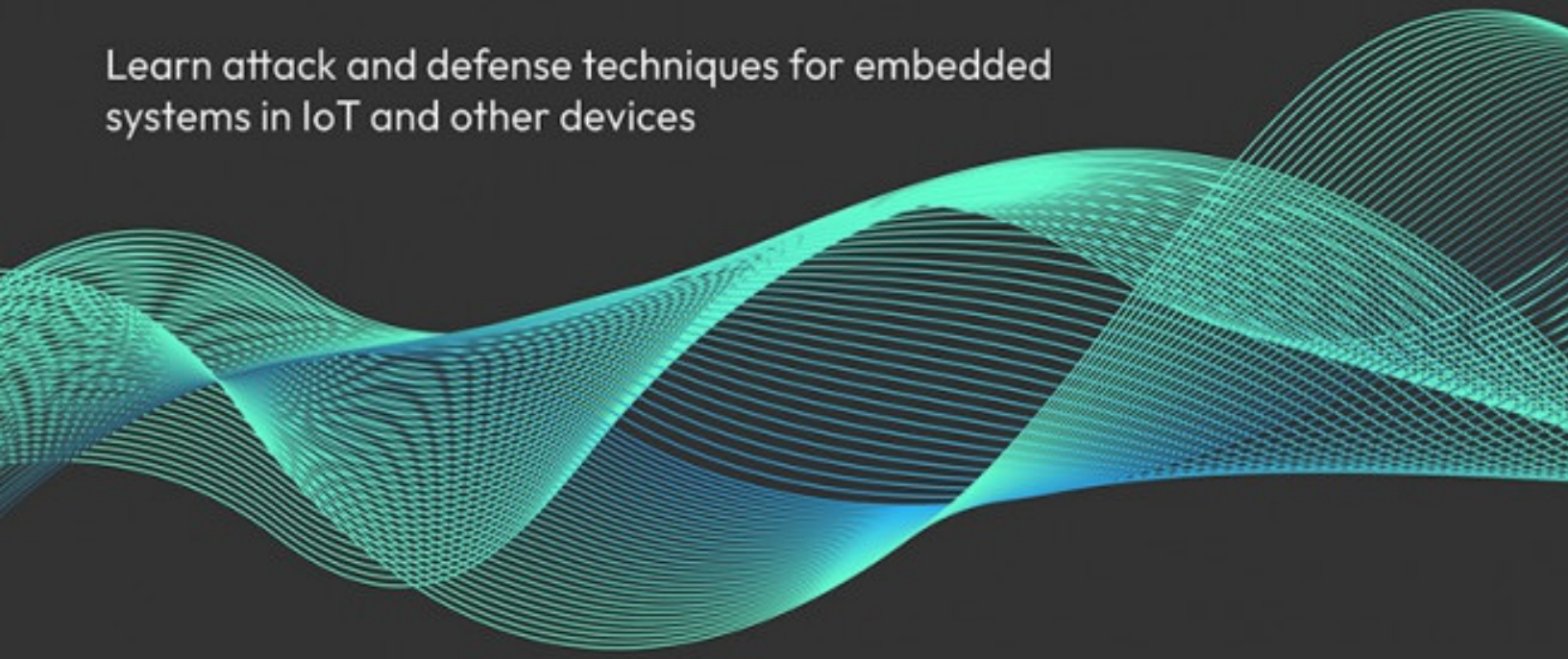
EXPERT INSIGHT

Practical Hardware Pentesting

Learn attack and defense techniques for embedded systems in IoT and other devices



Second Edition

Jean-Georges Valle

‹packt›

Contents

Practical Hardware Pentesting, Second Edition: Learn attack and defense techniques for embedded systems in IoT and other devices

Welcome to Packt Early Access . We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

Feedback

We are constantly looking at improving our content, so what could be better than listening to what you as a reader have to say? Your feedback is important to us and we will do our best to incorporate it. Could you take two mins to fill out the feedback form for this book and let us know what your thoughts are about it? Here's the link:
https://packt.link/HardwarePentesting2E .

Thank you in advance.

Embedded systems, in the broadest definition of the term, are all around us in our everyday lives (examples being our phones, our routers, our watches, our microwaves, and more). They all have a small computer inside them and take care of very critical aspects of our lives, and also collect and protect data that is very critical to us. Sadly, the embedded system industry is lagging behind the usual computing industry in terms of security. In the last 10 years, we have seen examples of how this lack of security in these kinds of systems can lead to very tangible impacts on the real world (for example, the Mirai botnet; a slew of viruses targeting the industrial world's embedded systems across a large number of vendors (stuxnet, darkenergy, industroyer, pipedream/incontroller, etc.) ; a wave of attacks against routers; some countries stealing other countries' drones by spoofing the **Global Positioning System** (**GPS**); and so on). This is why it is very important to train more and more people on how to find problems in these kinds of systems, not only because the problems are already here but also because there will be more and more such systems, and their ever-growing number will manage more and more crucial aspects of our lives (think about autonomous vehicles; drone delivery; robots to assist the elderly; and so on).

Helping you start with assessing the security of these kinds of systems is the first goal of this book. The second goal of this book is that you have fun while you learn because testing these kinds of systems is going to be interesting, and I take great pleasure in making the learning process enjoyable for you. You may ask yourself: *How is it going to be fun for me?* For me, it is

because you are messing with the most trusted part of the system: the hardware. Not only you are messing with the most fundamental elements of the system, but you also are in direct contact with it; you will be soldering, drilling, scrapping, and touching the system to pop a shell! You will not only code to compromise your target system, but (hopefully rarely) the blood, sweat, and tears will not be figurative!

In this chapter, you will learn how to set up your lab, from a simple, low investment suitable for learning at home up to a professional testing environment. This chapter will get you up to speed on how to invest your money efficiently to achieve results and, most importantly, how not to kill yourself on the job.

The following topics will be covered in this chapter:

- The basic things you will need to get started
- The different types of (common) tools available for your labs, what to get, and at which point
- The approach to acquiring test equipment, and the difference between a company and a home lab
- Basic items you will want in a lab, what they are, what are their uses, and the approach to setting up a lab
- Examples of ramping up your lab: basic, medium, and professional labs

Prerequisites – the basics you will need

Before going into the things you will need to buy, let's have a look at the basics you will need to go through our joint exploration of an unknown system (a Furby), and start working on your own systems.

Languages

To be able to script activities and interact automatically with most systems, you will need to be familiar with at least one high-level programming or scripting language (I will use Python for the examples in this book, but any other scripting language such as Perl, Bash, PowerShell, and more will also work) and one low-level programming language to write your own firmware and customize the examples. I will also use C (on the attack platform) since it is the most popular programming language for embedded systems, but any language that has a compiler for your target system will work.

Hardware-related skills

You will need to learn actual, manual skills that are not purely knowledge-based; the main obstacles people fear when starting hardware hacking are soldering and understanding of basic electronics concepts. For both of these skills, you can approach them in a knowledge-based way: learn about Ohm's law; the physics of semiconductors; what is an eutectic mixture and temperature; and all of the theoretical background. To be honest, I would not recommend approaching these skills like that.

Of course, you may need the knowledge down the road, but don't start with this. Solder things; make **light-emitting diodes** (**LEDs**) blink; learn how to use transistors as switches. In short: do things, accept failure, and learn from it; burning a transistor will cost you a few cents but you will not repeat your error; burning your fingers will hurt but this will heal in a few days (there are

safety instructions in the book—read them very carefully). You have far more chances to lose your motivation by learning a lot of laws and formulas while never using them than by having a problem, finding the correct formula, and solving your problem with it!

System configuration

Having a nice desktop computer will really improve your experience in the lab. Even if, in today's world, people tend to use laptops more and more, this can prove to be a challenge when you are attacking hardware. A laptop will not block you from attacking hardware, but a desktop will definitely make your life a lot easier.

A laptop's main challenge will be the very limited physical interfaces available on it (still, you can work with it).

You don't need a powerful computer to start with (I use a 7-year-old i7: nothing fancy), but really pay attention to the interfaces. It is very common for me to use 5-6 **Universal Serial Bus** (**USB**) ports when I am attacking hardware; for example, when operating on any embedded system, I typically have the following things attached to my computer (not even counting my convenience peripherals such as keyboard, mouse, headset, having a dual-screen setup, and so on):

- USB:
  - - A bus pirate
  - - An OpenBench logic analyzer
  - - One or two USB to **Universal Asynchronous Receiver/Transmitter** (**UART**) bridges
  - - A **microcontroller unit** (**MCU**) board
  - - A function generator
  - - My programmable power supply
- Ethernet:
  - - My internet connection
  - - My oscilloscope

Good luck doing that with a laptop without using an external USB hub,

especially when these hubs can interfere with the functionality of some peripherals (for example, the USB-UART bridges I use tend to become unstable if used over a USB hub—using a good-quality powered USB hub can help).

One of the main contention points is the operating system. I use Linux, but using a Windows-based machine (especially if you use the **Windows Subsystem for Linux** (**WSL**) for anything but access hardware peripherals) will not really limit you in the end. (I will base the examples in this book on Linux. If you don't want to install a machine with Linux, just run a **virtual machine** (**VM**) but be aware that some of the most popular and free virtualization software do not really support USB passthrough very well.)

Setting up a general lab

The setup of the lab itself is very important and will be quite determinant in terms of your ease of use and comfort in the lab. You will spend a lot of hours thinking and hacking in there, thus the room and its furniture will be quite important to your comfort. You will need to consider the following factors:

- **Your chair** : Invest in a good wheeled desk chair with easily movable arm support and good back and lumbar support. The race car seat-looking chairs targeted at gamers can be a good type to look into, but really pay attention to the armrests and the system that allows you to move them out of the way and set them to the desired height easily. More often than not, they will annoy you when using your soldering iron, but you will want them to support your arms when typing, for example.
- **Your work table** : Three factors are critical—the height of the table (so you don't kill your back when operating close to a **printed circuit board** (**PCB**), for example) and its surface. For the surface, I like clear colors (to be able to easily see a component that slipped, for example) with a slightly textured surface (so the components don't skid too far too easily). Also, the larger your work surface, the better it is to spread the inevitable clutter.
- **Shelving** : You will want to have shelving on top of your work table in

order to be able to have your instruments on top of your work area without them eating up the space available. I like to have the shelving approximately 50 cm higher than the surface of my work table in order to be able to easily manipulate the interface of the instruments and put back probes without having to stand up from my chair nor having to kill my neck when I look at waveforms or a specific knob or button.

- **Light** : Good and powerful lighting of your work area is crucial; not only you will be manipulating a variety of very small things (components, cables, connectors, and others), but it becomes even more important when operating under magnification (for example, for soldering). Pay attention to the intensity of your light but also its color temperature (more blue or more yellow, I prefer a balances 5000K daylight-like temperature), keeping in light that the bluer it is the higher the risks of it disturbing your sleep cycle.
- **Anti-static measures** : An anti-static mat is really practical to protect sensitive devices against electrostatic discharge. They come with a bracelet that ensures any electrostatic charge you may have built up is dissipated. It is also important to avoid flooring that will make you build up such charges (such as carpets).
- **Soldering mat:** Cheap silicon soldering mats can be bought for cheap on online stores. Get one, they avoid part slipping away easily and usually have integrated small sorting bins that are really usefull.

Safety

There are inherent risks linked with opening and interacting with live systems. Please read these carefully—safety first!

Please follow these safety tips at all times:

1. If there is a risk of electric shock, never ever do your tests alone and be sure to brief the person who is with you on how to quickly kill the power and react. Have emergency services' number preeminently displayed; a fire extinguisher that can be used on live electricity; first aid training; and so on.
2. Whenever your fingers or instruments go near a system, ensure it is either disconnected from the mains (that is, wall plug electricity—

110/220V (where **V** stands for **volts**)) or that you are physically isolated from the mains part of the board (for example, use silicon mats to isolate the dangerous part of the power section).

3. If a system is mains-powered, always, always use an insulation transformer.
4. Wear adequate clothing, remove jewelry and, if you are sporting long hair, always tie this up (which will prevent it from getting in the way).
5. If the system sports any kind of battery, insulate the battery rails appropriately (with electrically insulating sticky tape, for example). Some battery types are dangerous and can catch fire or explode if shorted. I really advise you to have a look at videos of shorted lithium-polymer batteries: you don't want this kind of catastrophic failure happening in your home, lab, or office.
6. You will work with sharp and hot tools and objects, so having a first aid kit available is always a good idea.
7. There is a debate about what is dangerous: voltage or current. Actually: energy kills, so both voltage and current can be dangerous. For example, you may have already survived a > 10 **kilovolt** (**kV**) electric shock from electrostatic discharge (the sparks you can feel when removing a pullover, for example), but 2,000 A at 1 V will char you to death, and people regularly get killed by mains power. The gist is, whether amps or voltage are present, treat it as dangerous.
8. Soldering equipment is very hot and will set things on fire if you are not cautious; always have a smoke detector in your lab, along with a fire extinguisher. Use the holder your soldering iron comes with (or buy one); they are usually shrouded to avoid contact with random objects.

Safety is of the utmost importance—there is no need for all the fancy test equipment we will now go through if there is no one left to operate it.

Approach to buying test equipment

These are my personal opinions and views. Especially regarding measurement equipment and tools, you will find a lot of heated argument about the different brands, models, and tools. Engineers tend to be reasonable but they are human beings, and there will be fanboys. You will find on different forums people with their opinions and the deeply rooted belief that what is working best for them is the best for anyone. The golden rule is the following:

- Get information upfront
- Make up your mind
- Be reasonable
- Get what works best for you

Home lab versus company lab

Some very important distinctions have to be made between your own personal laboratory equipment and what you use in a company laboratory. Not only will the money for the home lab come from your own pocket, but some options (such as renting) may not be realistic for a home lab. Additionally, a company lab is subject to the safety rules of a work environment. You should meet with your company's occupational safety manager in order to comply with the adequate regulations regarding the storage of hazardous or corrosive chemicals, ventilation/air extraction, handling of possible fire hazards, and so on (as a side note, this is a very practical and reasonable way to get out of this noisy open space we all love to hate).

Hacked equipment and Chinese copies

In a home lab, one of the best reminders of why you are doing the assessment is the fact that some instrument companies are suspected by the community of actually producing hackable instruments in order to boost their sales. And

their instruments get hacked. This is a reminder that there is a very real community (and not a fabled hacker hidden in their parents' cellar) that is going after electronic devices in order to get the most out of them, unlocking features that are normally paid for, and potentially costing money to the company that produces the instruments. From a hobbyist point of view, it may be not really legal, but it is a common practice for hobbyists to maximize their investment by modifying or hacking existing instruments.

Since legality and repeatability are key in a company laboratory, I would advise against hacking instruments in this context. If the current laboratory setup of your company is not enabling a test to take place, your company should have a budget to buy (or rent) the adequate instruments or be able to offset the cost to a client.

The same goes for Chinese copies of programmers and logic analyzers—you may not care about it in a private setting, but in a professional setting the lower quality can actually turn back to bite you. The gist is, as long as you are doing this as a hobby, the decision to hack your instruments is on you, but if you are doing this professionally, buy the real thing and get reimbursed, or bill your client.

Approaching instrument selection

Measurement instruments are like cars; it's all a question of balance

- The Italian sports car type—the luxury thing that will be able to do everything (short of cooking for you), which costs an insane amount of money and actually very few people can get the most out of. It may not be worth it in an assessment context unless you have a really specific need. If it is the case, it may be smarter to just rent the instrument. Brands that I classify in this category: Teledyne-LeCroy, Rohde & Schwartz, and high-end Keysight (formerly Agilent).
- The good-quality German car that is doing everything quite well. It may be a good investment if you are actually doing this a lot and need a reliable, solid instrument that will get you far for a long time. Brands that I classify in this category: mid-range Keysight, Tektronix, Yokogawa, and very high-end Siglent or Rigol.

- Le French car type—it's going to be doing almost the same thing that the German car does, for a fraction of the price, with a lot less style, and maybe for a shorter time. Brands that I classify in this category: mid-range Siglent or Rigol.
- The no-frills, cheap Japanese car—it's going to be efficient and cheap, get you from point A to point B, but you're not going to get a lot out of it on the speedway. Brands that I classify in this category: low-range Siglent or Rigol.
- The "el cheapo" Chinese car. It is cheap; it's a box with an engine and a driving wheel, but not much more. Also, don't have a crash in it: its safety is not so well engineered. Brands that I classify in this category: OWON.

And just as with a car, you can find very interesting second-hand deals! Don't underestimate second-hand instruments—a lot of renting companies sell their used equipment second-hand, and you can score pretty sweet deals like that. (My first oscilloscope was a second-hand 100 MHz-bandwidth Phillips, which I scored on eBay and used for 3 years without a problem.)

What to buy, what it does, and when to buy it

Here is a table of the main types of different instruments, what they are used for, and how much they are needed (0 being the highest priority):

Instrument	Description	Priority
Digital multimeter (DMM)	A DMM is a fundamental tool that allows you to measure voltage, resistance, and current intensity and also to check for continuity. Advanced models allow you to measure other values such as frequency, inductance, and capacitance.	0

Soldering iron	Just as with the DMM, a soldering iron is one of the pieces of equipment you will use the most. Directly go for a temperature-controlled one. This will allow you to make your own circuits, remove and exchange components, and more.	0
Bus pirate	This is a very useful multi-tool to interact with in-circuit buses- more on it in the in-circuit communication chapter: *Chapter 6 , Sniffing and Attacking the Most Common Protocols* .	0
Logic analyzer	A logic analyzer reads digital protocols and allows you to decode them in software later. This is extremely useful for spying on inter-chip communication, developing and debugging your custom tools, and more.	1
	An MCU platform you will get to know well and will learn to use efficiently. This will be	

MCU platform	very useful to send fake messages on buses, impersonate a chip, and pretty much interact programmatically with the target system's electric signals. We will go for a cheap and flexible one (the blue pill) later in the book.	1
JTAG adapter	*JTAG* (named after the Joint Test Action Group) is historically an interface to test the soldering of chips. It has been extended to offer chip-specific programming and debug interfaces and functions.	1
Oscilloscope	An oscilloscope allows you to measure voltage in function of time and trace the curve of this voltage. Current models can do additional measurements (frequency measurement, frequency spectrum, and more), trace voltages in function of another, decode digital protocols, and so on.	2
	A hot air station is an advanced version of a soldering iron. It is very practical to work with	

Hot air station	surface-mounted components since it will allow you to heat all leads and underlying pads of a component at once.	2
Lab power supply	Lab power supply comes in two main flavors: variable ones (where you can set a fixed output voltage and a maximum current limit manually) and programmable ones (where you can set the voltage and current limit programmatically). The first kind is all you need to start and do most of your work. The programmable ones are more advanced and, should you need one, you'll be knowledgeable enough to know it. I personally only have a manual one and have never needed a programmable one.	0 (var) / 3(prog)
FPGA platform	A field-programmable gate array (FPGA) is a programmable logic platform that allows you to do really fast and high throughput operations. This piece of equipment is among the more	4

advanced that you
should look into when
you have become more
familiar with procedural
programming or if you
have a specific need to
do something really fast.

DMM

The **DMM** is your principal tool—you will be using it all the time. I really mean all

DMM basics

Your DMM will come with a manual. Read it. Even if you have used a multimeter before, you have to know the basic characteristics of the tool you will be using.

If you have never used a multimeter, it should come with at least these functions:

- **Voltage measure** : This will measure the voltage difference between the two test leads. If your DMM doesn't have an auto-range function (like most entry-level meters), you will have to set the measuring range and set it to direct or alternating voltage.
- **Current measure** : This will measure the current (the amount of electricity) passing through the leads. Again, pay attention to the range. Most of the time, you will have to change the connector one of the leads is plugged into (from V to A; sometimes there is even a mA connector for lower ranges).
- **Resistance measure** : This will measure the resistance between leads by creating a known voltage between the leads and measuring the current that the resistance lets go through. Again, pay attention to the range. The resistance is inferred by using Ohm's law:

Voltage (in volts: V) = Resistance (in Ohms: Ω) x Current (in amperes: A).

- **Continuity test** : When the test leads are connected with a negligible resistance, the multimeter will beep. A fast continuity test will really make your life easier and a slow response continuity beep is a very common downside of very cheap DMMs.

TIP

Never use the continuity measurement or resistance measurement modes on a live circuit—not only can the reading be false but you can also damage your DMM!

Getting your workhorse

You will be able to find a curated list of DMMs with their characteristics and comparison on the *EEVblog* forum. (I also warmly encourage you to watch the videos from EEVblog—Dave Jones' style isn't for everybody, but I personally like it a lot and his videos are always very educative.)

The list can be found here:
https://www.eevblog.com/forum/testgear/multimeter-spreadsheet/ .

I really don't recommend going for a very cheap Chinese DMM, nor can I point you toward an exact model since it may not be valid in a few months.

The elements to pay attention to when selecting a DMM (in order of priority) are the following:

- The DMM really should be of a safety rating compatible with what you are measuring (at least CAT III, as you will be measuring main voltages at some point) and the probes should be really sharp. In a worst-case scenario, you can always buy replacement probes.
- Bandwidth, precision (the number of displayed digits), and the count numbers should be as high as your budget allows.
- The speed of the continuity test (try to find review videos)—you want it to be as fast as possible.
- The available ranges—you really want as wide a range of measurement as possible, both of **alternating current** (**AC**) and **direct current** (**DC**) (it should range from millivolts to at least 1,000 volts; from a few

ohms to a few dozens of megaohms; and from a few microamps to 10 or 20 amps for current).
- The input impedance (that is, the capability of the meter to read the voltage from a circuit without disturbing the circuit)—you want at the very least 10 megaohms (the higher the better).
- A serviceable fuse that you can replace easily.
- Good back-lighting to help with screen visibility when you are working late.
- The battery lifetime is also a common default that plagues the cheap chinese DMMs, having to stop your test to run to the closest shop to buy batteries can be annoying.

Soldering tools

Get a good temperature-controlled soldering iron with widely available replacement tips. Again, it is desirable to have a good workhorse and a lower-quality secondary iron (you will very rapidly be confronted with the necessity to rework surface mount parts; it is often tricky with a single iron and very often results in damaged PCB pads). The temperature control is very important since you will be confronted with leaded and unleaded solder, which have a different melting temperature; different-sized components with their own thermal mass (that is, how much heat does the component source from your iron before getting hot); and so on (get both irons with temperature control; the secondary doesn't need to be as precise as the main one). Some additional supplies are also extremely useful, as listed here:

- **Liquid and tacky flux** : This allows the melted solder to flow much more easily on the leads and pads. You will be constantly removing and re-soldering parts from PCBs, and flux will be helping you tremendously, especially for su **rface-mounted device** (**SMD**) parts. I am partial to Kingbo RMA-218 for tacky flux.
- **Soldering wick** : This is an invaluable tool to remove excess solder and clean PCB pads before soldering back a part. I was always kind of disappointed with solder wick… until the day I met chemtronic’s solderwick. The best one around, end of story.
- **Fluxed, leaded solder** : Get two different thicknesses, one in the 0.5 mm range and the other one as thin as you can get for SMD rework. You

will find leaded solder a lot easier to work with as it melts at lower temperatures, flows better, is much easier to wick out, and allows you to drown unleaded solder on multi-leaded chips to remove them. Since unleaded solder has a lower melting temperature, it is tricky to keep multiple leads in a nice melted blob of solder on all leads to remove it. Alloying the unleaded solder with additional leaded solder will help you a lot with this.

- **A third hand** : Yes—this tool's name sounds strange but it is a common tool. It is a heavy-based tool with two (or more) springy pincers that will hold components in place while you are soldering. To get how it is helpful, just imagine yourself soldering, with a soldering iron in one hand and the solder wire in the other. How would you hold parts or wires in place? These are really small, very light things that can move under the smallest shock and tend to do this at the worst moment possible.
- **Tips** : When you select your iron, try to find one for which the tips are reasonably cheap for different shapes; you will find the default conical tip that most irons come with to be actually impractical compared to a truncated cone.
- **Tweezers** : A soldering iron will get too hot for your grubby little fingers very fast. Having a nice set of cheap tweezers with different tip shapes will be very helpful to hold and manipulate small components.
- **Side cutters** : Flush side cutters are very useful to cut component leads very close to the PCB.
- **A PCB holder** : This will allow you to hold firmly a PCB (and orient it easily) while you work on it.

Logic analyzer

Here, there are two distinct ways, either open source software-based (sigrok) or proprietary ones (there are plenty, but Saleae is well known as being easy to use). Saleae hardware is, in my opinion, a little bit expensive for the punch they pack but it is balanced by very good software. It is possible to find Chinese copies of some of their (either older or smaller) models, but I would refer to the excerpt on knock-offs at the beginning of the chapter. Sigrok is compatible with a very wide list of hardware (you can find it here: https://sigrok.org/wiki/Supported_hardware). I personally use both: an

OpenBench Logic Sniffer (by dangerous prototypes) with sigrok at home, and Saleae at work.

Here is what to look for in a logic analyzer:

- **Sample speed** : This is the speed at which the analyzer samples the signal and determines the maximum speed of signal you can read accurately. The Nyquist criterion tells us that to read a signal accurately, you have to sample it at least at twice the speed of the signal.
- **The number of inputs** : The higher the better, but you can cover a very large percentage of buses with the basic 8-channel analyzers.
- **The input protection** : You may plug a probe on the wrong thing; you may accidentally burn a test system when fiddling with wires; your soldering iron may be badly grounded; and more
- **The input impedance** : Similar to the DMMs—at the very least, 10 megaohms.

Bus pirate

Easy—there is only one. There is a debate about which version to use (v4 can be buggy sometimes and is not always working with flashrom, so go for v3). The bus pirate is a tool that will allow you to interact and play with the most common protocols used to talk with chips.

MCU platform

The MCU platform will be the most controversial piece on the forums and on the internet in general.

I strongly recommend getting familiar with a vendor platform in the **Advanced RISC Machine** (**ARM**) family because of these factors:

1. The ARM architecture will be a very common target.
2. It is widely supported in term of compilers and debuggers with open source toolchains (GCC, OpenOCD, GDB, and so on).
3. Development boards are very cheap, plentiful, easy to find, and quite complete (if you read this after the COVID-chip-mageddon that is).

4. You can find screaming fast platforms for quite a cheap price.
5. Packages with a large number of very fast I/O are very common.
6. The necessary passive components to support the MCU can be quite low.

I am very partial to the STM32 family from STMicroelectronics. It may have its quirks, but the development boards are incredibly cheap. Some quite capable MCUs can be found mounted on cheap Chinese boards, in the 4 USD range (delivered) on popular websites (eBay, AliExpress, amazon and so on) offering a ton of I/Os and quite decent hardware peripheral. A few bucks more will get you an official board, which includes a programmer (that can be used to program the cheap ones quite easily). This is my personal opinion and mainly comes from the fact that these cheap development boards were among the first ones I had access to and, hence, I learned to use the quirks and features of the family quite well.

Plenty of other vendors (Texas Instruments, Cypress, NXP, and so on) offer quite comparable boards in the same price range. My main advice would be: choose a vendor and a family, get well acquainted to it, and stick with it. The chances are that you'll be able to select the family member with the speed and peripheral set that will fit your needs best when you have a specific requirement set.

JTAG adapter

JTAG, to start with, is an interface that was designed to test the soldering of integrated circuits. It was designed as a shift register that was able to activate all the leads of a CPU in order to be able to test the electrical connections. The basic design of JTAG was conceived to allow for the daisy-chaining of chips in order to have a single chain that could be leveraged to test a board. It was later enriched with CPU-specific features (that are not well standardized) in order to allow for in-circuit debugging and programming. It can be very useful for your own developments or to get access to the internal states of a chip if it is not disabled in production.

JTAG is based on a (minimum) four-wire bus (data in, data out, test, and clock). This bus is piloting a state machine in each target chip. (JTAG will be

covered in more depth in *Chapter 10 , Accessing the Debug Interfaces* .)

Oscilloscope

An oscilloscope will be a very useful tool for exploring signals and probing different lines. Basically, an oscilloscope will allow you to visualize a voltage in function of time. To get a good grip on the basic operation of an oscilloscope, please refer to Tektronix's guide *XYZs of Oscilloscopes* and read your oscilloscope manual from front to back.

Selecting your oscilloscope is almost easy—the baseline is that you want to get the most bandwidth and the most memory size for your budget. The question of whether to select a two-channel or a four-channel oscilloscope is very common. As usual, it boils down to a tradeoff. If you can get a four-channel with a bandwidth of 100 MHz or more within your budget, get it. A four-channel oscilloscope is very useful if you are exploring systems where more analog electronics are used and where you want to correlate an event's occurrence relative to another event.

Before taking your decision, it is really important that you watch test videos and, if possible, teardowns to compare the usability of your different candidates and the possibilities of repairing them in the case of problems. Do not underestimate repairability, I broke the screen of a 500 USD scope and I was really happy to be able to fix it with a 30 USD Chinese screen.

The bandwidth

The bandwidth of an oscilloscope is actually not equal to the maximal speed you will be able to measure. It is what is called a -3 **decibel** (**dB**) bandwidth. A -3 dB bandwidth is the frequency at which the instrument will measure a signal at half of its actual power.

This means that a 100 MHz-bandwidth oscilloscope will measure a 100 MHz, 1 V peak-to-peak p sine wave as a 0.7 V peak-to-peak signal!

To accurately read a sine wave (that is, at its actual voltage level), you will need at least three times the bandwidth of the signal.

Bandwidth is the characteristic of an oscilloscope with the most impact on the buying price. Take what the maximal and usual frequencies that you need to measure will be and make your decision accordingly (a 50MHz oscilloscope is good enough for a start but be prepared for it not to be sufficient after a few years. I still use my 100MHz oscilloscope in most cases (and I am not sweating bullets if I have to take it with me on a plane)).

Regarding the number of channels, it is very simple: the more channels you have, the better it is. Take into account in your decision that, most of the time, you will need one or two channels; measuring three and more signals is not something you will need every day, but you will be happy to have it when you need it.

The probes

There are two main types of probes: active and passive. To make it simple, you can only use passive probes under 350MHz (for higher speed, you will need active probes). Passive probes are quite cheap and come with a manual switch between different "damping ratios" that can be taken into account in the oscilloscope's interface. The probes are really important, same as the DMMs; you will want very sharp probes with a wire grabber. Good-quality probes are quite common with oscilloscopes. Don't forget to compensate your probes—the procedure should be described in your scope's manual.

Display

Most modern oscilloscopes come with additional display functions, such as **Fast Fourier Transform** (**FFT**), which allows you to see the signal in the frequency domain instead of the usual time domain); XY display (which allows you to see the signal on a channel in function of another channel); and X/Sin(X) (read Chris Rehorn's excellent paper *Sin(x)/x Interpolation: An Important Aspect of Proper Oscilloscope Measurements* and about the Nyquist-Shannon Signal sampling theorem).

Interfaces

It is very common to find network (Ethernet) remote commands and display; **Video Graphics Array** (**VGA**) output; USB storage of measured waveforms. This can be very useful to display waveforms on your computer or extract the samples from a measurement for later processing.

References

Just as with DMM, a list is maintained on the EEVblog forum: https://www.eevblog.com/forum/testgear/digital-oscilloscope-comparison-chart/

Hot air gun

A hot air gun shoots hot air at a controllable temperature and flow rate. This is very practical to solder or unsolder surface-mounted components. Some accessories and consumables are inseparable companions to an hot air gun: solder paste (to tin your pads, this can be deposed pad by pad with a toothpick) and Kapton tape (this is a type of heat-resistant sticky tape that can be used to protect components next to the one you are soldering or desoldering). I would recommend using leaded solder paste but this can be tricky to get in Europe or the US. The use of a hot air gun requires practice to be efficient and I would recommend watching technique videos and train on junk/broken boards before going at it on an important PCB.

Here are the things that you have to look for in pretty much all of the hot air stations you will find:

- Regulated temperature
- Regulated airflow
- Replaceable air gun head (to be able to have thin or wide flows; it can also be interesting to replace the head with a square one for bigger **quad-flat packages** (**QFPs**) or **quad-flat no-leads packages** (**QFNs**).

FPGA platform

FPGAs are really practical for fast logic processing. Their main downside is

that most of them require a proprietary programming and synthesis (the FPGA lingo for compilation). At the time of writing of this book, only the Lattice iCE40 had an open source development tool chain available (and support for the Xilinx 7 series is supposed to be coming up soon). Most of the proprietary environments are quite expensive if you want to cover most of the chips of the vendor, but some development kits come with a development environment limited to the chip that is on the board. I personally use an Artix-7 Arty board that I was trained on by Toothless Consulting's Dmitry Nedospasov, and I am very happy with it.

Vendor

A few vendors share most of the FPGA market: Xilinx; Intel (who acquired Altera); Lattice; and Microsemi (who acquired Actel). As for MCUs, most of them are almost equivalent (short of their development environments); depending on the time you are buying, just take the best development board you can find and stick to the vendor.

Language

A very common question is the language to develop with, being Verilog or VHDL. Verilog tends to be more common in the US, while VHDL is more common in Europe. The most important part is that both languages are equivalent; you can achieve exactly the same results and it is more a matter of taste. From my point of view, I tend to find VHDL is a bit more descriptive but as a downside, it requires more boilerplate code. I personally prefer Verilog since it is terser and easier to find examples for.

Lab power supply

Your lab power supply will allow you to power up your circuits and your target system. Some very practical features you really want on your supply are listed here:

- **Current limitation** : This will allow you to prevent things from burning when you are messing with the circuitry. I usually measure the current

consumption of the circuit in a normal context (over an hour, for example) and set the current limit 5-10% higher than the measured consumption.
- **Current measurement** : This will allow you to detect some more power-consuming behaviors in the target system, such as **radiofrequency** (**RF**) emission.
- **Multiple (at least two) variable outputs** : This will allow you to run some part of your target system at a voltage less than what they are intended to run at, or at a current limited to less than what they need, potentially triggering some interesting errors.
- The ability to chain outputs in case you need some higher voltage than usual.

Programmable power supplies aren't needed to start, but they can come in handy later when you need to program some behavior in function of time or other behaviors on your target system. They are usually more expensive than the simple ones but can come in handy.

Small tools and equipment

You will need a lot of different small tools in your lab. I personally use multiple mugs and boxes to keep them ready near my work area. Some examples are listed here:

- **Tweezers** : There are different point shapes and quality. You will have a very frequent use for sharp pointy ones for very small SMD components (0201, for example) and rounded, slightly larger ones for more common packages (0805, for example). The lowest-quality ones tend to bend quite easily, and I find that investing in medium-quality tweezers can be advantageous. You can find these for quite cheap on bidding or e-commerce sites such as eBay, Taobao, Aliexpress, Amazon, etc.
- **Scalpels** : I tend to use n°4 medical scalpel handles with detachable blades. They replace very advantageously the usual X-ACTO knives (even if the blades are a little less sturdy) since the blades are very cheap in packs of 100 and are available in a lot of different shapes.

I keep a stock of the following blades:

- *n°26* : for general cutting work and scrapping traces

- *n°23* : for cutting work that needs some force and cutting plastic

- *n°19* : for scrapping traces

- **Screwdrivers** : You will need a set of long- and thin-precision screwdrivers with multiple heads (at least flat, pozidriv, torx, and hex) in multiple sizes. The best approach here is to buy a set of screwdrivers with multiple heads and sizes. I would also advise that, when you have to buy a set of security bits, you buy one with the following: security hex, security torx, tri-wings, tri-groove, pig noses, and clutch A and G.

Some vendor-specific and even customer-specific screw/screwdriver couples exist, but this can usually be defeated with a bi-component epoxy compound or, in extreme cases, with a bit of aluminum casting or **computer numerical control** (**CNC**) machining.

- **Clamps** : The type of clamps you will be most interested in are called Kelly forceps. This type is used to keep things together with a bit of force, like holding boards together while soldering or holding wires in place while glue is curing.
- **Pliers** : You will very often use cutting pliers and long-necked ones to cut leads, remove connectors, and for a variety of different tasks. Again, buying decent-quality pliers will ensure they can survive small amounts of abuse that is very common in regular usage. I would advise investing in a good-quality wire stripper plier (of the simplest, flat kind that looks like a pair of pliers with multiple teeth sizes for the different wire sizes). I find that self-stripping tools tend to rip and break the cables that usually come with embedded systems far too easily.
- **Breadboard** : A breadboard is a tool where you can plug multiple wires and through-hole components temporarily. This is very useful to make small temporary circuits to power components and to have some glue logic, level shifting, modulation, and so on. You can easily start with cheap breadboards from bidding and e-commerce sites but they degrade quite quickly. Better quality brands such as 3M degrade less quickly, are a bit expensive, but hold better value over time.

Breadboarded circuits tend to be very fragile due to the way the components are mounted. Due to stray capacitance, I would not advise using breadboards with frequencies over 5 MHz. The indispensable companions to the breadboard are jumper wires (a length of wire with male or female connectors crimped at the end). Just find cheap lots of male-male, female-female, and female-male on bidding or e-commerce sites and buy some. I consider these consumables since I regularly cut them for ease of connection to a breadboard.

- **Perfboard/Stripboard** : These plates of PCB have either copper dots or strips you can cut and solder together in order to create circuits. They are more solid than breadboards and behave a bit better at higher frequencies.
- **Magnification** : As a first step, I recommend buying a few magnifying glasses that you can mount on your third hand (if it doesn't come with one already). At a later stage, and especially if you are working with very small components (0201 SMD or a lot of very fine-pitch MCUs, for example), a stereo microscope is very useful to see what you are actually soldering and keep a sense of depth to position your iron accurately.

Renting versus buying

It is quite common for companies to rent their test equipment long-term. It may or may not be interesting depending on your volume of use for a certain type of equipment. For example, you may need a specialized piece of equipment (such as a high-end **software-defined radio** (**SDR**); a vector network analyzer; a very very fast oscilloscope) for a specific engagement but you will very rarely use it in your normal work; then, it may be very practical and economically right to rent the piece instead of buying it. In a professional context, my approach for it is the following:

- If it is less than 2,000€, just buy it—renting will not be worth the hassle
- If I know I will not use it again in the next 6 months or if it is over 10,000€, rent it.
- The scope in the middle is then just a matter of calculation, as follows:

- (daily rent cost) x (number of days foreseen in the following year) < 50%

price: rent it.

- else, buy it.

Additionally, renting a piece of equipment before buying it will allow you to evaluate its interface and its performance across the spectrum of your different usages. Now that we have seen the different instruments we need to interact with components, let's have a look at those.

The component pantry

You will need a component pantry—by that, I mean that you will need at least an assortment of common resistors, capacitors, transistors, and voltage regulators always at hand. More often than not, you will find yourself in need of a jellybean component and will actually gain a lot of time by just having it available.

The pantry itself

Buy some of those drawer cabinets commonly sold to people that are making jewelry or doing any other hobby involving a lot of small pieces. Buy enough of them so that you can sort easily the (quite large) number of parts you will end up storing. Start by buying two to three of them; that will cover you for a few years. They are not really expensive and are really worth it.

I would advise labeling the drawers as quickly as possible and finding an organization system that suits you. For example, I have a column for through-hole resistors; another for surface mount; some drawers for capacitors; some for coils; and a column dedicated to silicon (diodes, transistors, voltage regulators, **electrically erasable programmable read-only memory** (**EEPROM**) , and others)

I also have a lot of custom shelves made out of cheap **medium-density fiberboard** (**MDF**) planks and brackets just screwed in the wall. There, I keep labeled boxes with development kits, instruments, a lot of electronic waste for cannibalization, instruments I rarely use, and others.

The stock

To start, I would advise keeping the following in stock:

- A collection of common resistors (buy some cheap E12 resistor kit on eBay) in through-hole (THT) and surface mount (SMT— a lot in 0805

and a few in 0402).
- A (small) collection of chemical and ceramic capacitor in common values (a few in the picofarad range: 0.1µ, 10µ, 47µ mainly, and a few big ones for power decoupling). For the packages, same thing as the resistors: a mix of through-hole and surface mount.
- A few power (1N4004) and signal (1N4118) diodes. A few Zener diodes for common voltage levels won't hurt (5, 3.3, 2.5, 1.8, 1.2). Zener diodes are designed to let current flow at a given voltage level, allowing you to protect circuitry against voltage spikes or to use them as a crude voltage conversion.
- At least a dozen fixed voltage regulators for the common voltages (5, 3.3, 2.5, 1.8, 1.2) and a few beefy adjustable ones (LM317 in a TO-220 package is very, very useful).
- Some standard transistors (both **Field Effect Transistors** (**FETs**) and **Bipolar Junction Transistors** (**BJT**), again in a mix).
- A few salvaged power supplies that can provide you with 24, 12, and 5 V (the powerful USB chargers that come with modern phones will give out a nice stable 5 V with decent amperage, are plentiful). Power supplies are very common e-waste and you can usually score a dozen for a small bill in any flea market

To keep my stock filled and enrich it, my strategy is to always order 10-15% more than I need in projects, just to cover the usage and not to have to follow individual component use (1 minute of your time is worth more money that the few fractions of cent a resistor costs).

Now, you should really play around with the components in your stock, learn about them, and make a few classical circuits to learn how they work and what they are actually doing, since keeping things you don't know how to use just for the sake of hoarding wouldn't make much sense, would it?

Now that we have looked at our instruments and components, let's have a look at a possible evolution path for your lab.

Buying components : There are plenty of electronics sellers, based in the eastern or western hemisphere (DigiKey, Mouser, LCSC, Element14,etc.). Basically the only difference is whether they have what you want or not. Most of them offer free delivery over ~50USD/€. I always use this free

delivery thing to get my 15% additionnal quantity to keep my pantry filled. I don’t charge my clients for this small signal diode or three 10KOhm resistors I used from my pantry and they usually don’t care if I use the free shipping to keep it that way.

Sample labs

In this section, we will be looking at different states of a home laboratory (from beginner to pro) that you could take inspiration from. When a piece of equipment is not described at a given level, it means that the piece is kept from the level before. Some pieces of equipment are not necessary before a given level of maturity (for example, the pro level doesn't have a new hot air station because it is kept from the amateur level).

Beginner

At this stage, the goal is to kickstart the activity as cheaply as possible, acquire knowledge, and check that you like it without burning too much money. Have a look at the following table:

MCU	A Chinese Arduino copy-start with Arduino and move later to a raw C context with avr-gcc and avrdude
platform	
Breadboard	Cheap Chinese from a bidding site
Oscilloscope	Any cheap secondhand 50 MHz bandwidth from a bidding site
Logic analyzer	A cheap bidding site Cypress FX2 repurposed board with homemade clamping diode input protection
Bus pirate	The one and only
Soldering station	Cheap bidding site temperature-controlled iron-the TS100 is very popular but you need an external supply
Function generator	A cheap **Direct Digital Synthesis (DDS) device** from eBay

Power supply	Repurposed phone chargers or **Advanced Technology eXtended** (**ATX**) power supply breakout (this is a small board that you can plug a computer power supply to)
DMM	El cheapo 10$ multimeter (do not work on mains voltage with this)

Price: <500€.

Amateur

At this point, you like the activity but you are starting to be limited by your equipment. You have circumvented some limitation by doing hacks, you have rolled out your own code to drive peripherals for common protocols on your current MCU and bit-banged some, but your platform is starting to become slow, your scope is not fast enough or lacking digital trigger, and more. Here are some pieces of equipment you can buy to solve these problems:

MCU platform	A fast STM32F4 (such as the Discovery), in pure C with arm-gee and link
Breadboard	A wide 3M with multiple rows
Oscilloscope	A low-level oscilloscope with at least 100 MHz of bandwidth, potentially a hackable one for better bandwidth or decoding of digital protocols
Logic analyzer	An open bench logic analyzer
Soldering station and hot air gun	A Chinese brand (Bekka, Yihua, OneHungLow, and so on
Power supply	A dual variable output with a fixed SV power supply. It is really easy to find one on a bidding site for a quite

	reasonable budget.
DMM	A reasonably priced DMM from a reputable brand (in the 100€ range) will do the job nicely.
Good helping hands	The Chinese "octopus" style help hands are easy to find on bidding sites. They will allow you to hold probes easily, even if you have a four-channel oscilloscope. They have an articulation system that looks like the feet of a gorillapod.
JTAG programmer	Any development boards based on an FTDI FT2232H will do the job nicely (it is compatible with OpenOCD). It won't give you crazy fast speed, but this is not something you really need at this point.

Price: <2,000€

Pro

At this point you are doing it regularly, so you will pretty much know what you will need. Have a look at the following table:

Oscilloscope	A good oscilloscope with a 350-500 MHz bandwidth from a major vendor {Rohde & Schwarz, Keysight, Tektronix, LeCroy, and so on) will be a serious investment. At this point, you will know what you need but will still need to research a lot since these instruments cost quite a bit of money.

Power supply	Choose a nice, programmable power supply from a mid-tier vendor with at least two variable outputs, such as the Rigol DP832.
Function generator	An entry-level function generator such as one from the Rigol DG900 series will cover your needs.
Logic analyzer	Saleae Logic Pro 16 is a very good logic analyzer with very practical software.
DMM	A mid-range DMM from Fluke {the DMM117, for example) will be good enough for what you will have to do. If you need something with more performance, have a look at bench multimeters.
JTAG programmer	A SEGGER J-Link will give you very nice speeds.
FPGA platform	Arty A7,57, or Z7, depending on your needs of having an onboard ARM CPU

Price: ~8,000€

Summary

In this chapter, we have seen the different tools that you will use and the different elements you will need to pay attention to when creating your laboratory.

A usually underestimated aspect of the lab is comfort—you will really spend a lot of time in there, so a good chair and a lot of natural light are quite important. I hope you will find all of these tips useful in the long run and that they will avoid you having to learn the hard way (like I did

In the next chapter, you will learn how to approach a target system and harvest information about it.

Questions

1. Why would you want two DMMs?
2. What is a 3 dB bandwidth?
3. Above which frequency will a breadboard parasitic capacitance interfere with the signals?
4. Who produces the bus pirate?
5. What is an oscilloscope?
6. What is the gist of the Nyquist-Shannon signal sampling theorem?
7. What is the main difference between active and passive oscilloscope probes?

Feedback

We are constantly looking at improving our content, so what could be better than listening to what you as a reader have to say? Your feedback is important to us and we will do our best to incorporate it. Could you take two mins to fill out the feedback form for this book and let us know what your thoughts are about it? Here's the link:
https://packt.link/HardwarePentesting2E .

Thank you in advance.

We cannot interact physically with the systems (humans are not very well equipped to see and produce precise and fast electrical signals, are they?) and we may not want to risk our main computer platform by connecting it directly to a **device under test** (**DUT**). We will need a specialized tool for this.

In this chapter, we will look at the main tool we will use to actively attack our targets. Both possible boards the code will work on (the bluepill or the blackpill boards) we are going to use are very cheap, accessible, and can be programmed with an entirely open source toolchain. We will review what they are exactly, their hardware, their variants, and how to program them (with a little introduction to C) before actually using it to attack protocols and chips in the next chapters.

In this chapter, we will cover the following topics:

- Introduction to the boards
- Why C and not Arduino?
- The toolchain
- Introduction to C

Technical requirements

In order to be able to program and use the boards, it is essential to have the following:

- A board (I'd advise you to buy a few, as they are always useful.
- Search for `bluepill stm32f103` or `blackpill stm32f411)` **on any bidding or cheap site (ebay, taobao, aliexpress)**
- A breadboard
- An STLINK USB stick: This looks like a USB stick with pins on the side opposite to the USB connector.
- A few wires for connections.

For the examples, you will require the following:

- Protocol: I2C: Chip: A PDIP 24LC I2C EEPROM
- Protocol: SPI: Chip: An MX25L8008 flash on a DIP breakout
- Protocol: UART: Any USB-to-serial adapter (the cheap ones based on CP2102 will do the job perfectly and they are useful tools too. Ordering more than one is a great idea; you need at least two)
- Protocol: Dallas 1-Wire: Chip: A DS18B20 (a temperature sensor)

You may want to also buy or find components that are using the same protocol but that are slightly different, so as to train yourself in adapting the examples.

In terms of the compilation of programs and flashing, install the following (for a Debian-based system):

- `gcc-arm-none-eabi`
- `libnewlib-arm-none-eabi`
- `binutils-arm-none-eabi`
- `gdb-multiarch`
- `openocd`
- `make`

- `texane st-link` (https://github.com/texane/stlink)

NOTE

Please be aware that the version that your distribution sports may not be sufficiently new. If this is the case, it could have a problem with the cheaper clones (in that case, install from source by following the instructions here: https://github.com/texane/stlink/blob/master/doc/compiling.md).

You can refer to the code used in this chapter at the following link:

https://github.com/PacktPublishing/Practical-Hardware-Pentesting

Check out the following link to see the Code in Action video:

https://bit.ly/307nM2u

Introduction to the boards

A board to do what? What are the boards? What can they do? How much does it cost? Why this one? Where is the documentation? Yes, you surely have plenty of questions! You will sometimes need a reminder while testing or doing the exercises, so I will also point to the chip's documentation. These questions are exactly what we are going to be talking about in the following sub-headings.

A board to do what?

Well, we will need to interface the board with the circuit we will want to attack. Since a general-usage PC doesn't really have a readily accessible interface board to connect with the most common protocols, we will use a bluepill or a blackpill to do so.

What are they?

The bluepill is a very cheap board and the blackpill is slightly more expensive but more capable. Both are very capable of being used to follow the exemples in this book (exemples have been tested with both).

For general usage the only major difference is the presence of a CAN interface and an additional analog-to-digital converter (ADC) on the bluepill the are lacking on the blackpill but in exchange the blackpill is about 20% faster.

The bluepill

The bluepill is a colloquial name for many different boards that have the following characteristics:

- Are cheaply available on bidding or Chinese goods sites such as eBay,

Taobao, and AliExpress (in the €1.5 range before the chipmageddon caused by Covid19 but in the 3€ range at the time of writing this book)
- Host an STM32F103C8T6 (or drop-in replacement parts from Chinese chip manufacturers) and its basic power circuity
- Break out most of the interesting pins in a format that can be plugged into a breadboard

The STM32F103C8T6 is a quite capable (32 bits, 72 MHz) microcontroller produced by STMicroelectronics that comes with a wide range of typical general-use peripherals:

- Two 12-bit ADCs
- Two I2Cs
- Two SPIs
- Three USARTs
- A USB peripheral
- A CAN
- GPIOs

We can now use these to interface with our target systems. Also, in quite practical terms, it is possible to program it directly in C (which we will use in the book) or use the Arduino IDE and API to program.

IMPORTANT NOTE

Some vendors are selling boards that have a clone of the STM32F103C8T6 on it. These should be fine, but the programming software may complain about it.

The blackpill

The blackpill is a colloquial name for many different boards that have the following characteristics:

- Are cheaply available on bidding or Chinese goods sites such as eBay, Taobao, and AliExpress (in the €3 range before the chipmageddon caused by Covid19 but in the 5€ range for the stm32f401 version and 6€

range for the stm32f411 version at the time of writing this book)
- Host an STM32F401 or STM32F411 and its basic power circuity
- Break out most of the interesting pins in a format that can be plugged into a breadboard

The STM32F411 and STM32F401 are quite capable 32 bit, fast(F411 : 100 MHz, F401 : 84 MHz) microcontrollers produced by STMicroelectronics that comes with a wide range of typical general-use peripherals:

- One 12-bit ADC
- Three I2Cs
- Five (F411) or four (F401) SPIs
- Three USARTs
- A USB peripheral
- GPIOs

And… a major difference between the two chips is that the F4 family comes with a hardware implementation for floating point arithmetics. This means that a small subunit in the microcontroller (the floating point unit or FPU) knows how to do operations like 1.2 + 2.1 = 3.3. In the bluepill the CPU does not have this sub-unit and can't do that in hardware. This means that, if your code does this kind of operations on the F103, this have to be done in software (by emulating the FPU if you want to look at it like that), making floating point operations really, really slow in comparison to the F4.

The compiler (we will explain what this is later) prevents you from mixing different pieces of software compiled for chips with and without FPU.

This impacts some compilation options but don't worry about that, the makefiles in the book's repository take care of that for you.

CAVEAT

If you have both bluepills and blackpills, you can't just change the flags in the makefiles managing the target board, you also need to clean the library. The makefile come with a target called

Why C and not Arduino?

The C programming language has a reputation for being hard to use and complex. Trust me, it is not. This reputation comes from the fact it doesn't come with a lot of the convenience functions of more modern languages. The simplicity that comes with this language makes it shine when the resources are constrained and when the execution needs to be really efficient, like on a microcontroller!

While I am quite sure that most of the examples in the book could be written using the Arduino IDE and API, it would do the following:

- Hide too much of the compilation chain and the programming process from you
- Prevent you from actually understanding the capabilities of the chip
- Make it difficult for you to actually know what is happening on the chip (since it uses some of the chip capabilities to provide you with convenience functions)
- Actually consume quite a bit of storage space to provide you with these convenience functions

All of this (unless you actually have a degree in electrical engineering or experience in programming embedded systems) would hinder your ability to understand your actual targets! It would do so because you will understand some fundamental concepts about the way in which microcontrollers work and are used on your targets!

Aside from that, you definitely should buy an Arduino and play around with it, but I will not focus on that here. You can even use the STM32duino libraries on this platform!

The documentation

The datasheet has a scope that is restricted to the model itself. Like most of the chip manufacturers, their chips are named in a nomenclature that allows

us to decipher the capabilities of the chip that is soldered on the board. For example, let's look at the nomenclature for the chip on the bluepill : the STM32F103C8T6:

- STM32: The family; a line of 32-bit cortex M-based MCUs.
- F1: This is a general-purpose, medium-density chip (F0s are even cheaper, L0s are energy-efficient chips, F3s are used for digital signal processing, and so on).
- F103: This is a 73 MHz chip with a CAN and USB.
- C: This is the pin count (48 for C).
- 8: This tells us that the chip has 64 KB of Flash and 20 KB of RAM.
- T: This is the package (the dimensions of the plastic capsule that encloses the silicon). T is `LQFP` (`low-profile quad flat package`).
- 6: This chip is designed to work in a "normal" range of temperature and not be exposed to too much heat or humidity, and so on.

In STMicro vocabulary, the document that will provide you with the detailed information of the family is a "reference manual." It will give you the addresses of the different memory-projected registers. It also explains the way in which the peripherals are programmed and all the things that are shared across the family members, irrespective of how much memory they have, how many leads are available on this package or that package, and so on.

Execice : Do the same thing for the chip that is on the typical implementation of the F411 version of the blackpill : STMF411CEU6. What is the package ? What the is onboard SRAM quantity ?

Board	**Chip**	**Datasheet**	**R**
Bluepill	F103	https://www.st.com/resource/en/datasheet/stm32f103c8.pdf.	ht st ar st
Blackpill	F401	https://www.st.com/resource/en/datasheet/stm32f401ce.pdf	ht st 32 ht

Blackpill F411	https://www.st.com/resource/en/datasheet/stm32f411ce.pdf	st 32

Reading the documentation

In the reference manual, you will find a description of all the peripherals that are on the chip. While reading the documentation for a peripheral, you should expect to always find the same following sequence:

- **Functional description of the peripheral** :
  - How the peripheral type behaves in general
  - What the available functionalities of the peripheral type are
  - How to initialize and configure the peripheral type
  - How to use the internal peripheral behavior (what the interrupts are, how they play out together, which bit is flipped by which events, and so on)
- **Configuration of the registers for the peripheral** : A description of all the registers (their addresses and all of their bits) that manage the peripherals, and for each instance of the same peripheral type
- **A register map** : A brief overview of all the registers described in the configuration

Memory-projected registers

Like most (if not all) programming languages, the main thing C does is make the CPU core move values from memory locations to other memory locations. In order to react to the programming, the chip has special memory regions where memory locations are actually special storage units ("the registers," as opposed to generic storage locations) that react to the stored value by altering the chip behavior. At some of these special addresses (that is, some registers), it is the behavior of the chip itself (such as its clock and turning peripherals on and off) that is set, and for others, it is the behavior of peripherals around the CPU that is altered. This concept is called **memory-projected register** and is the basis of the operation of MCUs and CPUs. Let's now dive into how this is translated in a binary that defines the MCU's behavior.

The toolchain

We will use a set of tools to transform a high-level language (yes, I wrote that, C is a high-level language) into the binary code that the chip understands and is laid out in a file that it can execute. To make it short, it's called **compilation** (compilation is actually one step of it, but it is a quite easy shorthand). We will push this file to the chip and have it run our code. In order to do that, we will have to use a set of tools and I will describe these in the following sections.

The compilation process

Under the generic compilation concept, the way it is understood by most people, we turn the code into something that can be executed by a computer. From the push of a button or a sternly typed command line, we see a file appear that we can run (a `.exe` file, a `.elf` file, or other formats). In reality, this is (of course) a little bit more complicated.

The compilation in itself

The goal of the compilation process is to turn a human-readable language (C, C++, assembly opcodes, Java, and so on) into a sequence of instructions that the decoding unit in the CPU can understand.

For the bluepill, we will use the **GNU Compiler Collection** (**GCC**) and, more specifically, a flavor (`gcc-am-none-eabi`) that is geared toward our architecture (`arm`) without any specific operating system (`none-eabi`).

In order to be able to understand the process, we will perform this operation on our local machine since it is easier to see the result than on the bluepill, and the process is essentially the same.

First, let's compile a simple hello world code:

Here, `gcc -c` means compile only. When we try to execute `hello.o` , the

error tells us that this is not a binary file that our computer knows how to execute. This is because we need to put it in a format it understands.

If you need to include header files (header files described by the functions provided by a library or another `.o` file), use `-I` to provide the path to the header directory and use the `#include` directive in the source file.

The linking

The linking turns object files into an understandable format for the operating system. In our example, the `printf()` function is provided by an external library (the description of what the library provides comes from the `#include <stdio.h>` line), but the operating system has no clue as to which library just by looking at the object file. This is the linker's job (we will use `gcc` to call the linker) to link it (and put the relevant information) into a file format that the operating system will understand:

This process (since it is not very clear in our very small example) is very important as soon as a project is divided into multiple source files. Each will become a `.o` object file and will be linked together as something that is usable.

Driving the compilation

Of course, a project can do the following:

- Encompass dozens of files.
- Need to be compiled in a debug version.
- Search for the location of libraries.
-

That is why there are tools to drive the compilation process. The simplest and most ubiquitous one is Make. Make is driven by a description file called a **Makefile** .

Anatomy of a Makefile

A Makefile can be complex (if you look at a big file for a complex project) but is composed of very simple elements:

- **Variables** : These are usually used to store things that you can use later. It is very common to put the name of the compiler, options, and path in variables. The affectation is done with the `VARIABLENAME=` value outside of targets and the evaluation with `$(VARIABLENAME)` .
- **Targets** : The things `make` must do in order to achieve the goal (the goal in question is usually a file). Targets can be described with dependencies in order to take care of the tasks required by the current target (again, usually a file). The file's dependencies follow a `:` after a target name. `make` looks at the change date of the files listed in the dependencies and only launches the tasks for the dependency files that are more recent than the target.

Let's have a look at a very simple Makefile to compile our hello world example:

Let's discuss a few terms from this Makefile:

- `CC` : A variable that contains the name of the compiler executable
- `hello` : Our main target, which requires `hello.o` in order to be started
- `hello.o` : A requirement target for `hello`

IMPORTANT NOTE

In Makefiles, before a list of tasks (such as the $(CC) directive (the tasks for the target)), there must be a tabulation (\t) and not just a space. If the make command tells you a separator is missing, this means that your editor transformed the tab into multiple spaces, and this will not work.

To illustrate the dependencies system, let's try a number of things:

Let's understand this code:

- First, we see that everything is up to date `(1)` .
- If we remove the executable file (the `hello` file), `make` will rebuild just

that `(2)` .
- If we make the source file more recent than the outputs produced `(3)` , `make` rebuilds everything.

Make is very powerful and allows much more than this simple example. I strongly encourage you to read some Makefiles to get used to its possibilities and, of course, read the documentation on Make's website: https://www.gnu.org/software/make/

Now we can build code, let's see how we can push it to the chip.

Flashing the chip

The easiest and most versatile software for STM32 chips on Linux is an open source implementation of ST's programming protocol. This software is available in the most modern distribution in a packaged format as the `stlink-tools` package.

INFORMATION BOX

For more information on the `stlink-tools` package, you can refer to the following link: https://github.com/texane/stlink .

It comes with different tools:

- `st-flash` : The basic tool to read from and write to the embedded Flash of an STM32.
- `st-info` : This tool gives you information regarding the connected chip.

Now, enough with the examples, let's do the real thing.

Putting it into practice for your board

In order to make our first program for our chips, we will need to do the following:

1. First, we will need to write a simple C program that will initialize the

chip and blink the onboard LED.
2. In the second step, we will use a linker script that will tell our compiler how to arrange the executable format in a way that is understood by our STM32.
3. Finally, we will flash it to the chip.

Using libopencm3

Before we start coding, we will need a corpus of information that will help us with providing all of the addresses of the different registers and constants that will help set them up without constantly doing (usually quite error-prone) bitwise arithmetic with raw values. Additionally, the `opencm3` library comes with convenience functions to set up and use peripherals that we will use later on.

Here is how to get the library:

At this point, the library is ready to be used.

The code

The chip needs to be initialized for the following purposes:

- To tell the chip which clock source to use (its internal oscillator or a more precise external crystal)
- To know what to do with the clock source in order to clock itself (via an internal component called a PLL, it can multiply or divide the clock source to feed the different clock signals it needs)
- To determine what peripherals to initialize in order to use the general-purpose input/output to which the LED is attached
- To toggle the pin that commands the LED wait a bit, toggle the pin that commands the LED

The entire code and Makefile can be found in the book's Git repository in code `/ch2/blink_noop` (do not forget to clone it and its submodules with `--recursive`).

Try to read the Makefile and understand what it does, as well as what the different targets do:

- Connect your STLINK stick to the bluepill and flash the code to it (with `make flash`). Connect the GND on the STLINK to the GND on the bluepill, 3V3 to 3V3, SWD to SWD, and SWCLK to SWCLK).
- Try to change the value in the second `while` loop to make it blink slower.
- Try to change the value in the second `while` loop to make it blink faster.
- Search the `libopencm3` documentation to see how you could replace the rcc_clock_setup_pll function.
- Read the function code and the reference manual to understand how it works (in the RCC chapter of the reference manual).
- Make the MCU run at 48 MHz from the HSI through the PLL (there is an already made set of pll parameters in libopencm3 for that) and see how it influences the blinking speed.
- Download ST STM32Cube software, start a new project with the correct chip for your board, and then go to the clock management tab and look at how the different peripheral buses are clocked.

Now that we've seen how code is transformed into a binary that can be transferred to the chip, let's look a bit more into the code and how it works.

Introduction to C

C will be your bread and butter for developing your attacks. Yes, there are easier, more modern, less cumbersome languages, but the following is true:

- The abstraction level prevents you from understanding what is happening on the hardware.
- Most of your reversing targets will be C-based.

So, pony up, and learn the language that makes the hardware run!

This is really intended as a crash course that will just allow you to understand the code that comes with this book. There are plenty of resources on C on the internet if you want to dig deeper (and trust me, you will want to).

Operators

C comes with most of the operators you are expecting:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`>>` and `<<`	Bit shift (right and left)
`|`	Binary OR (the bits at 1 in the first or second operands will be at 1 in the result); widely used to set binary flags
`&`	Binary AND (the bits at 1 in the first and second operands will be at 1 in the result); widely used to mask or unset binary flags
`^`	Exclusive binary OR (the bits at 1 in the first and second (but not in both) operands will be at 1 in the result); widely used to toggle binary flags
`%`	Modulo (remainder of the division)
`&&`	Logical AND
`||`	Logical OR
`==`	Test for equality
`<` (and `<=`), `>` (and `>=`)	Test for is smaller than (and is smaller than or equal to), and is greater than (and is greater than or equal to)
`!` (and `!=`)	Logical NOT and test for "is different"
`var++` and `var--`	Increment and decrement a variable (+1 and -1)
`a ^= b`	Shorthand a = a ^ b
`a |= b`	Shorthand for a = a | b
`a &= b`	Shorthand for a = a & b
`test ? if_true : if_false`	The `?` operator (known as the ternary operator is a shorthand for `if`; it is equivalent to `if (test) { if_true } else {if_false}`

You may already be familiar with the majority of the statements:

`if(test){block}`	Executes the lines in a block if the test is `true`.
`if(test){block1} else{block2}`	Executes the lines in `block1` if the test is `true`, otherwise the lines in `block2`.
`while(test){block}`	Executes the lines in a block while the test is `true`.
`for(pre_action;test;post_action){block}`	Executes the lines in a block while the test is `true`, but executes `pre_action` once before the block and `post_action` after every block. Usually, `pre_action` initializes a counter, while `post_action` changes the counter.

The comments can come in two forms:

// comments	A single-line comment
/* comments */	A multi-line comment

Numeral bases as literals are also very straightforward:

`1234`	Base 10; decimal number
`0x321b8f1`	Base 16; hexadecimal
`0b101`	Base 2; binary
`010`	Base 8; octal; this is rarely used, but since it starts with a 0, it can pretty easily look like a decimal number!

Types

Variables have a type. This is so that the compiler knows what kind of operation to apply to the variable.

The main types in C are as follows:

- `int` : an integer value, usually 4 bytes

- `short` : a short integer, usually 2 bytes
- `char` : enough to hold a character, usually a byte
- `float` : a representation of a real (floating point) value, usually 4 bytes. Attention, the precision is limited !

That's it. There are no evolved types such as strings, lists, and hash maps out of the box. This is a very concise language where you have to create the evolved types you may need from the basic types. But don't underestimate C. The chances are that it is still the language that created the code managing the hardware in most of the devices you own. The majority of the kernels, the low-level libraries, are written in C because it is extremely efficient, both for size and for pure code performance.

The dreaded pointer

Pointers are making people afraid of C, and this is somewhat ridiculous. Pointers, just by themselves, are making people afraid of this language. Generations of students have been frustrated by the dreaded and mystical beast called "segmentation fault" (the error that usually comes from flawed pointer operations).

It is true that people are scared of pointers, and I cannot fathom why. They are easy.

A variable is held at a memory location. The pointer is the address of this location. Done

The notation for pointers is `*` (a pointer is a type and it points to a value with a type so that the compiler can perform a size calculation). The notation of "get address of" is `&` , while, within an expression, `*` is used as a dereference (that is, "this thing that is at the address I am applying the `*` to"):

In C, pointers are the way in which arrays are managed, either with dynamic allocation (almost never used in MCUs), or statically with the `[]` shorthand syntax:

Since the array is so easy to use, it is also used to hold strings:

Like I said before, this is just a crash course, but for now, you are able to code for the bluepill, push code onto it, and start having fun!

Preprocessor directives

Preprocessor directives are directives that a special piece of code in the compiler (the preprocessor) understands. They begin with `#` and are used by the preprocessor to do text replacement or file inclusion.

The most frequently used directives are the following:

`#include <filename>`	This directive pastes the content of the file "filename" (in the search path of the preprocessor) into the processed file. Two forms exist: `<filename>` means in the search path, and "filename" means relative to this file.
`#define text replacement_ text`	This directive replaces `'text'` with `'replacement text'`. This is usually done to define file-wide constants and text replacement variables and macros. It is possible to define replacements with variables like so: `#define min(X, Y) ((X) < (Y) ? (X) : (Y))` For example, most of the register addresses in `libopencm3` are defined this way.

`#ifdef ...`
`#endif`
	This directive is used to test for the definition of a text replacement variable. The negative form (`#ifndef`) is commonly used to avoid the inclusion of a header file multiple times, like this:
	`#ifndef __MY_VARIABLE THAT_PREVENTS_THIS_ HEADER_MULTIPLE_INCLUSION`
	`#define __MY_VARIABLE THAT_PREVENTS_THIS_ HEADER_MULTIPLE_INCLUSION`
	`...header content...`
	`#endif`

Multiple other directives exist including `#undef` , `#else` , and more besides.

Functions

Declaring a function in C is very easy:

Then, the `function_name` variable simply holds a pointer to the assembly code that implements the function. One consequence of this is that it is possible to use function pointers as variables that hold a reference to a function that you can change and call dynamically.

Explore more and look into the `blink_systick` folder. This is also making the led blink but by using and external peripheral (called `systick`). Follow the exploration instruction in the code and look how you can overclock your chip (and why you shouldn’t).

Explore even further and look into the `blink_timer` folder. This is also

making the led blink but by using and external peripheral (called a timer). First understand how this timer allows us to get this nice dimming effect (this is called pulse width modulation or PWM). Rea the timers documentation in the reference manual and try to play with it… Ain’t this neat ?

Summary

In this chapter, we have programmed our main attack platform for the first time and then installed and compiled the library that will help us interact with its peripheral. We also had a brief introduction to the language we are going to use to program it – C.

In the next chapter, we will go through the most common protocols used in embedded systems, and learn how to find them, sniff them, and then attack them with our bluepills.

Questions

1. What is the `GPIOC_ODR` register that I XOR in the blinking example? Can you achieve the same effect by using other registers?
2. Is it possible to have the MCU run at 72 MHz for the HSI? Why or how? What is to be expected then?
3. What are the premade frequency assignment functions available in libopencm3?
4. XOR each character of the string `Z9kvzrj8` with 0x19 in a C program. What does this mean?
5. What is the address of the `GPIOC_ODR` register? How can we find that easily?

Further reading

Read more about the C language:

- The seminal C book: *The C Programming Language* , by Brian Kernighan and Dennis Ritchie; ISBN 978-0131103627
- *21st Century C* , by Ben Klemens; ISBN 978-1449327149, because just because the language is 40 years old, doesn't mean you have to write it like it was 40 years ago

Read more about GNU Make:

- *Managing Projects with GNU Make* , by Robert Mecklenburg, Andy Oram, and Steve Talbott; ISBN 978-0596006105

Feedback

We are constantly looking at improving our content, so what could be better than listening to what you as a reader have to say? Your feedback is important to us and we will do our best to incorporate it. Could you take two mins to fill out the feedback form for this book and let us know what your thoughts are about it? Here's the link:
https://packt.link/HardwarePentesting2E .

Thank you in advance.

Now that we've seen how to program the chip, let's apply it to an application and use it to actually start attacking systems. We will do that by looking into a number of standard protocols that are used to communicate between chips and the outside world. They usually define only the physical layer for chip-to-chip communication and (almost) never go into higher levels of abstractions. In this chapter, we will learn how to operate, sniff, and attack I2C, SPI, UART, and **Dallas 1-Wire** (**D1W**).

In this chapter, we will cover the following topics:

- Understanding I2C
- Understanding, sniffing, and attacking SPI
- Understanding, sniffing, and attacking UART
- Understanding, sniffing, and attacking D1W

Technical requirements

In this chapter, we will look into, sniff, inject, and man-in-the-middle the most common hardware protocols. There are a small number of things that you can get for yourself if you want to replicate the practical demonstrations. (These are not absolutely necessary but there is both a theoretical and a practical know-how aspect to what is covered in this chapter. I warmly recommend that you actually replicate the exercises.)

Hardware

In order to be able to follow along, get yourself the following:

- A breadboard
- Two blue or black pills (that are covered in the previous chapter)
- An STLink to program them (sometimes the UART bootloaders are not wired correctly)
- Jumper wires
- Any logic analyzer (we will use an open bench analyzer)

The following peripherals are required:

- **I2C** : A PDIP 24LC I2C EEPROM
- **SPI** : An MX25L8008 flash on a DIP breakout
- **UART** : Any USB-to-serial adapter (The cheap ones based on CP2102 will do the job perfectly, as they are useful tools. Ordering more than one is a great idea.)
- **D1W** : A DS18B20 (a temperature sensor)

The software needed for Linux is as follows:

- `arm-gcc-eabi-none`
- `stlink`
- `sigrok`
- Fritzing (to see the breadboard implementation of the components)

In this chapter, there are a few schemas that show how to connect components to a breadboard. Since the book is printed in grayscale, it may not always be very easy to differentiate the wires. If it is not clear enough for you, please download Fritzing, a software that can show (in color) the breadboard schematic files that are in the GitHub repository: https://github.com/PacktPublishing/Practical-Hardware-Pentesting/tree/main/bluepill/ch6 .

The Fritzing schematic show the location for the blue pills. If you use blackpills, just reuse the same pin numbers but be aware that

The GitHub repository also contains all of the example code.

Let's start with I2C.

Check out the following link to see the Code in Action video:

https://bit.ly/3q2TkRK

Understanding I2C

I2C (pronounced as *I-two-see* , or *I-square-see*), short for **Inter-Integrated Circuit** , is a protocol that was invented by Phillips in the early 80s to be used in televisions. Due to its easy topology and low part count, it is now widely adopted.

Mode of operation

I2C connects chips with two wires: one is data (bidirectional) and the other is clock (of course, with a shared ground). On the bus, one chip acts as the master and the others as slaves (but they can exchange this role if this functionality is foreseen).

Physical layer

A very important feature on the I2C bus is that both lines (classically called **Serial Data** (**SDA**) for the data line and **Serial Clock** (**SCL**) for the clock line) are pulled up. This means they both have a resistor to the logical positive rail (also called VCC or VDD) in order to guarantee that the bus is high when no chip is pulling it to ground level (low). The bus normally uses a bus topology, but at low speeds, it is possible to use a star topology. Both the bus master and slaves can (and will) clock the bus.

The bus classically looks like this:

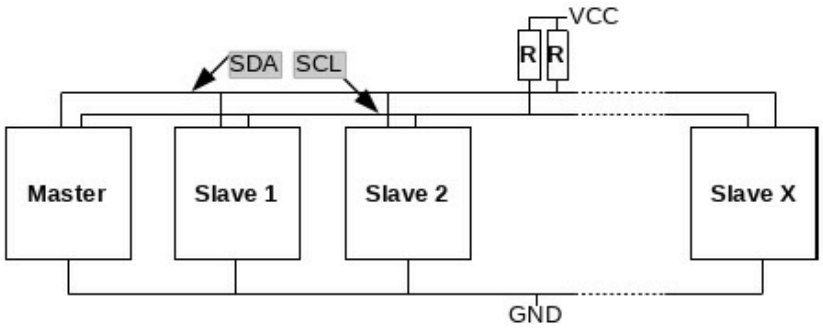


Figure 6.1 – General I2C architecture

Speed is very important regarding the physical layer (and will have an impact when you want to impersonate a chip, for example).

I2C comes in multiple speed grades, as shown in the following table:

Mode	Abbreviation	Frequency	Min. charge current (Imin, typical)
Standard	Sm	$f \leq 100$ KHz	2 mA
Fast	Fm	100 KHz $< f \leq$ 400 KHz	2 mA
Fast plus	Fm+	400 KHz $< f \leq$ 1 MHz	3 mA
High speed	HSm	1 MHz $< f \leq$ 3.4 MHz	6 mA
Ultra-fast	UFm	3.4 MHz $< f \leq$ 5 MHz	20 mA

The speed mainly has an impact on the values of the pullup resistor for the following reasons:

- Everything on a board (traces, components, and more) has parasitic values (that is, the values that come from the environment, the package itself, and a plethora of other factors).
- The traces of the bus have a parasitic capacitance (but also their own resistance, inductance, and more); it is possible that the traces and the pullup resistor act as resistance that is charging a capacitor.

While the impact of this is very limited at slower speeds, it can be measurable and even disturbing at higher frequencies (this is the reason behind not using a breadboard at fast speed in *Chapter 1 , Setting Up Your Pentesting Lab and Ensuring Lab Safety*).

INFORMATION BOX

The rule of thumb is, the higher the speed, the lower the resistance (lower resistance implies more current, which implies faster response – this is a rule of thumb; the capacitance of the circuit has an impact but you can't change it).

To calculate the needed resistance, the following formula can be applied. This is a simplified version; there is a formula that takes the parasitic capacitance of the traces into account, but it is hard to

measure. Typical capacitance is taken into account in the *Imin* values indicated in the following formula:

$$Rpull = \frac{VCC - min(0.4, (0.1 * VCC))}{Imin}$$

Here is a table of resistance for common logic-level values (in reality, select the closest smallest standard value in the resistors you have available):

Mode	VCC: logic level						
	5 V	3 V	2.5 V	1.8 V	1.5 V	1.2 V	1 V
Standard	2300	1350	1125	810	675	540	450
Fast	2300	1350	1125	810	675	540	450
Fast plus	1533	900	750	540	450	360	300
High speed	766	450	375	270	225	180	150
Ultra-fast	230	135	1125	81	67.5	54	45

IMPORTANT NOTE

Not every MCU can sink this much current easily! If you don't pay attention, you can easily burn your pin (or your MCU). Some additional transistors can solve this problem. Selecting a value too low for the pullup resistor will lower the state change time but can also change the minimal voltage and make it not close enough to GND. This can prevent the system from working.

Logic levels and voltage translation

This logic translation is mainly used for I2C but can also be applied to other protocols.

I2C doesn't really force a specific value for the logic levels, and both the slaves and the master can pull the SDA low. However, it is then necessary to

use a bidirectional voltage-level translation that is able to cope with this. Dedicated chips exist but they tend to be expensive. Thankfully, an engineer for Philips has provided us with a clever trick to do this using just two MOSFETs. The arrangement is shown in the following diagram:

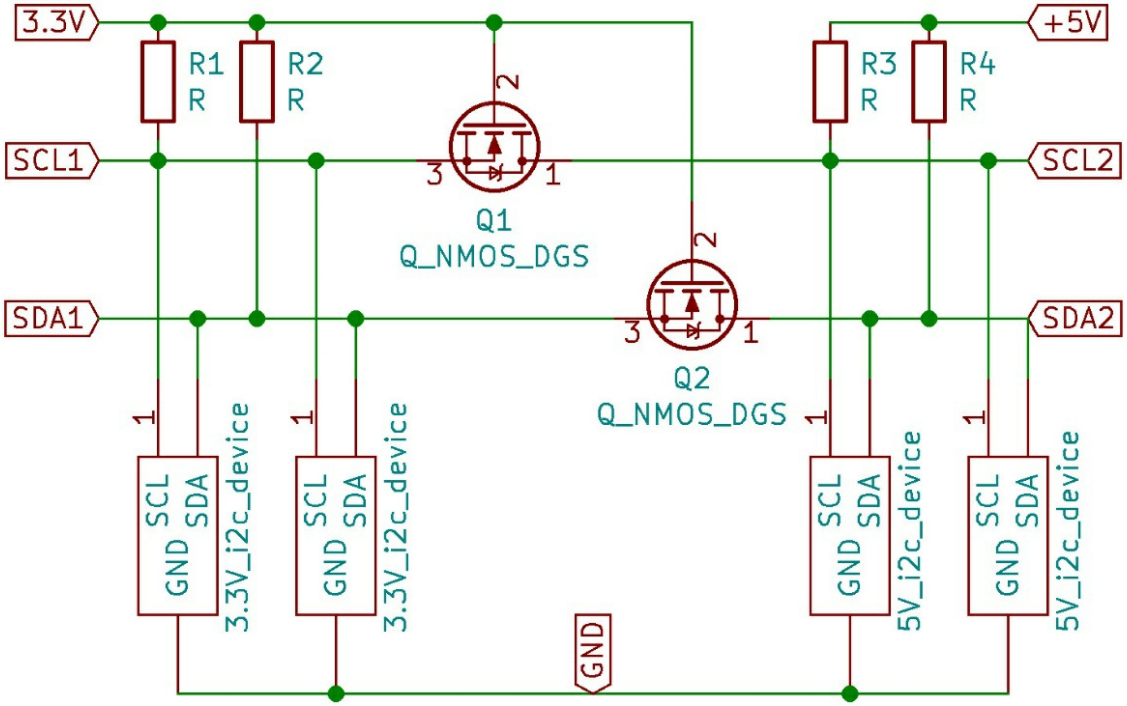


Figure 6.2 – Bidirectional voltage translation

The original author (Herman Schutte) suggests using BSS138 MOSFETs for 5 V<>3.3 V translation. This MOSFET typically drives with a gate voltage of 1.3 V (refer to the datasheet). So, you will need to find replacements if your logic levels are lower (than 1.8 V to be on the safe side). You may need to find a MOSFET with a lower gate threshold. Some vendors offer BSS138 with a very low minimal gate threshold. You may need to buy a few dozen, find the ones that are more on the lower end of the spectrum, and select those specific ones in the lot for your voltage translation. I found BSS138 with a Vgs as low as 700 mV in a lot from Diode Incorporated.

The physical format of the bits

The bits are transferred by encoding them in the way SDA and SCL behave relative to one another.

The bits are transmitted as shown:

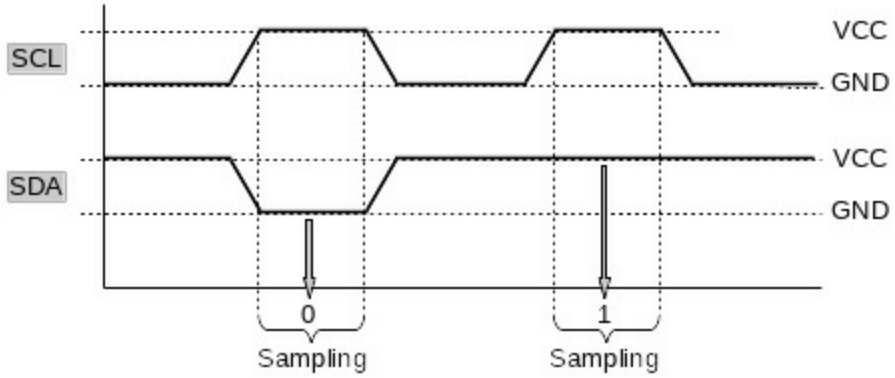


Figure 6.3 – I2C sampling

The sampling is typically done around the middle of the active clock cycle (that is, SCL is high, as shown in the preceding figure) or on an SCL raising edge. The sampling reads the state of SDA to get a 0 or 1 from the signal.

A few special conditions that are not following the normal bit encoding or behavior rules are used to support additional signaling between the devices:

- **Start condition** : To start communication on the bus, the bus master pulls SDA low while SCL is kept high:

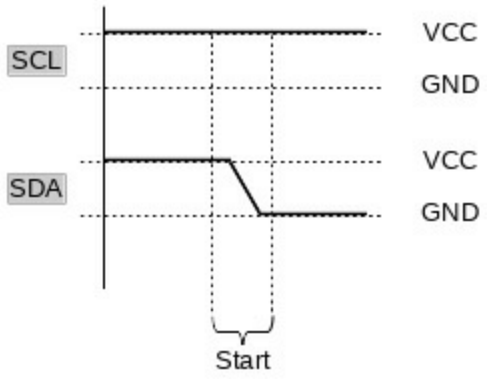


Figure 6.4 – I2C start condition

- **Stop condition** : To stop communication on the bus, the bus master pulls SDA high while SCL is kept high:

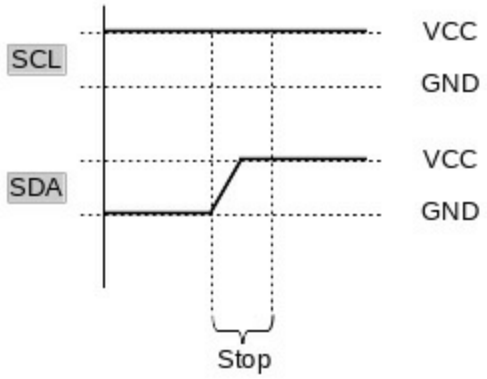


Figure 6.5 – I2C stop condition

- **Restart condition** : This is similar to the start condition. During a transaction, SDA goes low while SCL is kept high:

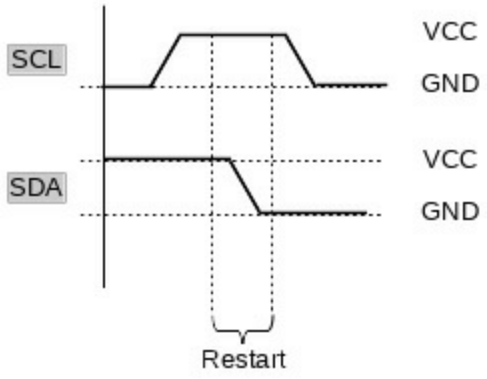


Figure 6.6 – I2C restart condition

- **Pause** : The slave keeps SCL low, preventing the master from clocking SCL (the master detects it and pauses the transmission) and frees up SCL for the master when it is done:

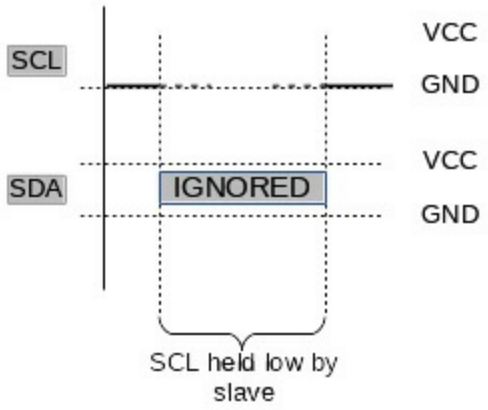


Figure 6.7 – I2C pause

We have covered the physical layer, so let's have a look at the logical layer now.

Logical layer

I2C supports an addressing system over 7 bits (or 10 bits with an extension but this is not supported by all devices). This means that the theoretical maximum number of devices is 126 (the zero address is supposed to be a broadcast, but this is not actually implemented in all chips).

Every device has a 7-bit address and the last bit of the byte is used to indicate whether this is a read or write request:

Bit READ	Emitter WRITE	Bit signification WRITE		Bit WRITE	Emitter READ	Bit signification READ	
	Master	Start			Master	(Re)start	
0	Master	Address bit 1		0	Master	Address bit 1	
1	Master	Address bit 2		1	Master	Address bit 2	
2	Master	Address bit 3		2	Master	Address bit 3	
3	Master	Address bit 4		3	Master	Address bit 4	
4	Master	Address bit 5		4	Master	Address bit 5	
5	Master	Address bit 6		5	Master	Address bit 6	
6	Master	Address bit 7		6	Master	Address bit 7	
7	Master	Write: 0		7	Master	Read: 1	
8	Slave	Acknowledgment		8	Slave	Data bit 1	
9	Slave	(Optional) Pause		9	Slave	Data bit 2	
10	Master	Command bit 1		10	Slave	Data bit 3	

Bit READ	Emitter WRITE	Bit signification WRITE		Bit WRITE	Emitter READ	Bit signification READ	
11	Master	Command bit 2		11	Slave	Data bit 4	
12	Master	Command bit 3		12	Slave	Data bit 5	
13	Master	Command bit 4		13	Slave	Data bit 6	
14	Master	Command bit 5		14	Slave	Data bit 7	
15	Master	Command bit 6		15	Slave	Data bit 8	
16	Master	Command bit 7		16	Master	Acknowledgment	
17	Master	Command bit 8			Slave	(Optional) Pause	
18	Slave	Acknowledgment		17	Slave	Data bit 1	Repeat as needed
	Slave	(Optional) Pause	Repeat as needed	18	Slave	Data bit 2	
	Master	(Re)start		19	Slave	Data bit 3	
0	Master	Address bit 1		20	Slave	Data bit 4	
1	Master	Address bit 2		21	Slave	Data bit 5	
2	Master	Address bit 3		22	Slave	Data bit 6	
3	Master	Address bit 4		23	Slave	Data bit 7	
4	Master	Address bit 5		24	Slave	Data bit 8	
5	Master	Address bit 6		25	Master	Acknowledgment	
6	Master	Address bit 7			Slave	(Optional) Pause	
7	Master	Write: 0					
8	Slave	Acknowledgment					
9	Slave	(Optional) Pause					
10	Master	Command bit 1					
11	Master	Command bit 2					
12	Master	Command bit 3					
13	Master	Command bit 4					
14	Master	Command bit 5					
15	Master	Command bit 6					
16	Master	Command bit 7					
17	Master	Command bit 8					
18	Slave	Acknowledgment					
	Slave	(Optional) Pause					

This is a complete description of an I2C transaction between a master and slave. Now that we know this, let's sniff a communication and see how the protocol is used in the communication.

Sniffing I2C

There are at least two ways to sniff I2C: the generic way (with any logic analyzer) or by using the Bus Pirate.

The target circuit we use as an example is any micro-controller (a blue pill for us) connected to an I2C chip (PCF8574P for me).

INFORMATION BOX

> Since it will probably be the first time you interact with your USB devices, don't forget to set up udev rules (for Linux) in order to be able to interact with them without needing superuser privileges.

Look into your system log (`dmesg` for Linux) after plugging in the device and note down the `vendorid` and product ID values. For example, this is what `dmesg` says for my logic sniffer:

Create a file for `udev` (usually in `/etc/udev/rules.d/`) with a line like this:

Reload the rules with the following (as `root` / `sudo`):

Now that we have seen how the protocol is behaving, let's have a look at how we can read it.

Using a generic logic analyzer

While we're using the logic analyzer for the first time for I2C, it is of course also usable for the other protocols.

We will use an open bench logic analyzer and `sigrok` (the open bench logic analyzer is open source hardware and `sigrok` is open source software).

We will need to connect the following.

The blue pill, the EEPROM, the two pullup resistors, and the serial adapter need to be connected like so:

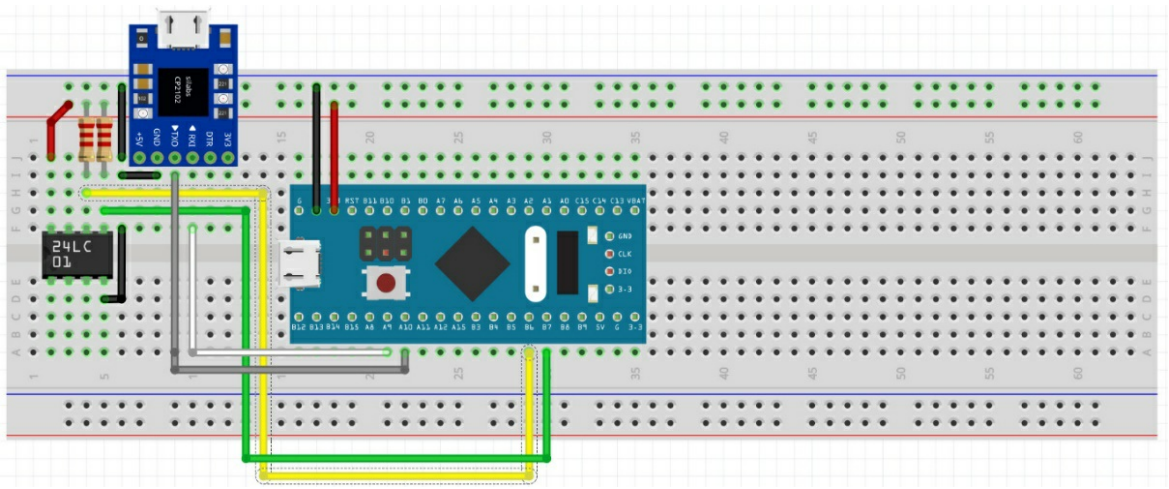


Figure 6.8 – I2C usage connection

Open the Fritzing schematic (https://github.com/PacktPublishing/Practical-Hardware-Pentesting/tree/main/bluepill/ch6/fritzing_Schematics) to identify the components.

To connect the analyzer to the circuit, do the following:

1. Connect the ground together.
2. Connect pin 0 of the analyzer to the SCL line.
3. Connect pin 1 of the analyzer to the SDA line.
4. Connect the analyzer to USB.
5. Launch PulseView (`sigrok` 's GUI) and connect it to the logic analyzer.

6. Click on the **Connect device** button and set up the analyzer as shown:

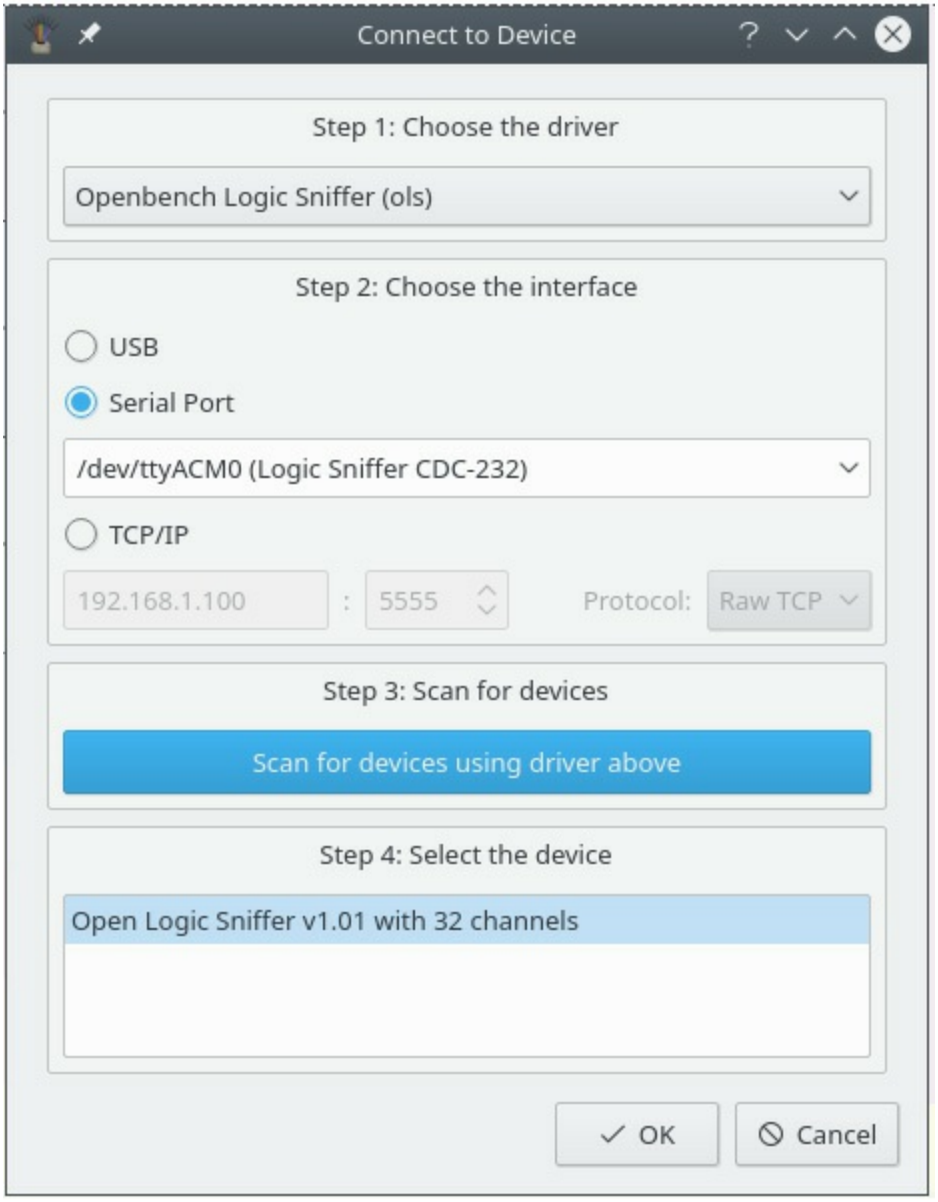


Figure 6.9 – Connecting the logic analyzer

7. Select the logic analyzer pins you used from the menu with this icon:

. Click on the sniffing button until you see a waveform that looks like this:

Figure 6.10 – Adding a decoder

8. Add a decoder (I2C) by clicking on the yellow-and-green button on the top bar.
9. Click on the I2C decoder to select which line is SCL and which is SDA (in my sniff, I used pins 6 and 7: 7 as SCL and 6 as SDA).

Now we can see on the decoder line what the values that transited on the I2C bus are!

Now you can sniff I2C with a logic analyzer.

Using the Bus Pirate

This is also applicable to other protocols.

The Bus Pirate offers multiple easy ways to interact with I2C but has the downside of coming without a GUI. It is available as a serial device on your computer and you can interact with it on the command line. It can sniff I2C up to 100 KHz.

All the commands related to I2C are documented here: http://dangerousprototypes.com/docs/Bus_Pirate_I2C (the commands for the other protocols have their own pages).

The Bus Pirate can sniff (relatively) low-speed I2C and render the traffic in the same syntax it would use to emit the I2C traffic. You will then be able to replay the traffic really easily. Just connect the Bus Pirate pins to the previous breadboard.

Let's give this a try:

1. Connect to the command-line interface of the Bus Pirate (screen or minicom, whichever is your favorite serial client; for me, it is screen). Don't forget the `udev` rules; you can even give a cool persistent name such as `/dev/buspirate` with the SYMLINK directive. I'll let you search how to use `udev` directives by yourself; maybe there will be questions on this at the end of the chapter!:

```
$screen /dev/ttyUSB0 115200
```

1. Put it in I2C mode in the terminal (4) and launch the `snif` macro (2).
2. Power the circuit.

You should see something like this:

```
[0xa0+0x00+0x01+[0xa1+0xXX-0xff]
```

`0xXX` is dependent on the content of your EEPROM. If it has never been written to, it will be a random value.

Let's look into the datasheet for the EEPROM and make sense of this traffic:

- `[` : The start condition
- `0xa0` : But the device address was `0x50` in the `sigrok` sniff! Remember how the address is on 7 bits and uses the eighth bit of the byte to indicate read or write? `0x50 << 1 = 0xa0 =` is the write address of the peripheral at `0x50` .

INFORMATION BOX

We need to be familiar with the following.

The C notation for bit-wise and Boolean operators (as these operations are extensively used in embedded systems) is really important. Please refer to the GNU C manual: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Bitwise-Logical-Operators.

Memory representation of various types (that is, how an

integer, a float, or a long long is represented in memory and what this representation entails in term of limitations) is also very important to know; you don't really need to know the details intimately but knowing they exist and how to find out how they work will sometimes open interesting doors for you.

- `+` : The address is ACKed by the EEPROM.
- `0x00` : First bit of the address
- `+` is ACKed.
- `0x01` : Second bit of the address
- `+` is ACKed.
- `[` : We restart.
- `0xa1` : `0xa0 | 1 = 0xa1` – the last bit is 1, so we want to read.
- `0xXX` : We read a byte that we NACK (since it is the last byte we want).
- And the state machine of the MCU just clocks in a byte anyway (because that is how it works) and stops.

Injecting I2C

Injecting I2C can be really tricky for two reasons:

- Not all masters on an I2C bus are able to follow the multi-master arbitration protocol. In order to be able to inject I2C on a live bus, if the multi-master is not supported, we will need to get crafty (either by studying the period at which the master is transmitting and leverage pauses or by actually doing a man in the middle).
- Remember how I2C is an open collector bus with pullup resistors? This means that you have to pull down the bus (actually do a bus stretching like you were a slave) and use its pullups. Sometimes, this makes the bus masters that don't support the multi-master functionality behave really weirdly and sometimes crash (that can also be interesting but, in my experience, not very useful).

Otherwise, injecting I2C is just a matter of connecting another master on the bus without a pullup resistor (it is using the existing ones) and avoiding collisions.

Exercise

1. Connect one blue pill and flash the code for the I2C sniffing example.
2. Do the same for the second blue pill without dedicated pullup resistors (there shouldn't be any collision unless you are really, really, really unlucky).
3. What do you see on both serial outputs?
4. What do you conclude?

I2C man in the middle

By using a micro-controller with two I2C peripherals (such as the blue pills), we can put ourselves physically in between the master and slaves, acting as a slave to the master and as a master to the slaves. For this, we just have to know the expected address of the slave.

Here is how to put the components on a breadboard:

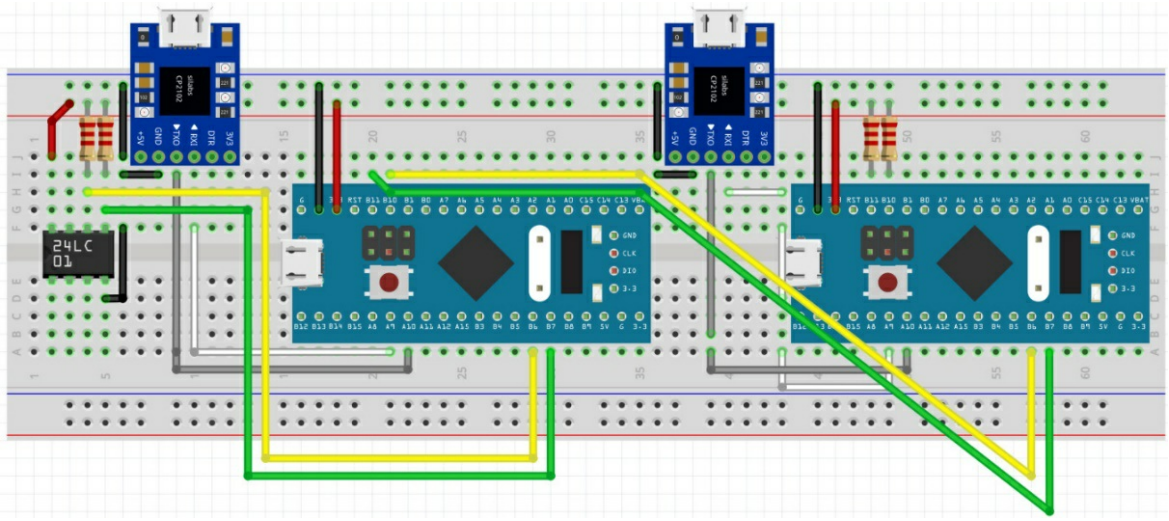


Figure 6.11 – I2C man-in-the-middle connection

We will alter the traffic to have the master read `p-e-n-t-e-s-t` every 100 bytes.

The code is in this chapter's folder in the cloned directory. Compare it to the reading code, play with the interrupts, and try to understand it.

Understanding SPI

SPI , or **serial-to-parallel interface** , is a (usually minimum) three-wire bus. One acts as the clock (CLK), one as **Master Out Slave In** (**MOSI**), and one as **Master In Slave Out** (**MISO**). If multiple slaves are present in the bus, there is also an additional wire per slave called **CS** or **SS** (**Chip Select** or **Slave Select** , usually active low).

Here is how multiple slaves are connected:

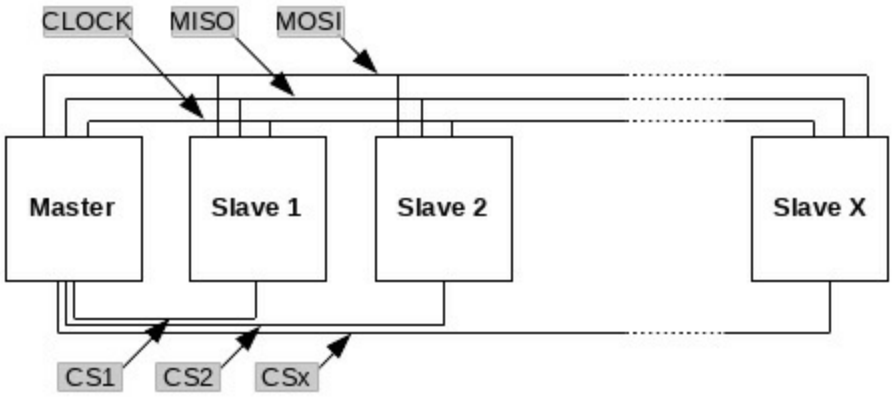


Figure 6.12 – SPI general architecture

SPI only manages how the bits are transferred on the line; there is no logical layer in the protocol (like I2C has).

On systems where the speed of transfer is important, SPI can come in the **QSPI** flavor (**queued SPI** / **quad SPI**) where there are four data lines. You should note that some chips support both modes and can switch between them with internal commands (that is, commands in the data that are transported by SPI, not commands determined by the SPI protocol itself).

Now that we have seen how the chips are connected, let's see how it works.

Mode of operation

First things first, SPI has a frequency. This frequency is determined by the

master (which is pulsing the clock) and must fall within the max frequency that the currently selected peripheral (with a CS wire) supports.

The second thing to take into account with SPI is two parameters called CPOL and CPHA. These parameters manage the clock polarity and clock phase:

- **Clock Polarity** (**CPOL**) governs the fact that the clock wire is considered active high or low.
- **Clock Phase** (**CPHA**) governs the timing at which the data will be sampled on the adequate wire in respect to the clock cycles.

This creates four "modes" (CPOL and CPHA names are inherited from PIC MCUs but this became a de facto standard).

SPI mode 0

In this mode, the clock is active high. Data is sampled on the leading edge of the clock cycle and changed on the trailing edge:

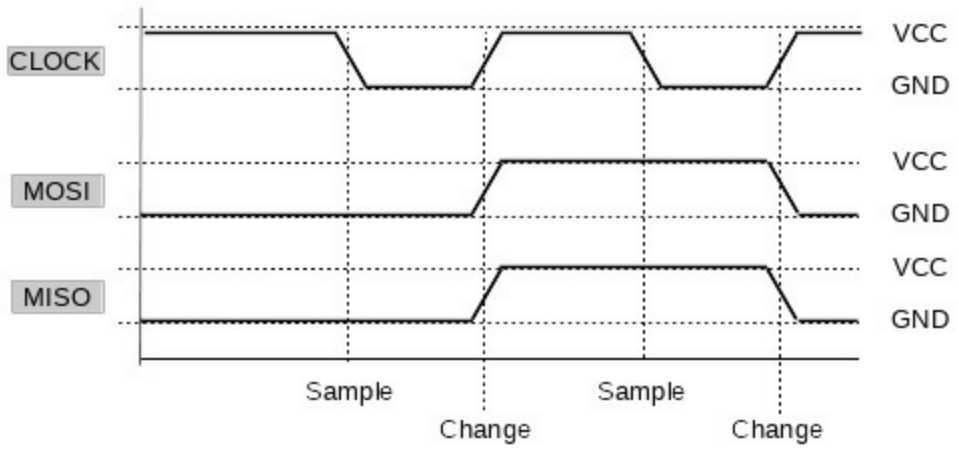


Figure 6.13 – SPI mode 0 timing

SPI mode 1

In this mode, the clock is active high. Data is sampled on the trailing edge of the clock cycle and changed on the leading edge of the following clock cycle:

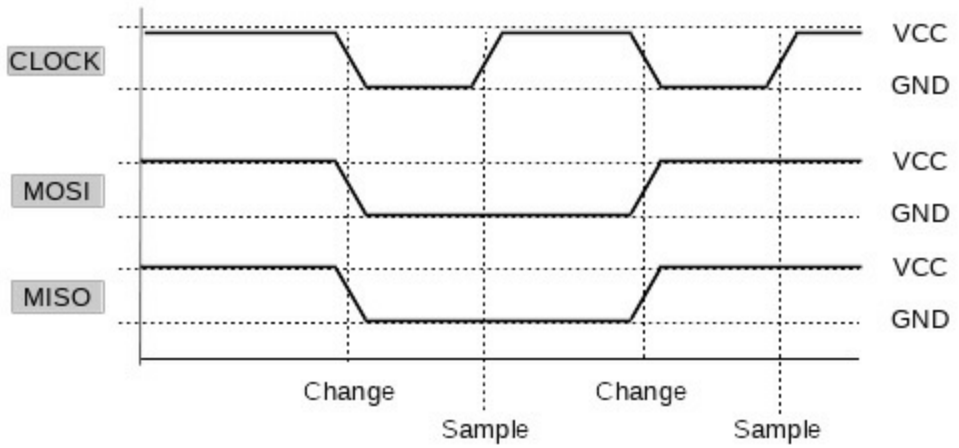


Figure 6.14 – SPI mode 1 timing

SPI mode 2

In this mode, the clock is active low. Data is sampled on the leading edge of the clock cycle and changed on the trailing edge:

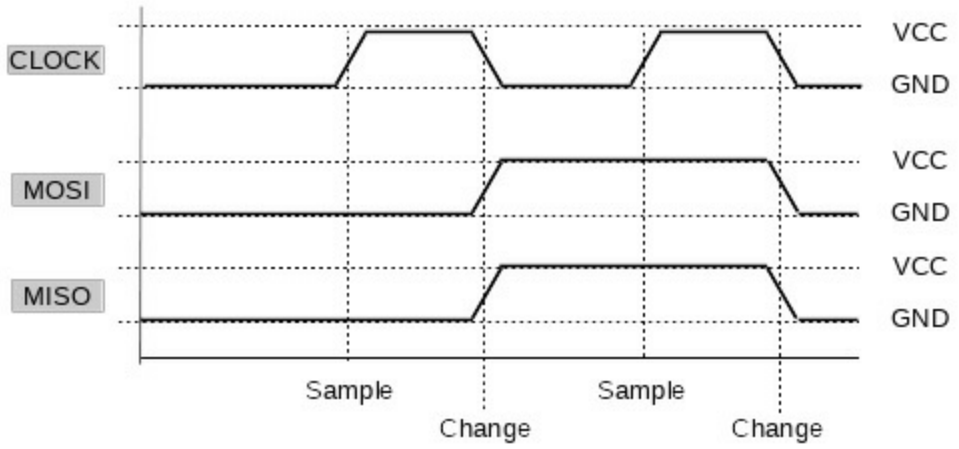


Figure 6.15 – SPI mode 2 timing

SPI mode 3

In this mode, the clock is active low. Data is sampled on the trailing edge of the clock cycle and changed on the leading edge of the following clock cycle:

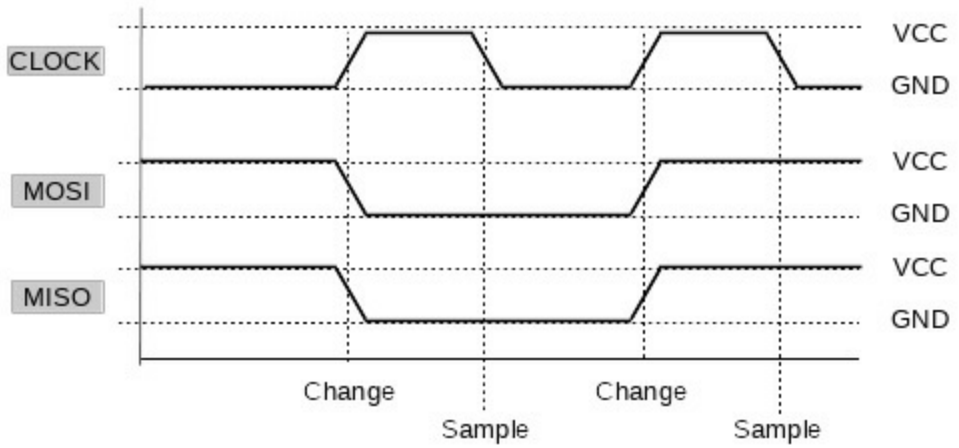


Figure 6.16 – SPI mode 3 timing

Now that we know what SPI is supposed to look like, let's have a look at it.

Sniffing SPI

Like we did for I2C, we will now sniff SPI.

The sniffing protocols are largely the same for all of them: put the circuit using the protocol in place, connect your logic analyzer to the appropriate pins, and launch PulseView. Refer to the *Sniffing I2C* section if you have forgotten.

Build up the circuit as shown here:

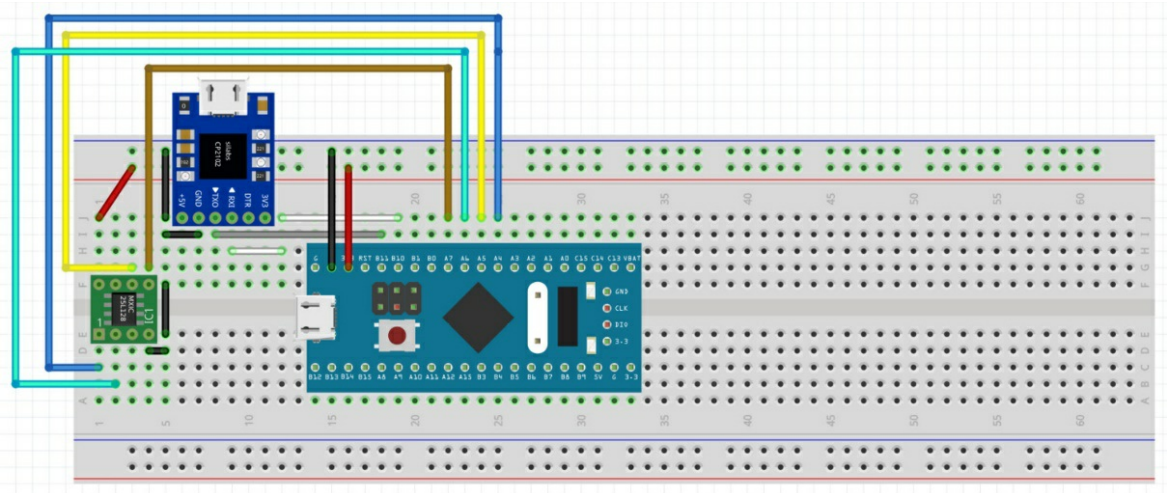


Figure 6.17 – SPI usage connection

This is basically the same deal as for I2C: just connect the ground together, then connect pin 0 to CLK, pin 1 to MISO, pin 2 to MOSI, and pin 3 to CS.

Launch PulseView and add an SPI decoder.

The code for the blue pill is here: https://github.com/PacktPublishing/Practical-Hardware-Pentesting/tree/main/bluepill/ch6/spi_client .

We will follow the same pattern and now inject data on an SPI bus.

Injecting SPI

To inject SPI, just add your master to the bus, the MOSI to the MOSI line, the MISO to the MISO line, CS to CS, and CLK to CLK, and listen to the CLK line to establish a pattern and avoid collisions.

If multiple peripherals are present, you will also have to manage the CS line.

SPI – man in the middle

Put yourself between the master and the slaves; act as a slave to the master and as a master to the slave. An (easily worked around) problem is that most of the hardware peripherals included in the MCUs need an external CS/SS. Just connect to the ground so that the peripheral believes it is always selected if the MCU you use needs it.

IMPORTANT NOTE

Depending on the speed of the communication between the original master and the original slave (and the speeds supported by the original slave), it is not always possible to man-in-the-middle SPI with a micro-controller. Especially with SPI EEPROMs, the communication is not transactional (that is, the MCU finishes asking its questions and then the EEPROM answers). For some EEPROMs, the EEPROM starts to send back data while the MCU is still sending commands. If you have to face this kind of situation,

look into SPI Spy (https://github.com/osresearch/spispy), an FPGA-based tool that can solve this.

Here is the connection schema in the Fritzing folder. Open the Fritzing document here to better see the components and connection points as shown in the following figure:

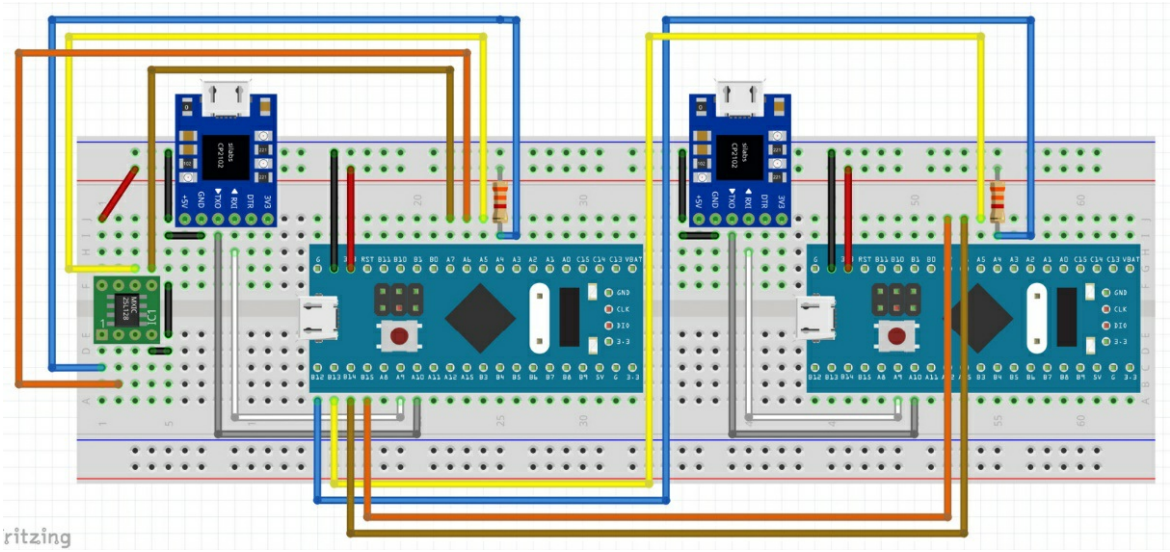


Figure 6.18 – SPI man-in-the-middle connection

The code is here: https://github.com/PacktPublishing/Practical-Hardware-Pentesting/tree/main/bluepill/ch6/spi_mitm .

Now we are going to look into UART (serial link).

Understanding UART

UART (otherwise known as RS232 or serial) is a time-based protocol. The data travels on two wires.

From the MCU point of view, they are named as follows:

- **RX** (**Receive**): The wire on which data comes from the peripheral
- **TX** (**Transmit**): The wire on which data goes to the peripheral

The flow control can come in two main flavors:

- **With hardware flow control** : Two additional control wires control the flow of the data. This hardware flow control itself can come in two flavors: either with control from the master, **CTS** (**Clear To Send**), or from the slave, **DTR** (**Data Terminal Ready**).
- **Without hardware flow control** : UART without hardware flow control only takes care of "transporting the bits." There is no logic layer to it.

Error detection is also possible in the form of a parity bit added at the end of the transmission.

It can connect multiple devices but is not taking care of the addressing (the payload will have to take care of this). It also serves as a base of multiple "flavors" of communication (IrDA, smartcard communication, and more).

Here are the different signals:

Pin name	Description
TX	Transmission wire.
RX	Reception wire.
CTS/DTR	This signal gives the name to the "style" of the flow control: either Clear To Send or Data Terminal Ready.
RTS/DSR	Ready To Send (RTS) or Data Set Ready (DSR).

Now that we have seen the different signals, let's see how they are used to send data.

Mode of operation

In the UART schema, idle lines are high, and the signal is sent by pulling it low.

This is assured by a pullup resistor that is normally taken care of on the TX line (meaning the MCU takes care of the pullup resistor on its TX and the peripheral on its own TX (which is the MCU's RX)). But why is it important, you ask? In the end, there is a pullup on each line. True, but these resistors can be internal to both the MCU and the peripheral and as we will need to put ourselves on these lines, we will need to pay attention that we don't disturb this mechanism too much.

The first parameter to know is called the baud rate, which is the number of bits sent by second. This is a critical parameter since the protocol is time-based. This determines the time of the transmission for one bit.

There is a list of "usual" baud rates and their corresponding symbol times. The symbol time is very practical to determine the correct baud rate to set on your devices:

Baud rate (bps)	Symbol time
110	0.009090909
300	0.003333333
600	0.001666667
1200	0.000833333
2400	0.000416667
4800	0.000208333
9600	0.000104167
14400	6.94E-05
19200	5.21E-05
38400	2.60E-05
57600	1.74E-05
115200	8.68E-06
128000	7.81E-06
256000	3.91E-06

TIP

UART operations are very sensitive to the precision of the clock. Try to always use a crystal oscillator as the clock source since internal RC oscillators (build with resistors and capacitors) tend to be less precise and drift more.

UART transmissions always start with a start bit (to signal the line is not idle and a finish with one or more stop bit). The transmission can be of any number of bits (but is usually 7 or 8 bits long). The transmission can also contain a parity bit to allow for error detection (this bit is optional).

A very common way to describe the settings of serial communication is a string that goes: *Baud rate/number of bits – Presence of the parity bit (Yes/No) – Number of stop bits* .

For example, 9600/8-N-1 is a very common configuration (9,600 bps/8 bits – No parity – 1 stop bit).

Sniffing UART

Sniffing UART with a logic analyzer is very straightforward. Connect the ground together and your analyzer in the usual way: PIN 0 to the RX and PIN 1 to the TX. Then sniff and add a UART decoder in PulseView. You can also just connect the ground of a USB-to-serial adapter to the circuit ground and its RX pin to the direction you want to sniff (do not connect its TX pin as it could disturb the communication).

Injecting UART

The simplest way is to connect the TX pin of a USB-to-serial adapter to the line you want to inject traffic in. This does not always work because your TX pin could pull the line too high for the original sender to transmit.

Here is an example situation:

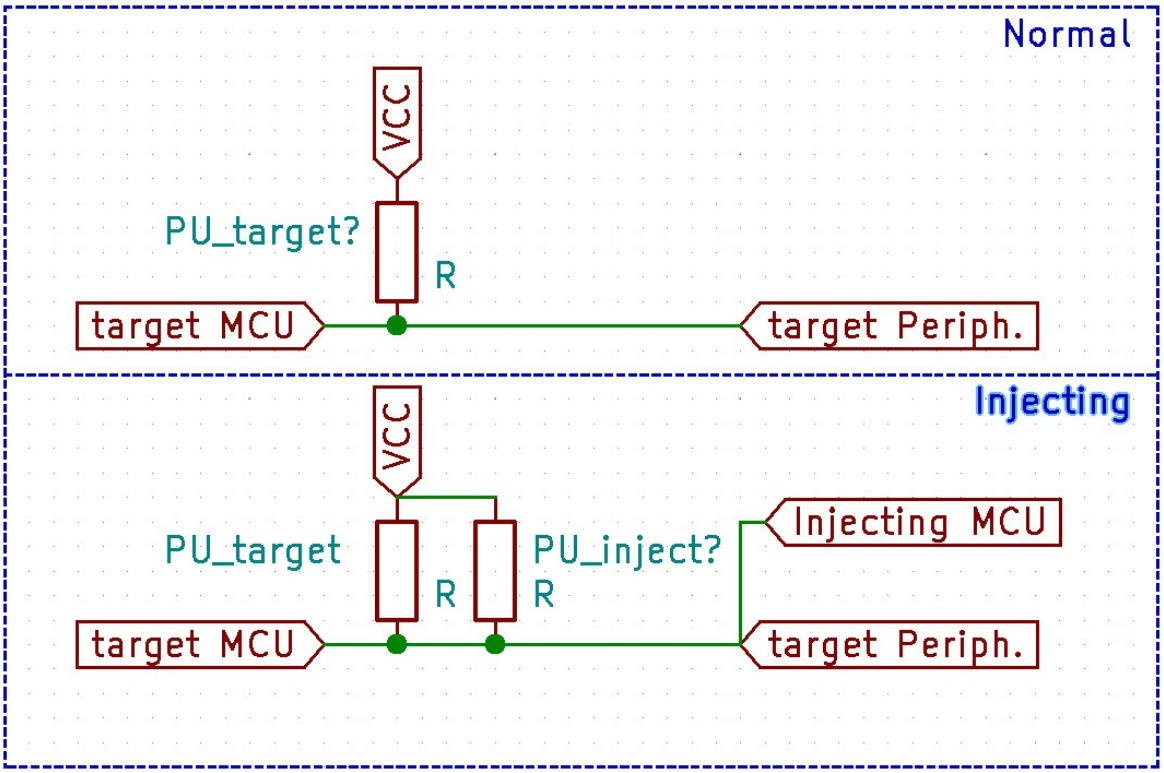


Figure 6.19 – The problem with UART injection

In a normal situation, the MCU pin pulls the pin to ground with a given strength (usually it is quite "strong," with very low resistance). This changes the voltage on the line to be very low (*not* null, since there is still a little bit of resistance; the pullup and the resistance act as a voltage divider), low enough to be under the threshold that the peripheral pin detected as `0/LOW` .

When we add our pullup in parallel to the normal pullup, we actually lower the resulting resistance (resistors in series = sum of resistances. Resistors in parallel = sum of the inverse). This means that it is possible that we (or the original MCU) aren't able to pull the line low enough for the MCU to detect a `LOW` (this also means that too much current can flow through the MCU's or your UART adapter's pin, damaging it in the process).

In this case, do the following:

- Remove the pullup on your adapter if possible or change the value of your pullup resistor (to a higher one; the already-present pullup of the peripheral TX will act as a divider).
- If this doesn't work, go in the man-in-the-middle direction.

Exercise

Adapt the code of the UART script of the I2C example to inject your UART traffic instead of showing the I2C traffic (get inspiration from the bit of script in the next section).

UART – man in the middle

You can use two USB-to-serial adapters on your computer and use a simple Python program to alter the content of the communication (do not forget to cut or disconnect the original connections).

For example, the following code adds `1` to every byte received on `ttyUSB0` and sends it to `ttyUSB1` :

Now we are looking into a protocol that we have to bit-bang since there is no hardware peripheral for it. D1W is a pretty nifty protocol that is used for simple applications such as ensuring that a guard did their rounds. With what I will show you in the next section, you will be able to take a guard job and stay in your sentry box during cold winter nights

Understanding D1W

D1W is a one-wire bus. It is usually used for simple sensors (temperature or humidity) and has "buttons" that just show a unique identifier. This is an interesting bus where the power of the device can also come from the wire that is used to transmit data. This is usually not supported by hardware peripherals in MCUs; you need to bit-bang the protocol. Bit-banging a protocol means that we will implement the protocol manually by using the GPIOs of the blue pill. 1-Wire is an open-drain bus (like I2C or UART) and hence needs an external pullup resistor (usually of 5k ohms) to set the voltage to a known state when the MCU disconnects the pin (also called floating as in the code).

Mode of operation

The communication on the D1W is time-based and is initialized by sending a reset pulse that the slave will answer to (the presence pulse).

The reset pulse

The reset pulse is initialized by the master pulling low the data line for at least 480 µS.

So, the first challenge is to set up the GPIO. For this, we will use and have a precise enough time source to measure the 480 µS. So, we will need to use a timer. Please have a look at the **client** (since the man-in-the-middle code is event-driven, it is better to start with a more sequential program) code for this in the GitHub repository here: https://github.com/PacktPublishing/Practical-Hardware-Pentesting/tree/main/bluepill/ch6/1w_client . The setup code is in `libLHP .`

The presence pulse

Now the master listens on the bus for at least 480 µS and each device will

pull down the line for 80 to 240 µs as shown in the following figure:

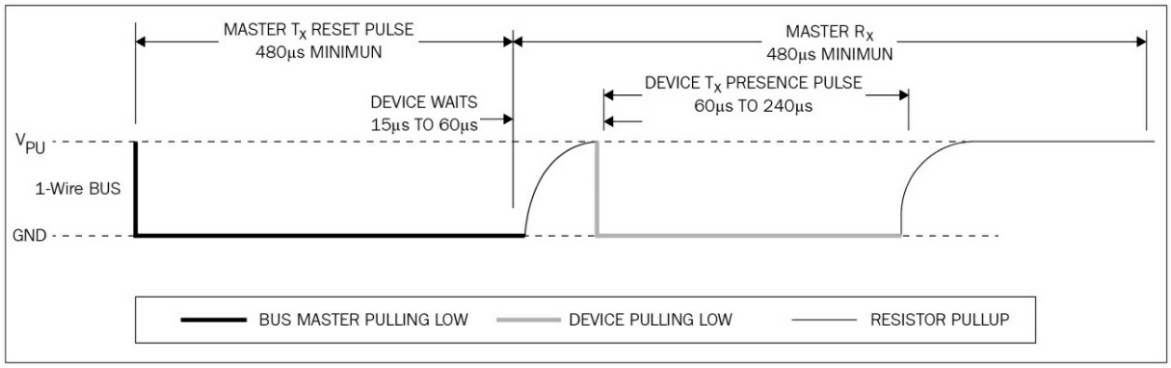


Figure 6.20 – D1W timing diagram

Now that we have seen what the presence pulse is, let's look at the other operations.

Reading and writing

Basically, all reads and writes are started by the master. To write a 0, the master pulls down the line for 80 to 120 µs and to write a 1, it pulls down the line very shortly (1 µS) and lets the resistor pull up the bus.

To read, the master pulls the line low and measures whether the bus goes high in the next 15 µs. If it does it is a 1 and if it's doesn't it's a 0. All the timings are described in the MAX31820 datasheet (https://github.com/PacktPublishing/Practical-Hardware-Pentesting/tree/main/datasheets) on *page 17* :

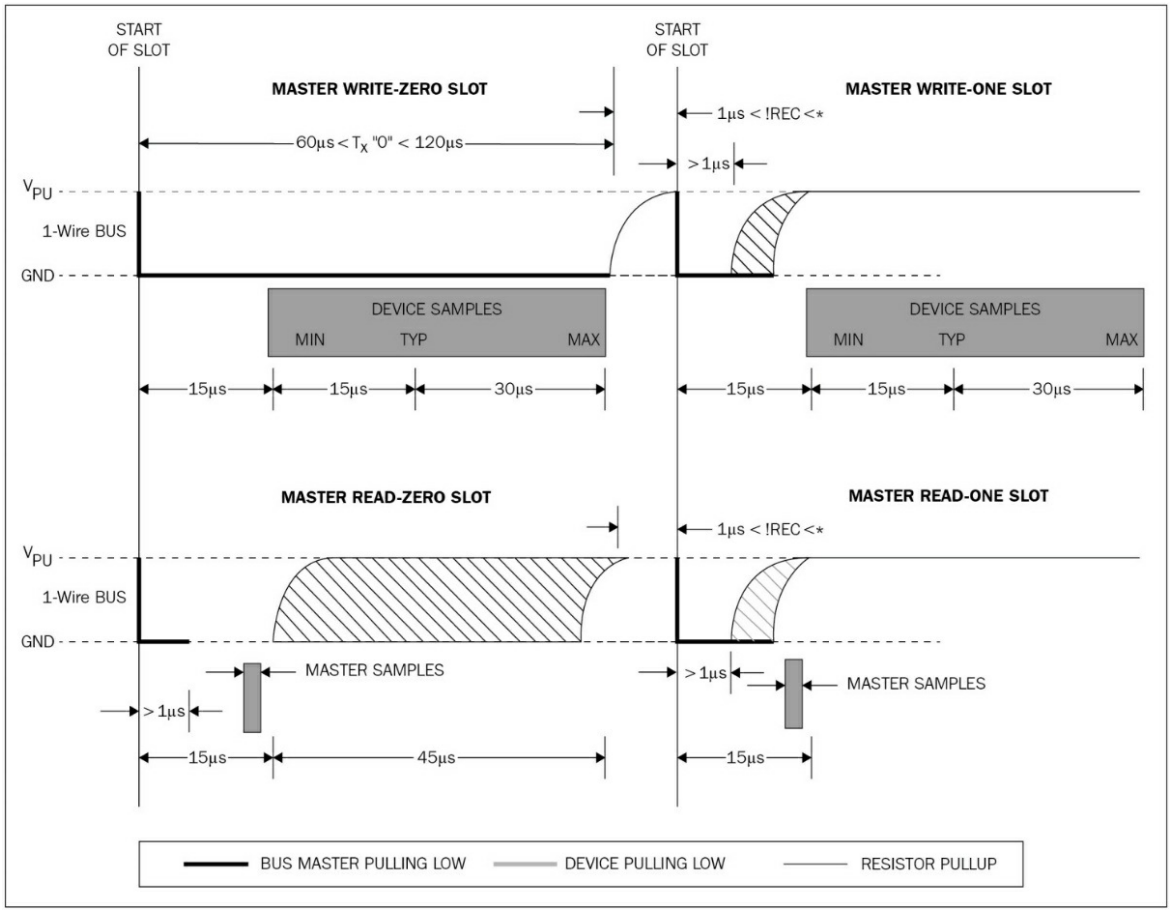


Figure 6.21 – D1W timing diagram, READ and WRITE

D1W also uses a **Cyclic Redundancy Check** (**CRC**) to enable the detection of errors. Please refer to the D1W documentation to learn how to use it (this will have an impact on the man in the middle since the CRC has to be corrected if we change data).

Sniffing D1W

Here is the connection schema: https://github.com/PacktPublishing/Practical-Hardware-Pentesting/tree/main/bluepill/ch6/fritzing_Schematics . Open the Fritzing document here to better see the components and connection points as shown in the following figure:

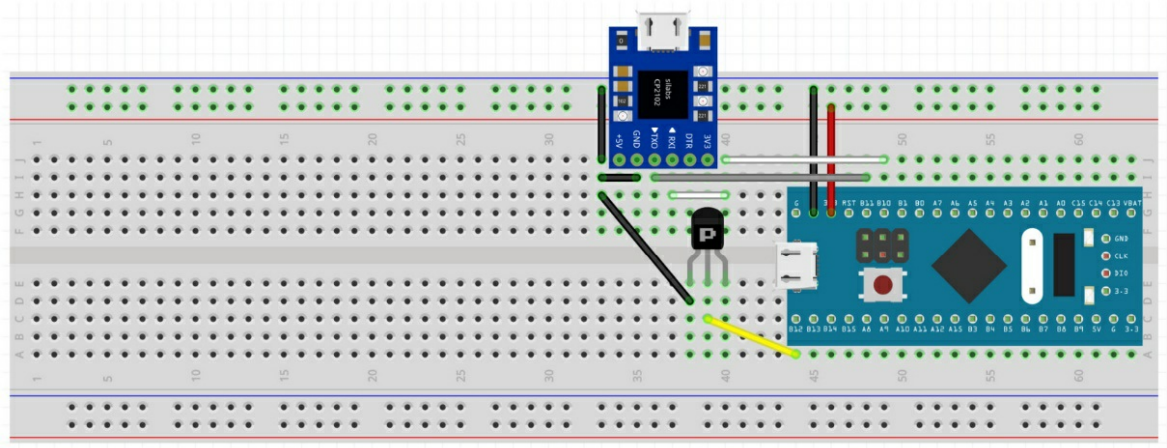


Figure 6.22 – D1W connection

As usual, connect the ground together, connect pin 0 to the data line, and sniff and add the D1W analyzer.

Injecting D1W

Connect a master to the data line and just act as the master (sending requests and reading answers).

The communication on the D1W bus is (in the vast majority of cases) very spaced out. The chances of collision are very low but if you get a CRC error, just wait for a few ms before retrying).

D1W – man in the middle

Open the Fritzing document to better see the components and connection points as shown in the following figure:

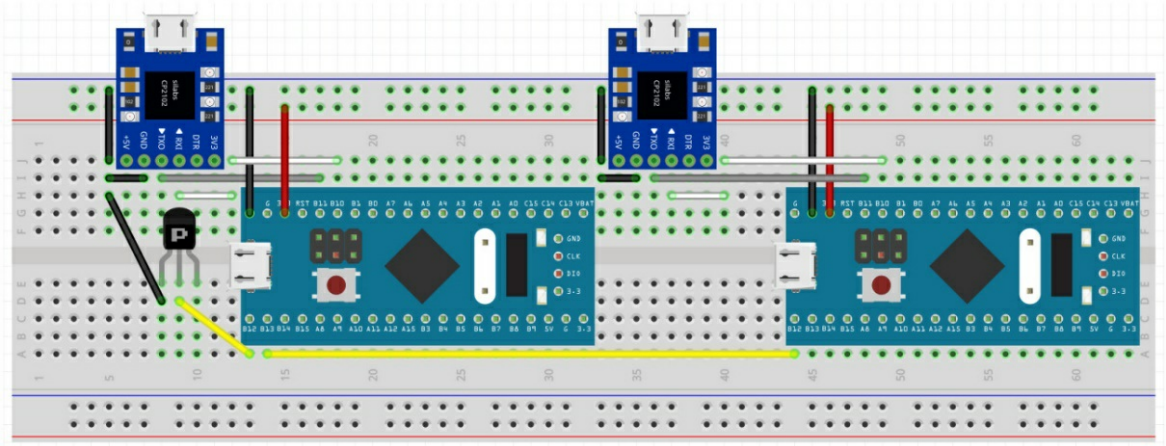


Figure 6.23 – D1W man in the middle

This is basically the same deal as usual: act as a slave to the master and as a master to the slave. Here we are modifying the temperature readout and the CRC.

Summary

In this chapter, we have seen the most common of circuit protocols, how to sniff them, and how to man-in-the-middle them. This will allow you to take control of most of the slower-speed protocols and will provide you with the necessary tools and approaches to alter the behavior of a system and find secrets that are exchanged on the wires.

In the next chapter, we will learn how to identify the access points to these signals and, if necessary, how to create our own access points.

Questions

1. You are visualizing something and you are pretty sure there is some UART traffic on your scope. You see the following waveform. What is the baud rate?

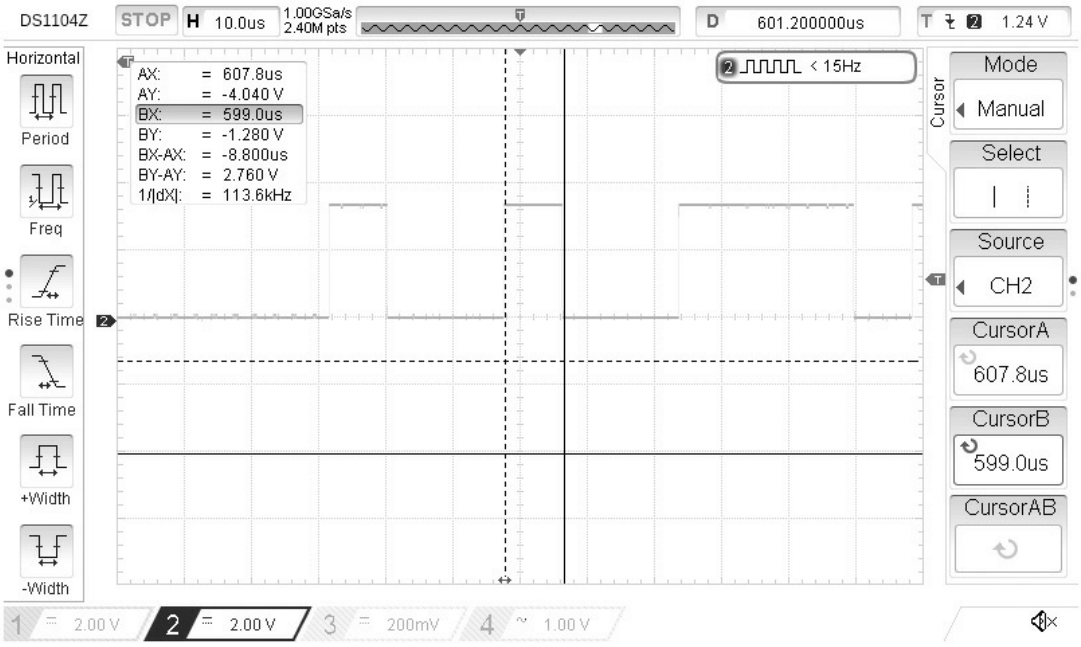


Figure 6.24 – UART oscilloscope signal: what is the baudrate?

2. What is QSPI?
3. What is the usage of the parity byte in UART?
4. Who invented the I2C protocol?
5. How can you use multiple 24LC EEPROMs on the same I2C bus?
6. You have to man in the middle an I2C bus with two different devices and a master. Sadly, the hardware peripheral on the blue pill can only have a single address and you can't think of a way to have it alternate between the addresses (the master apparently talks randomly to the devices). What would be your approach?

```
0x41 0x20 0x76 0x65 0x72 0x79 0x20 0x76 0x65 0x72 0x79 0x20 0x73
0x65 0x72 0x69 0x6f 0x75 0x73 0x20 0x6b 0x65 0x79 0x21

0x08 0x00 0x1a 0x0a 0x04 0x1c 0x00 0x14 0x0c 0x1c 0x18 0x52 0x0a
```

0x45 0x1d 0x19 0x0a 0x07 0x12 0x54 0x04 0x17 0x0a 0x00 ?

Feedback

We are constantly looking at improving our content, so what could be better than listening to what you as a reader have to say? Your feedback is important to us and we will do our best to incorporate it. Could you take two mins to fill out the feedback form for this book and let us know what your thoughts are about it? Here's the link:
https://packt.link/HardwarePentesting2E .

Thank you in advance.

Embedded systems store their data and sometimes their code on media that can take multiple forms on the board (chips, external storage such as SD cards, and so on). Getting access to this storage is crucial to be able to analyze the code and get access to security-relevant elements. In this chapter, we will go through multiple components that can hold this data, how to extract data from the component, how to understand how it is stored, and lastly, how to peruse and change the data (in raw form or with a filesystem).

The following topics will be covered in this chapter:

- Finding the data
- Extracting the data
- Understanding unknown storage structures
- Mounting filesystems
- Repacking

Technical requirements

The software required is as follows:

- Binwalk
- A Linux machine

The hardware required (to repeat the examples) is as follows:

- The Bus Pirate
- An I2C or SPI EEPROM (you should already have this from the previous chapter)

Check out the following link to see the Code in Action video:

https://bit.ly/383OzkJ

Finding the data

Before parsing the data, we have to find it. In addition to classical storage media (hard drives, **Solid State Drive** (**SSDs**), onboard USB storage, and more), embedded systems use more specific chips and systems to store data, and some of them are listed as follows:

- EEPROMs
- EMMC and NAND/NOR Flash
- Static RAM, and so on

Let's look at each of them in the following sections.

EEPROMs

EEPROM (**Electrically Erasable Programmable Read-Only Memory**) and flash memory are "one-chip" storage solutions that keep the data even when the power is off. They are available on pretty much every existing protocol (I2C, (Q)SPI, 1-Wire, and more). Locating these chips is not always easy (especially if they are unmarked or rebranded) but (as already discussed in the component identification section in the previous chapter), it is possible to identify them by elimination or by sniffing the protocol on the board. Typically, the storage capacity is small, and the storage structure is custom made for the system (that is, it does not embed a typical, well-known filesystem like bigger storage mediums).

EMMC and NAND/NOR Flash

EMMC is a physical variant of **MMC** (**MultiMediaCard**) in which the chip is soldered on the board instead of being removable. It is entirely possible to remove the chip (doing so is called chip-off) and use an adapter to read it as a classical MMC (with a USB adapter), basically transforming it into a USB thumb drive.

NAND and NOR Flash are also soldered directly on the board but don't really offer a "standard" protocol to talk to the chip and need a lower-level approach (such as programming the adequate protocol on a micro-controller) or a specific adapter/programmer.

These chips come with multiple "standard" footprints (EMMC: BGA221, BGA162, BHA186, and more; NAND/NOR: BGA137, BGA63, and more) that all require a different adapter (to accommodate the footprint). These adapters can be found on auction sites or from Chinese retail sites (AliExpress, TaoBao, and so on). Single-use adapters are reasonably priced but require a reflowing phase (that is, resoldering the chip on the adapter, for which you will very probably need a paste application stencil), which can be tricky to master and comes with a risk of damaging the chip (it is pretty common to short pins while manually reflowing a BGA chip).

Reusable ones (with a clamshell adapter) are more expensive but (if you are doing this often) are worth the price since you will bypass the reflowing step and the risks that come with it.

Hard drives, SSDs, NVME and other storage mediums

While less common on simple systems (such as systems running from simple micro-controllers), it is pretty common to find systems that are running a full-fledged OS (that is, the same kind that runs on a laptop or desktop computer) for the simple reason that the system is a computer in the commonly understood sense of the term.

For example, anything running Linux kernel (such as Android) is a complete computer and usually comes with high-speed data interfaces (PATA/SATA, M.2, PCI-E/X, and more). It is then pretty tempting for a system constructor to leverage the relatively cheap storage price by using commonly available interfaces, especially given the small form factors currently available. M.2 sticks or 1.8" SSD drives are very small and provide huge capacities compared to the usual specific embedded solutions.

Some older styles of storage are also found on more legacy/industrial systems

(for example, CompactFlash/Microdrive) that need specific adapters. USB adapters for these styles of storage are quite commonplace on auction sites. For these formats, general purpose USB adapters (for compact flash cards, for M.2 MVME sticks, etc…) are available. They will present the device as a generic storage decive to your computer.

Extracting the data

For cases where we don't have the data already (that is, we did not succeed in getting updates), we need to extract the data from its storage place to our computer. Being able to process and modify the data on a computer will allow us to use higher-level programming languages and tools.

Let's have a look at the most common things we have to extract.

On-chip firmware

Most micro-controllers will embed their programs (that is, their firmware), at least partially, on on-chip (or on-module) flash or other forms of storage, such as EEPROM. The worst-case scenario for us is cases where programs are stored in **One-Time Programmable** (**OTP**) memory, such as the MCU used in furbies or most low-end calculators for example (they can come with in a mask ROM that is integral to the making of the chip silicon or a real memory that can be programmed only once) or a lot of very cheap MCUs.

For example, most ARM chips come with on-chip flash. The ESP family of chips has a flash storage chip on the module from where the chip retrieves its program. These can usually store long-term variables (across reboots). It is very important for us to be able to retrieve this data if we want to be able to reverse-engineer the program behavior.

An essential step in acquiring the firmware is to find an adequate hardware programmer and the associated software. Most chips will use some form or variation of the JTAG interface (we will talk in more detail about JTAG in *Chapter 10* , *Accessing the Debug Interfaces*). In modern chips, it is very common to find the correct hardware programmer (bundled with the software) integrated into the development kit for the target chip. This programmer usually allows us to read back the binary form of the program that is stored in the onboard flash. Some commercial products implement protection against reading back this data, but they can sometimes be bypassed. Bypassing these protections usually requires some more advanced

and specific attacks (such as finding bugs in bootloaders or glitching), but these attacks are usually complex and fall out of the scope of this introductory book.

Depending on the complexity of the device, it is also possible that a bootloader (for example, U-Boot) will be present on the device. If this is the case and you manage to get access to it (from a debug serial console, for example), it should be possible to extract the storage via the serial cable; for example, U-Boot's `md` or **nand** commands can help:

- https://www.denx.de/wiki/view/DULG/UBootCmdGroupNand
- https://www.denx.de/wiki/view/DULG/UBootCmdGroupMemory

Onboard storage – specific interfaces

Unless the device is supported by mainstream tools (such as flashrom), you will have to implement a specific dumping tool on a micro-controller or use a tool such as the Bus Pirate to access the content. Flashrom is usually used for BIOS flash chips but also supports many other flash chips, for example, the Macronix 25L8008 SPI flash we used in the previous chapter.

The details of the behavior can be found by reading the datasheet. To dump the content to your computer, you will have to implement this behavior. This is usually implemented on a micro-controller and the data is transferred over serial. This transfer is achieved using a USB-to-serial device and a variant of the script we used to man-in-the-middle UART. The exercise to modify the script to write to a file instead of writing to the other UART serial bridge is left to you.

Dedicated chip programmer are also very useful for this purpose (Such as the very popular TL866 or T56 from Xgecu). They can read and write a wide variety of devices, from DIP I2C EEPROMs (as seen in the previous chapter) to very large BGA flash chips (such as the ones used in mobile phones or on a Raspberry Pi).

Onboard storage – common interfaces

If the device uses a standard interface (SATA, MMC, SD card, or others), it should be recognized (at least as a device) by your computer. It should show up in your logs (you can display your log with the `dmesg` command) as being available as a device (in the `/dev` directory).

For example, my USB adapter connects and detects an 8 GB micro SD card as `/dev/sdg` (this specific adapter is based on a Realtek RTS5169 chip and supports multiple media, CompactFlash, SD card, memory sticks, and more):

If your media is recognized like this (here, it shows `sdg: sdg1 sdg2`), the best tool to image it is `dd` . `dd` is a part of `coreutils` (and is most probably installed by default on your Linux box).

`dd` has a peculiar syntax for a Linux command-line utility, without the usual `-` or `--` as a flag indicator. The options you will want to use are as follows:

- `if=<path to the input file>` : This can be `/dev/ device` (in the end, you are using a Unix box, where everything is a file).
- `of=<path to the outputfile>` .
- `bs=<block size>` : This support `k` , `M` , and `G` shorthands; usually `1M` will do.
- `oflag=<a comma separated list of flags for output>` : I personally like to use the `sync` flag here, especially when writing to flash devices, in order to ensure that the data doesn't end up in a kernel buffer, which makes `dd` quit while the kernel is actually still writing.
- `status=progress` : (Only available in recent versions of `dd` .) This will show you a progress indication (older versions of `dd` will only print this when they receive a USR1 signal; use `kill -USR1 <dd process id>` if you have an older version).

Take the following example:

- To dump an SD card to a file, use the following:

```
dd if=/dev/sdg1 of=./dump.bin bs=1k status=progress
```

- To put the modified data on an SD card, use the following:

```
dd if=./dump_modified.bin of=/dev/sdg1 bs=1k oflag=sync
```

```
status=progress
```

Now that we have saved the storage to our machine, let's look into it.

Understanding unknown storage structures

More often than not, light systems (those not embedding a full-fledged OS such as Linux) will have a pretty well-documented way of storing their firmware internally (since this storage form is crucial for the target MCU to function properly, it is well described in the target MCU datasheet). On the other hand, the way the data is stored by the firmware itself is very much left to the firmware developer device.

Unknown storage formats

There is no definitive way to reverse engineer the way data is stored, as for most reverse engineering, it is as much an art as it is a science. The only way to get a good knack for it is, just like soldering, doing it again and again, but having spent a fair share of my time reversing a lot of different things, such as network protocols, storage structures, and more, I can give you some pointers.

Understanding the way the data is organized for storage depends on multiple factors. There are some general hints that can help you along the way.

Note the following about the data itself:

- Will the data change a lot in terms of content (such as settings or readings) or is it "mostly static" (such as firmware and executable code)?
- Will the data change a lot in terms of size (such as strings or structure data with members that are optional) or is it mostly static (such as binary readings, binary fields, and so on)?
- Will the changing data be of variable size or is it well organized? Will it be of fixed-size chunks?

Consider the following data processing commonalities:

- The developers will tend to store similar data together (images next to images, text together, and so on).
- Code reuse. For example, when you look at compression, the same algorithm will be reused multiple times, and hence block and code structure will be repeated, and so on.
- Storage optimization. For example, data that would be relevant on a per-file storage basis on a computer wouldn't make sense on a space-constrained system (why keep image data headers if I am guaranteed by construction that I will only store 64x64, 24-bit, per-pixel color image information?).

Consider the following for the storage media:

- How is the storage media itself organized?
  - - Is the chip organized in blocks, pages, or sectors?
  - - Is it easy to make random, relatively small accesses or does the MCU have to read "big chunks" of data?
- How is the storage media behaving regarding large numbers or write cycles? (For example, flash only supports a given number of write cycles before dying. If the chip controller is not implementing wear leveling (to spread data in order to avoid killing blocks prematurely), the firmware author may have been tempted to implement a home-made system for that.)

Consider the following for a classical storage scheme:

- **FAT** : FAT is well known as a filesystem, but this filesystem actually gets its name from the concept of **File Allocation Tables** . It is very common to put a "table of contents" in front of your storage, with the start address and size of your storage item (possibly with a filename and other attributes such as timestamps of modification, and so on).
- Size and value pairs, the same way it is used in a lot of network protocols.
- Fixed-size "blocks," for example, 100 bytes for preference, then 2 KB for static strings, then 5 KB for pictures, and so on.

So, understanding the way storage is organized requires some detective work and does not necessarily involve well-known structures and mechanisms.

Well-known storage formats

Sometimes, the storage format is well known (because someone reversed and documented it before or because it uses well-known mechanisms) and tools are available to extract it. One of the best known and most extensive in terms of support of different packing mechanisms is Binwalk (https://github.com/ReFirmLabs/binwalk).

Binwalk will search the target file for well-known headers and try to extract them for you (with `-e`).

Binwalk is Python-based and is a very useful tool to analyze firmware and storage images (even if the format is not a well-known one, as it contains tools to help you analyze it). You really should read the documentation (https://github.com/ReFirmLabs/binwalk/wiki) and train yourself on multiple firmware images (router updates are really ideal for that).

Binwalk will be able to find the following:

- General compression formats (`.gz` , `.lzma` , `.xz` , and so on)
- Linux kernels and images
- Filesystems (SquashFS, JFFS2, and so on)

Dumping Chips

Let's look have a look into dumping the two most common storage medium you will find in your assessments. After dismanteling a commerical children toy (a Furby to be exact), I identified two possible places the data could be stored : An I2C EEPROM and a SPI flash.

First candidate – An ATMLH306 I2C EEPROM

The first data store we identify is an I2C EEPROM. Let's dump it to a file. This step can be repeated with the I2C EEPROM you used in the previous chapter.

Extract it from the **Printed Circuit Board** (**PCB**), mount it on a breakout,

and connect it to our Bus Pirate as shown in the following photo:

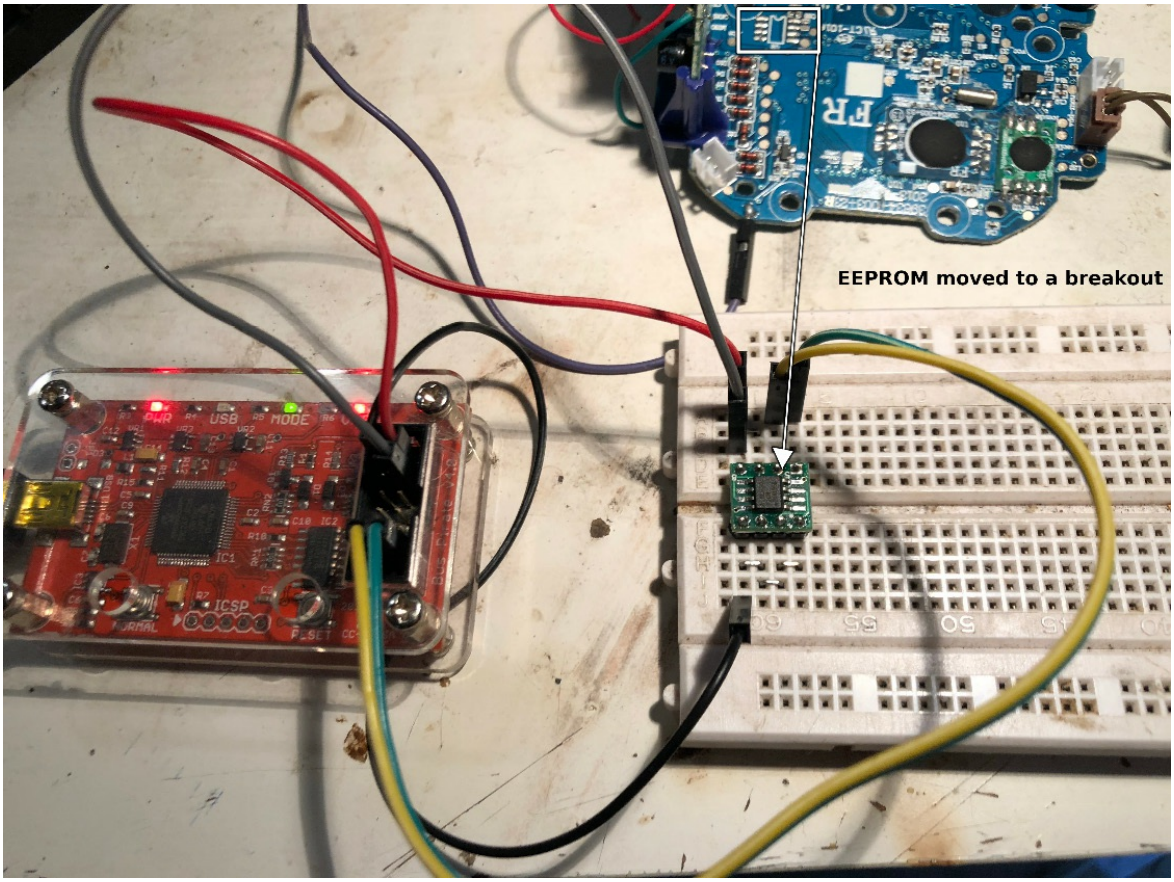


Figure 7.1 – I2C EEPROM extracted and connected to the Bus Pirate

Once the Bus Pirate has connected successfully (it will show in the output of `dmesg`), we can go on.

Let's verify that our connection works by scanning the I2C bus with the following actions in the Bus Pirate text menu:

Depending on your version, the Bus Pirate output can be slightly different. Now we will connect to the Bus Pirate in binary mode using a Python script and save the EEPROM to a file. You will be able to find the script and the dumped data on the GitHub repository in the `ch7` directory to output the content to a file (and the output if you can't find a Furby).

Now let's look into this file (The file is available on the book’s github repository):

We are starting by just doing a hex dump to look at the general content (Is there a lot of data? A lot of 0s? What does it look like?).

This EEPROM is very small, but looking at the content, we can already see some kind of structure emerge. There is a lot of symmetry between the left side (the first 8 bytes) and the right side (the next 8 bytes). We don't have a lot of data to start a static analysis (and certainly not to launch Binwalk on) but this is promising for the dynamic analysis (that is, changing the EEPROM content and seeing what happens). We can deduce from the size of the storage that the sounds and the eye pictures are not stored here (too small).

Let's now look at the second storage chip.

Second candidate – the FR-marked SPI blob

The SPI chips are a little more complicated and contain more data, and we will approach it in two steps (dump and analysis). By covering I2C and SPI, you will be ready to face most on-board storage for systems that use bare-metal MCUs. More complex systems (such as ones that embed a full-fledged Linux system) will usually use parallel flash chips that require more advanced soldering skills and equipment to dump (but are largely within your reach with training).

Dumping it

We know this is an SPI chip (from the markings on the traces that were leading to it, CLK, MISO and MOSI), so let's remove it from the board and put it on a breadboard.

SPI flash are usually compatible with a standard called JEDEC. This standard allows us to identify the exact chip by reading the JEDEC identifier through standard commands/.

We can get this identifier with this script :

0xC2 : this is a Macronix chip (https://www.jedec.org/standards-documents/docs/jep-106ab)

let’s have a look at the IDs know by flashrom : https://github.com/flashrom/flashrom/blob/master/include/flashchips.h

Bingo ! : #define MACRONIX_MX23L3254 0x0516

This is a MX23L3254, a mask ROM (we can’t reprogram it but the MX25L8008E happens to be a perfect drop is replacement)

We should be able to easily dump it with flashrom. Let’s give it a try.

NB : Flashrom have troubles with Bus Pirate v4, use a Bus Pirate v3.

The original chip is a mask ROM, so we cannot change the content.

The MX25L8008E we used in *Chapter 6* , *Sniffing and Attacking the Most Common Protocols* , is a perfect, writable drop-in replacement.

Having a well-stocked component stock and keeping a decent amount of questionable e-waste around will both (and just as often!) create routine "is it really useful?" discussions with your significant other but may save your hide during engagements.

Unpacking it

We know from the presentation that Binwalk and strings will not yield results. But let's try anyway.

Dealing with strings

There are two command-line tools of interest:

- *Strings* well
  . Something a lot of people overlook is trying all the different possible encodings (depending on what you are analyzing, this can be relevant, especially if you are dealing with Windows executables and DLLs or non-Latin alphabets; you will find a script in this chapter's folder do automate this: `string_all_enc.sh`).

- *iconv* converts between different encodings. This is especially useful for non ascii composite characters, such as é, ñ, and so on.

Dealing with packed data

Binwalk will try to peruse a file for known formats but will also allow us to have an overview of the entropy in a file (look into https://en.wikipedia.org/wiki/Entropy_(information_theory)). Entropy is roughly the measure of how random data looks. Measuring the entropy is a good way to get an idea of whether the data you are looking at is cyphered or compressed and the global layout of a file.

Launch this command:

The following figure shows the entropy in the file:

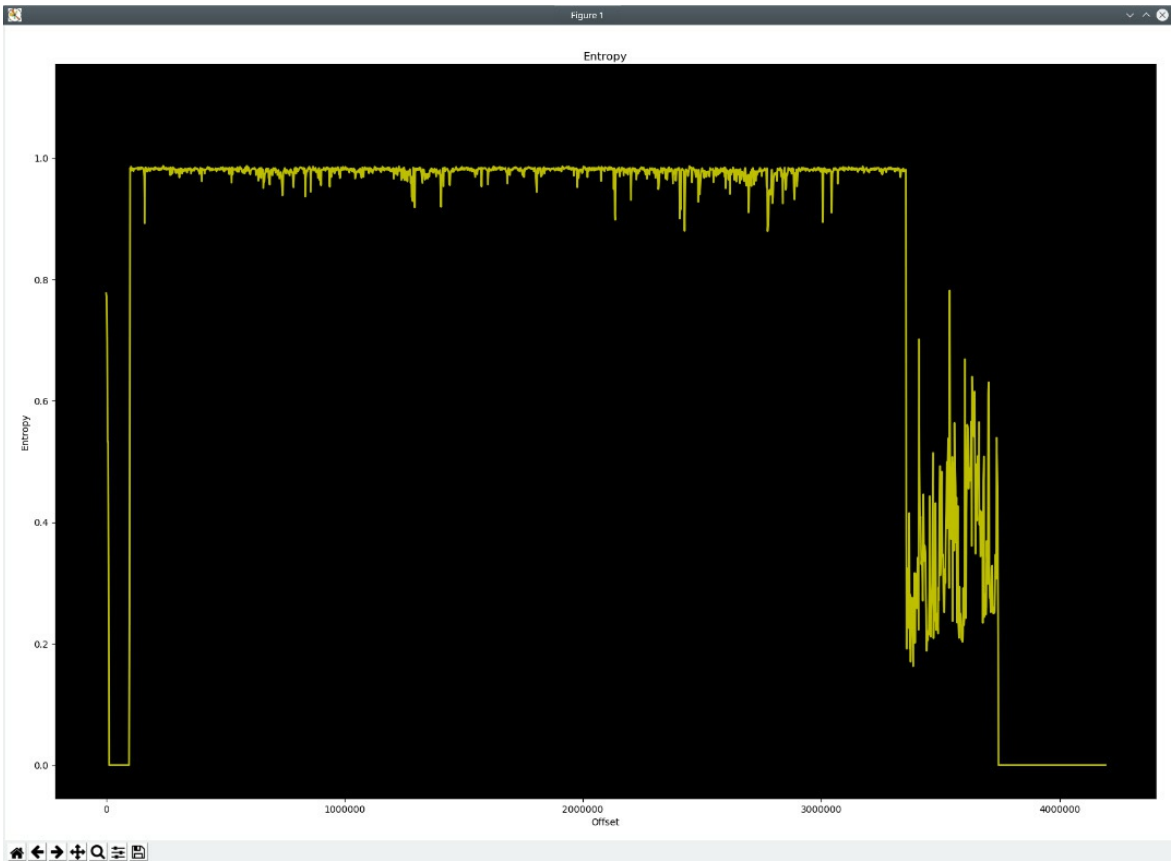


Figure 7.2 – Entropy in a file

We can clearly see four zones:

- The header zone, which should contain information about the organization of the file
- A big, very entropic zone
- A medium zone with widely varying entropy
- An anentropic zone

These are probably the following:

- The zone with the offsets
- Image or sound data (we remember from the patent that the sound may be voice-synthesized, taking up much less space than actually digitized sound)
- Image or sound data (well, the other type compared to the previous section)
- Padding (the data is actually full of `0x01` at the end)
- Michael's parser works on our data and can extract pictures, great!

We can see that the data Michael found and ours differ:

His data has 2,806 pieces of data and ours has 2,820. Even if it is the same version of the toy (2012), we may have another firmware version.

I'll leave it as an exercise to you to write a small script to identify the zones with the two entropies:

- The first zone (with the entropy's "plateau") is the sound data.
- The second is the images.

When we look at the organization of the storage in the header of the EEPROM, we see that it is based on the concept of FAT, with the number of "files," and the offsets to the different files with their size stored in front of their content. This is a very classical organization scheme.

Mounting filesystems

The `mount` command (you have to be privileged to use it; use `sudo`) is the main tool for this.

Modern versions of the command recognize the filesystem automatically. If the detection is not working but you know the filesystem in use, the `-t` option will allow you to force the filesystem format to be used.

To list the filesystems your kernel is currently supporting, look into the `/proc/filesystems` file (as a side note, not all modules can be mounted; to get a list of what it does support, look into the `/lib/modules/$(uname -r)/kernel/fs` directory).

Some filesystems used in embedded systems may not be supported in some usual distribution kernels and so you may need to do the following:

- Recompile your kernel with more filesystems.
- Compile additional modules for your kernel.
- Use userspace filesystem management (such as FUSE).

Since most of the firmware or storage images we get are in the form of a file instead of a block device, some options are useful for managing this specific case. They are managed through the `-o` command-line switch. This switch uses a comma-separated list to manage multiple options (whether the options are global or filesystem-specific):

- `loop` : Makes `mount` use a file as a block device
- `offset=xxx` : Skips `xxx` bytes in the target block device

Repacking

The repacking process is mainly taking the reverse path we took for packing, recreating a consistent image with the modifications we want.

I would strongly encourage you to look into the firmware modkit if you need to repack routers and other xx-WRT-based firmware (https://code.google.com/archive/p/firmware-mod-kit/).

Since most of the standard filesystems that are mounted from a file with a `-o` loop option will be read-only, a common approach is to work on the files on a normal directory on your computer, create an empty image of the necessary size, recreate an empty filesystem, and copy the files onto it.

Some systems may not implement the filesystems completely and you may need to tailor the filesystem creation (or use specific versions) for it to work with the final target system.

Summary

In this chapter, we saw the different media that can be used in embedded systems and the tools we need to approach them, extract them, understand their structures, and modify them. Since the ways to store data are very variable from one system to another, it is not possible to go through every possible variation but, after reading this chapter, you will know (at least partially) the possible tools that you can use, how things are generally organized, and some concepts you could think about when reverse-engineering storage schemes. These tools are very powerful but, like any tool, are limited by the skill of the person that uses them. That's why you should practice and read the documentation of the tools as much as possible.

In the next chapter, we will look into how to modify the stored elements and, from the changes in the system behavior, better understand the structure of the stored data.

Questions

1. What tool can you use to take an image of a peripheral that is recognized by your Linux machine?
2. What is the use of the `-o loop` command-line switch for `mount` ?
3. Why are the lists in `cat /proc/filesystems` and `/lib/modules/xxx/kernel/fs/` different?
4. You found a module marked `eUSB` on a device you are testing. What is it? How would you read it?
5. What is the eMMC standard? How would you read it?
6. What is FUSE? What is user space? How can you use it?

Further reading

Read the `mount` , `iconv` , `dd` , and Binwalk documentation (use the `man` command). Look at the firmware modkit wiki, and check how to recompile a kernel or modules for your distribution.