the virtual addresses that maps to the physical memory addresses of the registers.

In the other case where there is no such address we need to generate them. For that we utilize a character device file that represents memory as a file, such as `/dev/mem` or `/dev/gpiomem`. For us, as we need to access the memory location where the registers of the GPIO are, we are going to use `/dev/gpiomem`. Then we use the `mmap` function provided by the OS on that device driver file. It maps the physical memory associated to the GPIO registers to the virtual address space of the program. The `mmap` operation is not a complex one, it works with the page table of the process[32, Chapter 15]. If we use an OS with virtual address space then we already assumed that the virtual address space abstraction works correctly. Thus, we don't add that much to our trusted computing base. Moreover, because of the low complexity, what we add to the trusted computing base without formal verification from us is reasonable, as it is quite likely to be correctly implemented. In addition to that, the `mmap` is simple to use, thus reduces a possibility of error when calling it. The others abstractions used for the other ways of controlling the GPIO are more complex to use[39].

4.3 Why not write a driver

Using Rust to write a Linux kernel driver is in development but is still, as of today, not supported[33].

However, there exists projects to write Linux drivers[37][38]. But it is quite new and we would need to assume that those framework have no error in their implementation, or we would have to prove that the framework has no error.

Regardless, ultimately we would like to run the program in a minimal OS, in which there are virtual addresses that map to the physical addresses where the registers controlling the GPIO are. And the solution discussed in detail can be easily adapted to it, while removing the uses of abstractions given by the OS, and therefore the drawbacks associated with them. This implementation will retain the same benefits as the implementation of a driver.

4.4 Chosen way of controlling the GPIO of the Raspberry Pi

From those 3 alternatives to control the GPIO, we have found that controlling the GPIO directly with its memory mapped registers is the best solution for us. Indeed, even if the solution is less general, we know which hardware we are going to use. And it confers us all the benefits we have discussed in contrast to the other alternatives.

4.5 Implementation in Rust

From the BCM2711 documentation [31, Chapter 5] we know we will control the GPIO depending on their BCM pin number. We also know that BCM2711 offer 58 GPIO pins, but that the Raspberry Pi 4 makes only 28 of those pins available from the pin header[40, Chapter 5]. To make sure that the right BCM pin number is associated to the right GPIO pin, and to only allow the users of this GPIO controller to use the available pin of the Raspberry Pi. We provide an enum type with its variants being the available GPIO pins. It makes the use of our GPIO simpler and safer. Note that we did not provide the GPIO 0 and 1 as they are reserved for HAT ID EEPROM[40].

Those available GPIO pins can then be transformed into a GPIO pin abstraction type. For simplicity of use and understanding, only one GPIO pin abstraction per real GPIO pin can exist at a given time (this is what we will refer to later on with singleton object).

To interact with the GPIO one needs to have a GPIO pin abstraction and a mutable GPIO controller abstraction. This choice has been made so that at all times, only one GPIO can be interacted with. Again for simplicity of use and reasoning. To respect this statement, the GPIO controller abstraction also needs to be a singleton object.

To implements those singleton objects, we tried to use the reference lifetime of Rust. In such a way, that we would obtain the guarantee if the program compiles that only one reference to a singleton object exists at each point in time. The idea was to provide a function that would return a mutable reference of the requested object singleton. Because Rust guarantee at each point in time there is only one mutable reference to an object, our singleton object abstraction would be guaranteed at compile time. However, we could not manage to do it without unsafe code which would remove the guarantee issued by the compiler. Instead the function that returns the requested singleton object panics if the requested singleton object is still owned in the program when asked. Thus when testing it will panic if the singleton object idea is violated, and therefore to simplify our coding and understanding we can assume it is valid.

We only implemented GPIO output functionality, in order to provide a code as small and simple as possible. This is sufficient as it is the only functionality we need for our smart home implementation.

We did not make the access to the GPIO pin concurrent safe. This choice was taken to put even less amount of code, hardware functionalities (e.g. atomic action) , abstractions to our trusted computing base. Moreover, it also allows the code to be as simple as possible. The fact that it is not concurrent safe additionally requires us (if we do not transform it to be concurrent safe) to make all access to GPIO in one thread. programming and safety guaranty in one thread is much more simple.

As you can notice, our primary thought while implementing access to the GPIO pins was to do it as simple as possible. For the implementation but also for the use of it. Thus respecting a well-known important security principle:

- Economy of mechanisms [41]: "keep the design as simple and small as possible".

This principle allows us to put as little as possible in our trusted computing base. Indeed, to keep it simple, we need to use the minimum number of abstractions and amounts of code. The principle also allows us to verify manually line by line all the code that we wrote (it is reasonable as the code size is small and simple) and that is in our trusted computing base. Therefore, it allows us to verify a bigger proportion of our trusted computing base. Other verification are also easier as less abstractions are used. It on tops makes us less likely to commit implementation errors. I will not enumerate all the benefits of this principle, but note that there are big benefits in computer systems, in security [36] and consequently for the implementation of the smart home.

5 Possible further work

One could implement the second part of the selected smallest set of features. That is to say a camera and a front door speaker.

One could also implement more bindings for more commonly used protocols such as KNX or MQTT for instance. Hardware design for KNX can be found on this project's drive [30]

In this implementation we trusted that the relay and the pin wouldn't fail (the hardware). To be more confident, it is maybe possible to do a "defense in depth". The idea would be to make a parallel wiring with multiple relays for each before wiring (e.g. for a door lock). And when in a situation where the relay lets through state is important (e.g the elderly falls, people need to enter to help) query that the multiple "parallel" relays let the current pass. That could enable a relay failure resistant door lock.

A Appendix

A.1 User stories

- As an elderly user, I want to open the smart door remotely so that I don't have to move to the door.
- As an elderly user who has had a fall, I want to send an emergency alert to health assistants so that I can get help fast.
- As an elderly user who has a detectable emergency health issue, I want to be warned so that I can get treatment or help for this issue.
- As an elderly user who has a detectable emergency health issue, I want to be warned so that I can get treatment or help for this issue.
- As an elderly user, I want the smart home system I am using to be low-maintenance so that I do not have to understand how it works and that I don't give up on it.
- As an elderly person who needs emergency help, I want a health assistant to be able to open my door so that they can help me.

A.2 Smart devices considered

Table 1: Comparison of smart devices (score between -5 and 5)

	Health/safety	Independence and psychological comfort	Living conditions	Comments
Connected door	5	3	1	Allow people to enter without the elderly participation, or with in some case with unconscious elderly
Air quality sensor	1	0	0	
Connected camera	5	3	2	Allow a health assistant to check if everything is ok.
Smart camera	5	3	0	Detect falls, intrusion, fire. All the danger
Connected Speaker	0	3	5	Allow to notify the elderly that they are going to be watched.
Connected Tv	0	3	5	Allow to display front door lock, and to visual call if we have connected speaker
Electronic bed	0	4	3	
Smart lighting	5	5	1	Allow easy movement during light, and less movement when it's going dark.
Smart vacuum	3	5	5	Less movement is safer
Voice assistant	0	5	5	
Smart shutter	3	5	5	Less movement is safer. Forces the elderly to see daylight every day and fresh air.
Monitoring sensor for activity	4	0	0	Could be combine to produce less movement and to check everything is ok
Smart plug	2	5	5	
Professional monitoring	5	-2	0	Check that everything is ok every time there is a doubt
Smart heater and cooling system	5	3	5	Less movement and optimal temperature is safer and healthier

References

[1] URL: https://www.openhab.org/docs/developer/bindings/ (visited on 06/20/2021).

[2] URL: https://www.conrad.ch/fr/p/raspberry-pi-4-b-4-gb-4-x-1-5-ghz-raspberry-pi-2138865.html (visited on 06/20/2021).

[3] URL: https://www.conrad.ch/fr/p/carte-microsdxc-sandisk-extreme-pro-sdsqxcy-064g-gn6ma-64-gb-1780915.html (visited on 06/20/2021).

[4] URL: https://www.conrad.ch/fr/p/sygonix-sy-3386826-gache-electrique-1693413.html (visited on 06/20/2021).

[5] URL: https://www.conrad.ch/fr/p/module-relais-makerfactory-mf-6402393-1-pc-s-2134131.html (visited on 06/20/2021).

[6] URL: https://www.conrad.ch/fr/p/cable-de-cavalier-renkforce-jkff406-rf-4599684-arduino-banana-pi-raspberry-pi-40x-pontage-filaire-de-prise-femelle-2299842.html (visited on 06/20/2021).

[7] URL: https://www.conrad.ch/fr/p/bloc-d-alimentation-a-tension-fixe-mean-well-sga60e12-p1j-sortie-12-v-dc-5000-ma-60-w-1-pc-s-1439216.html (visited on 06/20/2021).

[8] URL: https://www.conrad.ch/fr/p/raspberry-pi-zero-wh-512-mo-1667360.html (visited on 06/20/2021).

[9] URL: https://github.com/tberkane/doge-home-app (visited on 06/20/2021).

[10] URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (visited on 06/20/2021).

[11] URL: https://github.com/dimitrisraptis96/fall-detection-app (visited on 06/20/2021).

[12] URL: https://github.com/dimitrisraptis96/fall-detection-app (visited on 06/20/2021).

[13] URL: https://ngrok.com/ (visited on 06/20/2021).

[14] URL: https://mitls.org/ (visited on 06/20/2021).

[15] URL: http://plv.mpi-sws.org/rustbelt/ (visited on 06/20/2021).

[16] URL: https://doc.rust-lang.org/rust-by-example/testing.html (visited on 06/20/2021).

[17] URL: https://altsysrq.github.io/proptest-book/proptest/vs-quickcheck.html (visited on 06/20/2021).

[18] URL: https://github.com/GaloisInc/crucible/tree/master/crux-mir (visited on 06/20/2021).

[19] URL: https://github.com/dwrensha/seer (visited on 06/20/2021).

[20] URL: `https://github.com/facebookexperimental/MIRAI` (visited on 06/20/2021).

[21] URL: `https://github.com/seahorn/seahorn` (visited on 06/20/2021).

[22] URL: `http://klee.github.io/` (visited on 06/20/2021).

[23] URL: `http://llvm.org/` (visited on 06/20/2021).

[24] URL: `https://github.com/project-oak/rust-verification-tools` (visited on 06/20/2021).

[25] URL: `https://alastairreid.github.io/automatic-rust-verification-tools-2021/` (visited on 06/20/2021).

[26] URL: `https://github.com/BajacDev/rust-klee-docker` (visited on 06/20/2021).

[27] URL: `https://gitlab.henriktjader.com/pln/cargo-klee` (visited on 06/20/2021).

[28] URL: `https://gitlab.henriktjader.com/pln/klee-sys.git` (visited on 06/20/2021).

[29] URL: `https://gitlab.henriktjader.com/pln/panic_klee.git` (visited on 06/20/2021).

[30] URL: `https://drive.google.com/drive/u/1/folders/0AEd1-eHcG1kDUk9PVA` (visited on 06/20/2021).

[31] *BCM2711 ARM Peripherals*. 3rd release. © 2012 Broadcom Europe Ltd., 2020 Raspberry Pi (Trading) Ltd. Oct. 16, 2020. URL: `http://datasheets.raspberrypi.org/bcm2711/bcm2711-peripherals.pdf`.

[32] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. 3rd edition. O'Reilly Media, Mar. 1, 2005. ISBN: 978-0596005900.

[33] Sergio De Simone. *Using Rust to Write Safe and Correct Linux Kernel Drivers*. URL: `https://www.infoq.com/news/2021/04/rust-linux-kernel-development` (visited on 06/22/2021).

[34] Harry Fairhead. *Raspberry Pi IoT In C*. I/O Press, Oct. 2, 2016. ISBN: 978-1871962468.

[35] Li Jiang, Da-You Liu, and Bo Yang. "Smart home research". In: *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.04EX826)*. Vol. 2. 2004, 659–663 vol.2. DOI: `10.1109/ICMLC.2004.1382266`.

[36] *KISS principle*. URL: `https://en.wikipedia.org/wiki/KISS_principle` (visited on 06/22/2021).

[37] Zhuohua Li et al. *linux-kernel-module-rust*. URL: `https://github.com/lizhuohua/linux-kernel-module-rust` (visited on 06/22/2021).

[38] Zhuohua Li et al. “Securing the Device Drivers of Your Embedded Systems: Framework and Prototype”. In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. ARES ’19. Canterbury, CA, United Kingdom: Association for Computing Machinery, 2019. ISBN: 9781450371643. DOI: `10.1145/3339252.3340506`. URL: `https://doi.org/10.1145/3339252.3340506`.

[39] Sergio Prado. *Linux kernel GPIO user space interface*. URL: `https://embeddedbits.org/new-linux-kernel-gpio-user-space-interface` (visited on 06/16/2021).

[40] *Raspberry Pi 4 Model B Datasheet*. 1st release. © 2019 Raspberry Pi (Trading) Ltd. All rights reserved. June 21, 2019. URL: `https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0_preliminary.pdf`.

[41] J.H. Saltzer and M.D. Schroeder. “The protection of information in computer systems”. In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. ISSN: 1558-2256. DOI: `10.1109/PROC.1975.9939`.