REPUBLIC OF TUNISIA
Ministry of Higher Education and Scientific Research
University of Carthage
National Institute of Applied Sciences and Technology

Graduation Project

to obtain the

National Engineering Diploma
in Applied Sciences and Technology

Specialty: Instrumentation and Industrial Maintenance

Wireless Serial Wire Debug interface

Prepared by

Firas ISMAIL

Hosted by ACTIA ES

Defended on the 20th of June 2022 in front of a jury composed of:

Mr. Mohamed Chaker ZAGHDOUDI	President of the jury
Mrs. Nedra BOHLI	Reviewer
Mr. Mehdi HANZOUTI	Supervisor at ACTIA ES
Mr. Jaouhar SAKLY	Supervisor at INSAT

University year: 2021 / 2022

REPUBLIC OF TUNISIA
Ministry of Higher Education and Scientific Research
University of Carthage
National Institute of Applied Sciences and Technology

Graduation Project

to obtain the

National Engineering Diploma
in Applied Sciences and Technology

Specialty: Instrumentation and Industrial Maintenance

Wireless Serial Wire Debug interface

Prepared by

Firas ISMAIL

Hosted by ACTIA ES

Supervisor at ACTIA ES Mr. Mehdi HANZOUTI	Supervisor at INSAT Mr. Jaouhar SAKLY
Date:	Date:

University year: 2021 / 2022

Dedication

It is my genuine gratefulness and warmest regard that I dedicate this dissertation to my beloved people who contributed in a way or another to my success.

To my grandparents' memory, Warda and Ammar, who would have been very happy and proud to see me here today, may you find peace.

To my teacher's memory, Zoubeir. He was an exceptional teacher who believed in me throughout highschool. I hope I will live up to your expectations, may you find peace.

To my parents Adel and Salwa, my siblings, Jacer, Samer and Yosr and my whole family who never stopped supporting me through all my life. I hope I will make you prouder.

To all of my friends, wherever you are in the world now, my life wouldn't have been the same without you, and I appreciate every moment I have spent with you. With a special shoutout to the memesquad.

Acknowledgments

I would like to express my thanks and gratitude to all the people who supported me in the realization of this end of studies project, in particular :

Mr. **Mehdi HANZOUTI**, my LLSW team manager and supervisor at ACTIA, for welcoming me into the team, for his technical guidance and support throughout the internship and for helping me understand many concepts in world of embedded systems.

Mr. **Jaouhar SAKLY**, my teacher and academic supervisor at INSAT, for his precious help, encouragement and valuable advice during the realization of this end of studies project.

Dear members of the jury for accepting to evaluate my work.

The whole LLSW team for their friendly welcome and professional cooperation.

My family and friends for their never-ending support during my whole academic journey.

The kind strangers on Stackoverflow for always providing answers to my questions.

My thanks finally go to everyone who contributed directly or indirectly to the development of this work.

Abstract

This work serves as a proof of concept of a wireless interface for the Serial Wire Debug protocol. This interface allows users to program and debug microcontrollers that are not accessible through wired interfaces.

This project is considered to be a first of its kind. It made use of some key properties of SWD in order to implement a workaround solution that enables us to transfer the debug protocol wirelessly.

This study will be followed by an establishment of a detailed technical structure of the solution and the implementation of all the intermediary steps.

Keywords : SWD, STM32, ESP32, UART, Wi-Fi, TCP, ESP-NOW.

Table of contents

List of figures

List of acronyms

- **SWD.** Serial Wire Debug.
- **SWCLK.** Serial Wire Debug Clock.
- **SWDIO.** Serial Wire Debug Input/Output.
- **MCU.** Micro-Controller Unit.
- **CPU.** Central Processing Unit.
- **UART.** Universal Asynchronous Receiver/Transmitter.
- **SPI.** Serial Peripheral Interface.
- **GPIO.** General Purpose Input/Output.
- **TCP.** Transmission Control Protocol.
- **UDP.** User Datagram Protocol.
- **DAP.** Debug Access Port.
- **RTOS.** Real-Time Operating System.
- **API.** Application Programming Interface.
- **GDB.** GNU Debugger.
- **IDE.** Integrated Development Environment.

Introduction

The microcontroller has played a fundamental role in the technological revolution that has shaped modern life. Microcontrollers are small, versatile, inexpensive devices that can be successfully implemented and programmed to achieve a particular purpose. But it can be truly frustrating when it doesn't function as expected. Most issues can be traced back to one of two things : a failure to understand a specific MCU feature, or a mistake in the code. It is in such situations that the ability to "look inside" the device would be very helpful. Today's MCUs typically offer integrated debugging interfaces that allow them to be programmed (flashed) and debugged.

Projects involving microcontrollers can be so complex, and they require of their developers to be able to constantly observe the workflow of the program and the state of its varibales. because of that, there exists tools that facilitate the communication with embedded devices. such tools are called debug probes, their role is to interface between a computer and a microcontroller and to serve as a wired bridge between the two different architectures.

Firmware can be updated and debugged using physical links if there are specific physical connectors on the device. But physical access is not always possible, either because of the distance between the PC and the firmware or because the firmware is in a moving object(e.g. a drone, an e-bike..).
Our project aims to study and develop a wireless debugging interface, a solution that can overcome the necessity of a wired connection.

With the increasing growth of the mobility industry and the expanding demand for microcontrollers inside vehicles, it became apparent that using physical connections to communicate with these devices can be problematic and inefficient. As a solution, we suggest a wireless debug interface that enables developers to program and debug mobile

devices without the need of physically connecting the computer to the target device.

This report summarizes and presents the different stages of the project's life cycle. It is organised ad follows :

In the first chapter, *Project context*, we will present the host company and give an overview of the project and the problematic behind it. The next chapter, *Technical definitions*, will start with the explaining basic debugging concepts as well as the software and hardware tools that enable us to program and debug microcontrollers. We will also explain some key networking and communication concepts. The third chapter, *Architecture and Implementation*, will describe the project's proposed solutions, their general architectures and their implementations phases.

1

Project context

Introduction

This first chapter aims to define the general context of the project, by introducing the host organization, its functional structure and its activity sectors. This chapter will also explore the needs driving the development of this project as well as the solution adopted.

1 Host company presentation

In this Section, we will present the host company and then give a description of its activity sector.

1.1 Presentation of the ACTIA group

The ACTIA Group is a medium-sized, family-owned company with its headquarters in head office is located in France. This family character of the Group, since its creation

in 1986, has played an important role in what ACTIA is today and what it will be tomorrow. It guarantees the Group's long-term future and its independence in a constantly renewed entrepreneurial dynamic. ACTIA was created in 1986 by Mr. CALMELS, Mr. CHABRERIE and Mr. PECH in Toulouse. One of the first products marketed by this company was an electronic diagnostic equipment, the first of its kind. This group is a merger with the company Electroniques Manufacturing and the company Sodie-lec, a company specialized in satellite telecommunications. Today, the ACTIA Group has 22 locations in 16 countries (figure 1.1) and this international dimension is an integral part of the group's identity in the same way as its pioneering spirit and industrial culture [1]. Figure1.1 shows the location of the ACTIA Group in various countries.

FIGURE 1.1 – Establishment of the ACTIA group around the world

1.2 ACTIA ES

ARDIA is an engineering company located in the Technopole Ghazala in Ariana. ARDIA represents the acronym of "ACTIA Groupe Recherche et Développement en Informatique Appliquée", was created in July 2005.
In 2019 the name of the company's name became ACTIA Engineering Services, it constitutes a very important subsidiary located in the North African territory.
ACTIA and its subsidiaries are specialized in automotive diagnostics, on-board electronics and telecommunications.
ACTIA ES has a strong experience in projects carried out in an international context. Since its creation, the company has always supported its customers in the development

and integration of embedded electronics and communicating software or associated software packages. ACTIA ES has developed a real know-how for :

- The development of embedded software or PCs.
- Mechanical and electronic studies.
- Test and validation of complex systems.
- Equipment services.

Figure 2.6 constitutes a photo of the ACTIA Engineering Services headquarters in Tunisia.

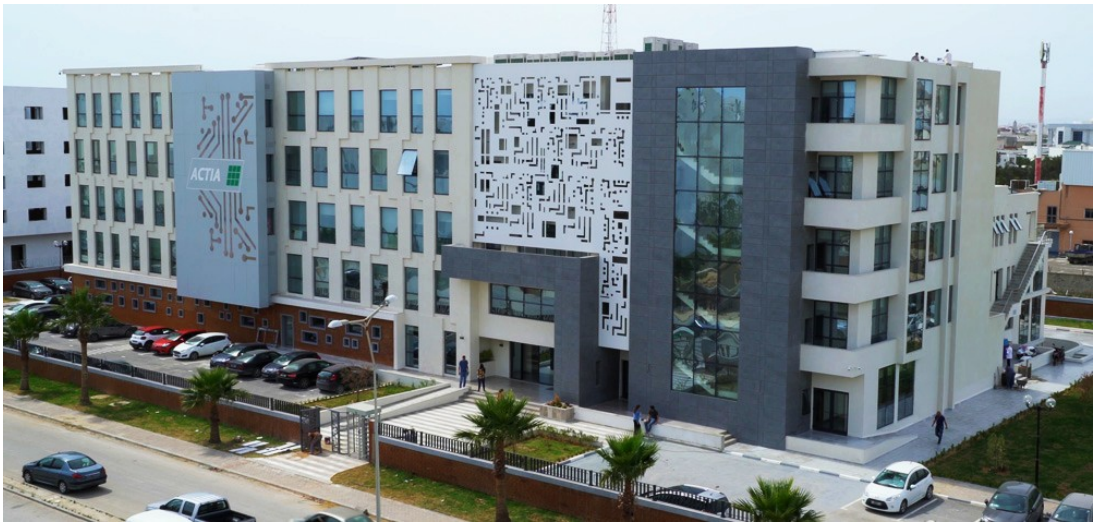

FIGURE 1.2 – A photo of the ACTIA ES Headquarters

ACTIA ES has set up an efficient and constantly-improving quality system, as well as a proven methodology and expertise in project management based on the best practices and from the best practices and standards of the software industry.
ACTIA ES is certified ISO 9001 version 2008 for all its design activities, development, test and validation, diagnostic tools, mechanical systems, production tools for production tools for automotive applications, industrial support and quality testing.

1.3 Activity sectors

1.3.1 Automotive and transportation

With 10 years of experience in the automotive sector, ACTIA ES is involved in projects with high added value. Working on complete and diversified projects (embedded software, application software, operating security, mechatronics development,. . .).

Its experience and know-how in the automotive and transportation sectors, developed during these ten years, have enabled it to extend its interventions to other fields of activity such as the aeronautics sector, which uses the same skills and know-how.

1.3.2 Information technology

With increased competition, creativity is becoming essential to survive in this industry. ACTIA ES helps its customers to develop sustainable and profitable applications by offering solutions adapted to new technologies.

1.3.3 Industry

ACTIA ES takes on projects for the design and integration of various electronic and mechanical products. Its teams work on diversified projects with value-added projects. Its manufacturing, service and functional test teams work together to to meet the needs of its partners in terms of production tools.
Downstream of its value chain, its laboratory team also performs quality testing of various industrial equipment.

1.3.4 Multimedia and telecommunication

ACTIA ES is involved in the development of short life cycle applications, connected objects and smartphones. As a software solution provider, ACTIA ES has the necessary skills to develop customized applications for its partners.

2 Project context and concepts

During this internship, I was tasked to create a wireless interface for programming and debugging microcontrollers through the SWD protocol.
This project is considered to be a first of its kind with unprecedented implementation ideas and techniques.

2.1 Preliminary key concepts

2.1.1 Microcontrollers

A microcontroller, or an MCU, comes in the form of a silicon chip. It is a combination of computer hardware and software designed for a specific function. an MCU may also function within a larger embedded system. The systems can be programmable or have a fixed functionality. Industrial machines, consumer electronics, agricultural and processing industry devices, automobiles, medical equipment, household appliances, airplanes, as well as mobile devices, are possible locations for an embedded system. Microcontrollers are low-cost, low-power-consuming, small computers that are embedded in other mechanical or electrical systems. Generally, they comprise a processor, power supply, memory and communication ports.
To achieve a particular functionality, the microcontroller needs to be programmed and debugged.

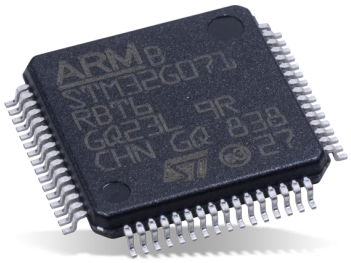

FIGURE 1.3 – An example of a microcontroller chip (STM32G071RB)

2.1.2 Programming an MCU

the MCU's core program is coded on a computer with a supported programming language. With the help of the proper toolchain, the program is then compiled into machine instructions that are coded in one binary file. This binary file, currently still on the computer, needs to be transferred to the microcontroller's internal flash memory where it can be executed by the MCU's microprocessor.

2.1.3 Debugging an MCU

After programming, developers often run into problems as testing can reveal unwanted behaviour. when an MCU doesn't work as expected, the issue can be traced back to a

mistake in the code. in big projects, those mistakes can be hidden between thousands of lines of code, which makes it hard for the developers to find. Luckily, microcontrollers can be debugged.
Debugging is widely used in all fields of software development and not just embedded systems. It is the process of detecting and removing of existing and potential errors (also called as 'bugs') in a software code that can cause it to behave unexpectedly or crash. To prevent incorrect operation of a software or a system, debugging is used to find and resolve bugs or defects. During debugging, we use a computer to view the execution state and data of an application as it is running inside the microcontroller. debugging allows users to halt the execution of the program, examine the values of variables and registers, step execution of the program line by line, and set breakpoints on lines or specific functions that, when hit, will halt execution of the program at that spot. In this view, seeing how the program is viewed by their computer, the user can discover how a program flows, identify incorrect code and data, and find semantic errors in the program. In order to program and debug a microcontroller, a connection needs to be established between the MCU and a computer. this connection is set up using a debugger.

2.1.4 Debuggers

Debuggers are a combination of hardware and software tools that enable the developer to transfer data to an MCU and monitor the execution of a program. the main hardware component of a debugger is the debug probe.

We can describe debug probes as a hardware intermediary between the host machine and the debug port of the target embedded system. The main task of any debug probe is to convert commands from the debugger into signals understandable to the target device. Usually, debug probes are connected to the host via USB and to an MCU via a specific protocols. one of the most popular debugging protocols is Serial Wire Debug (SWD), which is a serial bi-directional protocol developed for the purpose of transferring debug instructions and data to and from an MCU.

FIGURE 1.4 – An example of a debug probe (ST-LINK/V2)

2.2 The Problematic

As we have mentioned, programming and debugging a microcontroller requires a wired connection through a debug probe. Such physical connection is not always possibles, for many reasons.

2.2.1 Mobility

ACTIA's main activities revolve around the automotive sector. One of the main projects being currently worked on is the development of an electrical bicycle (e-bike.
during testing, it was impossible to debug the e-bike in real time as it was being driven by someone because debugging requires the device (the e-bike in this case) to be physically connected to a computer through a debug probe.
Generally, debugging mobile devices, such as drones, robots and micro-mobility systems, can waste a lot of time as developers need to come up with workarounds to trace the contents of the MCU.

2.2.2 Safety

When working with dangerous materials and systems, where software bugs can cause harmful effects, such as Lithium batteries or nuclear machinery, being physically close to the debug target can be dangerous to the user. A simple mistake in the code can cost human lives. therefore, such systems need to be programmed and debugged from a safe environment far away.

2.3 Solution

To overcome the inconveniences of programming and debugging microcontrollers through a debug probe, the proposed solution is to create a wireless debugging device that allows developers and testers to program and debug embedded systems without being physically connected to the device through a debug probe.

The wireless debug interface can be used, if there is movement between the development computer and the target system. This is particularly useful in micro-mobility systems and robotic applications. In other use cases, where the target system operates inside a hazardous environment, the wireless connection enables the developer to stay outside the dangerous area with his development system, even if there is a physical barrier between development and target system. The proposed solution makes use of the already existing debug protocol "Serial Wire Debug" while taking advantage of some its key properties. The main idea is to interface between the debug probe and the target MCU and to transfer the debug data without a physical wired connection.

Conclusion

This chapter presented the project's general scope. It also defined some important key concepts related to the project. In addition, we presented the motives for the project and we finished by proposing a solution that we're going to design and implement. The next chapter will present the requirements of the project as well as the architecture of the solution.

2

Technical definitions

Introduction

In this chapter we are going to introduce a multitude of concepts that will be frequently used in the upcoming chapters to explain the implementation process of our solution. The most important concept that we will be explaining is the Serial Wire Debug protocol since it is the backbone of our project.

1 Debugging

As we briefly explained in the first chapter, debugging is the process of detecting and removing of existing and potential errors within a microcontroller's software.

Debugging an MCU requires a combination of hardware and software tools that enable the programmer to transfer data to an MCU and monitor the execution of a program. These software and hardware tools are linked together to form what we call a "Debug chain".

This chain usually starts from an integrated development environment (IDE) that creates a GDB client that passes command to a GDB server created by a software that interfaces with a hardware debugger (usually openOCD). giver the correct debugger configuration, openOCD interfaces with the connected debug probe through a driver and a USB connection. The debug probe then converts the commands received to a debugging protocol, with the two most popular protocols being SWD and JTAG. These protocols are understood and interpreted by microcontrollers that support them by special debug ports.

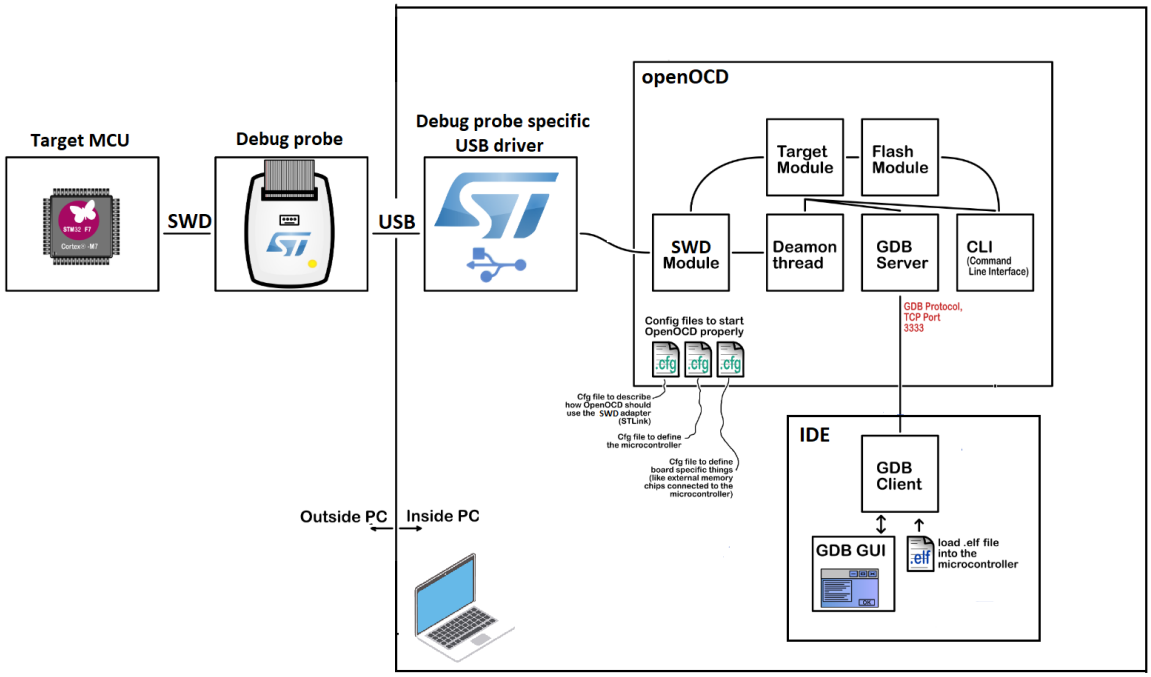

FIGURE 2.1 – General view of the debug chain

In the upcoming subsections, we're going to dive into each component of the debug chain with the main focus being on the SWD protocol.

1.1 Integrated development environment (IDE)

An integrated development environment (IDE) is a software application that provides comprehensive facilities to programmers for software development. An IDE normally consists of at least a source code editor, build automation tools and a debugger. IDEs present a single program in which all development is done. Using tools such as GCC,Make and GDB, this program typically provides many features to build the program, flash it into the microcontroller and then debug it

1.2 GNU Project Debugger (GDB)

The purpose of a debugger such as GDB [2] is to allow you to see what is going on "inside" another program while it executes–or what another program was doing at the moment it crashed. GDB can be used to debug programs written multiple programming languages such as C , C++ and Java. GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act :

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

GDB offers a remote debugging option, which is a very important feature when it comes to debugging embedded systems. Remote debugging is the process of debugging a program running on a different system (called target) from a different system (called host). In our case, OpenOCD creates a local GDB server on port 3333. A GDB client can establish a connection to that server on OpenOCD via localhost
Remote debugging is often useful in case of embedded applications where the resources are limited.

1.3 The Open On-Chip Debugger (OpenOCD)

OpenOCD (Open On-Chip Debugger) [3] is open-source software that interfaces with a hardware debug adapter. OpenOCD complies with the remote gdbserver protocol and, as such, can be used to debug remote targets. OpenOCD provides debugging and in-system programming for embedded target devices. Before running OpenOCD on the host computer, two configurations files of the debug probe and the target MCU need to be provided.

1.4 Debug probe

An additional hardware device is required to allow the host PC you are developing on to connect to the target microcontroller. These devices have several names such as debugger, debug probe or programmer. Sometimes they are referred to by a specific brand

or protocol they speak, e.g. ST-Link, JLink. These devices are typically distinct pieces of hardware.
During the phases of our project development, we used an ST-Link as our SWD debug probe.

1.4.1 ST-Link

The ST-LINK is an in-circuit debugger and programmer for the STM8 and STM32 microcontroller families.
ST-LINK is a USB device and has to be connected to a PC host. It can be either embedded on ST boards or provided as standalone dongle.
ST-LINK can support different debug protocols depending on ST-LINK hardware version and on its embedded firmware version : SWIM,JTAG and SWD. with the SWD being the main focus of our project.

1.5 Serial Wire Debug (SWD)

1.5.1 Overview

Serial Wire Debug (SWD) is a debug protocol developed by ARM to replace the 5-pin JTAG protocol by providing a simpler interface that uses a single bi-directional data connection. [4]
Serial Wire Debug (SWD) is a two-wire protocol for accessing the ARM debug interface. It is part of the ARM Debug Interface Specification v5. The physical layer of SWD consists of two lines :

- **SWDIO** (Serial Wire Debug Input/Output) : a bidirectional data line
- **SWCLK** (Serial Wire Clock) : a clock driven by the host

Connecting to these pins allow an external device (such as a debug probe) to communicate directly with the Serial Wire Debug Port (SW-DP) inside an ARM microprocessor architecture. The SW-DP in turn can access one or several Access Ports (APs) that give access to the rest of the system.

1.5.2 The SWD protocol

In SWD terminology, the host refers to the system controlling the debugger, i.e. the debug probe. The target is the system which is under the debug operation.
SWCLK is a clock signal which is always driven by the host. When the host controls the SWDIO line, bits are banged onto this wire on the falling edge of the clock. When the target controls the line, bits are written on the rising edge of the clock.
Both sides will drive the SWDIO line to send data. A high value on SWDIO indicates a logical '1', a low value is a logical '0'. Figure 2.2 presents an examples SWD Data being sent over the SWDIO line on the falling edge of SWCLK. [5]

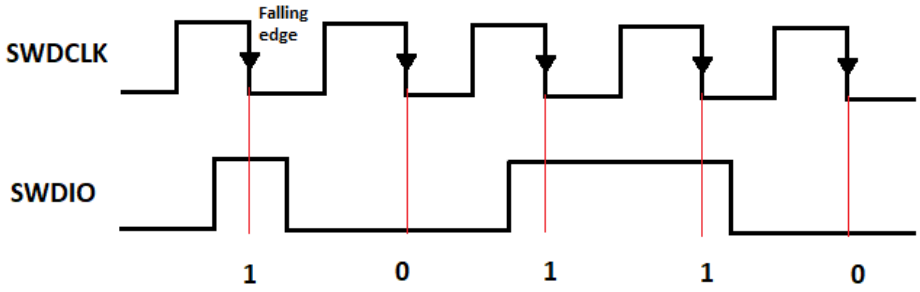

FIGURE 2.2 – A sample of SWD data sampling and transmission

The protocol specifies when each side will drive the SWDIO line.
Three different phases are specified : Each transaction begins with the host sending a request. The target answers with an acknowledge which is followed by a data phase.

As mentioned, transfers in SWD are 'packet' based, and successful requests will consist of three components :

- **8-bits Request Header**, Sent from the host-to-target
- **3-bits Acknowledgement (ACK)**, Sent from the target to the host.
- **33-bits Payload**, with 32-bits for data, and 1-bit for parity. This may be host-to-target in the case of a write. And may be target-to-host in the case of a read.

These phases are represented in Figure.2.3

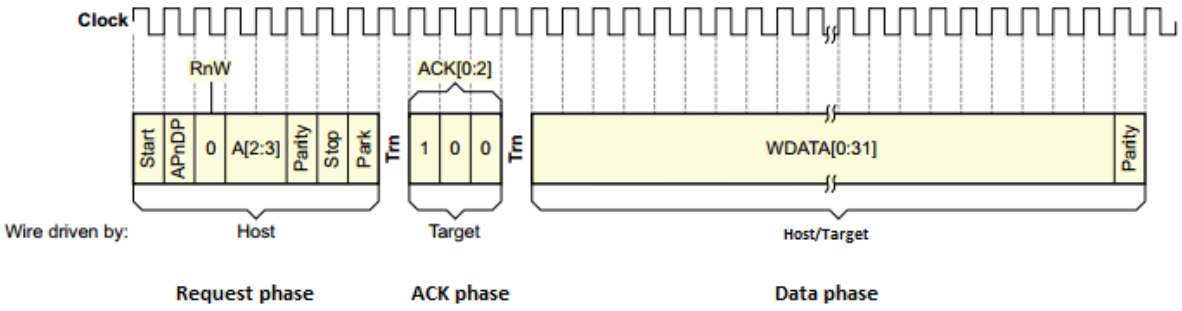

FIGURE 2.3 – The 3 phases of a successful SWD transmission

Who controls the line during the data phase depends on the type of request issued by the host. If the host issued a write request, the host will drive the line. On a read request the target will drive the line to transmit data from the target to the host. In all phases data is transmitted LSB (least significant bit) first. The target will both sample and put data on the line on a rising clock edge.

1.5.3 SWD request phase

The request phase consists of 8 bits. The meaning of each bit in the request is illustrated in Figure 2.3.

- **Start** - The start bit, 0th bit, is always 1.
- **APnDP** - The next bit specifies whether the transaction is a DP (Debug Port) or AP (Access Port) transaction. If this bit is zero, the transaction is a DP access.
- **RnW** - Bit 2 is the read/write bit. If this bit is 1 the transaction is a read access (from target to host).
- **A[2 :3]** - Bit 3 and 4 are address bits A2 and A3. These bits specifies which out of four registers are selected for the transaction. Register selection is described in the next subsection .
- **Parity** - Bit 5 is a parity bit. The parity bit is used by the target to verify the integrity of the request. This bit should be 1 if bits 1-4 contains an odd number of 1's. If the number of 1's are even, the parity bit should be zero
- **Stop** - Bit 6 is the stop bit. This bit is always zero.
- **Park** - Bit 7 is the park bit. This bit is always one.

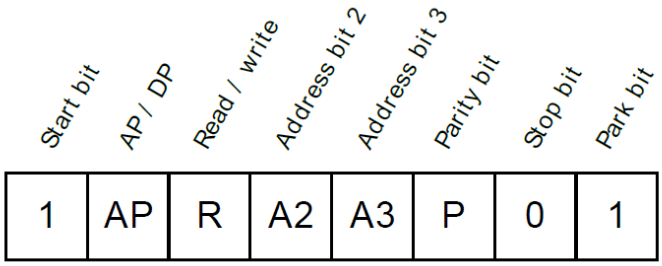

FIGURE 2.4 – A SWD request example

1.5.4 Acknowledgement phase

The Acknowledgement phase, or simply ACK, is a the taret's acknowledgement to the request sent by the host. the ACK consists of 3 bits and can take 3 values corresponding to 3 possible responses from the target's DAP : OK, WAIT and FAULT.

- An **OK** response has the value 1 (001 in binary). Since values are put on the line LSB first an OK response looks like a 1 followed by two 0's on the line (100). Once the host has received an OK response the data phase can begin.
- a **WAIT** response has the value of 2 (010 in binary). an ACK WAIT sent by the target indicates that the debug port is currently busy or incapable of responding to the request at that moment.

  In this case, the host will abort the current request and will send the request header again.

  The ACK WAIT property of the SWD protocol inspired the key idea behind our project. In the next chapter we will explain in depth how we manipulated this property to implement a wireless bridge for the SWD protocol. [6]
- a **FAULT** respose has the value of 4 (100 in binary) and is represented LSB first (001) on the SWDIO line. an ACK FAULT respose indicates that something went wrong, an error occured and one of the sticky bits in CTRL/STAT is set. It can be a transmission error or a wrong parity bit sent by the host.

1.5.5 Payload phase

Payload phase, or data phase, follows both Write and Read requests. In a write request, the host uses the payload to transmits data to the target's debug port. This mostly occurs when we flash the MCU over SWD or send debug instructions to start,halt or ask to read from a specific memory address. In a read request, the playload will carry data from the

target back to the host, this data can be identification data that can be used to identify the target or memory values that the user has asked for to read. The data phase consists of 32 data bits (one word) followed by 1 parity bit. The parity bit is calculated based on all the 32 data bits. If the number of 1's in the data word is odd, the parity bit should be 1. The host must continue to clock the interface for at least 8 cycles after the data phase to make sure the transaction is clocked through the SW-DP. This can be done either by :

- Starting a new transaction
- Inserting idle clock cycles

During idle clock cycles the SWDIO is driven low by the host.

1.5.6 Turnaround periods

Every time the SWDIO changes data direction, a one-cycle turnaround period is inserted which both sides should ignore. This means there is always a turnaround period between the request and acknowledge. On a write request, there is a turnaround period between acknowledge and the data phase. On a read request there is a a turnaround after the data phase. in the case of an ack WAIT, the host will attempt to re-send the same request, so there's also a turnaround after the WAIT acknowledgement and the next request.

1.5.7 Initialization

Before using the SW-DP an initialization sequence must be performed to establish communication and bring the SW-DP to a known state.

1. Perform a line resest
2. Send the JTAG-to-SWD switching sequence
3. Perform a line reset
4. Read the IDCODE register

A line reset is performed by clocking at least 50 cycles with the SWDIO line kept HIGH(1) by the host.
The reason for the JTAG-to-SWD sequence is that the Debug Port implementation is actually a SWJ-DP. SWJ-DP is a wrapper around both SW-DP and JTAG-DP, the JTAG counterpart to SWD.
The JTAG-to-SWD sequence is 0b0111100111100111 in binary or 0xE79E in hexadecimal

transmitted LSB first.
After transmitting the JTAG-to-SWD sequence, the host will perfom another line reset. This ensures that if SWJ-DP was already in SWD mode, before sending the select sequence, the SWD goes to line reset.
The host must read IDCODE register after line request sequence. This requirement gives confirmation that correct packet frame alignment has been achieved.

1.6 Debug Access Port (DAP)

The Debug Access Port(DAP) is split into two main control units. the Debug Port (DP) and the Access Port (AP), and the physical connection to the debugger is part of the DP.
The DAP supports two types of access, Debug Port (DP) accesses and Access Port (AP) accesses. External device communicate directly with Serial Wire Debug Port(SW-DP) over SWDIO/SCLK pins. And SW-DP in return can access one or several Access Ports(APs) that give access to the rest of the system. The MEM-AP is important AP which provide a way to access all memory and peripheral registers residing on the internal AHB/APB buses. [7]
The DAP's architecture is represented in Figure 2.5

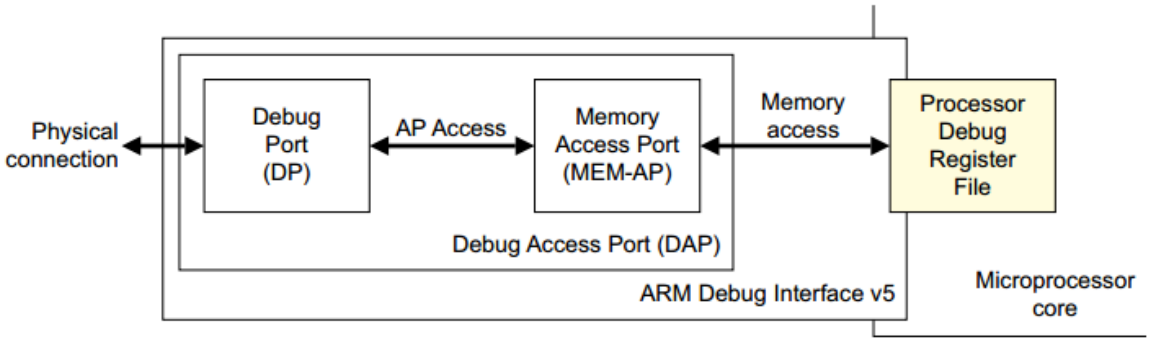

FIGURE 2.5 – DAP's internal architecture

One of the four registers within the DP is the AP Select Register, SELECT. This register specifies a particular Access Port, and a bank of four 32-bit words within the register map of that AP. It enables up to 256 Access Ports to be implemented, and gives access to any one of 16 four-word banks of registers on the selected AP.

An AP provides a connection from the DP to a subsystem on a system on chip. Many subsystems consist of multiple debug components that are arranged in a memory map.

An AP provides the connection to these memory mapped components. If more than one subsystem is accessed, more than one AP is used. The most important AP is the AHB-AP which is a bus master on the internal AHB1 bus.
In other words the AHB-AP can access the internal memory map of the core. Since the internal flash, SRAM, debug components and peripherals all are memory mapped, this AP can control the entire device including programming it.
The next figure displays the SW-DP connection to the APs.

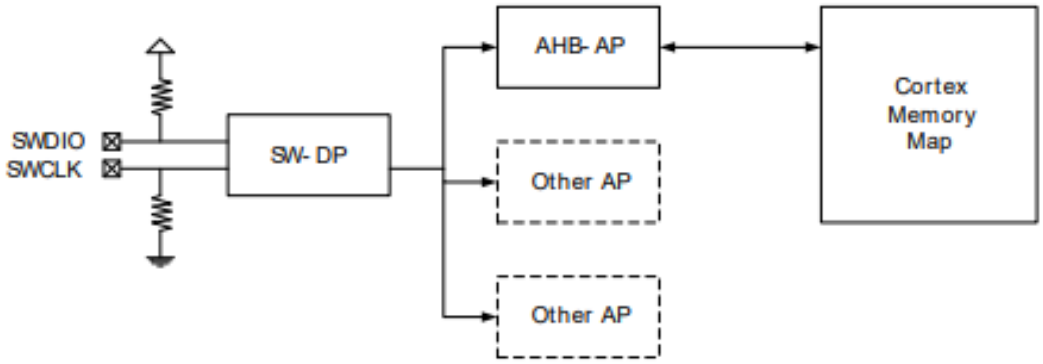

FIGURE 2.6 – SW-AP connection witht he APs

2 Networking

2.1 Overview

Networking refers to interconnected computing devices that can exchange data and share resources with each other. These networked devices use a system of rules, called communications protocols, to transmit information over physical or wireless technologies. Nodes and links are the basic building blocks in computer networking. A network node may be some data communication equipment (DCE) such as a modem, hub or, switch, or data terminal equipment (DTE) such as two or more computers and printers. A link refers to the transmission media connecting two nodes. Links may be physical, like cable wires or optical fibers, or free space used by wireless networks.
Networks use models to describe the interconnection system that links devices together. The most commonly used model is the OSI model.

2.2 OSI Model

The OSI Model (Open Systems Interconnection Model) is a conceptual framework used to describe the functions of a networking system.
Created at a time when network computing was in its infancy, the OSI was published in 1984 by the International Organization for Standardization (ISO). Though it does not always map directly to specific systems, the OSI Model is still used today as a means to describe Network Architecture. [8]
The OSI model characterizes computing functions into a universal set of rules and requirements in order to support interoperability between different products and software. In the OSI reference model, the communications between a computing system are split into 7 different abstraction layers : Physical, Data Link, Network, Transport, Session, Presentation, and Application. Figure 2.7 illustrates the OSI model.

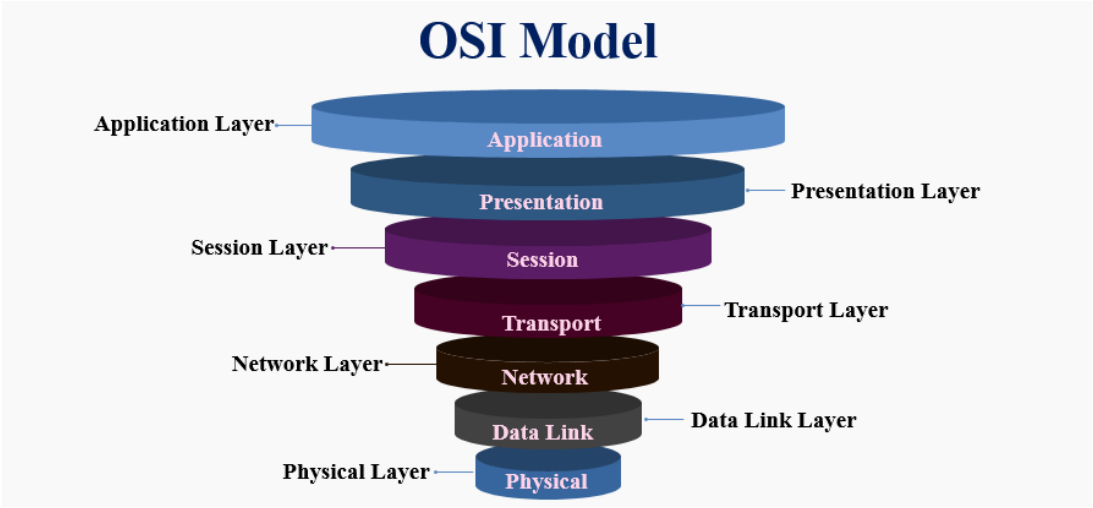

FIGURE 2.7 – A representation of the 7 OSI model layers

2.3 OSI Layers

The Open Systems Interconnection (OSI) model describes seven layers that computer systems use to communicate over a network. We'll briefly describe the 7 OSI layers "top down" from the application layer that directly serves the end user, down to the physical layer.

7. Application Layer The application layer is used by end-user software such as web browsers and email clients. It provides protocols that allow software to send and receive information and present meaningful data to users. A few examples of application layer

protocols are the Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Post Office Protocol (POP), Simple Mail Transfer Protocol (SMTP), and Domain Name System (DNS).

6. Presentation Layer - The presentation layer prepares data for the application layer. It defines how two devices should encode, encrypt, and compress data so it is received correctly on the other end. The presentation layer takes any data transmitted by the application layer and prepares it for transmission over the session layer.

5. Session Layer - The session layer creates communication channels, called sessions, between devices. It is responsible for opening sessions, ensuring they remain open and functional while data is being transferred, and closing them when communication ends.

4. Transport Layer - The transport layer takes data transferred in the session layer and breaks it into "segments" on the transmitting end. It is responsible for reassembling the segments on the receiving end, turning it back into data that can be used by the session layer. The transport layer carries out flow control, sending data at a rate that matches the connection speed of the receiving device, and error control, checking if data was received incorrectly and if not, requesting it again.
This layer is important for our project as it contains the Transmission Control Protocol (TCP).

3. Network Layer - The network layer has two main functions. One is breaking up segments into network packets, and reassembling the packets on the receiving end. The other is routing packets by discovering the best path across a physical network. The network layer uses network addresses (typically Internet Protocol addresses (IP)) to route packets to a destination node.

2. Data Link Layer The data link layer establishes and terminates a connection between two physically-connected nodes on a network. It breaks up packets into frames and sends them from source to destination. This layer is composed of two parts—Logical Link Control (LLC), which identifies network protocols, performs error checking and synchronizes frames, and Media Access Control (MAC) which uses MAC addresses to connect devices and define permissions to transmit and receive data.

1. Physical Layer - The physical layer is responsible for the physical cable or wireless connection between network nodes. It defines the connector, the electrical cable or wireless technology connecting the devices, and is responsible for transmission of the raw data, which is simply a series of 0s and 1s, while taking care of bit rate control.

Figure 2.8 illustrates a peer-to-peer communication between two devices as seen by the OSI model.

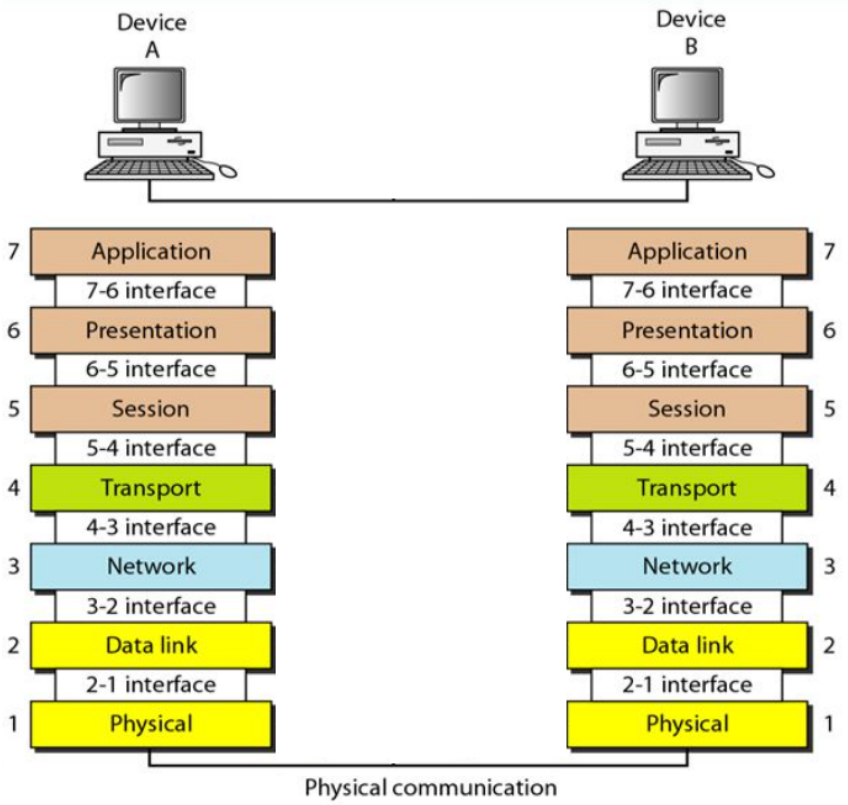

FIGURE 2.8 – An OSI model diagram of a communication between a device A and a device B

2.4 Wireless data transmission

Wireless transmission is a form of unguided media. Wireless communication involves no physical link established between two or more devices, communicating wirelessly. Wireless signals are spread over in the air and are received and interpreted by appropriate antennas.

When an antenna is attached to electrical circuit of a computer or wireless device, it converts the digital data into wireless signals and spread all over within its frequency range. The receptor on the other end receives these signals and converts them back to digital data. [9]

A little part of the electromagnetic spectrum can be used for wireless transmission as represented in figure 2.9.

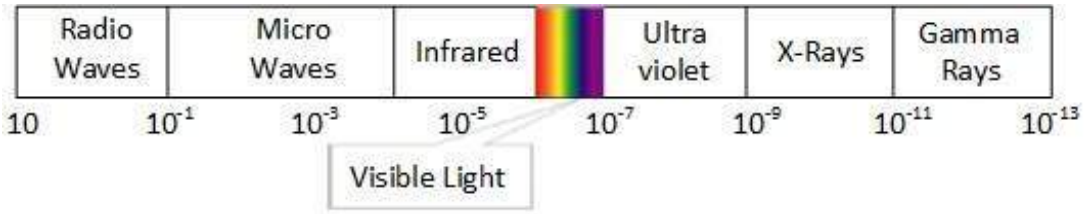

FIGURE 2.9 – The electromagnetic spectrum

Nowadays, wireless technological systems have been one of the vital parts of various types of wireless communication devices. This permits users and device networks to communicate even from any remote operated areas. There are various technologies used for wireless communication and data transmission like Wi-Fi, LoRaWAN, Infrared, Bleutooth, Zigbee and even satellite-based communication technologies like GPS.
Developers of wireless devices pick the most optimal wireless technology for their projects based on multiple properties and factors like communication distance, transmission delay and implementation costs.
For our project, we decided to go with the Wi-Fi family of wireless networks to implement the host-to-target SWD communication for its proven effectiveness.

2.5 IEEE 802.11

IEEE 802.11 is part of the IEEE 802 set of local area network (LAN) technical standards, and specifies the set of media access control (MAC) and physical layer (PHY) protocols for implementing wireless local area network (WLAN) computer communication. The standard and amendments provide the basis for wireless network products using the Wi-Fi brand and are the world's most widely used wireless computer networking standards. [10]

2.6 Wi-Fi

Wi-Fi, or WiFi, is a family of wireless network protocols, based on the IEEE 802.11 family of standards, which are commonly used for local area networking of devices and Internet access, allowing nearby digital devices to exchange data by radio waves. Wi-Fi

operates at the first two layers of the OSI model, in other words, the physical layer and the data link layer.Wi-Fi forwards packets received by the network layer regardless of their content. Wi-Fi is the most widely used computer network in the world, used globally in home and office networks to link desktop and laptop computers, smartphones, smart TVs, printers, and other smart devices in world of IoT.

2.6.1 Operational principles

WiFi uses radio frequencies, or radio waves, to communicate between devices ; the frequencies are measured in gigahertz (GHz). WiFi uses frequency bands 2.4GHz or 5GHz for signals.

Every WiFi frequency band has multiple channels within it to so our devices can send and receive data. The purpose of the WiFi channels is to decrease interference and overlap between your WiFi device and other WiFi devices. Interference and overlaps can cause the communication speed to slow down.

Wi-Fi stations communicate by sending each other data packets : blocks of data individually sent and delivered over radio. As with all radio, this is done by the modulating and demodulation of carrier waves.

Figure 2.10 represents the format 802.11 MAC frame .

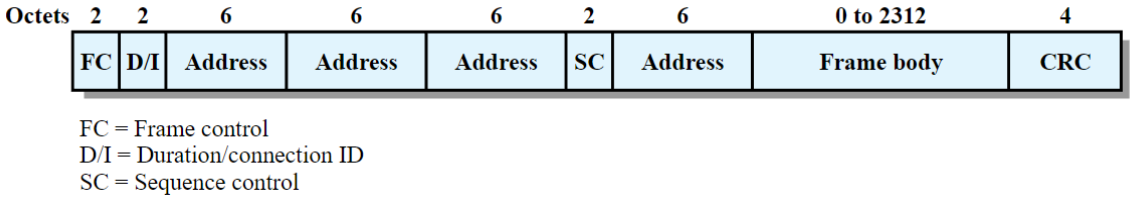

FIGURE 2.10 – IEEE 802.11 MAC frame format

2.7 Transmission Control Protocol (TCP)

TCP stands for Transmission Control Protocol, a communications standard that enables application programs and computing devices to exchange messages over a network.

TCP lives in the Transport layer of the OSI model. It is designed to send packets across the network and ensure the successful delivery of data and messages.

TCP organizes data so that it can be transmitted between a server and a client. It guarantees the integrity of the data being communicated over a network.

Before it transmits data, TCP establishes a connection between a source and its destination. The protocol ensures that the connection remains live until communication begins. It then breaks large amounts of data into smaller packets, while ensuring that data integrity is in place throughout the process.

To guarantee successful segment transmittion, TCP uses a Three-Way handhhake, which is a process used to make a connection between the server and the client. It is a three-step process that requires both the client and server to exchange synchronization and acknowledgment packets before the real data communication process starts.

Three-way handshake process is designed in such a way that both ends help you to initiate, negotiate, and separate TCP socket connections at the same time. It allows you to transfer multiple TCP socket connections in both directions at the same time.

Figure 2.11 illustrates the 3-way-handshake as well as the communication model follwed by TCP.

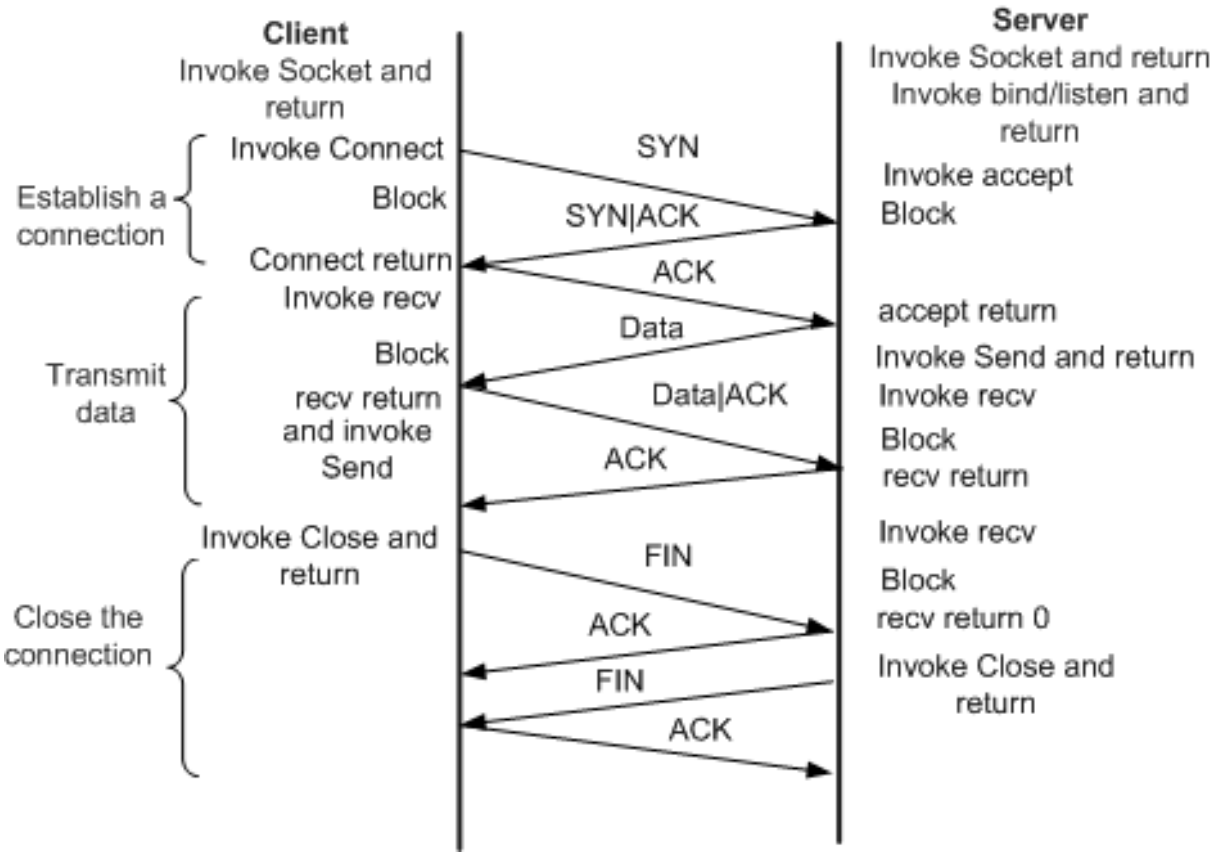

FIGURE 2.11 – TCP connection and communication model.

3 Communication Protocols

3.1 Overview

Communication between electronic components is similar to that between humans. Both the components need to speak in the same language in order to understand each other. For electronic components and microcontrollers these languages are known as communication protocols. Communication protocols are rules required to exchange information between computers and instruments. Serial Peripheral Interface (SPI), Inter- Integrated Circuit (I2C), Universal Asynchronous Receiver / Transmitter (UART), Controller Area Network (CAN) and Universal Serial Bus (USB) are ideal communication protocols which people use for communication between microcontrollers.
During the implementation phases of our project, two communication protocols are used : UART and SPI.

3.2 Universal Asynchronous Receiver / Transmitter (UART)

UART (Universal asynchronous receiver transmitter), as the name indicates, is an asynchronous communication protocol which does not use any clock source for synchronizing the data, UART is a physical circuit in a microcontroller or an integrated circuit that is used to implement serial communication between devices in an embedded system. Essentially, a UART's main purpose is to transmit and receive serial data.
In UART communication, two UARTs communicate directly with each other ; the UART on the sender device, or the transmitting UART, receives parallel data from the CPU (microprocessor or microcontroller) and converts it to serial data. This serial data is transmitted to the UART on the receiver device, or the receiving UART. The receiving UART converts the received serial data back to parallel data and sends it to the CPU. In order for UART to convert serial-to-parallel and parallel-to-serial data, shift registers on the transmitting and receiving UART are used.

In UART communication, only two wires are required for communication : data flows from the Tx pin of the transmitting UART (Transmitter Tx) to the Rx pin of the receiving UART (Receiver Rx) , as illustrated in Figure 2.12

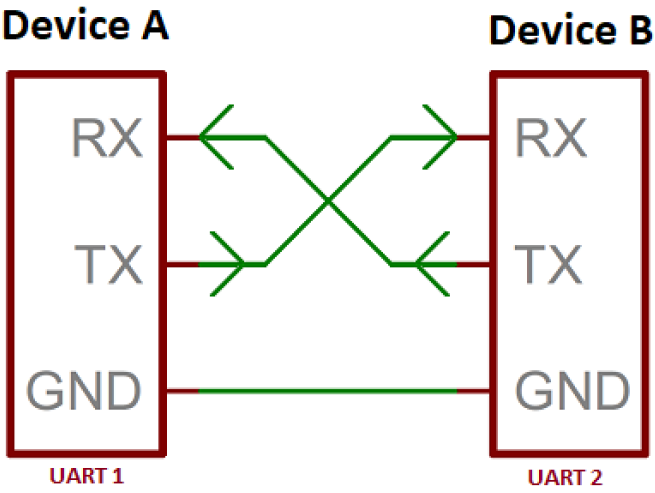

FIGURE 2.12 – UART Tx/Rx wiring

3.3 Serial Peripheral Interface (SPI)

SPI, or Serial Peripheral Interface, is a serial communication protocol often used in embedded systems for high-speed data exchanges between devices on the bus.
Unlike UART, It operates using a master-slave paradigm that includes at least four signals : a clock (SCLK), a master output/slave input (MOSI), a master input/slave output (MISO), and a slave select (SS) signal,As can be see in Figure x.x

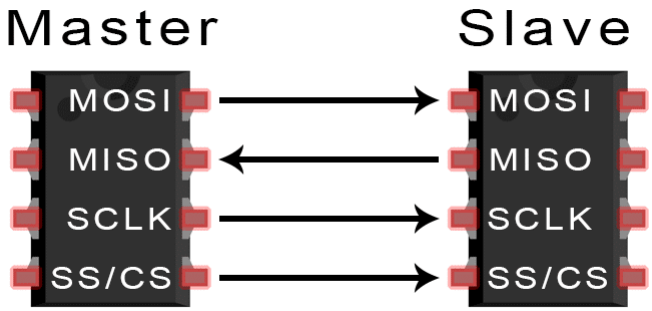

FIGURE 2.13 – SPI Wiring

The SCLK, MOSI, and MISO signals are shared by all devices on the bus. The SCLK signal is generated by the master device for synchronization, while the MOSI and MISO lines are used for data exchange. Additionally, each slave device added to the bus has its own SS line. The master pulls low on a slave's SS line to select a device for communication. An SPI communication can be seen on Figure 2.14.

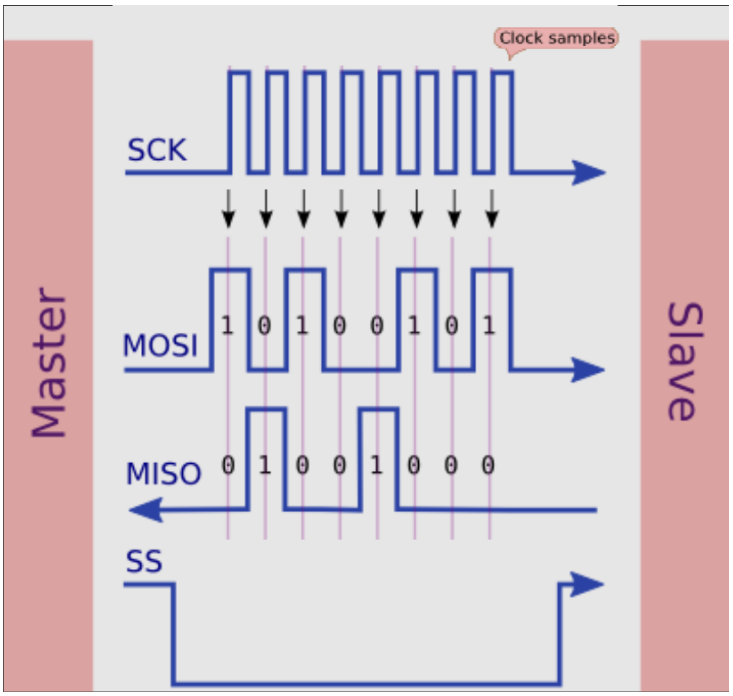

FIGURE 2.14 – Data transportation on the SPI MISO/MOSI lines

Data on the SPI MOSI and MISO lines can be sample on the rising,falling edges of the SCK signal, similar to the way SWDIO data is sampled on the edges of the SWCLK clock, this will inspire implementation ideas that will be discussed further in the next chapter.

3

Project Architecture and implementation

Introduction

In this chapter we will explain in detail the architecture of our project and dive into the implementation of every step of the wireless debugging chain which will be explained in the next section.

1 Overview

The general idea of our project is to create a wireless interface to the Serial Wire Debug protocol. The final goal is to have two Wi-Fi-enabled microcontrollers communicating SWD data back and forth between the debug probe and the target device.
The debug probe will be connected to a host computer via USB which is then connected to an MCU we'll call master which will decode the SWD frames and send them wirelessly to another MCU we'll call slave which will interact with our target MCU. as illustrated in figure 3.1.

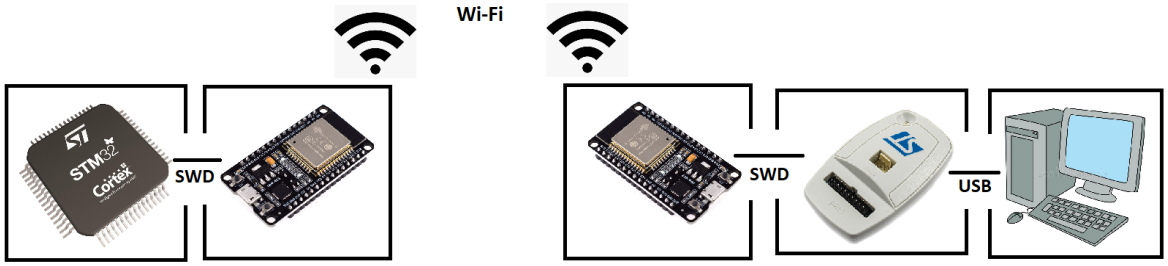

FIGURE 3.1 – The imagined final wireless SWD interface

The project is mainly technical and a large portion of it constitutes of programs written in the C programming language.
The implementation of this project went through multiple interconnected steps using a large set of different tools before reaching its final architecture. We abstracted these steps into 4 main phases.

2 Tools

During the different phases of the project's life cycle, we used a combination of software and hardware tools that played a critical role in facilitating the development of the project.

2.1 STM32CubeIDE

STM32CubeIDE is an all-in-one multi-OS Integrated Development Environnement (IDE) , which is part of the STM32Cube software ecosystem. It is based on the Eclipse/CDT framework and GCC toolchain for the development, and GDB for the debugging. It allows the integration of the hundreds of existing plugins that complete the features of the Eclipse IDE.
STM32CubeIDE integrates STM32 configuration and project creation functionalities from STM32CubeMX to offer all-in-one tool experience and save installation and development time. Figure 3.2 shows a screenshot of the CubeIDE as seen by the user

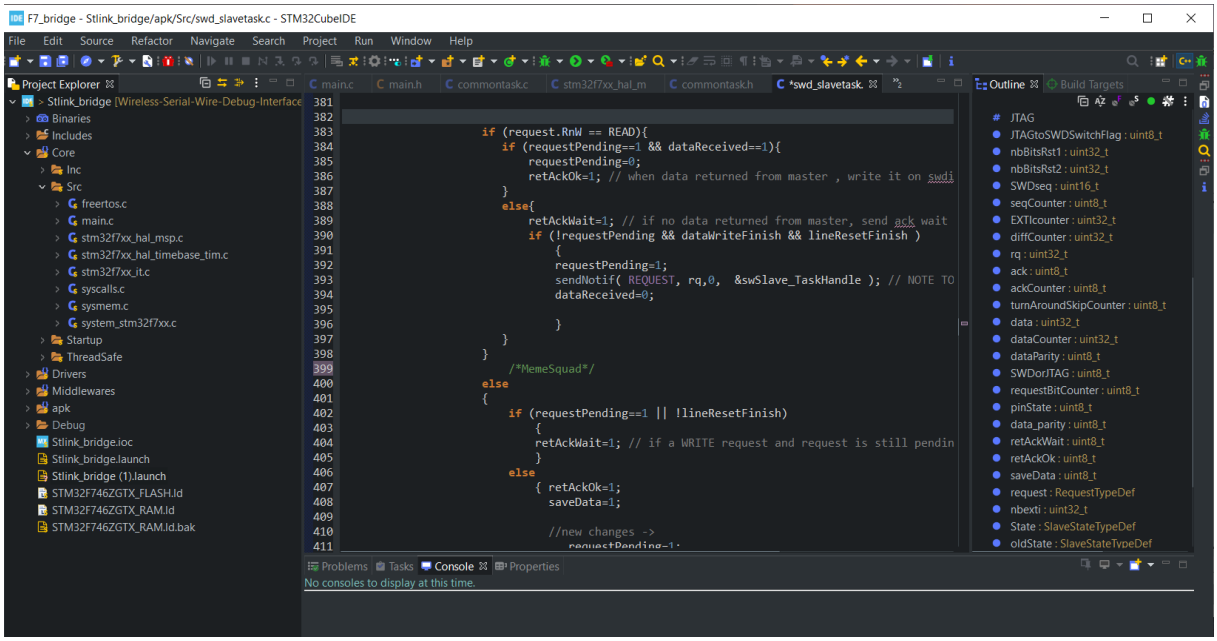

FIGURE 3.2 – A screenshot of STMCubeIDE.

2.2 STM32F7

STMicroelectronics STM32F7 is a 32-Bit MCU based on the high-performance Arm Cortex®-M7 32-bit RISC core. The Cortex-M7 core operates at up to 216MHz frequency Figure 3.3 shows the development board nucleo-F746z containing the STM32F7 MCU chip. A development board is used because of the external circuitry it contains that facilitate connection to the MCU.

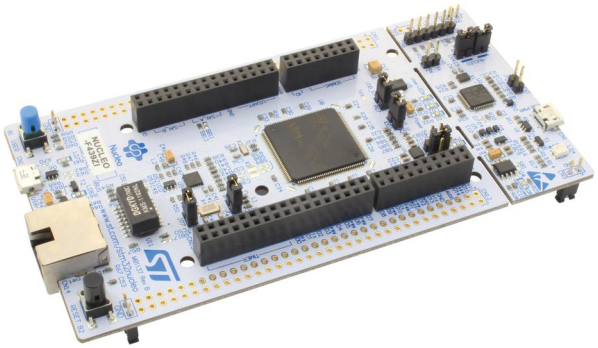

FIGURE 3.3 – A nucleo-F746z development board

The STM32F7 was chosen to be connected to the debug probe and to run the state machine for its high-performance and relatively fast operating clock frequency.
Such performing MCU is needed to successfully react in a short time to interrupts generated by the the input pin of the SWCLK signal.

2.3 Real Time Operating System (RTOS)

A real-time operating system (RTOS) is an operating system (OS) for real-time applications that processes data and events that have critically defined time constraints. A RTOS is distinct from a time sharing operating system, such as Unix, which manages the sharing of system resources with a scheduler, data buffers, or fixed task prioritization in a multitasking or multiprogramming environment. Processing time requirements need to be fully understood and bound rather than just kept as a minimum. All processing must occur within the defined constraints. Real-time operating systems are event-driven and preemptive, meaning the OS is capable of monitoring the relevant priority of competing tasks, and make changes to the task priority. Event-driven systems switch between tasks based on their priorities, while time-sharing systems switch the task based on clock interrupts.

2.3.1 FreeRTOS

FreeRTOS is an open source, real-time operating system for microcontrollers that makes small, low power edge devices easy to program, deploy, secure, connect, and manage. Maintained by Amazon and distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of software libraries suitable for use across industry sectors and applications.

FIGURE 3.4 – The logo of FreeTROS

A real time application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the RTOS scheduler itself. Only one task within the application can be executing at any point in time and the real time RTOS scheduler is responsible for deciding which task this should be. The RTOS scheduler may therefore repeatedly start and stop each task (swap each task in and out) as the application executes.

As a task has no knowledge of the RTOS scheduler activity it is the responsibility of the real time RTOS scheduler to ensure that the processor context (register values, stack contents, etc) when a task is swapped in is exactly that as when the same task was swapped out.

FreeRTOS helps us run concurring tasks simultaneously, such as a networking task and a state machine task. It maintains the safety and integrity of data being shared between different tasks and interrupts within the same MCU.

2.4 Logic analyzer

Logic analyzers are test instruments that are widely used for testing logic circuits and communication protocols. They meet the need for users who need to be able to investigate and understand the operation of these logic signals.

Oscilloscopes can perform many of the functions of a logic analyser but the the logic analyzer is able to display relative timing of a large number of signals. Essentially a logic analyser enables traces of logic signals to be seen in such a way that the operation of several lines in a digital circuit can be monitored and investigated.
We used the Saleae logic analyzer shown in Figure 3.5.

FIGURE 3.5 – Saleae logic analyzer

The logic analyzer played an enormous role in facilitating the debugging and the understanding of the comportment of our project during the different phases of our project.

2.5 ESP32

ESP32 is a series of low-cost, low-power system on a chip microcontrollers with integrated Wi-Fi developed by Espressif.
ESP32 can perform as a complete standalone system or as a slave device to a host MCU, reducing communication stack overhead on the main application processor. ESP32 can interface with other systems to provide Wi-Fi and Bluetooth functionality through its SPI, I2C and UART interfaces.

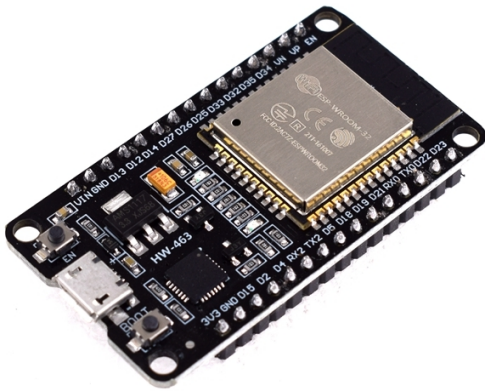

FIGURE 3.6 – An ESP32 development board

2.6 ESP-IDF

ESP-IDF is the official development framework for the ESP32.
ESP-IDF offers a development environment for ESP microcontrollers as well as Application Programming Interfaces (APIs) that implement and abstract many functionalities of the chip.

3 Implementation phases

3.1 Phase 1 : SWD State machine

In this part, we started by familiarizing ourselves with the SWD protocol, by reading the available SWD documentation provided by ARM.
Then, we linked the SWCLK and SWDIO output pins of the ST-Link debug probe to our logic analyzer in order to sniff the packets being transmitted on the SWD lines. The

protocol is visualized and studied with the logic software on the computer

Figure 3.7 clearly shows the 3 distinctive phases of a SWD frame : the request, the acknowledgment and then the payload.

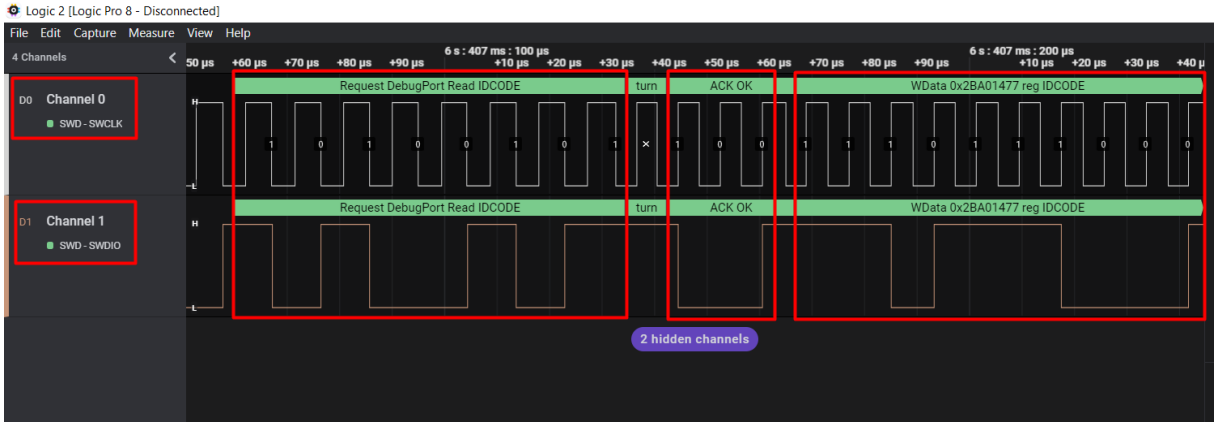

FIGURE 3.7 – A screenshot of the SWD protocol captured by the logic analyzer

We then proceeded to make a connection between an ST-Link debug probe and a target MCU (STM32F4). We also linked the SWD lines to 2 of our STM32F7 pins in order to intercept the SWD signals.
After that's done, we started writing a program for the STM32F7 microcontroller that takes the debug probe's SWD as input and proceeds to decode it.
Figure 3.8 shows the current setup for this phase.

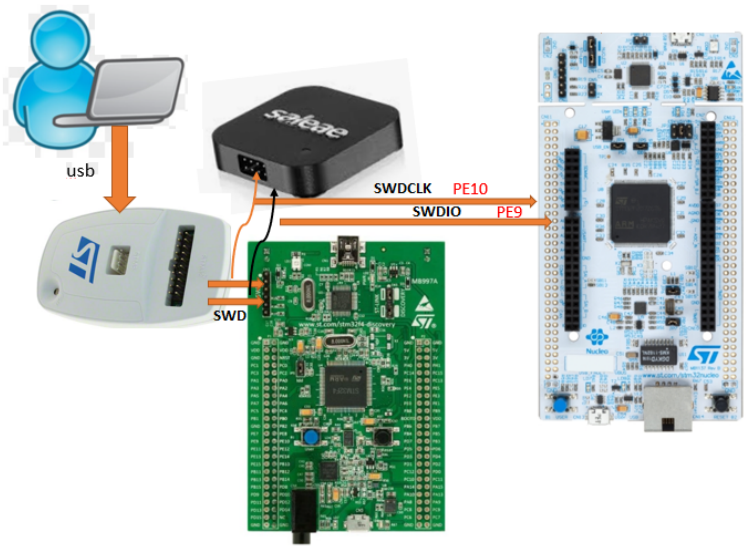

FIGURE 3.8 – The setup of the first phase

To technically implement this phase, we started by receiving the SWD protocol through the SPI internal peripheral, which caused problems since SWD requests,acknowledgments and payloads come in different lengths while the SPI frame size needs to be of a fixed length.
A better and more practical solution was making use of an important key concept in digital computing, that is interrupts.

3.1.1 Interrupts

In computers and microcontrollers, an interrupt is a request for the processor to interrupt currently executing code, so that the event can be processed in a timely manner. If the request is accepted, the processor will suspend its current activities, save its state, and execute a function called an interrupt handler (or an interrupt service routine (ISR)) to deal with the event.
This interruption is often temporary, allowing the software to resume normal activities after the interrupt handler finishes.
Interrupts are commonly used by hardware devices to indicate electronic or physical state changes that require time-sensitive attention.

3.1.2 SWCLK external interrupt

In our project with defined and implemented a hardware GPIO external interrupt (EXTI) on the falling edge of the SWCLK pin. This interrupt enables us to execute an interrupt service routine function on each falling edge of the SWCLK. The ISR function will make a call to a State Machine Shifter function, in which we will read the GPIO level on the SWDIO input pin . In other words, we read the current bit being transmitted by SWD on the SWDIO line on the falling edge that generated the interrupt and determine which state the SWD is currently in.
Figure 3.9 illustrates the state machine shifter function being run after getting called by the SWCLK interrupt.

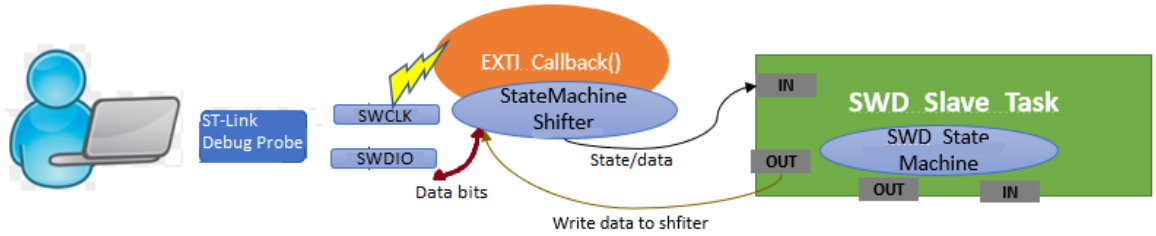

FIGURE 3.9 – A diagram showing the state machine being called by the interrupt

As we have mentioned, the *SWCLK* interrupt runs the state machine shifter, which is the core of the whole project.

3.1.3 State Machine Shifter

The state machine function is written in the form of a C switch case. It was gradually developed and changed at each phase of the project to meet the needs of each subtask being worked on. In the first phase, the state machine kernel was created. Its task was to simply read the current *SWDIO* bit and determine which state it belonged to and what state to transit to next.

The state machine we created is constituted of 9 states :

- **SWD_SLAVE_WAIT_FOR_START** - The initial state of the state machine. The program has this state as long as the *SWCLK* line is idle. When the first falling edge is detected, the sate transits to the first line reset state

- **SWD_LINE_RESET_1** - This state is the start of the SWD initialization phase. When entered in this state for the first time, the state machine will start counting the number of consecutive state entries in which the *SWDIO* is high(1). As mentioned in the previous chapter, a line reset is represented by at least 50 consecutive 1's on the *SWDIO* line. once we encounter the first low *SWDIO* level, we check if we really collected at least 50 consecutive high values before moving to the next state.
- **JTAG_TO_SWD** - The second initialization step is to the JTAG to SWD switching sequence. This makes sure the target's DAP is ready to accommodate SWD. The state machine will stay in this sate for 16 falling edges of the SWCLK signal. At each edge the state machine reads the corresponding SWDIO bit and assembles it into a 16-bit sequence.

  After the 16th bit, the program checks and confirms the correctness of the sequence. The *JTAG-to-SWD* sequence needs to be 0xE79E transmitted LSB first (0x79E7 in hexadecimal or 0b0111100111100111 in binary) . once the check is positive, we move to the next state which is the second line reset.
- **SWD_LINE_RESET_2** - A second line reset is performed to make sure the target is in a reset state after switching from JTAG to SWD. Again, we need to count at least 50 consecutive 1's on the *SWDIO* line. Once that's done, the SWD initialization is fully performed and we start waiting for a request in the next state.
- **SWD_WAIT_FOR_REQUEST** - During this state, the *SWDIO* is kept low by the debug probe, an SWD request is started by the start bit which is equal to 1. When a the high value is encountered, we enter the request state.
- **SWD_REQUEST** - The request is constituted of 8 bits. with 3 fixed bits (0th bit :$Start = 1$; 6th bit : $Stop = 0$; 7th bit : $Park = 1$).

  The most important bit in the request byte is the 2nd bit, *RnW*, indicating whether the request is a Read or Write request. This bit decides the format of the rest of the SWD operation. We also read the 4th bit, *parity*, and check its correctness.
- **SWD_TURNAROUND_RQ_ACK** - This turnaround phase happens on the first falling edge after the request has ended. During this turnaround period, the target takes control of the *SWDIO* line, and he's now responsible for writing data on it to be transmitted to the debug probe.

  No data is being transported on this *SWCLK* edge during this one cycle. It's a period of time left empty for the debug probe and the target to get ready to inverse their internal writing and reading states.

Our state machine takes advantage of this empty state to re-configure the interrupt to start running on the rising edge of the SWD clock, since the target will bang his *ACK* bits on the rising edge. we now get ready for the

- **SWD_ACKNOWLEDGE** - During the next 3 rising SWCLK edges, the target will write 3 acknowledgement bits as a response to the request. In this state we will read the 3 bits and depending on their value we decide the next state.
  As explained in *Chapter 2*, The *ACK* can take 3 values :
  - **1. *ACK OK* (100)** : an *ACK OK* response from the target indicates its readiness to proceed the SWD operation, whether it's a read or write request. After an *ACK OK*, we shall pass to the Payload phase if it's a Read request or we move to an intermediary, single-cycle, turnaround period for the debug probe to take back control of the SWDIO line.
  - **2. *ACK WAIT* (010)** : an *ACK WAIT* response means the target in not ready yet to answer to that request, The debug probe acknowledges that and loops back to the request phase.
  - **3. *ACK FAULT* (001)** - The *ACK FAULT* response is very rare as it indicates an error happened within the transmission, most probably caused by a hardware failure on one of the sides of the transmissions. an *ACK FAULT* is followed by the debug probe forcing a line reset and moving to the initialization phase again.
- **SWD_DATA_TRANSFER** - This is the payload phase. In the case of a read request, after the target has responded with its affirmative acknowledgment, it moves to sending back the requested data in the form of a 32-bit payload.
  On each rising *SWCLK* edge, we read the corresponding data bit from the SWDIO line, and shift it into the ith position of a 32-bit integer that represents the doubleword being transmitted.
  We read an additional 33rd bit that represents the parity of the payload. we check for its correctness and loop back to the *WAIT_FOR_REQUEST_STATE* as we give the debug probe control over the SWDIO line again. the debug probe will then take hold of SWD and start new requests.
- **NOTE :** A turnaround happens each time SWD changes data directions.
  4 Intermediary turnaround periods are inserted between some of the following states as a consequence to nature of the SWD operation and the target's response.

  1. There's always a turnaround between the request and the acknowledgment.

2. In the case of an *ACK WAIT*, a turnaround is performed before giving the debug host control over SWDIO.
3. On the Write request, after a positive acknowledgement by the target, a turnaround happens to enable the debug probe to write the data to target.
4. In a Read operation, after the target finishes writing its data to the debug host, it will again give control to the host.

Figure 3.10 represents a flowchart that explains the various possible states of the machine.

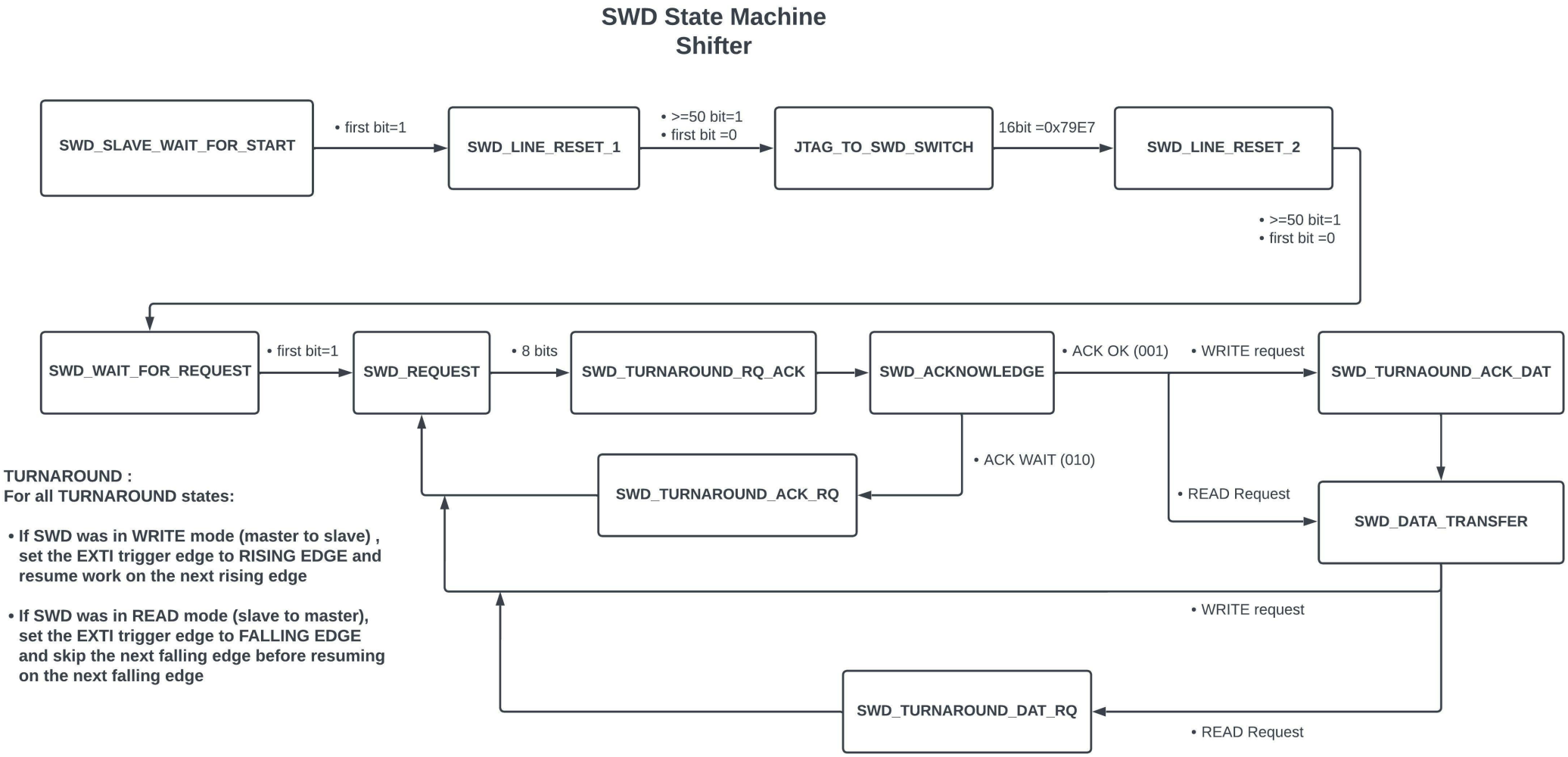

Figure.3.10: The SWD state machine flowchart

3.2 Phase2 : Interacting with the *SWDIO* line

In this section, we removed the target MCU and directly connected the debug probe to our development MCU containing the state machine.

The goal of this phase is to interact with the SWD protocol by playing the role of the target's DAP and answering requests sent by the ST-Link debug probe.

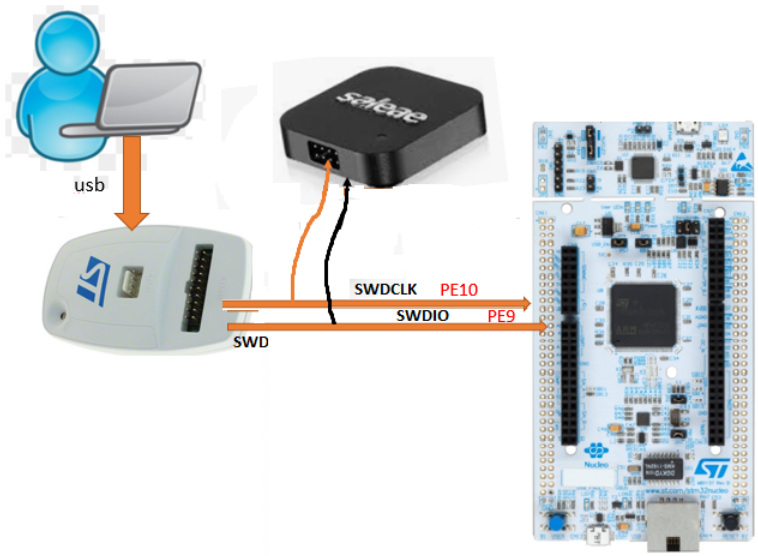

FIGURE 3.10 – An illustration of the phase 2 setup

All we had to do was receive the *SWDIO* request bits arbitrarily and the *SWD_ACKNOWLEDGE* state is reached, we change the *SWDIO* GPIO pin mode to output and acknowledge the request by responding with *ACK WAIT* (Since responding with *ACK OK* on read requests obliges us to answer with a correct payload)

As shown in Figues 3.11, pushing the *ACK WAIT* response to the debug probe forces it to re-send the same request which creates a loop.

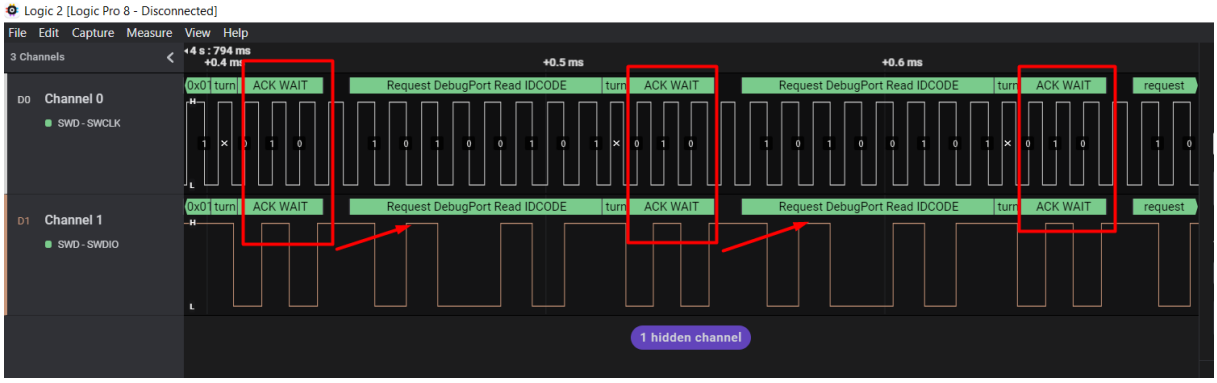

FIGURE 3.11 – A screenshot of the SWD in an ACK WAIT loop

Unless we respond with *ACK OK*, the SWD will stay in this Request-Wait loop. In the next phases of development, we'll make use of this behaviour in order to "pause" the SWD protocol and gain time while sending the data over Wi-Fi.

3.3 Phase 3 : Wired SWD bridge

This step played an important role in the project's development. During this phase we implemented a wired SWD bridge on the STM32F7.
This wired bridge will take the SWD as input, run it through the state machine, and bit-bang it on two GPIO output pins.
These pins will play the role of a debug probe outputting *SWDIO* and *SWCLK* signals, as illustrated in Figure 3.12.

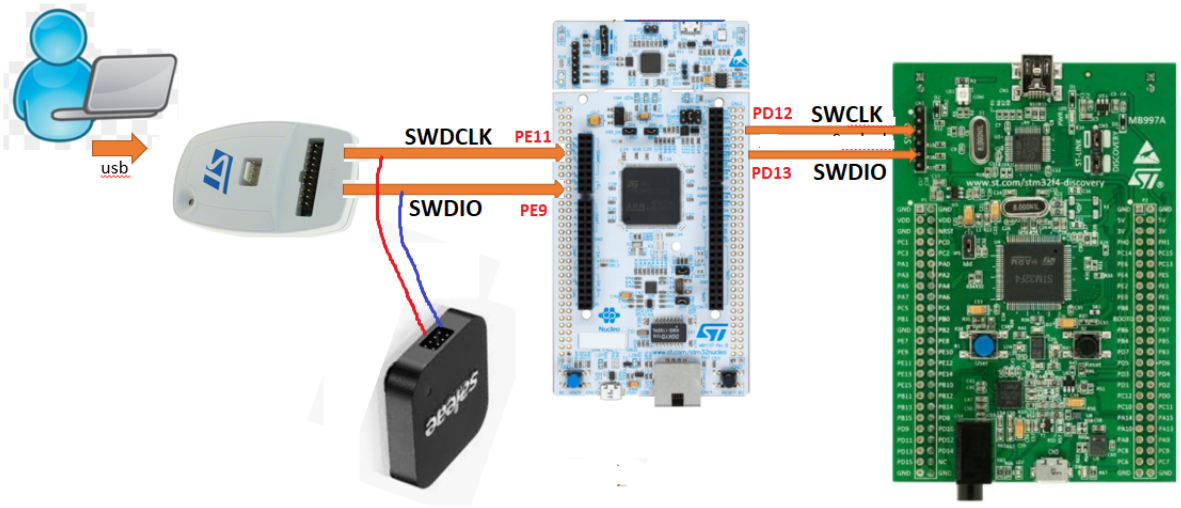

FIGURE 3.12 – An illustration of the setup of phase 3

3.3.1 The ACK trick

This subsection will explain a key concept that made this project possible. A simple trick to force the debug into tolerating response delays.

The debug probe doesn't know the protocol in being transformed over a bridge. Its clock won't stop and the probe will always send requests and ask for acknowledgement. If we fail to answer to the acknowledgement, it's considered to be a transmission failure and the debug probe will perform a line reset.
To keep the protocol running, an acknowledgement needs to be delivered by the state machine on the STM32F7. As we explained in *Chapter.2*, an SWD request can either be a *Read* or a *Write* request. In a read request the debug host is asking the target to send him some particular data. In a Write request, the debug probe is attempting the send his data to the target.

Transporting the requests over the bridge can take time, which prohibits us from answering with a positive acknowledgment as we aren't sure we can do the SWD operation on the target's side in time. We came up with a straightforward solution for this problem in both cases of *Read* and *Write* requests.

- **Read request** - In a read request, wel will answer with an ACK WAIT while we send the request to the master side. The debug probe will send the same request over and over as we keep answering with ACK WAIT. During this time, the request is transmitting to the master, which is then written to the target. The master reads the requested payload data and sends it back to the state machine. Just then we can break the ACK WAIT loop and answer with an ACK OK followed by the target's 32-bit payload.
- **Write request** - In a write request, the payload data is supplied by the debug probe, So there's no purpose of answering with an ACK WAIT as we need to get hold of that data first before sending it. In a write request we answer with an ACK OK, we then read the payload data from the debug probe and send the request header and the payload in the same packet to the master side. The master will return a confirmation after if finishes writing on the SWD lines, until that confirmation is returned, the state machine will answer with an ACK WAIT on the next request regardless of its nature(a

read or a write request).

This part was technically implemented inside the state machine by verifying the state of multiple flags coming from the master side.
To communicate flags and SWD data between the slave and the master sides, we created 2 FreeRTOS tasks ; a Slave_task and a Master_task.

3.3.2 The Slave_task

After assembling the necessary bits that form an SWD operation, we notify the *Slave_task* and we pass it the corresponding state and data of that phase.
For example, after the state machine finishes reading the last bit of the *IDCODE* request, the ISR awakens the *Slave_task* and passes it two values in the form of a struct ; the first field representing the current state (*{request}*) and the second field representing the request data (*{10100101}*).
The tasks understands this data, puts it in a buffer, and sends it to the *Master_task.*

3.3.3 The Master_task

As the name indicates, the *Master_task* is responsible for acting as a master to the debug target. The *Master_task* implements his own state machine and is responsible for bit-banging the SWD data on two GPIO pins.
The *Master_task* generates his own clock and writes data to its edges. We implemented a straightforward solution to create these edges by simply inverting the *SWCLK* output level. For example ; To create a simple falling edge, we just pull the *SWCLK* pin up to a high(1) level, then pull it down to a low(0) level.

To explain further with a simple example : when *Slave_task* passes the IDCODE request (*10100101*) to the Master_task, the task is awakened from its blocked state and receives the following struct *{request,10100101}*. The task then loops through the request value bit by bit. For each bit, *Master_task* does the following :

1. Sets the *SWCLK* pin high(1)
2. Pulls the *SWDIO* pin to the value of the *ith* bit of the request.

3. Resets the *SWCLK* pin to low(0), creating a falling edge.
4. Repeats the process for bit *i+1* .

By following this simple 4-step algorithm, we successfully wrote SWD data without the need of any complex peripherals.

In order to read acknowledgement or data from the target, We should deliver rising edges so the target can write his bits on the *SWDIO* line.
a similar algorithm is implemented that does the following :

1. Resets the *SWCLK* pin low(0)
2. Reads a bit from the the *SWDIO* line.
3. Inserts that bit into the *ith* position of a variable.
4. Sets *SWCLK* pin to high(1), creating a rising edge.
5. Repeats the process to get the *(i+1)th* bit .

After reading acknowledgement or payload data, The *Master_task* is now ready to send that data back to the *Slave_task*.
The *Slave_task* gets awakened. And depending on the current operation performed, the task will set values and flags inside the State Machine function.
These flags determine whether the state machine should exit the *ACK WAIT* loop and write the received data to the debug probe.

Figure 3.13 shows how the state machine responds with *ACK WAIT*, forcing the debug probe to re-send the request, while transmitting that request to the *Master_task* through an intermediary *Slave_task*. We can also notice how the *Master_task* bit-bangs that requests and proceeds to read the acknowledgement and the payload from the target. We also can also verify the correctness and integrity of the the payload as it gets sent back to the *Slave_task* and written on the debug probe's *SWDIO* line after exiting the *ACK WAIT* loop.

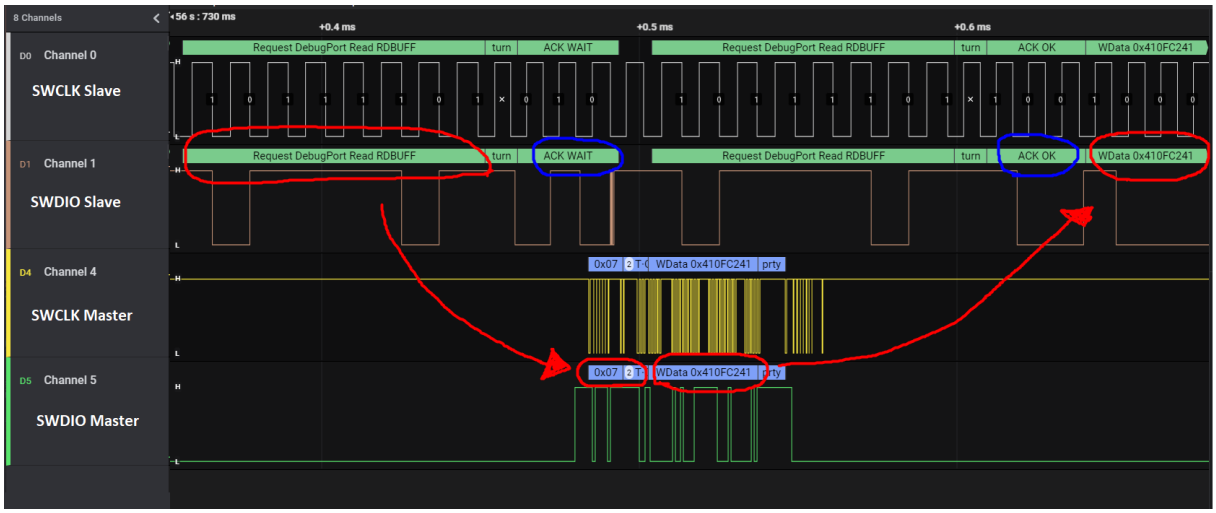

FIGURE 3.13 – A screenshot of the SWD signals transmitted on the wired bridge.

This wired SWD bridge works as the middleman between the debug probe and the target MCU.
The wired bridge's purpose is to set a clear architecture and a developing environment for the next and final phase of the project.
The 2-task architecture was adopted to make it easier for us to port the code on two microcontrollers connected wirelessly in phase 4.

3.4 Phase 4 : Wireless SWD interface

This is the last step of the project. We couldn't reach this phase without going through the previous intermediary steps.
In this part, we're going to build on the existing wired bridge.

The same wired SWD bridge code will be ported on two different MCU's. A *Slave* and a *Master*. The imagined solution looked like the Figure 3.14 shows.

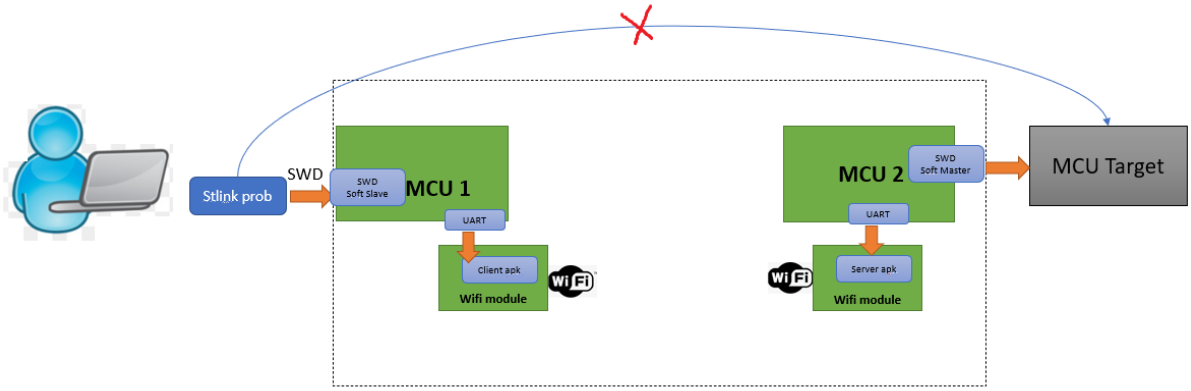

FIGURE 3.14 – An illustration showing the master ans slave MCU's connected to Wi-Fi modules.

These *Slave* and *Master* MCU's will be connected through UART to 2 Wi-Fi-enabled modules communicating encoded SWD data.

The proposed solution for the Wi-Fi modules is two ESP32-S2 microcontroller.
ESP32-S2 is a highly integrated, low-power, single-core Wi-Fi Microcontroller system on chip, designed to be secure and cost-effective, with a high performance and a rich set of peripherals and input/output capabilities.

For its high performance, We decided to have an ESP32-S2 as the master MCU. The master's job was to receive *slave* data and bit-bang it to the target's SWD lines, then return another set of data over Wi-Fi to the *slave*.
Since the bit-banging task wouldn't interfere with the network task responsible for the Tx/Rx routines, we can have the ESP-S2 as the the master MCU and Wi-Fi module simultaneously.

On the Slave side, the single-core ESP32-S2 couldn't run the Wi-Fi network task while reacting to the SWCLK interrupts in time. Because of that, we had to keep the STM32F7 on the slave side. The STM32F7 will be intercepting the *SWCLK* interrupts, running the state machine and finally communicating the data over UART to a slave ESP32-S2 responsible for directly communicating with the master ESP32-S2 as illustrated in figure 3.15.

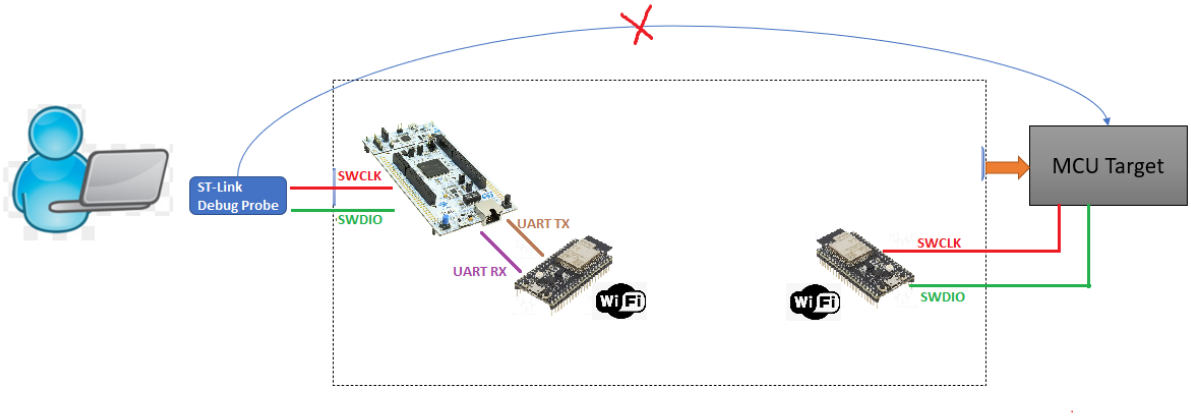

FIGURE 3.15 – An illustration of the SWD-to-UART setup

3.4.1 UART communication

We enabled the UART peripherals on the STM32F7 and the slave ESP32-S2. The *Slave_task* that receives data from the state machine will now fill it in a fixed size buffer and send send that buffer over its UART Tx line.
The ESP32-S2 will receive that data on its Rx line. The reception of the data buffer will generate an UART interrupt service routine and a callback will be executed.
The Slave ESP32-S2 will then proceed to send the data wirelessly.
If data comes from the master side, the ESP32-S2 will intercept that data and send in through its UART Tx line to the STM32F7's *Slave_task*.

The slave ESP32-S2 is nothing but an UART-Wi-Fi bridge.
The MCU's purpose is to simply transport whatever received on its Rx buffer over Wi-Fi. It doesn't implement any logic related to SWD.
This slave ESP32-S2 can serve as a general UART-to-Wireless solution for transmitting and receiving UART data wirelessly. This MCU can be used in other projects requiring UART communication over a long distance between two or more devices.

3.4.2 Wireless solutions

The implementation of a successful and reliable wireless connection played a decisive part in our project's life cycle.
The wireless transmission needs to transport data without any errors or missing data in with a short delay.

Remember, each SWD request needs to be send over the air, treated by the Master MCU that sends confirmation and data back to the Slave. This maneuver can cause large delays, as debug operations are usually constituted of tens of thousands of SWD requests.

Before establishing any communication, we implemented the necessary Wi-Fi initialization code on both ESP32-S2 MCUs. The master MCU will start an Access Point (AP) and will be configured with a an SSID and a password. after the chip is powered on, the network is created and will appear as a normal Wi-Fi network on other devices. Computers and smartphones can connect to this network when provided with a password as shown by the screenshot in figure 3.16.

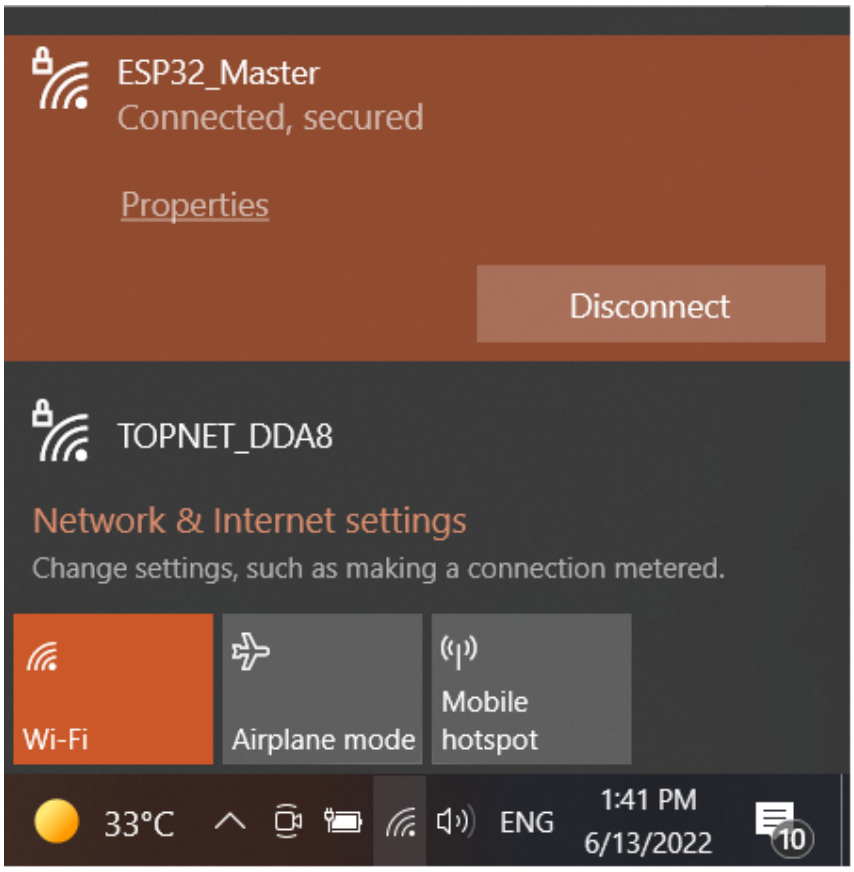

FIGURE 3.16 – A screenshot showing the Wi-Fi network created by the ESP32-S2 master

As explained in *Chapter.2*, Wi-Fi operates at the first two OSI layers : Data-Link and Physical layer.In order to communicate data in an organized manner, We need a solution that operates a higher layer.
After doing some research on the available protocols for communicating data. We came across 3 possible solutions : UDP, TCP and ESP-NOW.

3.4.3 UDP

User Datagram Protocol (UDP) is a connectionless communications protocol that is primarily used to establish low-latency connections between applications over a network. UDP speeds up transmissions by enabling the transfer of data before an agreement is provided by the receiving party.
The main disadvantages of UDP is the unreliability. UDP doesn't acknowledge received packets, so if a transmission error is detected, the receiving device will just drop the datagram.

Our project can't tolerate data loss, as every byte transmitted over SWD in necessary for the programming and debugging process.
This solution is dropped instantly.

3.4.4 TCP

TCP stands for Transmission Control Protocol a communications standard that enables application programs and computing devices to exchange messages over a network. It is designed to send packets across a network and ensure the successful delivery of data and messages.

TCP organizes data so that it can be transmitted between a server and a client. It guarantees the integrity of the data being communicated over a network. Before it transmits data, TCP establishes a connection between a source and its destination, which it ensures remains live until communication begins. It then breaks large amounts of data into smaller packets, while ensuring data integrity is in place throughout the process.

To implement the TCP solution on both ESP32-S2 microcontrollers, we needed to establish a connection first. A connection on TCP happens through sockets.
A socket is an endpoint in a bidirectional data. It's characterized by an IP address and a port number.

After assigning a fixed IP address and port number for both ESP chips, We set the master MCU to be the TCP server and the slave MCU to be a client. A connection was then established after a request from the client to the server.
Data packets can now be send and received through the opened sockets.

We assembled the different project components together and we tested it using the

logic analyzer as shown by the capture in figure 3.17.

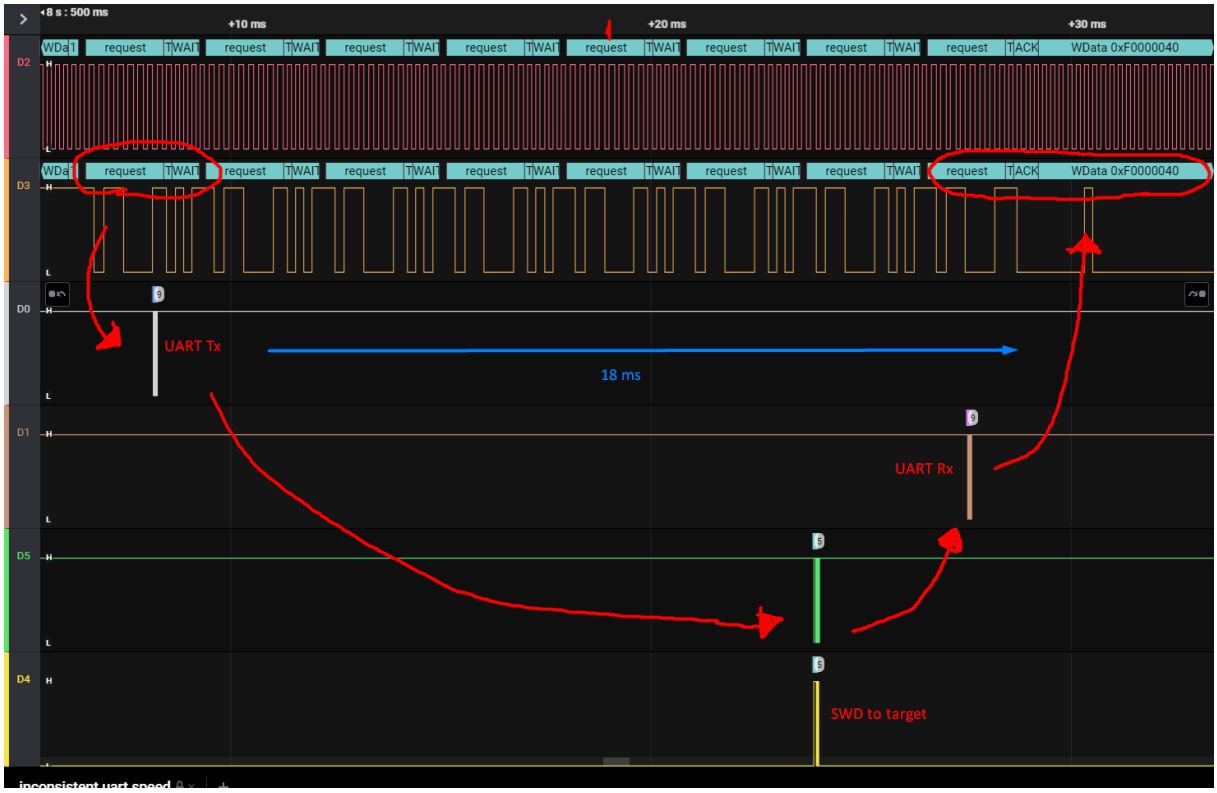

FIGURE 3.17 – A screenshot showing SWD being transmitted on TCP.

The Figure 3.17 shows the SWD data being communicated between the different project components using different protocols.

The 6 channels represent :

— **Channel 0 :** SWCLK from the debug probe.

— **Channel 1 :** SWDIO from the debug probe. A read request being forced into the *ACK WAIT* state by the STM32F7 state machine.

— **Channel 2 :** SWD read request encoded in a UART frame on the Tx line.

— **Channel 3 :** The returned SWD data/confirmation from the master ESP32-S2. Transmitted from the ESP32-S2 bridge back to the STM32F7 slave Rx over UART.

— **Channel 4 :** SWCLK clock signal created by the *Master_task* running on the master ESP32-S2.

— **Channel 5 :** SWDIO data bit-banged by the master ESP32-S2 on the target SWDIO line.

As you you can see in the Figure above, The implemented SWD transportation chain lasted approximately 18 milliseconds to transport a SWD request to and from the ESP32-S2 master through the UART and the TCP stack.

18 milliseconds is considered relatively high for transporting a single request. transporting tens of thousands of SWD requests would result in minutes of operation delay. To put it into context, flashing a 5KB binary file through this TCP wireless interface would approximately take 6 minutes.
The chain does the work as intended, but the long delay makes it unpractical to use in real embedded development situations.

This delay is mainly caused by the heavy TCP stack overhead. We have to look for a TCP alternative with less overhead in order to reduce the wireless round-trip-time.

3.4.5 ESP-NOW

ESP-NOW is a wireless communication protocol developed by Espressif, which enables multiple devices to communicate with one another wirelessly. It's best described as a low-power 2.4GHz protocol. The pairing between devices is needed prior to their communication. After the pairing is done, the connection is safe and peer-to-peer, with no handshake being required.
ESP-NOW features short packet transmission with a maximum packet size of 250 bytes. The protocol offers fast peer-to-peer wireless data transfers among members of Espressif's ESP MCU family with very little overhead.

With ESP-NOW, each ESP32 MCU can be a sender and a receiver at the same time. A two-way communication can be established between boards as shown in the Figure 3.18.

FIGURE 3.18 – An illustration of two ESP MCU's connected with ESP-NOW

To communicate via ESP-NOW, we need to know the MAC Address of the ESP32 receiver and transmitter. That's how we know to which device we'll unicast the data to. Each ESP32 has a unique MAC Address and that's how we identify each board in a the

network.

The ESP-IDF frameworks provides many APIs that act as endpoints to ESP-NOW protocol. All we have to do in the current code, is to remove the TCP implementation and replace it with the proper ESP-NOW code for initialization, sending and receiving

Each ESP32-S2 is provided with its peer's MAC address. We then pass it the data we want to send. Upon data reception, a ESP-NOW callback is triggered. Inside this callback we perform the same routines that we did in TCP code.

Finally, and after implementing all the necessary code, we link the components together to test it. The Figure 3.19 shows the same signals captured in Figure 3.17.

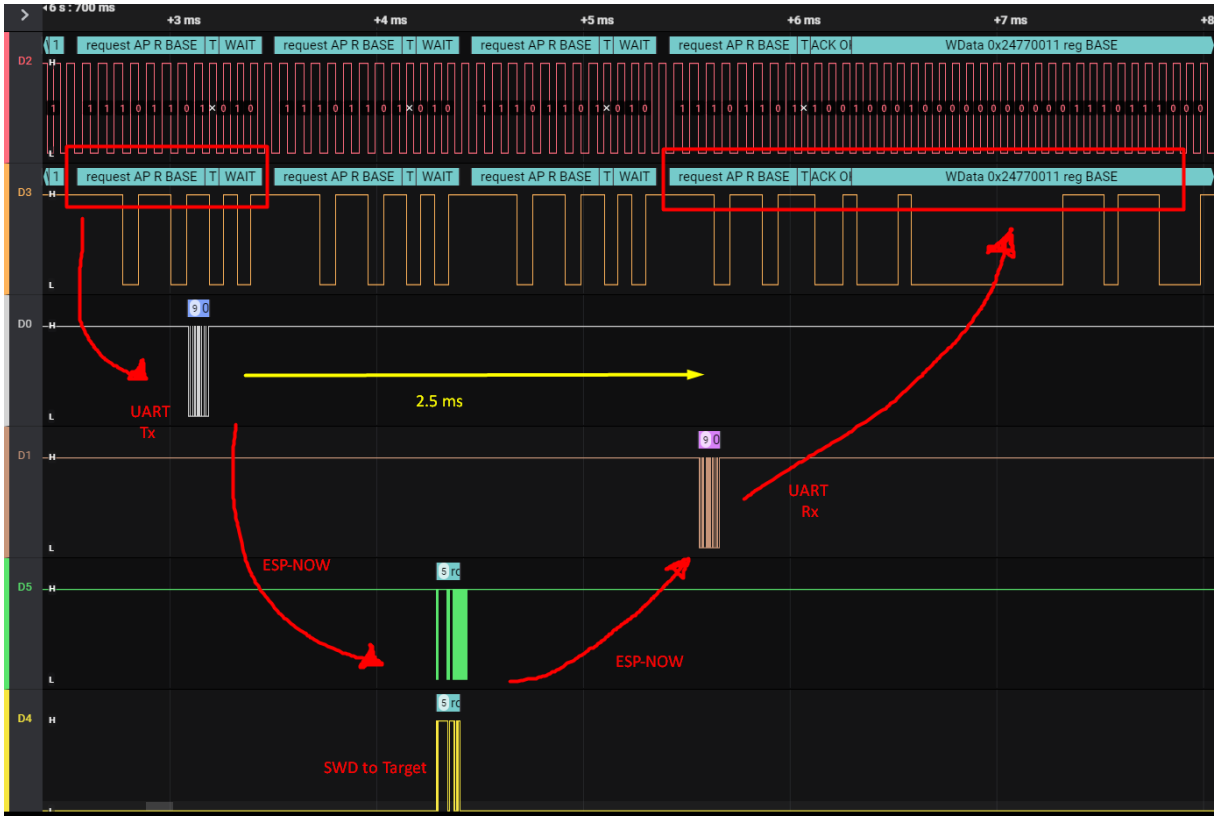

FIGURE 3.19 – A screenshot showing the SWD data being sent over UART and ESP-NOW

As we can observe, The wireless bridge worked perfectly. With a round-trip time approximately in the 2.5 milliseconds range.

It's now possible to program and debug a microcontroller without physically connecting it to a computer.

3.5 Conclusion

In this chapter we have proved the technical feasibility of a wireless Serial Wire Debug interface. A functional proof of concept was developed through multiple phases.

This project serves as a practical wireless debugging interface that will solve many problems related to debugging microcontrollers embedded in mobile device, such as micro-mobility systems.

3.6 Testing and benchmarking

In this part, we're going to test and discuss the wireless debug chain that we have implemented. The testing and benchmarking will be bases on two criteria : delay and distance.

- **Delay :** Even though ESP-NOW is approximately 7 times faster than the TCP implementation, it still takes on average 30-40 seconds to flash a 5KB binary file, which is slower than a direct wired connection. But it's still practical in some situations where making a wired connection to a distant target can be more time and effort consuming. When debugging with an IDE, setting breakpoints takes on average 4-5 seconds to take effect. on the other hand, reading data through live watchpoints happens at a fast rate and almost instantly.
- **Distance :** The ESP-NOW can be set to be used in a long range mode, exceeding 300 meters.

  We tested the wireless interface at two distance points : 2 meters and 20 meters. The 2 meters test resulted in a shorter RTT. A longer range communication is often blocked by objects and is immensely interfered by other wireless signals operating at the same 2.4Ghz frequency. Using a less frequent channel can improve RTT by up to 15% .

Conclusions and perspectives

Our end of studies project consisted of developing a Wireless Serial Wire Debug interface for microcontrollers. This project is considered to be a first of its kind with innovative technicals workarounds.
The problem that we tried to solve with our implementation was that of the difficulty to program and debug mobile devices that are hard to reach through a wired interface. A wireless debugging interface also enables embedded developers to safely program and debug dangerous systems from a safe zone.

Our solution consisted of using two Wi-Fi-enabled microcontrollers connected to a debug probe through a state machine, making a wireless debugging bridge capable of transporting the SWD protocol between a debug probe and a target device.
This was done through the development and the implementation of an architecture and an algorithm that would run on three different interconnected microcontrollers using various embedded software techniques and communication protocols.

Our wireless debugging solution can be better improved by letting go of the debug probe and using an ESP32 board as a USB dongle that directly parses the USB debug data from the host computer and sends it over the air to an SWD master, consequently reducing the volume of bi-directional data being transported wirelessly. Implementing a wireless USB-to-SWD interface would result in faster and more robust debugging that is prone to less delay.

References

[1] Actia group - accueil. https://www.actia.com/fr/. (Accessed on 12/04/2022).

[2] Gnu debugger. https://www.sourceware.org/gdb/documentation/. (Accessed on 02/05/2022).

[3] Openocd documentation. https://openocd.org/doc/html/About.html. (Accessed on 12/03/2022).

[4] Serial wire debug, arm docs. https://developer.arm.com/documentation/100893/0100/Debug-and-trace-interface/Serial-Wire-Debug--SWD--signals. (Accessed on 26/03/2022).

[5] Swd, cdsn. https://www.cnblogs.com/shangdawei/p/4753040.html. (Accessed on 21/04/2022).

[6] Swd ack wait behaviour. https://www.kernelpicnic.net/2018/12/29/Messing-with-SWD-Part-I.html. (Accessed on 21/04/2022).

[7] Debug access port, arm docs. https://developer.arm.com/documentation/102585/0000/what-is-a-debug-access-port. (Accessed on 01/04/2022).

[8] Osi model. https://www.imperva.com/learn/application-security/osi-model/. (Accessed on 07/05/2022).

[9] Tcp. https://www.fortinet.com/resources/cyberglossary/tcp-ip. (Accessed on 10/05/2022).

[10] Ieee802 website. https://www.ieee802.org/11/. (Accessed on 10/05/2022).

Title: Wireless Serial Wire Debug interface.

Abstract: This work serves as a proof of concept of a wireless interface for the Serial Wire Debug protocol. This interface allows users to program and debug microcrollers that are not accessible through wired interfaces.
This study will be followed by an establishment of a detailed technical structure of the solution and the implementation of all the intermediary steps. .

Keywords: SWD, STM32, ESP32, UART, Wi-Fi, TCP, ESP-NOW

Titre: Interface Serial Wire Debug sans fil.

Résumé: Ce travail sert comme une preuve de concept d'une interface sans fil pour le protocole Serial Wire Debug. Cette interface permet aux utilisateurs de programmer et de déboguer des microcrollers qui ne sont pas accessibles par des interfaces filaires.
Cette étude sera suivie par l'établissement d'une structure technique détaillée de la solution et la mise en œuvre de toutes les étapes intermédiaires.

Mots-clés: SWD, STM32, ESP32, UART, Wi-Fi, TCP, ESP-NOW

العنوان: واجهة لاسلكية لبروتوكل **Serial Wire Debug**

ملخص: يعتبر هذا العمل بمثابة إثبات لمفهوم واجهة لاسلكية لبروتوكول Serial Wire Debug . تتيح هذه الواجهة للمستخدمين برمجة وتصحيح وحدات التحكم الدقيقة التي لا يمكن الوصول إليها من خلال الواجهات السلكية.
سيتبع هذه الدراسة إنشاء هيكل تقني مفصل للحل وتنفيذ جميع الخطوات الوسيطة.