



**Politecnico
di Torino**

Politecnico di Torino

Master's Degree in Mechatronic Engineering
A.a. 2022/2023
Sessione di Laurea Aprile 2023

Development of a Linux Networking Driver and Firmware for the microcontroller ESP32- WROVER-E.

Relatore:
Prof. Luciano Lavagno

Candidato:
Enrico Valsania

Tutor aziendale:
Ing. Giuseppe Poma

Summary

This Master's Thesis was carried out at Bitron S.p.a, in the R&D Charging department. The purpose of this work is to write a Linux driver and corresponding firmware for the ESP32-WROVER-E Wi-Fi module in order to use this device on an electric vehicle charging station as a Wi-Fi Station Point or Wi-Fi Access Point. Moreover, is necessary to understand if this microcontroller can be adopted in industrial environment. The adoption of this module allows the user and the company providing the service to monitor the functioning of the charging station, to have a wide range of data available, and to set a suitable charging profile for the user. Being a microcontroller, it implements MAC layer functionality of the 802.11 protocol within its firmware, making it a 'HardMAC' (also called 'FullMAC') device. Therefore, the mac80211 layer does not need to be implemented within the Linux driver, as will be explained in greater detail in the following sections.

Specifically, this Linux driver is used for recognition and proper communication by the SoM (which consists of an STM chip on which the Linux kernel 5.10.61 is implemented) with the Espressif module. In addition, an SPI protocol is used for packet and information exchange between the SoM and ESP. The SoM acts as a Master in the communication while the ESP32 acts as a slave.

The overall code (on the Linux side) consists of a module that implements SPI communication, network interface configuration, and transmission of 802.11 packets from the SoM to the ESP32, while the firmware implemented on the ESP32 side consists of the SPI protocol and the ability to function as a station (STA) or as a hotspot (AP).

Overall, the work carried out covers various aspects that will be addressed in the following pages, such as the 802.11 protocol and packet analysis through Wireshark, the use of SPI and a logic analyzer, APIs provided by Espressif using FreeRTOS, and the use and examination of Linux device drivers associated with the networking layers of Linux.

Table of Contents

List of Figures	VI
Acronyms	VIII
1 Analysis of the proposed project and related protocols	1
1.1 OCCP	1
1.2 The proposed project analysis	2
1.3 Protocols analysis	4
1.3.1 SPI	4
1.3.2 802.11	6
1.4 Hardware Analysis	11
1.4.1 ESP32-WROVER-E	11
1.4.2 SoM	15
2 Explanation of softawre environment	16
2.1 Espressif	17
2.1.1 Wi-Fi driver	17
2.1.2 Event-Handling	18
2.1.3 Wi-Fi APIs Error Code	19
2.2 Linux Networking and Device Drivers	19
2.2.1 Linux Device Driver	20
2.3 FreeRTOS	23
3 Code Implementation	26
3.1 Software Architecture	26
3.1.1 ESP32	27
3.1.2 Linux Driver	28
3.2 How to build	28
3.2.1 Linux Driver	28
3.2.2 ESP32	30
3.3 SPI	30

3.4	Networking configuration	39
3.4.1	Firmware ESP32	39
3.4.2	Linux Driver	43
4	Results obtained	51
5	Conclusion	60
A	Driver Code	62
A.0.1	cfg80211esp.c	62
A.0.2	espspi.c	72
A.0.3	netdevice.c	78
A.0.4	cfg80211esp.h	78
A.0.5	espspi.h	81
	Bibliography	83

List of Figures

1.1	Architecture of the project	3
1.2	Basic connection diagram between devices using the SPI interface .	5
1.3	Correspondence between modes and CPOL, CPHA parameters. . .	6
1.4	ISO/OSI model	7
1.5	IEEE 802.11n channels in the 2.4 GHz band	8
1.6	MAC data format	9
1.7	Frame control format	10
1.8	Project specifications	12
1.10	Architettura periferica SPI	13
1.9	Caratteristiche Wi-Fi	14
2.1	Wi-Fi programming model	18
2.2	Device Linux Driver	20
2.3	Split view of the Kernel	21
3.1	Block diagram of the above implementation	34
3.2	Block diagram of the policy to send packet from ESP to SoM	37
4.1	SoM and ESP32	52
4.2	Loading of esp32_bb module	53
4.3	ESP32-WROVER-E's mac address on SoM terminal.	54
4.4	ESP32-WROVER-E's mac address ESP32 terminal.	54
4.5	enrico0 network interface	54
4.6	Wiphy enrico capabilities	55
4.7	Read of the information provided by ESP32 on SoM terminal. . . .	56
4.8	Wi-Fi's find by ESP32	56
4.9	Connection of ESP32 to network Enrico	57
4.10	enrico0 network capabilities.	57
4.11	Packet sent by SoM to ESP32, seen on SoM terminal.	58
4.12	Packet sent by SoM to ESP32, seen on ESP32 terminal.	58
4.13	Update metrics of the enrico0 interface.	58

Acronyms

AP

Access Point

API

Application Programming Interface

CSMS

Charging Station Management System

DMA

Direct Memory Acces

EV

Electric Vehicle

HAL

Hardware Abstraction Layer

OCPP

Open Charge Point Protocol

SPI

Serial Pheriperal Interface

STA

Station

Chapter 1

Analysis of the proposed project and related protocols

As stated in the introduction, the purpose of this Thesis is the implementation of a Linux module and firmware for the ESP32-WROVER-E Wi-Fi module to provide connectivity to an electric vehicle charging stations. Currently, the company uses the ATWILC1000 controller module for this purpose.

1.1 OCCP

The electric vehicle charging stations require an internet connection to comply with the Open Charge Point Protocol (OCPP), which is the industry-supported de facto standard for communication between a Charging Station¹ and a CSMS² (Charging Station Management System) and is designed to accomodate any type of charging technique.

Therefore, the purpose of this protocol is to ensure that each EV (Electric Vehicle) Charger works properly with any charger management software using JSON over Websockets supporting Compression. Among the main features offered by this protocol is Device Management, which effectively manages a network of (complex) charging stations (from different vendors). It provides inventory reporting, improved error, state reporting, and configuration, and customizable monitoring. Finally, it allows user authentication, enabling charging via RFID (Radio-Frequency Identification) technology. Additionally, it allows users to engage

¹A Charging Station is the physical system where an EV can be charged

²A CSMS manages Charging stations and has the information for authorizing Users for using it

in Smart Charging: the amount of energy used can vary based on the number of people using electricity at that time, so as not to burden the network too much. Smart charging also prevents the building's maximum load capacity from being exceeded, which is normally defined by the local network capacity and the energy power provided by one's tariff.

Furthermore, smart charging allows public utility companies to set certain limits on energy consumption, thereby preventing network overload due to energy usage that exceeds what is produced.

Thanks to this system, time, money, and especially energy are saved, contributing to the protection of the planet's resources. Therefore, it can be asserted that it is necessary to provide the charging station with internet connectivity as it allows for a smarter use of the charging station.

1.2 The proposed project analysis

As previously stated, the project proposed to me consists of writing a Linux networking driver and its corresponding firmware to implement internet connectivity to a charging station. The Wi-Fi is regulated by the IEEE 802.11 protocol. Among other features, IEEE 802.11 protocol defines a set of transmission standards for WLAN networks developed by the IEEE 802 group 11, in various releases, with particular emphasis on the physical and MAC layers of the ISO OSI model. These versions vary in characteristics such as bandwidth, transmission speed, modulation standards, collision avoidance protocols, and many others, which will be explained in more detail in the upcoming sections. As previously mentioned, the SoM and ESP32-WROVER-E communicate through the SPI protocol³: the SoM acts as the master and the ESP32-WROVER-E as the slave during communication. Through this protocol, the SoM sends the 802.11 packets it needs to transmit on the network and the ESP32 receives and forwards them to Linux, as well as initial configuration messages that the SoM sends to the ESP32 for proper microcontroller setup. Of course, this protocol will be analyzed in more detail in the following sections. The following figure shows the macro structure of the overall project architecture.

³SPI (Serial Peripheral Interface) is a serial communication protocol between microcontrollers and other integrated circuits, or between multiple microcontrollers.

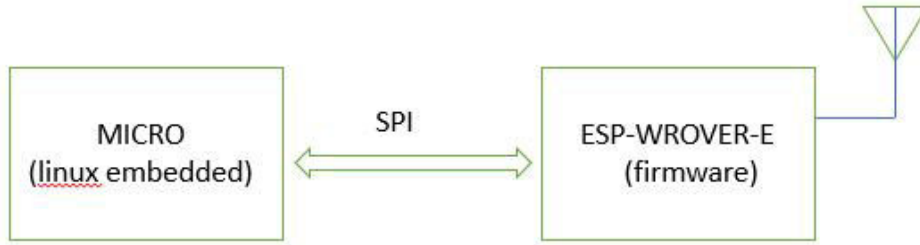


Figure 1.1: Architecture of the project

Especially, in the case at hand, the driver is to be implemented on a System on Module (SoM) that will be mounted on a charging column. In our company's projects, the SoM generally serves as the intelligence where the software side (i.e. the Linux operating system) and, among other functionalities, the OCPP protocol are implemented. Meanwhile, other boards located within the column serve for low-level implementation, namely data acquisition, inverter driving, and other functions. In the case at hand, the mounted Linux kernel is a custom-built project built using Yocto⁴ of the version 5.10.61. To correctly compile a Linux module, it is necessary to build it against the Linux libraries of the specific kernel version adopted, to avoid discrepancies between expected and actual functionality. In fact, being an Open Source world, the entire community is constantly making changes to the Kernel source code to make it more reliable, reduce the number of possible bugs, and implement new functionalities. Therefore, it is essential to analyze the source code of the Kernel version to have a clear idea of its functioning. In the specific case, I focused on the Linux Cfg80211 layers⁵, netdev⁶, SPI. Therefore, I built a module using the functions offered by these layers to compile a single binary that is built, transferred to the SoM, and then loaded against the currently Kernel on the SoM.

As for the firmware, the foundation is FreeRTOS⁷. It is a popular operating system kernel used in embedded devices, compact and easy to use. It is used

⁴Yocto is an open-source set of tools that allows you to obtain custom operating systems for embedded systems based on Linux.

⁵Cfg80211 is the configuration API for 802.11 devices in Linux. It bridges userspace and drivers, and offers some utility functionality associated with 802.11. cfg80211 must, directly or indirectly via mac80211, be used by all modern wireless drivers in Linux, so that they offer a consistent API through nl80211.

⁶netdev is used to describe and register correctly to the Linux Kernel the feature of the device connected to the hardware where is load Linux.

⁷Real-time operating system for microcontrollers.

for creating threads or multiple instructions, mutexes, semaphores, and software timers, supporting priorities between tasks. Typically, it is used for industrial applications and does not need to be fast: the important thing is not the interval of time in which the operating system must react, but that it responds within a predetermined maximum time. In other words, the system must be predictable or rather deterministic, meaning that the real timing (in best or worst-case scenarios, terms that come from English) of a specific process or computation can be known in the system.

In practice, a real-time system must guarantee that a computation (or task) ends within a given temporal constraint or deadline. To ensure this, it is required that the scheduling of operations is feasible. The concept of scheduling feasibility is at the basis of real-time systems theory, and it is what allows us to determine whether a set of tasks is executable or not based on given temporal constraints.

On this kernel generated through FreeRTOS, the Networking APIs are used⁸ provided by Espressif. In particular, I focused on the implementation of Wi-Fi driver that can be considered a black box that knows nothing about high-layer code, such as the TCP/IP stack, application task, and event task. The application task (code) generally calls Wi-Fi driver APIs to initialize Wi-Fi and handles Wi-Fi events when necessary. Wi-Fi driver receives API calls, handles them, and posts events to the application. As will be explained later, an application should not be written in the User Space but rather in the underlying layers.

1.3 Protocols analysis

In order to better understand the project and make appropriate considerations, it is necessary to examine the SPI and 802.11 protocols.

1.3.1 SPI

The SPI protocol or more precisely the SPI interface was originally developed by Motorola (now Freescale) to support their microprocessors and microcontrollers. The SPI interface describes a single Master single Slave communication, and is synchronous and full-duplex. The clock is transmitted on a dedicated line (not necessarily a synchronous transmission has a dedicated clock line) and it is possible to transmit and receive data simultaneously. The interface also has 4 connection lines (excluding the necessary ground), hence the SPI standard is also known as a 4 Wire Interface. It is the Master's responsibility to initiate communication and provide the clock to the Slave.

⁸Application programming interface

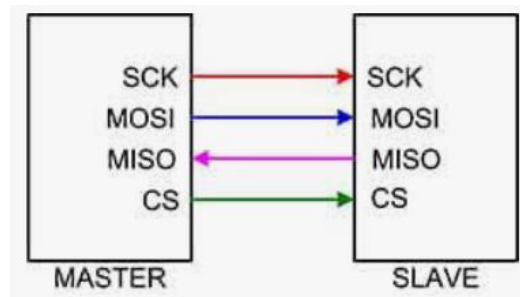


Figure 1.2: Basic connection diagram between devices using the SPI interface

As seen in 1.2, the 4 connecting lines so called:

1. MISO: Master Input Slave Output
2. MOSI: Master Output Slave Input
3. SCLK: Serial Clock (generato dal Master)
4. SS: Slave Select

Note that SPI communication requires the presence of an SS (Slave Select), so even if communication occurs between a single Master and a single Slave, the Master can select the Slave with which to communicate, both for writing and reading data. If there are multiple Slaves present, their connection is generally made as shown in Figure 2. As you can see, there is still a single Master, which this time has the task of selecting the Slave with which to initiate communication; in fact, only one Slave at a time must be active. This connection is only possible if the Slave peripheral supports the option of having the MISO line of the three-state or floating type (high impedance). Let's now analyze how the transmission of a byte takes place. Before starting communication, the Master activates the SS line relative to the Slave with which it wants to communicate and subsequently provides the clock at the frequency at which the transmission will occur. After the Slave is activated, the bits inside the Master's shift register are shifted out (MOSI line) starting from the most significant bit (MSB). The shifted bit enters the Slave's register, which in turn begins to empty its own register by sending the most significant bit through the MISO line. The communication ends when the eighth bit is transmitted.

From what has been said, it can be understood that if the Master wants to read from the Slave, it must still send fake data to the Slave. Similarly, if the Master only wants to set the Slave or send only Data, it will still receive fake data from the Slave, unless the MISO line is ignored and not connected.

The SPI interface can be set to transmit or receive in four different modes. The selected mode must be the same for both the Master and the Slave. The SPI

interface can be set to transmit or receive in four different modes. The selected mode must be the same for both the Master and the Slave.

The four modes are usually set by means of two parameters (often implemented with two bits), named CPOL (Clock Polarity) and CPHA (Clock Phase). The polarity simply consists of having or not having a NOT gate on the Clock line, i.e., when CPOL = 0, the clock is "normal", while when CPOL = 1, the clock is inverted. This means that all operations that occur on the rising edge, if CPOL is set to 1, will occur on the falling edge. Similarly, operations that normally occur on the falling edge, setting CPOL = 1, will occur on the rising edge.

The CPHA parameter allows setting the sampling phase, i.e., when the data must be read (sampled).

Modalità	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Figure 1.3: Correspondence between modes and CPOL, CPHA parameters.

As for the transmission frequency, there is no fixed value set by the protocol. SPI interface applications use communications with frequencies ranging from a few tens of KHz up to tens of MHz (80MHz-100MHz). The maximum limit depends on the peripherals being used. For example, many microcontrollers support maximum frequencies up to 10MHz. This speed is more than sufficient for the case at hand.

1.3.2 802.11

In computer networking and telecommunications, IEEE 802.11 defines a set of transmission standards for WLAN networks, developed by the IEEE 802 group 11 in various releases, with a particular focus on the physical and MAC layers of the ISO/OSI model, which are the first two layers in the following figure.

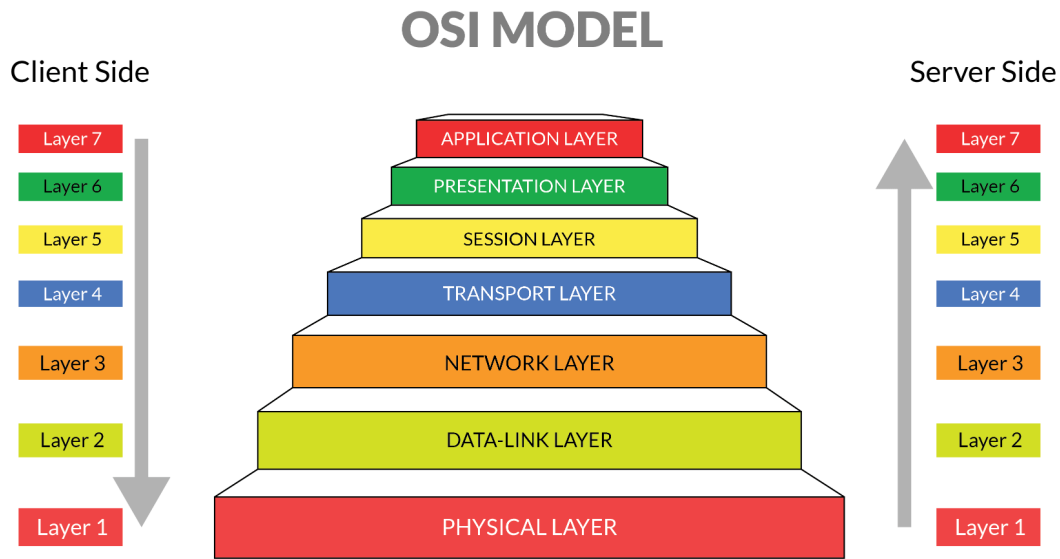


Figure 1.4: ISO/OSI model

By specifying both the interface between the client and the base station (or access point) and the specifications between wireless clients, the term "802.11 legacy" should be preferred to define the first series of 802.11 equipment. The 802.11 family consists of four protocols dedicated to information transmission (a, b, g, n), with security included in a separate standard, 802.11i. Other standards in the family (c, d, e, f, h, ...) concern extensions of basic services and improvements to services already available. The first widely spread protocol was b, followed by the a protocol, and especially the g protocol.

802.11b and 802.11g use the frequency spectrum of 2.4 GHz (ISM band). This is a frequency band regularly assigned by the national (and international) distribution plan to other services, and left free for use only for applications that require EIRP (Equivalent Isotropic Radiated Power, i.e. the maximum power radiated by an isotropic antenna) levels of no more than 20 dBm and used within private property (no crossing public ground). Working in frequency bands where other devices are already operating, b and g devices can be affected by cordless phones, audio/video repeaters for distributing satellite TV programs or other devices in an apartment that use that frequency band.

802.11a uses the 5.4 GHz ISM band. However, it does not comply with the European ETSI EN 301 893[2] standard, which requires DFS (Dynamic Frequency Selection), TPC (Transmit Power Control), and meteorological radars; this European harmonization standard is valid in Italy at the request of the Ministry of Communications with the ministerial decree of January 10, 2005.

To overcome the problem in Europe, the 802.11h protocol was introduced in

2004, which meets the required standards. Therefore, a WiFi device transmitting on public ground in Italy at 5.4 GHz must use this standard.

Some useful definitions to understand the technical documentation:

1. BSS (basic service set): a set of nodes that use the same channel access function (for example, a group of computers connected wirelessly to an access point).
2. ESS (extended service set): multiple BSSs interconnected at the MAC layer, for example the set of wireless networks in a public building such as a faculty.

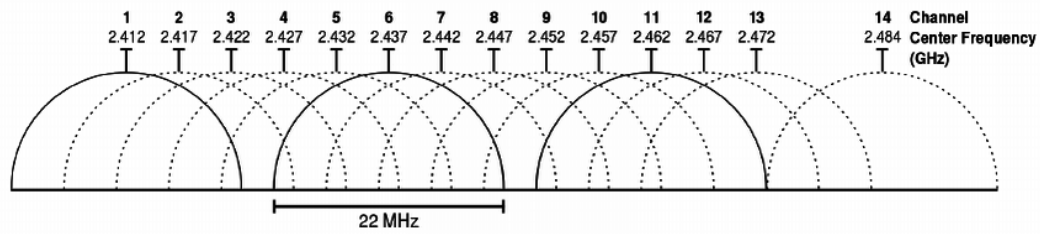


Figure 1.5: IEEE 802.11n channels in the 2.4 GHz band

For channels and bandwidth, they are defined by the standard itself. 802.11 networks at 2.4 GHz divide the spectrum into 14 sub-channels with a width of 20 MHz or 40 MHz (only for the n protocol) each, while for the 802.11 legacy and 802.11b protocols. On the other hand, 802.11 networks at 5 GHz divide the spectrum into 30 subchannels, each with a width of 20 MHz up to a maximum of 160 MHz. For the 802.11n protocol, the maximum bandwidth is 40 MHz. The bandwidth used allows for defining two limits, the transfer speed, and compatibility with nearby devices. While transmission occurs in a specific frequency range, it also has a range above and below it where the signal is transmitted, although with a significantly lower power, it can reduce the effectiveness of other repeaters.

The channels of the 2.4 GHz band partially overlap each other in frequency, so there is strong interference between two consecutive channels. To check which channel has less traffic and, therefore, less interference, there are applications available for Wi-Fi communication spectrum analysis.

The 3 groups of channels, 1, 6, 11 and 2, 7, 12, and 3, 8, 13, are combinations of channels that do not overlap with each other in case of 22 MHz bands and are used in environments with other wireless networks. After this introduction, an examination of the frame format of the Wi-Fi packet will be conducted. First of all, there are three types of frames:

1. Data frame, used for data transmission;

2. Control frame, used to control access to the vehicle (RTS/CTS, ACK);
3. Management frames, which are sent as data plots, but serve to exchange management information and are not passed to higher levels;

Each frame at the physical level consists of 4 basic elements:

1. Preamble;
2. PLCP Header;
3. MAC data;
4. CRC;

The preamble contains two fields: the first is the synchronization field, consisting of 80 bits, which is used to adapt the receiver to the exact transmission frequency, and the second is the Start Frame Delimiter used to synchronize the frame.

The PLCP header consists of three separate fields: the PSDU Length Word (PLW), the PLCP Signaling Field (PSF), and the PLCP Header Error Check Field (HEC). The PLW represents the number of octets present in the packet and serves at the physical layer to determine the end of the transmission correctly. The PSF contains information about the bit-rate used, while the HEC is a field that serves to check for the absence of errors within the header itself and is realized using a cyclic code.

The CRC allows the detection of errors in the entire packet itself. MAC data is nothing but the MAC frame passed to the higher layer in the architecture. Its maximum length is 2304 bytes and is formed by three main components:

1. MAC header;
2. Data;
3. Frame Check Sequence (FCS), contiene un CRC (Cycle Redundancy Check) di 32 bit per controllare la presenza di errori;

As described in the next picture:

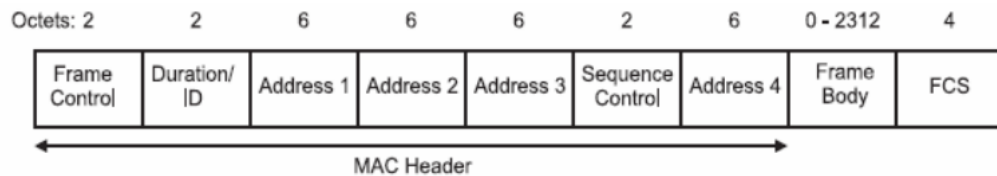


Figure 1.6: MAC data format

In the frame control, as shown in the following photo, there are various control fields.

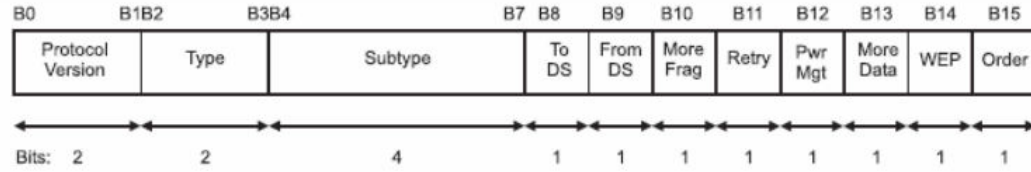


Figure 1.7: Frame control format

In the protocol version field, the protocol used is specified. It is mainly for compatibility with future evolutions and for now, both bits are set to zero. In the type and subtype fields, the type of frame (management, control, or data) and its specific function are specified. The To Ds field is set to one when the frame needs to be addressed to the AP, which will redistribute it through the Distribution System, while the From DS bit indicates that the frame comes from the DS. The more fragments bit will be set to 1 if the frame is part of a set of fragments and is not the last in the sequence. The Power Management field is used by stations that use power save mode, as is the More Data field, which tells the station that there are packets stored in the AP. Finally, the WEP field indicates the use of the encryption algorithm, while the Order field specifies whether the frame uses the StrictlyOrdered service.

The duration field has a dual function depending on the type of frame: it is used in the Virtual Carrier Sensing mechanism to specify the duration of subsequent transmission operations and thus update the NAV, or in Poll operations of individual stations.

Each MAC data format can contain up to four Address fields, and their meaning is dependent on the values of the To and From DS fields seen earlier. The addresses used are 48 bits and comply with the IEEE 802 standard. The first two are always respectively the address of the recipient and the source of the message within the same BSS. Then there is the Frame Body, which would be the payload of the packet. Typically, it starts with the upper layer ISO/OSI frame format, namely TCP/IP and subsequent encapsulations up to the application layer.

This is a brief overview of the protocol, as this thesis work has analyzed it in much more detail. However, it is not described here as it would be too long and would go beyond the description of the thesis work. It is a very long and complex protocol with many variations compared to different versions.

1.4 Hardware Analysis

In this section, the hardware on which the previously described modules will be executed will be analyzed.

1.4.1 ESP32-WROVER-E

ESP32-WROVER-E is a generic Wi-Fi + Bluetooth + Bluetooth LE MCU modules that target a wide variety of applications, ranging from low-power sensor networks to the most demanding tasks, such as voice encoding, music streaming and MP3 decoding. ESP32-WROVER-E comes with a PCB antenna. In the case at hand, the available flash memory is 8 MB. Moreover, there is also a PSRAM⁹ of 8 MB. Such memory, during the design phase, was considered more than sufficient for the firmware that needs to be flashed inside the module. For what concern computational power, there are two CPU cores that can be individually controlled, and the CPU clock frequency is adjustable from 80 MHz to 240 MHz. In detail, the chip integrated (ESP32-D0WD-V3) contains two ESP32-D0WD-V3 (ESP32-D0WDR2-V3) contains two low-power Xtensa® 32-bit LX6 microprocessors. The internal memory includes:

1. 448 KB of ROM for booting and core functions;
2. 520 KB of on-chip SRAM for data and instructions;
3. 8 KB of SRAM in RTC, which is called RTC FAST Memory and can be used for data storage; it is accessed by the main CPU during RTC Boot from the Deep-sleep mode;
4. 8 KB of SRAM in RTC, which is called RTC SLOW Memory and can be accessed by the co-processor during the Deep-sleep mode;
5. 1 Kbit of eFuse: 256 bits are used for the system (MAC address and chip configuration) and the remaining 768 bits are reserved for customer applications, including flash-encryption and chip-ID;

The modules uses a 40-MHz crystal oscillator and the operating Vdd is 3.3V. Moreover ESP32 integrates a rich set of peripherals, ranging from capacitive touch sensors, Hall sensors, SD card interface, Ethernet, high-speed SPI, UART, I2S and I2C. In the following image, the main features of the hardware are summarized.

⁹Psuedostatic DRAM

Categories	Items	Specifications
Certification	RF certification	See certificates for ESP32-WROVER-E and ESP32-WROVER-IE
Test	Reliability	HTOL/HTSL/uHAST/TCT/ESD
Wi-Fi	Protocols	802.11 b/g/n (802.11n up to 150 Mbps) A-MPDU and A-MSDU aggregation and 0.4 μ s guard interval support
	Frequency range	2412 ~ 2484 MHz
Bluetooth	Protocols	Bluetooth v4.2 BR/EDR and Bluetooth LE specification
	Radio	NZIF receiver with -97 dBm sensitivity
		Class-1, class-2 and class-3 transmitter
		AFH
	Audio	CVSD and SBC
Hardware	Module interfaces	SD card, UART, SPI, SDIO, I2C, LED PWM, Motor PWM, I2S, IR, pulse counter, GPIO, capacitive touch sensor, ADC, DAC, Two-Wire Automotive Interface (TWAI [®]), compatible with ISO11898-1 (CAN Specification 2.0)
	On-chip sensor	Hall sensor
	Integrated crystal	40 MHz crystal
	Integrated SPI flash	See Table 1 and Table 2
	Integrated PSRAM	See Table 1 and Table 2
	Operating voltage/Power supply	3.0 V ~ 3.6 V
	Minimum current delivered by power supply	500 mA
	Package size	(18.00 \pm 0.15) mm \times (31.40 \pm 0.15) mm \times (3.30 \pm 0.15) mm
	Moisture sensitivity level (MSL)	Level 3

Figure 1.8: Project specifications

In particular, the peripherals that will be examined and used by this project are the Wi-Fi and SPI peripherals. in Fig.1.8 First of all, this module support 802.11 b/g/n Wi-Fi MAC Protocol¹⁰, con 802.11 n that supports up to 150 Mbps. Additionally, it supports the sending and receiving of A-MPDU frames and the receiving of A-MSDU¹¹, Immediate Block ACK, defragmentation, Automatic Beacon monitoring. Mainly, the Wi-Fi peripheral consists of a Radio Module which in turn is composed of:

¹⁰Every letter represents a different throughput and network frequency.

¹¹The 802.11n amendment addresses new enhancements to the MAC sublayer of the Data-Link layer to increase throughput and improve power management. Frame aggregation is a method of combining multiple frames into a single frame transmission.

1. 2.4 GHz receiver;
2. 2.4 GHz transmitter;
3. bias and regulators;
4. balun and transmit-receive switch;
5. clock generator;

The 2.4 GHz receiver demodulates the 2.4 GHz RF signal to quadrature baseband signals and converts them to the digital domain with two high-resolution, high-speed ADCs. To adapt to varying signal channel conditions, RF filters, Automatic Gain Control (AGC), DC offset cancelation circuits and baseband filters are integrated in the chip. The 2.4 GHz transmitter modulates the quadrature baseband signals to the 2.4 GHz RF signal, and drives the antenna with a high-powered Complementary Metal Oxide Semiconductor (CMOS) power amplifier. The use of digital calibration further improves the linearity of the power amplifier, enabling state-of-the-art performance in delivering up to +20.5 dBm of power for an 802.11b transmission and +18 dBm for an 802.11n transmission. The clock generator produces quadrature clock signals of 2.4 GHz for both the receiver and the transmitter. All components of the clock generator are integrated into the chip, including all inductors, varactors, filters, regulators and dividers. This configuration results as the following;

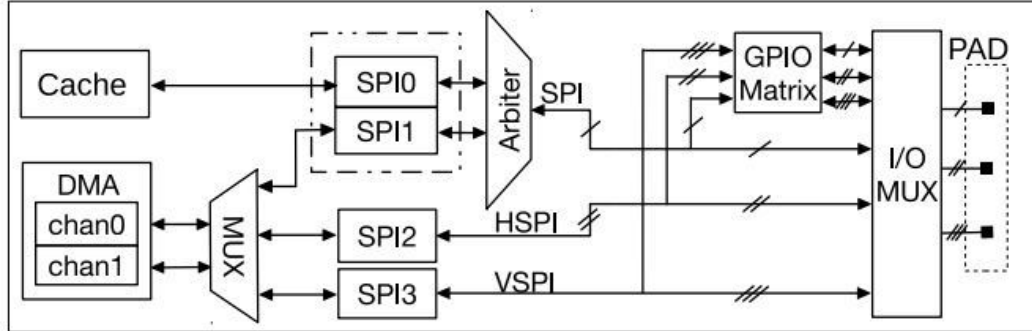


Figure 1.10: Architettura periferica SPI

¹²As for SPI, however, as can be seen in the photo1.10ESP32 integrates four SPI controllers which can be used to communicate with external devices that use the SPI protocol. Controller SPI0 is used as a buffer for accessing external memory. Controller SPI1 can be used as a master. Controllers SPI2 and SPI3

¹²Note that all terms related to SPI will be covered in the dedicated chapter

Parameter	Condition	Min	Typical	Max	Unit
Operating frequency range <i>note1</i>	-	2412	-	2484	MHz
Output impedance <i>note2</i>	-	-	*	-	Ω
TX power <i>note3</i>	11n, MCS7	12	13	14	dBm
	11b mode	18.5	19.5	20.5	dBm
Sensitivity	11b, 1 Mbps	-	-97	-	dBm
	11b, 11 Mbps	-	-88	-	dBm
	11g, 6 Mbps	-	-92	-	dBm
	11g, 54 Mbps	-	-75	-	dBm
	11n, HT20, MCS0	-	-92	-	dBm
	11n, HT20, MCS7	-	-72	-	dBm
	11n, HT40, MCS0	-	-89	-	dBm
	11n, HT40, MCS7	-	-69	-	dBm
Adjacent channel rejection	11g, 6 Mbps	-	27	-	dB
	11g, 54 Mbps	-	13	-	dB
	11n, HT20, MCS0	-	27	-	dB
	11n, HT20, MCS7	-	12	-	dB

Figure 1.9: Caratteristiche Wi-Fi

can be configured as either a master or a slave. When used as a master, each SPI controller can drive multiple CS signals (CS0 – CS2) to activate multiple slaves. Controllers SPI1 – SPI3 share two DMA channels. The I/O lines included in the abovementioned signal buses can be mapped to pins via either the IO-MUX module or the GPIO matrix. The SPI controller supports four-line full-duplex/half-duplex communication (MOSI, MISO, CS, and CLK lines) and three-line half-duplex-only communication (DATA, CS, and CLK lines) in GP-SPI mode. Le seguenti sono le caratteristiche più importanti

1. Programmable data transfer length, in multiples of 1 byte;
2. Four-line full-duplex/half-duplex communication and three-line half-duplex communication support;
3. Master mode and slave mode;
4. Programmable CPOL and CPHA;
5. Programmable clock;

When not using DMA, the maximum length of data received/sent in one burst is 64 bytes. The data length is in multiples of one byte. ESP32 SPI

generates two types of interrupts. One is the SPI interrupt and the other is the SPI DMA interrupt. ESP32 SPI reckons the completion of send- and/or receive-operations as the completion of one operation from the controller and generates one interrupt. When ESP32 SPI is configured to slave mode, the slave will generate read/write status registers and read/write buffer data interrupts according to different operations. Then, after the interrupt is generated, the processor understands that the transaction has taken place and can proceed to copy in a proper manner the data received saving them in appropriate structures.

In conclusion, it can be asserted that this hardware is more than sufficient for our purpose. Although there is no possibility of sending data at 5 GHz frequency over Wi-Fi network, it can be safely said that the 802.11 b/g/n configuration is optimal for our functionalities. In fact, the packets it has to send are small in size as it has to exclusively send data and information, high transfer speed is not required which could require streaming of videos or downloading of very heavy applications.

As far as the computational power offered is concerned, it is more than sufficient as the system clock is high (80 - 240 MHz).

However, in the chapter relating to data reception, it will be shown how this module does not receive all packets when monitor mode is set. It does not have any consequences that make its use impossible, but it slows it down a bit. This issue will be discussed in detail later.

1.4.2 SoM

For what concern the SoM board, this is a custom board developed by Bitron itself, so is not possible to provide lots of details. The processor is an High-Performance Arm Cortex-A7 32 bit RISC core operating system operating at up to 800 MHz. Has a Ram of 512 MB providing USB, USART, I2C, SPI, CAN and Ethernet connection.

Chapter 2

Explanation of software environment

In this chapter, we will explain in more detail the architecture of the project itself and then analyze in the next chapter the results obtained. This process was carried out entirely in a VMware virtual machine on which the image of the Ubuntu operating system was loaded. The IDE used was Visual Studio on which the plugin provided by Espressif was installed, so that the IDE is properly configured to build correctly. In fact, this plugin correctly sets the dependencies of libraries. However, the APIs offered (HAL¹) from Espressif are "opaque", so you can't see what the available functions actually do because you don't have access to source code. This aspect has been a problem in the development of the project itself as sometimes the description of the APIs on the Espressif website can be misunderstood, and in such low-level work on individual bits it is necessary to have greater knowledge of the functioning of certain functions. In particular, on the actions of receiving and sending packages. This will be discussed in more detail in the next sections. Instead, the functions of FreeRTOS, being a Free project, you have free access to the source code and so you can understand in detail how it behaves. As for the Linux side, the IDE Visual Studio is always used and setting the dependencies of the libraries to properly build that driver. I was provided by the company the SDK of Yocto's custom project which was generated by a Bitron employee. As mentioned above, it is the Linux version 5.10.61.

¹Hardware Abstraction Layer

2.1 Espressif

As for the APIs provided by Espressif, I focused mainly on analyzing the libraries related to those of the Wi-Fi driver. The Wi-Fi libraries provide support for configuring and monitoring the ESP32 Wi-Fi networking functionality. The structure is quite simple for this part. It's explained very well by the documentation provided by the Espressif on its web page where it explains the different functionalities that this microcontroller can have and the general architecture. The only drawback, as previously analyzed, is that the APIs are not open source and therefore you can not analyze the APIs operation in detail to understand exactly how it behaves on individual registers.

This includes configuration for:

1. Station Mode;
2. AP mode;
3. STA/AP mode;
4. Various security modes for the above;
5. Scanning for access point;
6. Promiscuous mode for monitoring of IEEE802.11 Wi-Fi packets;

This section provides an overview of the architecture and management of the firmware, without the actual description of the APIs to avoid redundancy with the following sections where they will be explained in greater detail.

2.1.1 Wi-Fi driver

The ESP32 Wi-Fi programming model is depicted as follows:

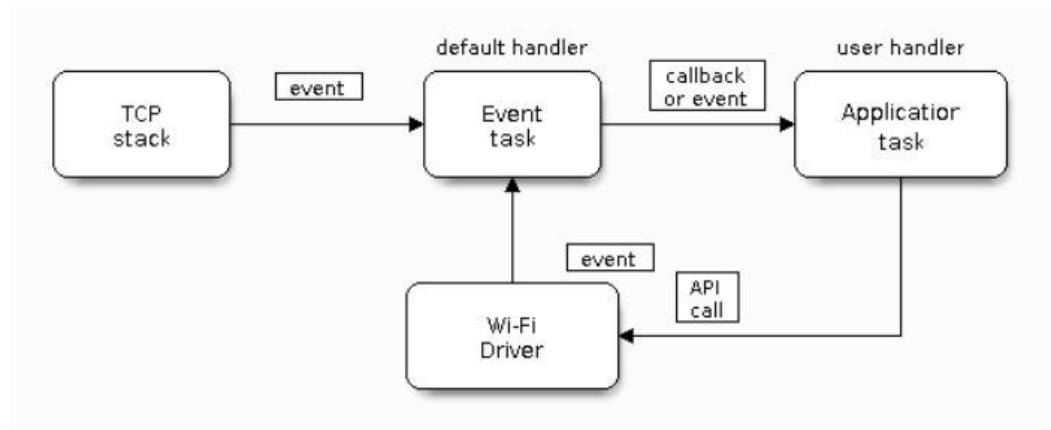


Figure 2.1: Wi-Fi programming model

This is the guideline provided by Espressif for a correct setting of the microcontroller according to the APIs provided by the manufacturer. In fact, the Wi-Fi driver is seen as a kind of black box from the upper layers that know nothing about what's going on. In particular, in our case, ideally, the TCP stack is implemented by the upper layers of Linux networking, while I have no application tasks in this case because the task of such a thesis is to send packages from the charging column to a remote center. These applications are always implemented on the Linux side, so all the management of the OCPP protocol, log etc. are managed by someone. The only task of this microcontroller is to receive and send individual 802.11 packages. So, you have to implement the Wi-Fi driver with some Event Task. You want to specify how at the beginning of the project, despite careful documentation, it was not clear the focal point as I was quite unaware of the whole 802.11 world and the features. Then, in a first phase of the project, to become familiar with the Espressif environment, as later analyzed in more detail, tests were made with the APIs offered.

2.1.2 Event-Handling

Special attention should be given to the automatic generation of events. As described in a series of Events (similar to interrupts) are offered to inform the code that some event has happened. An event in this area tends to indicate that something has happened. That is, events such as `WIFI_EVENT_SCAN_DONE` (which indicates that the network scan is finished), `WIFI_EVENT_STA_SCONNECTED` (that is, that the station has connected to an AP station). Such events are very useful as they allow to make a structured firmware to events within tasks. In addition, events for TCP_IP protocol management are also offered. As previously

stated, this ISO_OSI level must be managed by the Linux kernel, so it should not be used by the microcontroller as it is not its task. In addition, we want to emphasize that events are divided into those for AP and STA, as each configuration has its own events.

2.1.3 Wi-Fi APIs Error Code

All APIs offered by Espressif have default values of return value. They can be divided into 4 categories:

1. No error, the APIs worked correctly;
2. Remedial errors;
3. Mistakes not remediable but not critical;
4. Non remediable and critical errors;

As a good general line, to write a robust structure, it is always good to check the return value of the APIs offered, as it allows the management of unexpected things that, however, must be taken into account to ensure that the firmware is able to manage. As an extreme case, you may need to reboot the micro after an error that is not remediable and critical. If this aspect is not handled, the micro could be cracked at a certain execution point within the task and not proceed with the execution.

2.2 Linux Networking and Device Drivers

As mentioned above, the purpose is to write this Linux Device Networking Driver and in this section we will analyze in more detail the structure of this part. In the analysis of the documentation it was noted that most of the books refer to versions much more dated than the Linux kernel used. For example, Linux Device Drivers, Third Edition, is a great starting point for an analysis of the operation of the Linux Kernel but is still based on Kernel 2.6.10. The problem, as mentioned above, results in the fact that the source code of the Kernel evolves very quickly and then you may lose some nuances of its operation. That said, to have an initial understanding of the operation of the Linux operating system is more than enough. In particular, I focused on the operation of the Device Drivers at first and then in detail of the Network Drivers. Moreover, being an Open Source world, on the main line of the Kernel there are several Drivers proposed by different companies. In particular, the company has given me the code developed for the WILC6000 of Microchip. The problem arises in the fact that there are about 30000 lines of

code and also is a WNIC, so not implementing the MAC layer is different than the micro that I'm looking at. In fact, basically a special network card is used, while in this case a microcontroller is driven that in turn drives (through the firmware) the data sent and received by the antenna itself. In addition, the main difference is that with a SoftMAC network card the driver goes directly to the network card registers and manages the reception if using linked list, interrupt or polling etc. While the driver now drives another microcontroller: then the driver must manage incoming and incoming traffic by piloting a microcontroller which in turn drives an antenna. This thing can be a waste of time as there is an extra paper trail in between than doing a direct read on the network card logs when a packet arrives or leaves. But specific considerations will be made in the relevant part.

2.2.1 Linux Device Driver

Initially, as mentioned above, I focused on the analysis of the Linux Device drivers.

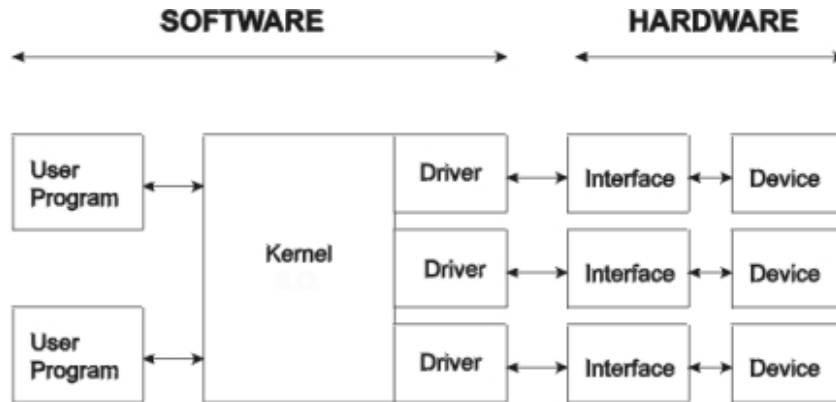


Figure 2.2: Device Linux Driver

The figure 2.2 is a simple but at the same time good representation of the Driver Architecture. As you can see, the Driver resides in the Kernel Space and not in the User Space: therefore, Any applications (that are written in the User Space) use what is in the Kernel Space. Thus, such a Driver would be the interface between the Kernel and the Hardware. The kernel searches the hardware based on what the application requires. The hardware and the kernel are distinct “black boxes” that make a particular piece of hardware respond to a well-defined internal programming interface. This interface completely hides the details of how the device works. User activities are carried out through a set of standardized calls independent of the specific driver. Mapping calls to device-specific operations that act on real hardware is therefore the role of the device driver. This programming

interface is such that drivers can be built separately from the rest of the kernel and “connected” to runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds available. The driver should take care of making the hardware available, leaving all the problems on how to use the hardware for the applications. A driver is flexible if it offers access to hardware features without adding constraints. Sometimes, however, some political decisions have to be taken. For example, a digital I/O driver can only offer byte-level hardware access to avoid the additional code needed to manage individual bits. One of the good features of Linux is its ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel (and remove functionality as well) while the system is up and running. Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program.

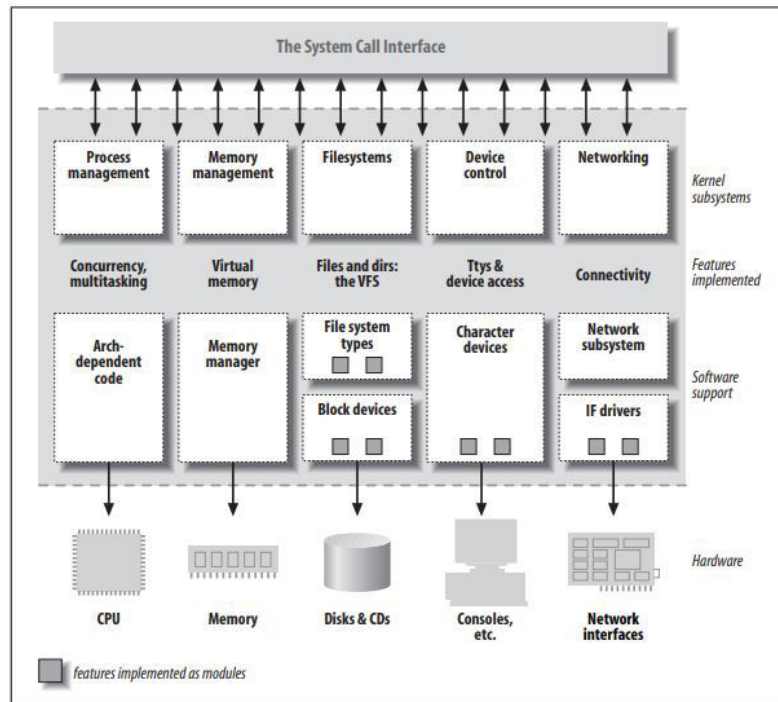


Figure 2.3: Split view of the Kernel

The Linux way of looking at devices distinguishes between three fundamental

device types:

1. Character devices: a character device is one that can be accessed as a stream of bytes (like a file), and a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (`/dev/console`) and the serial ports (`/dev/ttyS0` and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as `/dev/tty1` and `/dev/lp0`. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. Nonetheless, there exist char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using `mmap` or `lseek`.
2. Block devices: A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.
3. Network interfaces: Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are usually designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets. Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as `/dev/tty1` is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as `eth0`), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely

different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

So, my task is to write a network interfaces, to allow different applications to send a package over Wi-Fi to run it to the kernel and the kernel 'directs' it to my driver. It takes care of the different packages it receives from the Kernel and adopts an optimal policy to manage these packages and run them to the ESP32, which will take charge of those packages and in turn through a package management will run them over Wi-Fi to the device in charge. Mirror, once the ESP32 receives a package, manages it, runs it via SPI to my driver, which in turn processes it and informs the kernel of the received package and runs it 'at higher ISOsystem levels' so that the applicative requesting it gets what it requested. Additionally, Network drivers also have to be prepared to support a number of administrative tasks, such as setting addresses, modifying transmission parameters, and maintaining traffic and error statistics. The APIs for network drivers reflects this need and, therefore, looks somewhat different from the interfaces we have seen so far. The APIs offered by Linux for management will be analyzed directly during the explanation of the code, this is because they are multiple and extremely customizable.

2.3 FreeRTOS

FreeRTOS is ideally suited to deeply embedded real-time applications that use microcontrollers or small microprocessors. This type of application normally includes a mix of both hard and soft real-time requirements. Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless. For example, responding to keystrokes too slowly might make a system seem annoyingly unresponsive without actually making it unusable. Hard real-time requirements are those that state a time deadline—and breaching the deadline would result in absolute failure of the system. For example, a driver's airbag has the potential to do more harm than good if it responded to crash sensor inputs too slowly. FreeRTOS is a real-time kernel (or real-time scheduler) on top of which embedded applications can be built to meet their hard real-time requirements. It allows applications to be organized as a collection of independent threads of execution. On a processor that has only one core, only a single thread can be executing at any one time. The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer. In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements. This would ensure that hard real-time threads are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic. Why Use a Real-time Kernel?

There are many well established techniques for writing good embedded software without the use of a kernel, and, if the system being developed is simple, then these techniques might provide the most appropriate solution. In more complex cases, it is likely that using a kernel would be preferable, but where the crossover point occurs will always be subjective. As already described, task prioritization can help ensure an application meets its processing deadlines, but a kernel can bring other less obvious benefits, too. Some of these are listed very briefly below.

1. Abstracting away timing information: The kernel is responsible for execution timing and provides a time-related APIs to the application. This allows the structure of the application code to be simpler, and the overall code size to be smaller.
2. Maintainability/Extensibility: Abstracting away timing details results in fewer interdependencies between modules, and allows the software to evolve in a controlled and predictable way. Also, the kernel is responsible for timing, so application performance is less susceptible to changes in the underlying hardware.
3. Modularity: Tasks are independent modules, each of which should have a well-defined purpose.
4. Improved efficiency: Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done. Counter to the efficiency saving is the need to process the RTOS tick interrupt, and to switch execution from one task to another. However, applications that don't make use of an RTOS normally include some form of tick interrupt anyway.
5. Idle time: The Idle task is created automatically when the scheduler is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.
6. Power Management: The efficiency gains that are obtained by using an RTOS allow the processor to spend more time in a low power mode. Power consumption can be decreased significantly by placing the processor into a low power state each time the Idle task runs. FreeRTOS also has a special tick-less mode. Using the tick-less mode allows the processor to enter a lower power mode than would otherwise be possible, and remain in the low power mode for longer.

The ones listed above are the main reasons to use a FreeRTOS, and it fits perfectly for our application under consideration, especially as I already have previous experience in writing using this methodology. In addition, the main features are the following:

1. Pre-emptive or co-operative operation;
2. Very flexible task priority assignment;
3. Flexible, fast and light weight task notification mechanism;
4. Queues;
5. Binary semaphores;
6. Counting semaphores;
7. Mutexes;
8. Stack overflow checking;
9. Software timers;
10. Event groups;

Chapter 3

Code Implementation

In this chapter there is the description of the project itself. The first thing I did, after documenting several protocols, I started to test the project's foundation, namely how to properly build the LKM to be loaded on the linux kernel and firmware.

3.1 Software Architecture

Before see a detailed explanation of the code, is important to give an overview and how this environment is thought. Firstly, the ESP32 has just the role to configure itself as the SoM tell it to do. Secondly, it has to manage the packet received from the net and the one to send from SoM to the net. Through the lecture of online documentation, i see that a network interface can work with two different behaviour:

1. interrupt mode;
2. polling mode;

This choice is the most important one for the design of both firmware and driver because it is determinant in the overall behaviour. The first, and most simple design, is simply handled by interrupt. Each time that a packet has to be send between SoM and ESP32, who has to send raise an interrupt in order to tell the other board that a stream of data is ready. While the second one is handled by Linux from the newer version of the Kernel itself and is quite different from the interrupt mode. Indeed, the here the packet are not send immediately but are accumulated and a transaction between the board are made in periodic time. So, a linked list is needed and a FIFO hardware in order to store the data to be send or start a long transaction. Despite the fact that the polling mode is more efficient

how is developed in the newer version of Linux Kernel, I used the interrupt mode for the following reasons:

1. In the SoM that I use, are already present driver that use SPI interface and the employer of Bitron suggest me to not focus too much in the speed and efficiency of the entire work but try too understand which are the main aspect of Linux driver and ESP32 firmware;
2. Usually, the fundamental usage of the ESP32 is to exchange data and take data from the OCPP in an aperiodic way and for light package. Moreover, seen the fact that the Linux version built for this SoM must be the lightly possible, make something that weighs more on the processor is not the correct idea. Please, take in account that this SoM is responsible to handle different process and service and colleague told me that the main idea is to make something that does not weigh too much in terms of speed of execution and optimization;

So, after this choice was made I started to think how can be a suitable solution to handle this problem. In the next subsection are explained from an higher point of view how I thought to implement in the most suitable manner the task.

3.1.1 ESP32

For what concern the firmware, the ESP32 provide the initial configuration of the project (such as the Makefile and so on) while, of course, the source files are provided by me:

1. `main.c`: contains just the map partition table, function that initialise the pin configuration, creation of a global queue in order to handle the exchange of data between task in a more safe and better way than just the usage of global variable and creation of the two main freeRTOS task that handle the transmission of packet;
2. `spi.c`: contain the pin configuration of what concern the communication between SoM and ESP32 and the function that are necessary to a correct exchange of data;
3. `connect.c`: there is defined the function to handle the exchange of data with the net, the handle of Wi-Fi event and interrupt;
4. `include.h`: it is the header file that contains the declarations of function and extern variable;
5. `CMakeLists.txt`: it tells which are the source file to build together;

In the following are only reported the key part of the code. Is superfluous to explained in detail every single aspect, some things are of course routine stuff.

As seen from the above considerations, is necessary to work with interrupt in reception of packet from SoM and from the net. In the above sections is described how is implemented this feature.

3.1.2 Linux Driver

For what concern the Linux driver, the two source code are the following one:

1. `cfg80211esp.c`: this is the 'main' source code, where the driver is initialized, are initialized the GPIO and more over are created the struct `work_struct`. This are created to let the Linux scheduler to create workqueue where are scheduled the function associated which the corresponding `work_struct`. Moreover, there is the initial configuration of the network interface, as explained in major detail later;
2. `espspi.c`: here there are the initialization of the spi peripheral and moreover the function to handle the packet to send and receive;
3. `espspi.h`: this is the header file, where there are the declaration of function and variable;

3.2 How to build

In order to make a complete description of the project done, in this section are examined how to build and compile the source code.

3.2.1 Linux Driver

For what concern the Linux build of the module, other then the `.c` e `.h` source code, are necessary the Kbuild e Makefile.

The following is the content of the Kbuild file:

```
1 obj-m := esp32_bb.o
2 esp32_bb-objs:= cfg80211esp.o espspi.o netdevice.o
```

This tells the compiler to compile the three source code `cfg80211esp.c`, `espspi.c` and `netdevice.c` into the object file `esp32_bb.o`. Moreover, is necessary to create a Makefile, which the following is the content:

```

1 # Path to the directory that contains the Linux kernel source code
2 # and the configuration file (.config)
3 KERNEL_DIR ?= $(KERNEL_SRC_PATH)
4
5 # Path to the directory that contains the generated objects
6 DESTDIR ?= $(KERNEL_DIR)/install_artifact
7
8 # Path to the directory that contains the source file(s) to compile
9 PWD := $(shell pwd)
10
11 default:
12     $(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules
13
14 install:
15     $(MAKE) -C $(KERNEL_DIR) M=$(PWD) INSTALL_MOD_PATH=$(DESTDIR)
16     modules_install
17
18 clean:
19     $(MAKE) -C $(KERNEL_DIR) M=$(PWD) clean

```

So, to compile correctly, is necessary to write in the command line of linux make clean, make and then make install. Then, is possible to copy the .ko obtained by the compilation with the command scp, that copy the file into the SoM. Then, with the command modprobe the .ko is build against the Kernel. Please, take in account the fact that the enter point of the module when is loaded is the part of the source code that start with the function

```

1 static int __init virtual_wifi_init(void) {
2
3     // code
4
5 }
6

```

To remove the module, is necessary to write on the command line of the SoM rmmod, that unitilizza and remove the gpio, controller and everything that is initilized. Its looks like that:

```

1 static void __exit virtual_wifi_exit(void) {
2
3     //code
4
5 }
6

```

3.2.2 ESP32

While for what concern the firmware, the project the environment to build is created by the plug-in provided by Espressif and the main file is the sdkconfig and the CMakeLists.txt that provide a list of information to let the compiler compiles correctly the source code created by me.

3.3 SPI

As already said before, the Linux driver act as Master while the ESP32 acts as slave. To configure it properly, we have to check which pin can be used by ESP32 and SoM to use it as CS, MISO, MOSI and SCLK. For what concern the ESP32, I have choosen the pin 12 for the MOSI, 13 for the MISO, 15 for the SCLK while for the CS can be used every available GPIO I-O, because for the slave it is just important that when CS is low it activates the sending/receiving of the data packet. I have set all this four pin in a pull-up configuration. Moreover, I have adopted the MODE 1 for the SPI, as suggested in the previous chapters. Moreover, I have choose to use the controller SPI3 of the Espressif, because how you can see in the image 1.10 the SPI3 controller can have access to DMA. So, how you can see in the next lines of code, I have to populate the two struct `spi_bus_config_t` and `spi_slave_interface_config_t` and then register them using the API `spi_slave_initialize`.

```
1 void spi_configuration()
2 {
3     spi_bus_config_t buscfg={
4         .mosi_io_num = GPIO_MOSI,
5         .miso_io_num = GPIO_MISO,
6         .sclk_io_num = GPIO_SCLK,
7         .quadwp_io_num = -1,
8         .quadhd_io_num = -1,
9     };
10
11     spi_slave_interface_config_t slvcfg={
12         .mode=1,
13         .spics_io_num=GPIO_CS,
14         .queue_size=1,
15         .flags=0,
16     };
17
18     gpio_set_pull_mode(GPIO_MOSI, GPIO_PULLUP_ONLY);
19     gpio_set_pull_mode(GPIO_SCLK, GPIO_PULLUP_ONLY);
20     gpio_set_pull_mode(GPIO_CS, GPIO_PULLUP_ONLY);
21 }
```

```

22     ESP_ERROR_CHECK(spi_slave_initialize(RCV_HOST, &buscfg, &slvcfg,
23     SPI_DMA_CH_AUTO));
    }

```

Moreover, I have added one line, called GPIO-HANDSHAKE (added to the GPIO number 2) in order to inform the SoM that the ESP32 is ready to accept packet and command. So, the logic implemented is the following one: when the ESP is ready to accept data and command from the SoM, it set the value of that pin to 1, otherwise it is set to 0.

```

1     gpio_set_direction(GPIO_HANDSHAKE, GPIO_MODE_OUTPUT);
2     gpio_set_pull_mode(GPIO_HANDSHAKE, GPIO_PULLUP_ONLY);
3     gpio_set_level(GPIO_HANDSHAKE, 0);

```

For what concerns the Linux implementation, looking at the APIs provided I see that I have to populate the struct `spi_board_info` where I provide the configuration adopted (MODE 0, max speed of the SPI) and the information of the slave device.

```

1 static struct spi_board_info etx_spi_device_info =
2 {
3     .modalias      = "esp32_driver",
4     .max_speed_hz  = 40000000,
5     .bus_num       = SPI_BUS_NUM,
6     .chip_select   = 0,
7     .mode          = SPI_MODE_1
8 };

```

Then, I have to associate the above struct to the controller of the SPI, the pointer to the struct `spi_master` and in order to do this is used the function `spi_busnum_to_master`. Then, after that, I have to create this device given the master and device info using the API `spi_new_device`. After all, I have to setup the SPI configuration, using the function `spi_setup` it tells that I have 8 as bits per word. Moreover, I have to set the HANDSHAKE pin as an Input pin.

```

1 int etx_spi_init(void)
2 {
3     int ret;
4     struct spi_master *master;
5
6     handshake(HANDSHAKE_NUM);
7
8     master = spi_busnum_to_master(etx_spi_device_info.bus_num);
9     if( master == NULL )

```



```

10 {
11     pr_err("SPI Master not found.\n");
12     return -ENODEV;
13 }
14
15 // create a new slave device, given the master and device info
16 etx_spi_device = spi_new_device( master, &etx_spi_device_info );
17 if( etx_spi_device == NULL )
18 {
19     pr_err("FAILED to create slave.\n");
20     return -ENODEV;
21 }
22
23 // 8-bits in a word
24 //etx_spi_device->bits_per_word = 8;
25 // setup the SPI slave device
26 ret = spi_setup( etx_spi_device );
27 if( ret )
28 {
29     pr_err("FAILED to setup slave.\n");
30     spi_unregister_device( etx_spi_device );
31     return -ENODEV;
32 }
33
34 pr_info("SPI driver Registered\n");
35 return 0;
36 }

```

Please take into account the fact that the number of the PIN to associate to the CS, MISO, MOSI and CLOCK are already set by the layout design of the SoM. My only role is to wire the correct PIN to the one already set.

In order to test this implementation, I have created (on the firmware) this simple function that return the value received by the SoM.

```

1 spi_slave_transaction_t t;
2
3 char my_receive_cmd( void ){
4     char cmdd;
5     char * ricevuto;
6     ricevuto = (char *) malloc(4);
7     t.rx_buffer=ricevuto;
8     t.tx_buffer=NULL;
9     t.length=8*sizeof(ricevuto);
10    gpio_set_level(GPIO_HANDSHAKE, 1);
11    ESP_ERROR_CHECK(spi_slave_transmit(RCV_HOST, &t, portMAX_DELAY));
12    gpio_set_level(GPIO_HANDSHAKE, 0);
13    cmdd = *ricevuto;

```

```

14 |     return cmd;
15 | }

```

The core of this function is the API `spi_slave_transmit`, that using the struct `t` (that represent the single transaction) send the buffer associated to `rx_buffer` and send the one to `tx_buffer`. Both `rx_buffer` and `tx_buffer` are the pointer to the data to be sent. While, on the Linux side, I have implemented this simple function that send the comand that is passed to the function itself.

```

1 | int etx_spi_write_cmd(cmd_spi cmd_){
2 |
3 |     char tx[4] = {(char)cmd_,0,0,0};
4 |     struct spi_transfer tr =
5 |     {
6 |         .tx_buf = tx,
7 |         .rx_buf = NULL,
8 |         .len = sizeof(tx),
9 |         .bits_per_word = 8,
10 |    };
11 |
12 |    while(!gpio_get_value(HANDSHAKE_NUM));
13 |    spi_sync_transfer( etx_spi_device, &tr, 1 );
14 |    return 0;
15 | }

```

So, this function is pretty similar to the one implemented on ESP but the fact that here Liunx check the value of the HANDSHAKE and if it is different from 0 it can start the transaction.

Moreover, the HANDSHAKE is set to one by the ESP32 and when the SoM has something to send it can send it only when the line is high. So, this is the basic implementation of the SPI on both device. Now, on the ESP32 side there is just the main function that initialize the flash partition by default, the pin and SPI configuration and just a task that call the function to read the data incoming. While, on the Linux side, there is the sending of a command (that is just an integer) in order to see if everything work properly. So, I load and unload the Linux module and every time the transaction work properly. After that, it's time to create something of more robust.

In order to configure properly the ESP32 according to the necessity of the EV charger, is necessary that the SoM communicate to ESP32 how to works. For example, if it has to be a STA, AP or both. Moreover, the SoM can ask information about the MAC layer to the ESP, as for example the MAC address, at which network is connected, and so. In parallel, there are the packet that Kernel has to send over the Wi-Fi that the SoM send over SPI to the ESP32. So, here two ways can be adopted to achieve this behaviour. Firstly, I can adopt other two

wire between SoM and ESP32 in order to inform the ESP32 that the SoM has to send something when the HANDSHAKE provided by ESP is equal to one. One line is to inform that SoM has to send a command and another one to tells that a packet has to be sent. So, just rising to one the signal the ESP32 handles it as an hardware interrupt on the rising edge. After that, the SoM can send its packet over the SPI if the HANDSHAKE is equal to one. This implementation can be optimal because according to which interrupt is generated, the ESP32 will handle the packet received with two different callbacks.

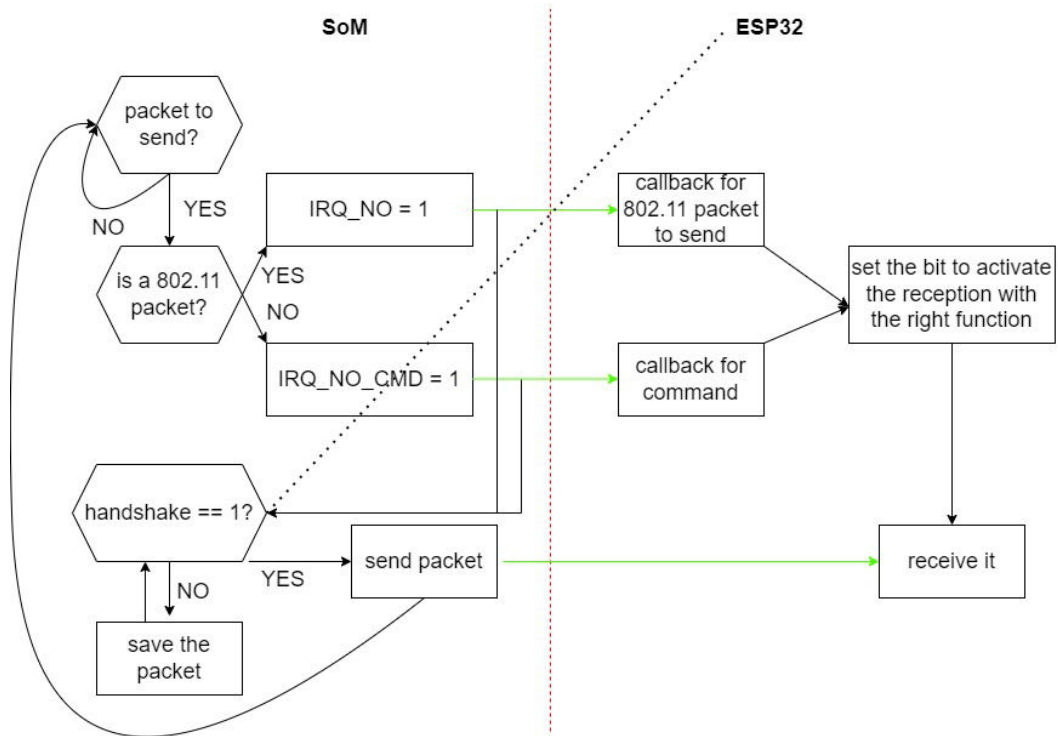


Figure 3.1: Block diagram of the above implementation

As you can see it the Figure 3.1, the line of the interrupt to inform the ESP32 that a 802.11 packet is ready to be sent is called `IRQ_NO`, while the one for the command `IRQ_NO_CMD`. This implementation can be achieved with the following code implementation (ESP32 side).

```

1  gpio_install_isr_service(5);
2  gpio_set_direction(GPIO_IRQ_INPUT, GPIO_MODE_INPUT);
3  gpio_set_intr_type(GPIO_IRQ_INPUT, GPIO_INTR_POSEDGE);

```

```

4   gpio_isr_handler_add(GPIO_IRQ_INPUT, isr_handler_packet_coming,
   NULL);
5
6   gpio_set_direction(GPIO_IRQ_INPUT_CMD, GPIO_MODE_INPUT);
7   gpio_set_intr_type(GPIO_IRQ_INPUT_CMD, GPIO_INTR_POSEDGE);
8   gpio_isr_handler_add(GPIO_IRQ_INPUT_CMD, isr_handler_cmd_coming,
   NULL);

```

First of all, is necessary to register the interrupt and the respective callback on the rising edge for the two different lines.

```

1 void isr_handler_cmd_coming(void *arg){
2     xEventGroupSetBits(evtGrp, cmd_to_receive);
3 }
4
5 void isr_handler_packet_coming(void* arg){
6     xEventGroupSetBits(evtGrp, packet_to_send);
7 }

```

The callback, that for definition has to be light function, just set a bit (cmd_to_receive or packet_to_send) into a mask (evtGrp). Then, I have created a task with the following code.

```

1 void spiTask_pkt_sender(void*params){
2
3     //inizialization of variable
4
5     while(1){
6
7         uxBits = xEventGroupWaitBits(evtGrp, packet_to_send |
   cmd_to_receive | //other stuff, false, false, portMAX_DELAY);
8
9         if(uxBits & cmd_to_receive){
10
11             cmd = my_receive_cmd();
12             printf("COMANDO RICEVUTO: %d\n", cmd);
13
14             switch(cmd){
15
16                 case READ_MAC:
17                     send_mac();
18                     break;
19
20                 case MAKE_SCAN:
21                     wifi_connect_sta();
22                     break;

```

```

23
24         // other cases
25
26     }
27
28     xEventGroupClearBits(evtGrp, cmd_to_receive);
29
30     }else if(uxBits & packet_to_send){
31
32         rx_packet(); //function that send the packet over Wi-Fi
33         xEventGroupClearBits(evtGrp, packet_to_send);
34
35     }
36
37     //other stuff
38
39 }
40 }

```

The logic is the following: this task is blocked on the function `xEventGroupWaitBits` thanks to the `portMAX_DELAY` (this is a `freeRTOS` function) until one of the bits of the `evtGrp` is set. Once is set, the code can go on and then based on which bit is set is activate the reception of the command with the function `my_receive_cmd()` that return an `ENUM` that activate other actions thanks to the switch case. Instead if is set to one the bit for the reception, the function `rx_packet()` is called that send the packet over Wi-Fi. The implementation of this function is explained later.

While on the Linux side, this implementation is adopted.

```

1     gpio_set_value(IRQ_NO_CMD, 1);
2     etx_spi_write_cmd(READ_MAC);
3     etx_spi_read32(my_mac_address);
4     gpio_set_value(IRQ_NO_CMD, 0);

```

This is an example to read the MAC address of the ESP32. It just raise the `IRQ_NO_CMD` and then it send the command (the implementation of that function is explained above). While in order to read the packet is send by the ESP32 to the SoM, and so, even if the ESP32 is the slave it informs the SoM that it has to send a packet.

For this purpose, in the following figure is explained the policy to send packet from ESP32 to SoM.

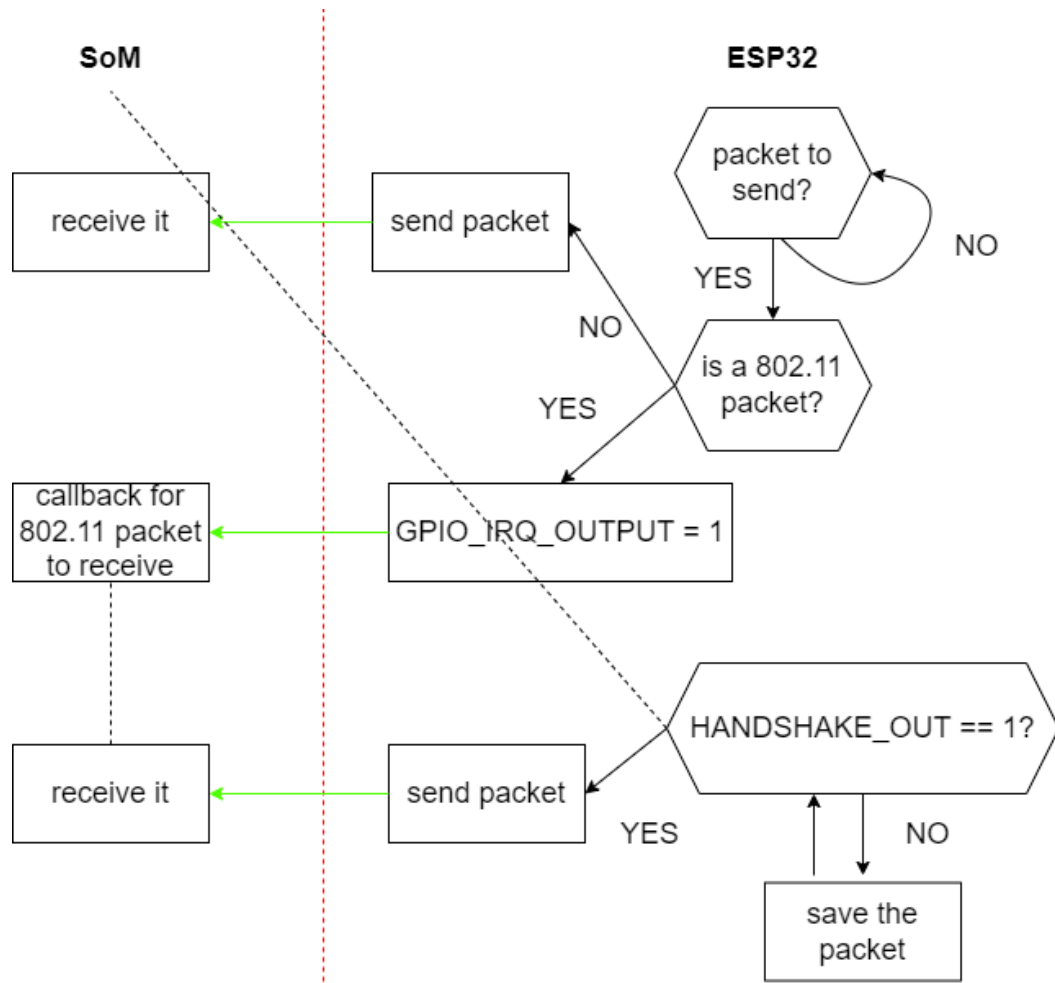


Figure 3.2: Block diagram of the policy to send packet from ESP to SoM

In the Figure 3.2 is described the logic of the implementation. Here, are adopted two more lines: the `HANSHAKE_OUT` is used to inform the ESP32 that the SoM is ready to accept a packet and moreover is used in order to sincronize the sending and receiving of packet. This because can happen that a both SoM and ESP32 has to send a packet, and thanks to this `HANSHAKE_OUT` the SoM inform the ESP32 if the wires are empty. As you can see, if the packet is not a 802.11 one, it means that is a packet send by ESP32 in response of a command sent by SoM to ESP32. So, for the implementation the SoM is of course ready to accept it.

```

1 void spiTask_pkt_received(void*params){
2

```

```

3 // variables initialization
4
5 while(1){
6
7     xQueueReceive(queue, &pacchetto, portMAX_DELAY);
8     ESP_ERROR_CHECK(gpio_set_level(GPIO_HANDSHAKE, 0));
9
10    while(gpio_get_level(HANDSHAKE_OUTPUT) == 0);
11    ESP_ERROR_CHECK(gpio_set_level(GPIO_IRQ_OUTPUT, 1));
12    send_packet_received_over_wifi(pacchetto.payload, pacchetto.
rx_ctrl.sig_len);
13
14    gpio_set_level(GPIO_IRQ_OUTPUT, 0);
15    gpio_set_level(GPIO_HANDSHAKE, 1);
16    xEventGroupClearBits(send_grp, packet_receive_by_esp);
17 }
18 }

```

The above description describes the manner to send the packet over SPI received by ESP to SoM. So, is created another task that wait that a packet is received by the ESP32 (this logic is explained in more detail in later section). When the packet is received, the code can go on and first of all it put the HANDSHAKE to 0 in order to does not receive any packet by SoM. Then, it waits that the SoM can accept any packet. When HANDSHAKE_OUTPUT is set to one by SoM, the ESP32 send the interrupt rising the line GPIO_IRQ_OUTPUT and the SoM is ready to receive the packet: then the packet is effectively sent.

Overall, the implementation of this logic to send packet over SPI from SoM to ESP32 seems working well but probably is not the most efficient one. Infact, as you can see in the following figure there is a delay between the readiness of the packet to be sent and the effective end of reception. Infact, the SoM raise the line, the ESP32 has to handle the callback, set the bit and the SoM has to wait that the ESP32 raise the HANDSHAKE and then it can send the packet. Moreover, 5 additional wires are necessary and the resepctively 5 GPIO other than the 4 for the SPI. But this implementation is quite safe, because there are check for the availability of the resource.

For sake of simplicity, I have adopted the above implementation for the rest of my thesis because I have to focus more on the effectively Networking part, that is lot more difficult. Or, another possibility is to use just one wire (HANDSHAKE) This solution of course can be upgradable. The first and big problem is the fact that every transaction just one packet is effectively sent over the SPI in just one direction. One better implementation can done by 'accumulating' the packet in a sort of linked list and then send at the same time packet from ESP32 to SoM until all the buffer is empty. To do that, to the packet can be put an header to inform the receiver which type of packet is the one just received. And, for this

implementation, is just necessary one wire (HANDSHAKE) instead of five more. For what concern how and when the packet are send by the SoM, is described in the next section because is strictly related with the implementation of the cfg80211 protocol.

3.4 Networking configuration

3.4.1 Firmware ESP32

In this section I will describe the network configuration of the ESP32 microchip microcontroller. As explained in the above sections, the network interface must act as a STA or AP in order to implement the MAC level of the ISO_OSI protocol. So, the main.c firstly i initialize la GPIO_configuration and the inzialization of the vector table. In prima istanza, utilizzando le APIs offerte dall'Espressif, viene inizializzata l'interfaccia Wi-Fi e si registrano gli handler devi eventi. Inoltre, si setta il salvatto dei dati wifi nella RAM.

```
1 void wifi_init(void)
2 {
3
4     ESP_ERROR_CHECK(esp_netif_init());
5     esp_event_loop_create_default();
6     wifi_init_config_t wifi_init_config = WIFI_INIT_CONFIG_DEFAULT();
7     ESP_ERROR_CHECK(esp_wifi_init(&wifi_init_config));
8     ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT,
9     ESP_EVENT_ANY_ID, event_handler, NULL));
10    ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT,
11    IP_EVENT_STA_GOT_IP, event_handler, NULL));
12    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
13 }
```

Inoltre, non creo task appositi for the initial configuration of the network but I just initialize the periphery with very basical configuration, and then add more details along the way. In order to do this, I have implemented STA and AP configuration in base to the cmd send by the SoM to the ESP32. So, in base at the comand receive through the switch case I can for example set the STA configuration and then try to connect to a known wifi networking by comparison of the SSID and password of the AP find. As you can see, in the STA implementation there is the setting of a base STA, settato il numero di canale su cui lavora tale network, e the structure wifi_config are in the later sections in order to make the infermace as the SoM prefer.


```

1 esp_err_t wifi_connect_sta()
2 {
3     //initial configuration and inizialization of variables and
4     buffer
5
6     esp_netif = esp_netif_create_default_wifi_sta();
7     ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
8
9     wifi_config.sta.channel = 1;
10    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config));
11    ESP_ERROR_CHECK(esp_wifi_start());
12    ESP_ERROR_CHECK(esp_wifi_scan_start(NULL, true));
13    ESP_ERROR_CHECK(esp_wifi_scan_get_ap_records(&number_max, ap_info));
14    ESP_ERROR_CHECK(esp_wifi_scan_get_ap_num(&ap_count));
15    make_scan(NUMBER_MAX_NETWORK);
16 }

```

Infact, are stored into the struct `ap_info` the networks info of the networks scanned. And then, every time that the info of a struct is printed on the consolle screen, the strcut `ap_find` is filled with the info of the network find and is send to the SoM. The function used in the Linux side is described in the above sections.

```

1 int make_scan(int NUMBER_MAX_NETWORK)
2 {
3     for (int i = 0; i < NUMBER_MAX_NETWORK; i++)
4     {
5         ESP_LOGI(TAG_SCAN, "MAC ADDRESS \t%x:%x:%x:%x:%x:%x", *
6         ap_info[i].bssid, *(1 + ap_info[i].bssid), *(2 + ap_info[i].bssid)
7         , *(3 + ap_info[i].bssid), *(4 + ap_info[i].bssid), *(5 + ap_info[
8         i].bssid));
9
10        memcpy(apfind.bssid, ap_info[i].bssid, 6 * sizeof(uint8_t));
11        memcpy(apfind.ssid, ap_info[i].ssid, 33 * sizeof(uint8_t));
12        apfind.rssi = ap_info[i].rssi;
13        apfind.primary = ap_info[i].primary;
14
15        ESP_LOGI(TAG_SCAN, "SSID \t\t%s", ap_info[i].ssid);
16        ESP_LOGI(TAG_SCAN, "RSSI \t\t%d", ap_info[i].rssi);
17        ESP_LOGI(TAG_SCAN, "CHANNEL \t%d\n", ap_info[i].primary);
18
19        send_buffer(apfind, sizeof(wifi_ap_record_t_send));
20        memset(&apfind, 0, sizeof(wifi_ap_record_t_send));
21    }
22 }

```

```

19
20 //other stuff not relevant
21
22 }
23

```

So, after that the AP networks are find, they are sent to the SoM that if it wants to connect to a certain network send another command to the ESP that activate the following function, that let to conect the microcontroller to connect to a networking providing the right information of the network.

```

1  esp_err_t connection()
2  {
3      bool connected = false;
4      int i = 0;
5      while ((!connected) && (i < NUMBER_MAX_NETWORK))
6      {
7          if (memcmp(STA_SSID, ap_info[i].ssid, strlen(STA_SSID)))
8          {
9              strncpy((char *)wifi_config.sta.ssid, STA_SSID, sizeof(
wifi_config.sta.ssid) - 1);
10             strncpy((char *)wifi_config.sta.password, STA_PASS,
sizeof(wifi_config.sta.password) - 1);
11             ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &
wifi_config));
12             ESP_LOGI(TAG_SCAN, "SSID E PASS GIUSTE, TENTATIVO CON %s"
, STA_SSID);
13             ESP_ERROR_CHECK(esp_wifi_connect());
14             connected = ESP_ERROR_CHECK(esp_wifi_connect());
15         }
16         i++;
17     }
18     return 0;
19 }
20

```

As already discussed before, the Espressif environment develop a list of event (sort of software interrupt) that are called when there are certain circumstances. Two examples are exposed: the event 'WIFI_EVENT_STA_CONNECTED' and 'IP_EVENT_STA_GOT_IP'. The first is called when the STA is effectively connected to the desired AP, the second is called when the ESP32 got the IP by DHCP protocol.

```

1  static void event_handler(void *event_handler_arg, esp_event_base_t
event_base, int32_t event_id, void *event_data)

```

```

2 {
3     switch (event_id)
4     {
5
6         case WIFI_EVENT_STA_CONNECTED:
7             wifi_event_sta_connected_t* event_mac = (
8                 wifi_event_sta_connected_t*) event_data;
9             memcpy(pacchetto_p+4, event_mac->bssid, 6);
10            memcpy(pacchetto_p+16, event_mac->bssid, 6);
11            ESP_LOGI(TAG_STA, "connected\n");
12            break;
13
14        case IP_EVENT_STA_GOT_IP:
15            ip_event_got_ip_t* event = (ip_event_got_ip_t*) event_data;
16            int ip_to_send[4];
17            ip_to_send[0] = esp_ip4_addr1_16(&event->ip_info.ip);
18            ip_to_send[1] = esp_ip4_addr2_16(&event->ip_info.ip);
19            ip_to_send[2] = esp_ip4_addr3_16(&event->ip_info.ip);
20            ip_to_send[3] = esp_ip4_addr4_16(&event->ip_info.ip);
21            printf("to ap: %x%x\n", *(4+pacchetto_p), *(5+pacchetto_p+1));
22            send_ip(ip_to_send);
23            ESP_LOGI(TAG_STA, "got ip: "IPSTR, IP2STR(&event->ip_info.ip));
24
25            ;
26            xEventGroupSetBits(evtGrp, activate_reception);
27            break;
28
29            // rest of event
30    }
31 }

```

Firstly, when the STA is connected it fulfil the packet that is send to the SoM in order to inform that is effectively connected to the desired Wi-Fi network.

Secondly, when the ESP32 got the IP, the array `ip_to_send[4]` is fullfilled with the actual value of the IP obtained and then is send over SPI to the SoM with the function `send_ip`. Moreover, as you can see, with the FreeRTOS function `xEventGroupSetBits(evtGrp, activate_reception)`; that bit is set and start the promiscuous mode of the ESP32.

On the other hand, is possible to configure the ESP32 as AP. During this project I focused more on the STA side: here, i just created a basic configuration in order to test it's capabilities. In the following piece of code, is described the creation of an AP station with the desired SSID, password. With this configuration, is possible to a max of 4 devices to connect to it, the beacon interval is of 500 ms and just for the chanel 1.

```

1 void wifi_connect_ap(void)

```

```

2 {
3     wifi_config_t wifi_config;
4     memset(&wifi_config, 0, sizeof(wifi_config_t));
5     strncpy((char *)wifi_config.ap.ssid, AP_SSID, sizeof(wifi_config.
6     sta.ssid) - 1);
7     strncpy((char *)wifi_config.ap.password, AP_PASS, sizeof(
8     wifi_config.sta.password) - 1);
9     wifi_config.ap.authmode = WIFI_AUTH_WPA2_PSK;
10    wifi_config.ap.max_connection = 4;
11    wifi_config.ap.beacon_interval = 500;
12    wifi_config.ap.channel = 1;
13
14    // esp_netif = esp_netif_create_default_wifi_ap();
15    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_AP));
16    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP, &wifi_config)
17    );
18    ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_PS_MAX_MODEM));
19    ESP_ERROR_CHECK(esp_wifi_start());
20 }

```

The last function described is when is tell to the ESP32 to disconnect from the AP associated.

```

1 void wifi_disconnect(char *TAG)
2 {
3     ESP_LOGI(TAG, "*****DISCONNECTING*****");
4     esp_wifi_disconnect();
5     esp_wifi_stop();
6     esp_netif_destroy(esp_netif);
7     ESP_LOGI(TAG, "*****DISCONNECTING COMPLETE*****");
8     xEventGroupSetBits(evtGrp, endDisconnection);
9 }

```

These are the basic network configuration for the ESP32, in the next section will be described the function that effectively handle the packed send and received after that the STA is connected to the desired network.

3.4.2 Linux Driver

For what concern the Linux driver environment, the setup of a network interface is an interesting and quite challenging task. First of all, is necessary to full the wiphy struct (that describe the wireless hardware) and the net_device struct that describe the device structure. The problem with this struct is (as suggested by the header file that describes it) as a 'BIG MISTAKE' because it mixes I/O data with strictly "high-level" data and it has to know about almost every data structure used in the INET module. So, one of the first function in the loading of the linux driver

is the creation of a pointer to the 'navigfly_struct', called g_ctx. This pointer than is used to handle the callback from the kernel space when the user space tells to do something etc. So, in this code, this point 'g_ctx' represent the overall behaviour of my network interface. In better details, this are the component of the struct:

```
1 struct esp_context {
2     struct wiphy *wiphy;
3     struct net_device *ndev;
4
5     struct semaphore sem;
6     struct work_struct ws_scan;
7     struct work_struct ws_connect;
8     struct work_struct send_packet;
9     struct work_struct received_packet_over_wifi;
10    struct work_struct ws_disconnect;
11    char connecting_ssid[33];
12    char connecting_bssid[6];
13    uint8_t real_connecting_bssid[6];
14    uint8_t real_connecting_ssid[33];
15
16    u16 disconnect_reason_code;
17    struct cfg80211_scan_request *scan_request;
18
19    struct net_device_stats stats;
20    struct napi_struct napi;
21    int status;
22    struct esp_packet *ppool;
23    struct esp_packet *rx_queue;
24    u8 data[ETH_DATA_LEN];
25    int rx_int_enabled;
26    int tx_packetlen;
27    u8 *tx_packetdata;
28    struct sk_buff *skb;
29    spinlock_t lock;
30};
```

A part from the first two struct that are briefly explained before, a semaphore sem is initialized. This is used to preserve the different callback to handle the same data interrupting themselves. So, at the beginning of every callback a semaphore is taken and then release at the end in order to preserve error. Moreover, In Linux kernel development, work_struct is a data structure used to represent a task that can be executed in the kernel's context. It provides a simple and efficient way to queue work to be performed later, either immediately or at a later time. The work_struct is typically used for deferred processing, for example, for tasks that cannot be performed immediately because they sleep, but need to be done later, when the system is not busy with other tasks. Once the work is queued, it can be

scheduled for execution by calling the `schedule_work` function. The actual work is performed by the function pointed to by the `work_struct`'s `work` member. So, associating it with a function, let the scheduler itself to executed the associated function without error. Then, the following `bssid` and `ssid` are used to store the name of the `bssid` and `ssid` associated with this interface. Then, are declared the `disconnect_reason_code` and the struct `'cfg80211_scan_request *scan_request'` to handle the scan request made by the kernel itself. It describe the main behaviour that a scan must have, such as the number of SSIDs, channels to scan on, how long to listen on each channel, in TUs and moreover. Net device stats refer to the statistics or performance metrics related to network devices such as routers, switches, and network adapters. These statistics can include information such as the number of bytes transmitted, number of packets sent, errors, and other details that help to measure the performance and health of a network device. The following `'esp_packet'` are not used.

Moreover, is declared the struct `'sk_buff *skb'`. `sk_buff` (socket buffer) is a data structure in the Linux kernel that is used to store network data packets. It contains information about the packet, such as its source and destination addresses, the protocol it uses, and the payload data. The `sk_buff` structure is used by the networking stack to manage and process network packets. It provides a convenient way for various components of the networking stack, such as the network drivers and network protocols, to access and manipulate network data. The `sk_buff` structure is a key part of the Linux kernel's networking infrastructure and is used extensively in the processing of network packets. The last stuff is the `'spinlock_t lock'`. A spinlock is a synchronization mechanism used to protect shared resources from being simultaneously accessed and modified by multiple processes. So, it is similar to the semaphore already discussed above, but with the difference that spinlocks are best suited for performance-critical parts of the system where the lock is held for a short period of time, whereas semaphores are more suitable for situations where the lock might be held for an extended period of time, or for controlling access to multiple resources.

So, the function that create the pointer is the following one:

```
1 static struct esp_context *esp_create_context(void) {
2     struct esp_context *ret = NULL;
3     struct esp_wiphy_priv_context *wiphy_data = NULL;
4     struct esp_ndev_priv_context *ndev_data = NULL;
5
6     ret = kmalloc(sizeof(*ret), GFP_KERNEL);
7     if (!ret) {
8         goto l_error;
9     }
10 }
```

```

11     ret->wiphy = wiphy_new_nm(&nvf_cfg_ops, sizeof(struct
esp_wiphy_priv_context), WIPHY_NAME);
12     if (ret->wiphy == NULL) {
13         goto l_error_wiphy;
14     }
15
16     wiphy_data = wiphy_get_navi_context(ret->wiphy);
17     wiphy_data->navi = ret;
18
19     ret->wiphy->interface_modes = BIT(NL80211_IFTYPE_STATION);
20
21     ret->wiphy->bands[NL80211_BAND_2GHZ] = &nf_band_2ghz;
22
23     ret->wiphy->max_scan_ssids = 10;
24
25     if (wiphy_register(ret->wiphy) < 0) {
26         goto l_error_wiphy_register;
27     }
28
29     /* allocate network device context. */
30     ret->ndev = alloc_netdev(sizeof(*ndev_data), NDEV_NAME,
NET_NAME_ENUM, ether_setup);
31     if (ret->ndev == NULL) {
32         goto l_error_alloc_ndev;
33     }
34     ndev_data = ndev_get_navi_context(ret->ndev);
35     ndev_data->navi = ret;
36
37     ndev_data->wdev.wiphy = ret->wiphy;
38     ndev_data->wdev.netdev = ret->ndev;
39     ndev_data->wdev.iftype = NL80211_IFTYPE_STATION;
40     ret->ndev->ieee80211_ptr = &ndev_data->wdev;
41
42     ret->ndev->netdev_ops = &nvf_ndev_ops;
43     memset(&ret->stats, 0, 23 * sizeof(unsigned long));
44
45
46     spin_lock_init(&ret->lock);
47     esp_rx_ints(ret->ndev, 1);
48     netif_start_queue(ret->ndev);
49
50     if (register_netdev(ret->ndev)) {
51         goto l_error_ndev_register;
52     }
53
54     return ret;
55     l_error_ndev_register:
56     free_netdev(ret->ndev);
57     l_error_alloc_ndev:

```

```
58     wiphy_unregister(ret->wiphy);
59     l_error_wiphy_register:
60     wiphy_free(ret->wiphy);
61     l_error_wiphy:
62     kfree(ret);
63     l_error:
64     return NULL;
65 }
```

As you can see, at the beginning of the function, are declared three pointer to different struct:

1. struct esp_context *ret = NULL;
2. struct esp_wiphy_priv_context *wiphy_data = NULL;
3. struct esp_ndev_priv_context *ndev_data = NULL;

The first is the same as discussed above, while the latter ones are used to store particular data from the root one struct, esp_context.

So, first of all is dynamically allocated the space for the struct esp_context.

Then, is allocated the wiphy context associated with the root struct by the function 'wiphy_new_nm()'. The function is used in Linux kernel programming to create a new wireless device (wiphy) and register it with the kernel's wireless subsystem. The key factor is the parameter nvf_cfg_ops: This parameter is a pointer to the configuration operations structure for the new wireless device. This structure contains the set of functions that will be used to configure the device. Then, 'WIPHY_NAME' is the define used to set a name name of the new wireless device.

After the storing of data and get the private data of the wiphy, are set the type of the station and the channel 802.11 associated. Then, is allocated the network device context through The function is used in Linux kernel programming to allocate and initialize a new network device (net_device) structure and provide basic configuration.

Then, are set the network device hook by set the he "netdev_ops" field is a pointer to a "net_device_ops" structure that defines the operations that can be performed on the network device. "net_device_ops" is a structure in Linux kernel programming that defines the operations that can be performed on a network device. It is a structure that is associated with a network device (struct net_device) and defines the functions that can be called to perform network-related tasks, such as transmitting packets, changing the device's configuration, etc. This is explained in a better way later on.

Then after basic stuff is started the The function "netif_start_queue" is part of the Linux network subsystem and is used to start the transmission queue for a network device in the Linux kernel. The function takes as an argument a pointer to the

network device structure (`ret->ndev`). The network device structure is represented by a "struct `net_device`" in the Linux kernel. The function "`netif_start_queue`" starts the transmission queue for the network device pointed to by "`ret->ndev`".

The last function is to register a network device with the kernel. When a network device is registered with the kernel, the kernel is made aware of the device's existence and the device becomes usable. The "`register_netdev`" function registers the network device pointed to by "`ret->ndev`" with the kernel, making the device usable for network-related tasks.

So, as already said, there are two hooks, the '`cfg80211_ops`' and the '`net_device_ops`'. The first is implemented in this way:

```
1 static struct cfg80211_ops nvf_cfg_ops = {
2     .scan = nvf_scan,
3     .connect = nvf_connect,
4     .disconnect = nvf_disconnect,
5 };
```

So, when the user want to make a scan of the Wi-Fi net, connect to a network or disconnect from a net this are the callback. So, basically, this three callback when are called schedule the work associated with it. So, when the work is scheduled the function associated to it is called. The function called, basically, send command to ESP32 and then receive an answer from the ESP32. Then, the function has to inform the kernel for the result obtained and tell it to the user itself. The same reasoning he be done for the latter function:

```
1 static struct net_device_ops nvf_ndev_ops = {
2     .ndo_start_xmit = nvf_ndo_start_xmit,
3     .ndo_init = esp_init,
4     .ndo_get_stats = esp_stats
5 };
```

The first callback is the key part: infact, every time that a packet must be send from the kernel to the net, this function is called. The `.ndo_init` is called when is initialised the dev and `.ndo_get_stats` is called when is necessary to provide to the user the main stats of the network.

For what concern the struct '`cfg80211_ops`', is superfluous to describe detailed their implementation but they can be seen in the Appendix. Infact, as already said before, this three functione just send a command to the ESP32 that respond to the driver. The results are described into the next chapter.

For what concern the struct '`net_device_ops`', the key function is the one associated with the '`nvf_ndo_start_xmit`'. Before the explanation of this, is important to describe the key struct used by the kernel itseld to exchange buffer

packet, called 'sk_buff'. sk_buff (short for "socket buffer") is a data structure used in the Linux kernel networking stack to represent network packets. It contains information about the packet's header, payload, and metadata, such as its source and destination addresses, protocol type, and network interface. sk_buff is used extensively throughout the kernel's network stack, including in the network device drivers, network protocols, and socket implementation.

The sk_buff data structure is designed to be flexible and efficient, with a variable-length data area that can be expanded or trimmed as needed to accommodate network packets of different sizes. The sk_buff also provides support for efficient packet handling, such as fast data copying and efficient memory management.

sk_buff is an important part of the Linux networking infrastructure, as it provides a unified and efficient way to represent network packets throughout the system. It is used by many networking components, including the TCP/IP stack, network device drivers, and the socket interface, to provide a seamless and high-performance networking experience on Linux.

Infact, the function has as parameter the pointer to the sk_buff and the net_device associated with the following prototype:

'static netdev_tx_t nvf_ndo_start_xmit(struct sk_buff *skb, struct net_device *dev)'. This code defines the nvf_ndo_start_xmit function which is a callback function that is called when a network device driver needs to transmit a packet. The function takes two arguments: a pointer to a sk_buff structure that represents the packet to be transmitted, and a pointer to a net_device structure that represents the network device that will transmit the packet.

The function first gets a pointer to a esp_ndev_priv_context structure from the network device using the ndev_get_navi_context function. It then performs some error checks on the sk_buff structure, including checking its length and comparing its device with the given net_device. If any errors are found, the function frees the sk_buff and returns NETDEV_TX_OK (indicating success).

If the sk_buff is valid, the function calculates the length of the packet and adds a padding length of sizeof(struct esp_payload_header) to it. It then checks if there is enough headroom in the sk_buff for the padding, and if not, it linearizes the sk_buff and reallocates a new one with enough headroom. If there is enough headroom, it simply pushes the padding length to the start of the sk_buff.

The function then populates the padding data with the necessary information such as the packet length, offset, and checksum. It then schedules a work queue to send the packet using priv->navi->send_packet.

Finally, the original sk_buff is freed, and the function returns NETDEV_TX_OK indicating that the packet transmission was successful.

Moreover, when is scheduled the function it is called the function 'esp_send_packet'

This code defines the function esp_send_packet which is a callback function for a workqueue and is responsible for sending the packet over the network interface.

The function first obtains a pointer to the `esp_context` structure by using the `container_of` macro. This structure contains all the information required for sending the packet.

Next, it acquires a semaphore to prevent concurrent access to the data structure. If it fails to acquire the semaphore, the function returns without doing anything.

The network interface is temporarily disabled to ensure that no other packets are sent while this packet is being transmitted.

The IRQ line is then set to 1, which is used to signal the ESP32 that the packet is being sent.

The `send_rx_packet` function is called to actually send the packet. This function is likely part of the lower-level driver code that is specific to the ESP32 hardware.

After the packet is sent, the function updates the statistics of the network interface and wakes up the queue for the interface so that it can transmit more packets.

Finally, the semaphore is released to allow other threads to access the data structure.

So, the packet is sent to the ESP32 that acquire it and sent over the net.

Instead, for what concern the reception of a packet, first of all is necessary to activate an input interrupt that schedule a work, this work is associated with a function that must be scheduled by the scheduler of Linux itself. When the function is scheduled, it call another function that handle the majority of the stuff and start the reception itself of the packet and then inform the kernel.

Chapter 4

Results obtained

In the following chapter, the results obtained will be analyzed, providing an examination of what happens from the moment the Linux driver is loaded and the firmware is started on the ESP32-WROVER-E microcontroller.

In the next Figure is shown the board on which this project is developed. The SoM is used through a SoM tester, a custom board that provide PIN to connect with the SoM itself. In order to communicate with the SoM and load the driver, Putty is used via an SSH connection using an Ethernet cable. For this protocol, you connect using the SoM's IP address. Additionally, a logic analyzer is used for initial debugging of the SPI protocol and packet management. The driver is copied from the virtual machine to the SoM using the Linux 'scp' command, which also utilizes the SSH protocol.

For what concern the loading of the firmware, a serial protocol is used. So, through a Mirco-USB cable is possible to both flash and monitor the log of ESP32 itself.

The supply of the SoM is about 5V and the ESP32's one is 3.3V.

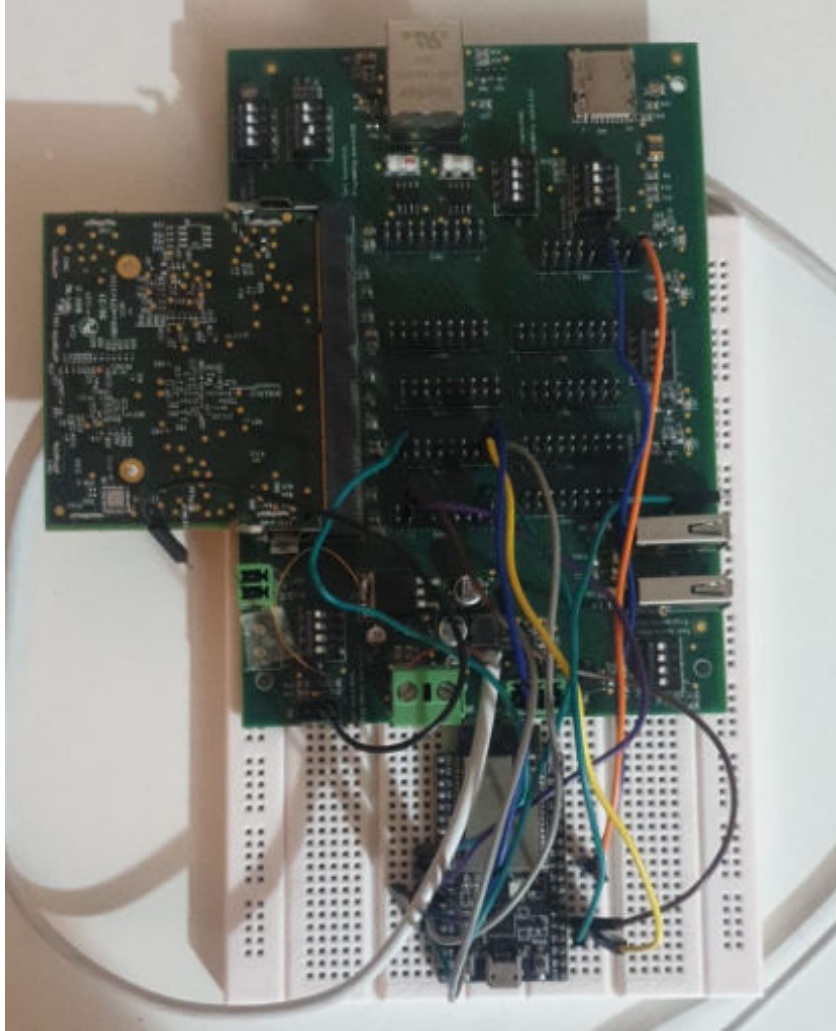
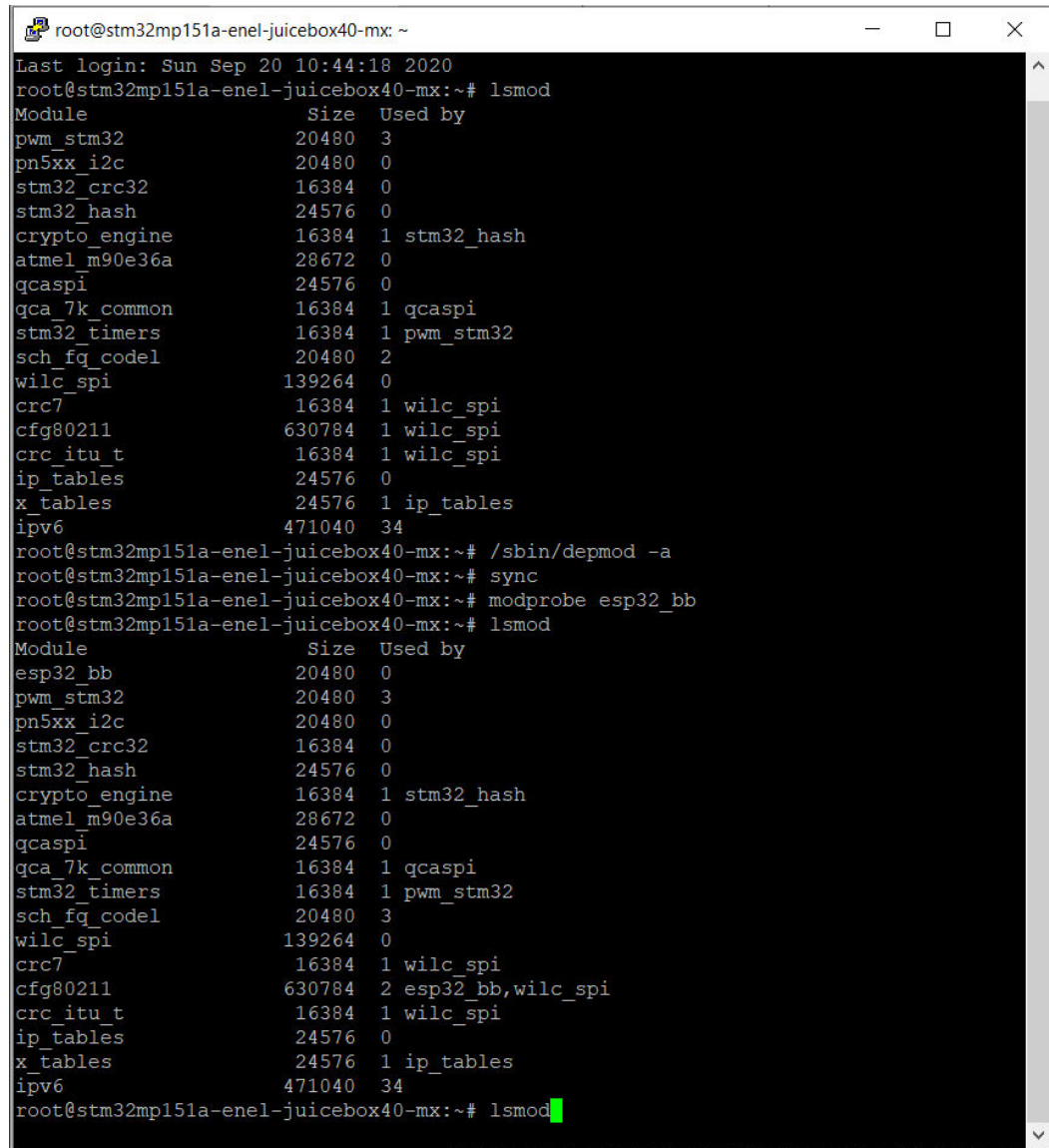


Figure 4.1: SoM and ESP32

The driver is copied from the virtual machine to the SoM using the Linux `'scp'` command, which also utilizes the SSH protocol.

Once is logged, in order to update the dependencies, the command `'/sbin/depmod -a'` and `'sync'` are used. In order to effectively load the driver the command `'modprobe nameofdriver'` is used. Using the command `'lsmod'` it is possible to see all the modules that are currently loaded into the kernel.

As you can see in the next image, after the command `modprobe` is utilized, the `esp32_bb` (name of the driver) is correctly load and recognized by the kernel itself as shown in the next image.



```

root@stm32mp151a-enel-juicebox40-mx: ~
Last login: Sun Sep 20 10:44:18 2020
root@stm32mp151a-enel-juicebox40-mx:~# lsmod
Module                  Size  Used by
pwm_stm32                20480  3
pn5xx_i2c                20480  0
stm32_crc32             16384  0
stm32_hash              24576  0
crypto_engine           16384  1 stm32_hash
atmel_m90e36a           28672  0
qcaspi                  24576  0
qca_7k_common           16384  1 qcaspi
stm32_timers            16384  1 pwm_stm32
sch_fq_codel            20480  2
wilc_spi                139264  0
crc7                    16384  1 wilc_spi
cfg80211                630784  1 wilc_spi
crc_itu_t               16384  1 wilc_spi
ip_tables               24576  0
x_tables                24576  1 ip_tables
ipv6                    471040  34
root@stm32mp151a-enel-juicebox40-mx:~# /sbin/depmod -a
root@stm32mp151a-enel-juicebox40-mx:~# sync
root@stm32mp151a-enel-juicebox40-mx:~# modprobe esp32_bb
root@stm32mp151a-enel-juicebox40-mx:~# lsmod
Module                  Size  Used by
esp32_bb                20480  0
pwm_stm32                20480  3
pn5xx_i2c                20480  0
stm32_crc32             16384  0
stm32_hash              24576  0
crypto_engine           16384  1 stm32_hash
atmel_m90e36a           28672  0
qcaspi                  24576  0
qca_7k_common           16384  1 qcaspi
stm32_timers            16384  1 pwm_stm32
sch_fq_codel            20480  3
wilc_spi                139264  0
crc7                    16384  1 wilc_spi
cfg80211                630784  2 esp32_bb,wilc_spi
crc_itu_t               16384  1 wilc_spi
ip_tables               24576  0
x_tables                24576  1 ip_tables
ipv6                    471040  34
root@stm32mp151a-enel-juicebox40-mx:~# lsmod

```

Figure 4.2: Loading of esp32_bb module

Then, using the command 'dmesg' is possible to see if the driver has communicate something. As you can see from the next image, the SPI configuration was initially recorded, including the controller, etc., and the network interface configuration was also recorded. During recording, I made sure that a command was sent to the ESP32, and it responded with a message containing the MAC address.

```
[ 486.944174] SPI driver Registered
[ 487.075168] e0:e2:e6:7b:d6:74 , MAC ADDRESS
```

Figure 4.3: ESP32-WROVER-E's mac address on SoM terminal.

```
COMANDO RICEVUTO: 0
e0:e2:e6:7b:d6:74, 48 6
```

Figure 4.4: ESP32-WROVER-E's mac address ESP32 terminal.

Furthermore, as you can see from the following image, by typing the 'ifconfig' command it's possible to see that the network interface called 'Enrico0' has been properly generated with the corresponding MAC address received from the ESP32.

```
enrico0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500 metric 1
    ether e0:e2:e6:7b:d6:74 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure 4.5: enrico0 network interface

Moreover, using the command "iw list" that is typically used in Linux-based systems to display information about available wireless interfaces and their capabilities. In the following image are shown some of the capabilities of the wi-fi interface Enrico.

```
root@stm32mp151a-enel-juicebox40-mx:~# iw list
Wiphy enrico
    max # scan SSIDs: 10
    max scan IEs length: 0 bytes
    max # sched scan SSIDs: 0
    max # match sets: 0
    Retry short limit: 7
    Retry long limit: 4
    Coverage class: 0 (up to 0m)
    Available Antennas: TX 0 RX 0
    Supported interface modes:
        * managed
    Band 1:
        Bitrates (non-HT):
            * 1.0 Mbps
            * 2.0 Mbps
            * 5.5 Mbps
            * 11.0 Mbps
            * 6.0 Mbps
            * 9.0 Mbps
            * 12.0 Mbps
            * 18.0 Mbps
            * 24.0 Mbps
            * 36.0 Mbps
            * 48.0 Mbps
            * 54.0 Mbps
```

Figure 4.6: Wiphy enrico capabilities

So, overall, the Linux driver is correctly recognized.

In the following, the ability to scan the present wireless networks will be tested. With the command 'iw dev enrico0 scan' is tell to the SoM to make a scan of the wireless network.

As you can see in the next images, three WI-FIs are find and the ESP32-WROVER-E informs the SoM that print it on the 'dmesg'.


```

[ 76.525894] MAC ADDRESS      62:ed:d9:f5:ff:53
[ 76.528568] SSID:            Enrico
[ 76.531312] CANALE:          10
[ 76.533750] RSSI:            -50
[ 76.555430] TROVATA
[ 76.585912] MAC ADDRESS      62:cc:21:30:69:d9
[ 76.588581] SSID:            TIM-98619448
[ 76.591828] CANALE:          11
[ 76.594367] RSSI:            -50
[ 76.645932] MAC ADDRESS      3c:37:12:27:a7:e8
[ 76.648602] SSID:            SMARTWEB24-GPC
[ 76.652052] CANALE:          1
[ 76.654490] RSSI:            -89

```

Figure 4.7: Read of the information provided by ESP32 on SoM terminal.

```

3 WI-FIs ARE FIND:
I (16743) SCAN: MAC ADDRESS    62:ed:d9:f5:ff:53
I (16743) SCAN: SSID           Enrico
I (16743) SCAN: RSSI           -50
I (16743) SCAN: CHANNEL        10

I (16773) SCAN: MAC ADDRESS    62:cc:21:30:69:d9
I (16773) SCAN: SSID           TIM-98619448
I (16773) SCAN: RSSI           -50
I (16773) SCAN: CHANNEL        11

I (16833) SCAN: MAC ADDRESS    3c:37:12:27:a7:e8
I (16833) SCAN: SSID           SMARTWEB24-GPC
I (16833) SCAN: RSSI           -89
I (16833) SCAN: CHANNEL        1

```

Figure 4.8: Wi-Fis find by ESP32

So, how you can see, the Wi-Fi network with the name Enrico is find. This net is created with the Hotspot of my cellphone.

Next step, is to send command to ESP32-WROVER-E to connect to a wireless network, the same network Enrico.

```
COMMAND RECEIVED: 2
I (26323) SCAN: SSID AND PASSWORD MATCH WITH THE NETWORK: Enrico
I (26443) wifi:new:<2,0>, old:<1,0>, ap:<255,255>, sta:<2,0>, prof:1
I (27123) wifi:state: init -> auth (b0)
I (27123) wifi:state: auth -> assoc (0)
I (27163) wifi:state: assoc -> run (10)
I (27203) wifi:connected with Enrico, aid = 5, channel 2, BW20, bssid = 62:ed:d9:f5:ff:53
I (27203) wifi:security: WPA2-PSK, phy: bgn, rssi: -35
I (27213) wifi:pm start, type: 1

I (27213) CONNECTION STA: connected

W (27233) wifi:<ba-add>idx:0 (ifx:0, 62:ed:d9:f5:ff:53), tid:0, ssn:0, winSize:64
I (27253) wifi:AP's beacon interval = 102400 us, DTIM period = 2
192
I (28113) esp_netif_handlers: sta ip: 192.168.130.109, mask: 255.255.255.0, gw: 192.168.130.252
I (28113) wifi:ic_enable_sniffer
```

Figure 4.9: Connection of ESP32 to network Enrico

Thanks to the DHCP, the ESP32 itself obtain the IP 192.168.130.109 with a gateway 192.168.130.252 and more over.

So, now Linux see the fact that its network interface, Enrico0, is connected to the Wi-Fi net, Enrico. Moreover, in order to store the packet arriving to the ESP32 from the net, is necessary to put the microcontroller in 'promiscuous mode'.

As you can see, the IP address is correctly added to the enrico0 interface. As you can see, if i try to ping some ip that can be achieved by the enrico0 interface, linux driver try to send packet through it but it does not receive a correct answer by no one.

As you can see from the images above, is shown as the metrics of the network are uploaded correctly by the interface itself:

```
enrico0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500 metric 1
    inet 192.168.130.109 netmask 255.255.255.0 broadcast 192.168.130.255
    ether e0:e2:e6:7b:d6:74 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 28 bytes 6758 (6.5 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure 4.10: enrico0 network capabilities.

```
[ 299.215479] TO ESP32: packet length: 62 offset: 14
[ 299.253327] TO ESP32: packet length: 124 offset: 16
[ 299.290433] TO ESP32: packet length: 213 offset: 15
[ 299.362918] TO ESP32: packet length: 90 offset: 14
[ 299.400462] TO ESP32: packet length: 62 offset: 14
[ 299.466031] TO ESP32: packet length: 90 offset: 14
[ 299.522972] TO ESP32: packet length: 213 offset: 15
[ 299.621968] TO ESP32: packet length: 90 offset: 14
[ 299.773452] TO ESP32: packet length: 213 offset: 15
[ 299.973965] TO ESP32: packet length: 195 offset: 13
[ 300.283029] TO ESP32: packet length: 124 offset: 16
[ 301.003383] TO ESP32: packet length: 195 offset: 13
```

Figure 4.11: Packet sent by SoM to ESP32, seen on SoM terminal.

```
FROM SOM: packet length: 62 offset: 14
FROM SOM: packet length: 124 offset: 16
FROM SOM: packet length: 213 offset: 15
FROM SOM: packet length: 90 offset: 14
FROM SOM: packet length: 62 offset: 14
FROM SOM: packet length: 90 offset: 14
FROM SOM: packet length: 213 offset: 15
FROM SOM: packet length: 90 offset: 14
FROM SOM: packet length: 213 offset: 15
FROM SOM: packet length: 195 offset: 13
FROM SOM: packet length: 124 offset: 16
FROM SOM: packet length: 195 offset: 13
```

Figure 4.12: Packet sent by SoM to ESP32, seen on ESP32 terminal.

As you can see in the next figure, the driver itself also reckon correctly the packet that the ESP32 send to the SoM.

```
enrico0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500 metric 1
    inet 192.168.130.109 netmask 255.255.255.0 broadcast 192.168.130.255
    ether e0:e2:e6:7b:d6:74 txqueuelen 1000 (Ethernet)
    RX packets 25 bytes 5380 (5.3 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 28 bytes 6025 (5.9 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure 4.13: Update metrics of the enrico0 interface.

So, stuff works properly but the ping seems to not works properly. There is an exchange of packet but the Linux Kernel does not reckon correctly the packet. It is necessary further debugging to understand at what level the error happens.

I tried to using Wireshark in order to understand at bit level what happens to the packet send by the ESP32 on the net. At first glance, the packet sniffed and

shown by the software include correctly the MAC address of the ESP32, but further analyses has to be performed in order to understand at which level of incapsulation happens the error.

Chapter 5

Conclusion

Overall, the ESP32 seems to be a quite good microcontroller with an high potentials, providing good hardware capabilities, Wi-Fi and Bluetooth for a low price. But, for what concern the topic of this Thesis, this microcontroller is not what Bitron S.p.A. is looking for. The major downside of this microcontroller are the APIs offered by the Espressif. Infact, they are quite opaque:

1. In debug mode, the IDE can not shown you what happens during an execution of a function provided by the Espressif. This puts the programmer in trouble because does not have clear at hand what happens inside the microcontroller;
2. Differently from the HAL provided by ST for their microcontroller, you can not see how the function interact with the low layer;
3. Espressif does not currently provide detailed documentation of its APIs, but it does offer a summary description of what the functions offered do at a high level;

So, in order to avoid the usage of their APIs, is necessary to write starting from the User Manual of the microcontroller in order to have a detailed idea of how the peripheral are configured, how data is exchanged between the peripheral and micro, etc. But this will takes an enormous amount of time in order to achieve the goal.

Moreover, during the period of my internship, Espressif does not provide any Linux driver on the main line of the Kernel as their competitors do. My work could be a good starting point in order to develop it but much work is still needed to achieve the stability required by the industrial world. Infact, from a enterprise prospective, is not sustainable write an entire Linux driver for an object of another industrial reality. It is therefore more attractive to use another network interface, already used and tested in the industrial world and let Bitron S.p.A. to focus on the developing of thier products, since its business is not the IoT world.

In conclusion, I can say that the microcontroller itself and the APIs offered are good enough to build some application, such as Web Server and so on. Its great in order to approach the IoT world and in some way it remembers the Arduino approach. It has of course great potentials that can express in the future: the hardware behaviour is already quite good but what lack is a detailed documentation for their software environment that offer.

Appendix A

Driver Code

A.0.1 cfg80211esp.c

```
1 #include <linux/module.h>
2 #include <net/cfg80211.h> /* wiphy and probably everything that would
   required for FullMAC driver */
3 #include <linux/skbuff.h>
4 #include <linux/interrupt.h> /* mark_bh */
5 #include <linux/workqueue.h> /* work_struct */
6 #include <linux/semaphore.h>
7 #include <linux/netdevice.h>
8
9 #include "espspi.h"
10 #include "cfg80211esp.h"
11 #include "netdevicee.h"
12
13 /* helper function that will retrieve main context from "priv" data
   of the wiphy */
14 static struct esp_wiphy_priv_context *
15 wiphy_get_navi_context(struct wiphy *wiphy) { return (struct
   esp_wiphy_priv_context *) wiphy_priv(wiphy); }
16
17 /* helper function that will retrieve main context from "priv" data
   of the network device */
18 static struct esp_ndev_priv_context *
19 ndev_get_navi_context(struct net_device *ndev) { return (struct
   esp_ndev_priv_context *) netdev_priv(ndev); }
20
21 /* Helper function that will prepare structure with BSS information
   and "inform" the kernel about "new" BSS */
22 static void inform_dummy_bss(struct esp_context *navi) {
23     struct cfg80211_bss *bss = NULL;
```

```

24     struct cfg80211_inform_bss data = {
25         .chan = &navi->wiphy->bands[NL80211_BAND_2GHZ]->channels
26         [0], /* the only channel */
27         .scan_width = NL80211_BSS_CHAN_WIDTH_20,
28         /* signal "type" may be specified before wiphy
29         registration by setting wiphy->signal_type */
30         .signal = 1337,
31     };
32
33     /* ie - array of tags that usually retrieved from beacon frame or
34     probe response. */
35     char ie[(sizeof(navi->connecting_ssid)-1) + 2] = {WLAN_EID_SSID,
36     (sizeof(navi->connecting_ssid)-1)};
37     memcpy(ie + 2, navi->connecting_ssid, (sizeof(navi->
38     connecting_ssid)-1));
39
40     /* also it possible to use cfg80211_inform_bss() instead of
41     cfg80211_inform_bss_data() */
42     bss = cfg80211_inform_bss_data(navi->wiphy, &data,
43     CFG80211_BSS_FTYPE_UNKNOWN, navi->connecting_bssid, 0,
44     WLAN_CAPABILITY_ESS, 100,
45     ie, sizeof(ie), GFP_KERNEL);
46
47     /* free, cfg80211_inform_bss_data() returning cfg80211_bss
48     structure refcounter of which should be decremented if its not
49     used. */
50     cfg80211_put_bss(navi->wiphy, bss);
51 }
52
53 /* "Scan" routine. It just inform the kernel about BSS and finished
54 scan.
55 * When scan is done it should call cfg80211_scan_done() to inform
56 the kernel that scan is finished.
57 * This routine called through workqueue, when the kernel asks about
58 scan through cfg80211_ops. */
59 static void esp_scan_routine(struct work_struct *w) {
60     struct esp_context *navi = container_of(w, struct esp_context,
61     ws_scan);
62     struct cfg80211_scan_info info = {
63
64         .aborted = false,
65     };
66     int i;
67
68     gpio_set_value(IRQ_NO_CMD, 1);
69     etx_spi_write_cmd(MAKE_SCAN);
70     for(i=0; i < 10; i++){

```



```

58     wifi_ap_record_t_send *apfind = kmalloc(sizeof(
wifi_ap_record_t_send), GFP_KERNEL);
59
60     etx_spi_read_struct(apfind, sizeof(wifi_ap_record_t_send));
61     gpio_set_value(IRQ_NO_CMD, 0);
62     pr_info("MAC ADDRESS \t%x:%x:%x:%x:%x:%x\n", *apfind->bssid, *(1+
apfind->bssid), *(2+apfind->bssid), *(3+apfind->bssid), *(4+apfind->
bssid), *(5+apfind->bssid));
63     pr_info("SSID: \t\t%s\n", apfind->ssid);
64     pr_info("CANALE: \t\t%x\n", apfind->primary);
65     pr_info("RSSI: \t\t%d\n", apfind->rssi);
66
67     memcpy(navi->connecting_ssid, apfind->ssid, strlen(apfind->ssid))
;
68     memcpy(navi->connecting_bssid, apfind->bssid, strlen(apfind->
bssid));
69     memset(navi->connecting_ssid, 0, 33);
70     memset(navi->connecting_bssid, 0, 6);
71     /* inform with BSS */
72     inform_dummy_bss(navi);
73     kfree(apfind);
74 }
75
76 if(down_interruptible(&navi->sem)) {
77     return;
78 }
79
80 /* finish scan */
81 cfg80211_scan_done(navi->scan_request, &info);
82
83 navi->scan_request = NULL;
84
85 up(&navi->sem);
86 }
87
88 /* It just checks SSID of the ESS to connect and informs the kernel
that connect is finished.
89 * It should call cfg80211_connect_bss() when connect is finished or
cfg80211_connect_timeout() when connect is failed.
90 *
91 * This routine called through workqueue, when the kernel asks about
connect through cfg80211_ops. */
92 static void esp_connect_routine(struct work_struct *w) {
93     struct esp_context *navi = container_of(w, struct esp_context,
ws_connect);
94
95     if(down_interruptible(&navi->sem)) {
96         return;
97     }

```

```

98
99     if (memcmp(navi->connecting_ssid, SSID_DUMMY, sizeof(SSID_DUMMY))
100         != 0) {
101         cfg80211_connect_timeout(navi->ndev, NULL, NULL, 0,
102         GFP_KERNEL, NL80211_TIMEOUT_SCAN);
103     } else {
104         /* we can connect to ESS that already know. If else,
105         technically kernel will only warn.*/
106         /* so, lets send bss to the kernel before complete. */
107         inform_dummy_bss(navi);
108
109         /* also its possible to use cfg80211_connect_result() or
110         cfg80211_connect_done() */
111         cfg80211_connect_bss(navi->ndev, NULL, NULL, NULL, 0, NULL,
112         0, WLAN_STATUS_SUCCESS, GFP_KERNEL,
113         NL80211_TIMEOUT_UNSPECIFIED);
114     }
115
116     up(&navi->sem);
117 }
118
119 /* Just calls cfg80211_disconnected() that informs the kernel that
120 disconnect is complete.
121 * Overall disconnect may call cfg80211_connect_timeout() if
122 disconnect interrupting connection routine.
123 * This routine called through workqueue, when the kernel asks about
124 disconnect through cfg80211_ops. */
125 static void esp_disconnect_routine(struct work_struct *w) {
126
127     struct esp_context *navi = container_of(w, struct esp_context,
128     ws_disconnect);
129
130     if(down_interruptible(&navi->sem)) {
131         return;
132     }
133
134     cfg80211_disconnected(navi->ndev, navi->disconnect_reason_code,
135     NULL, 0, true, GFP_KERNEL);
136
137     navi->disconnect_reason_code = 0;
138
139     up(&navi->sem);
140 }
141
142 /* callback that called by the kernel when user decided to scan.
143 * This callback should initiate scan routine(through work_struct)
144 and exit with 0 if everything ok.
145 * Scan routine should be finished with cfg80211_scan_done() call. */

```

```

136 static int nvf_scan(struct wiphy *wiphy, struct cfg80211_scan_request
    *request) {
137     struct esp_context *navi = wiphy_get_navi_context(wiphy)->navi;
138     static int i = 0;
139
140     if(down_interruptible(&navi->sem)) {
141         return -ERESTARTSYS;
142     }
143
144     if (navi->scan_request != NULL) {
145         up(&navi->sem);
146         return -EBUSY;
147     }
148
149     navi->scan_request = request;
150
151     up(&navi->sem);
152
153     if (!schedule_work(&navi->ws_scan)) {
154         return -EBUSY;
155     }
156
157     return 0; /* OK */
158 }
159
160 /* callback that called by the kernel when there is need to "connect"
    to some network.
161 * It inits connection routine through work_struct and exits with 0
    if everything ok.
162 * connect routine should be finished with cfg80211_connect_bss()/
    cfg80211_connect_result()/cfg80211_connect_done() or
    cfg80211_connect_timeout(). */
163 static int nvf_connect(struct wiphy *wiphy, struct net_device *dev,
    struct cfg80211_connect_params *sme) {
164     struct esp_context *navi = wiphy_get_navi_context(wiphy)->navi;
165     size_t ssid_len = sme->ssid_len > 15 ? 15 : sme->ssid_len;
166
167     if(down_interruptible(&navi->sem)) {
168         return -ERESTARTSYS;
169     }
170
171     pr_err("%s\n", sme->ssid);
172     memcpy(navi->connecting_ssid, sme->ssid, ssid_len);
173
174     up(&navi->sem);
175
176     if (!schedule_work(&navi->ws_connect)) {
177         return -EBUSY;
178     }
179

```

```

180     return 0;
181 }
182 /* callback that called by the kernel when there is need to "
183    disconnect" from currently connected network.
184    * It inits disconnect routine through work_struct and exits with 0
185    if everything ok.
186    * disconnect routine should call cfg80211_disconnected() to inform
187    the kernel that disconnection is complete. */
188 static int nvf_disconnect(struct wiphy *wiphy, struct net_device *dev
189 ,
190                          u16 reason_code) {
191     struct esp_context *navi = wiphy_get_navi_context(wiphy)->navi;
192
193     if(down_interruptible(&navi->sem)) {
194         return -ERESTARTSYS;
195     }
196
197     navi->disconnect_reason_code = reason_code;
198
199     up(&navi->sem);
200
201     if (!schedule_work(&navi->ws_disconnect)) {
202         return -EBUSY;
203     }
204     return 0;
205 }
206
207 /* Structure of functions for FullMAC 80211 drivers.
208    * Functions that implemented along with fields/flags in wiphy
209    structure would represent drivers features.
210    * */
211 static struct cfg80211_ops nvf_cfg_ops = {
212     .scan = nvf_scan,
213     .connect = nvf_connect,
214     .disconnect = nvf_disconnect,
215 };
216
217 static int esp_init(struct net_device *dev){
218     uint8_t * my_mac_address;
219     my_mac_address = kmalloc(6, GFP_KERNEL);
220     gpio_set_value(IRQ_NO_CMD, 1);
221     etx_spi_write_cmd(READ_MAC);
222     etx_spi_read32(my_mac_address);
223     gpio_set_value(IRQ_NO_CMD, 0);
224     pr_info( "%x:%x:%x:%x:%x:%x  , MAC ADDRESS\n", *my_mac_address,*(
225 my_mac_address+1),*(my_mac_address+2),*(my_mac_address+3), *(
226 my_mac_address+4), *(my_mac_address+5));
227     memcpy(dev->dev_addr, my_mac_address, ETH_ALEN);
228     kfree(my_mac_address);

```

```

222
223     return 0;
224 }
225
226 void send_packet_to_esp(char *data, int len, struct net_device *dev){
227     gpio_set_value(IRQ_NO, 1);
228     send_rx_packet(len, data);
229     gpio_set_value(IRQ_NO, 0);
230
231 }
232
233 static void esp_send_packet(struct work_struct *w) {
234
235     struct esp_context *navi = container_of(w, struct esp_context,
236     send_packet);
237
238     if(down_interruptible(&navi->sem)) {
239         return;
240     }
241     netif_tx_disable(navi->ndev);
242     gpio_set_value(IRQ_NO, 1);
243     send_rx_packet(navi->tx_packetlen, navi->data);
244
245     gpio_set_value(IRQ_NO, 0);
246     pr_err("RETURN DA ESP_SEND_PACKET\n");
247     netif_wake_queue(navi->ndev);
248
249     up(&navi->sem);
250 }
251
252 /* Network packet transmit.
253  * Callback that called by the kernel when packet of data should be
254  * sent.*/
255
256 static netdev_tx_t nvf_ndo_start_xmit(struct sk_buff *skb,
257                                     struct net_device *dev) {
258
259     struct esp_ndev_priv_context *priv = ndev_get_navi_context(dev);
260     char *data, shortpkt[ETH_ZLEN];
261     int len;
262     static int counter = 0;
263     counter++;
264
265     if(skb->dev!=dev){
266         pr_err("ho dispositivi diversi\n");
267     }
268
269     len = skb->len;
270     pr_err("LUNGHEZZA PACCHETTO N %d DA INVIARE: %d\n", counter, len)
271 ;

```

```

268
269     priv->navi->stats.tx_packets++;
270     priv->navi->stats.tx_bytes += len;
271     priv->navi->tx_packetlen = len;
272
273     memset(priv->navi->data, 0, 1500);
274     memcpy(priv->navi->data, skb->data, len);
275     /* Dont forget to cleanup skb, as its ownership moved to xmit
276     callback. */
277     schedule_work(&priv->navi->send_packet);
278     netif_stop_queue(priv->navi->ndev);
279     kfree_skb(skb);
280     pr_err("RETURN\n");
281     return NEIDDEV_TX_OK;
282 }
283
284 struct net_device_stats *esp_stats(struct net_device *dev)
285 {
286     //struct snull_priv *priv = netdev_priv(dev);
287     struct esp_ndev_priv_context *priv = ndev_get_navi_context(dev);
288     return &priv->navi->stats;
289 }
290
291 /* Structure of functions for network devices.
292 */
293 static struct net_device_ops nvf_ndev_ops = {
294     .ndo_start_xmit = nvf_ndo_start_xmit,
295     .ndo_init = esp_init,
296     .ndo_get_stats = esp_stats
297 };
298
299 /*
300 * Enable and disable receive interrupts. da snull
301 */
302 static void esp_rx_ints(struct net_device *dev, int enable)
303 {
304     struct esp_ndev_priv_context *priv = ndev_get_navi_context(dev);
305     priv->navi->rx_int_enabled = enable;
306 }
307
308 /* Function that creates wiphy context and net_device with
309 wireless_dev.
310 * wiphy/net_device/wireless_dev is basic interfaces for the kernel
311 to interact with driver as wireless one.
312 * It returns driver's main "esp" context. */
313 static struct esp_context *esp_create_context(void) {
314     struct esp_context *ret = NULL;
315     struct esp_wiphy_priv_context *wiphy_data = NULL;
316     struct esp_ndev_priv_context *ndev_data = NULL;

```

```

314
315     /* allocate for esp context*/
316     ret = kmalloc(sizeof(*ret), GFP_KERNEL);
317     if (!ret) {
318         goto l_error;
319     }
320
321     /* allocate wiphy context, also it possible just to use wiphy_new
322     () function.
323     * wiphy should represent physical FullMAC wireless device.
324     * One wiphy can have serveral network interfaces – for that u
325     need to implement add_virtual_intf() and co. from cfg80211_ops. */
326     ret->wiphy = wiphy_new_nm(&nvf_cfg_ops, sizeof(struct
327     esp_wiphy_priv_context), WIPHY_NAME);
328     if (ret->wiphy == NULL) {
329         goto l_error_wiphy;
330     }
331
332     /* save esp context in wiphy private data. */
333     wiphy_data = wiphy_get_navi_context(ret->wiphy);
334     wiphy_data->navi = ret;
335
336     /* set device object as wiphy "parent", I dont have any device
337     yet. */
338     /* set_wiphy_dev(ret->wiphy, dev); */
339
340     /* wiphy should determinate it type */
341     /* add other required types like "BIT(NL80211_IFTYPE_STATION) |
342     BIT(NL80211_IFTYPE_AP)" etc. */
343     ret->wiphy->interface_modes = BIT(NL80211_IFTYPE_STATION);
344
345     /* wiphy should have at least 1 band. */
346     /* fill also NL80211_BAND_5GHZ if required */
347     ret->wiphy->bands[NL80211_BAND_2GHZ] = &nf_band_2ghz;
348
349     /* scan – to define max_scan_ssids */
350     ret->wiphy->max_scan_ssids = 10;
351     //ret->scan_request->wiphy = ret->wiphy;
352
353     if (wiphy_register(ret->wiphy) < 0) {
354         goto l_error_wiphy_register;
355     }
356
357     /* allocate network device context. */
358     ret->ndev = alloc_netdev(sizeof(*ndev_data), NDEV_NAME,
359     NET_NAME_ENUM, ether_setup);
360     if (ret->ndev == NULL) {
361         goto l_error_alloc_ndev;
362     }

```

```

357  /* fill private data of network context.*/
358  ndev_data = ndev_get_navi_context(ret->ndev);
359  ndev_data->navi = ret;
360
361  /* fill wireless_dev context.
362   * wireless_dev with net_device can be represented as inherited
363   class of single net_device. */
364  ndev_data->wdev.wiphy = ret->wiphy;
365  ndev_data->wdev.netdev = ret->ndev;
366  ndev_data->wdev.iftype = NL80211_IFTYPE_STATION;
367  ret->ndev->ieee80211_ptr = &ndev_data->wdev;
368
369  /* set device object for net_device */
370  /* SET_NETDEV_DEV(ret->ndev, wiphy_dev(ret->wiphy)); */
371
372  /* set network device hooks. */
373  ret->ndev->netdev_ops = &nvf_ndev_ops;
374  memset(&ret->stats, 0, 23 * sizeof(unsigned long));
375
376  //snul_setup_pool(ret->ndev);
377  spin_lock_init(&ret->lock);
378  esp_rx_ints(ret->ndev, 1);
379  netif_start_queue(ret->ndev);
380
381  /* net_device initialization. */
382  if (register_netdev(ret->ndev)) {
383      goto l_error_ndev_register;
384  }
385
386  return ret;
387  l_error_ndev_register:
388  free_netdev(ret->ndev);
389  l_error_alloc_ndev:
390  wiphy_unregister(ret->wiphy);
391  l_error_wiphy_register:
392  wiphy_free(ret->wiphy);
393  l_error_wiphy:
394  kfree(ret);
395  l_error:
396  return NULL;
397 }
398
399 static void esp_free(struct esp_context *ctx) {
400     if (ctx == NULL) {
401         return;
402     }
403     unregister_netdev(ctx->ndev);
404     free_netdev(ctx->ndev);
405     wiphy_unregister(ctx->wiphy);

```



```

405     wiphy_free(ctx->wiphy);
406     kfree(ctx);
407     netif_stop_queue(ctx->ndev);
408 }
409
410 static struct esp_context *g_ctx = NULL;
411
412 static int __init virtual_wifi_init(void) {
413     etx_spi_init();
414     g_ctx = esp_create_context();
415     if (g_ctx != NULL) {
416         sema_init(&g_ctx->sem, 1);
417
418         INIT_WORK(&g_ctx->ws_connect, esp_connect_routine);
419         g_ctx->connecting_ssid[0] = 0;
420         INIT_WORK(&g_ctx->ws_disconnect, esp_disconnect_routine);
421         g_ctx->disconnect_reason_code = 0;
422         INIT_WORK(&g_ctx->ws_scan, esp_scan_routine);
423         INIT_WORK(&g_ctx->send_packet, esp_send_packet);
424         g_ctx->scan_request = NULL;
425     }
426     return g_ctx == NULL;
427 }
428 static void __exit virtual_wifi_exit(void) {
429     etx_spi_exit();
430     cancel_work_sync(&g_ctx->ws_connect);
431     cancel_work_sync(&g_ctx->ws_disconnect);
432     cancel_work_sync(&g_ctx->ws_scan);
433     cancel_work_sync(&g_ctx->send_packet);
434     esp_free(g_ctx);
435 }
436 module_init(virtual_wifi_init);
437 module_exit(virtual_wifi_exit);
438 MODULE_LICENSE("GPL v2");

```

A.0.2 espspi.c

```

1 #include "espspi.h"
2 #include "cfg80211esp.h"
3 #include "netdevicee.h"
4 #include <linux/slab.h>
5 #include <linux/netdevice.h>
6
7 static unsigned long flags = 0;
8
9 static struct spi_device *etx_spi_device;

```

```

10
11 static struct spi_board_info etx_spi_device_info = //information
    about slave device
12 {
13     .modalias      = "esp32_driver",
14     .max_speed_hz  = 40000000,                // speed your device (slave)
        can handle
15     .bus_num       = SPI_BUS_NUM,            // SPI 2 (della som)
16     .chip_select   = 0,                      //
17     .mode          = SPI_MODE_1              // SPI mode 1, cause esp32
        can work with DMA with this mode
18 };
19
20 void etx_spi_read_struct(void * struct_to_receive, uint8_t size){
21
22     if( etx_spi_device )
23     {
24         struct spi_transfer tr =
25         {
26             .tx_buf  = NULL,
27             .rx_buf  = struct_to_receive ,
28             .len      = size ,
29             .bits_per_word = 8,
30         };
31         int value = 0;
32         int check = 1;
33         while( check)
34         {
35             msleep(1);
36             value = gpio_get_value(HANDSHAKE_NUM);
37             if( value){
38                 spi_sync_transfer( etx_spi_device , &tr , 1 );
39                 check = 0;
40                 break;
41             }
42         }
43     }
44     //memcpy( buff , rx , 6);
45 }
46
47 int etx_spi_read32(char * buff)
48 {
49     int ret = -1;
50     char rx[6];
51
52     if( etx_spi_device )
53     {
54         struct spi_transfer tr =
55         {

```

```
56     .tx_buf  = NULL,
57     .rx_buf  = rx,
58     .len     = sizeof(rx),
59     .bits_per_word = 8,
60 };
61 int value = 0;
62 int check = 1;
63 while(check)
64 {
65     msleep(1);
66     value = gpio_get_value(HANDSHAKE_NUM);
67     if(value){
68         spi_sync_transfer( etx_spi_device, &tr, 1 );
69         check = 0;
70         break;
71     }
72 }
73 }
74 memcpy(buff, rx, 6);
75 return 0;
76 }
77
78 int send_rx_packet(int lunghezza, char *dati_da_inviare){
79
80     struct spi_transfer tr =
81     {
82         .tx_buf  = &lunghezza,
83         .rx_buf  = NULL,
84         .len     = 32,
85         .bits_per_word = 8,
86     };
87     int value = 0;
88     int check = 1;
89     while(check)
90     {
91         msleep(1);
92         value = gpio_get_value(HANDSHAKE_NUM);
93         if(value){
94             spi_sync_transfer( etx_spi_device, &tr, 1 );
95             check = 0;
96             break;
97         }
98     }
99
100
101     tr.tx_buf  = dati_da_inviare;
102     tr.rx_buf  = NULL;
103     tr.len     = lunghezza;
104 }
```

```
105     value = 0;
106     check = 1;
107     while(check)
108     {
109         msleep(1);
110         value = gpio_get_value(HANDSHAKE_NUM);
111         if(value){
112             spi_sync_transfer( etx_spi_device, &tr, 1 );
113             check = 0;
114             break;
115         }
116     }
117     return 0;
118 }
119
120 int etx_spi_write_cmd(cmd_spi cmd_){
121     char tx[4] = {(char)cmd_,0,0,0};
122     struct spi_transfer tr =
123     {
124         .tx_buf = tx,
125         .rx_buf = NULL,
126         .len = sizeof(tx),
127         .bits_per_word = 8,
128         //.delay_usecs = 10000
129     };
130     int value = 0;
131     int check = 1;
132     while(check)
133     {
134         msleep(1);
135         value = gpio_get_value(HANDSHAKE_NUM);
136         if(value){
137             spi_sync_transfer( etx_spi_device, &tr, 1 );
138             check = 0;
139             break;
140         }
141     }
142     return 0;
143 }
144
145 int etx_spi_write_size(int len){
146     char tx[4] = {(char)len,0,0,0};
147     struct spi_transfer tr =
148     {
149         .tx_buf = tx,
150         .rx_buf = NULL,
151         .len = sizeof(tx),
152         .bits_per_word = 8,
153         //.delay_usecs = 10000
```

```

154     };
155     int value = 0;
156     int check = 1;
157     while(check)
158     {
159         msleep(1);
160         value = gpio_get_value(HANDSHAKE_NUM);
161         if(value){
162             spi_sync_transfer( etx_spi_device , &tr , 1 );
163             check = 0;
164             break;
165         }
166     }
167     return 0;
168 }
169
170 int handshake(int gpio){
171
172     if(gpio_request(gpio , "rpi-gpio")) {
173         printk("Error!\nCan not allocate GPIO %d\n", gpio);
174         return -1;
175     }
176
177     if(gpio_direction_input(gpio)) {
178         printk("Error!\nCan not set GPIO %d to input!\n", gpio);
179         return -1;
180     }
181
182     return 0;
183 }
184
185 int etx_spi_init(void)
186 {
187     int ret;
188     struct spi_master *master;
189
190     handshake(HANDSHAKE_NUM);
191     //irq_configuration(IRQ_NO);
192
193     gpio_request(IRQ_NO, "aa");
194     gpio_direction_output(IRQ_NO, 0);
195     gpio_direction_output(IRQ_NO_CMD, 0);
196
197
198     master = spi_busnum_to_master( etx_spi_device_info.bus_num );
199     if( master == NULL )
200     {
201         pr_err("SPI Master not found.\n");
202         return -ENODEV;

```

```

203 }
204
205 // create a new slave device, given the master and device info
206 etx_spi_device = spi_new_device( master, &etx_spi_device_info );
207 if( etx_spi_device == NULL )
208 {
209     pr_err("FAILED to create slave.\n");
210     return -ENODEV;
211 }
212
213 // 8-bits in a word
214 //etx_spi_device->bits_per_word = 8;
215 // setup the SPI slave device
216 ret = spi_setup( etx_spi_device );
217 if( ret )
218 {
219     pr_err("FAILED to setup slave.\n");
220     spi_unregister_device( etx_spi_device );
221     return -ENODEV;
222 }
223
224 pr_info("SPI driver Registered\n");
225 return 0;
226 }
227
228 void etx_spi_exit( void )
229 {
230     if(etx_spi_device)
231     {
232         // Clear the display
233         spi_unregister_device( etx_spi_device );    // Unregister the SPI
234         gpio_free(HANDSHAKE_NUM);
235         //free_irq(IRQ_NO, NULL);
236         gpio_free(IRQ_NO);
237         gpio_free(IRQ_NO_CMD);
238         pr_info("SPI driver Unregistered\n");
239     }
240 }
241
242 static irqreturn_t irqHandler(int irq, void *data){
243     local_irq_save( flags );
244     pr_err( "SONO NELL' HANDLER\n" );
245     local_irq_restore( flags );
246     return IRQ_HANDLED;
247 }
248 void irq_configuration(int pin){
249     static short int irqNumber = 0;
250     int result;

```

```

251 handshake(pin);
252 irqNumber = gpio_to_irq(pin);           // map GPIO to an IRQ
253 result = request_irq(irqNumber,         // requested interrupt
254                     irqHandler,         // pointer to handler
255                     function            IRQF_TRIGGER_RISING, // interrupt mode flag
256                     "irqHandler",       // used in /proc/
257                     interrupts          NULL);               // the *dev_id shared
258 interrupt lines, NULL is okay
259 pr_err("IL NUMERO DI IRQ E': %d\n", irqNumber);
260 }

```

A.0.3 netdevice.c

```

1 #include "cfg80211esp.h"
2 #include "netdevice.h"
3 #include "espspi.h"
4
5 void esp_setup(struct net_device *esp_devv){
6
7     ether_setup(esp_devv);
8
9     esp_devv->flags |=IFF_NOARP;
10    esp_devv->flags |=IFF_MULTICAST;
11    //esp_devv->flags &= ~IFF_MULTICAST;
12    esp_devv->priv_flags |= IFF_LIVE_ADDR_CHANGE | IFF_NO_QUEUE;
13    esp_devv->features |= NETIF_F_SG | NETIF_F_FRAGLIST;
14    esp_devv->features |= NETIF_F_ALL_TSO;
15    esp_devv->features |= NETIF_F_HW_CSUM | NETIF_F_HIGHDMA |
16    NETIF_F_LLTX;
17    esp_devv->features |= NETIF_F_GSO_ENCAP_ALL;
18
19    esp_devv->hw_features |= esp_devv->features;
20    esp_devv->hw_enc_features |= esp_devv->features;
21
22    esp_devv->min_mtu = 0;
23    esp_devv->max_mtu = 0;
24    return;
25 }

```

A.0.4 cfg80211esp.h

```

1 #include <net/cfg80211.h> /* wiphy and probably everything that would
   required for FullMAC driver */
2 #include <linux/skbuff.h>
3
4 #include <linux/workqueue.h> /* work_struct */
5 #include <linux/semaphore.h>
6
7 #include <linux/module.h>
8 #include <linux/kernel.h>
9 #include <linux/netdevice.h>
10 #include <linux/etherdevice.h>
11 #include <linux/init.h>
12 #include <linux/moduleparam.h>
13 #include <linux/rtnetlink.h>
14 #include <net/rtnetlink.h>
15 #include <linux/u64_stats_sync.h>
16
17
18 #ifndef _cfg80211esp_h_
19 #define _cfg80211esp_h_
20
21
22 #define WIPHY_NAME "enrico"
23 #define NDEV_NAME "enrico%d"
24 #define SSID_DUMMY "BitronDevices"
25
26 // #define SSID_DUMMY_SIZE (sizeof("NETGEAR17_EXT") - 1)
27
28 #define CHAN2G(_channel, _freq, _flags) { \
29     .band = NL80211_BAND_2GHZ, \
30     .center_freq = (_freq), \
31     .hw_value = (_channel), \
32     .flags = (_flags), \
33     .max_antenna_gain = 0, \
34     .max_power = 30, \
35 }
36
37 static struct ieee80211_channel nvf_supported_channels_2ghz[] = {
38     CHAN2G(1, 2412, 0),
39     CHAN2G(2, 2417, 0),
40     CHAN2G(3, 2422, 0),
41     CHAN2G(4, 2427, 0),
42     CHAN2G(5, 2432, 0),
43     CHAN2G(6, 2437, 0),
44     CHAN2G(7, 2442, 0),
45     CHAN2G(8, 2447, 0),
46     CHAN2G(9, 2452, 0),
47     CHAN2G(10, 2457, 0),

```



```

48     CHAN2G(11, 2462, 0),
49     CHAN2G(12, 2467, 0),
50     CHAN2G(13, 2472, 0),
51     CHAN2G(14, 2484, 0)
52 };
53
54 #define RATETAB_ENT(_rate, _hw_value, _flags) { \
55     .bitrate = (_rate),           \
56     .hw_value = (_hw_value),      \
57     .flags    = (_flags),         \
58 }
59
60 static struct ieee80211_rate nvf_supported_rates_2ghz[] = {
61     RATETAB_ENT(10, 0, 0),
62     RATETAB_ENT(20, 1, 0),
63     RATETAB_ENT(55, 2, 0),
64     RATETAB_ENT(110, 3, 0),
65     RATETAB_ENT(60, 9, 0),
66     RATETAB_ENT(90, 6, 0),
67     RATETAB_ENT(120, 7, 0),
68     RATETAB_ENT(180, 8, 0),
69     RATETAB_ENT(240, 9, 0),
70     RATETAB_ENT(360, 10, 0),
71     RATETAB_ENT(480, 11, 0),
72     RATETAB_ENT(540, 12, 0)
73 };
74
75
76 /* Structure that describes supported band of 2ghz. */
77 static struct ieee80211_supported_band nf_band_2ghz = {
78     .ht_cap.cap = IEEE80211_HT_CAP_SGI_20, /* add other band
79     capabilities if needed, like 40 width etc. */
80     .ht_cap.ht_supported = false,
81     .channels = nvf_supported_channels_2ghz,
82     .n_channels = ARRAY_SIZE(nvf_supported_channels_2ghz),
83     .bitrates = nvf_supported_rates_2ghz,
84     .n_bitrates = ARRAY_SIZE(nvf_supported_rates_2ghz),
85 };
86
87 struct esp_packet {
88     struct esp_packet *next;
89     struct net_device *dev;
90     int datalen;
91     u8 data[ETH_DATA_LEN];
92 };
93
94 struct esp_context {
95

```

```

96     struct wiphy *wiphy;
97     struct net_device *ndev;
98
99     struct semaphore sem;
100    struct work_struct ws_connect;
101    struct work_struct send_packet;
102    char connecting_ssid[33];
103    char connecting_bssid[6];
104    struct work_struct ws_disconnect;
105    u16 disconnect_reason_code;
106    struct work_struct ws_scan;
107    struct cfg80211_scan_request *scan_request;
108    struct net_device_stats stats;
109    struct napi_struct napi;
110    int status;
111    struct esp_packet *ppool;
112    struct esp_packet *rx_queue;
113    u8 data[ETH_DATA_LEN];
114    int rx_int_enabled;
115    int tx_packetlen;
116    u8 *tx_packetdata;
117    struct sk_buff *skb;
118    spinlock_t lock;
119 };
120
121 struct esp_wiphy_priv_context {
122     struct esp_context *navi;
123 };
124
125 struct esp_ndev_priv_context {
126     struct esp_context *navi;
127     struct wireless_dev wdev;
128 };
129
130 void send_packet_to_esp(char *data, int len, struct net_device *dev);
131
132 #endif

```

A.0.5 espspi.h

```

1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/gpio.h>
4 #include <linux/spi/spi.h>
5 #include <linux/delay.h>
6 #include <linux/interrupt.h>

```

```

7
8 #ifndef _espspi_h_
9 #define _espspi_h_
10
11 typedef struct __attribute__((__packed__)) {
12     uint8_t bssid[6];           /**< MAC address of AP */
13     uint8_t ssid[33];          /**< SSID of AP */
14     uint8_t primary;           /**< channel of AP */
15     int8_t rssi;
16 } wifi_ap_record_t_send;
17
18 typedef enum {
19     READ_MAC,
20     MAKE_SCAN,
21     TX_PACKET_LINUX
22 } cmd_spi;
23
24 #define HANDSHAKE_NUM      10
25 #define IRQ_NO             72
26 #define IRQ_NO_CMD        24
27 #define SPI_BUS_NUM       2
28
29 int etx_spi_init(void);
30 int etx_spi_write_size(int len);
31 int etx_spi_write_cmd_irq(cmd_spi cmd_);
32 int etx_spi_read32(char * buff);
33 int send_rx_packet(int lunghezza, char *dati_da_inviare);
34 void etx_spi_exit(void);
35 int etx_spi_write_cmd(cmd_spi cmd_);
36 void etx_spi_read_struct(void * struct_to_read, uint8_t size);
37 static irqreturn_t irqHandler(int irq, void *data);
38 void irq_configuration(int pin);
39
40 #endif

```

Bibliography

- [1] Yangjian, 2017, *Development of ralink's wireless network interface card based on uclinux*.
- [2] Yongxiang Guo, Wu Deng, 2010, *Design of network device driver in embedded Linux*.
- [3] Y. Imai, T. Yokouchi, H. Inomo, W. Shiraki, H. Ishikawa, 2004, *A Linux-based engineering education with hardware implementation, device drivers' programming and network literacy learning*.
- [4] IEEE, 802.11-2012 - *IEEE Standard for Information technology-Telecommunications and information exchange between systems Local and metropolitan area networks-Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*.
- [5] *Networking Wi-Fi APIs Guide Espressif*, <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html>.
- [6] *Networking APIs Refernces Espressif*, https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_wifi.html.
- [7] *Wi-Fi Event Description Espressif*, <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.htmlesp32-wi-fi-event-description>.
- [8] *Wi-Fi Error Code Description Espressif*, <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.htmlesp32-wi-fi-api-error-code>.
- [9] *An Introduction to Device Drivers*, <https://static.lwn.net/images/pdf/LDD3/ch01.pdf>
- [10] *Linux cfg80211 subsystem*, <https://www.kernel.org/doc/html/v4.12/driver-api/80211/cfg80211.html>.
- [11] *FreeRTOS*, <https://it.wikipedia.org/wiki/FreeRTOS>.
- [12] *SPI Guide*, <https://www.unisalento.it/documents/20152/804790/SLIDES+LEZIONE+7+3Dispensa+SPI+Interface.pdf/>
- [13] *FreeRTOS Guide*, https://freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf
- [14] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman *Linux Device Drivers, Third Edition*,

https://freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf