INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Portierung und Entwicklung eines CC3200 Wi-Fi Treibers für RIOT-OS

Wladislaw Meixner

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Portierung und Entwicklung eines CC3200 Wi-Fi Treibers für RIOT-OS

Wladislaw Meixner

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Tobias Guggemos
Abgabetermin:	17. Oktober 2019

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 17. Oktober 2019

. . . . . .
(Unterschrift des Kandidaten)

Abstract

In the current day and age Wi-Fi is everywhere. With developments to manufacturing cost and efficiency, Wi-Fi has become a more viable option for IoT systems. These systems offer the benefit of using preexisting IEEE 802.11 infrastructure commonly found in households and businesses. Unfortunately, most if not all such embedded systems rely on closed source driver implementation potentially limiting the utility, security, extensibility and future proofing of these products. RIOT Operating System (OS), a popular IoT OS,provides integration for a number of networking protocols, but lacks this omnipresent IEEE 802.11 standard. In this thesis the possibility of a RIOT based open source IEEE 802.11 driver for an embedded systems is evaluated on the specific Texas Instruments CC3200-launchxl board. The process of adding new hardware, Central Processing Unit (CPU) and board abstraction layers is conducted. A new RIOT Generic network stack (GNRC) IEEE 802.11 layer is added and the communication between the Microcontroller (MCU) and its network co-processor is examined during the creation of the driver. The mostly software based driver strifes to provide a MAC Layer socket, which is then connected to RIOTs general network stack. The goal of this thesis consists of three distinct components. A port of the CC3200 board to the RIOT OS, a driver allowing for Wi-Fi communication on the MAC Layer and an IEEE 802.11 RIOT extension allowing for rudimentary frame parsing.

Contents

1 Introduction

The imagine of a world where smartphones require a wired connection to surf the internet or check the weather sounds highly obnoxious in the current day and age. While wired connections have their use cases, an ever growing rate of network traffic is being transmitted by portable wirelessly connected devices. With advancements in power efficiency and decrease of production costs, Internet of Things (IoT) devices are also becoming a popular addition to this traffic. Due to the highly error and interference prone nature of wireless transmissions, as well as the growing number of "Things" sending on the narrow range of allowed frequencies, technological and regulatory steps must be taken to sustain this trend in the future. This need for additional bandwidth, to handle the increased demand, drives the amount of frequency bands assigned to wireless communications.These developments influence future standards as the 5th Generation Mobile Networking, with its numerous frequency band additions, to combat network congestions. Many households and businesses today rely on the common IEEE 802.11 or Wi-Fi standard, allowing for untethered networking.
With the Wi-Fi infrastructure already in place, the quests arises, whether the omnipresent infrastructure can be used by IoT devices. The current boundaries to the using of IEEE 802.11 in this context are hardware cost and power efficiency. Both of these become more and more tangible with the ongoing advancements in technology. The software standard itself, the IEEE 802.11 protocol, is being extended continuously to offer not only improved performance but also increased reliability and efficiency.
Numerous vendors now provide Wi-Fi enabled low power solutions. These systems could open a window for IoT into the IEEE 802.11 based networking. The adoption of these technologies strongly depends on the availability of necessary software and hardware tools. Providing a unified abstraction layer for these systems could prove essential for their practicality.

1.1 Motivation

The advantages of IEEE 802.11 in the context of IoT are evident, direct network connection to preexisting infrastructure. The adoption of these technologies is currently dampened by the absence of a accessible, flexible software stack. Many of the current IoT capable products rely on proprietary drivers, Software Development Kit (SDK)s and operating systems. RIOT OS, a currently popular open source operating system, provides a wide ranging support for boards and networking stacks. It also aims at unifying the hardware usage for low powered embedded systems via its abstraction layers. While RIOT offers numerous specialized radio protocol implementation IEEE 802.11 is currently surprisingly absent from the OS. Adding IEEE 802.11 to RIOT would allow hundreds of devices to utilize this technology, via already embedded hardware or extension modules. With this, many IoT devices could be empowered to utilize this omnipresent technology. The requirement for a IEEE 802.11 addition to RIOT is apparent. With the increase in IoT application development, testing of diverse multi hardware and protocol systems becomes important. The implementation

of Wi-Fi functionalities provides a welcome addition to current IoT testbed projects. The CC3200 platform could be added to these systems, allowing for Wi-Fi development and testing. Furthermore, IEEE 802.11 and its IEEE 802.11s mesh extension provide a viable combination for group communication and encryption procedures, allowing for secure diverse network topologies. Wi-Fi in these cases would additionally allow IoT devices to communicate directly with other network hardware without the need for intermediary hardware, as is the case with 802.15.4 or other proprietary communication solutions.
The goal of this thesis is to enable Wi-Fi support for RIOT. Implementing a new networking protocol without a board using it would prove ineffectual. RIOT OS currently has support for the ESP8266, a Wi-Fi capable board. Unfortunately this board limits direct access to the network co-processor making its use for the endeavour problematic. Therefore the SimpleLink family Texas Instrument CC3200 board was selected. It features a socket implementation close to Portable Operating System Interface (POSIX), allowing for direct MAC Layer communication over Wi-Fi. The resulting task can be divided into three subtasks. Firstly the CC3200 must be ported to RIOT OS, since it is currently not supported by the operating system. Secondly a IEEE 802.11 implementation must be added to RIOTs networking stack and lastly a driver must be created capable of both, communicating with the CC3200 hardware and the networking stack of RIOT.

1.2 Outline

This thesis elaborates the steps required to introduce a new board family, implement a new hardware driver and integrate the IEEE 802.11 into RIOT OSs GNRC. The first chapter provides essential background knowledge about Wi-FI, RIOT and the landscape of current Wi-Fi capable MCUs. The IEEE 802.11 introduction will cover the PHY and MAC layer as specified by the IEEE and introduce the key data structures, later used in the concept. The third chapter then outlines the general approach and concept, without focusing too much on the hardware itself. In this chapter the driver integration and RIOT OS specific communication interfaces and technologies are described. The forth chapter provides a detailed implementation guide for the three main components of this endeavour, board, driver and IEEE 802.11 integration. The board implementation covers CPU specific vector table configurations as well as essential periphery abstraction layer, necessary for RIOTs operation and the driver itself. These include timers, Serial Peripheral Interface (SPI) and General Purpose Input Output (GPIO). The methodology of transferring the executable code to the CC3200 is additionally demonstrated. The coverage driver encompasses the communication protocol utilized for Network Processor (NWP) to host communication and shows how this can be utilized to provide a general purpose socket on Medium Access Control (MAC) layer 2. Lastly the essential IEEE 802.11 additions are performed to RIOTs GNRC and other modules. This extension forms a communication layer between the driver and RIOTs GNRC. In chapter five the implementation in terms of functionality is evaluated and a comparison to the previously created concept is provided. Furthermore it also discusses the limitations of the proposed solution. Lastly, the thesis is concluded with a brief outlook on the work as a whole, the problems encountered during the implementation and the potential for future development in this field.

2 Background

The current MCU environment is diverse, with a multitude of products offering low power IEEE 802.11 support for embedded systems. This chapter aims at providing a broader overview of the core protocol in this thesis, IEEE 802.11. Essential concepts of the standard are introduced, providing the necessary knowledge for the proposed concept and the later implementation. The IEEE 802.11 standard has a large number of extensions, covering mash networking, beam forming and other advanced features. These additional functionalities are omitted for simplicity and are only briefly mentioned when appropriate, to provide a more complete picture while maintaining a clear focus. Following the introduction of IEEE 802.11, the currently available Wi-Fi enabled MCUs, their software stacks and feature sets are listed and briefly compared. Lastly RIOT OS, the operation system of choice for this implementation, is introduced.

2.1 IEEE 802.11

The IEEE 802.11 is a ever expanding standard family. Its development driven by the rising demands for fast, efficient, reliable and efficient wireless connectivity. Wi-Fis peak theoretical bandwidth has increased from the original 2 mega bits per second by a factor of over 5000, between its first inception in 1997 and the latest 802.11ax (Wi-Fi 6) revision [Kho+19]. Wi-Fis history, its present and the direction it is heading with extensions currently in development is shown in the next chapter.

2.1.1 IEEE 802.11 naming scheme

Wi-Fi and IEEE 802.11 are often used interchangeably, but there are subtle difference. Therefore, the naming should be clarified. Wi-Fi is a branding term controlled by the Wi-Fi Alliance and used for general consumer marketing [09b]. The Wi-Fi Alliance itself is a non-profit organization, tasked with marketing and certification duties for Wi-Fi technologies on global scale. The Wi-Fi Alliance also holds trademark rights to the Wi-Fi brand and marketing material [19a]. Until 2019 the Wi-Fi Alliance used the IEEE protocol naming scheme for Wi-Fi versioning e.g. the common 802.11b/g/n. But starting with 2018 the naming was updated to provide a more human friendly notation. While an some people may know that 802.11ax refers to a newer protocol version then 802.11ac, this is unreasonable to expect from the general public. Therefore new products will market 802.11ax as Wi-Fi 6. Only major Wi-Fi revisions will be enumerated. This naming will also cover past protocol versions starting with 802.11n becoming Wi-Fi 4 [18b]. As previously mentioned the Wi-Fi branding is synonymous to major releases of the 802.11 standard. What conducts a major release is decided by the Wi-Fi Alliance. The base protocol IEEE 802.11 is a standard defined by the Institute of Electrical and Electronics Engineers (IEEE) and is in itself composed of the designation "802" to denote its networking role and family "11" for Wireless Local Area Network (WLAN). The protocol is continuously updated with new features and

improvements by numerous task groups in the form of extensions. These extensions are assigned a lettered codename and in some cases a human readable name as is the case for 802.11ax also being called "High Efficiency Wireless". The protocol does not guarantee backwards compatibility meaning that, for example devices only listing 802.11b protocol support will not operate on a 802.11a Access Point (AP) if not otherwise indicated. 802.11n networks on the other hand provide interoperability with 802.11a/b/g clients but will not perform in the optimal "Greenfield" mode, reducing general network performance for other 802.11n capable devices on the same network [09a]. Therefore, the compatibility is defined on a per extension basis.

2.1.2 IEEE 802.11 development

The wireless standard was initially intended to be a Physical (PHY) layer extension to one of IEEEs already available networking protocols, for example the well known IEEE 802.3 ethernet standard. As research in the field of WLAN progressed, it became clear that the existing protocols did not suffice the requirements for a sufficiently high throughput communication. Attenuation of the signal prohibited the usage of IEEE 802.3's carrier sense multiple access with collision detection (CSMA/CD) even for short distances transmissions. This fact lead to the switch from 802.3 to a coordination medium access based 802.4 protocol family [Hie+10]. While this token based medium access policy has proven to be superior to CSMA/CD, token based wireless communication was shown to be difficult to implement in practice. Based on this research a new project was started on the 21st of March 1991, IEEE 802.11. This protocol would not only provide a PHY layer specification but also have its own MAC layer, explicitly developed for radio communications based on Distributed Coordinated Function (DCF).
IEEE 802.11 was officially released in 1997 and is today referred to as IEEE 802.11-1997 or IEEE 802.11 legacy to avoid naming ambiguity. This early version allowed for up to 1-2 Mbps MAC layer speeds to be achieved over the Industrial Scientific Medical (ISM) frequency band at around 2.4 Ghz in an optimal environment and. Curiously, this standard additionally specifies a infrared transmission with equal speed, this standard was never implemented while being part of official IEEE 802.11-1997 specification [Val+98].
After two years a new revision of the protocol was released IEEE 802.11b featuring an increased theoretical maximal bandwidth of 11 Mbps and introducing the todays channel system. The addition of channels improved network consistency and decreased package loss for access points in close proximity to each other. The Wi-Fi standard is being updated and improved to this day. Wi-Fi 6 or 802.11ax being the newest revision incorporated into devices, already shipping to consumers in the middle of 2019. This new protocol extends the bands from 2.4Ghz and 5Ghz to the full ISM range of 1 to 7 Ghz. Although a transition period is to be expected, while other devices currently operating on these bands, will need time to adapt to this new traffic. While the main goal of this iteration as implied by its name, High Efficiency Wireless, is to offer improved latency and efficiency, bandwidth has still increased due to the expansion in available frequency ranges. Modern multi antenna devices can receive or transmit on multiple channels and frequencies at once. Wi-Fi 6 advertises 10 Gbps as its theoretical maximal data rate [Kho+19].

2.1.3 PHY Layer Data Transmission

The Wi-Fi standard can easily be mapped on to the OSI-Layer model. Wi-Fi defines two Layers, namely the PHY Layer performing the physical transmission, antenna configuration and the MAC responsible for congestion control and network management. Naturally both layers also handle transmission of data. Since the original standard multiple extensions to this simple model were made, most of these additions are not covered by the background chapter of this thesis. This chapter instead will focus on the core definitions proposed by IEEE 802.11b and to some degree by IEEE 802.11-1997 to deliver an overview. IEEE 802.11b as well as 802.11g/n/ax operate on the 2.4GHz frequency range, starting at 2412 MHz to 2484 MHz, other versions like 802.11a/h/j/n/ac/ax additionally define a available 5GHz range. Both spectrums are divided into channels. The 5GHz spectrum is highly region dependant in regards to the available channels. To keep the definitions simple only the 2.4GHz band will be discussed but most concepts are shared between all of the Wi-Fi bands, excluding specific frequency ranges, channel sizes and timings. Since radio frequency usage and licensing is subject to national or regional legislature not all 14 defined Wi-Fi channels can be utilized in every country. This is mostly due to other civilian or military radio applications already operating on these frequencies. To guarantee transmission reliability each channel in the 2.4GHz band is required to be separated by at least 16.25 to 22MHz and a two 2 MHz guard band on the upper and lower edges of the channel, thus providing adequate interference prevention. The PHY layer is itself divided into Physical Medium Dependent (PMD) sublayer actually transmitting the bits over the air and the Physical Layer Convergence Procedure (PLCP) operating in terms of PHY layer packets or PLCP Protocol Data Unit (PPDU). The PPDU packets consist of a PLCP preamble, PLCP header and the actual payload. The preamble is mainly used to separate PPDUs from radio background noise and indicate the beginning of a data burst. The PPDU includes physical layer configuration of the frame as for example signal modulation method and data length [03, Chapter 18].

2.1.4 MAC Layer and CSMA/CD

Like most shared medium communication system Wi-Fi is subject to interference with or without the presence of Wi-Fi devices. To combat these medium congestions DCF is used. DCF is based upon CSMA/CD to minimize collision probability between multiple Wi-Fi Stations (STA) operating on overlapping channels. CSMA/CD assumes that the highest probability of collision is just after the medium becomes idle following a prolonged occupational period. Wi-Fi MAC. Firstly to overcome these situations a random back-off is introduced to prohibit senders from retransmitting at the same delay after a collision, leading to a transmission deadlock. Before transmitting all Wi-Fi devices monitor ongoing traffic and await radio silence. Transmission is only performed after a DCF inter-frame space (DIFS) long gap and an optional back-off time passes without other radio traffic reported by the PHY layer [16]. The back-off time is a multiple of the time slot defined by the IEEE 802.11 standard. The back-off time is chosen randomly from a range window, the window size depends on the number of failed transmissions. The range used to determine the back-off is called Contention Window (CW). This range is normally doubled after every failed attempt until it reaches a predefined maximum value. The back-off time is reset after a successful transmission. All Wi-Fi frames are postfixed with 32 bit cyclic redundancy check (CRC)

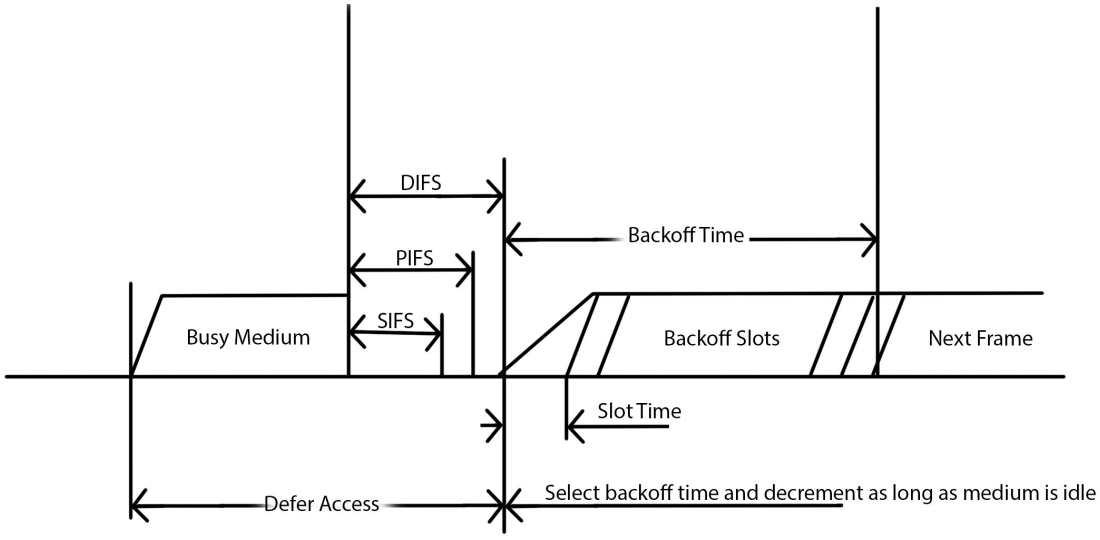

Figure 2.1: Medium access mechanism (adapted from [16])

error-correction code to allow the receiver to verify its integrity. This CRC is referred to in the context of Wi-Fi as Frame check sequence (FCS). Upon receiving a frame the receiver validates its integrity first. A valid frame is acknowledged by an ACK frame. The ACK frame must be send by the receiver within the Short inter-frame space (SIFS) time window, following the last symbol of the send data packet. If the frame was damaged during transmission and cannot be corrected using the FCS, no ACK is transmitted. If no ACK was received after the SIFS period a retransmission is performed. This increases the CW and by that the probability of a longer back-off time. This repeated sending can be performed a variable k_{max} number of times, the default value of 7 is common in consumer products. If after k_{max} retries were performed and the package has not reached its destination successfully, the transmission of the packet is cancelled. In newer versions of Wi-Fi this algorithm is extended with shorter versions of the DIFS to enforce prioritized traffic like management frames. This essentially causes all devices waiting DIFS duration before sending to defer their medium access. The principle of the DIFS and the back-off time can be observed in the illustration 2.1, additionally the image includes the shorter wait periods point coordination function interframe space (PIFS) and SIFS.

2.1.5 MAC Layer Addresses

Communication requires identification, without the ability to know who send a message direct device to device communication is not possible. The wireless medium has no restriction on the actors transmitting data on a given frequency. This requires an address system that guarantees unique device identification to a high probability. Because these addresses must

Table 2.1: MSDU Header address field contents

To DS	From DS	Address 1	Address 2	Address 3	Address 4
0	0	RA = DA	TA = SA	BSSID	-
0	1	RA = DA	TA = BSSID	SA	-
1	0	BSSID	TA = SA	DA	-
1	1	RA	TA	DA	SA

RA Receiver Address, **DA** Destination Address, **TA** Transmitter Address
SA Source Address, **BSSID** Basic Service Set IDentifier

also be available before data exchange with parties can commence, they must be generated on the device itself. Therefore, the 48-Bit Extended Unique Identifier (EUI-48) is defined as part of the IEEE 802.11 specification. The EUI-48 address was previously called MAC-48 but this terminology is now deemed obsolete by the IEEE and its use is discouraged [17]. This address is a six octet long identifier composed from two equally sized components, the Organizationally unique identifier (OUI) assigned by the IEEE and a Network interface controller (NIC) identifier. The NIC in most cases is set during manufactory and cannot be directly altered. It is still possible to transmit data using a different EUI-48 address, but the address stored in hardware cannot be removed easily. This address is used for MAC Layer communication and is an essential part of the 802.11 standard [IEEE Std 802a-2003, Chapter 8]. The MAC EUI address space has a number of addresses reserved by the IEEE and a broadcast address `FF:FF:FF:FF:FF:FF`.

2.1.6 IEEE 802.11 MAC Service Data Unit

The communication unit of the MAC Layer is the MAC Service Data Unit (MSDU). All MSDUs adhere to a basic structure. Every MAC frame begins with a header with a maximum length of 48 byte. The length can vary depending on the frame type and the Frame Control (FC) field. The FC field is 16 bit long and holds information about the content of the frame and its role. The FC starts with a version field which is currently zero and reserved for future use. The next six bits represent the full frame type. The type (bits 2 - 3) field denotes weather a frame is a Control, Management or Data Frame, the subtype (bits 4 - 7) clarifies the concrete type within that domain e.g. ACK frame. The FC is depicted in more detail in Fig. 2.2. The frame header also contains a maximum of four EUI-48 addresses. The contents of the address fields and their meaning is depicted with the relation to the fields of the FC in Table. 2.1

There are three frame types defined in IEEE 802.11 [16, Chapter 9] with numerous subtypes. These types are divided by function. Control Frames are all MSDUs used for medium access coordination. These Frames most importantly include the ACK frame, discussed as part of the MAC congestion control. As will be shown later Wi-Fi infrastructure also requires frames to manage device sleep activity states, authentication and association, this functionalities are handled by the Management Frames. Data Frames are lastly used for non management data transmissions on a IEEE 802.11 network. These MSDU types form the basis of MAC layer functionality.

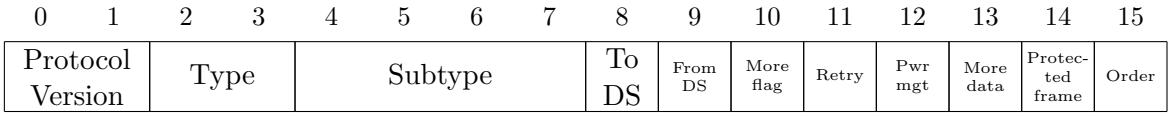

Figure 2.2: 802.11 FC Field

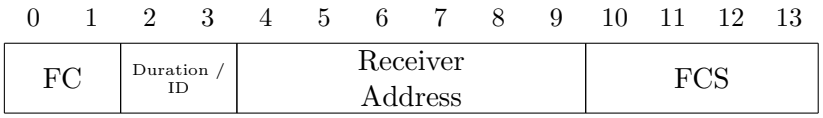

Figure 2.3: 802.11 ACK Frame

Control Frames

The Control Frames form the core for a reliable communication of multiple devices on the same wireless network. In newer extensions to the Wi-Fi standard these frames are used to form a layer of backwards compatibility. The most common frame is the ACK frame. The ACK does does not contain any payload as can be seen from its structure illustrated in Fig.2.3 The ACK is send after most data frame and some management frames to acknowledge their reception. The ACK requires 20 byte of data to transmit a single bit of information, therefore it is argued to be a major source of protocol overhead. This led to the introduction of BlockAck in newer MAC protocol revisions, allowing to acknowledge multiple frames at once reducing the protocol overhead [16, Chapter 9.3]. As part of 802.11 RTS/CTS protocol extension two more frames are categorizes to the Control Frames, Request to Send (RTS) frame and Clear to Send (CTS). These frames are used for an optional collision prevention mechanism, when utilized, any station requests permission to send before sending large data blocks.

Data Frame

The Data Frames transport all user payload carried via Wi-Fi. Each bit of the Data Frame subtype is used for a specific frame extension. Currently there are four subtype groups that are not mutually exclusive and can be combined within a single Data Frame: Contention Free (CF)-ACK,CF-Poll, No-Data and Quality of Service (QoS). The absence of any of those groups, a subtype of 0 (0000), denotes a basic data frame. The non basic frame subtypes are used for frame prioritization and combining ACK frames with Data Frames to reduce general Wi-Fi overhead and improve latency for applications like Voice Over IP (VoIP)

The layout of a basic IEEE 802.11 Data Frame is illustrated in Fig. 2.4. A IEEE 802.11 frame is composed of 32 byte MAC Header, optional variable length payload and a 4 byte FCS checksum. The payload depending on the subtype can also contain additional protocol specific information prefixing the actual payload. In a managed Wi-Fi network Data Frames can only be send after a successful association of the client with an AP. Meaning that any frames send from a non associated party will be ignored.

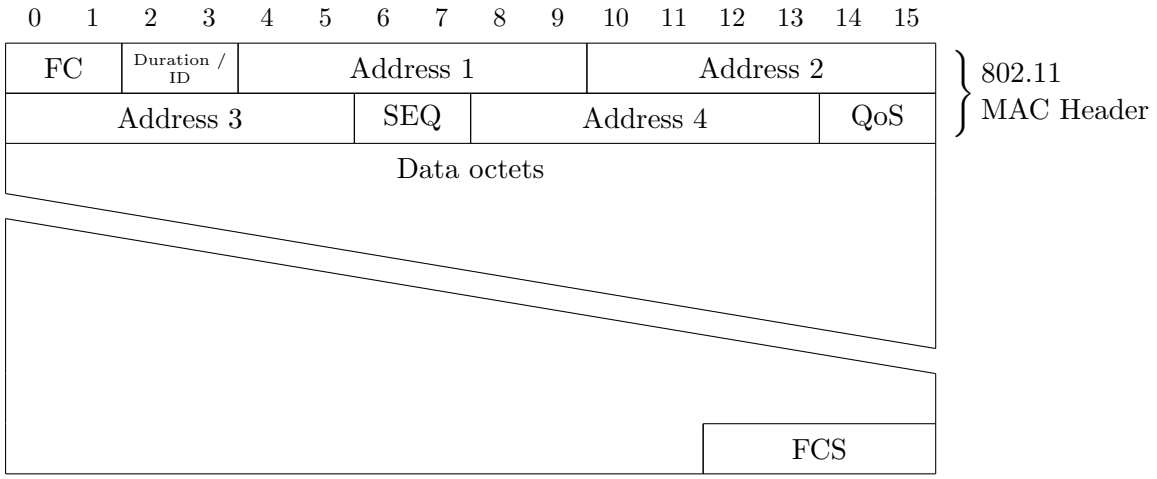

Figure 2.4: 802.11 Data Frame

Management Frame

The Management Frame is used to coordinate communication between associated wireless devices in their domain. The IEEE 802.11 standard and its amendments specify multiple topologies. The management of these networks, the association, authentication and other tasks are executed via the Management Frame types. Commonly a group of communicating IEEE 802.11 devices can also be referred to as a Service Set.

2.1.7 802.11 Service Set

A Service Set is a group of Wi-Fi devices operating with a common configuration. Commonly IEEE 802.11 Service Sets operate in hierarchical arrangements with a single root node. This hierarchical configuration is referred to as infrastructure mode. In addition to infrastructure mode the devices can operate in independent mode, allowing for direct peer-to-peer topologies. For example Wi-Fi ad-hoc is one such topology. The default infrastructure mode is called Basic Service Set (BSS). In this topology a client station can only be connected to one AP at a time. BSS can be identified by their Basic Service set ID (BSSID), a 48-bit identifier analogous to the EUI-48 address [16, Chapter 4.3.1]. Depending on the AP hardware used the BSSID can be factory set or configurable. In addition to the BSSID an AP also has a configurable Service set ID (SSID). The SSID is commonly used to display a human readable name when searching or connecting to a 802.11 wireless network. This ID can if configured be broadcasted by the AP to advertise its availability. The ID itself is represented by a variable long UTF-8 encoded string. The maximum length however is 32 octets. To allow backwards compatibility a Wi-Fi devices must be able to handle arbitrary encoded values as the SSID. BSSID and the SSID are used to advertise and discover surrounding IEEE 802.11 networks. In a conventional BSS the AP operates as a bridge between the wirelessly connected devices (client stations on this context) and the rest of the network. Before any data transmission can occur a client must associate to a BSS.

2.1.8 Network discovery

Before connecting to a Service Set a client station requires information about surrounding WLAN networks. A client in this context is a station (STA) seeking connection to an available AP station. Even when a client was already connected to a BSS the discovery is required since some APs can dynamically transition channels. Additionally the BSS hardware or software configuration could have been altered leading to a different BSSID. To inquire this information the client station can wait for incoming beacon frames. Due to hardware limitation of the radio a client can only listen to a single channel at a time, requiring the device to periodically switch channels. This process can result in considerable delay for the client, since beacon frames are send at configurable time intervals and the device needs to monitor all channels, leading to a high probability that the client misses multiple beacon frames. Since the client does not actively request station information this discovery is called passive scanning. In contrast, when performing active scanning, the client station sends out a Probe Request frame with the destination address send to the broadcast address. The Probe Request includes the capabilities of the client e.g. supported transmission speeds and IEEE 802.11 extensions. The client station then stays on the channel and awaits a Probe Response frame for a predefined timeout (Probe Timeout). If no Probe Response Frame was received, the client station switches to a different channel and repeats the procedure. An AP responds with a Probe Response frame if the client is compatible with the BSS configuration. The Probe Response contains a set of configurations picked by the AP and the advertised SSID.
When the BSSID is known to the client STA a directed Probe Request can be transmitted. For this the client STA sends a Probe Request frame with the Destination Address (DA) field set to the BSSID. When broadcasting an AP can omit the SSID by setting the SSID length field to zero. This is called a "wildcard SSID", this way an AP can broadcast beacon frames without announcing its SSID. Additionally the client can send Probe Request with a "wildcard SSID" resulting in the associated BSS returning a list of provided SSIDs.

2.1.9 Basic Service Set Association

Before exchanging data a client station must be associated to a BSS. The first probing step of the association process is already covered as part of the network discovery. After a successful Probe Response the two devices have chosen a compatible configuration. The next step is to authenticate the clients identity to the AP. If the AP is configured for Open System Authentication operation, the client STA sends out an Authentication Request with its MAC and the AP responds with a Authentication Response frame containing either a success or error code. The AP can also enforce a Shared Key Authentication. As suggested by the name this method requires the a configured key to be manually input on both client and AP. This authentication method was historically used for the now obsolete Wired Equivalent Privacy (WEP) Shared Key Authentication. After a successful authentication the device is authenticated but not associated to a network. A single client can be authenticated to multiple APs allowing for a quick transitions between them. On the other hand a client can only be actively associated to a single AP and only a associated client can send Data Frames. So to transmit data the client must send an additional Association Frame to the AP. More recent encryption and authentication methods utilize dynamic keying and perform the authentication after a successful association. For this reason the Association Request can

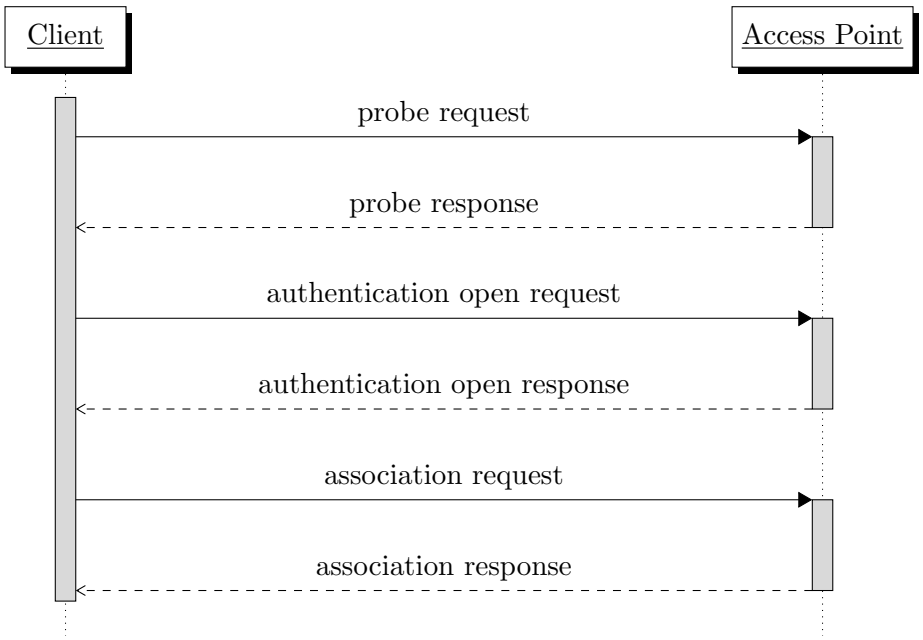

Figure 2.5: Association Process

also contain encryption configurations for the newer extensions, for example Wi-Fi Protected Access (WPA). Equally the AP matches advertised encryption capabilities provided by the client and responds with an Association Response frame containing a Association ID granting network access if the cypher suits are compatible. This encryption information is not included in the case of the Open System Authentication. [16, Chapter 4.5.4] The whole association process is visualized in Fig. 2.5.

2.2 Wi-Fi capable IoT devices and CC3200

2.2.1 Overview of IEEE 802.11 enabled low power microcontrollers

To provide a broader overview of the currently available MCU the following three boards will be briefly introduced and compared. The boards being CC3200, ESP8266 and ESP32. While the feature set varies significantly all support IEEE 802.11b/g/n. From the documentation it can be observed that all of these boards perform Wi-Fi operations off the host processor in a separate module. Additionally the host CPU and the network co-processor in all these boards is packed on a single System on a Chip (SoC).The Wi-Fi module in on all these platforms relies on communication interfaces like SPI or Universal asynchronous receiver-transmitter (UART). A further commonality of the reviewed boards is the undocumented network code operating on the network co processor. Some boards allow for a more low level access to the radio module while others limit the access or disable essential features when operating on lower OSI layers as will be discussed in the context of TIs CC3200 in future chapters. TIs board families are the only ones relying on closed source flashing methodology over commodity interfaces. A number of available open source reverse engineered flasher tools were tested and none were able to write files to the CC3200 filesystem. The Espressif board feature good OpenOCD support with the possibility flash the devices. All of the

Table 2.2: Comparison of Wi-Fi enabled MCUs (as specified by vendor)

MCU / Values	CC3200	ESP8266	ESP32
CPU	ARM® Cortex-M4	Tensilica L106	Xtensa single-/dual-core LX6
Arch	32bit	32bit	32bit
5Ghz	-	-	-
Clock Speed	80 MHz	80/160 MHz	240 MHz
RAM	128KB/256KB	64+96KB	520 KB
ROM	-	64 KB	448 KB
DMA	yes	-	yes
OpenSource Lib.	-	ESP8266WiFi	ESP32Wifi
802.11 Ver.	b/g/n	b/g/n	b/g/n
L2 Speed		72,2 Mb/s	150 Mb/s
Idle (DTIM3)	825 µA	900 µA	not listed
TX Power	229 mA	140 mA	190mA
RX Power	59mA	56mA	95 - 100 mA
Bluetooth	-	-	4.2
Vendor	Texas Instruments	Espressif	Espressif
Flash method	proprietary	OpenOCD	OpenOCD
IPv6	none	unofficial	unofficial
IPv4	build-in	build-in	build-in

reviewed boards support a GDB Debugger server to be attached to the device to examine its state. The Espressif boards also already have dedicated open source projects implementing the IPv6 standards. They are fairly equal in regards to power consumption when sending or receiving data over Wi-Fi. The ESP32 is the only examined providing bluetooth features. The SimpleLink family of devices is the only one from the examined platforms offering a standardized data exchange protocol via the network co-processor. The ESP family of devices uses fully proprietary solutions while the CC3200 allows for a POSIX like communication protocol. The communication is handled via sockets. By that the CC3200 offers a more direct communication over the IEEE 802.11 interface than its competitors.

2.2.2 TI SimpleLink device family

SimpleLink is a entry level Wi-Fi hardware family with an equally named network SDK. The SimpleLink devices come in two variants one being a standalone Wi-Fi module for example the CC3100 and the other being a full system with the Wi-Fi module embedded with a more powerful user programmable host system. In the case of CC3100 this system is the CC3200. The CC3200 connects to the CC3100 over a SPI or UART interface. This wiring is embedded on the CC3200 chip and cannot be altered. Development for these systems utilizes the SimpleLink driver to communicate with the CC3100. The CC3100 module in the embedded context of the CC3100 is also referred to as NWP.

2.2.3 Texas Instruments CC3200-launchxl hardware overview

Texas Instruments CC32xx Family is a MCU based on the popular ARM Cortex M4 CPU platform. This MCU features four GPIO ports à eight pins each, two UART and two SPI. Only one of the SPIs can be used for external devices since the other one is hard wired to the on board Network Processor or NWP. The NWP is arguably one of the most important features of this board. The NWP is a module capable of IEEE 802.11 b/g/n, supporting multiple power sawing modes. This board is capable of operation from battery power and is therefore especially useful for portable IoT devices. The onboard NWP is an embedded version of the Texas Instruments external SimpleLink Wi-Fi module, the CC3100. Therefore the CC3200 and CC3100 share documentation and parts of the SimpleLink software development kit. Both of these systems feature a fully capable embedded IPv4 stack. Unfortunately to this point the only way to use IPv6 in the SimpleLink family is to update to the newer more expensive CC3220 and CC3120 platform. All of the SimpleLink hardware runs an embedded operating system on the NWP. Its presence is only known from the Wi-Fi Alliance certification provided by TI. All other information is not accessible to the general public. The communication protocol between the NWP and the host are not documented and TIs forums advise users, seeking these information, to use the SimpleLink driver instead. The hardware itself can be assumed to be able to handle IPv6 with the high CPU clock speed and the size of the on board RAM.

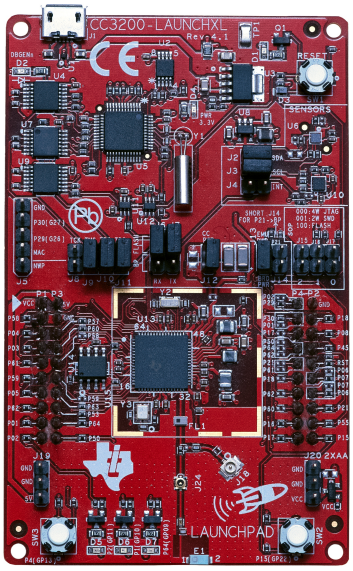

Figure 2.6: TIs CC3200 launchxl board

2.2.4 Texas Instruments SimpleLink Driver

The SimpleLink NWP drivers main task is to establish communication between the NWP and the host system. While not directly mentioned in TIs documentation but can be extracted

from the Wi-Fi certification of the CC3100 module, which is embedded on the CC3200 board, the NWP is running an unspecified version of ThreadX. TIs documentation provides essential information about word length and basic package structure but fails to provide essential information about all available commands of the NWP. The driver itself can operate in standalone or operating system mode. As previously mentioned the SimpleLink driver offers a socket interface for communication. This interface allows to setup direct Transmission Control Protocol (TCP) sockets. Additionally to the higher level sockets a raw socket on the MAC Layer is offered. Managed mode Wi-Fi operation features are also exposed by the driver. In addition the device can be configured to operate as a Wi-Fi station accepting up to one client.

2.3 RIOS Operating System

RIOT OS is a operating system for low power embedded IoT devices. Designed with efficiency and networking in mind. This operating system offers essential features as threads, scheduler, timers and provides access to a variety of networking stacks.

2.3.1 History of RIOT

RIOT OS history dates back to 2008. Some ideas were originally developed for FeuerWare an OS for wireless sensor networks. FeuerWare was developed for a firefighters monitoring project and as such was required to be reliable, secure and perform in real-time. The idea was carried on to become µkleos a direct predecessor to RIOT in 2010 featuring multiple new IETF protocols [WSS09]. After three years RIOT was made public. Since then RIOT has grown substantially counting over 200 contributors to its GNU Lesser General Public License v2.1 licensed codebase on Github and supporting most major low power CPU architectures, over 160 boards, many different networking protocols and device drivers.

2.3.2 RIOTs Goals and Principles

RIOT OS can be divided into to two parts based on hardware dependance. Periphery, drivers, CPU and boards form the hardware dependant part of RIOT, while the remaining system and network modules form the hardware independent core of the OS. RIOT aims to provide implementation and support for open networking protocols. Support a variate of low power IoT devices and thus guarantee a high level of performance and resource consumption optimization. While keeping a low profile RIOT still brings a number of advanced features to these systems. RIOTs modular structure allows for this degree of configuration and scaling in its functionality. Additionally RIOT strifes at avoiding vendor libraries in order to minimize vendor dependencies, lock-in and code duplication [Bac+18]. The comparison of RIOT OS to other popular IoT operating systems is shown in 2.3

GNRC and Netdev

GNRC is the default networking stack of RIOT OS. This stack can be configured to use numerous networking protocols and hardware via drivers. GNRC encompasses multiple different networking layers all operating in different threads. The closer a networking layer is to the hardware the higher its priority execution priority. This basic structure is illustrated

Table 2.3: RIOT OS Compariosn

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	<2kB	<30kB	-	-	partial	yes	partial	partial
Tiny OS	<1kB	<4KB	-	-	partial	yes	-	-
Linux	~1MB	~1MB	yes	yes	yes	-	partial	partial
RIOT	~1.5kB	~5kB	yes	yes	yes	yes	yes	yes

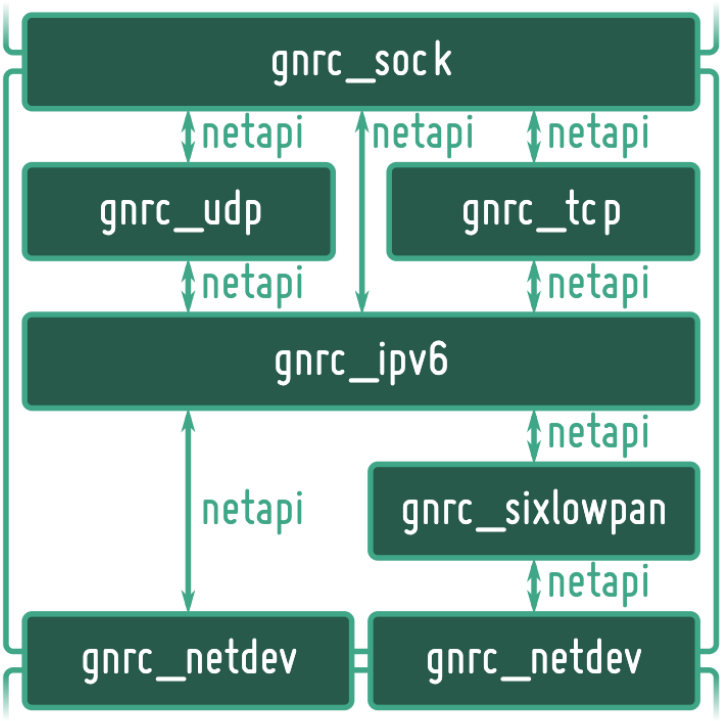

Figure 2.7: RIOT GNRC Network Stack [19c]

in Fig. 2.7. Internal communication is conducted via RIOTs Messaging API. This API allows for easy inter process communication (IPC). In most cases IPC will happen between directly neighboring layers of the GNRC. At the very top of the stack is the Socket API that can be used by the applications to communicate with the network stack. At the very bottom GNRC communicates with netdev, a Device Driver abstraction API. Netdev forms the opposite side of the spectrum, exposing low level driver features to the upper level GNRC stack. The Netdev API essentially handles driver configuration, transmission of data to and from a specific network interface and general driver interrupt handling. With these two parts the whole communication from a network device to the application using a higher level GNRC provided socket is covered.

3 Concept

Hopefully the problems of the current WiFi embedded ecosystems can be seen clearly from the previous chapter. The landscape of embedded 802.11 MCUs is quite diverse and most platforms try to follow their own path. Unfortunately, this leads to a number of problems. As can be seen from the board review in the background chapter of this work, most MCUs rely on proprietary drivers and software stacks. The closed source nature of the 802.11 implementations and hardware make most low level development difficult. While many boards could be updated to support newer standards this proves to be cumbersome due to lacking documentation or missing features.
In this chapter the concept for a mostly open software based Wi-Fi driver supporting PHY and MAC layer is presented. Incorporating base features required for a basic communication over IEEE 802.11 while not covering the whole protocl. For this purpose the TIs CC3200 board is selected due to its integrated debugging interface with OpenOCD/GDB support, rich SimpleLink family and powerful hardware. This will prove essential at debugging and developing at such a low level.

3.1 Choice of Operating System

In many cases when working with low power MCUs an operating system can be omitted and the software developed to run directly on the hardware. The aim here is to provide an easy way for other applications to utilize the hardware Wi-Fi features without the need for any proprietary software with a clean and understandable documentation. A system, abstracting multitasking, memory management and interrupt handling will prove essential to focus on the task at hand. The core criteria for the OS are as follows:

- multitasking
- memory management
- IoT friendly
- easily extensible / modular
- provides a rich ecosystem of supported hardware
- implements a reusable networking stack

While it is essential to choose an operating system simplifying the development process, the more important question is weather the OS will also support future applications utilizing Wi-Fi. As this thesis focuses on the usage of Wi-Fi in the IoT scenario the trivial use cases are IoT devices demanding higher bandwidths than other protocols can offer and require interconnection with WLAN infrastructure already in place. One further constraint is the requirement for an open source licensed operating system to match the goal of an open Wi-Fi

low power implementation. This may filter the list of available operating systems to some extend. Most results of this concept hold for implementations not rooted in RIOT OS as the driver limitations are inherit to the NWP of the CC3200 and not the operating system itself. Some parts of the concept of course are more closely coupled to the OS than others.
RIOT OS provides a number of features aiding the implementation of 802.11. Firstly RIOT OS supports numerous platforms and since the CC3200 NWP is also available as a SPI extension board, the CC31xx, it can potentially be used on other platforms. This way the functionality of the driver can be used on other platform outside of the SimpleLink family. This is not the primary target but an optional bonus. When viewed from an OSI Layer perspective, IEEE 802.11 only provides layer one (PHY) and two (MAC) specifications with some additional security. Any additional applications specific requirements beyond that require other networking stacks. In this scenario it is reasonable to assume that applications using the added bandwidth of Wi-Fi will in addition need IP or a comparable networking stacks. While in theory any protocol can be used with Wi-Fi, the default case is IP. This thesis will not attempt to deviate from this basic scenario. All of the SimpleLink NWPs provide an embedded IPv4 stack and additional support for IPv6 starting with the updated CC3220 and CC3120 hardware. Unfortunately, this "embedded" IP stack is fully close sourced and no information is available to the general public. By this the embedded stack conflicts with the goal of an open source embedded Wi-Fi and therefore will not be used in the context of this work. The driver still aims at providing fallback embedded IPv4 support. Here RIOT provides a significant benefit due to the multitude of already implemented software stacks available for the OS. We can utilize the previously mentioned GNRC to use IPv6 or any other network protocol implementation provided by RIOT or any of its extensions.
Multi threading is also essential for application development especially when Wi-Fi control and package handling is performed on the application processor and not outsourced to the NWP. By no means is multitasking a feature exclusive to RIOT, but it is a needed prerequisite while picking the operation system. This guarantees that other tasks can be performed between sending or receiving radio packages. For future multi-core systems, these tasks can even be performed in parallel significantly increasing the feasibility of a software based Wi-Fi solution.
At the moment of writing RIOT features over one hundred boards and most popular CPU architectures. The foundation of the IEEE 802.11 protocol can be reused by this growing community.

3.2 Texas Instruments CC3200-launchxl

The specific micro controller used for this driver concept is the before mentioned TI CC3200, but it is in no way the intent of this thesis to limit the concept to only this hardware or the SimpleLink platform. As can be seen from the hardware comparison, conducted in the background chapter, many embedded Wi-Fi capable MCUs have major overlap in the way 802.11 is configured on the hardware. Namely most connect the NWP over a standardized interface like UART or SPI, provide Direct memory access (DMA) allowing for direct memory access between the NWP and the host CPU and provide a close sourced Wi-Fi and/or networking stack executing on the NWP. Most devices provide an embedded IPv4 stack and newer boards even feature IPv6. While a big part of the implementation is deeply linked to the hardware, other components may be reused for drivers outside the SimpleLink

family. Therefore the concept will attempt to be platform agnostic as much as possible. The CC3200 also provides a full example of adding a new platform to RIOT OS. The CC3200 is the first Wi-Fi capable Texas Instruments based board being added to RIOT OS and as such requires not only a Wi-Fi driver but hardware abstraction layers and configuration to execute RIOT OS. Some boards already have RIOT OS support, like the ESP32 and the ESP8266.
The CC3200 provides essential debugging support without the need for additional hardware. Writing low level support for a new platform is a major challenge, only made exponentially more difficult by the absence of insights into the current machine state. System faults and CPU exceptions cannot be observed or even noticed before some software needs to be executed, in order to handle these exceptions and output them over same channel, be it an LED or UART. When developing so close to the hardware one cannot rely on these outputs and requires an ad-hoc way to evaluate the code being executed at runtime. For these very reasons a debugger is essential to maintain a rapid development pace. The CC32xx platform features a rich debugging interface supporting OpenOCD and Joint (European) Test Action Group (JTAG) which allows to debug using GDB directly after the bootloader. With the hardware and OS combination the IEEE 802.11 driver implementation can be introduced.

3.3 Host controllable MAC-Layer Socket

The driver requires a stable communication channel to communicate with the MAC layer. As previously mentioned most devices already provide an embedded IPv4 stack and in some cases even an IPv6 stack and therefore already operate at OSI Layer 3 and above. To provide a reliable and adaptable solution the Wi-Fi driver must expose NWP functionality at Layer 2 and below. But it poses significant drawbacks to assume full radio control at Layer 1 and the lower end of Layer 2. In this specific case, operating at a lower level would require manual signal modulation and Carrier Sense implementations. Most Layer 1 features additionally require low latency, which cannot be achieved on most UART or SPI connected NWP due to communication and processing overheads. Thankfully all of the required features are provided by all previously examined MCUs. The documentation and resources on this functionality are unfortunately restrictively limited as the board manufacturers focus on providing an easy to use higher level API. The goal here is to provide a thin software abstraction layer that can receive and send 802.11 frames in accordance to the MAC Layer specification. For a simple data exchange numerous preconditions must be met. The resulting driver must behave as a good "neighbor" and listen for ongoing communications on the selected wireless channel and also adhere to the slot timings of CSMA/CD. Additionally packets must be postfixed with an 32 bit FCS error-detection code calculated based on the MAC header and the frame body. This error-detection code must also be validated when receiving packets to guarantee an error free transmission. Since any singular Wi-Fi frame exchange must be completed within the time frame of a single slot (including receiver ACK transmission for non broadcast frames), it is essential to perform these tasks directly on the NWP to fulfill the strict timing requirements. Especially the ACK frames pose a serious concern and should, if possible, be send by the NWP. The slot time depends on the 802.11 protocol used, for intended 802.11g standard the two possible values are 9µs or 20µs. At this point of the concept it was assumed that the hardware will automatically ACK packets on arrival as suggested by the documentation. Unfortunately it was shown to be not true and

this option only became available in future CC3220 versions of the platform.
Direct radio control is not possible on most MCUs. But this allows the NWP to perform essential low level operations without the need to spend precious CPU cycles of our host system. Assuming the abstraction layer is able to configure the NWP to perform core MAC layer features without intervention, the only missing part to our abstraction layer is the actual transmission of data from and to the NWP. The specific data exchange between NWP and the host may vary but this procedure holds for most NWPs. The NWP and the host communicate via a simple message protocol requiring command header before actual payload. The structure of the message depends on the specific board. The frame parameter is assumed to be a fully constructed 802.11 frame and must adhere to the maximum Wi-Fi frame sizes, Maximum transmission unit (MTU). MTU is composed of the maximum payload, the mac header and an optional suffix depending on encryption ($p_{encryption}$), resulting in $MTU_{bytes} := 2304 + 34 + [p_{encryption}]$. The encryption component is disregarded in the concept. The proposed send functionality should be agnostic to the state of the 802.11 con-

Algorithm 1 Sending a single Wi-Fi frame

```
procedure SendFrame(frame)                          ▷ Send a constructed Wi-Fi frame
    PrepareNwpForRead                                          ▷ Prepare NWP for send
    SendCommandHeader(sendRawCmd)                    ▷ Announce a send operation
    toSendLen ← Length(frame)
    offset ← 0

    while offset ≠ toSendLen do                                ▷ send while payload left
        SendTransmissionUnit(frame[offset])
        offset ← offset + transmissionUnitSize
    end while

    while responseACK = NULL do                                ▷ send while payload left
        Wait for response ACK or timeout
        if TIMEOUT then          ▷ If timeout reached indicate transmission failure
            return FALSE
        end if
    end while
    return TRUE
end procedure
```

nection allowing to implement 802.11 association and connection management directly on host. Therefore it should be possible to send all three types of MAC frame. This Layer 2 send functionality requires the host to enforce that outgoing data is acknowledged by the recipient. Since the reception of the ACK does not require any time critical action this check can easily be performed by the host CPU. This implementation ignores QoS and other Wi-Fi extensions.
To provide a full duplex Layer 2 socket a receive method is required as well. Commonly the NWP notifies the host about incoming traffic via a peripheral interrupt. This interrupt halts the CPU execution and allows a user programmable method to be executed. This pattern is common for UART or SPI connected NWP. The interrupt will trigger the specified handler function. Since this is commonly the only way for the NWP to trigger code

execution on the host it is also used for configuration and device management. These tasks will also need to be handled by the interrupt handler to provide a robust solution. To handle this asynchronous response behavior a later discussed driver queue will be created to handle storage and management operations.

3.4 MAC Layer Packet filtering

MAC address filtering is essential for an efficient operation of the driver. Almost all 802.11 frames provide a source and destination fields as mentioned in the background chapter. These fields together with the type and subtype of the FC field of the header are used by all stations to determine relevance of a givin frame. The intent of this thesis is not to provide a Layer 2 sniffer (while still possible) but to provide an efficient MAC Layer socket. Performing MAC filtering on hardware would yield power efficiency gains. For example some Wi-Fi enabled boards can perform the MAC filter directly on the NWP removing the need to transport the whole frame to the host, which in term would need to read and evaluate the relevant fields. This may seem counter productive with our set goal to perform as much as possible on the host. Like many problems the Software Defined Radio (SDR) needs to be well balanced against usability. If a system is too reliant on software, memory, CPU time and power usage normally tend to increase while outsourcing all functions to the NWP remove potential control from the developer. To bridge this gap the suggestion is to provide "a use to use" API to enable embedded MAC filtering schemes if present but to also allow the direct filtering, if it is deemed more important, to perform this operations on the host.

3.5 Command Operation Queue

The Command Operation Queue is used to store NWP communication and configuration messages awaiting a response. As previously mentioned the driver will require to trigger multiple NWP operations with unpredictable response times and order. In order to solve this problem a simple Queue is proposed. The queue stores the last n commands called by the host code. The command in the queue is denoted by its request and response codes or other discriminatory fields specific to the NWP communication protocol. When a request awaiting a response is send to theNWP the request is added to Queue. Each request consists of four pieces of information request code, response code, timestamp and the callback which it to be executed when the response code is received. The structure specific to the CC3200 will be discussed as part of the implementation. The request is kept in the queue till either a response with a matching code is received or no matching response was returned within a timeout period. The timeout should be chosen on a per system basis and should prohibit request from blocking the queue indefinitely. The timeout can be time- or response based. The number of commands awaiting response is expected in the range of 1 - 10 so the queue utilizes a simple Array structure.
When the NWP triggers a peripheral interrupt the driver call reads the incoming data,parses and validates it and later matches the incoming message against the response queue. To prioritize 802.11 traffic over NWP control commands the 802.11 payload interrupts are handled before commands on the queue are matched and executed. If an incoming command is matched via its response code the callback is executed with the payload provided by the NWP. After this the request is removed from the queue. In order to prevent performance

intensive copy operations the queue items are not moved but rather the initial offset of the command queue. This index is pointing to the first not empty index in the queue and eventually loops back to zero, operating like a ring buffer. This way the average search time can be significantly smaller the length n of the command queue without the need for list migrations. This interrupt handler will not be directly coupled to the interrupt but will be called as part of the RIOT OS netdev event callback to adhere to RIOT netdev interface specifications.

3.6 802.11 PHY and MAC feature set

The goal of the driver is to not only provide a IEEE 802.11 MAC socket but also to allow the MCU to communicate with other devices. To achieve this the driver requires a client infrastructure mode BSS protocol implementation. As mentioned previously in the background chapter a BSS discovery must be conducted, establish a connection and authenticate. To limit the scope and complexity this concept only aims at operation on open access networks without any encryption. Potentially the proposed driver can be extended to operate as an AP but the current concept only covers the client station side of the 802.11 standard.
Operating as a Wi-Fi client requires numerous frame types and parts of the 802.11 standard to be implemented. The core requirements and mechanisms of a basic Wi-Fi client BSS setup are illustrated as part of the background chapter. The driver is not designed to perform active collision prevention. Therefore the list of frame types covered by the concept is as follows:

- ACK Frame
- Beacon Frame
- Probe Request Frame
- Probe Response Frame
- Association Request
- Association Response
- Authentication Request
- Authentication Response
- Data Frame (no QoS, CF-pull or CF-ACK)

All of the above frames are required for basic Wi-Fi operations in infrastructure mode. The concept provides an extensible definition for these frames and the 802.11 standard that can be reused outside of the driver or the specific board at hand. To achieve this it is intended to create a netdev and GNRC RIOT OS extension. One of the most challenging parts of the communication is the association process, requiring to parse other device configuration and expose the hardware capabilities of the MCU used. To keep this process simple the MCU will advertise only a small subset of its actual capabilities limiting the number of configurations needed to be handled on the host. Since most frames will be constructed on the host it is essential to utilize some sort of memory reuse to avoid CPU cycle used for memory allocations. Thankfully this is in part achieved by the utilization of RIOT GNRC.

3.7 RIOT GNRC Integration

RIOTs `netdev` interface allows for low level drivers to communicate with any networking stack designed to operate with the OS. The `netdev` interface and its utilization in combination with GNRC is discussed in depth as part of the background chapter on RIOT OS. In short `netdev` interface if implemented exposes three functionalities to above layers of the networking stack:

1. **Sending and Receiving**: Internally performing data transmissions and reading from the hardware.
2. **Transceiver configuration and initialization**: Performing all required setup procedures and configuring the hardware to the intended mode.
3. **Transceiver event handling**: Handle interrupt based device communication and guarantee that all data transactions are performed on the same thread avoiding interrupt shadowing.

When implemented and configured this interface is used by network layers above for package/frame exchange [Bac+18]. Therefore the driver aims to expose the Layer-2 send and receive functionality discussed previously in a `netdev` conforming manner. Thus allowing the use of RIOTs IPv6 stack as part of GNRC or any other compatible RIOT supported networking suite. Sending and Receiving matches the proposed Layer-2 socket, while the configuration and initialization is highly specific to the hardware and will be covered in depth as part of the implementation. The transceiver event handling on the other hard is a more general concern since most Wi-Fi enabled MCUs do not have a dedicated connection to the NWP. The connection is then carried out over a central bus commonly used for other periphery devices as GPIO or UART controllers. Many other periphery devices also communicate with the host via interrupts, thus it is possible that while reading data from the NWP a different device interrupts the process and reconfigures the hardware in its respective handler. To prevent such scenarios RIOT uses mutexes to lock the usage of asynchronous periphery on a driver level. Unfortunately these locks operate in a thread context by design and not from within an interrupt context, by that rendering them un-functional during interrupt handler invocations. Solving this problem requires the actual reading from the NWP to happen outside the interrupt handler context. Therefore the interrupt handler does not actually transfer data but simply denotes that the NWP has data awaiting processing and the actual read/write is carried out on the driver thread when called by the event handler. The full process for a NWP driver is illustrated in Fig. 3.1.

3.8 Summary and Findings

RIOT OS and the CC3200 were chosen for this concept most importantly for their flexibility, IoT support and extensive features. The CC3200 provides a powerful hardware with embedded debugging support aiding the implementation, while RIOTs modular structure and approach to drivers offers a solid foundation for the addition of IEEE 802.11 and the CC3200 Wi-Fi driver. The main concept is to firstly expose a Layer 2 socket capable of data transmissions from and to the Wi-Fi co-processor. This socket is then to be connected to RIOT OS netdev interface as a netdev driver. Additionally the IEEE 802.11 protocol will be added to RIOT OS core system.

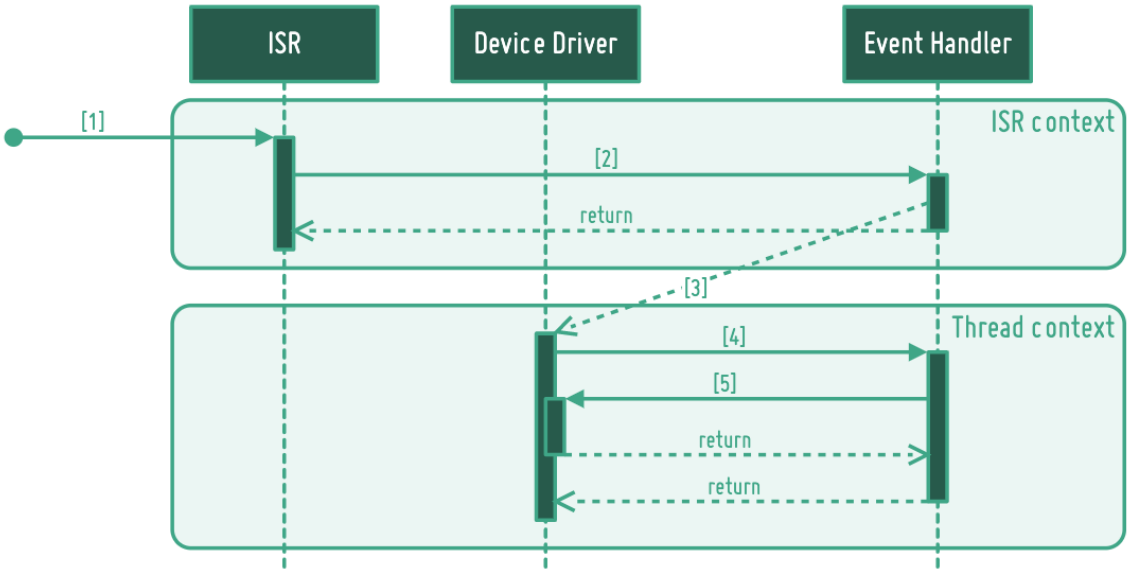

Figure 3.1: NWP receive event handling [19b]

4 Implementation

This chapter of the thesis covers the implementation of the previously discussed concept. The implementation covers three major points. Firstly the port of the CC3200/CC3200-launchxl to RIOT OS. During this porting process the build structure of RIOT will be elaborated and general periphery operation covered. With an operational CC3200 board running RIOT OS the NWP communication protocol will be analyzed for later use as part of the driver. Finally the integration of IEEE 802.11 into RIOT OS is conducted and the resulting netdev interface is connected to RIOTs GNRC.

4.1 TI CC3200-launchxl for RIOT

The proposed RIOT OS driver requires a functional foundation to operate. Networking and executing user applications on a single core CPU with the resource limitations of an embedded system is challenging. As mentioned previously RIOT OS is essential to this port since the added complexity of implementing threads, stacks, interrupt handlers and memory management would significantly increase the difficulty of this endeavour. But this poses a further challenge, namely the port of CC3200 core functionality to RIOT. It is still possible to base our software driven Wi-Fi on the foundation of TIs proprietary OS but this strongly contradicts the open source nature of this project. In the greater scheme of things it is considered simpler to port CC3200 and essential periphery devices to RIOT OS than to implement required functionality from scratch. In order to run RIOT on TIs platform three components are required. Firstly the CPU-Architecture must be supported by RIOT OS. In the case of the CC3200s ARM Cortex M4 this is already provided. Secondly peripheral devices like timers, SPI and UART must be operational. Finally a low level debug interface is required to quickly resolve programming errors at a point when no other form of hardware output is possible. In the following chapters the process of porting the CC3200 hardware to RIOT will be shown.

Executable memory

The CC3200 platform is different in regards to the way executables and memory is handled. The CC3200 has no directly accessible flash memory, but does feature a serial flash module. This storage is used for CC3200s firmware including an SDK, whose components will be referred to as `ROM_<MetodName>` in future code snippets. The CC3200 board utilizes a simple File Allocation Table (FAT) filesystem to manage the stored files. The flash can be written to by TI tools but not by other common technologies like OpenOCD. In addition to the firmware, the filesystem contains configurations and caches used by the MCUs internal components e.g. ARP cache, system calibration and NWP connection information. The flash is also used to store the user executable. When powering on the CC3200 a bootloader is executed, which copies the user programmable binary into the RAM from the filesystem.

Presumably the NWP can also access and write to the serial flash, since most of the configurations in the `sys` folder are network related. In total the CC3200 has either 996KB or 1260KB of serial flash depending on the board version and model. The flash memory can be read and written to by the host processor, allowing to use the flash as non executable storage for data and logs.

4.1.1 Makesystem

RIOT is designed to be modular and so allows far the addition of new CPUs, boards and drivers. For this to work RIOT relies on a hierarchical Make structures combined with naming conventions throughout the project. A simplified example for a Cortex M board is shown in Fig. 4.1. Each RIOT module, be it a board or CPUabstraction layer, provides its dependencies via `Makefile.dep` files. A module is commonly contained within a folder and can contain a `Makefile.dep`, a `Makefile.include` and a regular `Makefile`. If the module provides features, the module folder can also contain `Makefile.features`. The feature system is especially useful when executing a RIOT application on a different boards allowing RIOT to suggest missing board functionalities at build time. For example the CC3200 offers UART, timers and GPIO which are all listed within the Makefile of the CPU. When executing a build process depending on the selected board and target all modules are compiled and linked into a single binary. The CPU module must provide a Linker Script to properly map the compiled input files to the output binary and correctly layout the memory for the hardware architecture. Linker script files, denoted with the file extensions `ld`, can also be extended in the same way as Makefiles. Thus, making it possible to reuse Cortex M base linker script as a template for the CC3200. If the Cortex M linker script is imported the CC3200 script only has to provide the Random Access Memory (RAM) and Read Only Memory (ROM) regions.

4.1.2 CPU

The Cortex M4 platform is a widely available and popular choice for embedded systems. This CPU is found in a variety of other devices. Due to its popularity, the basic support is already provided by RIOT right out of the box. The Cortex M family of CPUs have a significant overlap in their instruction sets and other attributes, RIOT combines the shared code required to operate on this architecture into a single module. The specific version of the CPU family can be chosen via make macros when using the `cotexm_common` module.

4.1.3 Interrupt Vector Table

Cortex M4 as all Cortex M CPUs has an on chip Nested Vectored Interrupt Controller (NVIC) for interrupt handling allowing for a maximum of 240 separate interrupt codes with multiple levels [Yiu13]. An interrupt is a signal causing the CPU to pause current execution and perform a pre-configured action. Some of these interrupt codes can be connected to other hardware by the board manufacturer as is the case on the CC3200. RIOT already provides interrupt handling, only requiring the setup of an interrupt code to callback mapping. This mapping is called interrupt table or vector table. RIOTs Cortex M implementation will then configure the CPU to use the provided vector table when an interrupt is triggered. Listing 4.1 illustrates a partial implementation of this table. For example the GPIO input, if enabled, fires an interrupt with the code 16. The later used NWP interrupt code is also

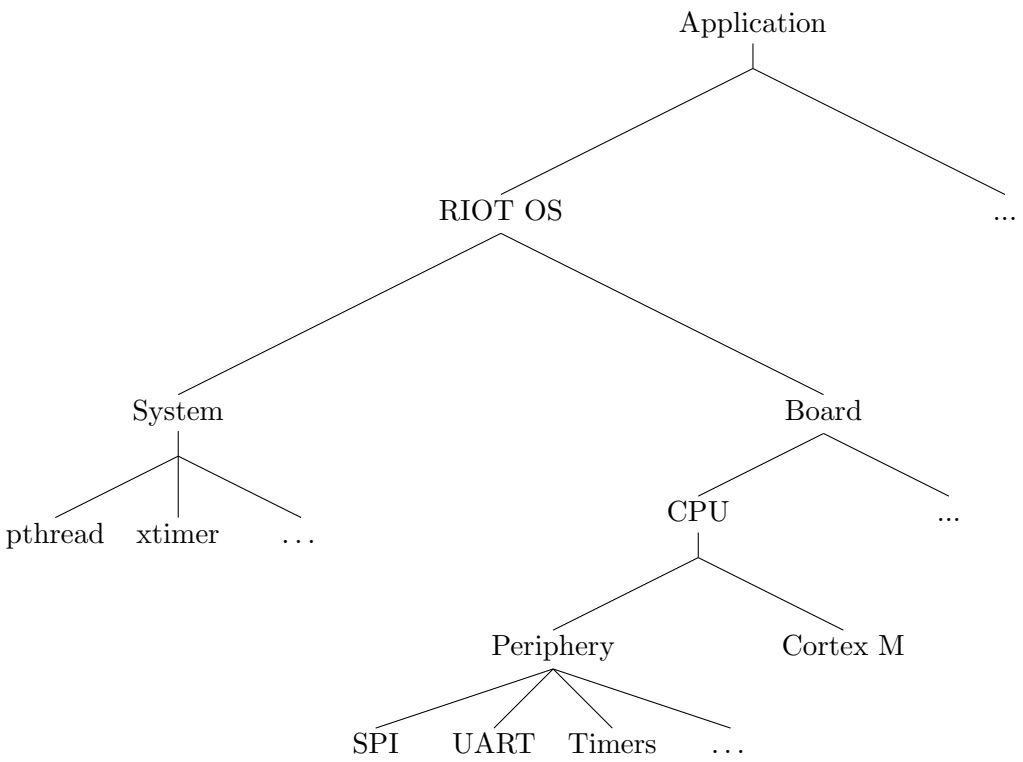

Figure 4.1: RIOT OS simplified Makefile build hierarchy of the CC3200

```
ISR_VECTOR(1)
const isr_t vector_cpu[CPU_IRQ_NUMOF] = {
    [0]     = isr_gpio_porta0, /* 16 GPIO Port A */
    ...
    [171]   = isr_nwp,         /* 187 NWP to APPS Interrupt */
    ...
    [177]   = isr_link_spi,    /* 193 Link SPI (APPS to NWP) */
};
```

Listing 4.1: Exceprt from CC3200s vectors.c vector table definition

```
// DIG DCDC Voltage Out trim settings based on
//  PROCESS INDICATOR (bits 22-25
//  of GPRCM_EFUSE_READ_REG0) of FUSE ROM
if (((GPRCM->GPRCM_EFUSE_READ_REG0 >> 22) & 0xF) == 0xE) {
    DIGI_DCDC_VTRIM_CFG =
            (DIGI_DCDC_VTRIM_CFG &
                ~
   HIB1P2_DIG_DCDC_VTRIM_CFG_mem_dcdc_dig_run_vtrim_M) |
            (0x32 << 18);
}
```

Listing 4.2: Undocumented init code

shown in this snippet. As can be observed in the codes in the comments do not match the table offsets, this is due to the way `ISR_VECTOR(x)` macro operates. RIOTs Cortex M code already implements the first 16 interrupts and `ISR_VECTOR(x)` combines the two tables into one. When an interrupt is triggered, the method referred to in the corresponding vector table offset will be executed. The methods referred to in the table (e.g.`(void)isr_nwp(void)`) must also be defined, yet the current implementation by default uses the build in `WEAK_DEFAULT` macro which points to a provided "dummy" method. The `WEAK_DEFAULT` can be overwritten by any other implementation of the same name in any other file. This will later be used to register periphery interrupts without altering the vector table manually.

4.1.4 CPU abstraction Layer and initialization

RIOT separates a board into two modules, the `board` and the CPU abstraction layers. The logic is designed to idiomatically split these two parts, allowing for multiple boards to reuse a single CPU abstraction implementation, as it is common for vendors to construct multiple boards upon a single CPU design. By this periphery device implementations can be shared. The CPU abstraction module can be further modularized, as can be seen with the Cortex M family implementation. RIOTs CPU abstraction modules are required to implement a single method `void cpu_init(void)`. This method is later called from within the boards initialization method to perform CPU specific preparations. The CPU initialization is responsible for configuring the interrupt handling which in the case of CC3200 is covered by RIOTs `cortexm_init()` method. Thus, it is only required to initialize the periphery devices directly connected to the CPU and some essential components. The low level setup of the CPU is covered by TIs `PRCMCC3200MCUInit` method which unfortunately is not documented and the respective source code is only sparsely commented. The implementation could have used a ROM version of TIs initialization `ROM_PRCMCC3200MCUInit` but to provide a clear and direct initialization step the init code is rewritten to match RIOT coding standards. This setup is performed during main CPU initialization before the periphery setup. While some parts of the initialization are difficult to understand without documentation and are therefore kept in place. Other parts of the initialization can be rewritten with the structures created for the CC3200 periphery. Fig. 4.2 illustrates some of the more complicated not described blocks which were kept in place with only minor additions. `GPRCM ->GPRCM_EFUSE_READ_REG0` in the snipped above is using a abstraction register GPRCM

which originally was computed by memory addition of the GPRCM base memory address. With all things considered `PRCMCC3200MCUInit` resets most digital to analog converters and resets other hardware registers to their initial values. After which the CPU initialization has to call the `periph_init()` method to invoke RIOTs periphery setup, which will init all periphery modules required by the applications `Makefile.dep` file.

4.1.5 RIOT board abstraction Layer

The CC3200-launchxl provides a number of extended devices unique to the boards, as the audio processing unit or the onboard ICE debugger module. This additional hardware is not provided by all CC3200 boards. The features are considered to be more closely coupled to the board and therefore placed into a board abstraction module. The board module in this case will be named `cc3200-launchxl` matching the board model name. As can be seen from the RIOT build structure discussed before, the board imports the CPU when building the OS and the application. This is configured via the `CPU` environment variable set in the make configuration. In order to use the previously defined `cc3200` the `CPU` is simply set to the name of the defined CPU module to be used, in terms RIOT infers the folder name. It is typical to place debugging and flashing configurations into the board folder since these also tend to vary based on the board and not the processor. These configurations must implement the `DEBUGGER` and `FLASHER` environment variables to allow the usage of RIOTs `debug` and `flash` targets respectively.
TIs CC3200 board can be flashed by a number of tools, unfortunately not including the common OpenOCD. Uniflash is a standalone tool for flashing most TI MCUs and allows to flash firmware as well as application binaries. Due to limited linux and darwin support of the required Uniflash tool a different program was chosen. TI offers a custom version of the Arduino IDE named Energia. Energia allows to work with the CC32xx platform of devices and downloads the SimpleLink SDK. A custom flashing tool is included in that SDK if downloaded via Energia. The `cc3200prog` tool is provided in Windows, Linux and macOS flavours and is able to flash the `sys/mcuimg.bin` to the device consistently on all platforms. To use `cc3200prog` as the flasher the `ENERGIA_TOOL` environment variable must be set when flashing to the MCU. The documentation provided with the source code for the CC3200 board explains how to obtain and use the cc3200prog tool in more detail.
The MCU also features embedded debugging, allowing it to be connected to Open On-Chip Debugger (OpenOCD). For this purpose the CC3200 has an FTDI FT2232D Chip connected to the micro USB port. This chip allows the board to expose two TTY connections to the device it is connected to via USB. This hardware is utilized to expose the onboard JTAG or Serial Wire Debug (SWD) controllers and enables communication to a GNU Debugger (GDB) instance operating on the development machine. Previously the OpenOCD configuration had to be downloaded from TIs webpage but now this configurations are already embedded into recent OpenOCD installation. For OpenOCD to use the correct parameters the source must be set to `board/ti-cc3200-launchxl.cfg` as illustrated in the content of the cc3200.cfg shown in Fig. 4.2. The MCU also features embedded debugging allowing it to be connected to OpenOCD. For this purpose the CC3200 has an FTDI FT2232D Chip connected to the micro USB port. This chip allows the board to expose two TTY connections to the device it is connected to via USB. This hardware is utilized to expose the onboard JTAG or SWD controllers and enables communication to a GDB instance operating on the development machine. Previously the OpenOCD configuration had to be

```
source [find board/ti-cc3200-launchxl.cfg]
$_TARGETNAME configure -rtos auto

$_TARGETNAME configure -event gdb-attach {
    halt
    }
```

Figure 4.2: OpenOCD configuration

downloaded from TIs webpage but now this configurations are already embedded into recent OpenOCD versions. For OpenOCD to use the correct parameters the source must be set to `board/ti-cc3200-launchxl.cfg` as illustrated in the content of the cc3200.cfg shown in Fig. 4.2. The source at the beginning finds the correct configuration file, followed by some configurations needed for GDB to operate properly with RIOT OS. The `halt` on line 5 will halt execution as soon as gdb has attached to the system allowing the debugging directly from the entry point of the binary. The behavior of GDB can be configured via an additional GDB configuration file. With the CPU and board configured, RIOT can be compiled for the CC3200 platform.

4.1.6 Periphery abstraction Layer

RIOT OS provides a number of additional abstraction interfaces to connect system features with hardware devices and delivers an unified access to its functionality. For this a low level periphery implementation is required. The CC3200 SDK exposes a basic abstraction layers definition for most onboard hardware. In this chapter it will be attempted to generalize the vendor implementation. Firstly the general abstraction layer for periphery is shown on the simple example of GPIO, timers and SPI. The SPI is of most importance since it will be used to communicate with the NWP from within the driver. To avoid redundancy UART and power management are omitted from this thesis but can be reviewed in detail in the provided source code.

General Purpose Input Output

The CC3200-launchxl features 24 GPIO pins. The GPIOs can be used to connect a variety of devices to the MCU. The GPIO pins are grouped into four GPIO ports of 8-bit each. Contradictory, the SDK lists the memory mappings for five ports instead of the four mentioned in the documentation. Most GPIO pins are single purpose with some allowing to be reconfigured and used for other purposes and modes. The GPIO controller is connected to the host processor via a bus system and can trigger host interrupts upon changes to the voltages of pins. The GPIO registers are directly mapped to the host memory.
Initial vendor implementation uses macros to compute memory offsets. To improve code readability these macros were rewritten to statically mapped structures. For example the original SDK code accesses the address of the data register as follows:

```
(*((volatile uint32_t *)(GPIOA0_BASE + GPIO_O_GPIO_DATA)));
```

```
#define GPIO_PIN(x, y) ((x << 6) | (y - 1))

/* led definition based on gpio pin macro */
#define LED_RED GPIO_PIN(PORT_A1, 64)
#define LED_ORANGE GPIO_PIN(PORT_A1, 1)
#define LED_GREEN GPIO_PIN(PORT_A1, 2)
```

Figure 4.3: GPIO port and pin encoding used for `cc3200_gpio_t`

`GPIOA0_BASE` is the numeric value of the address where the register is mapped to and resolves to `0x40004000` for the CC3200-launchxl platform. The same access can be done in RIOT using the implemented structure as follows:

```
gpio(0)->dir;
```

`gpio(0)` returns the address of the first GPIO port and is equal to the `GPIOA0_BASE` with an additional type conversion to `cc3200_gpio_t`. While in total RIOTs SPI abstraction layer requires the implementation of nine methods to adhere to the interface, only `gpio_init`, `gpio_read` and `gpio_set` will be symbolically examined in this chapter. All of the above methods operate on an 8-bit unsigned int parameter, commonly denoted as `dev`. This parameter specifies the port and pin to be accessed. This encoding is possible since only 2 bits are required to encode the index of a single port and the remaining 6 bit suffice to identify every pin within that port. The pin index is decremented by one to match the onboard pin labeling, since its enumeration begins at one (encoding shown in Fig. 4.3). Both RIOT and the onboard GPIO controller use numeric value to represent the configuration of a given pin, be it `GPIO_IN` for input or `GPIO_OUT` for output pin modes. Unfortunately these values do not match and thus `gpio_mote_t` and `gpio_flank_t` enumerations must be overwritten as part of the GPIO abstraction. `gpio_init` configures a given port and pin configuration to a provided mode corresponding to the values set in `gpio_mote_t`. Before the pin can be configured its value is returned to the initial state as is done in the vendor SDK. Following that the pin is configured to operate in GPIO mode since many pins can be used for other purposes like UART for example.

General Purpose Timer

Timing is essential for most wireless protocols and IEEE 802.11 is no exception. In addition to the protocol required timing RIOT OS requires precise timing to perform scheduling and other core functionalities. The CC3200 provides four Timer modules. In a concise description a timer is a hardware module that can perform operations with consistent delay. Timers used to count time essentially increment a register value at a stable rate in regards to time. The CC3200 timers support both 16 or 32 bit counter sizes but due to configuration limitations only 16 bit timers are operational with RIOT OS. The timer is linked to the CPU core clock and thus on a hardware level operates at 80Mhz [18a, Chapter 5]. The timer implementation is part of the CPU abstraction layer with additional configurations being set on a board level (number of timers, channels per timer), the full list of files is shown in Fig. 4.4. The Timer implementation is based on the already existing `cc26xx` and `cc2538`periphery source code. In future these boards could be combined to an unified TI

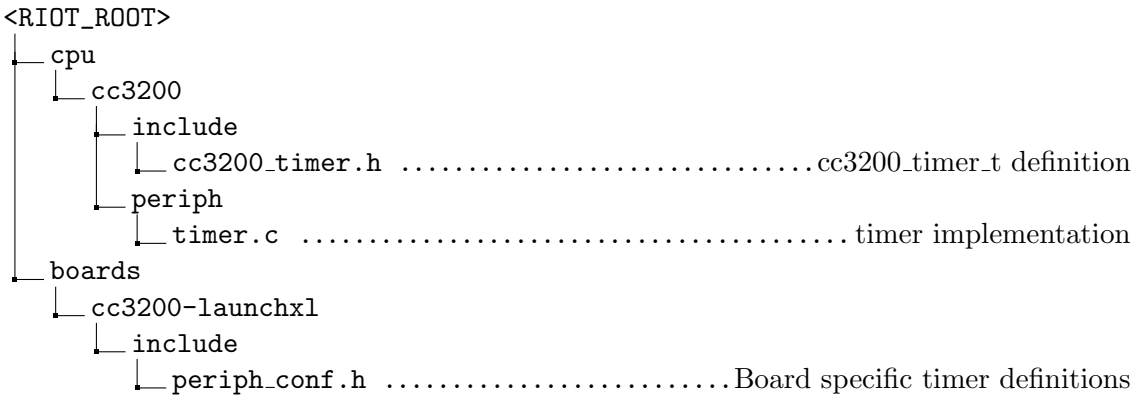

Figure 4.4: RIOT cc3100 Driver file additions and modifications

CPU abstraction layer to avoid redundancy. RIOTs timer abstraction interface requires the presence of methods for starting and halting a timer, initialization, reading of timer values and registering timer interrupts. The timer initialization is done within the `timer_init` method. Such options as split timers, a 32-bit timer can be used as two 16-bit timers, interrupt handling and frequency configuration are configured in this method. The two 16-bit timers are referred to as half timers or in the context of RIOT, timer channels. Timer frequency must be carefully constructed. RIOTs `xtimer` module requires a timer frequency easily convertible to one microsecond. RIOT provides a number of time conversion macros which are unfortunately not compatible with the 80Mhz timer clock. This requires the utilization of the hardware prescaler to reduce the counter increment frequency from 80Mhz to 1Mhz or one increment every microsecond. The prescaler can only be used for 16-bit half timers, hence the unsupported 32-bit timer configuration. The prescaler register is a one octet long register, when set this register slows down the timer action by the set value $0, ..., 255$. For example to slow down our timer to 1Mhz a prescaler value of 79 is used. This will trigger a counter increment every 80th timer pulse, instead of on a continuous increment on each timer pulse, when set to the default value of zero. The prescaler is computed dynamically in the `timer_init` method. The value is calculated as follows with $frequency_{cpu}$ being the CPU frequency in our case 80MHz and $frequency_{xtimer}$ in this case 1MHz.

$$prescaler = min((max(prescaler - 1, 0), 255)$$

$$prescaler := \frac{frequency_{cpu} + frequency_{xtimer}/2}{frequency_{xtimer}}$$

The prescaler value is then adjusted to fit the 8-bit value range and is decremented by one if its value is larger zero. The interrupt callbacks are overwriting the `WEAK_DEFAULT` values provided by `vector.c`, not requiring any changes to the interrupt table values as is case for the GPIO interrupts. The underlying timer controller is mapped to the `cc3200_timer_t` struct type. This struct is comparable to the GPIO hardware abstraction struct. Each timer can only trigger a single match interrupt used to indicate a reached time value set by `timer_set_absolute`. Therefore this event can be triggered multiple times for a single timer when configured in half timer mode. Therefore a `NUM_TIMERS` long array of `chn_isr_cfg` structures is used. This structure stores unique configurations for both channels of the timer.

```
typedef struct {
    uint32_t mosi; /**< pin used for MOSI */
    uint32_t miso; /**< pin used for MISO */
    uint32_t sck;  /**< pin used for SCK */
    uint32_t cs;   /**< pin used for CS */
} spi_pins_t;
```

Figure 4.5: SPI pin configuration struct

SPI

CC3200 SimpleLink NWP is connected to the host processor via a shared bus. The default mode of operation is SPI while the NWP also supports UART. SPI provides a full-duplex connection to the NWP module. Analogues to the GPIO example RIOT provides an abstraction layer which has to be implemented. For this reason two files are created in the CPU module `periph/spi.c` and `include/cc3200_spi.h`. `spi.c` contains the implementation and `include/cc3200_spi.h` all the types and overwrites.The pin layout and further configuration of the SPI is placed within the board. CC3200 SPI can operate at bus frequencies from 100 kHz to 20 MHz (CC3220 up-to 30 MHz). Not all of these speeds are defined as part of RIOT and hence the `spi_clk_t` has to be overwritten. The 20 MHz is essential since it is the preferred speed for the NWP module (NWP will cancel operation on the bus speeds for unknown reasons). A SPI connection is typically composed from four pins (in some cases five) this is modelled in the struct described in Fig. 4.5.

This type is used for SPI initialization. Some of the pins used can be reconfigured to operate in non SPI modes like GPIO. This configuration is performed by the RIOT abstraction layer method `spi_init`. The SPI interface to be selected is identified via a simple number of type `spi_t`, which in this case is a unsigned 32 bit integer. This format is not only the default for RIOT but is required for some of the SimpleLink ROM methods. SPI utilizes the mutex locking, mutexes provide thread safe access to hardware, to prevent simultaneous access to the hardware. The mutexes are locked when the SPI is configured `spi_init` or when it is acquired using `spi_aquire`. Per RIOT specification any SPI usage must acquire the device first, before performing any operations, thus the check and mutax lock is only performed in those two methods. When an operation has no need for SPI anymore the `spi_release` is called, freeing the lock. The SPI controller was converted from the vendor provided offset based implementation into a structure in a comparable fashion to the GPIO struct (Fig. 4.6). In addition to simple pin configuration SPI controller requires timing and other configurations like word length or chip select polarity. These configurations are performed in the `void _spi_config(spi_t bus, spi_mode_t mode, spi_clk_t clk)` method. This lengthy configuration is omitted from this thesis but condenses to the following. Firtsly the computation of the clock speed in proportion to the processor frequency of the MCU. Dev in this context is referring to an instance of the `cc3200_spi_t` struct. This struct is used to configure the transmission frequency, SPI Master Mode and Hardware Chip Select. Additionally operation mode, word length and turbo mode can be set via the 32bit `dev->ch0_conf` register. Enable relevant SPI dedicated timers. Restart the SPI bus for the configuration ot take affect. Actual read and write operations are performed in one method

```
typedef struct cc3200_spi_t {
    cc3200_reg_t rev;           // hardware revision
    // ...
    cc3200_reg_t ch0_conf;      // CH0CONF CTL
    cc3200_reg_t ch0_stat;      // CH0 Status register
    cc3200_reg_t ch0_ctrl;      // CH0 Control register
    cc3200_reg_t tx0;           // single spi transmit word
    cc3200_reg_t rx0;           // single spi receive word
} cc3200_spi_t;
```

Figure 4.6: SPI component register snippet

`spi_transfer_bytes`. This method in addition to the selected SPI device and Chip-Select mode gets a pointer to both an input `in` and output `out` buffers. Both buffers are optional but at least one must be set. The actual transmission is handled by the ROM method `SPITransfer`. This method was not ported to the RIOT SPI implementation due to time constraints. This method handles transfers over SPI configurations of any supported word lengths and returns a zero if read or write was successful. The full source code implements all remaining interface methods and thus enables RIOT to use SPI for communicating with the NWP and other devices.

4.1.7 Compiling and examining the executable

With the necessary hardware abstraction layer defined, the binary can be compiled for the target MCU. RIOT OS supports compilation with Docker allowing to compile on non linux systems and without the need to install the tool chain on the development OS. Additionally depending on the application to be compiled the Linker Script may require some adjustments. Currently RIOT cannot dynamically adjust ROM/RAM memory layout. This may lead to compilation failure if the ROM segment of the resulting binary is too small to fit the program. By default the Linker Script is configured to allocate 44 KB for ROM and the remaining 196 KB are used as RAM. This is more then enough for the `hello-world` examined at this step. The `hello-world` example can be compiled for the newly created board using the following commands (docker usage can be configured by `BUILD_IN_DOCKER` environment variable)

```
$ BOARD=cc3200-launchxl make all build
```

The command must be executed from the application folder so in this case the folder is `examples/hello-world`. During the development process there were multiple configuration problems with the entry point of the binary and the alignment of the Interrupt Vector Table. These values can be examined in the resulting binary. Per default RIOT compiles the binary to a sub folder `bin/<BOARD_NAME>` within the application folder. The binary can be analyzed by a variety of tools but Radare 2 was chosen for its rich visualization features. To make the binary examination simpler all compiler optimizations should be turned of `-O0` and all symbols embedded into the binary `-g`. This can be simply done in the application `Makefile` or board `Makefile.include` by adding `CFLAGS += -O0 -g` to the file. The build process creates two binary files `hello-world.elf` and `hello-world.bin`. In the following only

Figure 4.7: Radare 2 `.text` segment

`hello-world.elf` is used since the `.bin` cannot be examined by radara2 and seems to not contain any symbols. When examining the Executable and Linkable Format (ELF) binary we expect our `.text` to be placed at memory offset `0x2004000` since that is the first non reserved memory region and after 44 KB at address `0x200F000` the stack and other dynamic memory regions should start. Indeed as can be seen in Fig. 4.8 the vector table and other parts of the executable are where they can be expected. It can also be observed that non written to memory is set to `0xFF` in the `.text` segment of the binary. The RAM section of memory is also correctly listed in the binary as can be seen in Fig. 4.8. Radare 2 was used to debug memory layout issues and to better comprehend the in memory placement of RIOT. For example it is apparant that a non dynamic memory layout for RAM and ROM in the simple case of `hello-world` is wasteful, restricting the maximal usable amount of RAM due to not resizable nature of the ROM region. The last used ROM address is `0x20006210` resulting in a net loss of 35 Kilobyte which as about 80% of the total ROM region. Other than this discovery the generated binaries seem to be correct and can be transferred to the MCU for onboard testing.

4.1.8 Flashing code to CC3200

The prerequisites for flashing to the CC3200 were configured as part of the board abstraction layer. In this chapter the aim is to demonstrate how the flash process operates and what wiring is required on the hardware. CC3200 can be configured into flashing mode by settings its Sense Of Power (SOP) pin to ground. This will allow out configuration to flash the hardware but require a manual restart for the executable to be loaded into memory. To overcome this issues, it is recommended to connect the SOP pin to the JTAG Clock pin

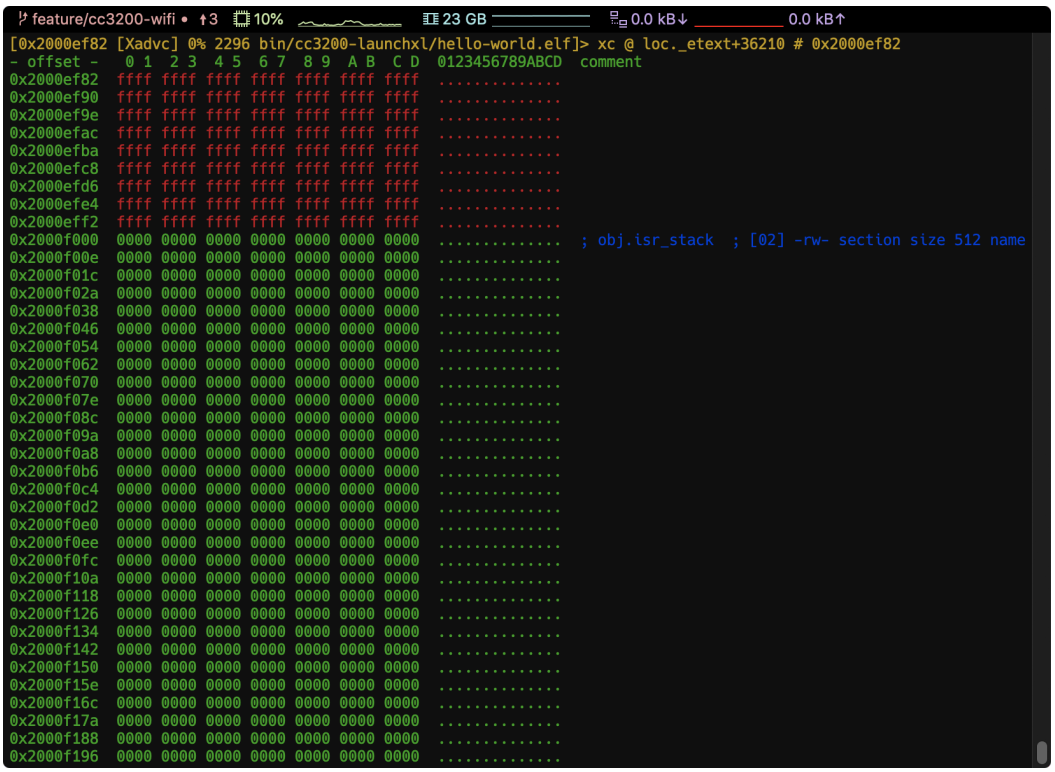

Figure 4.8: Radare 2 beginning of RIOT OS stack

which will alternate its signal after a completed flash and by that restarting the board. The pin connections for this are illustrated in Fig. 4.9 With the wiring setup the hello world example can be flashed to the board provided the `cc3200prog` is placed in a default location or the `ENERGIA_TOOL` environment variable is set.

```
$ BOARD=cc3200-launchxl make all build flash term
```

The flashing procedure may require up to four retries to successfully connect to the device via `UART` these retries are automatically performed by the `cc3200prog` tool. This operation still has a chance of failure depending on external factors and may require a retry on failure. With this configuration and preparation in place RIOT can be executed on the CC3200 and also successfully transferred to the system yielding a major milestone in controlling the NWP as a RIOT `netdev` device.

4.2 Network Processor Communication and Control

The CC3200 can be seen as a parallel system in itself, composed not only of single central CPU with "dumb" periphery devices. In addition to the host processor, executing RIOS OS thanks to the above efforts, there is an embedded secondary system managing Wi-Fi operation. This system is running an unspecified version of ThreadX on a secondary Cortex M3 cpu. Thus it is required to implement a system to system protocol. The protocol specifications while not publicly documented, were extracted from the available SimpleLink library included in the SDK. This chapter will describe the packets structure, transmis-

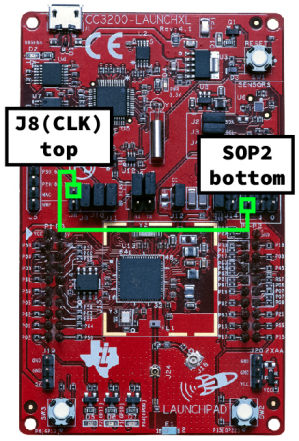

Figure 4.9: CC3200-launchxl preferred flash connection

sion algorithms and most important commands of the Host-To-Nwp command protocol and additionally explain the related implementation.

4.2.1 Network Processor communication protocol

The NWP can be communicated to utilizing SPI or UART, this implementation focuses on SPI. It should also be noted that NWP and MCU communication on the CC32xx family of boards supports DMA improving transfer speeds for larger data packets and most importantly removing freeing MCUs CPU for other tasks by essentially allowing both devices to directly write into each others memory. Due to time constraints and a focus on general operations this feature is not currently supported by the driver. This driver can potentially be used to connect any RIOT enabled board to a CC31xx NWP since the communication between a embedded CC3100 in the case of the CC3200 does not differ to the standalone CC3100. As both share the same SimpleLink library. Therefore the terms NWP and CC31xx will be used interchangeable in the following chapter. As mentioned in the WiFi certification the NWP is running its own operating system namely ThreadX, a real time operating system. Unfortunately no further information about the software executing on the NWP is given out to the general public which in itself opens up some concerns regarding security of the system. CC31xx family of boards communicate with a fixed word length of 32-bit and a clock rate varying depending on the generation and software revision of the NWP. The particular board used for the development of the driver is a CC3200 Rev. 1.3.2 with a supported SPI frequency of 20 MHz. While in theory since the MCU is operating as the SPI master device changes in the clock speed make it impossible to communicate with the MCU. Therefore it is essential to check for the device revision on driver setup or the driver will not operate. The revision of the CPU is stored in fixed memory location on CC32xx based systems but needs to be set when operating on third party boards. The bus speed is board dependant and thus is overwritten in the board configuration in addition to be provided as part of the driver itself. Initially the CC3200 could only operate at 13.3MHz but this speed was bumped to 20MHz in the currently available retail version. The followup

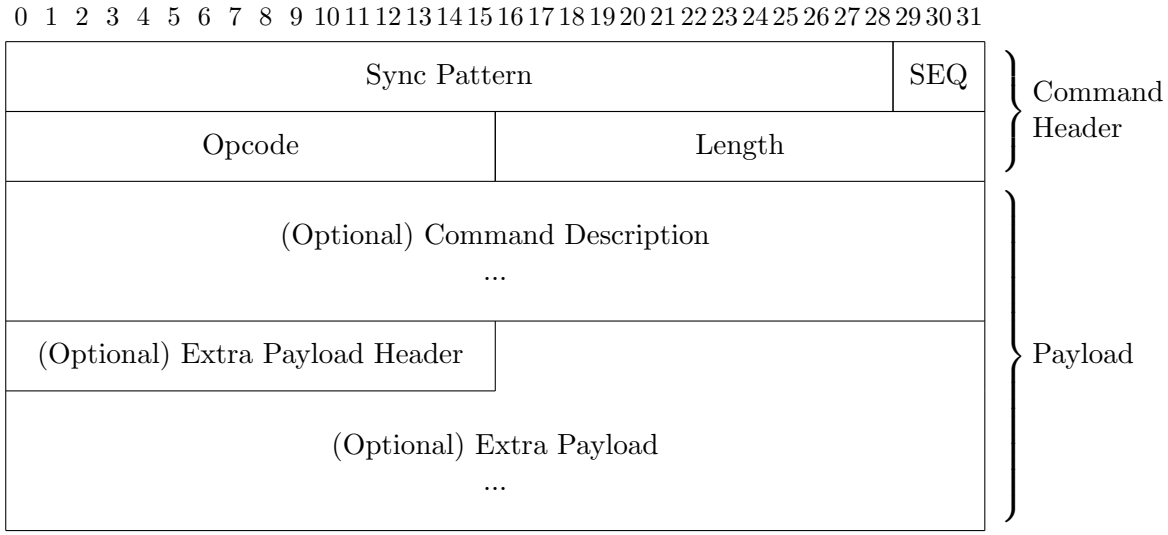

Figure 4.10: SimpleLink Request Command Packet Structure

board CC3220 supports speeds of upt 30MHz. The communication protocol is composed from three blocks firstly a short or long sync pattern depending on the current state of the device, secondly a generic header used for all packages and lastly and optional payload e.g. 802.11 frame. The sync pattern in addition to the long or short form can contain a sequency number counting up from the first transmitted command the NWP was last powered on. The sync pattern is a four octet long prefix to all transmissions from and to the NWP. The last 29 bits of the sync packet are used as the sync pattern. In the context of the NWP communication one word equals four octets or 32 bit. The remaining three bits of the word supply an optional sequency number of the packet. Therefore the sequency number is value in the range $[0, .., 3]$. The sync pattern is followed by a generic command header specifying the command type via a two octet operation code (Opcode) field and the length of the payload. The sequency number is ignored if bit 30 is set to zero. The layout of request command send by the host is illustrated in Fig. 4.10. Depending on the command at hand the payload can contain an additional command specific header. This structure is represented by the `cc31xx_nwp_header_t` in the implementation.

The actual transmission of words is handled by the SPI periphery abstraction layer discussed previously. Thus, before commencing the transmission it is required to obtain the SPI device using RIOTs `spi_acquire` call. The transmission itself utilizes `spi_transfer_bytes` to send and receive data from the Wi-Fi module. In regards to communication there are two cases to consider. The first one being the transmission of commands to the NWP. The transmission must occur after the NWP has finished its initialization routine. For simpler commands without payload headers the host transmits a packet as shown in Fig. 4.10. Some commands include a variable command description header at the beginning of the payload. Most often the command description is used to transmit values essential for execution of the command on the NWP comparable to parameters of a method. The command description header does not contain a length field since the size is fixed for a given Opcode. Additionally some commands require a variable length buffer or payload after the command descriptor.

The NWP protocol requires the payload to be prefixed with an additional extra payload header specifying the length of the extra payload.
The second transmission case is data send from the NWP. The NWP a comparable response packet illustrated in Fig. 4.11. The NWP does not start transmitting a packet before the host transmits a sync pattern to indicate the start of a read operation. At this point there is no guarantee that the next word from the NWP starts with the sync pattern. Thus requiring the host to be able to read multiple words and find the sync pattern even when the sync pattern starts at an offset inside a transmitted word. The host reads one word at time into a buffer, if the word contains the expected sync pattern the remainder of the packet can be read. When no sync is found the next word is read into the buffer. Since the sync can be at a byte offset smaller then the word length the sync match is performed in a loop from `buffer[0]` to `buffer[4]`. If the sync is not found in the next word the process is repeated. Depending on the buffer size it may be required to copy the last read word to the beginning of the buffer to continue this process. To prevent any additional allocations the `read_cmd_header` method performing this task uses the read buffer provided to it for the sync pattern search. If a sync is found a sequence counter is incremented. The sequency counter is used to validate the sequency field in the command header for future responses. The header layout of the response matches that of the request. The opcode of the response is defined as $opcode_{response} := opcode_{request} - 8000_{16}$, some request may not have a corresponding response. The response header features additional values for socket responses that are prefixed to all packets. Thus the minimal response packet size is 8 octets (6 octets of data plus 2 octets of padding). Frequently the NWP responds with a two octet long `cc31xx_nwp_basic_response_t` including a one octet response status code, used for error or success codes, followed by one octet of padding. The driver api provides an internal method to receive a command response from the NWP. The `uint8 recv_nwp_resp (cc31xx_nwp_resp * resp)` reads a command from the SPI connection. This method is not intended to be called directly rather is called by the `netdev event_callback` method, when a RX interrupt is triggered. Initially it was assumed that the NWP would trigger an interrupt in RAW socket mode as happens with TCP sockets. This is not the case for RAW sockets, therefore a additional thread is started requesting data from the NWP.
To simplify the response, request communication the method `uint8 send_req(cc31xx_nwp_req *req, cc31xx_nwp_resp * resp)` is provided. This method essentially combines the send and receive methods, it sends a given command. if `resp` is a non `NULL` value blocks the driver thread waiting for the response. This is especially useful when multiple configurations have to be transmitted in succession to each other, as will be useful for the NWP init.

4.2.2 Essential Commands and NWP initialization

The above requests and responses allow the exchange of messages, but what messages are required to manage the NWP. The messages implemented in the `cc31xx` driver are only a subset of all available commands and focus on the setup of the NWP, configuration of a MAC layer socket and the exchange of data. The NWP can by in a number of power safe modes or powered off. On boot the NWP has to be restarted to guarantee its configuration and state. While the NWP power on cannot be controlled directly the by the host, the Application Reset-Clock Manager (ARCM) provides a register which when set to one will eventually set the NWP to its power on state allowing it to be configured. This register is exposed by the ARCM abstraction struct `cc3200_arcm_t`defined in `cpu/cc3200/include/`

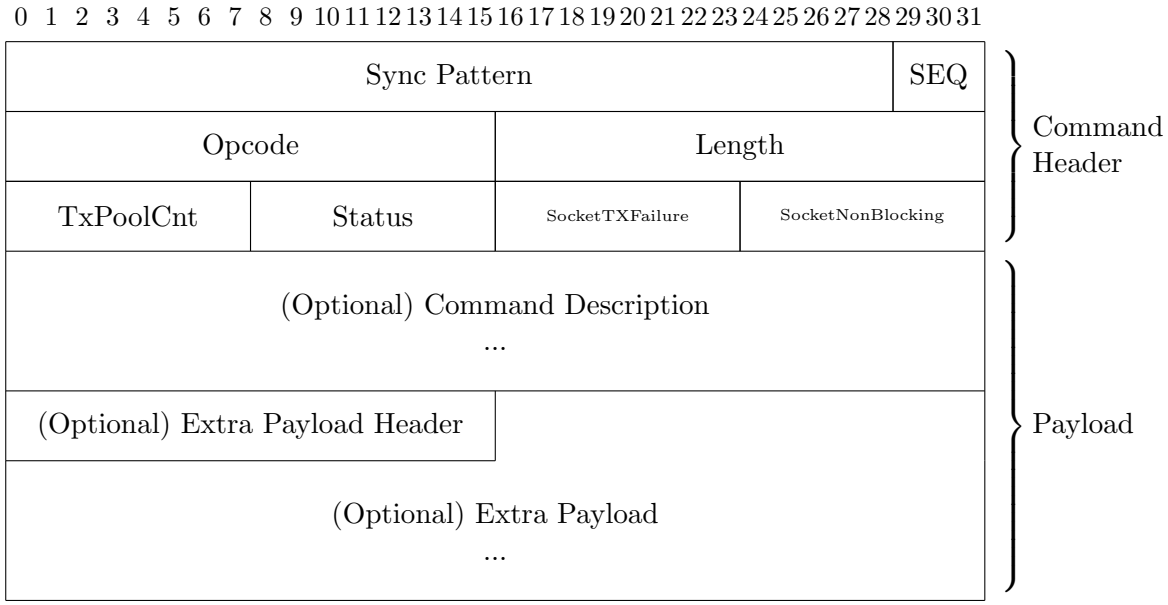

Figure 4.11: SimpleLink NWP Response Packet Structure

`cc3200_prcm.h`. The power on behavior of the cc3100 modules connected to a non CC32xx board differs and requires eventual GPIO pin configurations. On device initialization, after a brief power on period the driver registers a setup interrupt handler. It was initially planned to use the `netdev` interrupt handler directly, tests have shown that GNRC is not yet calling the callback when triggered at initialization.. Therefore a setup handler is registered initially and the actual `netdev` handler is assigned after the setup process completes. After the power up and NWP boot has completed the setup interrupt handler is triggered. In the NWP is operational a command with the `DEVICE_INITCOMPLETE` code can be read over SPI. With the NWP operational the driver configures the device
With the NWP initialized the Layer 2 socket can be initialized for this the NWP raw socket is used. As documented by TI the raw socket mode is mutually exclusive to other sockets and requires all Wi-Fi authentications to be disconnected. Testing with the cc3200 hardware show that NWP indeed prevents raw sockets to be open when the device is connected to a Wi-Fi network. This requires the implementation of the authentication process on the host. Thus, to open a raw socket all connection profiles stored on the NWP must be erased, all devices disconnected and auto-connect disabled. The connection profiles (password, BSSID and SSID) can be deleted via the `WLAN_PROFILEDELCOMMAND` command. This command requires the index of the profile to be deleted, to delete all profiles `0xFF`. After that the automatic connection and scanning features of the NWP must be turned of or the device may dynamically switch channels potentially disrupting the socket. The `WLAN_POLICYSETCOMMAND` is send to configure this features of the NWP. The policy configuration is configured via `cc3100_nwp_policy_set_t`. This will not disconnect the device from the currently connected network. To disconnect from any connected networks the `WLAN_WLANDISCONNECTCOMMAND` is used without any payload. The Wi-Fi module features extended IPv4 features like DHCP, these also need to be disabled. Additional NWP

features like Dynamic Host Configuration Protocol (DHCP) can be controlled with the `NETCFG_SET_COMMAND`. This command configures most extended features of the NWP. The feature is selected via an index e.g. 4 for DHCP. The `WLAN_CFG_SET` can be used to configure the power output of the NWP. For the purpose of this driver the power will be set to maximum to combat the abscence of a antenna on the unit. `WLANRXFILTERSETCOMMAND` is used to configure the receive filter allowing the incoming frames to be limited to the current device. The `open_socket` method wraps the `SOCKET_SOCKET` command and can open the required RAW socket or any othe socket type supported by the NWP. The socket required for the operation of the driver is the RAW socket also referred to as Transceiver Mode in the TI documentation. When creating the socket the Wi-Fi channel must be provided. This prevents the proposed implementation from scanning for available networks. This can be overcome by using the NWPs provided network scanner and after picking the network switching to the host implementation. If any of the above NWP configurations were not successful the raw socket command will fail with a negative error code. If the socket was opened successfully 128 is returned, representing the socket descriptor. With the setup completed the

interrupt handler is registered to NWP interrupt, leaving the driver in a operational state.

4.2.3 NWP Command Queue

All communications with the NWP are asynchronous and not guaranteed to be executed in the request order depending on the current state of the network module. For this reason a simple command queue was implemented. The idea being that when the host sends a command to the NWP in most cases a response is expected. For the simple proof of concept implementation parallel execution of such commands is disregarded and thus the command send operation blocks execution till the requires response command is triggered. The queue consists of `cc3100_drv_req_t` instances awaiting a response command. When an interrupt is triggered and the driver interrupt handler called by the GNRC event handler, the NWP command is read and depending if the command opcode matches one registered in the queue the queue object is deleted from the array. This will end the blocking loop awaiting a response and continue execution. This queue is initially used for device setup, since the operations must be performed in order and rapid succession.

4.3 IEEE 802.11 Implementation in RIOT

With the goal to integrate 802.11 into RIOTs modular structure a number of additions to the system are proposed. These additions are currently regarded as Work In Progress and are not considered to be fully operational. The goal here is to provide a foundation upon which a full IEEE802.11 implementation can be based on. To adhere to RIOTs structure the Wi-Fi integration was carefully modeled after the only other IEEE standard implemented in RIOT, 802.15.4. Being a wireless communication standard IEEE802.15.4 on the 2.4 Ghz band, it is comparable to Wi-Fi and thus considered to be reasonable template for it. This will require additional changes to the RIOTs system modules which will be described below, a full list of additions and alterations is shown in Fig. 4.12. Many of the additions are contained within folders and most others are simple extensions to Makefiles. The IEEE802.11 is contained within the `netdev_ieee80211` module and all requires submodules are imported automatically.

4.3.1 Picking a IEEE 802.11 specification

Before RIOT can be extended with 802.11, the version of 802.11 must be specified to avoid ambiguity. A small subset of IEEE 802.11g is set as the target due to its lower packet sizes fitting inside the MCUs RAM. In addition 802.11g has a reasonable support among currently available hardware in part caused by the backwards compatability defined in Wi-Fi 4 through Wi-Fi 6. 802.11g speeds are considered sufficient for this proove of concept. It is expected that the actual throughput of the device will be either way limited by the packet processing on the host and the data transfer to and from the NWP. The implementation incompases basic association, management and data transfer without never additions such as QoS or group ACK frames. With the 802.11 specification clarified, the addtion of IEEE802.11 to RIOT can commence.

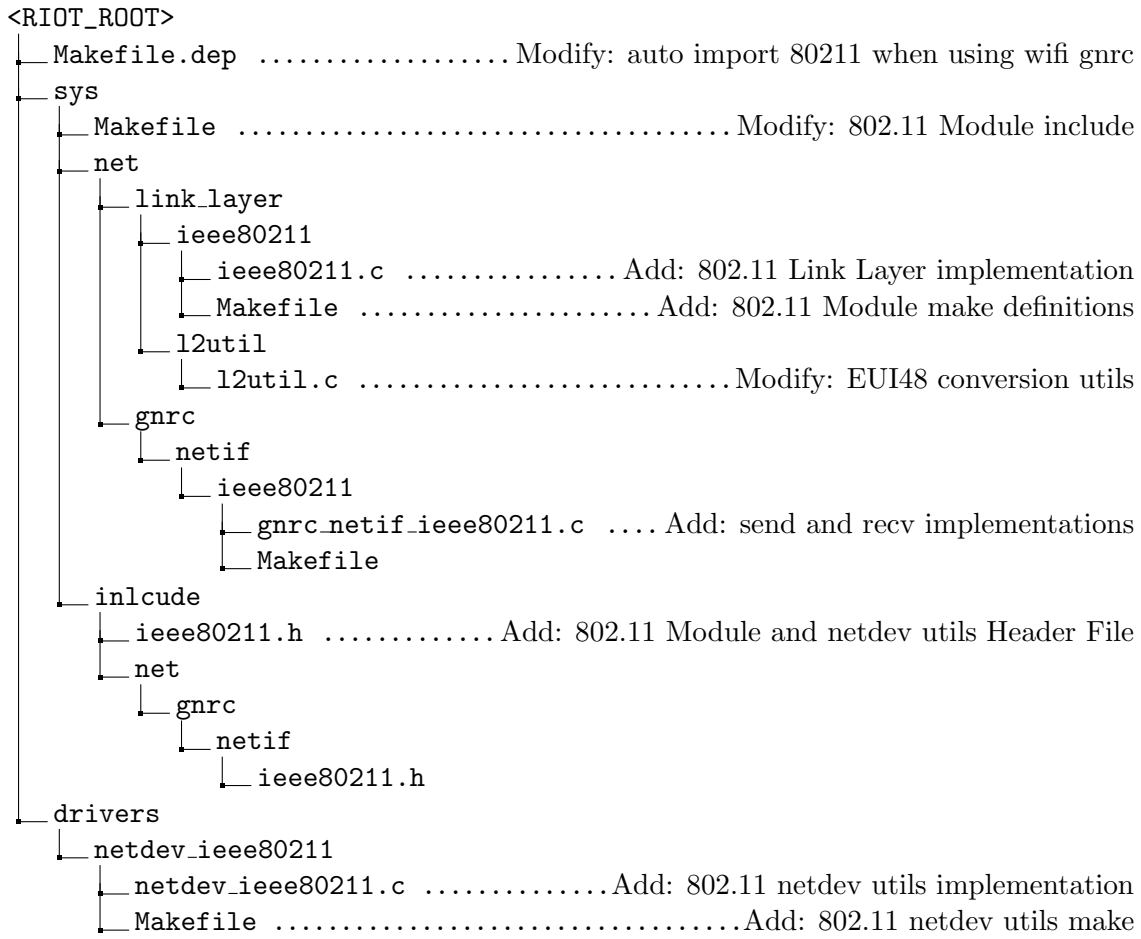

Figure 4.12: RIOT changes and additions required for 802.11 Link Layer support

4.3.2 IEEE 802.11 Link Layer

The Wi-Fi Link-Layer implementation is havily inspired by 802.15.4 RIOT integration. A overview of all additions can be seen in Fig. 4.12 inside the `link_layer` folder. This link layer implementation achieves one task: manage the MAC frame header. For this methods as `ieee80211_set_frame_hdr`, `ieee80211_get_frame_hdr_len` and `ieee80211_get_src` are provided. These methods simply extract or set data from and to the Layer 2 header. This additiona Link Layer implementation must be added to the `sys/Makefile` macro definition to include the 802.11 link layer if the `netdev_ieee80211` module is included. The `netdev_ieee80211` module provides basic MAC frame handling and defines 802.11 frame types and subtypes, defines available channels and the maximal MCU. Analogues to the `ieee802154` module the `ieee80211` module performs its Link Layer headers parsing and exposes essential methods to read and write MAC frame information via the `link_layer` submodule. `drivers/netdev_ieee80211` module provides a wrapper for accessing `ieee80211` module features via a `netdev` instance and adds Wi-Fi specific information to `netdev` by wrapping it in a `netdev_ieee80211_t` instance. This instance contains the configured channel, transmission power and mac information. MAC header construction is performed in the `ieee80211` module. The construction itself will be discussed in a future chapter.

```
int l2util_eui64_from_addr(int dev_type, const uint8_t *addr,
   size_t addr_len,  eui64_t *eui64)
{
    // ...
    switch (dev_type) {
       //  ...
#if defined(MODULE_NETDEV_IEEE80211)
        case NETDEV_TYPE_IEEE80211:
            ieee80211_get_iid(eui64, addr, addr_len);
            return sizeof(eui64_t);
#endif  /* defined(MODULE_NETDEV_IEEE80211) */
        }
    // ...
}
```

Figure 4.13: l2util 802.11 extension

4.3.3 IEEE 802.11 Extended Unique Identifier conversion

In the context of IoT IPv6 is gaining tramendios traction. Being a primarily IoT targeted OS, RIOT provides a deep IPv6 integration to the GNRC. To adhere to this mentality `netdev` devices must provide a 64-Bit Extended Unique Identifier (EUI-64) conversion. Because the 802.11 is a new protocol to RIOT it is required to extend the exisiting mappins it the utils file `l2util.c`. EUI-64 provides a globally or locally unique address which can depending on the network device can be generated from a unique hardware address. In the case of 802.11 it is specified that EUI-64 can be generated from a Layer 2 MAC address [RFC4291]. Adhering to the `netdev` protocol requires a EUI-64 conversion to be available for the netword device. As described in the background chapter a MAC address or EUI-48 is compsed from a OUI assigned by the IEEE and a NIC idenfier. The EUI-48 can determiniscly be converted to a EUI-64. The conversion is composed of three steps. Firtly the EUI-48 OUI is copied to the beginning and NIC to the end of the EUI-64, leaving a two octet hole between the two. Secondly the value `0xFFFE` is written to this segment of the address. This value was chosen by the IEEE because of its guarantee absence in the OUI allowing for a clear distinction. Lastly the global bit of the OUI in the resulting EUI-64 address must be flipped to form the modified EUI-64 also known as IPv6 Interface Identifiers (IID) [RFC7136, Chapter 1]. This bit flip is performed to simplify the manual input of local scope IPv6 addresses [RFC5342, Chapter 2]. The result is a fully qualified EUI-64 address that can be used as for example as a IPv6 address. This transformation in the context of RIOT is performed in the `l2util.c` file requering the EUI-48 converstion to be added there 4.13. The conversion intself can be found in `sys/include/net/ieee80211.h` and is demonstrated in 4.14.

4.3.4 IEEE 802.11 Header Construction

IEEE802.11 types are defined in the equally named `ieee80211` module as part of RIOTs network link layer implementation. The main IEEE 802.11 frame structure, configuration

```
static inline eui64_t *ieee80211_get_iid(eui64_t *eui64, const
    uint8_t *addr, size_t addr_len) {
    int i = 0;
    eui64->uint8[0] = eui64->uint8[1] = 0;
    // invert universal/local bit as of RFC4291
    // section 2.5.1
    eui64->uint8[0] = addr[i++] ^ 0x02;
    eui64->uint8[1] = addr[i++];
    eui64->uint8[2] = addr[i++];
    eui64->uint8[3] = 0xff;
    eui64->uint8[4] = 0xfe;
    eui64->uint8[5] = addr[i++];
    eui64->uint8[6] = addr[i++];
    eui64->uint8[7] = addr[i++];
    return eui64;
}
```

Figure 4.14: EUI48 to IID conversion

and other values are stored in the equally named `ieee80211.h` at `sys/net/`. Values from this file are used by the GNRC and netdev to construct appropriate buffers for the frame headers and fill the header fields. This module especially defines the previosly mentioned Wi-Fi frame and subframe types. All definitions are prepended with the `IEEE80211` prefix to clearly denote the Wi-Fi connection. When GNRC sends out or receives packets over the Wi-Fi network the associated netdev implementation is invoked which in terms uses the `ieee80211` module to compute header sizes and set or read appropriate header values. As covered in the background chapter the different frames depending on the type main omit fields of the MAC header. `ieee80211_get_src` and `ieee80211_get_dst` can extract the EUI48 identifier from the MAC header for later usage. When sending a packet `ieee80211_set_frame_hdr` to set the Frame Control and destination, source and the optional BSSID. This method is provided with a buffer fitting the maximum header size as defined by `IEEE80211_MAX_HDR_LEN`. At the end of the method the buffer contains a fully formed IEEE802.11 header corresponding to the specified frame and subframe types. The length of the resulting header is also returned by the method. Since the hardware prefixes the CRC field automatically on transmission and the body optional frame body will be set by other parts of the networking stack this concludes the ieee80211 protocl adherence. Therefore the extended GNRC can now create valid Wi-Fi frames ready for transmission.

4.4 CC31xx Driver

The driver utilized everything implemented thus far and implementes the `netdev` protocol to communicate with RIOTs GNRC. To some it may strange that a driver for the CC3200 platform is referred to as CC31xx, the reasoning behind this naming is the fact that the CC3200 platform uses essentially a on board CC31xx module wired direcly to the host processor. The documentation and SimpleLink driver for CC3100 and CC3200 are essentially

equal. Additionally the updated CC3120 in a prolimenary examination has shown to utilize the same communication and command protocols as the CC3100 by that attributing to the "xx" part of the driver naming. To facilitate the future posibility to extend this driver to be used with other host boards the more general CC31xx name is choosen. Per RIOT convention the `cc3100` driver is contained in an equally named folder inside `<RIOT_ROOT>/drivers`. To separate the hardware specific implementation from the IEEE 802.11 protocol in addtion to the before mentioned `netdev_ieee80211` module a generic netdev driver is added as well. The cleverly named `netdev_ieee80211` driver exposes `netdev` setters and getters and utils for address comparissons. While the CC31xx driver provides hardware specific setup and communication. The idea being if other 802.11 devices are added in the future they can reuse this base driver implementation. The architecture of the netdev communication is illustrated in the background and concept chapters of thsi very thesis. The full file structure with additions and changes is illustrated in Fig. 4.16 when observing the `driver` folder. The netden configuration is performed in the `cc3100_init` method. The setup method configures the `netdev`. The actual hardware setup is then perfomed on a separate `_gnrc_netif_thread` thread as part of the GNRC `netdev` initialization.

4.4.1 Netdev CC31xx Automatic Initialization

The `cc3100/cc3100_netif.c` provides adherence to the `netif` interface which in terms is used by GNRC. In order to automatically execute the cc3100 netdev initalization addtions to RIOTs `auto_init` process must be made. In detail a new auto init config specifying driver stack size, priority and the initialization process must be created. This task is performed by the `sys/auto_init/netif/auto_init_cc3100.c`. This file is also responsible for calling the driver `cc3100_setup` method discussed obave. The setup paramets and driver instance are then passed to `gnrc_netif_ieee80211_create` wich then finally creates a generic RIOT GRNC network interface. The configured network driver instance is at this point assigned to the `cc3100_params_t` attribute of the setup call. A generic RIOT driver has to expose six methods. These methods are stored in the `netdev_driver_t`. This driver instance is later used by the network interface to communicate with the driver, these are discussed in the following chapters. With the auto init setup it is only required to add the `auto_init_cc3100` call when the cc31xx module is bying used, the usage can be unfered by the presence of the `MODULE_CC3100` at compile time. Thus, this operation is only performed if the driver is included in the application. In RIOT this task is performed by the `sys/auto_init/auto_init.c`, hence the modification in that file. With this the driver will be automatically setup when its module is imported.

4.4.2 CC31xx Netdev Driver

The Automatic setup procedure will invoke a driver instance configured by the `cc31xx` driver module. For this a instance of the driver must exist in the first place. This `netdev_driver_t` is exposed by the `cc3100_netdev.c` ,which contains all `netdev` related source code. At its core the `netdev_driver_t` interface must implement six methods.

`send`	for sending a single frame of data
`recv`	for dropping or receiving a frame or inquire the length of a frmae
`init`	for initializing the netdev driver
`isr`	user space interrupt handler
`get`	reading an optional value from the network device e.g. network address or MAC layer protocol
`set`	setting an optional value of the network device e.g. transmission power or channel

These methods build the foundating of driver interaction and will be examined in a chronological order and not in the above implementation order. The GNRC after internal configuration will invoke the netdev driver `init` method. Here the cc31xx driver performs initial setup. The SPI connection is configured as part of the `cc31xx_conf_t` that must be set for the driver to operate. The driver then initializes the set `spi` interface and uses it to configure the NWP. Here the previosly mentioned NWP setup is performed and the `INT_NWPIC` interrupt is configured. Currently the interrupt implementation is assumptios to the interrupt configuration and only overwrites the `isr_nwp` previosly defined in the `vector.c`. Thus, the implementation to that regard is limited to the CC32xx family of boards, requiering minor adjustments to work with other MCUs. At this point the layer two socket is instanciated via the internal driver method `cc31xx_nwp_create_raw_sock` the resulting socket identifier is stored on the driver instance `cc31xx_t` as `sock_id`.
The `get` and `set` methods are then used by `netif` and GNRC implementations to configure the hardware and perform internal setup. To provide a specific example the `set` and `get` methods are being passed an `opt` parameter specifying a specific value to be read or modified. These values are defined in the `netopt_t` enum by RIOT. One if these is the `NETOPT_MAX_PDU_SIZE`. This option is send to the drivers `get` method to inquire the maximal packege size so that GNRC can compute needed memory for the network interface. Here the generic `netdev_ieee80211` driver finds it usage. Instead of implementing IEEE 802.11 related setters and getters here the set method of the cc31xx calls the generic driver first. If the generic implementation is setup to handle a action the cc31xx methods simply return the value. This implementation is demonstrated in Fig. 4.15. The `isr` method implementation simply invokes the `netdev` callback as described in the concept. `send` actually performs the send operations. Here the internal `cc31xx_nwp_send` is called to transfer the data frame generated by the IEEE 802.11 header implementation of the `ieee80211` module over the already discussed socket. The frame is potentially split into multiple NWP packets depending on payload size. The NWP then performs the transmission at its leasure and adheres to the 802.11 collision detection discussed in the background chapter as part of the MAC Layer. The `recv` implementation perform the countrary part. It is invoked by the GNRC event handler to read the incoming MAC frame from the NWP. This as well may require multiple packets if the frame is larger then the max NWP transmission unit.
Having implemented the full `netdev` interface the driver is now opeartional with GNRC and will be configured and setup automatically on boot.

```
static int _get( netdev_t *netdev, netopt_t opt,
                        void *val, size_t max_len) {
    // ...
    cc3100_t *dev = (cc3100_t *)netdev;

    // check generic ieee80211 implementation first
    int ext = netdev_ieee80211_get(
        &dev->netdev, opt, val, max_len
    );
    if (ext > 0) {
        return ext;
    }
    // ... (cc31xx specific handling)
}
```

```
 int netdev_ieee80211_get(netdev_ieee80211_t *dev,
                netopt_t opt, void *value, size_t max_len) {
    // ...
    switch (opt) {
        // ...
        /* compute size of the MAC PDU */
        case NETOPT_MAX_PDU_SIZE:
            assert(max_len >= sizeof(int16_t));
            *((uint16_t *)val) = IEEE80211_FRAME_LEN_MAX -
                                 IEEE80211_MAX_HDR_LEN;
            return sizeof(uint16_t);
        // ...
    }
    // ...
}
```

Figure 4.15: CC31xx driver netdev getter nested implementation

```
<RIOT_ROOT>
└── drivers
    ├── cc3100 ..............................Modify: append 802.11 Module include
    │   ├── include
    │   │   ├── cc3100_internal.h
    │   │   ├── cc3100_netdev.h
    │   │   ├── cc3100_registers.h
    │   │   └── cc3100_params.h
    │   ├── cc3100.c
    │   ├── cc3100_internal.c
    │   ├── cc3100_netdev.c
    │   └── Makefile
    └── include
        └── cc3100.h
```

Figure 4.16: RIOT cc3100 Driver file additions and modifications

4.4.3 CC31xx raw socket data transmission

On incoming netdev send commands the driver utilizes the previosly established raw socket to transfer the payload to the `NWP`. The NWP command based structure requires a single frame to be send in a single command. The generally used command parsing method is therefore not applicable to the send operation, since the `netdev` interface does not provide a single buffer with the whole frame but a linked list of buffers. To transfer this buffers a second NWP message transmission command was introduced as part of the `_nwp_send_raw_frame` method. This method uses the fact that the NWP will keep waiting for incoming data till the payload length send to it in the payload header was reched (or a new sync command was send). To transfer a multiple `iolist_t` items at once the same message is send to the NWP as previously described with the payload length set to the size of all `iolist` items. Unfortunately the current iolist implementation does not the full packet length requiring the linked list structure to be traversed one additional time to compute the size. For the purpose of this concept this is concidered reasonable. After this each item of this list is transmitted one by one.

4.5 Documentation

The codebase was written in accordance to RIOT OS formatting conventions. Especially the exclusion of vendor code was strongly enforced. The only TI or SimpleLink source files used for the compiled binary are the hardware definition header files stored in the repsective vendor folders of the modules requiring them. While massive changes to RIOT OS were conducted special care was put on folder structure and layout. Additionally the code is well documentated.

4.6 Summary and Findings

In this chapter two major chalanges and proposed implementational solution discussed. The first one being the general operation of the selected CC3200 device with the RIOT operating system. This required a number of CPU specific definitions, implementation of a numerious periphery abstractions like Timers, GPIO and SPI to name a few and also some board specific declarations. The thus resulting hardware abstraction layer is container within the `cpu/cc3200` and 0.6cpu/cc3200-launchxl folders as specified by RIOTs conventions. Surprisingly while exposing OpenOCD debugging features no way was found to flash user code to the board with the same framework. Therefore the setup and configuration of TIs propriatery `cc3200prog` tool was demonstrated as well as its integration into the RIOT make ecosystem. Having a compilable runnable and transferrable RIOT setup for the CC3200 the networking co-processor, cc3100, was examined more closely. The communication protocol was not included in TIs documentation, thus requiring it to be extracted from the TIs SimpleLink SDK. The communication to the NWP is performed over SPI using a command based approach. The implementation of this communication later was discussed as it is the foundating for later network data exchange. With all hardware specific implementations covered, the ingration of IEEE 802.11 into riots networking modules GNRC and `netdev` was demonstrated. This all was then combined into the driver itself, named `cc31xx` to symbolize its potential compatability with other SimpleLink modules.

5 Evaluation

The proposed IEEE 802.11 implementation and general board functionality is to be evaluated in this chapter. The proposed implementation and the driver hardware limitations prevent a full IEEE 802.11 test, therefore limited Wi-Fi functionality tests will be conducted. Including the reception and transmission of frames and the parsing of the core IEEE 802.11 MAC Layer headers. Additionally RIOT OS tests are performed to verify CC3200 board functionality.

5.1 CC3200 RIOT support

Porting the driver required the CC3200 to be operational first. To evaluate the operability of the implementation, RIOT OSs internal testing applications were used. The results of these tests are displayed below.

- **CPUID**: The CPU id of the CC3200 can successfully be read by the RIOT test
- **System Interrupt Handling**: The system interrupt handling test is completed successfully.
- **malloc**: The memory allocation test passes successfully.
- **SPI**: RIOS OS embedded SPI test was not executed and only compiler. The SPI connectivity was tested as part of the NWP communication.
- **UART**: The UART fails. UART communication from the CC3200 works. The UART interface does not trigger the read interrupt when data is send to the CC3200 over UART. Therefore, the shell tests were omitted and can also be expected to fail.
- **timer**: The periphery timer test passes successfully (The interactive UART input had to be disabled to pass this test).
- **xtimer**: Most xtimer tests pass with the provided periphery implementation. At unpredictable intervals a timer underflow can occur leading to missed timer callbacks.
- **GPIO**: GPIO was tested without the interrupt functionality. GPIO output is working and the connected LEDs can be controlled.

Most of the executed tests were completed successfully. A minor error in the timer implementation still occurs. This error leads to unpredictable failure states in some of the tests. The current assumption is that the error is due to timer underflow. This would mean, that the timer passes its interrupt value, before the interrupt match was configured leading to a differed or not performed interrupt.

5.2 CC31xx RIOT Driver

The driver for the Wi-Fi module of the CC3200 was named CC31xx to symbolize its more generic design and possible interoperability with other SimpleLink hardware. Internally, in regard to SimpleLink SDK features, the driver exposes most of the SimpleLink functionality rewritten to adhere to RIOT OS "no vendor code" and general coding standards. Table 5.1 lists the most important currently supported NWP commands and additionally shows their SimpleLink counterparts, if available. The `_nwp_send_raw_frame` and `_nwp_send_raw_frame` operations can still be performed with the SimpleLink driver but are not provided as standalone methods. Because these functionalities are frequently used by the driver, they were added to the implementation. While the preferred method of use, is combination with the IEEE 802.11 `netdev` driver, this implementation also allows to connect to Wi-Fi networks using the network stack of the NWP.
Additionally to the basic driver functionality, the reception and transmission of the driver was tested. For the following tests the Wi-Fi medium is monitored on a separate system using Wireshark, a communication monitoring tool. The IEEE 802.11 channel of the monitoring system was set to 11 to reducer network noise. Additionally, all CC3200 systems tested were configured to use the same channel.
Firstly the MAC Layer reception of the driver was tested. For that the CC3200 system was configured to dump all incoming traffic to the console. The results of this test are shown in Fig. 5.1. It can be seen, that the driver is able to receive Wi-Fi frames correctly.

Additionally, the transmission of generic data buffers by the driver is evaluated. For this a CC3200 system is configured to send a raw test frame via the CC31xx driver. The resulting Wi-Fi frame as captured by WireShark as shown in Fig. 5.2. The captured frame indicates that transfer was successful. To test communication between two systems, a second CC3200 board was configured to run the CC31xx driver and listen for the test frame. The second system uses the netdev and GNRC integration to parse the MAC Layer header of the received frame. The reception of the test frame was also possible on the second system. The MAC header was identified successfully. The output of system two is shown in Fig. 5.3.

With the basic communication tested, the driver was used to perform a network association process, as described in the background chapter. For this purpose an open Wi-Fi network was created, named "skynet". This network is configured without any encryption and operates on channel 11 with IEEE 802.11g. The CC3200 system was configured to initiate the authentication step with the "skynet" network. The resulting communication is shown in Fig. 5.4. The process was unsuccessful. While the network responded to the Authentication Request, sent by the client (first line in WireShark), the CC31xx system failed to acknowledge to response. This causes the network the resent the response seven times, which ends in a timeout. The ACK frame is only transmitted after all retries have already completed. The behavior was consistent for all repetitions of the test. Therefore it is assumed, that the current implementation is not capable of transmitting the ACK frame within the SIFS time period, required for managed Wi-Fi communication.

The ACK issue can be avoided using the successor to CC3200, the CC3220. The updated version allows the NWP to automatically transmit ACK frames. Preliminary tests with the CC3220 have shown that the CC3200 CPU and board abstractions layer also work for the updated hardware. While the NWP is connected in the same manner and the communication protocols on the surface seem comparable, some configurations need to be adjusted for the CC3220 to use the implemented driver. When testing the driver with the

cc31xx	SimpleLink	Operation
_nwp_set_net_cfg	sl_NetCfgSet	NWP embedded network stack configurations. For example DHCP enable, IPv4 configuration.
_nwp_get_net_cfg	sl_NetCfgGet	Inquire configuration values from the NWP, includes MAC address.
_nwp_set_wifi_cfg	sl_WlanSet	Configure Wi-Fi options. For example transmission power,
_nwp_set_wifi_policy	sl_WlanPolicySet	Configure Wi-Fi connection and reconnection policies.
_nwp_del_profile	sl_WlanProfileDel	Delete Wi-Fi connection profile.
_nwp_add_profile	sl_WlanProfileAdd	Add connection profile to the NWP, this profile can be then used to connect to a network.
_nwp_get_profile	sl_WlanProfileGet	Inquire connection profile values. Wi-Fi password is not included
_nwp_set_wifi_mode	sl_WlanSetMode	Configure the NWP to operate in access point or station mode.
_nwp_sock_create	sl_Sock	Create a new posix "like" NWP socket (RAW, TCP).
_nwp_send_raw_frame	-	Send raw MAC layer frame data.
_nwp_req_rcv_frame	-	Request the NWP to send the next MAC layer frame to the host.
_nwp_send_frame_to	sl_SendTo	Send data to a IPv4 address over a NWP TCP socket.
_nwp_set_wifi_filter	sl_WlanRxFilterAdd	Add a new receive filter to the NWP.

Table 5.1: CC31xx driver methods and their SimpleLink counterparts

Figure 5.1: Example NWP socket frame received via the CC31xx driver. Sensitive MAC addresses were redacted

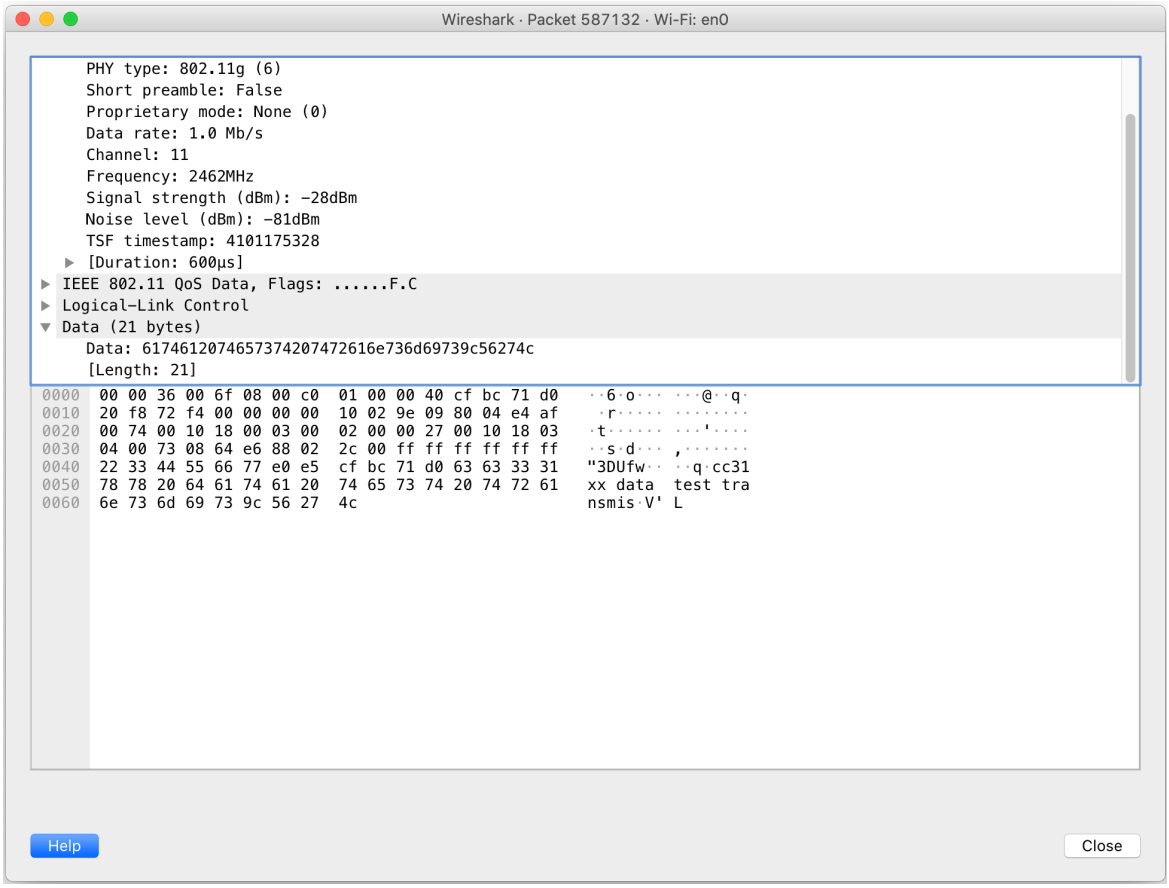

Figure 5.2: Test frame transmitted by the CC31xx driver

Figure 5.3: Reception of the frame on a second CC31xx board

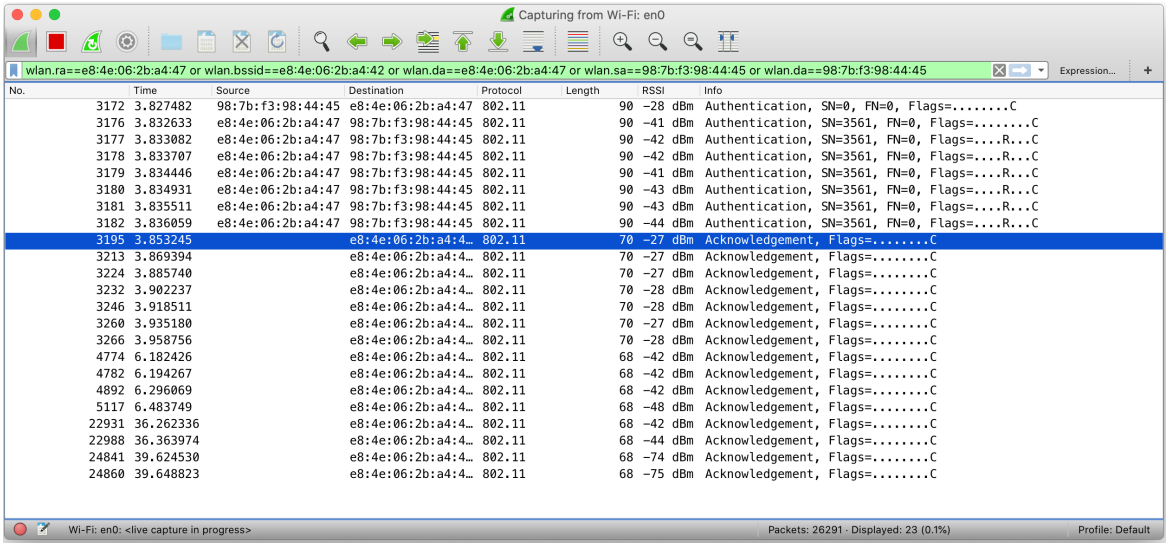

Figure 5.4: CC31xx open network authentication attempt

CC3220, the NWP does not send the boot completed command or trigger an interrupt on the host system. Potentially, this new feature could be back ported by Texas Instruments to the older CC31xx based platforms. Without any transparency to the code running on the NWP of both systems it is hard to tell if this is technically possible.
To summarize, the driver can send and receive IEEE 802.11 frames. Unfortunately the current implementation cannot transfer ACK frames within the short SIFS time frame. While the general performance of the implementation could be improved, the bottleneck is expected to be within the SPI based communication between host and the NWP. This renders commodity Wi-Fi data exchange impossible with the CC31xx driver. The current driver cannot perform a full authentication due to its inability to send ACK frames quickly. The transmission speed is so slow that all seven retries performed by the setup test AP timeout before the first ACK is transmitted. The test network could not be configured to extend the inter frame delay, to allow more testing. Further investigations must be conducted on the nature of this delay.

5.3 IEEE 802.11 in RIOT

The implementation of the IEEE 802.11 only covers basic header parsing. Therefore this implementation does not fulfill the full 802.11g standard targeted. Numerous omissions were made to the implementation. In the following the implemented and absent features are shown. The IEEE 802.11 integration, as mentioned in the implementation, closely mimics that of the already implemented IEEE 802.15.4, in code convention and structure. The current implementation includes all necessary files for a full IEEE 802.11 integration to RIOT. Leaving the following features open for future development.

- Dynamically configurable device capabilities. Currently the supported transmission speed, channel and other transmission parameters are fixed and cannot be changed dynamically. These must be connected to the netdev message structure. This potentially requires the driver socket to be recreated, because CC31xx opens a socket fixed to a single channel.

- Encryption support for the connection. The concept and implementation currently only support a un encrypted, open communication. The addition of encryption would require modifications to the header generation and potentially the frame handling. Currently no architecture for storing connection credentials is provided.

- The initially planned device discovery was left out from the implementation. The current implementation requires the `BSSID` to be known beforehand. In addition to the passive discovery a proactive approach could improve connection speed.

- Proper testing applications must be written, to test implementation coverage and detect problems. Additionally, RIOT OS network package provides numerous tests. All of these were omitted during the implementation process.

Summary and Findings

The provided board and CPU implementations supports RIOTs core features with numerous periphery extension, while still encountering minor timer related issues. The created cc31xx

driver for the NWP supports receive and send functionalities, due to its hardware limitations the driver cannot communicate with commodity grade Wi-Fi hardware in common networking topologies. The driver can transmit and receive valid Wi-Fi frames and pass these on to the `netdev` interface. The IEEE 802.11 implementation provides a foundation for future Wi-Fi operation for RIOT OS and already handles basic MAC Layer features. This layer can parse header information from a MAC layer frame but lacks full IEEE 802.11g coverage.

6 Conclusion

Wi-Fi support for Iot devices is a complicated endeavor. The goal of this thesis was shown to be potentially possible, but impractical in its current implementation, due to hardware and software constraints. This situation may alter depending on TIs future support for this platform. Some demonstrations by private companies have shown the usage of CC3200 hardware operation in IEEE 802.11s mode, which is not supported by the default driver and would also require strict ACK timings. With a mechanism in place, to update the CC3200 NWP, it can also be possible to flash the NWP itself with user defined code, offering the ability to execute the driver on the NWP itself and simply relay the Wi-Fi or raw data frames to the host processor. While not providing a definitive solution, there are viable options available to overcome the current communication limitations and make Wi-Fi available in RIOT and by that on many IoT devices.
A major implementation constraint is the absence of documentation on the SimpleLink NWP. The reasoning behind the missing documentation can only be speculated about. Many Wi-Fi capable devices do not expose their full functionality to the end user. It could be based on intellectual property restrictions or to some degree a belief in security by obscurity. Whatever the reason for the missing documentation of the communication interface is, its absence exponentially increased development time and complexity when developing the CC31xx driver. Many parts of the CC31xx driver rely on timings and specific data structure, not documented by Texas Instruments. The specifications had to be extracted from the SimpleLink SDK implementation. A detailed documentation on that manner would have substantially aided the development and potentially improved the quality of the resulting driver and its performance. Many of the commands and current configurations are based on experimentation and implications instead of hardware based facts, leading to an increased potential for errors.
The Wi-Fi additions to RIOT may also be of interest to the RIOT OS community and other developers seeking an open source Wi-Fi enabled operating system. The previously discussed GNRC and `netdev` additions provide a useful foundation for future expansion of IEEE 802.11 support in RIOT.
Combining Wi-Fi and RIOT would allow for new use cases within and beyond IoT applications. These Wi-Fi devices could be used to monitor ongoing traffic, while operating fully on battery or with sufficiently powerful hardware, even operate as a Wi-Fi AP and naturally opening the possibility for other IEEE 802.11 extensions to be implemented. For example 802.11s could be implemented to add decentralizes Wi-Fi mesh networking to RIOT.
The board definitions created as part of this thesis are already submitted to the RIOT project and are currently in active discussion and potentially will be included in upcoming RIOT OS releases.

6.1 Future Work

The results of this thesis, serve as a foundation for future research to the field if IEEE 802.11 in the context of low power embedded systems. Many concepts of this paper require improvements and finalization to become a fully viable solution. Some of the resulting opportunities are described below.

- **IEEE 802.11 implementation in RIOT**
  The foundation for IEEE 802.11 was layed out as part of this concept. As previously mentioned the implementation is in no way complete. The current concept provides the necessary extensions to RIOT to allow for a fully fledged IEEE 802.11 coverage. Additional frame types can be added to improve transmission efficiency and allow for QoS support.

- **IEEE 802.11ah**
  The IEEE 802.11ah standard extension could be added to the 802.11 implementation. This addition offers improved association time, throughput, latency and coverage range, when compared to IEEE 802.15.4, a popular low power wireless standard utilized in protocols like ZigBee. The extended range and other features would allow for communications of up to 1 kilometer, massively extending the use cases of enabled systems [ARH16] .

- **CC3220 RIOT support**
  The successor the TIs CC3200, the CC3220, offers numerous advantages. Firstly this new device provides advanced ACK control and therefore possibly offers a solution to the ACK latency problems. The CC3220S-launchxl was briefly tested with the provided CC3200 board implementation. Essential RIOT features were functional, but the communication to the NWP could not be controlled with the proposed driver implementation. As the goal of this thesis was the CC3200 support, the updated board implementation was left for future work.

- **CC31xx driver expansion**
  In its current form the provided driver cannot be used with the CC3100 and CC3120 standalone Wi-Fi expansion boards. The SimpleLink driver for these platforms is identical to that used in the embedded version, found on the CC3200. The current driver implementation could be further decoupled from the CC3200 platform, by implementing configurable communication interfaces. With this the driver could be used with non TI SimpleLink boards and further extend the usability of Wi-Fi in the RIOT ecosystem.

Conclusion

This thesis sets out to perform an ambitious task, adding Wi-Fi to RIOT with the CC3200 board. Unfortunately this task was shown to be not fully achievable in the context of this implementation. While the whole idea of software defined Wi-Fi is not fully of the table, its practicality on devices that are not designed for that form of operation is at question. This of course strongly varies from platform to platform.
While the goal itself was not reachable, multiple useful byproducts were created. Firstly a

new board and CPU combination were added to RIOT. Additionally to the CC3200 it was also discovered that this port may also work for newer versions of the SimpleLink device family. Similarities between TI implemented boards were discovered potentially leading to a unified code base for TI systems in RIOT OS.
Finally a basis for a new networking interface was layed out, potentially offering a simplified development experience for future IEE 802.11 integration to RIOT OS. The results of this thesis hopefully aid the future adoption of IEEE 802.11 in the RIOTs ecosystem.

List of Figures

Bibliography

[03] "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Further Higher Data Rate Extension in the 2.4 GHz Band". In: *IEEE Std 802.11g-2003 (Amendment to IEEE Std 802.11, 1999 Edn. (Reaff 2003) as amended by IEEE Stds 802.11a-1999, 802.11b-1999, 802.11b-1999/Cor 1-2001, and 802.11d-2001)* (June 2003), pp. 1–104. DOI: 10.1109/IEEESTD.2003.94282.

[09a] "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC)and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput". In: *IEEE Std 802.11n-2009 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009)* (Oct. 2009), pp. 1–565. DOI: 10.1109/IEEESTD.2009.5307322.

[09b] *Wi-Fi Alliance — Organization*. https://web.archive.org/web/20090903004711/http://www.wi-fi.org/organization.php. Accessed: 2019-09-12. 2009.

[16] "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". In: *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (Dec. 2016), pp. 1–3534. DOI: 10.1109/IEEESTD.2016.7786995.

[17] *Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID)*. IEEE. Aug. 2017. URL: https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/tutorials/eui.pdf.

[18a] *CC3200 SimpleLink Wi-Fi and Internet-ofThings Solution, a Single Chip Wireless MCU: Technical Reference Manual*. Texas Instruments Incorporated. Texas Instruments Incorporated, May 2018. URL: http://www.ti.com/lit/ug/swru367d/swru367d.pdf.

[18b] *Fi Alliance® introduces Wi-Fi 6*. 2018. URL: https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-wi-fi-6.

[19a] *history — wi-fi alliance 2019*. 2019. URL: https://www.wi-fi.org/who-we-are/history.

[19b] *Netdev - Network Device Driver API*. 2019. URL: https://riot-os.org/api/group__drivers__netdev__api.html.

[19c] *RIOT GNRC network stack*. RIOT OS, 2019. URL: https://riot-os.org/api/group__net__gnrc.html.

[ARH16] N. Ahmed, H. Rahman, and Md.I. Hussain. "A comparison of 802.11ah and 802.15.4 for IoT". In: *ICT Express* 2.3 (2016). Special Issue on ICT Convergence in the Internet of Things (IoT), pp. 100–102. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2016.07.003. URL: http://www.sciencedirect.com/science/article/pii/S2405959516300650.

[Bac+18] E. Baccelli et al. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT". In: *IEEE Internet of Things Journal* 5.6 (Dec. 2018), pp. 4428–4440. DOI: 10.1109/JIOT.2018.2815038.

[Hie+10] G. R. Hiertz et al. "The IEEE 802.11 universe". In: *IEEE Communications Magazine* 48.1 (Jan. 2010), pp. 62–70. DOI: 10.1109/MCOM.2010.5394032.

[IEEE Std 802a-2003] "IEEE Draft Standard for Local and Metropolitan Area Networks: Overview and Architecture". In: *P802-REV/D1.6 (Revision of IEEE Std 802-2001, incorporating IEEE Std 802a-2003, and IEEE Std 802b-2004)* (Oct. 2013), pp. 1–68.

[Kho+19] E. Khorov et al. "A Tutorial on IEEE 802.11ax High Efficiency WLANs". In: *IEEE Communications Surveys Tutorials* 21.1 (Firstquarter 2019), pp. 197–216. DOI: 10.1109/COMST.2018.2871099.

[RFC4291] R. Hinden and S. Deering. *IP Version 6 Addressing Architecture*. RFC 4291. http://www.rfc-editor.org/rfc/rfc4291.txt. RFC Editor, Feb. 2006. URL: http://www.rfc-editor.org/rfc/rfc4291.txt.

[RFC5342] D. Eastlake. *IANA Considerations and IETF Protocol Usage for IEEE 802 Parameters*. RFC 5342. http://www.rfc-editor.org/rfc/rfc5342.txt. RFC Editor, Sept. 2008. URL: http://www.rfc-editor.org/rfc/rfc5342.txt.

[RFC7136] B. Carpenter and S. Jiang. *Significance of IPv6 Interface Identifiers*. RFC 7136. RFC Editor, Feb. 2014.

[Val+98] R. T. Valadas et al. "The infrared physical layer of the IEEE 802.11 standard for wireless local area networks". In: *IEEE Communications Magazine* 36.12 (Dec. 1998), pp. 107–112. DOI: 10.1109/35.735887.

[WSS09] H. Will, K. Schleiser, and J. Schiller. "A real-time kernel for wireless sensor networks employed in rescue scenarios". In: *2009 IEEE 34th Conference on Local Computer Networks*. Oct. 2009, pp. 834–841. DOI: 10.1109/LCN.2009.5355049.

[Yiu13] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition*. 3rd. Newton, MA, USA: Newnes, 2013. ISBN: 0124080820, 9780124080829.

Acronyms

AP Access Point. 4, 8–11, 22, 57, 59

ARCM Application Reset-Clock Manager. 39

BSS Basic Service Set. 9, 10, 22

BSSID Basic Service set ID. 9, 10, 40, 45

CF Contention Free. 8

CPU Central Processing Unit. vii, 2, 13, 14, 18, 20–22, 26, 28, 37, 52, 61

CRC cyclic redundancy check. 5, 6

CSMA/CD carrier sense multiple access with collision detection. 4, 5, 19

CTS Clear to Send. 8

CW Contention Window. 5, 6

DA Destination Address. 10

DCF Distributed Coordinated Function. 4, 5

DHCP Dynamic Host Configuration Protocol. 41

DIFS DCF inter-frame space. 5, 6

DMA Direct memory access. 18, 37

ELF Executable and Linkable Format. 35

EUI-48 48-Bit Extended Unique Identifier. 7, 9, 44

EUI-64 64-Bit Extended Unique Identifier. 44

FAT File Allocation Table. 25

FC Frame Control. 7, 21

FCS Frame check sequence. 6, 8, 19

GDB GNU Debugger. 29

GNRC Generic network stack. vii, 2, 14, 15, 18, 22, 23, 25, 40, 42, 44–47, 50, 52, 59

GPIO General Purpose Input Output. 2, 13, 23, 26, 30–33

IEEE Institute of Electrical and Electronics Engineers. 3, 4, 7, 44

IID IPv6 Interface Identifiers. 44

IoT Internet of Things. 1, 2, 14

ISM Industrial Scientific Medical. 4

jtag Joint (European) Test Action Group. 19, 29, 35

MAC Medium Access Control. 2, 4, 5, 7, 8, 10, 17, 18, 20, 21, 43, 44

MCU Microcontroller. vii, 2, 3, 13, 17–20, 22, 23, 25, 29, 30, 33–35, 37, 42, 43, 47

MSDU MAC Service Data Unit. 7

MTU Maximum transmission unit. 20

NIC Network interface controller. 7, 44

NVIC Nested Vectored Interrupt Controller. 26

NWP Network Processor. 2, 12–14, 18–21, 23, 25, 26, 30, 33, 34, 36–42, 47, 49–52, 54, 57–60

OpenOCD Open On-Chip Debugger. 29, 30

OS Operating System. vii, 1, 14, 17

OUI Organizationally unique identifier. 7, 44

PHY Physical. 4, 5, 17, 18

PIFS point coordination function interframe space. 6

PLCP Physical Layer Convergence Procedure. 5

PMD Physical Medium Dependent. 5

POSIX Portable Operating System Interface. 2, 12

PPDU PLCP Protocol Data Unit. 5

Qos Quality of Service. 8, 20, 42, 60

RAM Random Access Memory. 26, 42

ROM Read Only Memory. 26

RTS Request to Send. 8

SDK Software Development Kit. 1, 12

SDR Software Defined Radio. 21

SIFS Short inter-frame space. 6, 52, 57

SoC System on a Chip. 11

SOP Sense Of Power. 35

SPI Serial Peripheral Interface. 2, 11–13, 18–20, 25, 30, 31, 33, 34, 37–39, 47, 57

SSID Service set ID. 9, 10, 40

SWD Serial Wire Debug. 29

TCP Transmission Control Protocol. 14

UART Universal asynchronous receiver-transmitter. 11–13, 18–20, 23, 25, 26, 30, 31, 33, 37

VoIP Voice Over IP. 8

WEP Wired Equivalent Privacy. 10

WLAN Wireless Local Area Network. 3, 4, 10, 17

WPA Wi-Fi Protected Access. 11