
UNIVERSITÀ DEGLI STUDI DI PALERMO

Department of engineering

Ph.D. in Electronic and Telecommunication Engineering


# Design, Implementation and Experimental Evaluation of a Wireless MAC Processor over commercial WIFI cards

Settore Scientifico Disciplinare : ING-INF/03


PH.D. CANDIDATE
Garlisi Domenico

TUTOR
Ing. Ilenia Tinnirello

COORDINATORE
Prof. Ing. Giovanni Garbo



XXIV cycle - 2013/2014


http://wmp.tti.unipa.it

Work carried out under the EU Project FLAVIA

# Abstract

Wireless networks importance for the future Internet is raising at a fast pace as mobile devices increasingly become its entry point. Wireless local area networks are becoming more and more important for providing wideband wireless local access thanks to the impressive success of cheap technologies, such as the IEEE 802.11, working on unlicensed bands. Because of the different application and networking scenarios, several standard extensions have been added to the original 802.11 technology (born as a standard for just cable replacement) with significant standardization efforts. Indeed, wireless service scenarios and application contexts evolve continuously and in an unpredictable way, and require significant and *fast* amendments of the underlying protocols.

In this thesis we face the problem of wireless network programmability as a solution for coping with context-dependent optimizations, moving from one-for-all standard solutions to the concept of programmable wireless interfaces. Although the wireless research and academic community has proposed interesting platforms (e.g. based on Software Defined Radio) for pushing forward dynamic reprogrammability of devices, we argue that it is important to identify a tradeoff between programmability space and usability of the programmable interface.

In this direction, we introduce the concept of Wireless MAC Processor and developed a running prototype over an ultra-cheap wireless card. Wireless MAC processor is a programmable device which provides a set of stateless Medium Access Control commands, and which embeds a MAC protocol engine in charge of executing a finite state machine able to exploit and compose the sequence of commands forming a desired MAC protocol logic. Wireless MAC Processor (WMP) commands can be considered analogous to the instruction set of an ordinary CPU. They are meant to implement elementary actions, namely MAC operations such as transmit a frame, set timers, etc., which may be then executed in the appropriate sequence and/or under the occurrence of specific events and conditions mandated by a protocol logic.

Instead of implementing a specific Medium Access Control (MAC) protocol stack, Wireless MAC processors do support a set of Medium Access Control which can be run-time composed (programmed) through Finite State Machine (FSM), thus providing the desired MAC protocol operation. Flexibility and ease of programmability is thus a consequence of the clear architecture-level decoupling made between what the device is able to do (the pre-installed MAC commands), and what it is instructed at run time, to do (the injected state machine).

The WMP platform has been designed and developed within the European project FLAVIA. The WMP implementation has also been released to the research community, together with a graphical tool for defining and compiling MAC state machines.

In the first chapter of This document we focus the concept of MAC protocol, on the motivation for introducing MAC-level programmability and on the discussion of the state of the art. The second chapter introduce the WMP architecture. We show the elements that compose the WMP and we present the Application Programming Interface, defined in terms of events, conditions and actions available for defining the MAC programs in terms of Extended Finite State Machines (eXtended Finite Sstate Machine (XFSM)). In the third chapter we show the MAC processor implementation details and we discuss some MAC program examples. In chapter four we describe the envisioned API and in chapter five we present the developing tools. In chapter six we explain the implementation of the proposed system over a commodity card (the off-the-shelf Broadcom AirForce54G chipset), by replacing its 802.11 WLAN MAC firmware implementation with our MAC protocol engine. At the end, in chapter seven we present some validation results, where we discuss some real experiments of on-the-fly MAC layer reprogrammability.
In the last chapter, we show as possible add flexibility also other layer, in this case we present **wpa_fast** , a own framework, developed to add flexibility in the other MAC layer functions. **wpa_fast** improve the performance for the nodes that work in environments that changing mostly rapidly as vehicular network.

# Acronyms

| | |
|---|---|
| AP | Access Point |
| API | Application programming interface |
| BSS | Basic Service Set |
| CCA | Clear Channel Assessment |
| CSMA | Carrier Sense Multiple Access |
| DCF | Distribute coordination Function |
| DCLS | Direct Channel Link Setup |
| DHCP | Dynamic Host Configuration Protocol |
| DIFS | DCF Interframe Space |
| DL | Direct Link |
| DLS | Direct Link Setup |
| FSM | Finite State Machine |
| GPR | General Purpose Register |
| GPT | General Purpose Timer |
| IFS | Interframe Space |
| IP | Internet Protocol |
| LLC | Logical Link Control |
| MAC | Medium Access Control |
| MCU | MicroController Unit |
| NAV | Network Allocation Vector |
| NIC | Network interface controller |
| PHY | Physical layer |
| SHM | SHared Memory |
| SPR | Special Purpose Register |
| SPROM | Serial Programmable Read Only Memory |
| TDMA | Time Division Multiple Access |
| TSF | Timing Syncronization Function |
| WMP | Wireless MAC Processor |
| XFSM | eXtended Finite Sstate Machine |

# Contents

**Abstract** **2**
Acronyms . . . 3

**1 Motivations: Why MAC flexibility?** **6**
1.1 The MAC level (link layer) . . . 6
1.1.1 TDMA . . . 6
1.1.2 CSMA/CA . . . 7
1.1.3 Polling MAC . . . 9
1.2 MAC: Why adding flexibility . . . 9
1.3 Flexibility through FSM . . . 10
1.4 State of art . . . 11
1.4.1 Other solutions . . . 12
1.5 MAC flexibility with WMP . . . 14
1.5.1 WMP overview . . . 14
1.5.2 Solutions in comparison . . . 14

**2 The Wireless MAC Processor** **16**
2.1 Introduction . . . 16
2.2 Elements of a FSM . . . 16
2.3 The WMP concept . . . 18
2.3.1 WMP architecture . . . 20
2.3.2 WMP development tool . . . 20
2.3.3 WMP implementation on wireless card overview . . . 21

**3 WMP Architecture** **22**
3.1 General WMP Architecture . . . 22
3.2 The MAC-Engine . . . 25
3.3 WMP API introduction . . . 26
3.4 Byte-Code . . . 26
3.4.1 Structure of the Byte-Code . . . 27
3.4.2 Byte-Code of the DCF State Machine . . . 30
3.5 WMP platform . . . 33
3.6 Binary-Byte-Code . . . 34
3.6.1 States region . . . 34
3.6.2 Transitions region . . . 36
3.6.3 State parameters region . . . 37

**4 API details: events, conditions and actions** **38**
4.1 State parameters . . . 40
4.2 Events . . . 41
4.2.1 Events to handle the TX . . . 42
4.2.2 Events to handle the RX . . . 42
4.2.3 Events to handle the timer . . . 42
4.3 Conditions . . . 43
4.3.1 Conditions to handle the TX . . . 43
4.3.2 Conditions to handle the RX . . . 43
4.3.3 Conditions to handle the parameters . . . 43
4.3.4 Conditions to handle the timer . . . 44
4.4 Actions . . . 44

4.4.1 Actions to handle the TX . . . 44
4.4.2 Actions to handle the RX . . . 45
4.4.3 Actions to handle the timer . . . 45
4.4.4 Actions to handle the parameters . . . 46

**5 WMP Development Tool 48**
5.1 WMP-Editor . . . 48
5.1.1 WMP-Editor element description . . . 49
5.1.2 WMP-Editor to define a MAC protocol . . . 49
5.2 From a model to a Binary-Byte-Code . . . 54
5.3 Byte-Code-Manager . . . 55
5.3.1 Byte-Code injection and activation . . . 55
5.3.2 Delayed Byte-Code switching . . . 59
5.3.3 Debugging options . . . 60
5.3.4 Forge control frame . . . 60

**6 WMP implementation on commodity hardware 62**
6.1 Introduction . . . 62
6.2 The Hardware Platform AirForce54G . . . 62
6.2.1 Platform : registers, memory addressing and subroutine system . . . 64
6.2.1.1 Special Purpose Register (SPR) . . . 64
6.2.1.2 Conditional registers . . . 66
6.2.2 Development tools . . . 66
6.3 WMP Implementation . . . 67

**7 MAC design and evaluation 70**
7.1 Design Overview . . . 70
7.2 Distributed Coordination Functions (DCF) . . . 70
7.2.1 DCF programmability . . . 73
7.2.2 DCF with Access Point feature . . . 73
7.2.3 DCF Experimental Results . . . 75
7.3 Time Division Multiple Access (TDMA) . . . 75
7.3.1 TDM Experimental Results . . . 76
7.4 Direct Link . . . 76
7.4.1 State parameter in D(C)LS . . . 79
7.4.2 DLS and DCLS performance evaluation . . . 79
7.5 Power consumption evaluation . . . 81

**8 High level flexibility 84**
8.1 Introduction . . . 84
8.2 Vehicular network . . . 86
8.3 WLANs Association Schema . . . 86
8.3.1 MAC level functionality . . . 87
8.3.2 Network level functionality . . . 88
8.4 WPA_FAST to improve the time connection se-tup . . . 88
8.5 WPA_FAST implementation . . . 89
8.6 Testebed set-up . . . 90
8.7 Experimental evaluation . . . 90

**A APPENDIX 94**
A.1 Hardware Deployment . . . 94
A.2 References and link . . . 94

# Chapter 1

# Motivations: Why MAC flexibility?

## 1.1 The MAC level (link layer)

In the seven-layer OSI model of computer networking, media access control (MAC) data communication protocol is a sublayer of the data link layer. The MAC sublayer provides addressing and channel access control mechanisms that make it possible for several terminals or network nodes to communicate within a multiple access network that incorporates a shared medium, e.g. Wireless [2].

The MAC sublayer acts as an interface between the logical link control (LLC) sublayer and the network's physical layer. The MAC layer emulates a full-duplex logical communication channel in a multi-point network. This channel may provide unicast, multicast or broadcast communication service.

A channel access mechanism is the part of the protocol which specifies how the node uses the medium (when to listen, when to transmit) and how it coexists with the other nodes belonging to the same network. In the next section we report same example of MAC, in detail we describe Time Division Multiple Access (TDMA), Carrier Sense Multiple Access (CSMA) and Polling which are the 3 main classes of channel access mechanisms for radio.

### 1.1.1 TDMA

TDMA (Time Division Multiplex Access) is very simple. A specific node, the base station, has the responsibility to coordinate the nodes of the network. The time on the channel is divided into time slots, which are generally of fixed size. Each node of the network is allocated a certain number of slots where it can transmit. Slots are usually organised in a frame, which is repeated on a regular basis [4].

The base station specifies in the beacon (a management frame) the organisation of the frame. Each node just needs to follow blindly the instructions of the base station. Very often, the frame is organised as downlink (base station to node) and uplink (node to base station) slots, and all the communications go through the base station. A service slot allows a node to request the allocation of a connection, by sending a connection request message in it. In some standards, uplink and downlink frames are one different frequencies, and the service slots might also be on a separate channel. The figure 1.1 summarizes a typical scenario of TDMA MAC protocol with a base station and two nodes, which highlights the slots of the nodes, but also the frame.

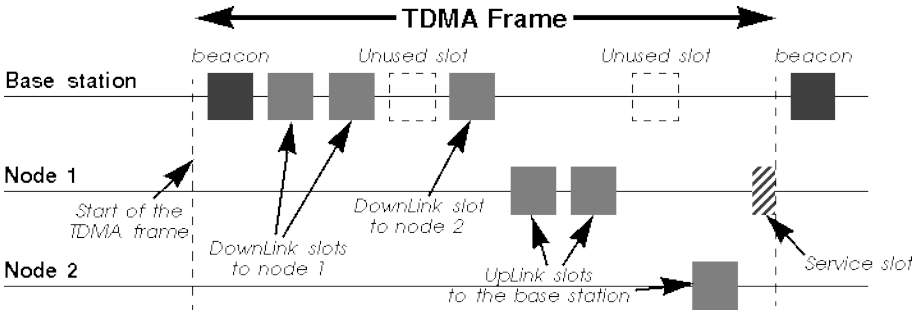



Figure 1.1: TDM MAC

TDMA suits very well CBR applications, such as voice calls, because those application have very predictable needs (fixed and identical bit rate). Each handset is allocated a downlink and a uplink slot of a fixed size (the size of the voice data for the duration of the frame). For these reasons TDMA is used into several cellular phone standards (GSM in Europe, TDMA and PCS in the USA) and cordless phone standards (DECT in Europe). TDMA is also very good to achieve low latency and provide bandwidth guarantees.

TDMA is not well suited for data networking applications, because it is very strict and inflexible. IP is connectionless and generates bursty traffic which is very unpredictable by nature, while TDMA is connection oriented (so there are some overheads for creating connections for single IP packets). TDMA use fixed size packets and usually symmetrical link, which doesn't suit IP that well (variable size packets).

TDMA performance are strongly affected by the interference conditions experienced on the allocated frequency band. In a dedicated clean band, as it is the case for cellular phone standard, TDMA is fine. Conversely, in unlincensed bands, because it does not explicitly take into account interference created by other networks, TDMA performance can be critical.

### 1.1.2 CSMA/CA

CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) is the channel access mechanism used by most wireless LANs in the ISM bands.

The basic principles of CSMA/CA are listen before talk and contention. This is an asynchronous message passing mechanism (connectionless), delivering a best effort service, but no bandwidth and latency guarantee. The protocol is very suitable for data bursty applications and is quite robust against interferences.

CSMA/CA is fundamentally different from the channel access mechanism used by cellular phone systems.

CSMA/CA is derived from CSMA/CD (Collision Detection), which is the base of Ethernet. The main difference is the collision avoidance: on a wire, the transceiver has the ability to listen while transmitting and so to detect collisions (with a wire all transmissions have approximately the same power). But, even if a radio node could listen on the channel while transmitting, the power of its own transmissions would mask all other signals on the air. So, the protocol can't directly detect collisions and only tries to avoid them.

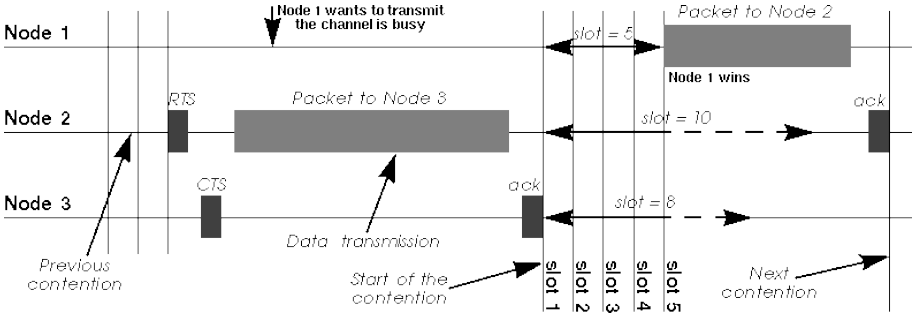



Figure 1.2: CSMA MAC

The protocol starts by listening on the channel (this is called carrier sense), and if it is found to be idle, it sends the first packet in the transmit queue. If it is busy (either another node transmission or interference), the node waits the end of the current transmission and then starts the contention (wait a random amount of time). When its contention timer expires, if the channel is still idle, the node sends the packet. The node having chosen the shortest contention delay wins and transmits its packet. The other nodes just wait for the next contention (at the end of this packet). Because the contention is a random number and done for every packets, each node is given an equal chance to access the channel (on average - it is statistic). The figure 1.2 shows a typical scenario of CSMA/CA with three nodes, the node two transmit after a RTS/CTS procedure, in the RTS/CTS (Request to Send / Clear to Send) procedure, two little frame are exchange between the nodes, before the real frame, to reduce frame collisions introduced by the hidden node problem. The node 1 in figure 1.2, win the contention procedure because it extracts a period of 5 slot before to start the frame transmit, node 2 extracts 10 and node 3 extracts 8.

As we have mentioned, we can't detect collisions on the radio, and because the radio needs time to switch from receive to transmit, this contention is usually slotted (a transmission may start

only at the beginning of a slot : 9 $\mu s$ in 802.11g ). This makes the average contention delay larger, but reduces significantly the collisions (we can't totally avoid them).

**The basic IEEE 802.11 MAC layer** uses the distributed coordination function (DCF) to share the medium between multiple stations. (DCF) relies on CSMA/CA and optional 802.11 RTS/CTS to share the medium between stations [20]. DCF employs a CSMA/CA with binary exponential backoff algorithm.

The time interval between frames is called the Interframe Space (IFS). A STA shall determine that the medium is idle through the use of the CS function for the interval specified. Five different IFSs are defined to provide priority levels for access to the wireless media. Figure 1.3 shows some of these relationships.

- SIFS short interframe space
- PIFS PCF interframe space
- DIFS DCF interframe space
- AIFS arbitration interframe space (used by the QoS facility)
- EIFS extended interframe space

DCF requires a station wishing to transmit to listen for the channel status for a precise interval time, named DCF Interframe Space (DIFS). If the channel is found busy during the DIFS interval, the station defers its transmission. In a network where a number of stations contend for the wireless medium, if multiple stations sense the channel busy and defer their access, they will also virtually simultaneously find that the channel is released and then try to seize the channel. As a result, collisions may occur. In order to avoid such collisions, DCF also specifies random backoff, which forces a station to defer its access to the channel for an extra period.

The different IFSs shall be independent of the STA bit rate. The IFS timings are defined as time gaps on the medium, and the IFS timings except AIFS are fixed for each PHY (even in multirate-capable PHYs). The IFS values are determined from attributes specified by the PHY.

This figure 1.3 gives description of DCF on a time scale as shown below. As we see when a STA becomes ready to transmit it senses the medium for a period of DIFS. If it senses that the medium was busy during this interval, the STA defers transmission to later time that depends on the backoff time. The backoff time is calculated using the following relationship

$BackoffTime = Random() * aSlotTime$

$Random()$ is a value in the interval $[0, CW]$, where $CW$ take a value between aCWmin and aCWmax i.e. aCWmin < CW <aCWmax. The values of aCWmin and aCWmax are defined for PHY. Value of CW is incremented exponentially as shown below. Value for inter frame space are different for different standard, the table 1.1 report a detail parameter for main IEEE WIFI standards.

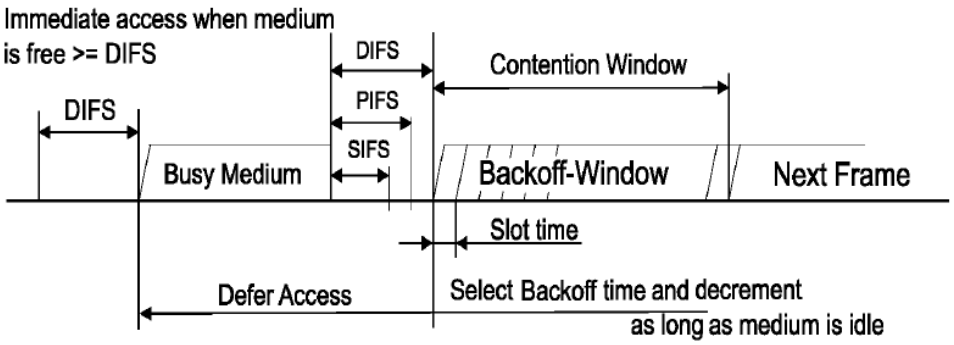



Figure 1.3: Inter Frame Space in 802.11 standards

| Parameters | 802.11b (HR/DSS) | 802.11g(OFDM) | 80211.a (OFDM) |
|---|---|---|---|
| $slottime$ | $20\mu$ | $9\mu$ | $9\mu$ |
| SIFS | $10\ \mu$ | $16\mu$ | $16\mu$ |
| PIFS | | $SIFS + t_{slot}$ | |
| DIFS | | $SIFS + (2 \times t_{slot})$ | |
| Operating Frequenzy | 2.4GHz | 2.4GHz | 5GHz |
| Maximum Data Rate | 11Mbps | 54Mbps | 54Mbps |
| CWmin | 31 | 15 | 15 |
| CWmax | 1023 | 1023 | 1023 |

Table 1.1: IEEE 802.11 parameters

### 1.1.3 Polling MAC

Polling is the third major channel access mechanism, after TDMA and CSMA/CA. The most successful networking standard using polling is 100vg (IEEE 802.12), but some wireless standard are also using it. For example, 802.11 offers a polling channel access mechanism (Point Coordination Function) in addition to the CSMA/CA one, that has been improved in the 802.11e extensions under the name of HCCA.

Polling is in fact in between TDMA and CSMA/CA. The base station retains total control over the channel, but the frame content is no more fixed, allowing variable size packets to be sent. The base station sends a specific packet (a poll packet) to trigger the transmission by the node. The node just waits to receive a poll packet, and upon reception sends what it has to transmit.

Polling can be implemented as a connection oriented service (very much like TDMA, but with higher flexibility in packet size and resource allocation) or connection less-service (asynchronous packet based). The base station can either poll permanently all the nodes of the network just to check if they have something to send (that is workable only with a very limited number of nodes), or the protocol use reservation slots where each node can request a connection or to transmit a packet (depending is the MAC protocol is connection oriented or not). The figure 1.4 shows an example of MAC polling with a base station and two node, any nodes transmit only after the poll frame sended by base station.

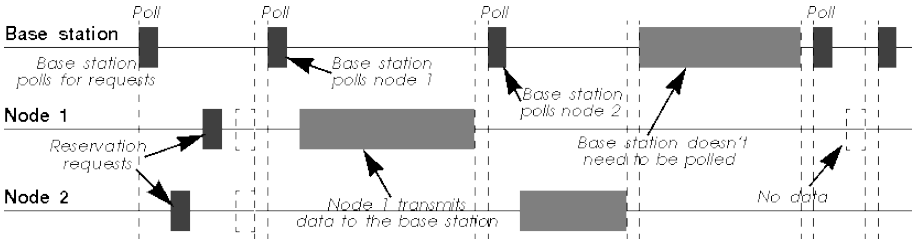



Figure 1.4: Polling MAC

In the case of 100vg, the polling mechanism doesn't use any bandwidth (it's done out of band through tones), leading to a very efficient use of the channel (over 96% user throughput). For 802.11 and wireless LAN, all the polling packets have to be transmitted over the air, generating much more overhead. More recent system use reservation slots, which is more flexible but still require significant overhead.

As CSMA/CA offers ad-hoc networking (no need of a base station) and similar performance, it is usually preferred in most wireless LANs. For example, most 802.11 vendors prefer to use the distributed mode (CSMA/CA) over the coordinated mode (polling).

## 1.2 MAC: Why adding flexibility

More than 20 years have elapsed since the establishment, in 1990, of the IEEE 802.11 Wireless Local Area Network committee. Initially foreseen as a technology for replacing Ethernet cables with wireless connectivity, IEEE 802.11 has been severely challenged by the highly heterogeneous needs emerged in the last two decades. Indeed, the original 802.11 CSMA/CA Medium Access Control (MAC) has shown significant shortcomings when facing the breakthrough rate improvements

made available by the latest PHY enhancements (802.11n, 802.11ac), as well as when applied to scenarios and contexts such as ad hoc and mesh networks, vehicular environments, directional antennas, quality of service support, real time media streaming support, multi-channel operation, dynamic spectrum access, and many others [5].

Actually, the WLAN research community has found effective and ingenious solutions for adapting the 802.11 MAC operation to these new challenges. However, as for instance detailed in a comprehensive analysis carried out in the frame of the FLAVIA FP7 European project [18], **most of the proposed MAC modifications do not comply with the 802.11 standard MAC operation**. In the best case, i.e. when the required MAC amendments are endorsed by some 802.11 standardization task groups, several years may elapse before they become available in commercial cards/devices. More frequently, when the promoted MAC amendments are either deemed out of the standard task groups scope, or mandate a way too significant departure from the native CSMA/CA MAC operation, their real world deployment is very unlikely, especially when they require changes in time-critical operations natively implemented in the network interface card.

Therefore, we need flexibility in MAC developing and maintenance. In order to insert flexibility when realize a MAC protocol, we need a model of the MAC protocol that optimize the development and maintenance. Therefore, a good modeling and abstraction method is essential, since it can significantly improve the quality and efficiency of the MAC. Another aspect that drives the development of MAC abstraction mechanisms is the ease-of-use and the degree of comfort that can be provided during the implementation and maintenance of it. One way to achieve this is to use a FSM to model the MAC.

## 1.3 Flexibility through FSM

A finite-state machine (FSM) is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a transition, a action can be execute when a transition is called. A particular FSM is defined by a list of its states, and the triggering condition for each transition[3].

The behavior of state machines can be observed in many devices in modern society which perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are vending machines which dispense products when the proper combination of coins is deposited, elevators which drop riders off at upper floors before going down, traffic lights which change sequence when cars are waiting, and combination locks which require the input of combination numbers in the proper order.

Finite-state machines can model a large number of problems, among which are electronic design automation, communication protocol design, language parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines have been used to describe neurological systems and in linguistics to describe the grammars of natural languages.

Figure 1.5 shows the parts of a finite state machine. The circles represent states. The arcs represent the transitions from one state to another. Each arc label is a condition. The slashes separate actions.

The proposed design approach, to realize a MAC protocol, is based on the finite state machine (FSM) mechanism. This mechanism fits the periodic structure of typical MAC applications. Thus, the transfer of a theoretical system description into a software implementation is easy to realize. Moreover, the FSM based abstraction makes it easier to split an algorithm into short segments (i.e. states), which can be developed and tested separately.

In order to improve the development process of MAC protocol programming and to handle the growing complexity of devices, different programming abstraction methods have been introduced. But, since each abstraction method brings certain advantages and drawbacks, developers often have to make a tradeoff between ease-of-use and efficiency, when selecting a programming mechanism[10]. Programming abstraction methods can be classified by a set of qualitative and quantitative parameters, which help developers to select an abstraction method that best fits their application. These parameters are Usability, Readability, Modularity, Reliability and Efficiency.

Modeling and abstraction methods should satisfy the following aspects, in order to bring an advantage during the software design and implementation process.

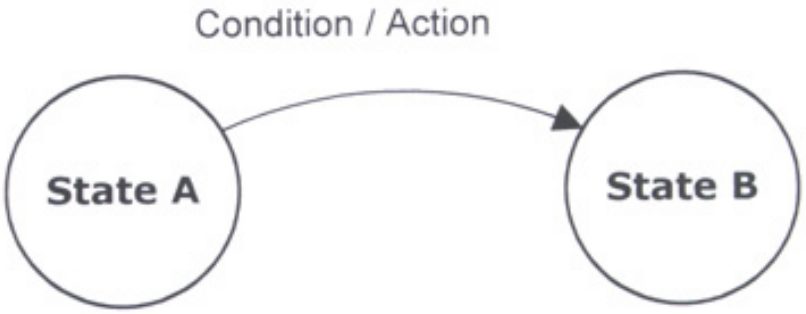



Figure 1.5: State machine example

1. Usability: A software abstraction method should help designers to formulate a real world problem in a standardized syntax. The syntax and code-structure should therefore be designed in such a manner that it is able to support a developer in understanding and formulating the problem at hand in a simple and time-efficient manner.

2. Readability: Enforcing a good software structure, the abstraction method can improve the readability of the resulting source code and therefore assists in the prevention of code errors.

3. Reliability: A good software abstraction method can help to detect and prevent invalid system behavior as soon as possible in the development process. Hence the embedded software can be implemented in such a way that it is possible to avoid a system lock-up or that it is possible for the system perform to an automatic recovery from invalid system states and lock-up conditions.

4. Efficiency: In order to make the best use of available resources, the generated code should have a small memory footprint and should create minimal runtime overhead.

An abstraction method, which very well fulfills the parameters mentioned previously, is the FSM mechanism, the idea is to use this method to develop the MAC protocol. Thus, the WMP architecture introduced in this thesis uses finite state machines as its basic working principle [10]. The following application examples emphasize the benefits to use FSM in WLAN applications.

- Flexible WLAN Applications: Wireless nodes often have to switch the currently running software according to the application scenario in which they are used. In sensor networks that rely on a dynamic mesh or a clustered topology the node must be capable of adapting to changes in the topology. In the that the system can handle different application scenarios and can provide different services, without requiring a modification of the hard software. No context switching between software modules is needed and so a full-scale real-time operating system, that comes with an enormous memory overhead, is not required. An FSM based implementation is well suited for these scenarios, since a top-level FSM can be used to select the required application, which is realized as a subFSM.

- Module Splitting: With the growing complexity of wireless applications, the embedded software used to run the systems becomes more advanced. In order to handle the system's complexity it is a common design practice to split an MAC protocol into sub modules, which can then be implemented and can be more easily maintained. In general these modules can be implemented as a sub-state machine and they are then called by the top-level FSM. Since the modules interact via well defined interfaces the different modules are interchangeable and can easily be reused.

## 1.4 State of art

A very first step to address the above concerns is to evolve from the current generation of closed network interface cards, implementing the very specific WLAN protocol stack, to programmable

WLAN platforms, capable of permitting software-based modifications in the wireless access operation. Obviously, the technical hurdle to face is how to support or modify time-critical operations which cannot be delegated to driver-level software modification, or controlled by dedicated overlay software modules running on the host computer [5].

The ability to modify the operation of commodity WLAN systems goes along with the availability of public-domain open-source 802.11 MAC protocol code. Besides the significant expertise required to modify existing code, a further deployment barrier for many appealing MAC extensions consists in the limited extent to which software changes may affect the device operation. Indeed, early 802.11 devices were designed according to a full-MAC approach. The MAC layer was almost entirely implemented in the card hardware/firmware, and programmability of the relevant drivers (when provided as open source) involved a marginal set of functionalities.

The flexibility of commodity WLAN cards has significantly improved since a number of vendors (including Intel, Ralink, Realtek, Atheros, Broadcom), started to exploit an innovative soft-MAC [22] design, transferring to the host processor non-time-critical MAC layer functionalities (figure 1.6).

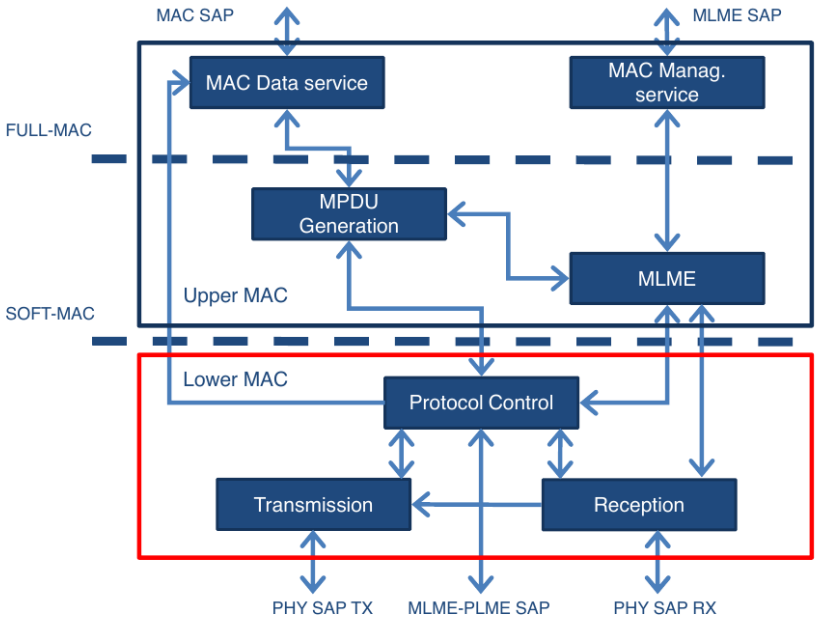



Figure 1.6: WLAN MAC architectures: PHY full-MAC vs soft-MAC.

Still, even in the soft-MAC case, the "Lower MAC", comprising crucial sub-systems such as transmission, reception and protocol control, remains hard-coded in the card.

### 1.4.1 Other solutions

Some chipsets (e.g. from Atheros and Broadcom) permit the tuning of selected MAC parameters (such as contention windows) via registers, more substantial MAC operation changes require access to the firmware code [5]. To the best of our knowledge, no vendor has to date released an open source firmware, and the only available public-domain code is OpenFWWF [14], a recently released simplified DCF firmware implementation for Broadcom/AirForce chipsets. However, OpenFWWF extensions require reimplementation of large portions of assembly code, thus making it usable only by experts.

In parallel, a significant effort has been spent on the development of overlay software modules. Solutions such as the Overlay MAC Project [23], MultiMAC [24], FlexMAC [25], Soft-TDMAC [26], etc, do exploit firmware configuration registers and some driver hacks for building quite advanced MAC programming interface (for instance, MultiMAC permits to override the frame format, disable ACKs, RTS/CTS, virtual carrier sense, disable transmission backoff, etc).

Even if notable implementations of custom MAC protocols, including TDMA-like ones, have been demonstrated, overlay approaches cannot get rid of some intrinsic limitations. Their scalability may be impaired by the need to overlap and duplicate similar functionalities at different layers; they remain constrained by the basic programming interface made available by the driver;

and their limited ability to accurately control the card's timing prevents to deploy features such as programmable management of frame replies and handshakes, precise scheduling of medium access times, fine-grained radio tuning control, etc.

The research community has mainly circumvented this issue by developing FPGA-based and/or Software Defined Radio platforms, such as [29], [30], [12], [13], [31], [27], [16], and has therein implemented and tested modifications to the wireless access operation.

Clearly, the shift from commodity wireless cards to dedicated wireless platforms permits to push programmability much farther, although the beneficiaries remain mainly confined within the research community - real world deployment of costly and/or bulky platforms being unlikely.

Early platforms such as CalRadio [29] reimplemented the 802.11 MAC protocol stack on, respectively, a Xilinx FPGA and a Texas DSP, interfaced to a commercial PHY-only Intersil 802.11b chip. As such, they permitted arbitrary MAC modifications, but protocol reconfiguration required a deep knowledge of the platforms and could only be done offline by recompiling the modified C code.

Software defined radio (SDR) platforms, such as GNURadio [12] and USRP [13], overcome the dependency on a specific PHY interface and permit to develop full-custom MAC/PHY cross-layer protocols. A large amount of work focuses on means to improve the slow SDR performance. On one side, solutions such as SORA [31] achieve a throughput comparable to commodity 802.11 hardware by distributing computation on multiple cores and by relying on sophisticated optimizations, as well as on an efficient radio control board. However, the software complexity makes protocol stack modifications not easy, as any update implies a redesign of the software block repartitions to multiple CPU cores.

On the other side, platforms such as WARP [16] and AirBlue [27] improve performance by delegating most processing functions to the FPGA Hardware, meanwhile retaining the ability to closely control such functions via, e.g., registration of interrupt handlers, hardware triggers, read-/write of hardware registers, etc. In the case of AirBlue, a modular organization coupled with careful design choices permits relatively easy modifications, changes in a module not affecting the others.

[32] and [33] start from a "breakdown" analysis devised to identify core MAC functions. Based on this, [32] proposes a split functionality architecture, where time-critical MAC functions are run on the radio hardware, but their control is kept on the host PC. The architecture is implemented over the GNURadio and USRP SDR platforms. Conversely, in the Decomposable MAC framework proposed in [33] and detailed in [34] both basic blocks and protocol logic are supported on a WARP platform. The MAC protocol is composed via a wiring engine which connects the basic blocks required to support the desired MAC operation.

The solution proposed in [33] uses MAC functions and support the MAC protocol logic directly on the radio card.

Different implementations of wireless systems used the concept of finite state machines. An over-the-air reprogramming protocol based on a state machine model is presented in [35]. In [36] a mechanism to derive a FSM based graphical description from existent TinyOS applications was designed. This approach is only usable for debugging purposes and does not improve the implementation or development process. Another approach is presented in [37], in which a state machine runtime environment has been implemented as a TinyOS component. This approach is rather simple and does not consider a hierarchical software structure. Hence the system is not usable in more complex applications [10].

Kasten and Römer introduced the OSM programming language and model, used to describe hierarchical and parallel finite state machines in [40]. The state flow is described using OSM code, which is, at a later stage, compiled into native C code. State and event functions, on the other hand, are directly implemented in C. An experimental OSM compiler was developed, but until now no full tool-chain has been published.

The Quantum framework [38] implements state machines based on an object oriented software framework in C++. The implementation realizes the state-flow by calling the different state handlers via a single function pointer. However, because the quantum framework enables parallel code execution, it requires a complex event posting and distribution mechanism.

The SenOS operating system [39] provides a runtime environment for a finite state machine based application development. The system kernel consists of a state sequencing and event queuing mechanism. But, instead of defining an application layer, each state directly enables a driver function in the callback library of the system.

## 1.5 MAC flexibility with WMP

### 1.5.1 WMP overview

The **Wireless MAC Processor (WMP)** is an architecture platform devised to run a wireless MAC program defined in terms of a **Finite State Machine (FSM)**, The WMP can be implemented on commodity hardware. The main element of WMP are: an execute finite state machine that named MAC-Engine and space memory to store 2 Binary-Byte-Code, the Binary-Byte-Code is a textual description of a FSM o MAC-Program [8]. It has been shown, in fact, that MAC protocols can be described in terms of state machines made of three main elements: actions, events and conditions.

In the WMP case, actions are commands for the radio hardware, such as transmit a frame, set a timer, and switch to a different frequency channel. Events include hardware interrupts such as channel up/down signals, indication of reception of specific frame types, expiration of timers and so on. Conditions are boolean expressions evaluated on internal configuration registers that can either explicitly updated by actions, or implicitly updated by events. Some registers are store general MAC layer information (like the current radio channel or the power level), or more specific MAC variables (like the contention window value and the backoff parameter).

Starting from an initial (default) state, the WMP waits for events which trigger state transitions. The actual transition can be enabled or disabled by verifying a boolean condition, while an action on the hardware system (i.e. on the transreceiver) can be performed before completing the transition to the new state.

We implemented the WMP on commercial card Broadcom AirForce54G to test the feasibility of the proposed solution, we obtain a proof of concept that good work and we use it to performance several experiment with different MAC protocol and different environment set-up.

### 1.5.2 Solutions in comparison

We clearly distinguish from related work in the previous section, unlike other works which rely on dedicated DSPs or programmable hardware platforms, we experimentally prove the feasibility of the wireless MAC processor concept over ultracheap commodity WLAN hardware cards. Specifically, we reflash the firmware of the commercial Broadcom AirForce54G off-the-shelf chipset, **replacing its** 802.11 WLAN MAC protocol implementation with our proposed extended **state machine execution engine**.

For these reasons, the WMP differs from off-the-shelf wireless NICs powered with their “vanilla” code: while the latter are tied to a specific MAC protocol (i.e., IEEE 802.11), the WMP architecture can run generic FSM, hence it can implement users’ designed MAC programs. On the basis of a predefined (hardware-dependent) set of actions, events and conditions which represent the platform API, a MAC programmer can easily compose different channel operations into a MAC program and execute it on the WMP.

[32] and [33] are probably the works more closely related to the Wireless MAC Processor approach presented in this thesis. Both start from a "breakdown" analysis devised to identify core MAC functions. The architecture is implemented over the GNURadio and USRP SDR platforms. The MAC protocol is composed via a wiring engine which connects the basic blocks required to support the desired MAC operation [5].

Similar to [33], we also support the MAC protocol logic directly on the radio card. However, our work differs from [32], [33] for at least three major aspects.

- Our breakdown analysis further includes events and conditions, in addition to MAC functions.
- We leverage injection in the radio hardware (and more specifically in the designed MAC protocol engine) of Extended Finite State Machines, thus permitting a greater flexibility as well as run-time re-programmability of the MAC operation without interrupting the MAC service.
- We rely on resource-constrained commodity WLAN cards instead of powerful and capable FPGA radio boards.

The WMP approach represents a major R&D leap from related work in this area since, unlike other works which rely on dedicated DSPs or programmable hardware platforms, we will experimentally prove:

- The feasibility of the wireless MAC processor concept over ultra-cheap commodity WLAN hardware cards.
- The possibility of programming time-critical medium access operations and event triggered PHY configurations without knowing any detail about the internal design of the hardware platform.

# Chapter 2

# The Wireless MAC Processor

## 2.1 Introduction

The **Wireless MAC Processor (WMP)** is a Finite State Machine (FSM) executor that runs inside a wireless Network interface controller (NIC): having direct access to the underlying hardware functions of the NIC, Radio, PHY and other facilities like timers, the finite state machine running in the WMP can be tailored to mimic a full featured Medium Access Control algorithm. This is achieved by exposing to the WMP a number of basic elements directly connected to the hardware, such as signaling from the Radio and the PHY that reports incoming frames from the air, and a few elementary actions, like frame passing to/from the radio. This approach shows many improvements with respect to classic implementations:

- Users can easily define new MACs by composing finite state machines using a graphical tool: thanks to a drag and drop interface, simple operations like frame sending or ack waiting can be composed into a complex MAC;
- Existing MAC can be easily modified/updated;
- Different MACs can be used in the same machine, either they can be selected for independent execution or they can coexist in the WMP at the same time and periodically activated (virtualization).

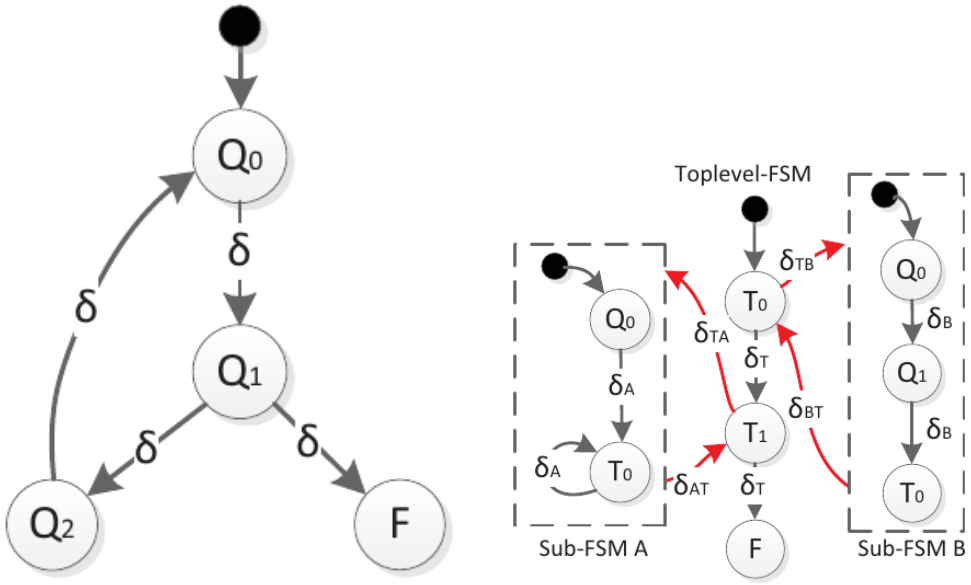
## 2.2 Elements of a FSM

A finite state machine is a model of a reactive system. The model defines a finite set of states and behaviour and how the system transitions from one state to another when certain events are triggered and conditions are true. Finite state machines are used to model complex logic in dynamic systems, from automatic transmissions to robotic systems to mobile phones. Examples of this complex logic include:

- Scheduling a sequence of tasks or steps for a system
- Defining fault detection, isolation, and recovery logic
- Supervising how to switch between different modes of operation

The formalism known as the finite state machine describes sequential logical systems and is frequently applied to describe algorithms in computer science, as well as to design digital electronic circuits. As previously mentioned, wireless systems can be implemented using FSM based design approaches [10]. A typical FSM (see 2.1) can either be represented graphically using state-charts [41] or can be formulated using a mathematical description [42] based on an 5-tuple that combines the following elements: $(\Sigma, Q, q0, \delta, F)$

- $\Sigma$ Input Alphabet (finite set of valid input events)
- Q Set of System-States
- q0 Initial State q0 $\epsilon$ Q
- $\delta$ Transition function $\delta$ : Q $\times\Sigma \rightarrow$ Q




(a) simple (b) complex

Figure 2.1: FSM

- F Set of Final States F $\subseteq$ Q

A hierarchical finite state machine, (see Figure 2.1) is a concept used to split large state machine definitions into sub-modules. The mathematical description can represent this splitting by describing the entire state machine as a set of sub-state machines Mhierarchical = { $M_0$ , $M_1$, $M_N$ } defines the system's top-level module, which contains the application's first entry point after the start-up. The 7-tuple ($\Sigma$, Q, q0, F, T, $\delta$, $\Sigma_s ub$ ) is derived from the description for non-hierarchical state machines. As can be seen below, every sub-FSM has to define a set of states that allow a transition and re-entry from and to other sub-state machines. Furthermore, a transition function which describes the program-flow between sub-state machines has to be defined.

- T States that allow entry/re-entry T $\subseteq$ Q
- $\Sigma_{sub}$ Transitions to sub-FSM $\Sigma_{sub}$ : $T_n$ x $\Sigma \longrightarrow T_M$

Hierarchical finite state machines can be applied in various wireless systems that require a complex system architecture. These systems often consist of multiple independent sub-modules, which have to interact via several well defined interfaces. An other form of state machine are the extended finite state machine, in a conventional finite state machine, the transition is associated with a set of input Boolean conditions and a set of output Boolean functions. In an extended finite state machine (XFSM) model, the transition can be expressed by an "if statement" consisting of a set of trigger conditions. If trigger conditions are all satisfied, the transition is fired, bringing the machine from the current state to the next state and performing the specified data operations. How explained in the section 2.3, the definition of the MAC logic in terms of extended finite state machines (XFSM) permits to control the actions executed by the hardware, for this reason we use this type to model the MAC protocol. The following application examples emphasize the benefits of FSMs in wireless applications.

**Wireless Applications** : Wireless system nodes often have to switch the currently running software according to the application scenario in which they are used. In wireless networks that rely on a dynamic mesh or a clustered topology the node must be capable of adapting to changes in the topology. In the design of end-user devices it is mandatory that the system can handle different application scenarios and can provide different services, without requiring a modification of the

embedded software. In this kind of system, several application modules exist in parallel in the MicroController Unit (MCU) memory but just one module is enabled at a time. Hence, no context switching between software modules is needed and so a full-scale real-time operating system, that comes with an enormous memory overhead, is not required. An HFSM based implementation is well suited for these scenarios, since a top-level FSM can be used to select the required application module, which is realized as a sub-FSM.

**Module Splitting** : With the growing complexity of wireless applications, the software used to run the systems becomes more advanced. In order to handle the system's complexity it is a common design practice to split an application into sub modules, which can then be implemented and can be more easily maintained. However due to the strong coupling between different segments in a typical embedded application it can be difficult to realize this kind of module splitting efficiently. An application based on a hierarchical state machine enables efficient module splitting in the different hardware abstraction layers, while avoiding the additional overhead of a full real-time kernel. A wireless system application has to perform tasks such as sensor sampling, data processing/storage and wireless communication. These modules can be implemented as a sub-state machine and they are then called by the top-level FSM. Since the modules interact via well defined interfaces the different modules are interchangeable and can easily be reused.

**Driver Implementation** : Additionally to the application layer, it can also be a good option to implement device drivers in the form of a state machine. Radio modules or intelligent sensor modules often require to trigger an interaction and then utilize the MCU to process the module's response, a behaviour which fits the FSM abstraction very well.

## 2.3 The WMP concept

For abstracting the hardware capabilities and defining hardware agnostic programs to be injected into the wireless cards, the WMP architecture implements a state machine execution engine, called the MAC engine, and a set of elementary actions and signals directly on the card. The MAC engine is an executor of MAC programs, (specified in terms of a high-level state machine) that can be ported on different card systems. The definition of the MAC logic in terms of extended finite state machines (XFSM) permits to control the actions executed by the hardware. The MAC protocol logic defines the medium access behavior whose evolution depends on events (frame arrivals, timer expiration, etc.) and condition verification on configuration registers [11]. The set of events generated and/or revealed by the card hardware, the set of actions coded in terms of predefined firmware modules, and the set of card registers whose settings can be tuned and verified represent the device API that cannot be modified by the user.

The proposed system architecture can be considered analogous to the instruction set of an ordinary CPU. They are meant to implement elementary actions, namely MAC operations such as transmit a frame, set a timer, freeze a backoff, etc, which may (or may not) be then executed in the appropriate sequence and/or under the occurrence of specific events and conditions mandated by a protocol logic. Going ahead with the same analogy, the MAC protocol engine can be somewhat related to an ordinary CPU control unit. It is in charge of executing an user-developed software program implementing the desired MAC protocol operation, which, in our proposed architecture, is provided in the form of an extended finite state machine. Flexibility and ease of programmability is thus a consequence of the clear architecture-level decoupling made between what the device is able to do (the pre-installed MAC commands), and what it is instructed - at run time - to do (the injected state machine). We describe the extended finite state machine through **events**, **conditions**, and **actions**. The theoretical approach followed in the realization of the WMP is that of a FSM with essentials like states and transitions enriched with three additional elements, namely events, conditions and actions that improve the flexibility of the resulting system. We report in chapter 4 all the elements that can be used define several different MAC programs.

MACs defined for the WMP can be considered following two abstraction layers: a textual one, where everything is described using text expressions, and a graphical one, where the state machine is described through a practical graph based approach. The former representation is the **Byte-Code**, a text file that can be either written at hand by users, or automatically generated by the **WMP-Editor**, a graphical tool that can be used to build the latter representation. This tool helps users composing new FSM, elements can be, in fact, connected together thanks to a straightforward drag and drop interface: furthermore it also checks for semantic errors during the design phase

and for this reason it is strongly recommended to use it and avoid to manually write a Byte-Code. The code isn't compiled but it is interpreted from MAC-Engine. The software for pushing the code to the MAC is called **Byte-Code-Manager**.

The Wireless MAC Processor architecture somewhat mimics the organization of ordinary computing systems [6], where programmability is accomplished by specifying

- an adequate instruction set which permit to perform elementary tasks on a machine;
- a programming language which conveys multiple instructions (suitably assembled to implement a desired behavior or algorithm) to the machine;
- a Central Processing Unit (CPU), which executes such program inside the machine, by fetching and invoking instructions, updating relevant registers, and so on.

**Instruction set: Actions, Events, Conditions** A breakdown analysis of MAC protocols reveals that they are well described in terms of three types of elementary building blocks: actions, events and conditions. Actions are commands acting on the radio hardware. In addition to ordinary arithmetic, logic, and memory related operations, dedicated actions implement atomic MAC functions such as transmit a frame, set a timer, build an header field, switch to a different frequency channel, etc.

**Actions** are not meant to be programmable. As the instruction set of an ordinary CPU, they are provided by the hardware vendor. The set of actions may be extended at will by the device vendor, and complex actions may be considered, so as actions not necessarily restricting to MAC primitives (e.g. perform a PHY encoding/decoding).

**Events** include hardware interrupts such as channel up/down signals, indication of reception of specific frame types, expiration of timers, signals conveyed from the higher layers such as a queued packet, and so on. As in the case of actions, also the list of supported events is a-priori provided by the hardware design.

**Conditions** are boolean expressions evaluated on internal configuration registers. These registers are either explicitly updated by actions, or implicitly updated by events.

Some registers are dedicated to store general MAC layer information (such as channel used, power level, queue length), frame related information (source or destination address, frame size, etc), or more specific MAC parameters (contention window, backoff parameters, etc - used to achieve a more compact protocol description in case of specific MAC designs such as CSMA-based ones).

Actions, events, and registers on which conditions may be set, form the application programming interface exposed to third party programmers. This API is implemented (in principle) once-for-all, meaning that programs may use such building blocks to compose a desired operation, but have no mean to modify them.

MAC protocols are well suited to be described in terms of Finite State Machines. **We chose to rely on the more powerful and expressive model of eXtended Finite State Machines (XFSM)**. XFSMs are a generalization of the finite state machine model and permit to conveniently control the actions performed by the MAC protocol as a consequence of the occurrence of events and conditions on configuration registers.

An XFSM is formally specified through an abstract 7-tuple (Q, $\Sigma$, O, D, F, U, $\delta$): the meaning of such symbolic states and the correspondence with the MAC terminology above introduced is summarized in Table 2.1 (configuration commands being a special case of actions, devised to update registry status). A MAC program is simply a table listing all possible state transition relations. Note that the number and meaning of the set of protocol states is specified by the programmer. By formally describing, per each protocol state, which events and conditions do trigger a state transition, and by associating actions and configuration commands to each state transition, the programmer may access the available hardware primitives, and enforce a desired MAC behavior within the radio hardware. Since the configuration memory is not explicitly represented in the state space, XFSMs allow to model complex protocols with relatively simple transitions and limited state space. For an example, the table programming the legacy 802.11 Distributed Coordination Function MAC protocol is coded in less than 350bytes (see section 3.6), and hence can be transmitted in just a single packet.

| Symbol | XFSM formal notation | meaning |
|---|---|---|
| Q | Symbolic states | MAC protocol states |
| $\Sigma$ | input symbols | Events |
| O | Output symbols | MAC actions |
| D | n-dimensional linear space D1 $\times \cdots \times$ Dn | all possible settings of n configuration registers |
| F | Set of enabling functions $f_i$ : D $\longrightarrow$ $\{0,1\}$ | Conditions to be veri fied on the configuration registers |
| U | Set of update functions $U_i$ : D $\longrightarrow$ D | Configuration commands, update registers' content |
| $\delta$ | Transition relation $\delta : Q \times F \times \Sigma \longrightarrow Q \times U \times O$ | Target state, actions and configuration commands associated to each transition |

Table 2.1: MAC programs expressed as Extensible Finite State Machines.

### 2.3.1 WMP architecture

The wireless MAC processor has been conceived as a CPU specialized for handling hardware/PHY events and actions by executing Extended Finite State Machines (XFSMs). The internal architecture of the WMP includes five main components:

- MAC-Engine: an execution engine, running the provided XFSMs;
- A memory block including both data and program memory space;
- An interruption block passing the signals coming from the hardware to the execution engine;
- Application programming interface (API) : a set of operations which can be invoked by the execution engine, which include logic, arithmetic and flow control operations plus specialized MAC operations;
- A set of registers for saving system state parameters.

We discuss every component present in the WMP architecture in the chapters 3.

### 2.3.2 WMP development tool

To avoid writing MAC programs in the above described machine language, we have developed an XFSM builder which includes a graphic XFSM editor developed in java language for composing MAC program, and a "Byte-Code compiler" which translates and XFSM graphical representation into the machine MAC program, a language understandable by the firmware's MAC engine. The Byte-Code can be loaded on the memory chipset by using a specific tool named Byte-Code-Manager that also performed a chipset debug tools. Loading a new Byte-Code on the chipset allows changing on-the-fly the card behavior without any recompiling operation. All these element belong to developing tools, in the specific it include:

- **WMP-Editor**, a graphical tool, working as an editor for composing a MAC program in terms of a graphical representation of state transitions and state labels;
- **Translate tool**, able to map the graphical representation into a textual transition table or Byte-Code;
- **Byte-Code-Manager** able to load, run, and verify the Byte-Code in the WMP platform.
- **Debug tools**, is part of Byte-Code-Manager and permit a useful debug.

The combination of the MAC-Engine, graphical editor, translate tool, Byte-Code-Manager is a complete and cheap tool-chain that allows developing and testing a new MAC scheme in a very simple, robust and quick way over an ultra-cheap platform. We have defined several model of MAC protocol, from common IEEE 802.11 DCF (**complaint with the IEEE standard**) to TDMA MAC protocol and multi channel protocol.

### 2.3.3 WMP implementation on wireless card overview

In this section we present an overview of the wireless card Broadcom "AIRFORCE 54G" in terms of hardware specified, we use this type of cards to implement the WMP platform. This 802.11g Wireless LAN PCI Card works with both IEEE 802.11b and IEEE 802.11g products. This card uses the Broadcom chipset Broadcom BCM94318KFBG and BCM94311KFBG, both cards are show in the figure 2.2. The BCM43xx is the third generation of Broadcom's single-chip wireless LAN solutions, which combine 2.4 GHz radio, 802.11a/g baseband processor, medium access controller (MAC) and other radio components onto a single chip. The cards are compatible with the linux kernel, and use the b43 driver, this driver is result of a active project, constantly updated. In this overview we present the main element on the card, while we refer more detail on the WMP implementation on this wireless card in the chapter 6, the main element present in the wireless card are:

1. **Broadcom CPU**: This is a 8 MHz CPU with 64 registers;
2. **Shared memory**: This memory space of 4 KB can be accessed also by the host and can be used for implementing the micro-instruction memory, i.e. the MAC program;
3. **Internal code memory**: This 32 KB memory is used for implementing the MAC commands and the MAC engine;
4. **Internal registers**: The internal registers keep hardware configuration settings. They can be set by the processor in response to changes in the program, and in the interface;

The Broadcom wireless card has a tools for debugging and compiling the source program code, all program are collect together in a project named b43-tool, all tools can be downloaded ad the address [17], and support different cards. Main tools perform the firmware compiling and the dump of the different memory and registers. The tool to compiling the source code is named b43-asm, other tool b43-fwdum shows the contain of the memory and registers.


(a) BCM4318 (b) BCM4311

Figure 2.2: Broadcom wireless card

# Chapter 3

# WMP Architecture

Wireless card architectures have tremendously evolved in the last years in order to transfer to the host processor the functionalities which do not require strict time constraints. While old devices were designed according to a full-MAC approach, where the control of all the MAC functionalities were entirely performed by the card, recent devices exploit an innovative soft-MAC approach, where important MAC primitives are supported by the driver on the host.

Our design starts from the consideration that most modern wireless cards do embed a general-purpose CPU for supporting the hardware control logic. We propose to push this approach farther, by transforming the card itself in a specialized processor, called Wireless MAC Processor (WMP) [5].

## 3.1 General WMP Architecture

As enlightened in the previous part, commercial 802.11 drivers and chipsets do not currently support the possibility of defining a customized protocol logic for managing the access to the shared medium and the sequence of frame transmissions. Indeed, making the protocol control logic programmable is not an easy task, because this logic cannot be programmed in the card host (due to the time contraints of the access operation) and cannot be supported by a simple configuration interface exposed by the card. In order to support this complex flexibility requirement, we envision a solution based on the introduction of wireless-processors exposing a standardized instruction set, and acting as a mediation layer between the vendor-specific hardware platform (i.e. the PHY layer and the processor implementation) and the upper-MAC functionalities. According to this vision, different medium access control protocols can be programmed through the standardized instruction set, and loaded and executed by the wireless-processor.

We define a MAC programs as programmable XFSM, rather than as programmable data flows. Our choice is motivated by the observation that data flows are usually more suitable for dynamic signal and data processing, while state machines are more effective for modeling the behavior of sequential control operations [19]. Moreover, the wireless-processor operations are much simpler than the MAC blocks used in [34], thus allowing a higher level of programmability. As an illustrative example, we consider the definition of the Distributed Coordination Function (DCF) [43], in the basic access case. Figures 3.1 shows (in normal style) the state machines of the two sub-systems in which the DCF behavior has been decomposed: a **transmitter sub-system** and a **receiver sub-system**. The state machine is depicted by specifying the state labels, the transition events, the guard conditions (in the square brackets), and the transition actions (in italic style). The figure shows that a MAC protocol is easy defined through a XFSM, and for this example we use events, conditions and action extract through a process of abstraction, in the next section we study in deep the API that can be implemented follow the logic of the hardware.

For sake of presentation completeness, we summarize the transmitter sub-system behavior. Starting from the IDLE state, as a new frame is enqueued in the hardware buffer (QUEUE_OUT_UP event) the transition to the state WAIT_MED or WAIT_DIFS_NO_BK is performed according to the state of the medium. If the medium is idle, a timer equal to the DIFS interval is set and a transmission is started (i.e. the trans-receiver is switched to the transmission mode and the system moves to the TX state) as the timer expires (END_TIMER event). If the medium is busy, the system remains in the state WAIT_MED until the event representing the end of the medium activity (CH_DOWN event) is registered. At this point, the system sets a timer equal to the DIFS interval and transits to the state WAIT_DIFS_BK. At the timer expiration, a backoff countdown has

to be performed. Therefore, the system runs a sequence of actions according to the verification of a condition on the residual backoff: if there is not a residual backoff time, a new backoff timer is set-up, otherwise the frozen backoff timer is resumed.

A transition to the BACKOFF state is then performed. If the backoff timer expires, the system can transit to the TX state, but if the medium is revealed as busy during the timer expiration (CH_UP event), the timer is frozen and the state comes back to the WAIT_MED state. From the TX state, the event signalling the end of the frame transmission (MED_DATA_CONF event) triggers the switch of the tran-receiver to the reception mode and the transition to the WAIT_ACK state. Finally, at the end of an acknowledgement reception (RCV_ACK event), the post-backoff is performed by transiting again to the WAIT_DIFS_BK state. A similar description can be detailed for the receiver sub-system. Note that both the state machines can be defined by using the same sets E, A, and C.

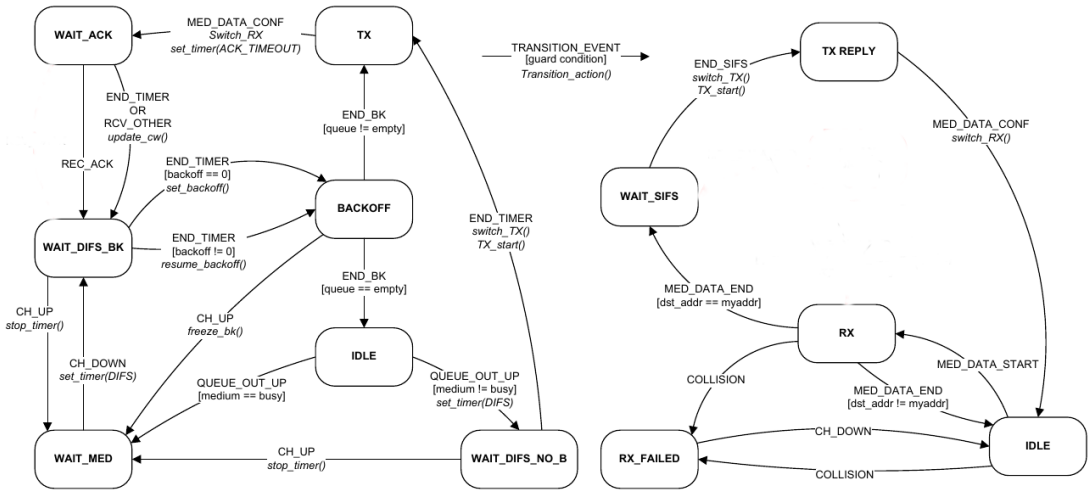



Figure 3.1: MAC Programs

The WMP is devised to specifically handle hardware/PHY events and schedule actions on the hardware/PHY card resources, thus leaving the MAC protocol developer with the much simpler task of describing when and under which events and/or conditions such actions should occur. In other words, similarly to other processors specialized for handling digital signals (DSPs) or graphical images (GPUs), we introduce a processor specialized for handling MAC operations.

The wireless MAC processor has been conceived as a CPU specialized for handling hardware/-PHY events and actions by executing Extended Finite State Machines (XFSMs). We chose to abstract the definition of the medium access control logic in terms of state machine because they are very effective in modelling the behavior of sequential control operations, and most MAC protocols are formally described in terms of state machines [5].

Figure 3.2 shows the internal architecture of the WMP, which includes five main components:

- **MAC-Engine** : an execution engine, running the provided XFSMs;
- **API** : a set of operations which can be invoked by the execution engine, which include logic, arithmetic and flow control operations plus specialized MAC operations;
- A **memory block** including both data and program memory space;
- An **interrupts block** passing the signals coming from the hardware to the execution engine;
- A **set of registers** for saving system state parameters.
- **Byte-Code-Manager** able to load and run the bytecode in the WMP platform.

Memory block, interrupts block and registers are physical elements able to save date and interact with the hardware, instead other element are main element and will be discuss in the next section of this chapter.

The core of the wireless-processor architecture is represented by an Extended Finite State Machine (XFSM) Execution Engine [19]. This block represents a micro-code executor, able to run a programmable finite state machine loaded in the instruction memory. Formally, a state machine is specified by a list of states and transitions. In a state machine, a transition is given by:

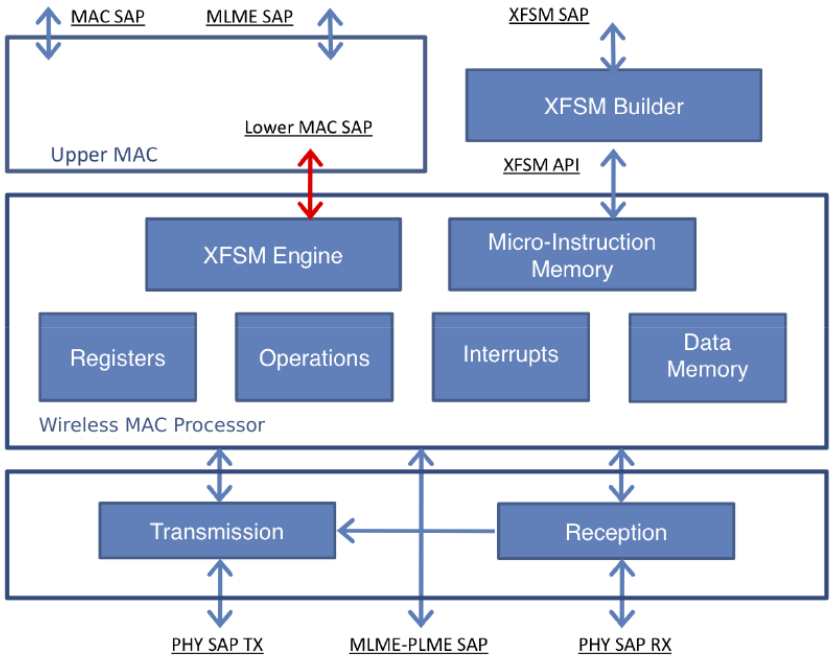



Figure 3.2: Internal architecture of the Wireless MAC processor

- a source state, from which the state machine starts its operations;
- a trigger event, such as a signal or a timer expiration;
- an optional guard condition, to be evaluated after the trigger event (which can be true or false);
- an action, representing an atomic program code, which can also create, move or destroy data objects;
- a target state, in which the state machine enters at the end of the action.

On the basis of: a set E of pre-defined **events** revealed and/or generated by the hardware platform, a set C of available **conditions** to be verified, and a set A of pre-defined elementary **actions**, a **state machine program** is given by a table, in which at location (i, j) the transition from state i to state j is specified in terms of the parameters ($event_x$ , $action_y$ , $cond_z$ ), with $event_x \in$ E, $action_y \in$ A, and $cond_z \in$ C. In our architecture, the set of events E is generated by the interrupt block, the set of actions A (and logic operators) is implemented in the pre-loaded operation block, and the set of state conditions C are memorized in the processor registers. The sets of events, actions and conditions supported by the processor are exposed in terms of API for the XFSM definition, while the MAC programs are loaded by the XFSM engine in the instruction memory in terms of transition tables.

If the XFSM engine supports multi-thread executions, i.e. it is able to manage potential conflicts of processes to be run in parallel, the protocol logic can be decomposed in multiple state machines that can independently react to the same events. This decomposition may drastically reduce the number of states to be included in the MAC program table.

The figure 3.2 also shows the interface towards the transmission/reception blocks and the PHY, and the interface towards the upper-MAC (external to the device, residing on the PC host). Figure 3.2 shows also a further module, XFSM Builder. This is an optional module, external to the WMP, and running on the host PC, which, analogous to a compiler, permits the user to write state machines in an higher level.

Further the register for the physical layer setting and the verification of the conditions, the WMP involves the use of two register and three memory location for save and comparison value that can be used in the definition of state machine, they are:.

- REGISTER 1 - REGISTER 2
- MEMORY 1 - MEMORY 2 - MEMORY 3

REGISTER 1 and 2 are two registers while MEMORY 1, 2 and 3 are three memory location, normally the access to read and write at registers is more fast to memory. The WMP provide a different condition and action that are used for set, reset, increase and comparison a specific register and memory, in this way it's possible create a different MAC state machine that collection statistical and use this for realize a logic behaviour. Through the Byte-Code-Manager is possible show a dump of the registers and memory.

## 3.2 The MAC-Engine

Our architecture, proposed within the EU Project FLAVIA [18], extends the soft-MAC approach by decoupling the protocol control into an hard-coded part implemented in the card firmware and a programmable part exposed to the host (a shown in fig. 3.3). In other words, we change the firmware role from implementing a given (unmodifiable) MAC protocol logic to implementing an executor of generic state machines, called MAC-Engine. Formally, a state machine is specified by a list of states and transitions. In a state machine the transition is given by a source state, a trigger event, an action, and a target state. On the basis of a predefined set of events and actions, representing the MAC programming interface, different state machines can be defined, loaded on the card shared memory and executed by the card. The MAC programming interface has been defined starting from the analysis of different use cases (e.g. a TDMA-like, an hybrid reservation/contention, and a multi-channel access protocol) which, in our opinion, cover most of the MAC programmability requirements emerged so far. We identified the list of events and actions supported by the card that allow representing the considered use cases in terms of FSMs.[8]

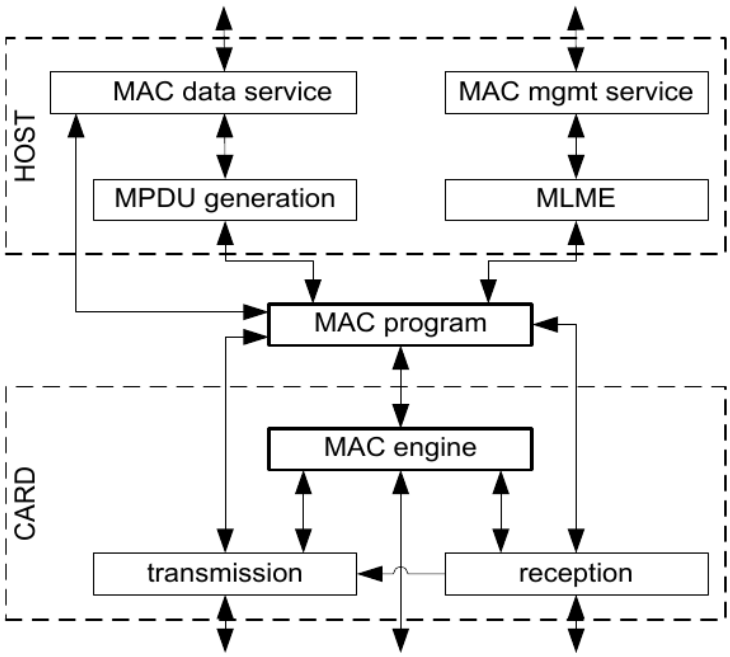



Figure 3.3: host/card functionalities

Every FSM is characterized by the following elements: states, transitions between couple of connected states, events, conditions and actions associated to each transition. While events and conditions drive state transitions, actions are performed by the MAC-Engine during a transition and eventually implement MAC specific functions. All these "elements" are specified by users into the Byte-Code.

**Events** are like interrupts and are internally represented by some flags flipping when the corresponding hardware section detect a change in the PHY: e.g. `TX_COMPLETE` reports to the MAC-Engine that a transmission that was started in the past has been completed. When in a given status, the WMP waits for the corresponding events, as specified in the Byte-Code, to occur. When one does, the transition can immediately take place, or one additional **Condition** can be evaluated, again as described in the Byte-Code. In the first case the transition is unique; in the second case, instead, there are two different transitions, triggered by the same event but having two different

arrival states, depending on the value taken by the condition. Finally, **Actions** are executed during a transition to a new state and they generally represent elementary activities like frame receving.

Starting from an initial (default) state, the MAC engine fetches the table entry corresponding to the state and loops until a triggering event associated with that state occurs. It then evaluates the associated conditions on the configuration registers, and, if this is the case, it triggers the associated action and register status updates (if any), executes the state transition, and fetches the new table entry for such destination state [11]. Since a MAC program is basically a list of labels specifying the events, actions, and conditions associated with each state transition, by defining a common set of labels for the API (i.e., a machine language), the MAC program can be transported over data frames from one node to another.

## 3.3 WMP API introduction

In order to define an interface covering most of the MAC programmability requirements emerged so far for WLAN systems, we analyzed several use cases [28] including a hybrid contention/polling, a TDMA-like, and a multi-channel access protocol, afterwards we also analyzed the general wireless card hardware architecture, to collect all the information on the register and signal interface with the physical layer. At the end of this process we identify a actual set of events, actions and conditions, which form a WMP programming interface able to support the analyzes use cases, a sub set is summarized in Table 3.1, and respect the actual hardware propriety of the wireless card. We introduce them in the reminder of this section, then we report all the API with more detail in the chapter 4.
We can find the events in the first column of the table 3.1. These are the set of signals either provided by the hardware interrupt block, or coming from the upper layers. The signals are generated by:

- the receiver sub system: `RX_PREAMBLE`, `RX_END` signals, i.e. start and end of reception;
- the transmitter sub-system: `TX_PREAMBLE`, `TX_COMPLETE` signal, i.e., end of a frame transmission;
- the transmission queues: `PACKET_IN_TX_QUEUE` signals, queuing of a new frame;
- the clock: `TIMER_n_TIMEOUT` signal when a pre-set timer expires.

The second column of the table 3.1 contain the Actions. In addition to arithmetic, logic and control flow primitives, the operation block supports MAC-specialized operations, categorized into configuration commands and hardware commands. The former work on the WMP registers storing the information about the configuration of PHY and MAC parameters, which refer to:

- the transceiver: `SET_CHANNEL`;
- the contention parameters: `ACTION_SET_VALUE` (cw);

The second group of operations drive different card sub-systems:

- the transceiver subsystem: `TX_DATA_FRAME`, `TX_CONTROL_FRAME`;
- the timers: `SET_TIMER_`$n$;

The last column contain the Conditions. The WMP contains registers explicitly updated by WMP actions and/or implicitly updated by WMP events, which store information on the card configuration and network state. These registers include: the station MAC address and the queue registers (queue length/type), the transceiver registers (channel and power), the contention registers (contention windows and backoff counter), the handshake registers, the frame registers (frame type, destination and source address, fragment).

## 3.4 Byte-Code

To permit the MAC engine to execute an XFMS, the latter must be coded in a suitable machine language, the name of machine language is **Binary-Byte-Code** and the name of the file that contain the Binary-Byte-Code is **Byte-Code**.

| *events* | *actions* | *conditions* |
|---|---|---|
| TX events | TX action | TX condition |
| PACKET_IN_TX_QUEUE | TX_DATA_FRAME | NEED_WAIT_ACK |
| TX_PREAMBLE | TX_CONTROL_FRAME | TX_PACKET_TYPE |
| TX_COMPLETE | TX_FRAME_FORGE | |
| RX events | RX action | RX condition |
| RX_PREAMBLE | RX_START | NEED_SEND_ACK |
| RX_END | RX_COMPLETE | RX_PACKET == MY_BEACON |
| | | RX_PACKET == ACK |
| | | RX_FRAME_FIELD_MATCH |
| Timer events | Timer action | |
| TIMER_n_TIMEOUT | SET_TIMER_$n$ | |
| | Parameters action | Parameters condition |
| | ACTION_SET_VALUE | BK_VAL != 0 |
| | SET_CHANNEL | TX_DST_ADDR == PARAM_TX_DST_ADDR_$n$ |
| | ACTION_INCREASE_VALUE | RX_SRC_ADDR == PARAM_RX_SRC_ADDR_$n$ |
| | | CUR_CHAN == PARAM_CHECK_CHANNEL |
| | | PARAM > CHECK_VALUE |

Table 3.1: WMP application programming interface: supported events, actions and conditions grouping them for functionality.

The Byte-Code is a simple text file that contain the Binary-Byte-Code or a table with states, transitions and program parameters. Programmers should design XFSM and customize their behavior using the graphical **WMP-Editor**: this tool really easy to use, we realize it to enables rapid prototyping of Byte-Codes with few clicks of the user, an example of DCF XFSM is showed in the figure 3.4. Nevertheless, Byte-Codes can also be generated at hand, so in the present section we explain the Byte-Code structure, how to write or modify it at hand, underlining the most meaningful details.

To inject the Byte-Code in the WMP we realize a specific tool, the name of this tool is **Byte-Code-Manager**, we explain the Byte-Code-Manager functionality in the section 5.3 in general you can use the Byte-Code-Manager to interact with the WMP system. Though users may run it directly from the command line, in server mode the tool waits for commands from the network.

Take into consideration that writing the Byte-Code without the WMP-Editor is a difficult and error-prone task because programmers need a deep knowledge of the MAC-Engine and the way Byte-Code is interpreted and they must strictly adhere to the Byte-Code structure: for this reason this chapter should not be considered as a full guide to Byte-Code hand-writing but as an additional source of knowledge to better understand how the MAC-Engine works.

### 3.4.1 Structure of the Byte-Code

A **Byte-Code** is a basic ASCII text file that defines a XFSM, for this reason it **contains a list of states and outgoing transitions**, including a set of events, actions and conditions associated to each transition. The Byte-Code is hence a "description" of the XFSM and it should be as compact as possible in order to meet the available memory limit. **Byte-Code contains also MAC parameters** as contention window, backoff parameters, channel, start state, etc., used to achieve a more compact protocol description in case of specific MAC designs such as CSMA-based ones.

Referring to the table 2.1, we define with $n_s$ be the number of symbolic states, and with $n_e$ the number of input events in $\Sigma$, the easiest approach is to code the XFSM as an $n_s \times n_e$ table. At each location (i, k), the table stores the state transition when event k is received at state i. Each transition has been defined by means of triplet $(e/c, a, s)$, where:

- $e/c \in F$ is event/condition enabling the transition;
- $a \in O + U$ is transition action;
- $s$ is the target state;

this technique allow to insert a single target state for each location (i, k), accordingly when multiple events/conditions are associated to a same transition, as a consequence of the above coding, the state with transition that enabling event and condition must be split into a sequence of intermediate states, each triggering at most one event and verifying at most one condition.

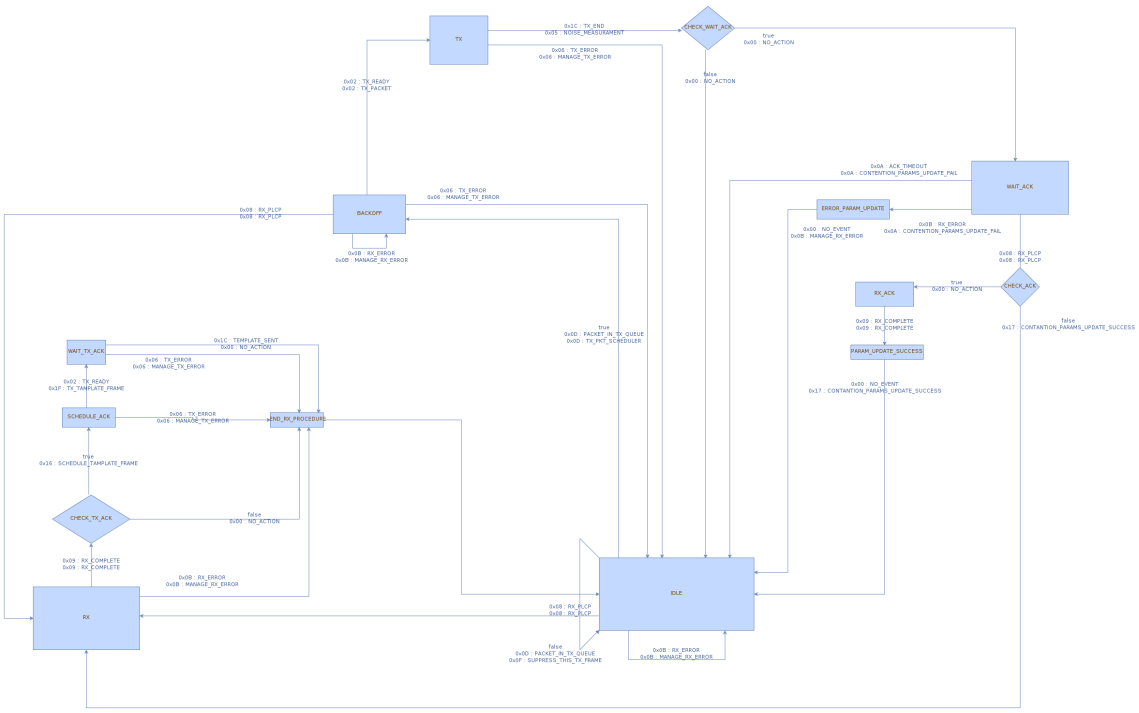



Figure 3.4: Example of DCF FSM graphical representation.

Accordingly the transition that have event and condition at the same time are split in two different state, one that incorporates the event and another that incorporates the condition, the second state is called virtual state, and is automatically performed from WMP-Editor.
In practice, to cope with the severe memory limitations of the broadcom chipset (only 2 KB are available for storing the MAC program table), we optimized the memory occupancy by replacing each table's row with a list containing only the non-null state transitions. As each state generally reacts to a number of input events much lower than the total inputs number (i.e., the table is sparse), skipping null-transitions significantly reduces the required memory space.

Definitely we have a **list of states** with the same dimension of 2-byte and a **list of consecutively variable number of transitions** with the same dimension 6-byte. Each state is easily addressable from the number of state, because it has a fixed size, while the transitions associated to a state (which can of different number) are addressable because the state contains the pointer of the first transition and the number of transitions associated with it. Summarize the Byte-Code contain the list of all states and the list of all the events that each state has enabled.

We use fixed 2-byte for state organized as follow

- 9-bit to point at first transition associated at the state;
- 3-bit to store the number of transition associated at the state;
- 4-bit inform if the state is a virtual state;

We use fixed 6-byte for transition organized as follow

- 28-bit for $e/e$ (condition/event), where the first 24-bit identifies the condition/event address routine, and the second 4-bit the condition/event parameter;
- 12-bit for $a$ (transition action), where the first 8-bit identifies the action label, and the remain 4-bit the action parameter (needed in case of configuration actions);
- 8-bit for $s$ target state number;

The table 3.2 shows the state and transition format.

The Byte-Code must contain only ASCII characters including numbers and letters ranging from "A" to "F" so that strings can be translated into equivalent hexadecimal byte sequences. Comments can be inserted after character "#" and may contain all ASCII characters.

For simplicity a Byte-Code is organized in two distinct regions, they are:

- state machine parameters list

| 4-bit : enable virtual state | 3-bit : num of transitions | 9-bit : pointer to first transition |
|---|---|---|

| 28-bit : address, label and parameter event/condition | 12-bit : label and parameter action | 8-bit : num target state |
|---|---|---|

Table 3.2: State and transition format

- states and transitions list

But the order of the region and the row is not a necessary requirement, because we use a text tag to identify the rows inside the Byte-Code, each row can be specify a parameter, or a state or a list of transition, and is preceded from a text tag. We consider two more tag to contain a Byte-Code, in way to have more Byte-Code in a unique file.

Tag `000001` is empty and must be the first in the byte code, used as start delimiter; tag `000099` is the stop delimiter.

The first part of byte code carrying information about the state machine is introduced by tag `000004` and contains the *state machine parameters*. The second part contains both *states* and *transitions*: each state is preceded by tag `000010` and the corresponding transitions by `000006`. Each transition block is terminated with special char `$`. In details:

- **000001** Delimits the beginning of the Byte-Code: what follows has to be injected into the WMP.
- **000003** It is a state parameter change position delimiter: informs the injector that next line is a value for change the position of the writer parameters. If the value of position not change the state parameters are sequentially written in a dedicated memory area. The injector starts from the position 0 and increment the position index for each state parameter. It is extremely important to have the state parameters in the same order the MAC-Engine expects them, but if you want write a single parameter or many at specific position start you can use this tag for fix the next write position.
- **000004** It is a state parameter delimiter: informs the injector that next line is a state parameter. State parameters are sequentially written in a dedicated memory area. The injector starts from the position 0 and increment the position index for each state parameter. It is extremely important to have the state parameters in the same order the MAC-Engine expects them.
- **000006** It is a state transition delimiter: informs the injector that next line is a list of state transitions. Transitions are written in a sequential order. The injector starts from the position 0 at the beginning of the state transition area and increment position every time a new state transition is added. Byte-Code programmer has to keep track about the position of first outgoing transition for each state. It allows the correct editing of each state.
- **000010** State delimiter: it informs the injector that the next line has to be interpreted as a state. Such information is sequentially written in the memory area dedicated to store states. The injector keeps track of the starting position of each state and increments it at any state addition. The order of states is important for a correct workflow.
- **000099** Delimits the end of the Byte-Code: what follows has not to be injected into the WMP.

Inside the Byte-Code data is aggregated in groups of 4 hex digits each. Since any hex digit has 4 bits, any group is 16 bit long. Values are represented in little endian, so the most significant byte is on the right of the least significant one: e.g., the following line

```
0E01010805082601010B010B3A01010D0200$
```

is split into groups of 4 hexadecimal digits:

```
0E01 - 0108 - 0508 - 2601 - 010B - 010B - 3A01 - 010D - 0200
```

inverted according to endianness:

```
010E - 0801 - 0805 - 0126 - 0B01 - 0B01 - 013A - 0D01 - 0002
```

and finally regrouped so that the output can be injected in the WMP and can be interpreted and executed by the MAC-Engine:

```
010E0801080501260B010B01013A0D010002
```

| | |
|---|---|
| **#Header** | |
| 000001 | |
| **#state machine parameters** | |
| 000004<br>0000<br>000004<br>FFFF<br>000004<br>0100<br>000004<br>FFFF<br>...[cut]... | |
| **# List of (state, transitions) items** | |
| 000010<br>0004<br>000006<br>0000FF0B000B0000FF08020800000F0D0E00$ | state tag<br>state IDLE #0x00<br>transition tag<br>transition data |
| 000010<br>0906<br>000006<br>0000F00205020000FF0B010B0000FF0802080000FF060006$ | state tag<br>state BACKOFF #0x01<br>transition tag<br>transition data |
| 000010<br>1502<br>000006<br>0000FF0904090000FF0B060B$ | state tag<br>state RX #0x02<br>transition tag<br>transition data |
| 000010<br>1B00<br>000006<br>0000FF090909$ | state tag<br>state RX_ACK #0x03<br>transition tag<br>transition data |
| 00010<br>1EF2<br>000006<br>0000FF0F0C160000FF000600$ | state tag<br>state CHECK_TX_ACK #0x04<br>transition tag<br>transition data |
| ...[cut]... | |
| 000010<br>5AF2<br>000006<br>0000FF0E010D0000FF00000F$<br>...[cut]... | state tag<br>Virtual State #0x0E<br>transition tag<br>transition data |
| **# Terminator**<br>000099 | |

Table 3.3: Byte code representation of the “State Machine” explained.

### 3.4.2 Byte-Code of the DCF State Machine

Table 3.3 contains and excerpt from the state machine of the DCF example. We dig into details in the following paragraphs.

**Byte-Code Header** First TAG is 000001 and indicates the beginning of the Byte-Code to inject.

**Byte-Code state parameters** The first “meaningful” section contains all the state machine parameters. Next TAG is 000004 and introduces the first state parameter that will be written in the first position of the dedicated memory area:

```
000004 #parameter @0x00
0000
```

where 000004 is the state parameter TAG, followed by a comment introduced by character “#”. Then we have the actual value of the first parameter, in this case PARAM_STATE_MACHINE_START,

that translates into 0000 according to endianness. Note that the order of parameters is important and has to be taken into consideration by the part of the Byte-Code defining states and transitions to correctly adderess the corresponding parameters, e.g., PARAM_STATE_MACHINE_START should be addressed as first state parameter at address zero. For the sake of clarity the second parameter in the example

```
000004 #parameter @0x01
FFFF
```

will be addressed as the second parameter at address 1. We remark also that state parameters are 16 bits long. Following lines contain other state parameters but we skip the actual description.

**Byte-Code states and transitions** The description of states and transitions follows that of the state parameters: here each state is followed by its outgoing transitions. To better understand this section we focus on the definition of the IDLE state which is composed of the following fields:

- 000010: state tag start
- 0004: translates into 0400 because of the used endianness that may be further decomposed as binary into

  ```
  0000 010 000000000
  ```

  where 0000 is set to 1111 if the state is a virtual state or condition, while is set to 0000 if is a normal state; 010 sets the number of outgoing transitions (three in this case, one is mandatory so 000 sets one transition) 000000000 is the position of the Transition area (zero in this case). It is computed in number of words (16bit per word) from the beginning of the memory region dedicated to transitions. The value can be obtained by considering the number of transitions written till the current one.
- 000006: transition tag start
- 0000FF0B000B0000FF08020800000F0D0E00$ this field represents three *Outgoing state transitions*, where each transition has a fixed length of 48bit (three 16bit words) and can in turn be further decomposed as follows (each subfield has already been converted according to endianness and splitted into components):

  1. Transition 0000 - 0B - F - F - 0B - 00

     0000 | address event/condition, this field is filled by Byte-Code-Manager when realize the injection
     0B | event/condition label: RX_ERROR
     F | no event/condition parameter
     F | no action parameter
     08 | action: MANAGE_RX_ERROR
     02 | next state: IDLE.

     This means that after executing action MANAGE_RX_ERROR, state machine evolves to state IDLE, indeed this is a self loop transition.
  2. Transition 0000 - 08 - F - F - 08 - 02

     0000 | address event/condition, this field is filled by Byte-Code-Manager when realize the injection
     08 | event/condition: RX_PREAMBLE
     F | no event/condition parameter
     F | no action parameter
     08 | action: RX_START
     02 | next state: RX.

     In this case, when event RX_PREAMBLE is raised, action RX_START is executed and state machine evolves to state RX.
  3. Transition 0000 - 0D - F - F - 00 - 0E

     0000 | address event/condition, this field is filled by Byte-Code-Manager when realize the injection
     0D | event/condition: PACKET_IN_TX_QUEUE
     F | no event/condition parameter

F | no action parameter
00 | no action
0E | next state: #0x0E - Virtual State.

The last transition is triggered by event PACKET_IN_TX_QUEUE: it is worth noting that in this case no action is specified and that when the event is raised, state machine evolves to a *Virtual State*, in this case #0x0E. See the example below.

The following state is BACKOFF, according to value F609 it is characterized by four transitions and, in fact, transition data 0000F00205020000FF0B010B0000FF0802080000FF060006$ can be split after endianness translation as

```
0000 - 02 - 0F - 02 - 05
0000 - 0B - FF - 0B - 01
0000 - 08 - FF - 08 - 02
0000 - 06 - FF - 06 - 00
```

**Byte-Code virtual states** Virtual states are not directly displayed by the GUI: they are, in act, generated during the translating process.

- 000010: state tag start
- 54F2: translates into F254 because of the endianness which can be further decomposed into

  1111 001 001010100

  where again 1111 inform that this is a virtual state, 001 means two transitionts, and 001010100 sets the first transition at address 0x54 of the transition memory area. Differently from other state with two transition, in virtual state the WMP-Editor filled the condition for the first transition only, the second transition is filled with the label 0x00, the meaning of this label is interpreted from MAC-Engine like a condition always true. This technical performed by MAC-Engine to work differently between state and virtual state, in the state the MAC-Engine keep in loop for waiting that an event is triggered, otherwise in virtual state MAC-Engine exit from state if the condition is not verified, but using the state associate with the condition false.
- 0000FF0E010D0000FF00000F$ represents the two outgoing state transitions, namely

  1. Transition 0000 - 0E - F - F - 0D - 01

     0000 | address event/condition, this field is filled by Byte-Code-Manager when realize the injection
     0E | condition: TX_PACKET == GOOD
     F | no event/condition parameter
     F | no action parameter
     0D | action: START_IFS_DATA_FRAME
     01 | next state: BACKOFF.

     This transition is chosen if the condition TX_PACKET == GOOD is true: to verify the condition the MAC-Engine runs procedure 0x0E, if returns true then action START_IFS_DATA_FRAME is executed and state machine evolves to state BACKOFF. If instead it returns false, then the MAC-Engine checks the next condition (here below).
  2. Transition 0000 - 00 - FF - 0F - 00

     0000 | address event/condition, this field is filled by Byte-Code-Manager when realize the injection
     00 | condition: 00, ALWAYS TRUE
     F | no event/condition parameter
     F | no action parameter
     0B | action: SUPPRESS_THIS_TX_FRAME
     00 | next state: IDLE.

     Condition index 00 is a “non condition” meaning that it is always verified. In this transition the condition is used to always execute action SUPPRESS_THIS_TX_FRAME and evolve state machine to state

     IDLE.

**Byte-Code terminator** Last tag 000099 concludes the Byte-Code.

## 3.5 WMP platform

In this section we describe the component element present in the WMP platform, figure 3.5 shows the platform system. The system consists in the platform architecture and a software to create, inject, and control the Byte-Code, the software running at the application level, and interacts with the WMP. This approach has the advantage because the selection of the MAC protocol can be based from high-level context.

The platform is based on four main components:

- **WMP-Editor**;
- **Byte-Code repository**;
- **Byte-Code-Manager**;
- **WMP Control interface**;

The Byte-Code repository or MAC program repository contains the Byte-Code available that the user has previously created with WMP-Editor, including either standard as well as customized (context-specific) MAC protocols. The Byte-Code-Manager is responsible of enabling and loading the Byte-Code on the WMP, and programming MAC reconfigurations and switching.

The WMP control interface is the interface to the hard-coded device, through which new MAC state machines and switching conditions are loaded on the WMP, as summarized in table 3.4.

The WMP Control interface receive the command from Byte-Code-Manager and performs the different operation. Such as showed in figure 3.5, two independent Binary-Byte-Codes can be stored inside the WMP, each limited to 992 Byte: since switching time is negligible, this design choice enables immediate reconfiguration of the MAC algorithm so that two different behavior can be selected either automatically given user request or periodically. In the latter case switching time can be synchronized between stations. The **Byte-Code-Manager** can be used either to inject Byte-Code to the WMP, to set parameters for tailoring the MAC behavior, or to monitor the WMP, so it can be used for

- Loading one Byte-Code in one of the two available slots;
- Activating one Binary-Byte-Code;
- Setting up timers for activation/shutdown of Binary-Byte-Code s;
- Showing conditions for activation/shutdown by means of timers;
- Write a control frame in tamplate ram to send with specific WMP API;
- Other options, set or reset values information for the WMP register and memory

| Command | WMP control interface |
|---|---|
| $load(i)$ | load a MAC program on memory slot i |
| $run(i,t)$ | activate MAC program on slot i (asynch. or at the timer time out) |
| $verify(i)$ | recognize trusted code by means of check current active Binary-Byte-Code |

Table 3.4: WMP Commands invoked

Regarding the wireless card firmware, we developed a micro-code with the MAC-Engine and procedures corresponding to the WMP API, also, we add same primitive to implement the WMP control interface (eg. switch primitives - the verify command has not been currently implemented) and added registers indicating the state of the program slots and the program under execution.

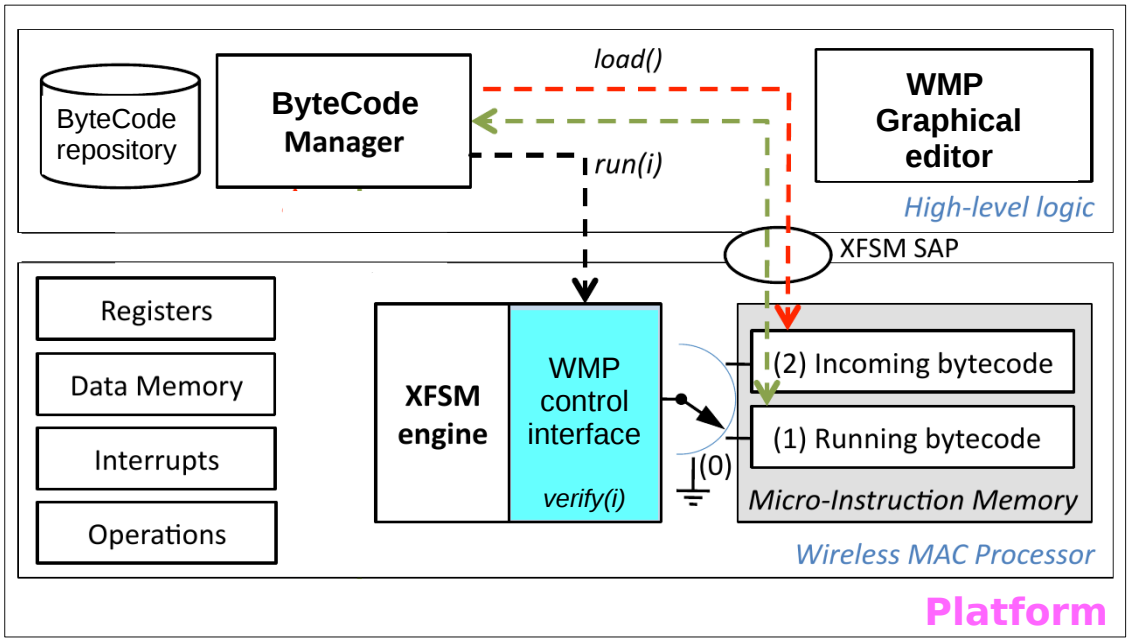



Figure 3.5: WMP platform

## 3.6 Binary-Byte-Code

Two Binary-Byte-Codes are stored in the shared memory of the wireless card in range [0x0830-0x0FFF], the total size of the shared memory is 4KB, we use half part about to save the Binary-Byte-Codes (2000B), the size of first Binary-Byte-Code is 1008B, the remaining part is used for the second Binary-Byte-Code. For a deeper analysis we will refer to the second Binary-Byte-Code because it displays more easily, it is saved in the range [0x0C20-0x0FFF] memory. For the first Binary-Byte-Code worths the same notions. We recall here that the reference architecture is little-endian: memory dump will show odd and even address bytes swapped for the ease of comprehension. The Byte-Code of the previous DCF example is been loaded in the WMP at second slot, and the equivalent Binary-Byte-Code is showed through a dump of the last shared memory area, the area start at location 0x0C20 and end at 0x0FFF, this area contain the Binary-Byte-Code injected from Byte-Code-Manager. In the following we will use the figure 3.6 to analyse the Binary-Byte-Code of the previous DCF MAC.

Each Binary-Byte-Code is organized in three distinct memory regions, which we contoured with dashed line in Figure 3.6, they are:

- States region
- Transitions region
- State parameters region

### 3.6.1 States region

This region extends in range [0xF90-0x0FFF], it counts 112 bytes. Each state is a 16bit value: up to 56 different states can be store, they are increasingly numbered beginning with 0. Each state encode the following fields:

- mask 0xF000: is set to 0xF if the state is a virtual state or condition, while is set to 0x0 if is a normal state, this part of state description is used from MAC-Engine for save the number of last transition that was triggered the last time that MAC-Engine passed from this state, this allows at MAC-Engine of remember the last event triggered; Since the events in WMP are not verified through interrupt but instead are seen through the verification of precise registers in a completely asynchronous, comes the need to save the information on the last transition

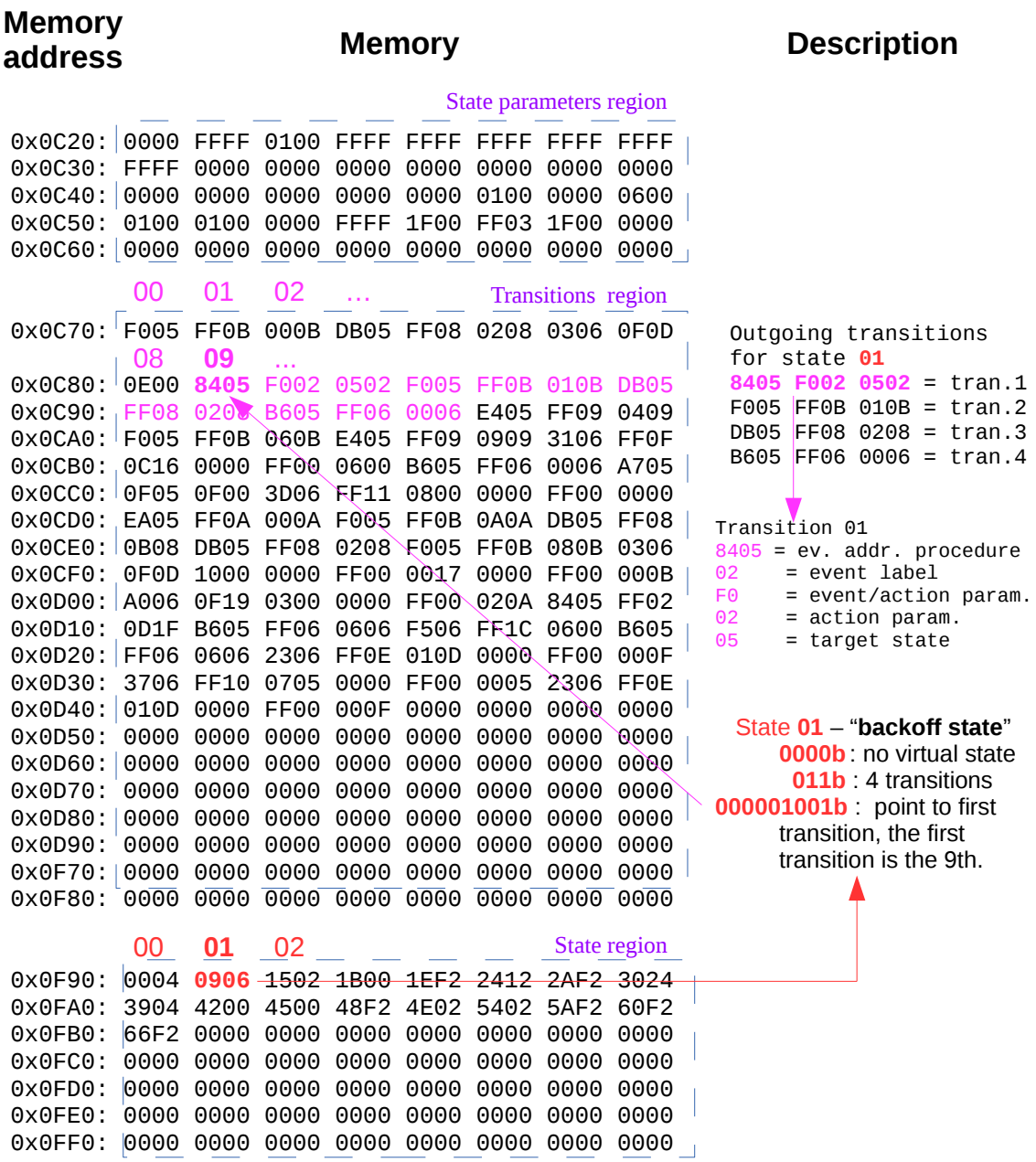



Figure 3.6: Binary-Byte-Code implementation, as stored in the shared memory

before leave the state, such that the next time that MAC-Engine return in the state, it start the event test from the transition occurred last, so as to eliminate any priority to transitions that are at the head of the list. The order in which they were queued by WMP-Editor and completely at random.

- mask 0x0E00: number of transitions from this state decreased by 1, for values between 0 (means 1 transition) and 6 (means 7 transitions). Value 7 means that the number of transitions is great of 6, in this case the WMP-Editor insert a delimiter tag (0xFFFF) at the end of the state transitions list(details follow), when this field contain the number 7 MAC-Engine learn the number of transition for the current state from delimiter.

- mask 0x01FF: offset to the transition map from this state, referred to the Transitions region. This map extends for a number of transitions encoded in the previous field, or up to a marker if the previous field is set to 7. Nine bits would allow up to 512 transitions but the memory does not permit such plenty of transitions.

Considering Figure 3.6 the value associated to state 01, after endianness correction, becomes:
0x0609 b(0000) (101) (000001001)
and encode a state with four transition, the first at offset 9 of the Transitions region.

### 3.6.2 Transitions region

Memory dedicated to transitions from the states extends over 816 bytes in range [0x0C70, 0x0F8E]. Each transition is encoded with 48 bit and holds the event or the condition, the action to be executed and the arrival state. Memory extension allow to represent up to 133 transitions (800 bytes / 6byte/tran.). All transitions from a given state are contiguous and starts at address specified in the state definition; each state finally might have a different number of transitions. A transition which considers both event and condition, the latter towards a couple of arrival state, is managed with three different transition. The first is triggered by the event and leads to an intermediate state. The second and the third are function of the condition and they both lead to the final state.

Each transition encodes the following fields:

- mask FF FF 00 00 00 00: address of the procedure that checks for events and conditions. In the first case, code loops in the initial state until one event is verified, then execute the transition. In the second case, instead, code immediately evaluates the condition and jumps to one of the two final states. It is worth noting that the code involved in these checks is dynamically generated by Byte-Code-Manager according to the second field (below): this improves the execution speed of code involved in events and conditions checking. Note also that in table 3.3 this field is empty because code will be generated only at injection time.
- mask 00 00 F0 00 00 00: condition/event parameter, see the following one.
- mask 00 00 0F 00 00 00: action parameter. This field and the previous one allows to pass some data chosen by users to the procedure that implements the condition/event checking. These parameters increase the flexibility of events, actions and conditions.
- mask 00 00 00 FF 00 00: index of the condition/event. It is filled during the definition of the FSM with the unique id of the condition/event. It is used by the injection code to get the address of the procedure to set in the field mask FF FF 00 00 00 00, in this way the MAC-Engine has the address of event and condition directly, this improves the execution speed of code involved in events and conditions checking.
- mask 00 00 00 00 FF 00: index of the arrival state. Used by the WMP to select the final state after having checked the condition/event and executed the action.
- mask 00 00 00 00 00 FF: action index. This is filled during the FSM definition with the id of the action and used by the WMP to get the address of the procedure to execute after the event or condition associated to the transition has been verified.

The four transitions from state 2, considered in the example of the figure 3.6 are stored starting at address 9, are underlined in Figure 3.6 and they extend in $6 * 4 = 24$ bytes.

- transition 0: 84 05 F0 02 05 02
- transition 1: F0 05 FF 0B 01 0B
- transition 2: DB 05 FF 08 02 08
- transition 3: B6 05 FF 06 00 06

First transition has the following meaning

1. 0584, event address
2. 0, parameter for condition/event, F means no parameter
3. F, parameter for action, F means no parameter
4. 02, identify event `TX_PREAMBLE`
5. 05, arrival state for the transition if event verify
6. 02, identify action `TX_DATA_FRAME`

### 3.6.3 State parameters region

This region extends on 64 bytes in the range [0x0C20-0x0C60] and contains two kind of parameters, those associated to the entire Byte-Code and those used in events, conditions and actions. Those in the first set are used during the Byte-Code bootstrap and after a Byte-Code switch, in the reconfiguration phase. All parameters can be set using an interface of the WMP-Editor, we remand to section 4.1 for the list with all the state parameters and relative detail.

Finally, figure 3.6 shows an actual example of Binary-Byte-Code for the legacy DCF, where we can recognize the state parameters region (for configuring the platform registers) and the transition region. As evident from the figure, the code to define a DCF MAC program is very compact (only $\sim 336bytes$).

# Chapter 4

# API details: events, conditions and actions

In order to define an interface covering most of the MAC programmability requirements emerged so far for WLAN systems, we analyzed several use cases [21] including a hybrid contention/polling, a TDMA-like, and a multi-channel access protocol. The set of identified events, actions and conditions, which form a WMP programming interface able to support the analyzes use cases is presented in the follow section in which we report the detail for all the API [5].

This chapter goes into details about fundamental elements of the WMP. As said in the previous chapter, a FSM is built on states, transitions, events, conditions and actions: transitions, that link states one way and are triggered by events or ruled by conditions, eventually start the execution of actions.

For each state the MAC-Engine checks only the events associated to the transitions that exit from that state and waits until one triggers. If a transition is associated only to an event, then it is unique to the given destination state. If instead it is also associated to a condition, then there exists another transition triggered by the same event and associated to the condition negated.

First the chapter report an explanation of all state parameters that the user can be setted when realize the MAC protocol, and after a detail of the events, conditions and actions are presented grouping them on the basis of the function to which they are connected. The table 4.1 summarize all the WMP API.

| *events* | *actions* | *conditions* |
|---|---|---|
| TX events | TX action | TX condition |
| PACKET_IN_TX_QUEUE | START_IFS_DATA_FRAME | TX_PACKET == GOOD |
| TX_PREAMBLE | TX_DATA_FRAME | NEED_WAIT_ACK |
| TX_COMPLETE | MANAGE_TX_ERROR | TX_PACKET_TYPE |
| TX_10us_ELAPSED | REPORT_TX_STATUS_TO_HOST | |
| TX_ERROR | SUPPRESS_THIS_TX_FRAME | |
| | START_IFS_CONTROL_FRAME | |
| | TX_CONTROL_FRAME | |
| | TX_FRAME_FORGE | |
| RX events | RX action | RX condition |
| RX_PREAMBLE | RX_START | NEED_SEND_ACK |
| RX_END | RX_COMPLETE | RX_PACKET == MY_BEACON |
| RX_ERROR | MANAGE_RX_ERROR | RX_PACKET == ACK |
| | | RX_FRAME_FIELD_MATCH |
| Timer events | Timer action | Timer condition |
| BEACON_TIMER_TIMEOUT | SET_TIMER_$n$ | TIMER_$n$ == ON |
| ACK_TIMEOUT | RESET_TIMER_$n$ | TX_SLOTTED |
| TIMER_n_TIMEOUT | RESET_ACK_TIMEOUT | |
| TX_SLOTTED | RESET_TX_SLOTTED | |
| | Parameters action | Parameters condition |
| | NOISE_MEASUREMENT | BK_VAL != 0 |
| | SET_CHANNEL | TX_DST_ADDR == PARAM_TX_DST_ADDR_$n$ |
| | RESET_CHANNEL | RX_SRC_ADDR == PARAM_RX_SRC_ADDR_$n$ |
| | SET_TX_MAC_ADDRESS | CUR_CHAN == PARAM_CHECK_CHANNEL |
| | SET_RX_MAC_ADDRESS | PARAM > CHECK_VALUE |
| | INFLATION_CW | |
| | DEFLATION_CW | |
| | ACTION_INCREASE_VALUE | |
| | ACTION_DECREASE_VALUE | |
| | ACTION_SET_VALUE | |
| | ACTION_RESET_VALUE | |
| | SET_RX_ANTENNA | |
| | SET_TX_ANTENNA | |

Table 4.1: WMP application programming interface: supported events, actions and conditions grouping them for functionality.

## 4.1 State parameters

In this section we report a short description of all state parameters that the user can be setted to define the MAC protocol through the WMP-Editor, indeed the WMP-Editor has a sub windows in which is possible fill a series of fields, this fields are used when the graphical representation will translate in the Byte-Code to obtain the state parameters. All the state parameters are grouping in four type, the first are the `BOOTSTRAP PARAMS` parameters, these are used by WMP when active a Binary-Byte-Code, `BOOTSTRAP PARAMS` parameters are loaded in the main register of WMP to a correct start-up of MAC protocol. Then we have the `ENHANCED PARAMS`, these collect the main parameter used by actions and conditions. The lasts two group of parameters are: `BACKOFF PARAMS` and `MAC FLOW`, they are perform the tuning of the contention windows and the forging/analysis of the frame field. A pictures of the WMP-Editor sub windows to modify the state parameters is reported in figure 5.3(a).

`BOOTSTRAP PARAMS`

- `PARAM_STATE_MACHINE_START` This is a bootstrap parameter, it defines the initial state of the FSM after Binary-Byte-Code starts. Switches between Byte-Codes can happen only if the FSM of the current Byte-Code is in its initial state. This requisite guarantees that when a Byte-Code is deactivated, there are no pending operations in the WMP.
- `PARAM_CHANNEL_MACLET` This is a bootstrap parameter, defines the initial channel of the radio for the designed Byte-Code.
- `PARAM_CW_MIN` This is a bootstrap parameter, defines the initial value of MIN CONTENTION WINDOWS for the designed Byte-Code.
- `PARAM_CW_MAX` This is a bootstrap parameter, defines the initial value of MAX CONTENTION WINDOWS for the designed Byte-Code.
- `PARAM_CW_CUR` This is a bootstrap parameter, defines the initial value of CUR CONTENTION WINDOWS for the designed Byte-Code.
- `PARAM_TIME_SLOT_POSITION` This is a bootstrap parameter, defines the value of position time slot when a tdm transmission is activated for the designed Byte-Code, with this value is possible specified the position in the frame of the station.

`ENHANCED PARAMS`

- `PARAM_BACKOFF` This parameter defines the the rule to access to the channel used by action `START_IFS_DATA_FRAME` in the phase that prepare the transmission, such as specified in the follow
  - NO_IFS to transmit immediately;
  - BK_SLOT=02 – BK_SLOT=24 to set a specific backoff value between 2 and 24;
  - SIFS to transmit after a SIFS value;
  - PIFS to transmit after a PIFS value;
  - STD to keep the legacy behavior with respect to the implemented;
- `PARAM_SET_CHANNEL` This parameter defines the channel used to tune the radio when action `SET_CHANNEL` is executed.
- `PARAM_TX_DST_ADDR_`$n$ This parameter specifies one 48bit MAC address that is compared in condition `TX_DST_ADDR == PARAM_TX_DST_ADDR_`$n$ with the destination address of the frame that will be transmitted, $n \in [1, 2, 3]$.
- `PARAM_RX_SRC_ADDR_`$n$ This parameter specifies one 48bit MAC address that is compared in condition `RX_SRC_ADDR == PARAM_RX_SRC_ADDR_`$n$ with the source address of the frame that is being received.
- `PARAM_GPT_`$n$`_`$m$`_CNTHI`, `PARAM_GPT_`$n$`_`$m$`_CNTHI` These two 16bit parameters line up in their corresponding 32bit timestamp parameter `PARAM_TIMER_[`$n$`, `$m$`]` to setup the initial value of timer GPT$n$ by action `SET_TIMER_`$n$, $n \in [0, 1]$, $m \in [0, 1]$. Given that $m$ can assume two different values, two different timestamps can be built for each timer (GTP0 and GPT1).

- PARAM_CHECK_CHANNEL This parameter is used in condition CUR_CHAN == PARAM_CHECK_CHANNEL to verify that the current radio channel is that in the parameter.
- PARAM_TIME_SLOT This parameter is used in condition TX_SLOTTED to compute the time slot for next transmission. It is fundamental to implementation of TDM based MACs.
- PARAM_SET_VALUE This parameter is used in the action ACTION_SET_VALUE to set the value specified in the field in the REGISTER or MEMORY.
- PARAM_CHECK_VALUE This parameter is used in condition PARAM > CHECK_VALUE to compute the value to the REGISTER or MEMORY, the condition exit with result true if the value is great to field PARAM_CHECK_VALUE.

BACKOFF PARAMS

- PARAM_INFLATION_MUL This is a backoff parameter, defines how to increase the value of the current contention window.
- PARAM_INFLATION_ADD This is a backoff parameter, defines how to increase the value of the current contention window.

Parameters PARAM_INFLATION_MUL and PARAM_INFLATION_ADD defines the function that is used to update the current contention window, that is

cwcur = cwcur * PARAM_INFLATION_MUL + PARAM_INFLATION_ADD

- PARAM_DEFLATION_DIV This is a backoff parameter, defines how to decrease the value of the current contention window.
- PARAM_DEFLATION_SUB This is a backoff parameter, defines how to decrease the value of the current contention window.

Parameters PARAM_DEFLATION_DIV and PARAM_DEFLATION_SUB defines the function that is used to update the current contention window, that is

cwcur = cwcur / PARAM_DEFLATION_DIV - PARAM_DEFLATION_SUB

MAC FLOW The WMP API contain conditions and actions that allow you to analyze or modify specific fields of packets that are transmitted or received, the fields offset and the value must be setted through the MAC FLOW parameters. This API used a fixed size for the part of the packet of 2 byte.

- RX_FLOW_CHECK_OFFSET This is a mac flow parameter, defines offset used by the condition RX_FRAME_FIELD_MATCH to check a part of receive packet.
- RX_FLOW_CHECK_VALUE This is a mac flow parameter, defines value used by the condition RX_FRAME_FIELD_MATCH to check a part of receive packet.
- TX_FLOW_CHANGE_OFFSET This is a mac flow parameter, defines offset used by the action TX_FRAME_FORGE to modify a part of transmission packet.
- TX_FLOW_CHANGE_VALUE This is a mac flow parameter, defines value used by the action TX_FRAME_FORGE to modify a part of transmission packet.

## 4.2 Events

Events are signal generated by the underlying hardware, events are like interrupts and are internally represented by some flags flipping when the corresponding hardware section detect a change in the Physical layer (PHY). In the following we report a list of events that can be used to define a state machine and a detailed description for each of them.

### 4.2.1 Events to handle the TX

- PACKET_IN_TX_QUEUE This event triggers when the upper layers enqueued a packet in the device queue: it signals to the WMP that the packet is ready for transmission, that will be handled by action START_IFS_DATA_FRAME.
- TX_PREAMBLE This event is triggered when the transmitter begins transmitting of the preamble of a packet so that the transmission of the frame can be handled by the MAC. The basic logic that leads to the transmission of a frame in the current WMP is composed of the following four steps: first, it is checked that a frame is available in the queue, if it is then the hardware is set up for transmitting the frame; a second step chooses when to transmit the frame, e.g., after a precise inter frame space (SIFS, PIFS, DIFS) or after a backoff stage, so that the transmission is scheduled and it will start independently of the WMP. After waiting the scheduled delay (third step), the WMP finalizes hardware setup for the undergoing transmission. It's more important that the event TX_PREAMBLE is triggered when the physical transmitter begins transmitting. The WMP implementation perform precise inter frame space opportunely selected when the user define the XFSM, the inter frame space are in according with 80211-2007 standard [20].
- TX_COMPLETE This event is triggered when the transmitter end, after the end of transmission is possible transition in a another state.
- TX_10us_ELAPSED This event is triggered 10us after the end of the current transmission. The WMP architecture, in fact, requires to measure the channel noise after each transmission but this can not be done immediately after otherwise the measure could be affected by the transmission itself. When the event triggers, action NOISE_MEASUREMENT is executed.
- TX_ERROR This event triggers when an error is detected during transmission and it should be checked by all states involved in transmission activity. If event triggers, action MANAGE_TX_ERROR must be executed to reset the transmitter.

### 4.2.2 Events to handle the RX

- RX_PREAMBLE This event triggers when the PLCP of a frame is received. The WMP checks this event to begin handling reception before it terminates, e.g., to schedule the transmission of the acknowledgement if the packet that is being received requires it. If this event triggers, action RX_START must be executed.
- RX_END This event is triggered at the end of the current reception, in this case action RX_COMPLETE must be executed.
- RX_ERROR This event triggers when an error is detected during reception and it should be checked by all states involved in reception activity. If event triggers, action MANAGE_RX_ERROR must be executed to reset the receiver.

### 4.2.3 Events to handle the timer

- BEACON_TIMER_TIMEOUT If the AP mode work, this event is triggered when the beacon timer is expired so that all relevant operations can be handled by the MAC. Usually after this event is possible schedule a beacon, afterwards the beacon is sent such as other frame.
- ACK_TIMEOUT This event triggers when the ack timeout timer expires. The value of the ack timeout is set up during the first stage of transmission if required (e.g., unicast data frame) and accordingly to the code and rate of the packet that is being transmitted. The acknowledgment, in fact, will be transmitted back at the same rate and this allow to correctly choose the ack timeout, that can not be fixed. The timer automatically started at the end of the current transmission and its expiration means that no ack has been received, otherwise the timer should have been stopped. This event is handled by action UPDATE_CW which, according to the number of transmission attempts will either set up a retry or start the operations to remove the current frame from the queue.
- TIMER_0_TIMEOUT, TIMER_1_TIMEOUT There are two timers that could be used by the user for MAC customization, GPT0 and GPT1. They are activated by a specific action and their timeout is signalled by these pair of events.

- TX_SLOTTED If yuo want a TDM (time division multiple access)trasmission, this event triggers when the exact time for the trasmission is reached evaluates: the time period must be set up in the state parameter PARAM_TIME_SLOT corresponding to this condition and the timer used by the WMP for evaluating this condition is the internal clock reference; this means that for MACs that maintain synchronization among stations the occurrence of the condition is distributed.

## 4.3 Conditions

Conditions are evaluated by checking values reported by hardware registers or by internal software register after they are changed and can be also function of multiple (mixed) values. Conditions can be evaluated by the WMP after an event is detected to choose which transition associated to that event should be followed and which action executed, leading to a given final state. This is also the main difference between an event and a condition: the former is considered only when it is raised by the hardware, the latter is always considered to decide which transition should be followed. In the following we report a list of the conditions that can be used to define a state machine and a detailed description of each of them.

### 4.3.1 Conditions to handle the TX

- TX_PACKET == GOOD This condition is evaluated before of a transmission and set up according to a number of checks that determine if the next transmission frame is in a correctly format. This condition is used after event PACKET_IN_TX_QUEUE triggers.
- NEED_WAIT_ACK This condition evaluates true if the transmitted frame requires an acknowledgment: in this case the WMP must drive the state machine in a "waiting ack" state, which comprises a waiting stage followed by ack reception.
- TX_PACKET_TYPE This condition evaluates true if the TX packet is a data frame. You can also use the optins menu for more features.

### 4.3.2 Conditions to handle the RX

- NEED_SEND_ACK This condition evaluates true if the received frame must be acknowledged: in this case the WMP drives the state machine in a state that will transmit the ack.
- RX_PACKET == MY_BEACON This condition evaluates true if the last received frame was a beacon and was transmitted by the associated AP, you can also specify through the optins menu if the checked beacon must bean a value of DTIM 0 or 1 (DTIM==0 | DTIM==1).
- RX_PACKET == ACK This condition evaluates true if the last received frame was an acknowledgment. This condition is typically used during ack waiting to detect if something different has been received in place of the expected ack or if the transmitted frame has been correctly acknowledged. You can also specify through the optins menu if the checked RX packet is a my beacon or a any beacon ( myACK | anyACK ).
- RX_FRAME_FIELD_MATCH This condition evaluates true if the word inside the receive packet matching with the mac flow value, the part of the packet is index with the offset specified in the state parameter MAC FLOW whose name is RX_FLOW_CHECK_OFFSET, and the value to matching is specified in the state parameter MAC FLOW whose name is RX_FLOW_CHECK_VALUE.

### 4.3.3 Conditions to handle the parameters

- BK_VAL != 0 This condition evaluates true if the backoff counter is not null. This condition is used if during the backoff a packet has been received and the backoff was freezed: in this situation the condition is used at the end of the current reception to understand that is need returned in the backoff state to continue the backoff countdown. In the other case, if the backoff value is zero the WMP must return in the idle state.
- TX_DST_ADDR == PARAM_TX_DST_ADDR_$n$ This condition evaluates true if the destination MAC address of the packet being transmitted corresponds to the one specified by the nth-state parameter of the condition PARAM_TX_DST_ADDR_$n$, $n \in [1, 2, 3]$.

- RX_SRC_ADDR == PARAM_RX_SRC_ADDR_$n$ This condition evaluates true if the source MAC address of the packet being received corresponds to the one specified by the nth-state parameter of the condition PARAM_RX_SRC_ADDR_$n$, $n \in [1, 2, 3]$.
- CUR_CHAN == PARAM_CHECK_CHANNEL This condition evaluates true if the current channel is equal to that specified in the state parameter of the condition whose name is PARAM_CHECK_CHANNEL.
- PARAM > CHECK_VALUE This condition evaluates true if the value of register or memory, specify through the option, is great of the parameter write in the appropriate area.

### 4.3.4 Conditions to handle the timer

- TIMER_$n$ == ON This condition evaluates true if timer GPT$n$ is running, $n \in [0, 1]$.
- TX_SLOTTED When used, this condition evaluates true periodically: the time period must be set up in the state parameter PARAM_TIME_SLOT corresponding to this condition and the timer used by the WMP for evaluating this condition is the internal clock reference; this means that for MACs that maintain synchronization among stations the occurrence of the condition is distributed.

## 4.4 Actions

Actions are elementary operations that can be executed during a state transition to implement complex MAC. In the following we report a list of the actions that can be used to define a state machine and a detailed description of each of them.

### 4.4.1 Actions to handle the TX

- START_IFS_DATA_FRAME When executed, it takes care of all operations involved in the second step of the transmission loop, that is after having checked that the current packet in the queue can be transmitted, it schedules the actual transmission by choosing the time it will start with respect to some event in the past (e.g., after a SIFS after the end of the reception of the previous frame). Backoff for transmissions that require it is chosen at this step. This action uses the enhanced parameter PARAM_BACKOFF for set the transmission delay and the value of the backoff as specified in the label, on more
  - NO_IFS to transmit immediately;
  - BK_SLOT=02 – BK_SLOT=24 to set a specific backoff value between 2 and 24;
  - SIFS to transmit after a SIFS value;
  - PIFS to transmit after a PIFS value;
  - STD to keep the legacy behavior with respect to the implemented;
- TX_DATA_FRAME When executed, it finalizes the transmission of the frame that has been previously checked in the queue, analyzed and scheduled for transmission after a selected delay, operations triggered by event TX_PREAMBLE. Select the appropriate parameter for realize specifc function.

| Value | Function |
|---|---|
| 0x0 | tx_packet_and_set_RX_ack |
| 0x1 | tx_packet_and_stop |

- MANAGE_TX_ERROR When executed, it handles errors that have been detected during transmission and reset the transmission module. It is triggered by event TX_ERROR.
- REPORT_TX_STATUS_TO_HOST When executed, it reports to upper layers information about transmission status (e.g., number of attempts, delivery, failure, etc.).
- SUPPRESS_THIS_TX_FRAME When executed, it discards the current packet in the transmission queue, e.g., this is executed to remove an outstanding packet because too old. Condition TX_PACKET == GOOD controls this action.

- START_IFS_CONTROL_FRAME When executed, it schedule the trasmission of a ACK or TEMPLATE-FRAME as specified in the parameter below the action, the possible parameter are:

| Parameters |
|---|
| SCHEDULE_FRAME |
| SCHEDULE_ACK |
| SCHEDULE_BEACON |

  In this stage the WMP can prepare the acknowledgment if the incoming packet requires it, the action setting the rate and and receive address according with the received packet.

- TX_CONTROL_FRAME When executed, it realize the transmission of a ACK, TEMPLATE-FRAME or BEACON as specified in the parameter below the action, the possible parameter are:

| Parameters |
|---|
| TX_FRAME |
| TX_ACK |
| TX_BEACON |

  if a transmission ACK is scheduling, it will transmit a SIFS after the reception will be concluded.

- TX_FRAME_FORGE When executed, the action modifies a part of 16 bits of the frame, the part of the packet is index with the offset specified in the state parameter MAC FLOW whose name is TX_FLOW_CHANGE_OFFSET, and the value to modify is specified in the state parameter MAC FLOW whose name is TX_FLOW_CHANGE_VALUE.

### 4.4.2 Actions to handle the RX

- RX_START The receiver loop is divided into two stages. The first stage starts when the PLCP of an incoming packet has been decoded correctly and it is triggered by event RX_PREAMBLE. The WMP configures the hardware for finalizing the frame reception (or if needed to stop it, discarding the bytes that have been already received).

- RX_COMPLETE It must be executed after event RX_END: the WMP concludes the reception of the frame (and the receiver loop as well).

- MANAGE_RX_ERROR When executed, it handles errors that have been detected during frame reception and that triggered event RX_ERROR.

### 4.4.3 Actions to handle the timer

- SET_TIMER_$n$ **(**PARAM_TIMER_[$n$, $m$]**)** When executed, it activates timer GPT$n$, $n \in [0, 1]$ using one of the two associated state parameters $m \in [0, 1]$ as specified in the parameter menu of the action.

- RESET_TIMER_$n$ **(**PARAM_TIMER_[$n$, $m$]**)** When executed, it stop and deactivate timer GPT$n$, $n \in [0, 1]$ using one of the two associated state parameters $m \in [0, 1]$ as specified in the parameter menu of the action.

- RESET_ACK_TIMEOUT When executed, the action reset the ack timeout condition.

- RESET_TX_SLOTTED When executed, it reset the pre-set time trasmission, if a TDM trasmission was enabled, re-set as next time trasmission a elapsed time of 16 micro seconds.

### 4.4.4 Actions to handle the parameters

- NOISE_MEASUREMENT When executed, it handled all operations needed to measure the channel noise. This task must be executed 10us after each transmission and for this reason it is triggered by event TX_10us_ELAPSED.
- SET_CHANNEL (PARAM_SET_CHANNEL) When executed, it modifies the channel by setting it to that specified by state parameters PARAM_SET_CHANNEL.
- RESET_CHANNEL When executed, it modifies the channel by setting the same used on the associated AP if the current is different.
- SET_TX_MAC_ADDRESS This action affects the MAC addresses of the frame that will be transmitted and on the MAC frame. When executed, the frame header is modified: to better understand the change we first report the original header, then the modified one:
  - **Original header** We consider a frame that a source station SA sends to distribution system DS for forwarding to destination station DA:

| To DS | From DS | Address 1 | Address 2 | Address 3 |
|---|---|---|---|---|
| 1 | 0 | BSSID | SA | DA |

Table 4.2

  - **Modified header** Distribution system DS (the AP) hop is skipped, so addresses are rearranged like follows:

| To DS | From DS | Address 1 | Address 2 | Address 3 |
|---|---|---|---|---|
| 0 | 1 | DA | BSSID | SA |

Table 4.3

  This allows a direct transmission between SA and DA, clearly they must be in their coverage. This action should hence be used when a direct link is activated between two stations and must be used paired with action ACTIVE RX DIRECT LINK on the receiver otherwise the acknowledgment will be handled in the wrong way.
- SET_RX_MAC_ADDRESS When this action is executed, the outstanding ack that will be transmitted to acknowledge the received frame will be modified: in this case the unique address in the ack will be copied from Address 3 of the incoming frame instead of Address 2. This action should be used paired with SET_TX_MAC_ADDRESS to implement a direct link between a pair of stations.
- INFLATION_CW When executed, it increases the values of the contention parameters according to the values of the backoff parameters(PARAM_INFLATION_MUL and PARAM_INFLATION_ADD). This action should be executed when the reply frame (i.e., ACK frame) for the last transmitted packet was not received within the given timeout. When the maximum number of transmission attempts has been reached, upper layers will be reported about that.
- DEFLATION_CW When executed, it decreases (or set to zero) the values of the contention parameters according to the values of the backoff parameters(PARAM_DEFLATION_DIV and PARAM_DEFLATION_SUB). This action should be executed when the reply frame (i.e., ACK frame) for the last transmitted packet was received within the given timeout. If the new value of the current contention window goes below the allowed minimum, then it is reset to the allowed minimum.
- ACTION_INCREASE_VALUE When executed, the action increases (value+1) the value of a register or memomery as specified in the parameter menu, followed the rule in the table.

| Parameters | Function |
|---|---|
| REGISTER_1 | increases the value of the REGISTER_1 |
| REGISTER_2 | increases the value of the REGISTER_2 |
| MEMORY_1 | increases the value of the MEMORY_1 |
| MEMORY_2 | increases the value of the MEMORY_2 |
| MEMORY_3 | increases the value of the MEMORY_3 |

- `ACTION_DECREASE_VALUE` When executed, the action decreases(value-1) the value of a register or memomery as specified in the parameter menu, followed the rule in the table.

| Parameters | Function |
|---|---|
| REGISTER_1 | decreases the value of the REGISTER_1 |
| REGISTER_2 | decreases the value of the REGISTER_2 |
| MEMORY_1 | decreases the value of the MEMORY_1 |
| MEMORY_2 | decreases the value of the MEMORY_2 |
| MEMORY_3 | decreases the value of the MEMORY_3 |

- `ACTION_SET_VALUE` When executed, the action set with a specic value get in the ENHANCED PARAM `PARAM_SET_VALUE` the register or memomery as specified in the parameter menu, followed the rule in the table.

| Parameters | Function |
|---|---|
| REGISTER_1 | set the value of the REGISTER_1 |
| REGISTER_2 | set the value of the REGISTER_2 |
| MEMORY_1 | set the value of the MEMORY_1 |
| MEMORY_2 | set the value of the MEMORY_2 |
| MEMORY_3 | set the value of the MEMORY_3 |

- `ACTION_RESET_VALUE` When executed, the action reset with 0 the value of a register or memomery as specified in the parameter menu, followed the rule in the table.

| Parameters | Function |
|---|---|
| REGISTER_1 | reset the value of the REGISTER_1 |
| REGISTER_2 | reset the value of the REGISTER_2 |
| MEMORY_1 | reset the value of the MEMORY_1 |
| MEMORY_2 | reset the value of the MEMORY_2 |
| MEMORY_3 | reset the value of the MEMORY_3 |

- `SET_RX_ANTENNA` When executed, it fixed the antenna used to receive the frames, the antenna is specified in the parameter menu followed the rule in the table.

| Parameters | Function |
|---|---|
| ANTENNA_0 | use main antenna |
| ANTENNA_1 | use secondary antenna |

- `SET_TX_ANTENNA` When executed, it fixed the antenna used to transmit the frames, the antenna is specified in the parameter menu followed the rule in the table.

| Parameters | Function |
|---|---|
| ANTENNA_0 | use main antenna |
| ANTENNA_1 | use secondary antenna |

# Chapter 5

# WMP Development Tool

In this chapter we presented a collection of tool to work with WMP, a tool is used to develop the MAC protocol through the definition of a FSM and convert it in a Byte-Code, other tool handled the injection of the Byte-Code inside the WMP and manage the activation of the two Binary-Byte-Codes in the WMP.

## 5.1 WMP-Editor

**W**ireless **M**AC **P**rocessor Graphic **E**ditor (WMP-Editor) is a graphical tool that represents state machine programs as transition graphs. Users can edit WMP state machine graphically, adding new states and transitions and customizing the WMP behavior working on its atomic elements, namely conditions, actions and events as introduced in Chapter 4. The same tool can be used as a compiler to translate the transition graph into a Byte-Code.

The WMP-Editor tool enables a easy and straightforward implementation of new MACs starting from their **Graphical Representation**. First, users place states in the project window and connect them with transitions. Then they can tailor each transition adding events, conditions and actions through pull down menus. Finally, when the MAC is ready, WMP-Editor can export it to the **Textual Representation**, which though being more compact is much harder to understand by users that do not know all specifics of the text format. Although one can start by writing the text file, it is always better to use the WMP-Editor: during the design phase, in fact, the tool does not allow semantic errors by construction while manual encoding might do.

The WMP-Editor renders a state machine into a user-friendly graphical representation. Thanks to the editor, the programmer can design a MAC program without having to care at Byte-Code labels for coding events, actions and conditions. It is up to the tool to translate the graphical representation of the designed state machine into a low-level Byte-Code table that can be interpreted by the MAC-Engine.

There are two main types of editor elements: **Blocks** and **Transitions**. Blocks are graphical boxes representing states of the MAC-Program, while transitions are graphical arrows representing *state changes*. Each block (i.e. each state) has a number of outgoing transitions triggered by the occurrence of events and enabled by the verification of an optional condition. The figure 5.1 show an example of simple FSM in which we highlight the main different element, the figure 5.1 also show the label for the element, if the element is a block or state the label locate the state name, otherwise, if the element is a transition, we have two label, the first identifies the name of the event, and the second identifies the name of the action, both associated with the transition. In the figure 5.1 we report a simple but worked XFSM with two states, IDLE and RX, the XFSM perform only the received operation through the three transitions able to start the receive frame, with the action `RX_START`, when the event `RX_PREAMBLE` is triggered. At the end of the frame, the event `RX_END` is triggered and the action `RX_COMPLETE` is executed. The last transition present in the figure 5.1 manage the error received event(collision or corrupted frame).

According to the transition table representation inside the Byte-Code (see section 3.4.1) and since the MAC-Engine implemented on the commercial AirForce One card by Broadcom is not able to detect interrupt signals (it reveals events only by means of a periodic polling of some event registers), in our implementation there is not a hardware difference between events and conditions. This means that the graphical compiler always maps a transition triggered by event $e$ and enabled by the verification of condition $c$ into a sequence of two transitions:

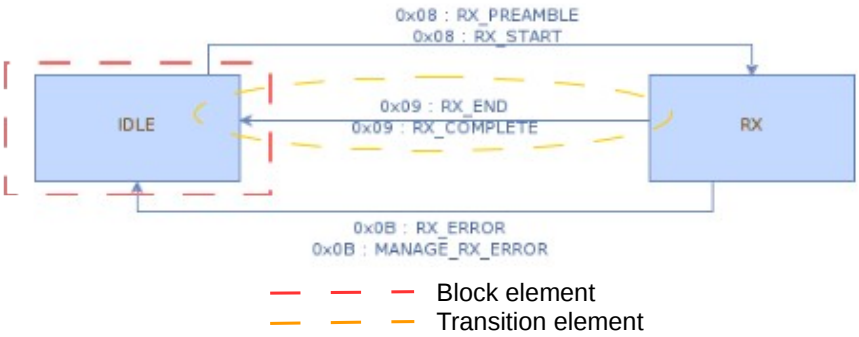



Figure 5.1: WMP-Editor element type

- an asynchronous one, triggered by transition event $e$, as soon as the Engine reveals the occurrence of that event;
- a synchronous one, immediately triggered after the first transition towards two possible states according to the TRUE/FALSE value of condition $c$.

The second transition is performed towards an intermediate state (whose permanence time is theoretically zero), from which only two outgoing transitions (corresponding to the TRUE/FALSE value of the condition) are possible. Although the programmer could completely neglect these implementation details and define state transitions by specifying both the triggering event and the enabling condition. The tool also allows to explicitly deal with the intermediate states required for verifying the enabling conditions, in this case, normal transitions are defined by means of a triggering event only, and condition states are added in the graphical representation of the machine. For improving the machine readability, conditions states are blocks with a different shape(rhombus shape), that gives emphasis on the flow splitting into two different possible outcomes (i.e. TRUE/-FALSE). In section 5.1.2 we describe how to effectively use the two different programming styles for customizing the machine behavior.

### 5.1.1 WMP-Editor element description

The basic Layout of WMP-Editor (see Figure 5.2) is an all-in-one window organized into three main frames: the left most frame, containing the the **state machine parameters**; the middle frame where the **graphical state machine** is composed; and the bottom frame that hosts the user interface for creating and modifying **program states and transitions propriety**. In details:

- **Parameters Frame** : It includes all the environment variables of the state machine. In Figure 5.3(a) we can distinguish four types of parameters: **Bootstrap**, **Enhanced**, **Backoff** and **MACflow** parameters. Bootstrap parameters set the value of the WMP configuration registers (e.g. the hardware register specifying the operating channel) and the initial state from which the MAC-Engine starts the execution. The Enhanced Parameters allow to specify other program parameters, not strictly related to the default configuration registers, such as MAC channel used by action `SET_CHANNEL` and other parameter. The Backoff parameter perform the rule to set inflation and deflation CW. At the end, MACflow parameter set offset and value to forging or match the field frame.
- **Machine Building Frame** : It displays the state blocks and the transitions defined by the programmer. WMP-Editor uses a simple right-click pop-up to add and edit state blocks and transitions, as shown in Figure 5.3(b).
- **User API Frame** : It is the bottom area of WMP-Editor where programmers modify the properties of state blocks, condition blocks (if explicitly included in the machine representation), and transition elements, by specifying events, conditions and actions for each transition from the set of available API.

### 5.1.2 WMP-Editor to define a MAC protocol

A MAC program is defined in terms of an extended state machine, i.e. a state machine in which transitions triggered by a given event can be enabled by the verification of a logical condition. An

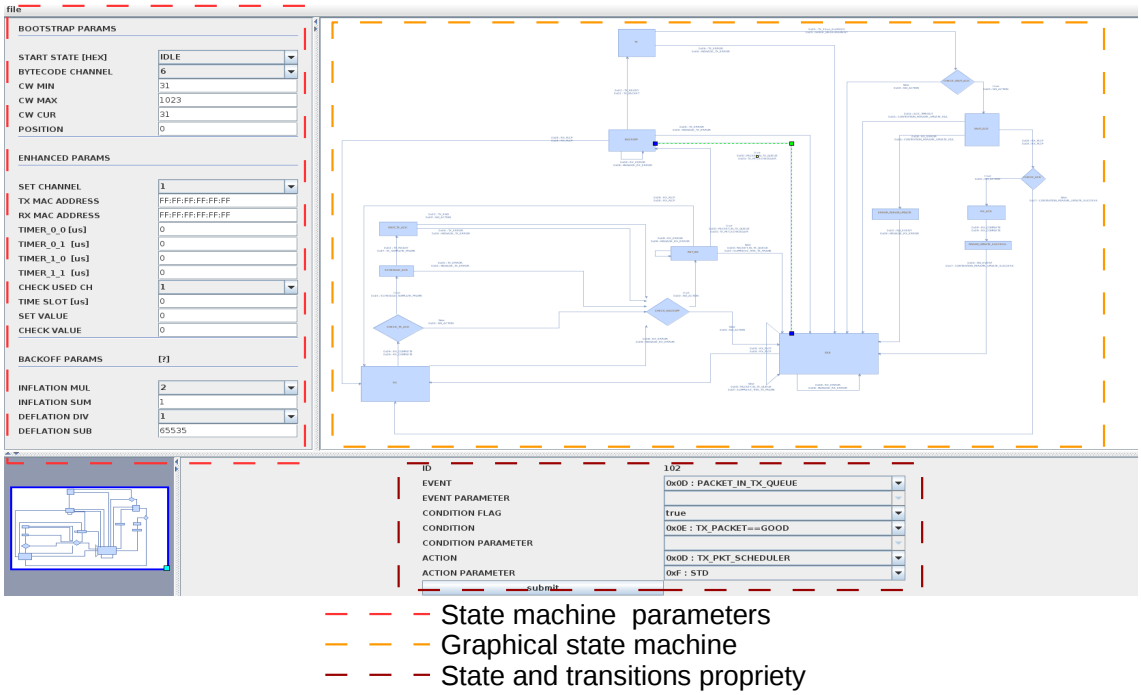



Figure 5.2: WMP-Editor Layout

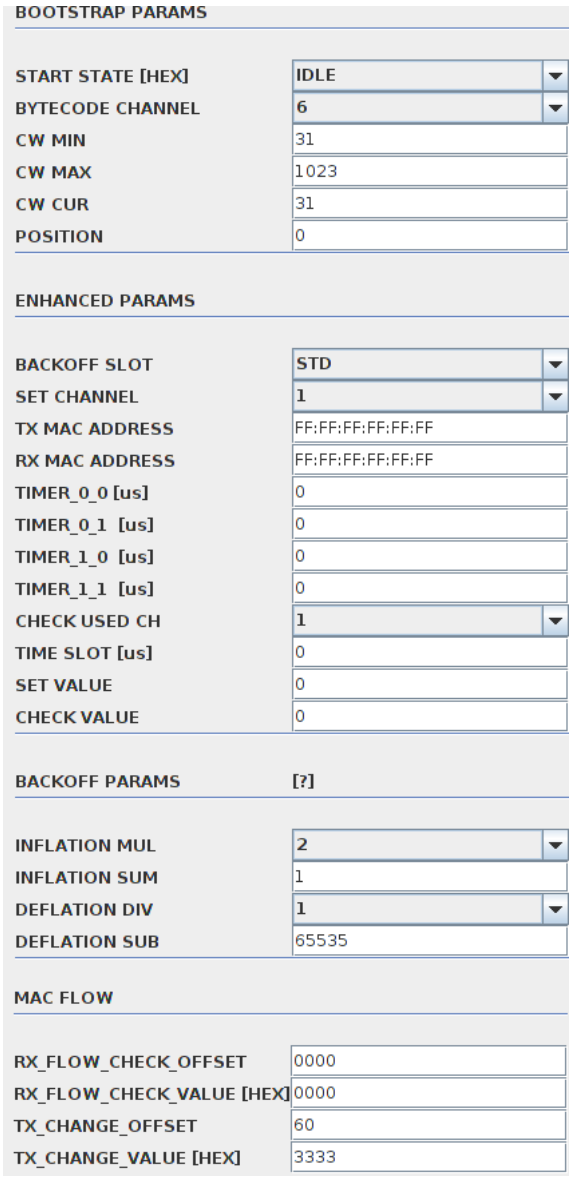
extended state machine allows to reduce the state space, since it decouples the actual state of the program into a "protocol" state (explicitly represented in the transition graph) and a "configuration" state (i.e. a list of registers), on which conditions are verified. Different approaches can be used for implementing a state machine, according to the sequence of condition verifications and action executions. In a state machine, the action is executed only if the transition is activated (i.e. the condition is verified after the occurrence of the transition event). In other cases, it could be useful to perform the action before the verification of the condition. This operation can still be mapped into a state in which (as anticipated before) a first transition without an enabling condition is performed towards an intermediate state. Before entering the new state, called condition state, the action is performed, while after entering the new state a new transition is immediately started (the trigger event is a null event) for verifying the condition and moving to the final state.

For defining a state machine in the graphical editor, it is enough to start with the definition of the states. A new state can be added in the machine building frame by means of the pop-up menu, Figure 5.4 shows a simple state example. Condition states can be created as normal states, with the only different that there are only two outgoing transitions linked to the same condition verification (TRUE/FALSE value). The condition to be verified from this state is specified in the state definition (in terms of condition label, selected from the available API). Such a state can also improve the machine readability, since it works as an IF statement in an imperative programming language.

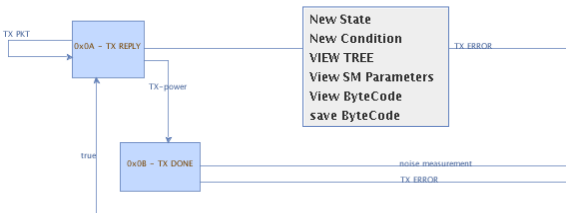
States (normal states and condition states) are connected by transitions. Transitions from normal states can specify events, conditions and actions or simply events and actions in absence of enabling conditions. Transitions from condition states specify only the TRUE/FALSE flag of the condition and the action.

Figure 5.6 shows three different examples of transitions: **transition with event** (Figure 5.6(a)), **transition with condition** (Figure 5.6(b)) and **transition with event and conditions** (Figure 5.6(c)).

A transition with event and condition can be defined through a single state like the example in figure 5.6(c) or can be defined through two series state, a state before and a condition state after, an example of two type of definition is showed in figure 5.7. The choice of a specific type of transition may depend on the programmer style, but in many cases may be optimized for reducing the number of states or the number of transitions. For example, the usage of an explicit condition state to be verified in multiple transitions (let $n$ be the number of these transitions) for entering the same states may reduce the number of transitions from $2 \cdot n$ (two transitions for each condition outcome) to $n + 2$ ($n$ transitions without enabling conditions and two more transitions from the condition state). Conversely, if we need to verify multiple conditions from a given state, it can be more efficient to use event-triggered condition-verifying conditions for limiting the state space.




(a)




(b)

Figure 5.3: Ambient parameters 5.3(a), Pop-Up Menu 5.3(b)

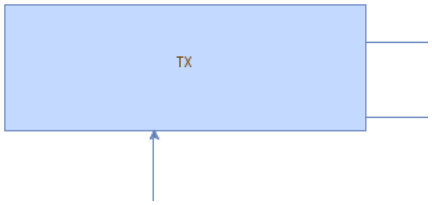
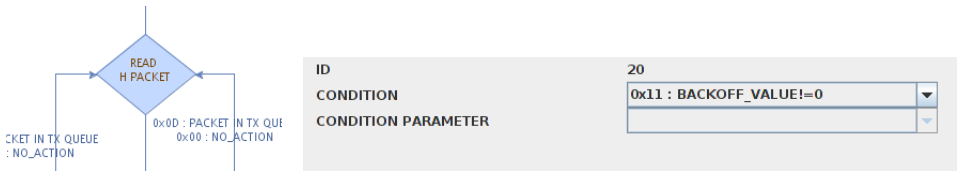



Figure 5.4: State




(a) Condition (b) Condition Parameters

Figure 5.5: Condition

Moreover, it is possible to use this type of transmissions for combining the verification of two simultaneous conditions. Each type of transition can be configured by specifying a sub-set of parameters offered by the user API frame.

**Transition with event** Figure 5.6(a) shows a transition with event in which the transition is selected(selected line). In this example, the current state is a `BACKOFF` state, from which different events are monitored including event `TX_PREAMBLE`. When the WMP reveals such an event (by polling the corresponding event register) a transition is immediately performed towards the TX state. The transition fields to be specified for this type of transitions are:

1. **EVENT**: the transition event (selected from the available API) in terms of event label;
2. **EVENT PARAMETER**: an optional parameter for defining a parametrized event (4 bits);
3. **ACTION**: the action to be performed before entering the new state (selected from the available API) in terms of action label.
4. **ACTION PARAMETER**: an optional parameter to be passed to the transition action (4 bits);

NOTE: Normal states might have multiple transitions of this type.

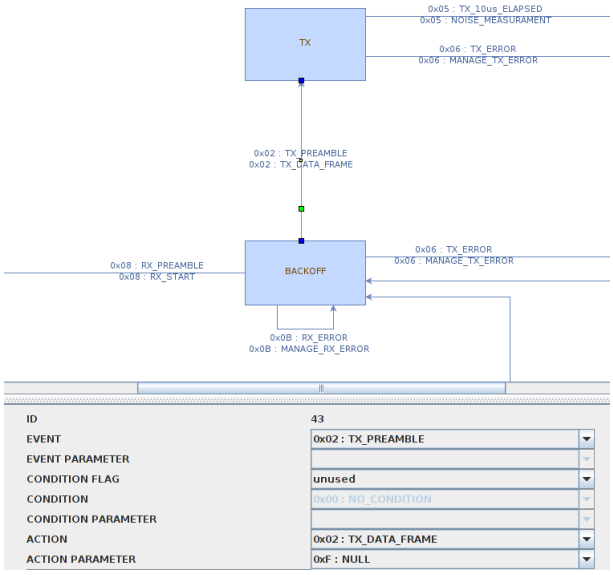
**Transition with condition** Figure 5.6(b) shows an example of transition with condition in which the transition is selected(selected line). This type of transitions are configured differently from the previous transition. Specifically, it is required to specify the following fields only:

1. **CONDITION FLAG**: the condition outcome, i.e. the TRUE or FALSE state of the condition register linked to the condition state;
2. **ACTION**: the action to be performed before entering the new state (selected from the available API) in terms of action label.
3. **ACTION PARAMETER**: an optional parameter to be passed to the transition action (4 bits);

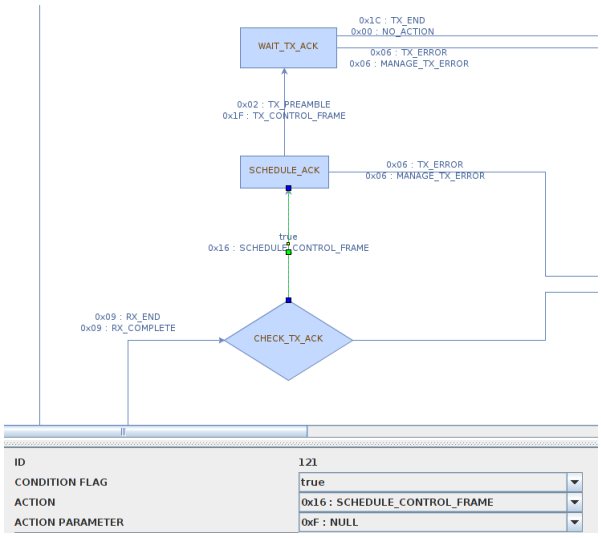
In this case the label for the condition is selected in the condition state propriety, condition state accept ONLY TWO transitions, and both have the same condition. So to simplify the definition and reduce the error, the editor perform the condition label setting in condition state propriety.

**Transition with event and condition** Figure 5.6(c) shows an example of transition with event and condition in which the transition is selected(selected line). This transition type corresponds to the transition type used in extended state machines. It is specified in transition with event and condition. In other words, after that the WMP reveals the occurrence of the transition event, a condition is verified before performing the state transition. The configuration of this type of transition requires to specify:

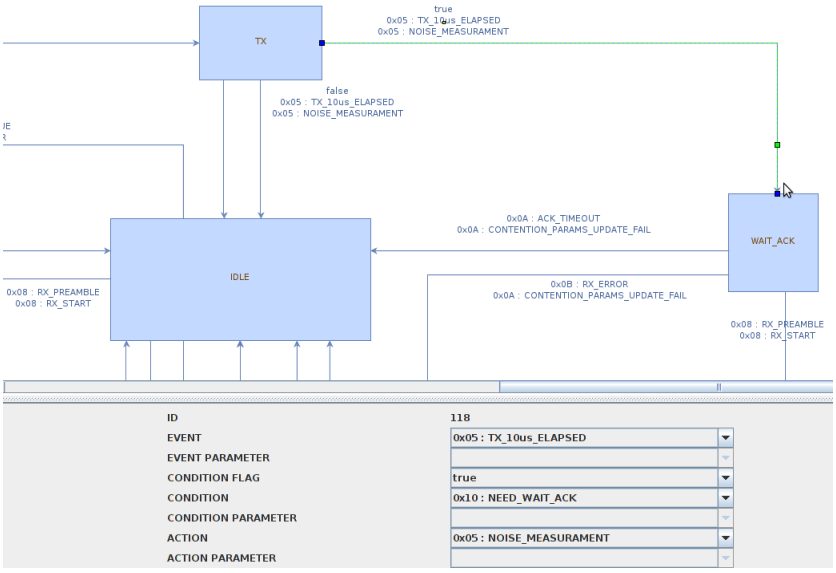
1. **EVENT**: the transition event (selected from the available API) in terms of event label;
2. **EVENT PARAMETER**: an optional parameter for defining a parametrized event (4 bits);
3. **CONDITION FLAG**: tristate field: UNUSED/TRUE/FALSE, used in transitions to enable the conditions, this field is setted unused when we have an event transition, differently it is setted TRUE or FALSE, respectively if the transition is associate with the verification of the condition or not.
4. **CONDITION**: the transition condition (selected from the available API) in terms of condition label.




(a) Transition with event




(b) Transition with condition




(c) Transition with event and condition

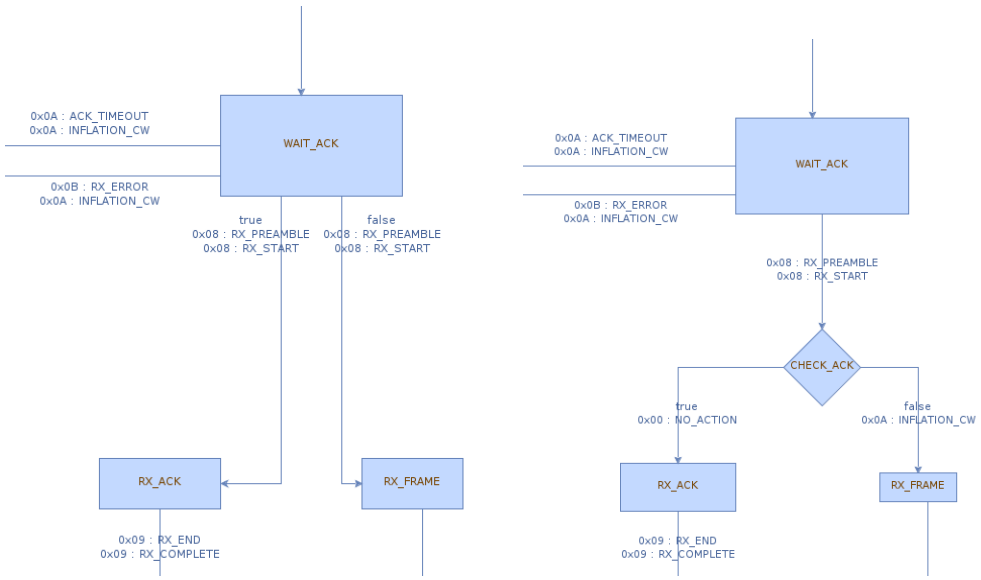
Figure 5.6: Type of Transition

5. **CONDITION PARAMETER**: an optional parameter for defining a parametrized condition (4 bits);

6. **ACTION**: the action to be performed before entering the new state (selected from the available API) in terms of action label.

7. **ACTION PARAMETER**: an optional parameter to be passed to the transition action (4 bits);

For the sake of clarity we report in Figure 5.7 a state transition characterized by an event and a condition: on the left we have two transitions from state WAIT_ACK, they are both triggered by the same event `RX_PREAMBLE` but the one that is actually followed by the WMP depends on the value of condition `NEED_WAIT_ACK`. The resulting graph involves only three states, and a conditional state (virtual state) is automatically introduced by the WMP-Editor to evaluate the condition during the translate procedure to create the equivalent Byte-Code representation. In the figure 5.7 the graph on the right, instead, involves four states even if the behavior is equivalent: here the conditional state is introduced by the user to evaluate the condition, during the model drawing.




(a) Implicit (b) Explicit

Figure 5.7: Transition with condition checking explained.

To clarify the main different between event and condition we report the follow propriety. In generic state WMP checks events in polling: if one or multiple events are to be checked in a given state, they are all polled and the first that verifies triggers the corresponding transition which is unique. If no events verify than loop restarts. A condition, instead, is evaluated istantaneously and either one or the other of the associated transitions are followed by the WMP according to the value taken be the condition itself.

## 5.2 From a model to a Binary-Byte-Code

Before to inject the XFSM created with WMP-Editor in the WMP memory to execute the MAC protocol defined, the project must be converted to the Binary-Byte-Code, in which states, virtual states, and transitions, are optimized in a logic that the WMP can understand, aimed at using optimized device memory. After create the XFSM with WMP-Editor, we can translate it in a Byte-Code and save it in a normal text file, then the Byte-Code-Manager convert the Byte-Code in Binary-Byte-Code when it realizes the injection inside the WMP memory.

To summarize the three different representation of a MAC protocol defined through a XFSM, we report in figure 5.8 a model of state machine that implement the Distribute coordination Function (DCF) MAC algorithm, for the sake of clarity we highlight the different part of the model, we have the RX procedure inside the grey box, the TX procedure inside the blue box, finally we have the Backoff and the Idle state, respectively inside the red and orange box. The WMP-Editor and the Byte-Code-Manager translates the graphical model into a running Binary-Byte-Code which actually implements DCF. More details about the graphical DCF representation will follow in Chapter 7.

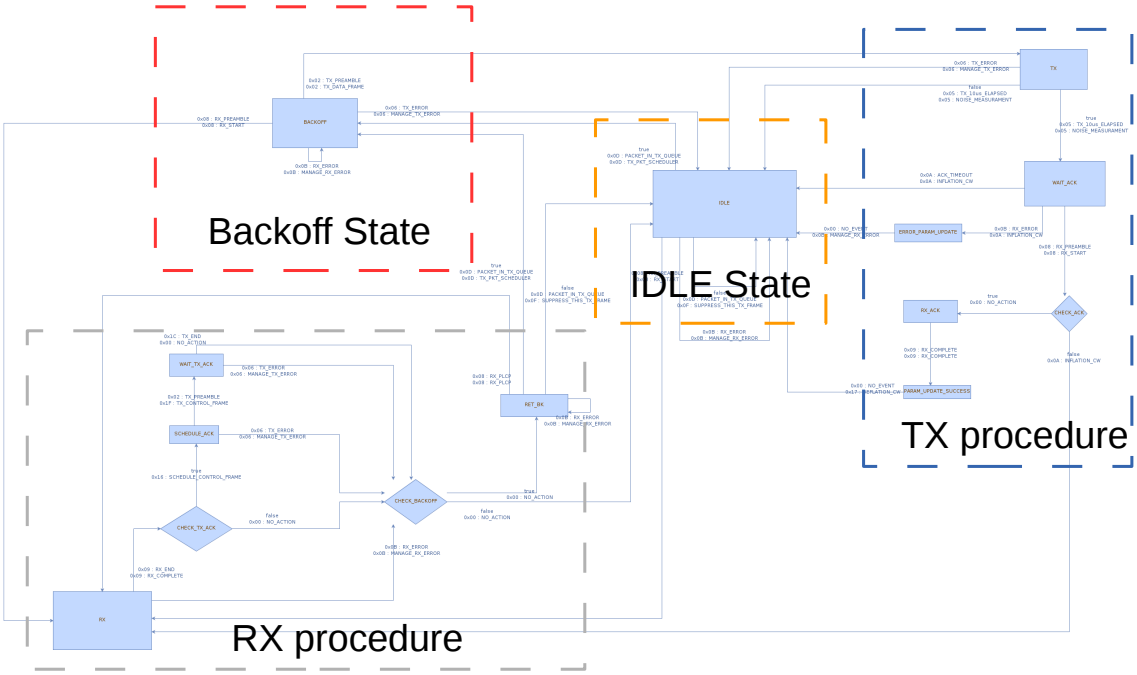



Figure 5.8: FSM implementing DCF, graphical representation.

Figure 5.9 reports an excerpt of the Byte-Code of the same state machine, and figure 5.10 report an excerpt of the equivalent Binary-Byte-Code.

## 5.3 Byte-Code-Manager

Byte-Code-Manager is a software tool that can be used to inject a Byte-Code state machine into the WMP and in general to interact with the WMP system to activate, verify and debugging operation. Though users may run it directly from the command line, in server mode the tool waits for commands from the network. The Byte-Code-Manager have many options to control the WMP, in the follows sections we explain the meaning of the different options.

The Byte-Code-Manager options list can be showed through the option "-h", an example of this command output is showed in the figure 5.11, the options "-h" also report same example to usage the Byte-Code-Manager.

### 5.3.1 Byte-Code injection and activation

The main feature of the Byte-Code-Manager is Byte-Code injection. A Byte-Code can be injected into one of two different Byte-Code areas: though area 1 is filled at startup with a default Byte-Code, it can be replaced with a new one.

This command injects the Byte-Code stored in file `mycode.txt` into area 2:

```
root@sta01# bytecode-manager -l 2 -m mycode.txt
```

and this one activates it:

```
root@sta01# bytecode-manager -a 2
```

To activate back the Byte-Code in area 1:

```
root@sta01# bytecode-manager -a 1
```

This command change the parameter Byte-Code stored in file `myparameter.txt` into Byte-Code located at the area 2:

```
root@sta01# bytecode-manager -l 2 -n myparameter.txt
```

```
000001
#State Machine Parameter
000004
0000
000004
FFFF
000004
0100
000004
FFFF

[...cut...]

#state
#combination
000010
0004 #00-IDLE
000006
0000FF0B000B0000FF08020800000F0D0E00$
000010
0906 #01-BACKOFF
000006
0000F00205020000FF0B010B0000FF0802080000FF060006$
000010
1502 #02-RX
000006
0000FF0904090000FF0B060B$
000010
1B00 #03-RX_ACK
000006
0000FF090909$

[...cut...]

000010
5AF2 #0E-Virtual State
000006
0000FF0E010D0000FF00000F$
000010
60F2 #0F-Virtual State
000006
0000FF1007050000FF000005$
000010
66F2 #10-Virtual State
000006
0000FF0E010D0000FF00000F$


000099
```

Figure 5.9: Excerpt of the FSM implementing DCF, Byte-Code representation.

```
0x0C20: 0000 FFFF 0100 FFFF FFFF FFFF FFFF FFFF
0x0C30: FFFF 0000 0000 0000 0000 0000 0000 0000
0x0C40: 0000 0000 0000 0000 0000 0100 0000 0600
0x0C50: 0100 0100 0000 FFFF 1F00 FF03 1F00 0000
0x0C60: 0000 0000 0000 0000 0000 0000 0000 0000
0x0C70: F005 FF0B 000B DB05 FF08 0208 0306 0F0D
0x0C80: 0E00 8405 F002 0502 F005 FF0B 010B DB05
0x0C90: FF08 0208 B605 FF06 0006 E405 FF09 0409
0x0CA0: F005 FF0B 060B E405 FF09 0909 3106 FF0F
0x0CB0: 0C16 0000 FF00 0600 B605 FF06 0006 A705
0x0CC0: 0F05 0F00 3D06 FF11 0800 0000 FF00 0000
0x0CD0: EA05 FF0A 000A F005 FF0B 0A0A DB05 FF08
0x0CE0: 0B08 DB05 FF08 0208 F005 FF0B 080B 0306
0x0CF0: 0F0D 1000 0000 FF00 0017 0000 FF00 000B
0x0D00: A006 0F19 0300 0000 FF00 020A 8405 FF02
0x0D10: 0D1F B605 FF06 0606 F506 FF1C 0600 B605
0x0D20: FF06 0606 2306 FF0E 010D 0000 FF00 000F
0x0D30: 3706 FF10 0705 0000 FF00 0005 2306 FF0E
0x0D40: 010D 0000 FF00 000F 0000 0000 0000 0000
0x0D50: 0000 0000 0000 0000 0000 0000 0000 0000
0x0D60: 0000 0000 0000 0000 0000 0000 0000 0000

[...cut...]

0x0F80: 0000 0000 0000 0000 0000 0000 0000 0000
0x0F90: 0014 0906 1502 1B00 1EF2 2412 2AF2 3024
0x0FA0: 3904 4200 4500 48F2 4E02 5402 5AF2 60F2
0x0FB0: 66F2 0000 0000 0000 0000 0000 0000 0000
0x0FC0: 0000 0000 0000 0000 0000 0000 0000 0000
0x0FD0: 0000 0000 0000 0000 0000 0000 0000 0000
0x0FE0: 0000 0000 0000 0000 0000 0000 0000 0000
0x0FF0: 0000 0000 0000 0000 0000 0000 0000 0000
```

Figure 5.10: FSM implementing DCF, Binary-Byte-Code representation.

```
root@alix3:~# ./bytecode-manager -h
-----------------------------------------
WMP Bytecode Manager V 1.6 - 2013
-----------------------------------------
WMP bytecode-manager byte-code injection
Usage: bytecode-manager [OPTIONS]
         -h                              Print this help text
         -l <#>                          LOAD Bytecode in specified # value (1 or 2)
         -m <name-file>                  LOAD Bytecode state-machine bytecode file
         -n <name-file>                  LOAD parameters only
         -a <#>                          Activate specified bytecode (1 or 2)
         -t <time>                       Timed Bytecode Activation [value in sec]
         -d <delay>                      Delayed Bytecode Activation in microsecond
         -f <time>                       Return the absolut time for precise
                                          equal activation [value in sec]
 -r reset setted timer
         -v                              view active and deactive condition Bytecode
         -c <ip address>                 IP address to server station Start in client mode
         -g <name-file>                  bytecode to send
         -s <interface to listen>        SERVER MODE
         -p <port number>                In server mode or client mode select specific port,
                                         if not use default port is 9898
         -x <1,2,3>                      Show Registers (1), Share Memory(2) or both(3)
         -w                              Write a frame in tamplate ram to send with
                                            specific action in the wmp API;


EXAMPLES:
--------
1. bytecode-manager -a 2                         Active the byte-code in the position 2
2. bytecode-manager -l 2 -m dcf-standard         Load the byte-code conteined in the file
                                                    dcf-standard in the position 2.
3. bytecode-manager -s                           Set the tool in server mode, in this
                                                    mode the tool listen for new byte-code
                                                    and ommand.
4. bytecode-manager -c 192.168.1.2 -a 2          Set the tool in client mode and send the
                                                    command to active the byte-code in he
                                                    position 2 for server 192.168.1.2
5. bytecode-manager -c 192.168.1.2 -g dcf-stan   Set the tool in client mode and send the
                                                    byte-code dcf-stan to server 192.168.1.2
```

Figure 5.11: Byte-Code-Manager: output of the option "h"

### 5.3.2 Delayed Byte-Code switching

The WMP can be performed an automatically switching of the preloaded Byte-Code, Byte-Code switching can be scheduled at a given time in the future, by either defining a delay or an absolute time: in both cases the event is handled by the WMP by periodically checking the internal clock. Since all stations in a given BSS synchronize their internal clock with that of the Access Point, the second mechanism allows to switch the Byte-Code on several station at the same time.

This command schedules a Byte-Code switch after twenty seconds:

```
root@sta01# bytecode-manager -t 20
```

This command schedules a Byte-Code switch at a given time:

```
root@sta01# bytecode-manager -d <value-time-us>
```

where <value-time-us> is an accurate clock reference expressed in microsecond. When the internal clock reaches <value-time-us>, the WMP deactivates the current active Byte-Code and activate the other one.

Again Binary-Byte-Code can be used to get the <value-time-us> corresponding to a given delay:

```
root@sta01# bytecode-manager -f <delay-in-second>
```

The output value is expressed in microseconds and is computed by summing the input <delay-in-second> to the internal clock. For example, if we want to switch the Byte-Code on all stations in 12 seconds we should first get the reference time on one station, i.e.,

```
root@sta01# bytecode-manager -f 12

-----------------------------------------
WMP Bytecode Manager V 1.6 - 2013
-----------------------------------------
Selected find absolute time
Current work mode : "local"
-------------------------------------
Calculation value of activation delay
time stamp : 3076057456
-------------------------------------
```

Then we must run option `-d` on all stations using the `time stamp` value that was returned (3076057456).

To cancel timers, run:

```
root@sta01# bytecode-manager -r
```

There are specifics actions and conditions in to api WMP that are used to manipulate the 2 registers and 3 memory locations present, is possible view the value of this registers also by Byte-Code-Manager with the usage of option **-v**, the option also display the information about timers run, an example of this option output is:

```
root@alix2:~# ./bytecode-manager -v
-----------------------------------------
WMP Bytecode Manager V 1.6 - 2013
-----------------------------------------
Current work mode : "local"
Selected view

-------------------------------------
WMP INFORMATION

CURRENT BYTECODE                = 1
Control Value                   = 0x4000
Timer Not Active
Delay Not Active
-------------------------------------
```

```
--------------------------------------
REGISTER AND MEMORY INFORMATION

Current contention windows       = 0x001F
Max contention windows           = 0x01FF
Min contention windows           = 0x001F
Register 1                       = 0x0000
Register 2                       = 0x0000
Memory 1                         = 0x0000
Memory 2                         = 0x0000
Memory 3                         = 0x0000
--------------------------------------
```

### 5.3.3 Debugging options

Finally, there is a option to debug the WMP, through the option "-x" is possible getting a dump of the WMP memory, when run the command <bytecode-manager -x 2> we obtain a print of the shared memory in which, we can find the two Binary-Byte-Code s and all other date saved in this memory, we report an excerpt of the option "-x" in the follow quote.

```
root@sta02# bytecode-manager -x 2
bytecode-manager -x 2
-----------------------------------------
WMP Bytecode Manager V 1.6 - 2013
-----------------------------------------
Current work mode : "local"
SHM dump 2

Shared memory:

0x0000: 9A01 7008 FFFF 0A7C 0000 0000 C000 0A00
0x0010: 1400 0000 8000 0900 4700 4700 8301 6400
0x0020: 3009 C0FC 0000 0000 0000 0000 0000 0000
[...cut...]
```

### 5.3.4 Forge control frame

There are specifics actions and events in to WMP API that are used for send a control frame that is stored in the template ram of the device, is possible change many field of this template frame with the option -w, the field that is possible modify are:

- length frame
- rate
- destination address
- frame string text

This section describes the option used to forge a template control frame that is possible send with specific action of the WMP.

```
root@alix2:~# ./bytecode-manager -w
-----------------------------------------
WMP Bytecode Manager V 1.6 - 2013
-----------------------------------------
Current work mode : "local"
Write frame into template ram

insert length frame : 200
[1] - 1Mbps
```

```
[6] - 6Mbps
[12] - 12Mbps
[18] - 18Mbps
[24] - 24Mbps
[36] - 36Mbps
[48] - 48Mbps
[54] - 54Mbps
insert rate : 24
insert destination address[12:34:56:78:9a:bc] = 12:34:56:78:9a:bd
insert frame string text : do
Write template frame success
```

# Chapter 6

# WMP implementation on commodity hardware

## 6.1 Introduction

To prove the viability of Wireless MAC processors, we challenged its implementation over an ultra-cheap commodity WLAN network interface card. We worked on the AirForce54G chipset from Broadcom [44]. Our implementation replaces the original card firmware with an assembly code implementing the WMP and its state machine execution engine, and maps the previously described WMP programming interface into actual signals, operations and registers of the card. For supporting the upper-MAC operations and interacting with the other protocol layers, we use the b43 [45] soft-MAC driver, which adapts the Linux internal mac80211 [46] interface to network card [5]. This work has been possible thanks to the availability of a documented open firmware for a specific chipset of the big AirForce54G family of Broadcom wireless NICs, namely the OpenFWWF Project [1]. In this chapter we make an overview of the Airforce54G hardware platform and we analyze in detail the main configuration registers and the implementation choices, at last we report the program flow chart for the MAC-Engine develop.

## 6.2 The Hardware Platform AirForce54G

The chipset AirForce54G has a general purpose processor for which assembling tools are available. No vendor has to date released an open source firmware for wireless card, and the only available public-domain code is OpenFWWF, therefore, since an open source firmware has been released for this chipset, information about registers, timers and transmission and reception commands are available. Specifically, the platform is equipped with:

- 8 MHz CPU with 64 registers;
- Arithmetic, binary, logic and flow control operations;
- 4KB of data memory with direct or indirect access;
- Template memory for composing packets;
- 32KB code memory;
- Hardware configuration registers (e.g. channel, power, etc.);
- Hardware notification or conditional registers (e.g. packet queued, plcp end, crc failure, etc);

The figure 6.1 shows a digram block of the airforce54g, all the block are connected through arrows that notify the user on the logical information flow control, the main element, at center of the figure 6.1, is the 8MHz CPU with 64 internal registers named General Purpose Register (GPR). The CPU controls the RX en TX engine through the registers named SPR, other type of registers are physical registers, they perform the control of the PHY block. The firmware can access to the physical registers through 2 SPR. The SPR registers keep hardware configuration settings. They

[1] OpenFirmWare for WiFinetworks, http://www.ing.unibs.it/openfwwf

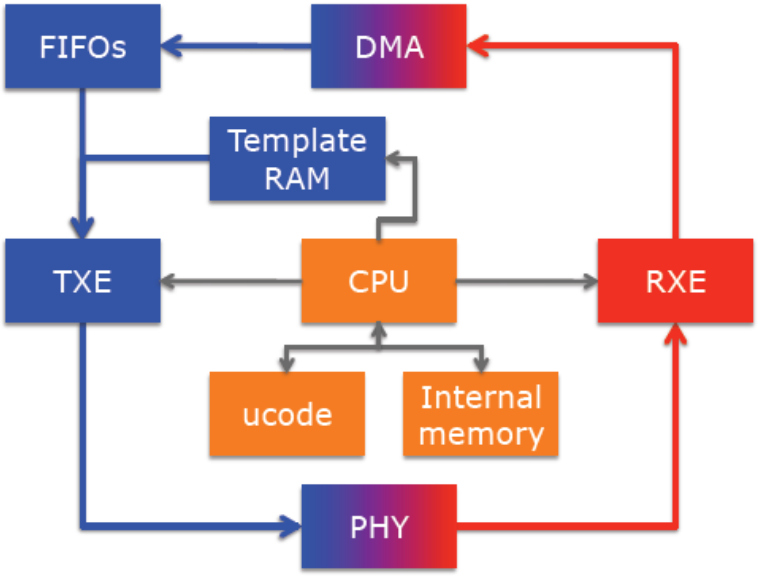



Figure 6.1: Airforce54 element

can be set by the firmware in response to changes in the program, in the radio interface and set up timers. Physical registers setting the parameters for the physical layer, for example, to set the channel bandwidth we need use a physical register. The last kind of registers are the conditional registers, the Broadcom chipset is not able to detect interrupt signals, for this reason the firmware implementing a periodic polling of the conditional register to detect the hardware status or up and down requests. The other main block are:

- **TX and RX FIFO queues** represented by FIFOs block shown in the upper left in figure 6.1, they are the queues interfaced to the host kernel. On the transmission path, packets forwarded from the driver are enqueued in the TX queue, from which the chipset pull frames and moves them into the TX engine. On the opposite path, the processor waits for a packet received by the RX engine, and pushes (or drops) the received data towards the host kernel.
- **TX and RX engine** positioned to the right and left of the figure 6.1, they encode and decode packets from internal representation to the 802.11b/g CCK and OFDM encodings; compute and verify the Frame Check Sequence; transmit and receive frames. TXE get the outgoing packets from the FIFOs block and sent them toward the PHY block. Instead the RXE move the incoming packets from PHY block to the FIFOs block, using the DMA block, indeed packet reception is performed by the RX engine in parallel to other processor tasks.
- **SHared Memory (SHM)** or internal memory together with the ucode block are display in the center of the figure 6.1 and are directly connected with the CPU. This memory space of 4 KB can be accessed also by the host, shared memory is used from firmware to save parts of oncoming and outgoing packets to content analysis and comparison, the firmware also uses this memory to save other parameter information that not require fast access to be retrieved.
- **Internal code memory** or ucode is a 32 KB memory used to save the compiled firmware that the CPU execute.
- **Template RAM** is a 32KB RAM memory used for composing arbitrary frames that can be pushed to the TX engine as if they came from the TX queue. The template memory also contain the ACK frame and beacon frame, this type of frames have always, the same content, they have only a variable little part like destination address or some flags, the firmware modify these little parts before send this type of control frames.

Ultimately, the airforce54g platform contain three different type of memory, they are Shared Memory, Template Memory and Serial Programmable Read Only Memory (SPROM), the first two have been discussed in previous sections, the latter, SPROM is a PROM able to contain permanent information as chipset code identification and station MAC address.

### 6.2.1 Platform : registers, memory addressing and subroutine system

Hardware notification and configuration is perform through several hardware registers belonging to two different types, SPR and physical registers, we discuss the latter in the previous sections, in this section we present a detail of the more important registers able to control the TX/RX engine and notify the chipset status, clustered based on their function.

The platform has different registers able to addressing the SHM and call subroutine. Two type of memory addressing are available, the first is a **Direct Addressing** in which the memory address is specified in the instruction code within square brackets, and another technique type **Register Indirect Address**, in which we use a register to contain a point to memory, this technique can has an offset explicit in instruction code, in this case is named "Displacement Addressing". The chipset has 6 SPR to perform the register indirect address, they named offset register (SPR_BASEx with x from 0 to 5), normally some of them contain constant values that point to important parts of the SHM like the headers for incoming and outgoing frames, these pointers not change, and the firmware use their frequently.

To implement the subroutine system, the platform uses four link registers (lrx with x from 0 to 3) to store the current program counter before perform a jump operation, the jump occurs through a instruction code named **call**, and the equivalent return through the instruction **ret**. Through the call instruction parameter we can specify what link register use to store the current program counter, the same register must been used inside the subroutine, as parameter, in the ret instruction to back return. We have only 4 link register, so, we ability up to 4 calls nested.

#### 6.2.1.1 SPR

**SPR to handle the TXE**

- **SPR_TXE0_TX_SHM_ADDR** : The header of the outgoing and incoming frames are copied in the SHM in order to evaluate part of them, we can use this register to write the SHM address used to copy the TX header.
- **SPR_TXE0_CTL** : This register is used to control the TX engine. The value of this register change according to the frame that must be sent. Set this register to schedule a transmission in the future, using different values for different frames. As long as we can see the SPR_TXE0_CTL has values reported in the table 6.1.

| Register value | time before TX |
|---|---|
| 0x4d95 | PIFS |
| 0x4021 | SIFS |
| 0x4c1d | DIFS |
| 0x4001 | transmit immediately |

Table 6.1: Value for SPR_TXE0_CTL

- **SPR_TXE0_PHY_CTL** : Used to set the frame encoding (OFDM or CCK), long or short preamble, and antenna to transmit the packet.
- **SPR_TXE0_FIFO_CMD** : This register is used to tell the TX engine which FIFO must be used and which operation must be performed, The Tx engine can perform different operations on the frame into the FIFO queues. The number of the FIFO from where the frame must be picked and the operation that must be performed on that frame were stored on FIFO command register. There are at least 2 operations that can be performed, flush the frame in the queue and transmit the frame in the queue.
- **SPR_TXE0_SELECT - SPR_TXE0_TX_COUNT** : These two registers are used to set copy informations in the TX engine. This register tells the TXE from which source it has to pick data and where it has to put it. It is possible to specify the length of data that TXE will take. We can specify the source (Template RAM or Fifos), Length through register SPR_TXE0_TX_COUNT, and Destination (Serializer or SHM). These registers allow firmware to transfer frames, or parts of them, from one memory to another. This is very useful when firmware needs to analyze frame header or to modify part of the frame.

- **SPR_TXE0_TIMEOUT** : This register handle the timer able to the ACK receive, when a station transmits a data frame it must wait for the ACK. If the station doesn't recive the ACK before time-out expiration it must perform a retransmission. This register set and enable the ACK time-out, the register contain the value of the ACK timeout in microseconds.

- **SPR_TXE0_Template_TX_Pointer - SPR_TXE0_Template_Pointer - SPR_TXE0_Template_Data_Low - SPR_TXE0_Template_Data_High** : This kind of registers perform the operations in the Template memory, the firmware can read, write date in the Template memory and send frame present in the tamplate, in this case the firmware must specify pointer to the beginning of the frame.

- **SPR_TXE0_WMx (0-2-4 .. 62) - SPR_TME_VALx (0-2-4 .. 62) - SPR_TME_MASKx (0-1)** : This kind of registers are able to modify the first portion of the outgoing frames, this functionality is activated from the Transmit Modify Engine (TME), and it is used by the firmware to modify the pending frame that is ready for transmission inside the serializer. Up to 64 bytes of the original frame can be modified, respectively the registers enabling the words modify, set the value for the word and set the words masks. For example, let's say that you want to change the 802.11 "Address 1" header field: to do that you can simply write the value into the "TME fill value", set the corresponding bit of the "TME fill mask" and enable the modification with TXE_WMx. It's important remember that TXE_WM0 + TXE_WM1 are 32 bits long (16 + 16). So if you put TXE_WM0 = 0x1 you enable byte 0 and byte 1 modification.

**SPR to handle the RXE**

- **SPR_RXE_RXHDR_OFFSET - SPR_RXE_RXHDR_LEN** : Equivalent to SPR_TXE0_TX_SHM_ADDR but for receive process, all the receive frames headers are copied in the SHM order to evaluated part of them, this two registers perform the setting for the receive SHM address, this is where the RXE puts the received frame header into the SHM, and receive copy length (amount of bytes the RXE copies into SHM).

- **SPR_RXE_FIFOCTL0 SPR_RXE_FIFOCTL1** : These two registers control the receive process, after the PLCP of a packet is received, the firmware enables data passing between the receiver and the FIFO. When rx is complete, then the firmware pushes the packet in the FIFO up to the host. All these operations are performed through this register setting.

- **SPR_RXE_FRAMELEN** : This register reports even while RX in progress and hence increases during reception.

- **SPR_RXE_Copy_Offset - SPR_RXE_Copy_Length** : These two registers are able to copies received frame into the RX host queue.

**SPR to handle the timers** This paragraph report all the registers are able to work with the chipset timers, they are Timing Syncronization Function (TSF), IFS, and Network Allocation Vector (NAV). The latter is virtual carrier sensing mechanism used with wireless network protocols such as IEEE 802.11. The virtual carrier sensing is a logical abstraction which limits the need for physical carrier sensing at the air interface in order to save power. The MAC layer frame headers contain a Duration field that specifies the transmission time required for the frame, in which time the medium will be busy. NAV is an indicator for a station on how long it must defer from accessing the medium

- **SPR_TSF_WORDx (0-3)** : The TSF is a 64 bit timer running at 1 MHz and must be updated by beacon and probe response frames from other stations. The value of the timer is located in the beacon and probe response frames as a timestamp. When an ad-hoc station or AP first begins operation, it resets its TSF timer to zero. It then starts sending beacon frames (the default is every 100mS) containing the TSF clock and the beacon period. This establishes the basic beaconing period of the IBSS. These four registers keep the timers for the station in the same Basic Service Set (BSS) synchronized.

- **SPR_IFS_slot_duration** : This register setup slot duration in $\mu s$.

- **SPR_TSF_GPTx_STAT - SPR_TSF_GPTx_CNTLO - SPR_TSF_GPTx_CNTHI - SPR_TSF_GPTx_VALLO - SPR_TSF_GPTx_VALHI (0-1)** : The broadcom chipset is equipped with three General Purpose Timer (GPT) that work at 8MHz or 88MHz, these registers are

able to set up the timers and switch on/off, the **SPR_TSF_GPT_ALL_STAT** register when read notify the time-out status for all GPT .

- **SPR_TSF_Random** : This register contain a random value, it is used to extract a random backoff value into the current contention window value.
- **SPR_IFS_STAT** : This register is used from firmware to trace the backoff process status.
- **SPR_IFS_BKOFFDELAY** : This register is setup after each transmission attempts by the firmware by loading a random value into contention window after each failure. It counts down the number of slots written by the firmware, countdown restarts each time after a channel busy episode as soon as the medium has been detected idle (phy+nav) for a couple of slots (read IFS_STAT). Countdown is paused during channel busy (either phy or nav).
- **SPR_NAV_CTL - SPR_NAV_STAT - SPR_NAV_ALLOCATION** : These registers setup the NAV carrier sensing mechanism used with wireless network protocols. The firmware listening on the wireless medium read the Duration field and set the NAV using the register SPR_NAV_ALLOCATION. SPR_NAV_CTL enables NAV countdown according to value in SPR_NAV_ALLOCATION. SPR_NAV_STAT is enables when NAV countdown is running. NAV is a 8MHz countdown, when counting down, medium is (NAV) busy. The NAV setup with time in microseconds.
- **SPR_IFS_med_busy_ctl** : Broadcom chipset maintains 16-bit register counters to track "medium busy time". The time counter is incremented at every $\mu$s for which the medium was sensed busy. The medium is considered busy if the measured signal strength is greater than the Clear Channel Assessment (CCA). The SPR_IFS_med_busy_ctl register counts the number of $\mu$s during which the medium is busy.
- **SPR_IFS_if_tx_duration** : This register counts the number of us for the current outgoing transmission and reset to zero at each tx start.

#### 6.2.1.2 Conditional registers

**Conditional registers to handle the TXE**

- **COND_TX_NOW** : This condition register is triggered when backoff procedure ended so that the transmission of the frame can be handled.
- **COND_TX_DONE** : This condition register is triggered when the PHY has finished transmitting all bits.

**Conditional registers to handle the RXE**

- **COND_RX_PLCP** : This condition register is triggered when the chipset receive a valid PLCP header.
- **COND_RX_FCS_GOOD** : This condition register is triggered when the FCS of a received frame is good.
- **COND_RX_BADPLCP** : This condition register is triggered when corrupt PLCP has received.
- **COND_RX_COMPLETE** : This event is triggered at the end of the current reception.
- **COND_RX_FIFOBUSY** : When rx is complete, then the firmware pushes the packet in the FIFO up to the host, to be sure the transfer completes, it is necessary to check that the FIFO begins working AND stop working by checking COND_RX_FIFOBUSY.

### 6.2.2 Development tools

The Broadcom wireless cards have a tools for compiling the firmware source code and debugging it, all tools can be downloaded ad the address [17], further the debugging tools contain other element to help the programmer aimed at find the bug, and dump the memory status, the developer tools are:

- Assembler
- Debug

- Disassembler
- fwcutter
- ssb_sprom

The main tool is the Assembler named "**b43-asm**" that perform the firmware compiling, the b43-asm also help the programmer to understand the microcode instruction and as addressing the memory locations.

The tool for the debug is "**b43-fwdump**" able to dump the memory information, b43-fwdump is an utility for dumping the current status of the device firmware on a running device. Different option can be used, if you want a Shared Memory dump, or if you want a automatically dump of the disassembled code at the current PC. This is convenient for debugging firmware crashes. b43-fwdump as it needs direct access to the hardware through debugfs [47].

From the compiled firmware, the **Disassembler** tool disassemble it to obtain a source code, the disadvantage of this tool is that the source code is not human-readable because not present the defined names. Other tool named b43-beautifier replace constant expressions in raw disassembled firmware code with human-readable defined names. The tool requires a path to the directory containing the hardware definitions. This is the "common" subdirectory found in the b43-ucode GIT repository [15].

**b43-fwcutter** is a tool to extract firmware from binary Broadcom 43xx driver files present in the [45].

Finally, **ssb_sprom** is a tool for the convenient modification of the SPROM. This tool read and write the SPROM and automatically adjusts the CRC after the modification.

## 6.3 WMP Implementation

In this section we report the implementation choices to develop the WMP, regarding the firmware, we also worked on the legacy micro-code to implementing the MAC-Engine work-flow, for allowing the execution of the Binary-Byte-Code. We developed a micro-code procedures corresponding to the WMP Control API, we implement them in the form of subroutine that the MAC-Engine can execute when an events or conditions or action must been performed. We also reserve part of GPR indicating the status of the program and the Binary-Byte-Code slot under execution.
After an analysis of the space required for saving the Binary-Byte-Codes, we chose of use the second part of the SHM to save two Binary-Byte-Codes, there is a other reason to make it, the SHM can be access by the host, accordingly we could develop the Byte-Code-Manager option that inject Binary-Byte-Code in the correct slot of the SHM. We also chose the internal code memory to implementing the MAC-Engine and the WMP API. We use template RAM to store the control frames that can be pushed to the TX engine as if they came from the TX queue. Set-up registers and notifications register are the basis to develop the WMP API.
Our actual MAC-Engine implementation works as summarized in figure 6.2 to trigger the events in a states and in figure 6.3 to trigger the condition in virtual states. The main difference with the WMP conceptual architecture is the management of events. Hardware signals are in fact directly handled by registers and cannot be asynchronously intercepted by the MAC-Engine implemented in the firmware, because the chipset not provide the interrupt functions. For example, the end of a frame reception (`RX_END` event of the WMP) is signalled by a change of status of a specific frame reception register (COND_RX_COMPLETE). In the next paragraphs we refer to the memory Binary-Byte-Code organization discuss in the section 3.6.

**State flow chart** The figure 6.2 reports the work flow chart execute from MAC-Engine to load the state and wait for events trigger that are associated with the transitions state.

In the initialization phase, identified by the elliptical box in the figure 6.2, the start state is loaded, from which, we must begin to execute the state machine. For all the states, the MAC-Engine uses the state number to point at memory location that contains that state, in the states region of the Binary-Byte-Code, from the state location memory, the MAC-Engine get the position of the first transition, and number of transitions associated with the state. Hence, the MAC-Engine pre-fetches the events list of the state under execution, and start the polling cycle. To limit polling delay, we insert in the Binary-Byte-Code transition specification, the physical address of the procedure (sub routine) that implement the API events and pre-fetch their before start the cycle. After checking of each event present in the list, the MAC-Engine checks if it is the last, if so, the MAC-Engine restarts the polling cycle. The MAC-Engine traces and stores the pointer to the last event occurred, when it

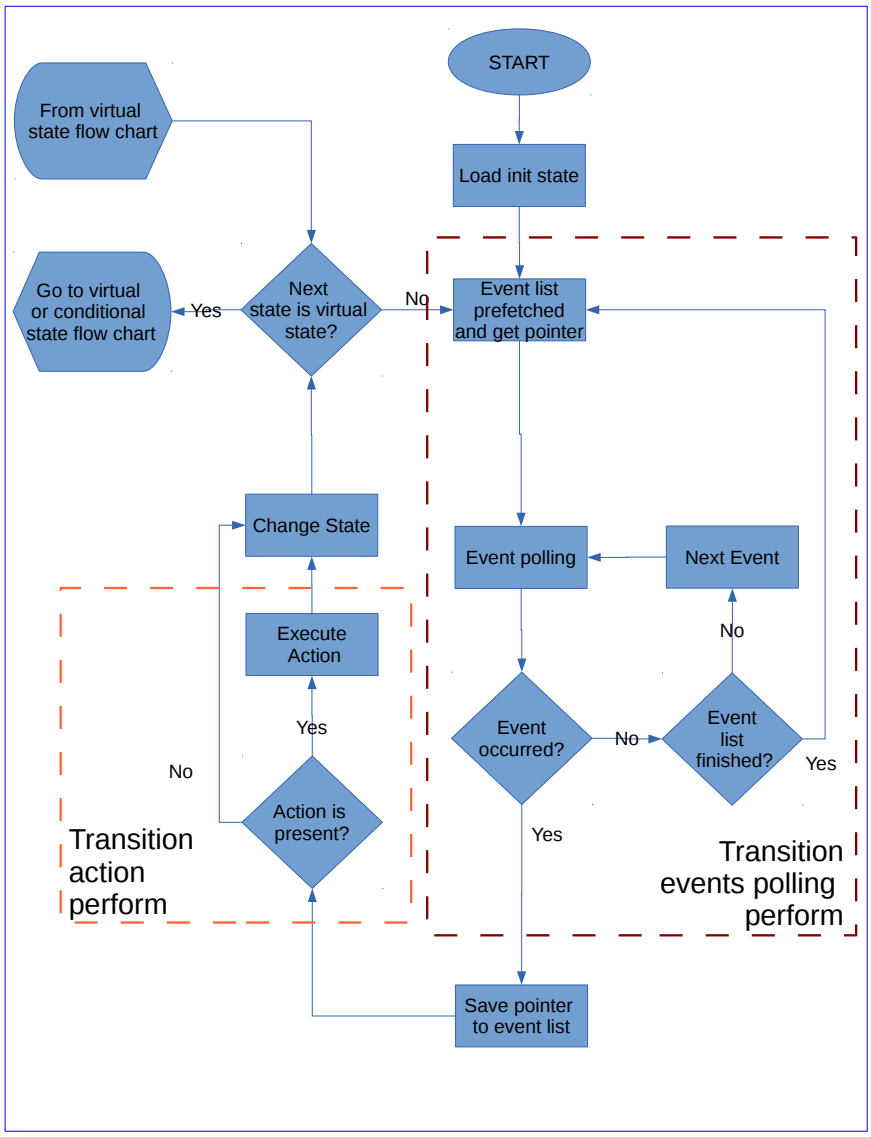



Figure 6.2: MAC-Engine flow chart state execution

came back in the state, MAC-Engine restarts the polling cycle from the next point to stored event, this feature ensures that all events have the same priority to be read triggered.

In case of events triggering, after save the pointer to last event triggered, the MAC-Engine evaluates the transition associated with the event, and it reads the action label to recover the action address procedure, if present, the transition action is executed by the engine, then the MAC-Engine performs a state changing. At this stage, the MAC-Engine check if the next state is virtual state or a conditional state, if so, the state executions follow a different flow chart. Otherwise the events list for the new state is pre-fetched and the above process starts again.

**Condition and virtual state flow chart** A condition state and virtual state, instead, is evaluated instantaneously and either one or the other of the associated transitions are followed by the WMP according to the value taken be the condition itself. In this case, the WMP state executions follow the flow chart shows in figure 6.3, no polling cycle is performed now, because this kind of states have only two transition associated on the same condition. One if the conditions is checked true and latter if condition is checked false. After, the flow chart executes the action associated with the transition triggered, if present, and perform a new change of state. Ultimately, return to flow chart state executions to check again the type of the next state.

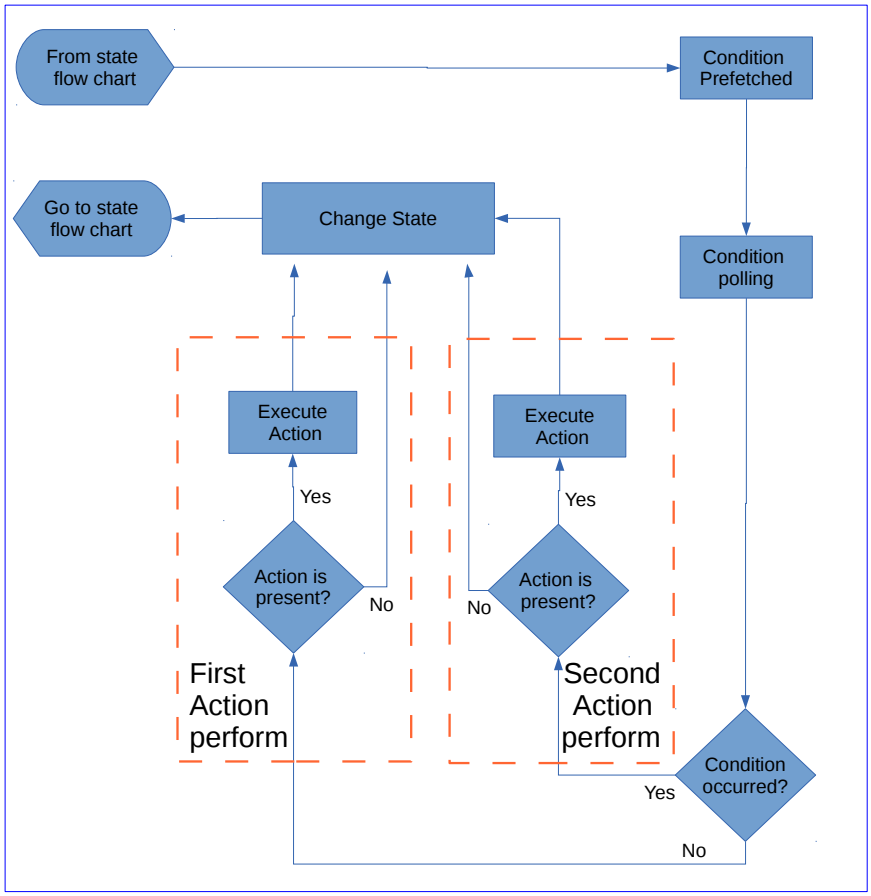



Figure 6.3: MAC-Engine flow chart virtual and conditional states

From the above discussion, it follows that, the our implementation does not strictly distinguish between events and conditions, both of them being verified by monitoring card registers. However, the condition verification does not require a cyclic polling.

# Chapter 7

# MAC design and evaluation

## 7.1 Design Overview

The combination of the WMP, WMP-Editor and Byte-Code-Manager is a complete and cheap toolchain that allows developing and testing a new MAC scheme in a very simple, robust and quick way over an ultra-cheap platform. As see in the previous chapter the WMP-Editor is easy way to obtain a Byte-Code, in this chapter we present different MAC protocol and how these can be designed by the tool.

We have realized several MAC example to test and support the WMP platform, current model implementation supports both the infrastructure and the ad-hoc mode, it is compatible (in terms of protocol timings, frame fields, etc.) with legacy DCF stations in b and g mode, and it provides throughput performance and power consumption comparable with the proprietary card firmware when executing the DCF state machine. For monitoring the behavior of the MAC program executions, apart from measuring the throughput performance, we also used a customized tool for acquiring and processing channel activity traces. The trace acquisition is based on USRP [13], while the trace processing is performed in MATLAB for deriving the power levels of the channel samples.

This chapter describes three examples of state machine graphical implementation, namely:

- **DCF** This state machine implements the standard IEEE Distributed Coordination Function (DCF), here the Backoff is customized according to three evolutions, normal, fixed and disabled.
- **TDM** This state machine is a basic variants of the DCF, does not update the contention windows parameter and schedules packet transmissions at fixed slot times inside a frame according with the other stations. Mac timer is used to synchronized all the stations in the same Basic Service Set (BSS).
- **DLS** The Direct Link Setup (DLS) is a state machine derived from DCF and changes the packet forwarding style according to the MAC address of the destination, for a set of selected targets the standard transmission procedure (each packet is sent to the AP) is overridden and packets are sent directly to destination stations without forwarding through the Access Point.

The following paragraphs explain the logic used to create a state machine and report the validation result:

## 7.2 Distributed Coordination Functions (DCF)

In order to validate the proposed approach, we (re)implement the legacy 802.11 DCF as an XFSM executed by the WMP, and compare its performance with the benchmark provided by the native Broadcom's firmware.

The DCF state machine implements standard IEEE 802.11 functions using WMP APIs. Figure 7.1 shows the graph associated to the state machine that can be logically split into two parts: one **handling incoming packets** reported on the left side of the Figure and put inside a grey box, another **handling outgoing packets** put inside a blue box and reported on the right side. Whether switching to one or the other is decided by the initial state `IDLE` according to the following events:

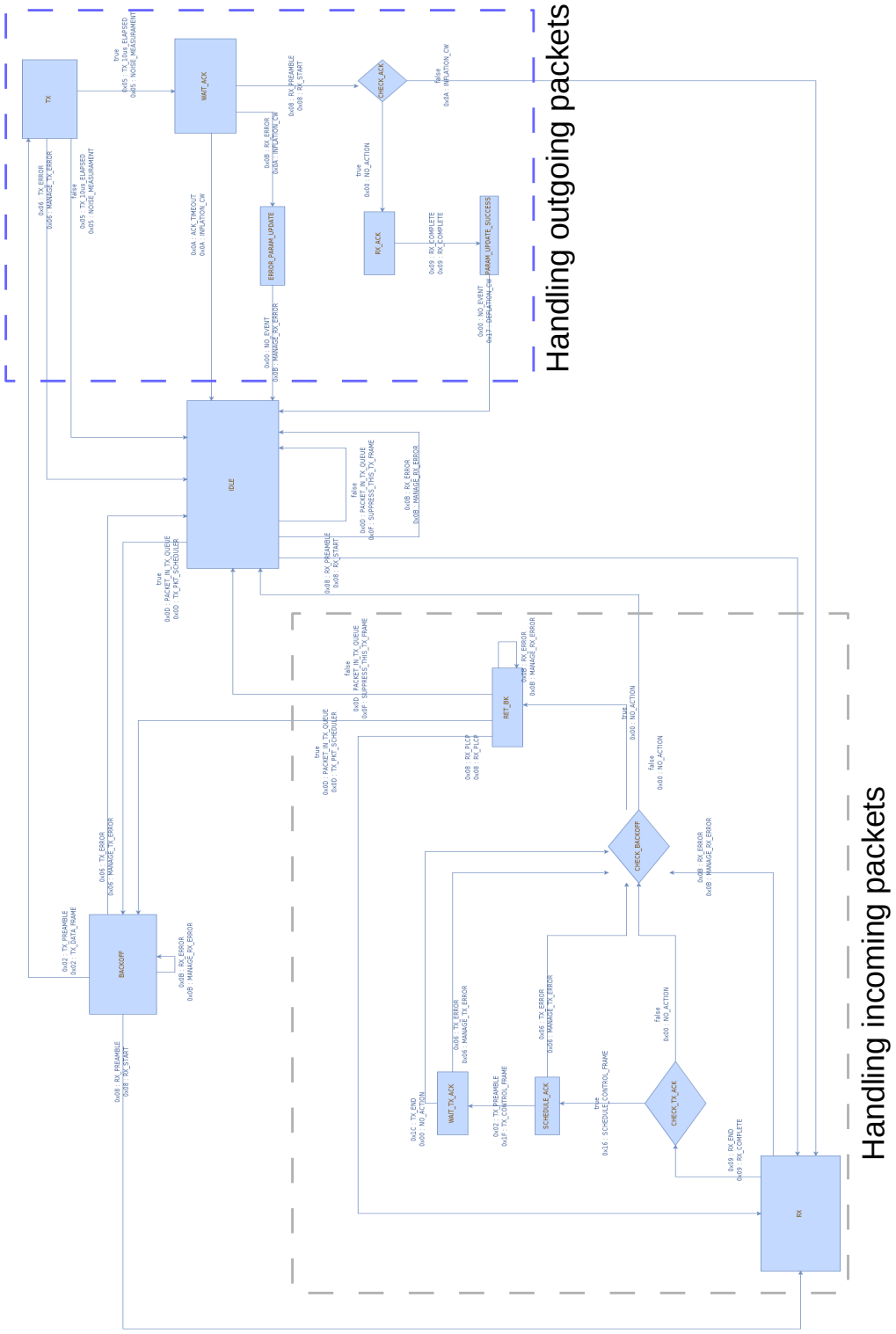



Figure 7.1: DCF

- PACKET_IN_TX_QUEUE - Start Transmit operation mode;
- RX_PREAMBLE - Start Receive operation mode;
- RX_ERROR - Manage error.

Event RX_ERROR is triggered by the receiver if an invalid packet is received, either because the receiver is not able to finish the reception of the current packet or because a corrupted PLCP is detected. The corresponding action MANAGE_RX_ERROR resets the receiver. It is worth noting that this event must be checked in all states that control reception. We describe Receive and Transmit operation modes respectively in two next paragraph.

**Reception operation mode** When in state RX, two events may be triggered:

- RX_END
- RX_ERROR

Event RX_END indicates that the incoming frame ended and moves the State Machine to conditional state CHECK_TX_ACK that checks condition NEED_SEND_ACK to determine if the received frame is to be acknowledged or not.

**Don't ack frame** When the received frame does not need ACK reply, the state machine evolves to conditional state CHECK_BACKOFF that checks the backoff value: if backoff is null, then state machine moves to IDLE state, otherwise to conditional state CHECK_BACKOFF.

**Do ack frame** When the received frame needs ACK reply, the state machine moves to state SCHEDULE_ACK, after execution the action START_IFS_CONTROL_FRAME with the ACK parameter selection, where it will wait for two events:

- TX_PREAMBLE, transition executes action TX_CONTROL_FRAME and moves the state machine to state WAIT_TX_ACK;
- TX_ERROR, transition manages transmission errors by executing action MANAGE_TX_ERROR and moves the state machine to state CHECK_BACKOFF.

For the sake of clarity event TX_ERROR should be checked by all states that manage frame transmission: in the following we will not mention that event explicitly anymore.

When in state WAIT_TX_ACK, the state machine waits event TX_COMPLETE, after transmission and evolves to state CHECK_BACKOFF that is used to verify if a backoff contdown timer was activated.

When conditional state CHECK_BACKOFF is reached, state machine checks condition BK_VAL != 0: if true evolves to state RET_BK without executing any action, otherwise state machine returns to state IDLE. State RET_BK waits for two events:

- RX_PREAMBLE, transition executes action RX_START and goes to state RX
- PACKET_IN_TX_QUEUE, transition leads the MAC-Engine to check condition TX_PACKET == GOOD: if verified then state machine evolves to state BACKOFF after executing action START_IFS_DATA_FRAME. If instead the condition is not verified, it evolves to state IDLE after executing action SUPPRESS_THIS_TX_FRAME.

**Transmit operation mode** state machine switches to this operation mode from state IDLE when it detects event PACKET_IN_TX_QUEUE. In this case the MAC-Engine checks condition TX_PACKET == GOOD: if verified then state machine evolves to state BACKOFF after executing action START_IFS_DATA_FRAME. If instead the condition is not verified, it evolves to state IDLE after executing action SUPPRESS_THIS_TX_FRAME.

State BACKOFF is characterized by four outgoing transitions, that are selected whenever one of the following events verify:

- RX_PREAMBLE - triggers evolution to state RX for handling frame arrival during backoff;
- TX_PREAMBLE - executes action TX_DATA_FRAME and goes to state TX;
- TX_ERROR - to manage transmission errors and goes to state IDLE;
- RX_ERROR - to manage errors in the receiver during backoff, selfloop.

When in state TX, two transition are associated with the event TX_10us_ELAPSED, the state machine waits event TX_10us_ELAPSED triggers checking condition NEED_WAIT_ACK: if verified state machine evolves to state WAIT_ACK otherwise to state IDLE. In both cases action NOISE_MEASUREMENT is executed.

State WAIT_ACK checks three events:

- RX_PREAMBLE - if verified state machine executes action RX_START and evolves to conditional state CHECK_ACK; otherwise to state RX;
- ACK_TIMEOUT - this event is raised when the timeout set for waiting the ACK reply expires, if verified then state machine executes action INFLATION_CW and state machine evolves to state IDLE;
- RX_ERROR - this event is raised if some error occurs during the reception of the ACK, if this event is verified the state machine executes action INFLATION_CW and evolves to state ERROR_PARAM_UPDATE.

Conditional state CHECK_ACK checks RX_PACKET == ACK: if true state machine evolves without actions to state RX_ACK otherwise goes to state RX.

State RX_ACK waits for two events:

- RX_END- that eventually execute action RX_COMPLETE and evolves state machine to state PARAM_UPDATE_SUCCESS;
- RX_ERROR - this event is raised if some error occurs during the reception of the ACK, if this event is verified the state machine executes action INFLATION_CW and evolves to state ERROR_PARAM_UPDATE.

In state PARAM_UPDATE_SUCCESS state machine checks no events, it executes action DEFLATION_CW and goes to state REPORT_TX_STATUS_TO_HOST. In last state REPORT_TX_STATUS_TO_HOST state machine checks no events, it executes action REPORT_TX_STATUS_TO_HOST and returns to state IDLE.

### 7.2.1 DCF programmability

The DCF state machine analyzed in the section 7.2 use standard state machine parameters, from it we can use different value for the parameter PARAM_BACKOFF, action START_IFS_DATA_FRAME uses it. By playing with this parameter it is possible to change the 802.11 backoff rule by setting specific values as follows, this is obtain through the selction of correct item in the menu parameters below the field action selections:

- DEFAULT mode: STD, i.e. standard exponential backoff rule;
- NO BACKOFF TRANSMISSION mode: NO_IFS, in this case transmit a packet without doing backoff;
- FIXED BACKOFF VALUE mode: select a possible value to used as backoff.
- SIFS mode: in this case transmit a packet after a SIFS time.

### 7.2.2 DCF with Access Point feature

Figure 7.2 shows the AP implementation MAC. This is a straightforward modification of the DCF State Machine: a new transition from state IDLE is triggered by event BEACON_TIMER_TIMEOUT which evolves the state machine to state SCHEDULE_BEACON after that the state machine schedule the beacon transmission.

Here state machine wait the event TX_PREAMBLE and evolves to state TX with the action TX_CONTROL_FRAME, the parameter of the last action is TX_BEACON.

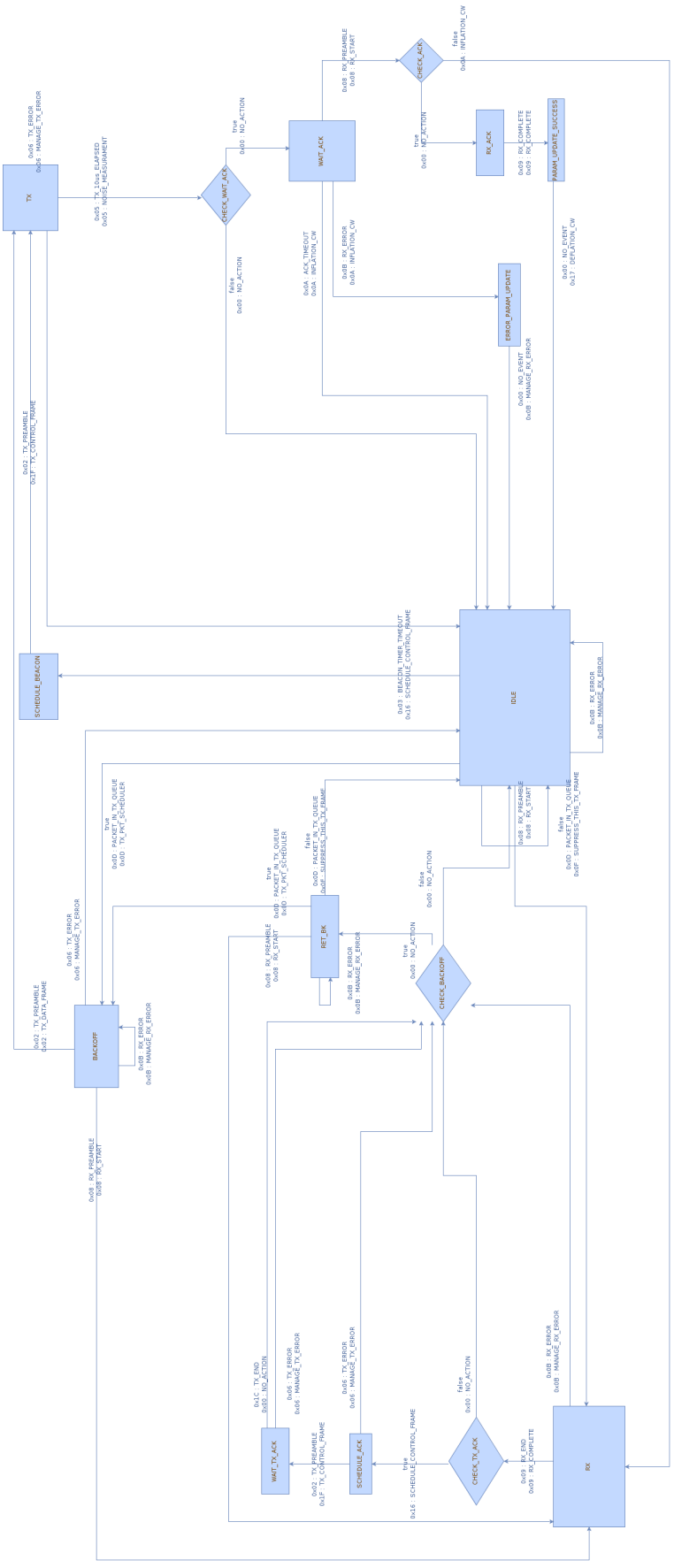

TX
WAIT_ACK
CHECK_ACK
RX_ACK
PARAM_UPDATE_SUCCESS
CHECK_WAIT_ACK
ERROR_PARAM_UPDATE
IDLE
SCHEDULE_BEACON
BACKOFF
RET_BK
CHECK_BACKOFF
WAIT_TX_ACK
SCHEDULE_ACK
CHECK_TX_ACK
RX

Figure 7.2: AP

### 7.2.3 DCF Experimental Results

Our performance tests, performed for all the supported PHY rates (up to 54 Mbps) did not show any noticeable difference with respect to the Broadcom firmware with legacy DCF.

Figure 7.3 shows an experiment in which DCF state machine and legacy DCF are compared. In this experiment we used an 802.11g PHY, with the data rate set to 36 Mbps. We considered a state machine DCF in which we select a standard contention rule, and a DCF legacy, both program working on channel 6. For quantifying the performance of the scheme, we compared the throughput perceived by a station DCF state machine (red star) or the DCF legacy program (blue star), when they contend together and when they use the channel exclusively. The experimental measurements allow to draw some interesting observations. When the stations use the channel exclusively, in the time intervals [0,21]s for station with legacy DCF, and [22, 42]s for station with DCF state machine **there is no remarkable performance difference between the legacy DCF scheme and DCF state machine scheme**, thus proving that the WMP architecture is almost negligible, the WMP does not involve delay or loss of performance. Second, also when the stations work together, in the time interval [45, 65]s, the performance of the different stations are the same, by maintaining almost a fixed 10 Mbps throughput. In this case the WMP architecture does not involve difference in the mechanism of the medium access and contention with the other stations.

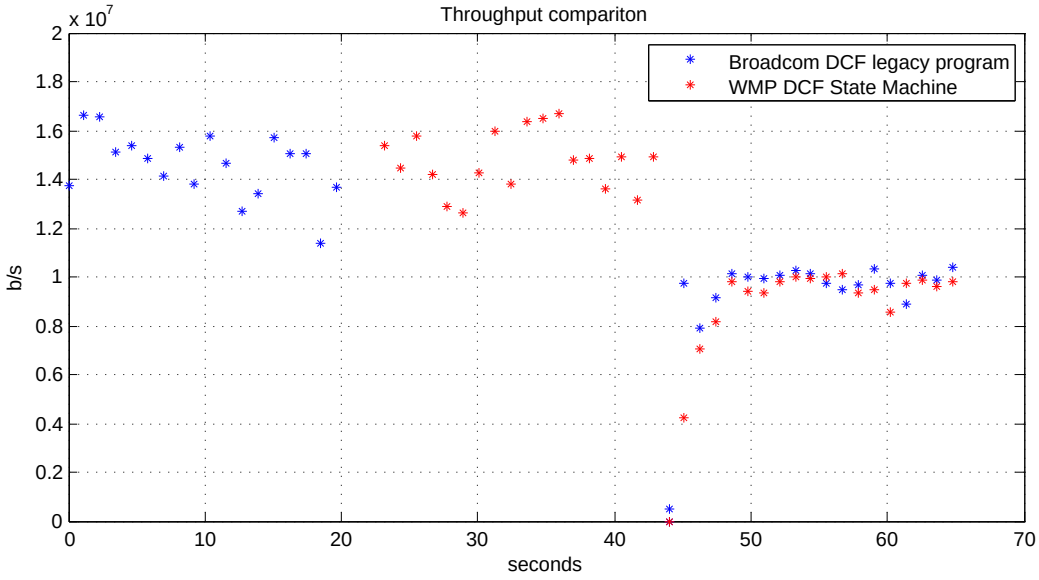



Figure 7.3

## 7.3 Time Division Multiple Access (TDMA)

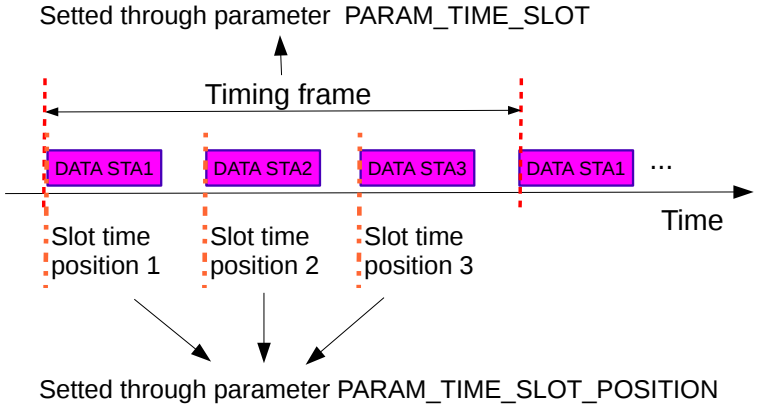



Figure 7.4: Timing setting in TDMA state machine

In order to prove that our platform can provide a precise scheduling of medium access times, we considered some simple DCF extensions devised to support a TDMA protocol mechanism. This is a simple TDMA access, through the state parameters is possible set the frame timing and the slot position for each stations present in the network [5]. A graphical representation of the frame organization is given in figure 7.4, where for space reasons the ACK frames are not explicitly shown and have to be considered included in the transmission boxes. The transmission is immediately performed the timer expires, as display in the figure 7.4 we use the parameter `PARAM_TIME_SLOT` to set the frame time, and the parameter `PARAM_TIME_SLOT_POSITION` to set the slot position time.

Figure 7.5 shows the TDMA implementation. This is a straightforward modification of the DCF State Machine: a new transition from state `IDLE` is triggered by event `TX_SLOTTED` which eventually evolves the state machine to conditional state `CHECK_PACKET_QUEUE`. Here if condition `PACKET_IN_TX_QUEUE` is true, state machine evolves to conditional state `CHECK_TX_PACKET_GOOD` otherwise returns to state `IDLE`. In conditional state `CHECK_TX_PACKET_GOOD` condition `TX_PACKET == GOOD` is checked: if true state machine executes action `START_IFS_DATA_FRAME` and goes to state `BACKOFF` otherwise it executes action `SUPPRESS_THIS_TX_FRAME` and returns to state `IDLE`.

**State Parameter in TDMA** The main parameters are `PARAM_TIME_SLOT` and `PARAM_TIME_SLOT_POSITION`. The former is set to `0` and it mean that the slot position fot this station is the first, the latter is the value in microseconds of the time slot for TDMA according with the frame duration and the number of stations in the network, in the example the time duration for the frame is 10 milliseconds. Figure 7.6 shows a snapshot of configuration parameters for TDMA state machine: On more is need set a SIFS parameter in menu parameter to the action `START_IFS_DATA_FRAME`.

### 7.3.1 TDM Experimental Results

Figure 7.7 shows an example of channel activity trace acquired by the USRP during an experiment, the figure shows two stations executing the TDMA state machine program, and they send packets of different size (namely, 500-byte packets from the first station STA1, and 100-byte packets from the second STA2) towards a common destination station (STA0). Both the stations employ an 802.11b PHY with a transmission rate set to 11 Mbps and a pseudo-frame interval of 1.536 ms. In the figure, the different frame lengths allow to distinguish the two transmitters, while the different power levels allow identifying acknowledgment transmissions and idle times. This experimental prove that the medium access times are scheduled by WMP with a precision (of the order of microseconds) not achievable with driver level hackings.

## 7.4 Direct Link

Direct Link (DL) is a variation of DCF that allows two stations to establish a direct connection bypassing the Access Point for interstation frames. DL is implemented in two variants: **Direct Link Setup (DLS)** and **Direct Channel Link Setup (DCLS)** .

The DL solution is obviously not nearly new, and indeed was specifically addressed by the 802.11e task group with the introduction of the Direct Link Setup (DLS), further extended in the 802.11z-2010 amendment. However, a direct link setup is not real implemented. Moreover, the direct link uses the same wireless channel, thus, although to a lower extent, the station connected to the Internet still suffers of a bandwidth reduction.

By default, their WMP card runs a MAC program implementing just the legacy 802.11 DCF operation. But we can load and activate different state machine that perform a DLS scheme. DLS transmits direct frames using the same radio channel of the AP, in this case there are no synchronization problem because beacons are received as usual. Moreover, to push bandwidth optimization further, by setting the direct link on a separate frequency channel, in this case we can load and activate the DCLS scheme. The DCLS is meant to be a simple variant of DLS able to simultaneously work on two different channels. The primary channel is that of the AP network; the station has to periodically access such channel for receiving beacons and retaining association. The secondary channel is independently managed by the peer stations. Under DCLS, the channel selection and the associated channel access mode is performed frame by frame. If the head of line frame is directed to the peer station, the frame is sent on the secondary channel as it was sent by the AP (i.e. with the from DS bit set to 1, and with the sender address of the AP). Definitely both variants of DL modify MAC header to transmit the frame addressed to the target station.

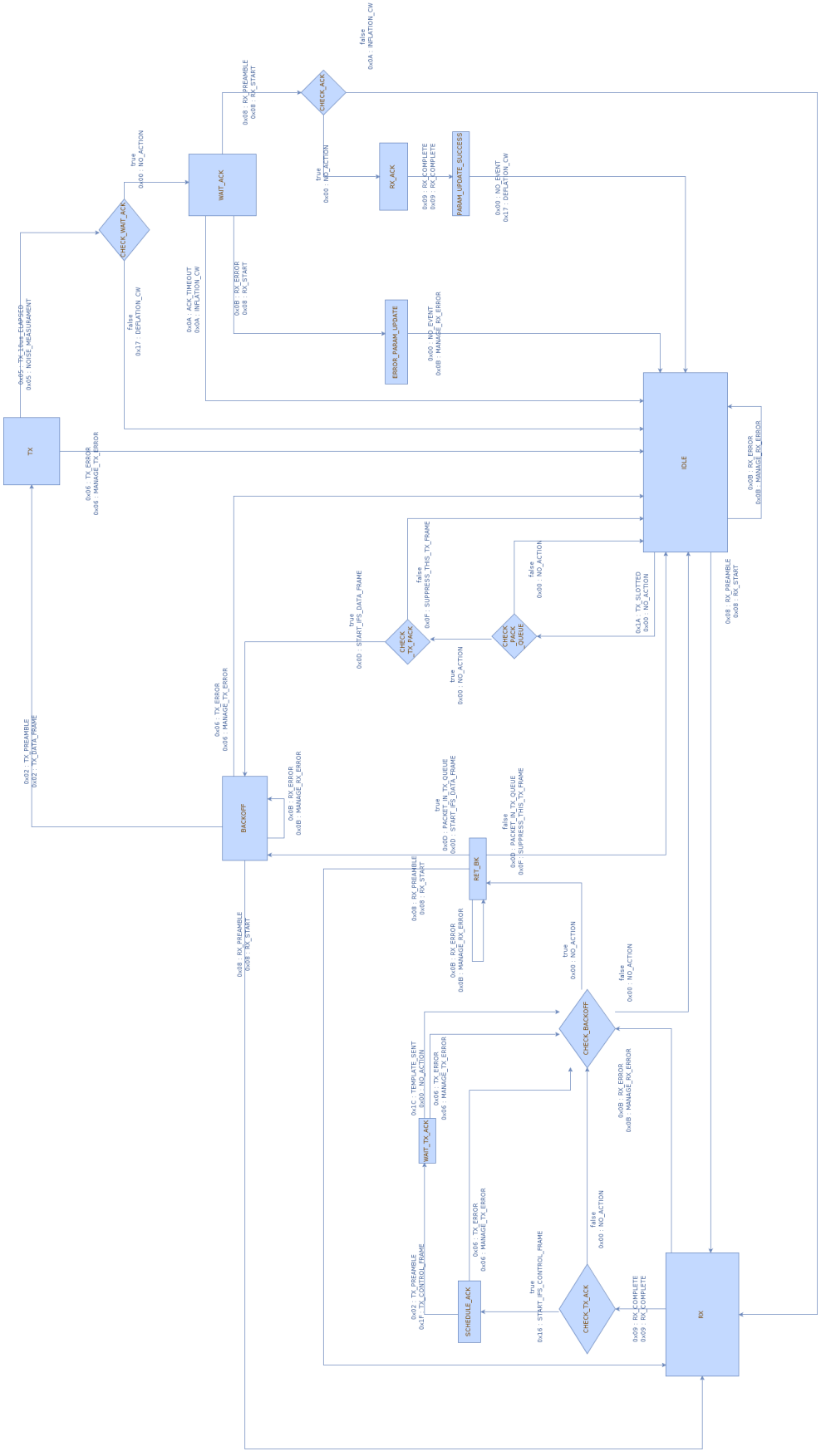



Figure 7.5: TDMA state machine

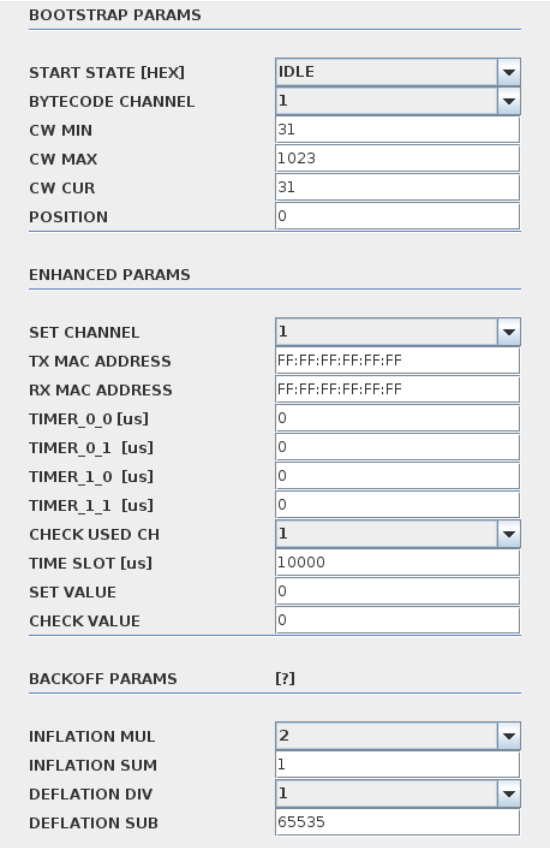



Figure 7.6: TDMA State Parameters

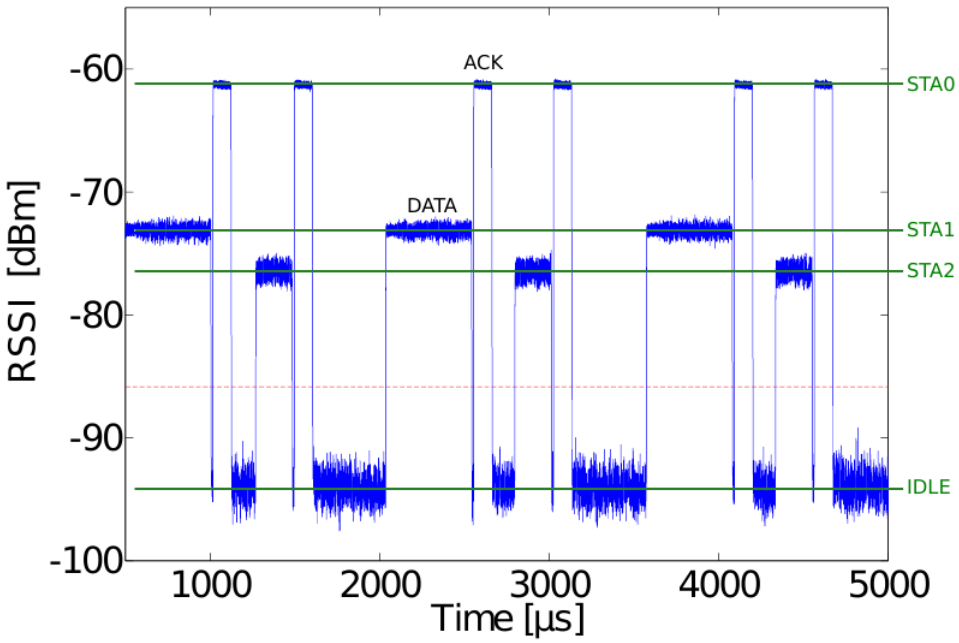



Figure 7.7: An experimental trace of medium access times under TDMA

Since this is a demo state machine, only one direct target is allowed and, in fact, it is encoded in the Byte-Code settings as a target MAC address (of the direct link station). When a DLS station sends a frame to its direct link target, the MAC-Engine changes the MAC header of the frame.

**DLS** shown in Figure 7.8, Direct Link Setup needs two additional conditional states respect to standard DCF, one before state `BACKOFF` and another one before state `RX`. First change is near state `BACKOFF` that is reached from states `IDLE` or `RET_BK`: in DLS state machine, before reaching state `BACKOFF` a new state called `CHECK_TX_ADDR` must be visited. In this state state machine checks condition `TX_DST_ADDR == PARAM_TX_DST_ADDR_0`: in both cases state machine evolves to state `BACKOFF` but if condition is true MAC-Engine executes action `SET_TX_MAC_ADDRESS`.

Second change is in state `RX` that is reached from states `IDLE`, `BACKOFF` and `WAIT_ACK`. DLS state machine implementation needs an additional state called `CHECK_RX_ADDR`. Here the state machine checks condition `RX_SRC_ADDR == PARAM_RX_SRC_ADDR_0`: in both cases state machine evolves to state `RX` but if condition is true MAC-Engine executes action `SET_RX_MAC_ADDRESS`.

**DCLS** transmits direct frames on a separate radio channel. This improves the throughput performance, even if a channel hopping mechanism is needed to periodically switch back to the AP radio channel for receiving the beacons and avoid synchronization issues. The DCLS scheme requires that the peer stations use standard DCF rules on the primary channel and DL access rules on the secondary one. This behavior can be programmed by defining a DCLS meta machine switching from DCF to DCLS and vice versa. Figures 7.9 shows the envisioned relevant timings. The direct link operations are executed after the reception of an AP beacon. At regular time intervals (slightly lower than a multiple N of $T_{BTT}$ beacon intervals), the station suspends DCLS transmissions in order to receive an AP beacon on the primary channel. This operation is necessary for keeping the synchronization to the AP clock. A transitions to DCF can also occur when the head of line frame is a probe request or another frame directed to the AP. At the expiration of another timer $T_{BSS}$, the station switches the channel and back to DCLS. In order to minimize the frame losses due to the use of the two channels, the peer stations should activate the DCLS protocol simultaneously. To this purpose, we used a synchronization mechanism based on the specification of an absolute time. The activation time is computed by adding the desired time offset to the current Access Point time-stamp, to which all the stations are continuously aligned.

As shown in Figure 7.10 the DCLS implementation slightly differs from that of the DLS. Stations in a BSS, in fact, synchronize their internal clock with that of the AP using the timestamp inside each received beacon. For this reason it is necessary to periodically switch the channel of the station back to that of the AP. New states provides periodical or per-event channel hopping. In RX section we introduce a block that checks condition `RX_PACKET == MY_BEACON` to verify if the frame being received is a BSS beacon: if true, two actions are executed: i) one to activate a "switch back" countdown; and ii) `SET_CHANNEL` to set up the Direct Channel.

### 7.4.1 State parameter in D(C)LS

Parameters used in D(C)LS are (see also Figures 7.11(b) and 7.11(b)):

- `PARAM_TX_DST_ADDR`: it is used by condition `TX_DST_ADDR == PARAM_TX_DST_ADDR_0`;
- `PARAM_RX_SRC_ADDR`: it is used by condition `RX_SRC_ADDR == PARAM_RX_SRC_ADDR_0`;
- `PARAM_SET_TIMER_0`: takes value `TIMER_0_0` as the Direct Link period expressed in microseconds;
- `PARAM_SET_CHANNEL`: used by action `SET_CHANNEL` to define D(C)LS channel.

### 7.4.2 DLS and DCLS performance evaluation

To prove as the improve the performance of the wireless network when we use the DL scheme, we set-up a testbed in our laboratory with two client stations (the ones with the peer-to-peer traffic) and the AP. A third client was statically set to the Access Point (AP) channel with a legacy DCF protocol. We repeated the loading and activation test periodically, by programming the alternatively the DL Binary-Byte-Code and the legacy DCF Binary-Byte-Code at regular intervals of 1 minute. The DL Byte-Code is built by programming the meta machine described above, while the legacy DCF Byte-Code is the bios program. Refer to the figure 7.9, the DCLS protocol has been configured

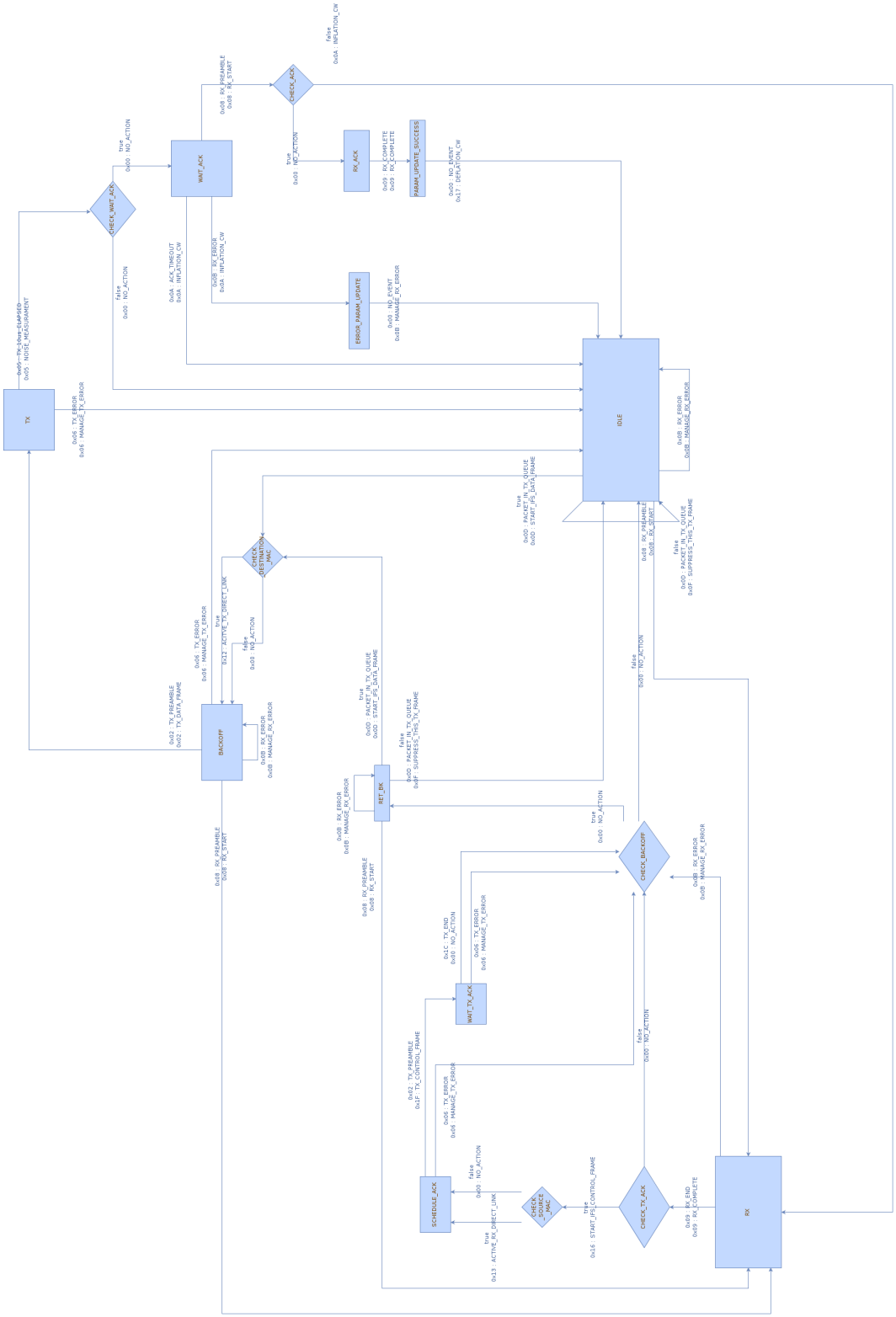



Figure 7.8: Direct Link Setup (DLS)

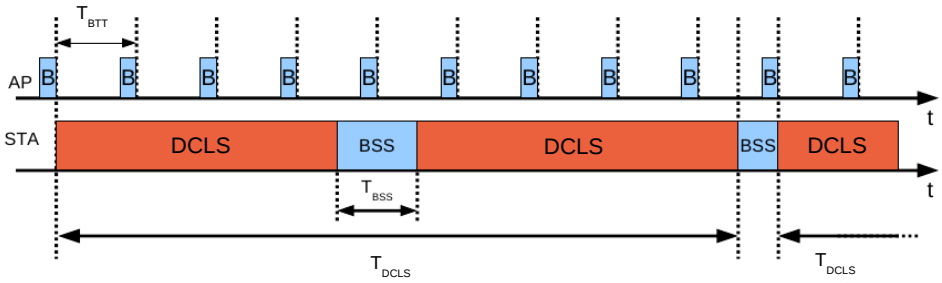



Figure 7.9: DCLS

by setting $T_{DCLS}$ to $890ms$, $T_{DCF}$ to $6ms$, and came back to the AP channel every 9 $T_{BTT}$. The DLS machine is derived from the DCF one by changing the addressing operations for both data frames and acknowledgments. Moreover, it can be configured with independent contention window values, thus allowing to support more aggressive access operations. Then we consider three different scheme for the testbed:

1. **DLS** in which we have a unique channel (channel 6);
2. **DLS-CH** a DCLS with the primary channel set to channel 6 (AP channel) and the secondary channel set to channel 11;
3. **DLS-CH-NO-BK** a DCLS with the primary channel set to channel 6 (AP channel), the secondary channel set to channel 11, and the secondary channel contention window set to 0.

Figure 7.12 shows the throughput results of the client station sending saturated UDP traffic to the second client under the three settings (labeled, respectively, as DLS, DLS-CH, DLS-CH-NO-BK). The experiments were carried out during the hours of the day (i.e. in presence of background traffic due to students and researchers working in our department). Starting from legacy DCF, the clients switch to the different DL configurations at 1, 3, and 5 minutes, and come back to standard DCF at 2, 4 and 6 minutes. From the figure it is evident that the customized DL access may bring dramatic improvements, especially when it is managed on a secondary channel without backoff (from about 12 Mbps of the normal DLS case to about 38 Mbps under the DCLS without backoff).

## 7.5 Power consumption evaluation

Other important element in wireless network is the power consumption, indeed energy efficiency of communication networks is becoming an increasingly important topic, especially with the expected explosion of traffic to be carried by those networks. In order to validate the proposed approach, we also compare the WMP performance with the benchmark provided by the native Broadcom's firmware in terms of power consumption. For this features we work together with NITlab group, the NITlab proposes NITOS, a Energy consumption Monitoring Framework (EMF) able to support online monitoring of energy expenditure, along with the experiment execution [21]. We run some tests to measure the power consumption of the different MAC versions discussed in our previous section. The experimental scenario consists of 2 NITOS Icarus nodes, attached with the BCM4311 m-PCIe chipset. We generate saturated traffic for 15 secs and measure the power consumption at the NIC level for the transmitter node. Power consumption is monitored for 25 secs, just to make sure that all frame transmissions are properly captured.

More specifically, we compared the power consumption of 5 different MAC versions:

- Original Firmware
- DCF-master WMP state machine
- TDMA WMP state machine with time frame $10ms$
- TDMA WMP state machine with time frame $0.1ms$

In the figure 7.13 we plot the average power consumption of each scheme. We confirm that DCF state machine implementation (DCF-Master - 0.9417 W) consumes even less amount of Power, compared to the standard firmware implementation (Original Firmware - 1.0127 W).

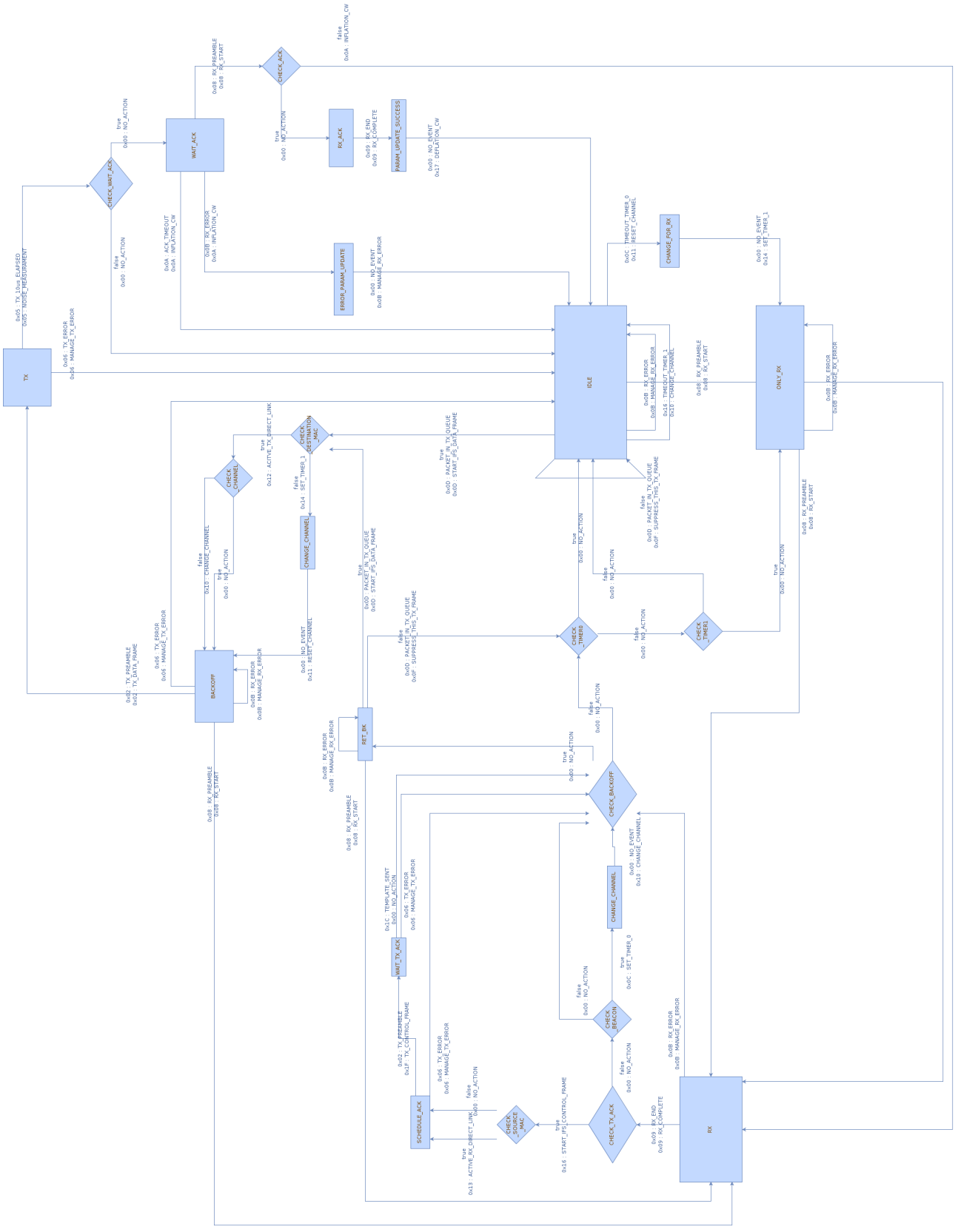
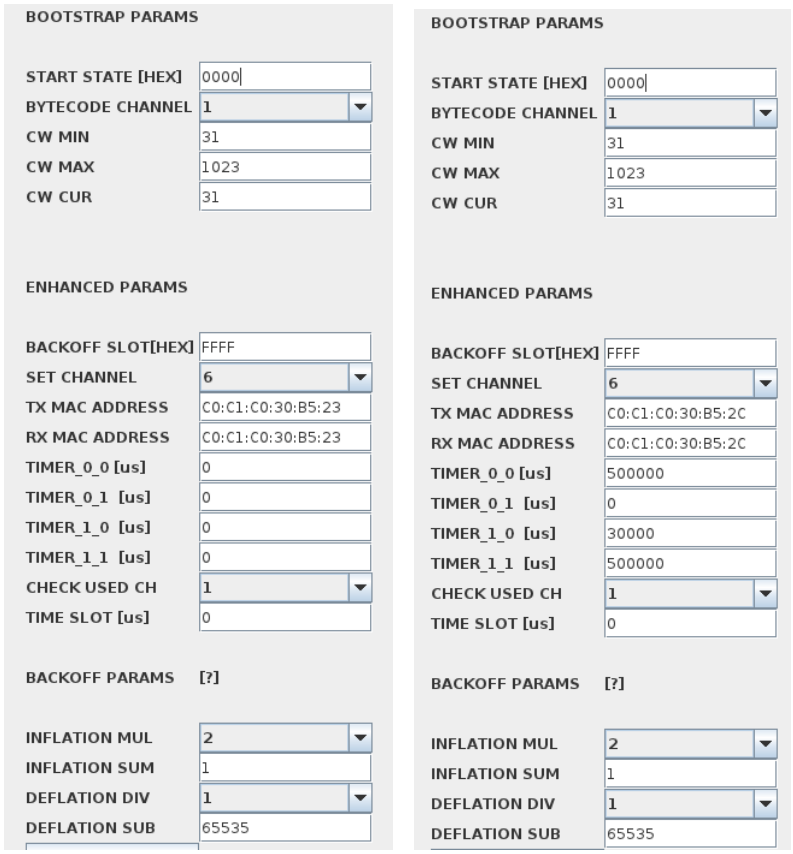


Figure 7.10: Direct Channel Link Setup (DCLS)




(a) (b)

Figure 7.11: DLS parameters 7.11(a), DCLS parameters 7.11(b)

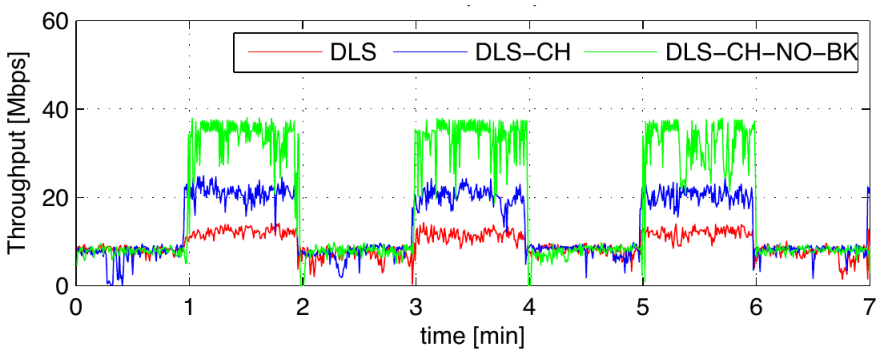



Figure 7.12: DL setup compairson

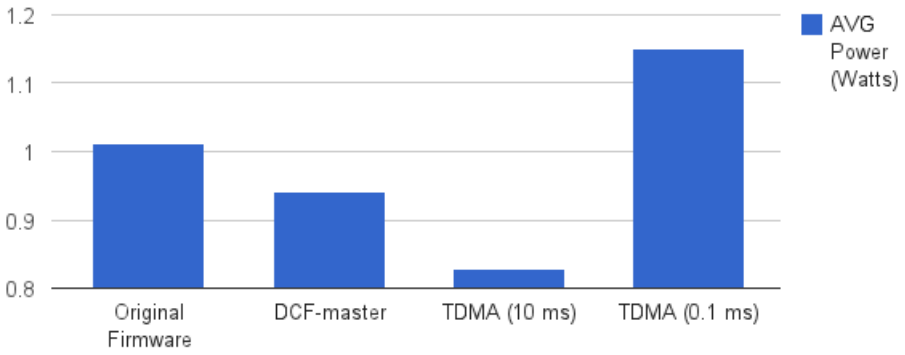



Figure 7.13: DL setup compairson

# Chapter 8

# High level flexibility

## 8.1 Introduction

The 802.11 standard specifies a common medium access control (MAC) Layer, which provides a variety of functions that support the operation of 802.11-based wireless LANs. In general, the MAC Layer manages and maintains communications between 802.11 stations (radio network cards and access points) by coordinating access to a shared radio channel and utilizing protocols that enhance communications over a wireless medium. The basis function performed from the MAC layer is the **Medium access**, defined the technique to shared the medium. We have shown the basis function in the chapter 1, and in other previous chapters we present the motivation to required flexibility in the MAC layer medium access function, then we showed the own solution based on WMP platform [61]. Other functions performed by MAC layer are:

- **Scanning**: In this function, un-associated STAs (e.g., STAs just turned their power on) scan the medium and listen to the beacons of the available APs (passive scanning). The 802.11 standard defines both passive and active scanning; Passive scanning is mandatory where each NIC scans individual channels to find the best access point signal. Periodically, access points broadcast a beacon, and the radio NIC receives these beacons while scanning and takes note of the corresponding signal strengths. The beacons contain information about the access point, including service set identifier (SSID), supported data rates, etc. The radio NIC can use this information along with the signal strength to compare access points and decide upon which one to use. Optional active scanning is similar, except the radio NIC initiates the process by broadcasting a probe frame, and all access points within range respond with a probe response. Active scanning enables a radio NIC to receive immediate response from access points, without waiting for a beacon transmission. The issue, however, is that active scanning imposes additional overhead on the network because of the transmission of probe and corresponding response frames.

- **Authentication**: Authentication is the process of proving identity, and the 802.11 standard specifies two forms: Open system authentication and shared key authentication. We report a detail of this process in the section 8.3.

- **Association**: Once authenticated, the radio NIC must associate with the access point before sending data frames. Association is necessary to synchronize the radio NIC and access point with important information, such as supported data rates. The radio NIC initiates the association by sending an association request frame containing elements such as SSID and supported data rates. The access point responds by sending an association response frame containing an association ID along with other information regarding the access point. Once the radio NIC and access point complete the association process, they can send data frames to each other.

- **RTS/CTS**: The optional request-to send and clear-to-send (RTS/CTS) function allows the access point to control use of the medium for stations activating RTS/CTS. The use of RTS/CTS alleviates hidden node problems, that is, where two or more radio NICs can't hear each other and they are associated with the same access point. If the radio NIC activates RTS/CTS, it will first send a RTS frame to access point before sending a data frame. The access point will then respond with a CTS frame, indicating that the radio NIC can send the data frame. With the CTS frame, the access point will provide a value in the duration field of the frame header

that holds off other stations from transmitting until after the radio NIC initiating the RTS can send its data frame. This avoids collisions between hidden nodes.

- **Power Save Mode**: The optional power save mode that a user can turn on or off enables the radio NIC to conserve battery power when there is no need to send data. With power save mode on, the radio NIC indicates its desire to enter "sleep" state to the access point via a status bit located in the header of each frame. The access point takes note of each radio NIC wishing to enter power save mode, and buffers packets corresponding to the sleeping station. In order to still receive data frames, the sleeping NIC must wake up periodically (at the right time) to receive regular beacon transmissions coming from the access point. These beacons identify whether sleeping stations have frames buffered at the access point and waiting for delivery to their respective destinations. The radio NICs having awaiting frames will request them from the access point. After receiving the frames, the radio NIC can go back to sleep.

The MAC layer together with the Logical Link Control (LLC) composed the Data-Link layer, this layer initial connection has been set up, divides output data into data frames, and handles the acknowledgements from a receiver that the data arrived successfully.

The layer above the Data-Link layer is the network layer, it is responsible for packet forwarding including routing through intermediate routers, whereas the data link layer is responsible for media access control, flow control and error checking. The network layer provides the functional and procedural means of transferring variable-length data sequences from a source to a destination host via one or more networks, while maintaining the quality of service functions. Functions of the network layer include the connection model, the host addressing and the message forwarding. Host addressing is important function, because every host in the network must have a unique address that determines where it is. This address is normally assigned from a hierarchical system. On the internet, addresses are known as Internet Protocol (IP) addresses, because managed from the IP protocol. The IP is responsible for addressing hosts and for routing datagrams (packets) from a source host to a destination host across one or more IP networks. A precise protocol is been developed to assign automatically the IP address, it name is Dynamic Host Configuration Protocol (DHCP) and allows a computer to join an IP-based network without having a pre-configured IP address. DHCP is a protocol that assigns unique IP addresses to devices, then releases and renews these addresses as devices leave and re-join the network.

Also, in the other functions of the Data-Link layer and in the network layer we can add flexibility to improve the feature for the network nodes. This could lead to a quick and clever adaptation of the nodes, when the environment changing dynamically.

In the traditional network, protocol stacks are logically organized in layers. These layers are strictly separated, and the cooperation between them is restricted by concise interfaces, which the application work only a single layer. In principle, all these layers have been designed to fulfill their functionality without interaction across the layers. History shows that this works well in wired and static environments. However, today's and upcoming networks consist of wireless links and highly mobile nodes. In order to adapt to the rapidly and frequently changing network conditions under those circumstances, a more sophisticated interaction between protocols than in a traditional layered architecture is desirable.

For example, the vehicular network are a system in which the nodes work in environment that changing mostly rapidly, in this network access can be obtain roadside installed Internet Gateways(Access Point) or from other vehicles. However, several difficulties must be addressed in such a scenario, the speed of the vehicles reduce the time to exchange the informations. **For this reason is needed that all operation for connection set-up are really fast**. In normal WLANs system the connection set-up phases are handled by different application and different protocol level, the set-up connection process is not optimized, because several actors work not synergically for the process. In generally, all these programs are developed for a specific part of the process, every level not interact with other layers.

If we consider the network vehicular example above, current solutions are not able to dynamically change of the environment i.e., adaptation of connection set-up phase at different environment during runtime. Moreover, an optimizations is possible through a cross-layer frameworks that manage synergically the connection set-up at different levels.

We propose new dynamic association and reassociation procedure that manage the association for both layers, data-link layer and network layer, the own solution is based on a monitoring status of the presence of AP in the environment. We present **wpa_fast** , a unique program that handled the different phase of set-up connection and managed open and security system.

**wpa_fast** is a system to optimize the connection set up time in vehicular network, reduce the connection set-up time, and improve the time that the node use for exchange the information, in this chapter, we characterize the WiFi connection set-up process and we are focusing on technique to improve the connection set-up. After, we show the results of a measurement evaluation of **wpa_fast** in different environments.

For this framework we work carried out with the Network Research Lab (NRL) at UCLA Computer Science Department. The NRL supports research projects in a broad range of topics in network communications including, car-to-car networks, VANETs and vehicular network.

## 8.2 Vehicular network

Vehicular network is a new challenging network environment that pursues the concept of ubiquitous computing for future. Vehicles equipped with wireless communication technologies and acting like computers will be on our roads soon and this will revolutionize our concept of travelling. Vehicular network bring lot of possibilities for new range of applications which will not only make our travel safer but fun as well. Reaching to a destination or getting help will be much easier. The concept of vehicular network is quite simple: by incorporating the wireless communication and data sharing capabilities, the vehicles can be turned into a network providing similar services to the ones we are used to in our office or home networks.

For the wide spread and ubiquitous use of vehicular network, a number of technical challenges exist. Several academic and industrial projects were initiated to address these challenges. One of the earliest European projects was FleetNet (Sep 2000 - Dec 2003) [48]. Its objectives were to develop a platform for inter-vehicular communication, implement demonstrator applications and then to standardize the solutions. Some other prominent projects include Network on Wheels (NoW)[49] and CarTALK2000 [50]. Car-to-Car Communication Consortium (C2C-CC) [51] is an umbrella organization overseeing VANET research activities in Europe. It includes many automobile industry members like Daimler, BMW, Audi, Fiat, Renault and some German universities. For the purpose of illustration, vehicular network applications may be divided into following categories.

- **Safety Applications** Road safety applications can play an important role in avoiding accidents or at least minimizing the impact of accidents, if accident is unavoidable. An early warning system can alert the driver about the road scenario, e.g.,an accident, thus giving the driver enough time to apply brakes well before hitting the accident place.
- **Traffic Management** Another application for vehicular network is to tackle road congestions and provide the best route to a driver with updated road conditions. This can involve the use of some road side equipment e.g., intelligent traffic signals, e-sign boards etc. Information about the road congestions ahead can definitely help in reducing the congestion and improving the capacity of roads.
- **User Applications** Besides road safety applications, information and entertainment applications are also envisioned for vehicular network. The passengers in a vehicle can enjoy the facility of Internet connectivity where other traditional wireless internet connectivity options (Wi-Fi, Wi-MAXetc.) are not available. Even in the presence of such options, a node connected to internet through these options, can share its connectivity with other vehicles through vehicular network.

## 8.3 WLANs Association Schema

In WLANs system to realize a data flow with access point, every station needed a connection set-up, afterwards, the transfer of data will be dependent on the mobility of the user, of the interference due to the radio channel characteristics and of the number of users in the WiFi coverage area. In this chapter, we show a technique to reduce the connection set-up time and develop a software to obtain this.
The connection set-up process consists of four phases:

- Network discovery
- Authentication
- Association

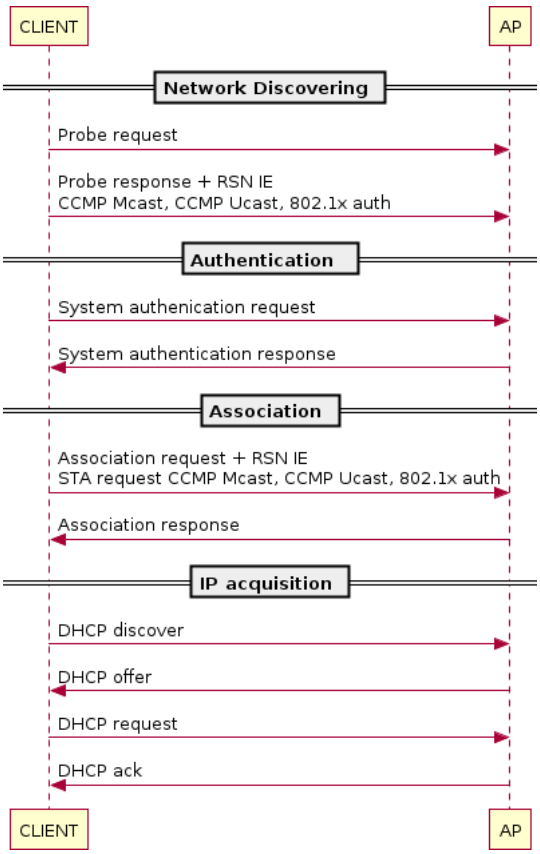



Figure 8.1: Connectin setup phases

- IP acquisition

all phases are showed in figure 8.1, we will discuss the different phases grouping them into two parts, they are: **MAC level functionality** and **Network level functionality**.

### 8.3.1 MAC level functionality

The first phase is **Network discovery**, in this phase, the stations can discover available access points either passively or actively. Passive discovery uses beacon frames broadcasted by the wireless access point (AP) at intervals of 100ms in generally.
We discuss the other two phases, **Authentication** and **Association** together, because, we consider open and protected netwrork system, indeed, **wpa_fast can managed both the solution**. In the case of open system, both, authentication and association phases are performed, but the procedure is very fast, because the association is not never deny. While, a security system uses different solution, the most recent standard are WPA and WPA2. The WPA2 standards were created by the Wi-Fi Alliance industry group that promotes interoperability and security for the wireless LAN industry. The Wi-Fi Alliance WPA and WPA2 standards closely mirrors the official IEEE 802.11i wireless LAN security standards group, but incorporates additional IETF EAP standards that the Wi-Fi Alliance considers secure.

The WPA2 standard has two components, encryption and authentication which are crucial to a secure wireless LAN. The encryption piece of WPA2 mandates the use of AES (Advanced Encryption Standard) but TKIP (Temporal Key Integrity Protocol) is available for backward compatibility with existing WAP hardware. The authentication piece of WPA2 has two modes: Personal and Enterprise. The Personal mode requires the use of a PSK (Pre-Shared Key) and does not require users to be separately authenticated. The Enterprise mode, which requires the users to be separately authenticated based on the IEEE 802.1X authentication standard, uses the Extended EAP (Extensible Authentication Protocol) which offers five EAP standards to choose from: EAP-Transport Layer Security (EAP-TLS), EAP-Tunneled Transport Layer Security (EAP-TTLS), Protected EAP vo/EAP Microsoft's Challenge Handshake Authentication Protocol v2 (PEAPvo/EAP-MSCHAPv2), Protected EAP v1/EAP-Generic Token Card (PEAPv1/EAP-GTC) and EAP-Subscriber Identity Module of the

Global System of Mobile Communications (EAP-SIM)[52]. The Enterprise mode has the following hardware/software implementation requirements:

- Selection of EAP types that will be supported on stations, APs (Access Point), and authentication servers.
- Selection and deployment of authentication servers typically RADIUS (Remote Authentication Dial In User Service) based authentication servers.
- WPA2 software upgrades for APs and clients.

**WPA2 establishes a secure communication context in four phases**. In the first phase the parties, AP and the client, will agree on the security policy (authentication method, protocol for unicast traffic, protocol for multicast traffic and pre-authentication method) to use that is supported by the AP and the client. In the second phase (applicable to Enterprise mode only) 802.1X authentication is initiated between the AP and the client using the preferred authentication method to generate an MK (common Master Key). In the third phase after a successful authentication, temporary keys (each key has limited lifetime) are created and regularly updated; the overall goal of this phase is key generation and exchange. In the fourth phase all the previously generated keys are used by the CCMP protocol to provide data confidentiality and integrity.

### 8.3.2 Network level functionality

The last phase of the connection set-up is handled by network level and is the same for all low level, this phase provides a **IP address**.

DHCP provides an automated method for dynamic client configuration. This process occurs through a series of steps, illustrated in figure 8.1. Start with a broadcasting a "DHCP Discover message", the station sends this message at all DHCP servers available on the network to provide an address, all the DHCP server present in the network respond with a "DHCP Offer message". The client device may receive multiple offers if multiple servers are on the network. "DHCP Request message" is sended form a client that choose one offer. Since the client is not authorized to use the offered address yet, the "DHCP Request message" is still a broadcast. The client accepts the first offer received unless another offer matches the last IP address that the client had. With a "DHCP Ack message", the DHCP server finalizes the process with an acknowledgment, or Ack, allowing the client device to start using the address. In rare cases, the server issues a Negative Acknowledgment, because it may have decided that the address is not available in the milliseconds that have passed since it offered the address. Once an IP address is acquired, the connection set-up is complete and the data transfer can commence. The DHCP standard provides different latency time to ensure that all DHCP server can send the offer, and if no response is received for the DHCP Request or DHCP Discover messages, the sender waits for a period of time determined using an exponential back-off algorithm with an initial values of $4(\pm 1)$ sec.

## 8.4 WPA_FAST to improve the time connection se-tup

The vehicular networks are a systems in which the nodes work in environments that changing mostly rapidly, a car that moves on the road entry and exit rapidly from the coverage of the AP, for this reason is needed that all operation for connection set-up are really fast. **wpa_fast is a unique program that handled the different phases of set-up connection and managed open and security system**. In current system, the connection set-up phases, are handled by different program and different protocol level, the **scanning** phase is handled by mac80211 module, the **authentication** and **association** phases are handled by wireless driver, and **IP acquisition** phase is handled by user level program like DHCPclient, and it is the same for all the interface.
**The set-up connection process, above detailed, is not optimize because several actors work for the process and not synergically**, in generally, all these programs are developed for a specific part of the process, without considering the interaction with other levels. For example, if at finish of a connection set-up process the station go out at the coverage of the AP, the wireless driver start a phase to disconnect the station, but not inform the upper level program DHCPclient that the IP address is not more valid. Individually, it must understand that the connection has been lost, the consequence is a delays in the next stage of address request, this feature plays an important role in vehicular networks environment, because the lost time to obtain a new address, result a reduced

time for the data flow.

**Instead, if all the elements work synergistically, and that is cooperate in sharing information, the station save more time between a current phase and the next phase**, the system result more reactive at the changing of the environment condition, and this is an important feature for vehicular network.

To develop **wpa_fast** we start from two famous program used in Linux system handled the different phase part of connection set-up, this program are **wpa_supplicant**, and **pump**. Pump is a DHCP client used in many Linux system and wpa_supplicant is the IEEE 802.1X/WPA component that is used in the client stations. wpa_supplicant is suitable program to realize the MAC connection for both desktop/laptop computers and embedded systems. It implements key negotiation with a WPA Authenticator and it controls IEEE 802.11 authentication/association of the wireless driver. wpa_supplicant is designed to be a "daemon" program that runs in the background and acts as the backend component controlling the wireless connection. wpa_supplicant supports a control interface for send command and set parameters [56].

We develop a unique program that handled all the phases in the set-up connection, we get the source code of wpa_supplicant and pump and put together in a unique program, **after we add several features that improve the set-up connection in order to consider vehicular network environment**. **wpa_fast** uses a flexible build configuration file that can be used to select the parameters to tuned the station in real time for different scenario or in function of the speed car. Also the file allows the configuration of credential for specific network with security system, **wpa_fast** is fully compatible with all the driver and wireless card, and not have limited of channels usage (2.4GHz and 5GHz channels). To interaction with the down level, **wpa_fast** uses the netlink (the nl80211 driver) to create interface for receiving and transmitting management frames realize scanning, authentication and association operations.

## 8.5 WPA_FAST implementation

To realize the phase of scanning, **wpa_fast** sends scanning command through the netlink with specific parameter, by default **wpa_fast makes a scanning on a single channel**, this is important, because the time for a scanning is proportional to number of channel in which we make the scanning(in generally 2.7 seconds for 32 channels). In vehicular network the scanning time is most important, if the time is over, when a car move the station receives the scanning result when the scanning is yet old. The choose channel to make the scanning is most important, in fact to obtaining an optimal scanning strategy for a moving car that is continually scanning for a usable AP, we seek a scanning strategy that chose the channel to scans with a different probability, the [53] shows that the distribution of AP channels is not uniform and biased toward i channels, but the channel distribution follow a precise probability. **wpa_fast** has the possibility of set the probability of each channel through the configuration file and in runtime, in this way, we have the possibility to tuned the behaviour in relation with the environment in which the car moves. *The average time for scanning phase have long 90ms*.
At the end of the scanning, **wpa_fast** obtains a list of AP information, **wpa_fast** checks in the configuration file if one o more AP is present, and perform an authentication/association phase, if the AP is konw, **wpa_fast** reads credential information in the configuration file. Otherwise **wpa_fast creates a list of the open system AP present, and use it for start a cycle of attempts connection**, by default **wpa_fast** makes 2 cycle of attempts connection, and after, it handles a new scannig phase, after a time of 10 seconds rerun a new scan. *The average time for authentication/association phase have long 90ms for the open system AP* and it can takes a long time when the station not receives a response. We use a time out value of 350ms to resend a control frame. All this parameter can be setted through configuration file e can be changed in run time to adapt the behaviour at the environment.
As result of monolithic form of **wpa_fast** , **the phase that performs the IP address acquisition is started immediately after the association**, so we don't have latency before the IP address request phase, **wpa_fast** send a DHCP discovery and after a DHCP offer send a DHCP request. This is the phase in which we spend more time, this depend of the server DHCP latency, the message exchanged are broadcast and the server is not reactive. By standard, the DHCP protocol provides a large time for consider different DHCP server in the network, but in wireless scenario when the network is busy, we can lost packets, the result is a long statistical time, for this reason we

modify the normal timing in DHCP phase in way to obtain a **retransmission of a DHCP frame management after 100ms**, we set a value of 5 number max of retransmission, and after restart a connection set-up with another AP. *The average time for IP addressing acquisition phase have long 280ms*
If we obtain an IP address **wpa_fast** checks if a internet access is present and it ends the set-up process, afterwards, **wpa_fast** check continually eventually connection lost. If we don't obtain an IP address, or a event of connection lost is triggered **wpa_fast** performs a new scan or carry on with a new authentication/association. **All stages are carried out in succession and without latencies between them**.

## 8.6 Testebed set-up

In this section we present the technical that we used to realize the experiment evaluation for **wpa_fast** , we use a vehicle equipped with a eeepc and gps to trace the position of the car, we perform several hours of experiment in different geographical area. The experiment were run in los angeles inside the UCLA(University of California Los Angeles) campus and near area to UCLA, the experiment run in the UNIPA(University of Palermo) campus in wich we associated with eduroam(education roaming) network. Eduroam is the secure, world-wide roaming access service developed for the international research and education community, eduroam allows students, researchers and staff from participating institutions to obtain Internet connectivity across campus and when visiting other participating institutions by simply opening their laptop because use the same credential in all site in which is present [59].

For the experiment we use a normally laptop eeepc with the follow characteristic:

- **Display** 10.1" LED Backlight WSVGA Screen
- **Processor** Intel Atom N2600 (Dual Core; 1.6GHz) Processor
- **Ram** DDR3, 1 x SO-DIMM, 2GB
- **GPS** Columbus V-800 USB GPS Receiver

we equip the eeepc with a wireless network card Atheros Communications AR928X (PCI-Express), we chose this card because support both the band channel 2.4GHz and 5GHz, and we install it with a IPX RP SMA pigtail for connect two external antenna that we put on the roof of the car, the name of antenna is HyperLink HG2458-7RDR-NM [58], this is a rubber-duck antenna designed to operate in 2.4 GHz to 2.5 GHz or 5.1 GHz to 5.9 GHz bands. The Dual Band design of this antenna eliminates the need to purchase different antennas for each frequency.

When run **wpa_fast** we collect two different type of log. The first log trace the different phase of connection set-up, we save the time spent for every stage and we logged many information related to AP, we log channel, MAC address and ESSID (Extended Service Set Identification). We use the first log to compute a average time to set-up connection for every stage and a cumulative time, besides we compute the successful rate for the different part of the connection. The second log stores the result of the all scanning, regardless of the fact that the station is associated or not to that particular AP, and in this case we consider in addition other information like type of encryption and channel. Through the GPS trace and the second log we can create a map of the AP position, the figure 8.2 show a map in which we point only the AP whence we have a network level set-up connection in a area near to UCLA campus.

## 8.7 Experimental evaluation

To evaluate **wpa_fast** , we comparison it with QuickWifi [53], another framework to improve the connection set-up, QuickWifi is a streamlined process that combines all the different protocols involved in obtaining connectivity (across all layers) into a single process, including a new optimal channel scanning policy. QuickWifi improves the connection set-up time but not provides a compatibly with different hardware and it manages only the open system authentication. **wpa_fast** differently, in addition to what you already make in QuickWifi, add compatibility with all hardware and ability to use the secure systems.

To evaluate the **wpa_fast** , we realize an experiment in which we use a car equipped with 2 eeepc at the same time, in the first eeepc we make run QuickWifi and in the second eeepc we

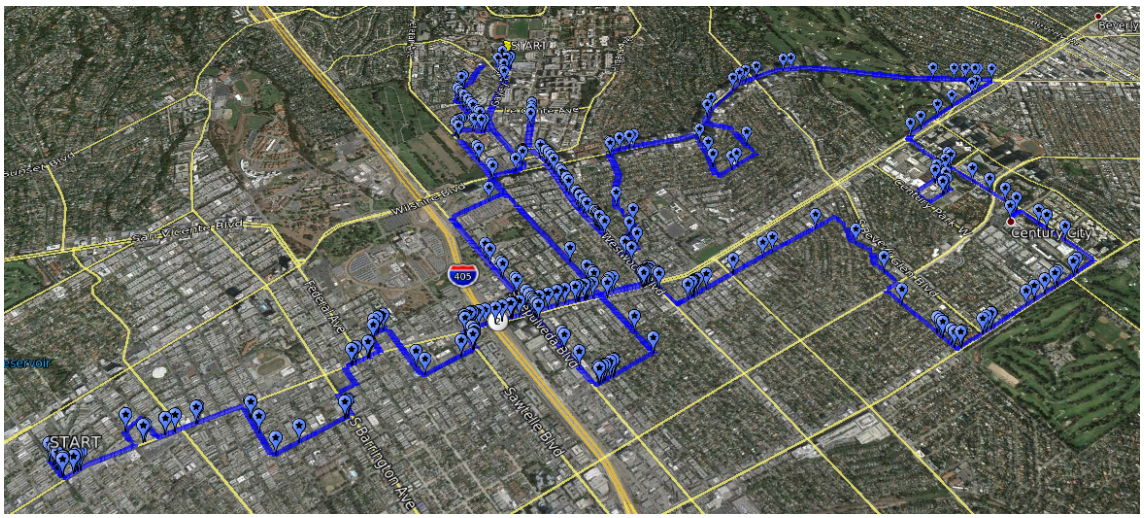



Figure 8.2: AP with good DHCP association

make run **wpa_fast** , in this way we can evaluate the result while the programs run in the same environment in the same moment, and realize attempts only for AP with open system access, because QuickWifi cant connect with other AP. We make same little modify at QuickWifi to return a log with a same format of **wpa_fast** as explained in the previous section, after the test we use the MATLAB tool to obtain a plot of the data collected and comparison the two programs. We realize the experiment in the area near to UCLA campus for a long time of 2 hours.

For every test we show three different plot, they are:

- **Rate successful**
- **Average time**
- **Distribution of the sum time**

The first plot is the **Rate successful**, we show **the ratio between the number of successful attempts and the total number of attempts** for the different stage of the connection set-up, it may happen several attempts fail at different stages, for example when a subsequently failed as the car moved out of range of the AP. In the second plot, **Average time**, we show **the time that wpa_fast spent for each stage of connection set-up**. The third plot, **Distributin of the sum time**, we present the **CDF of connection establishment times**, based on a total part of the connections set-up, in this case we consider only association, discovery and request.

Figure 8.3 shows the performance plot for the two programs, we can see the QuickWifi in the left side and **wpa_fast** in the right side. Start from the top, we have the distribution of connection success, these two plots are very similar for the stage, DHCP discovery, DHCP request, and PING, in this case we have a rate of, 0.57%, 0.53%, 0.01% for a total number of 2021 association request in QUICKWIFI, while 0.54%, 0.52%, 0.02% for a total number of 1963 association request in **wpa_fast**

For the first stage, Authentication/Association, we have a value of 0.78% for QuickWifi, and 0.68% for **wpa_fast** , this difference is due to different technical to managed this stage, in fact the QuickWifi skip the Authentication stage, QuickWifi send the association frame immediately after the authentication frame without wait the response, this because QuickWifi work in monitor mode and because not support the AP with security systems. While **wpa_fast** reports a value of 0.68%, this value is 0.10% less to QuickWifi, because **wpa_fast** manages both the procedures, authentication and association, through the netlink for to have the **compatibility with all the drivers modules and cards, and support the AP with security systems**. The compatibility and the support introduce little latency in the connection set-up, such as showed in the average time bar plot, and decreases the distribution of connection success, but increases the flexibility of **wpa_fast** As mentioned in [53], we note that the vast majority of failures happen in the DHCP phases. There are several possible contributing factors to this. Poor channel quality is in all likelihood the biggest culprit. Due to the broadcast nature of DHCP messages, there are no MAC layer retries, with a corresponding increase in packet losses. However, some DHCP servers may be unable to respond in time and in this case we restarts the process.

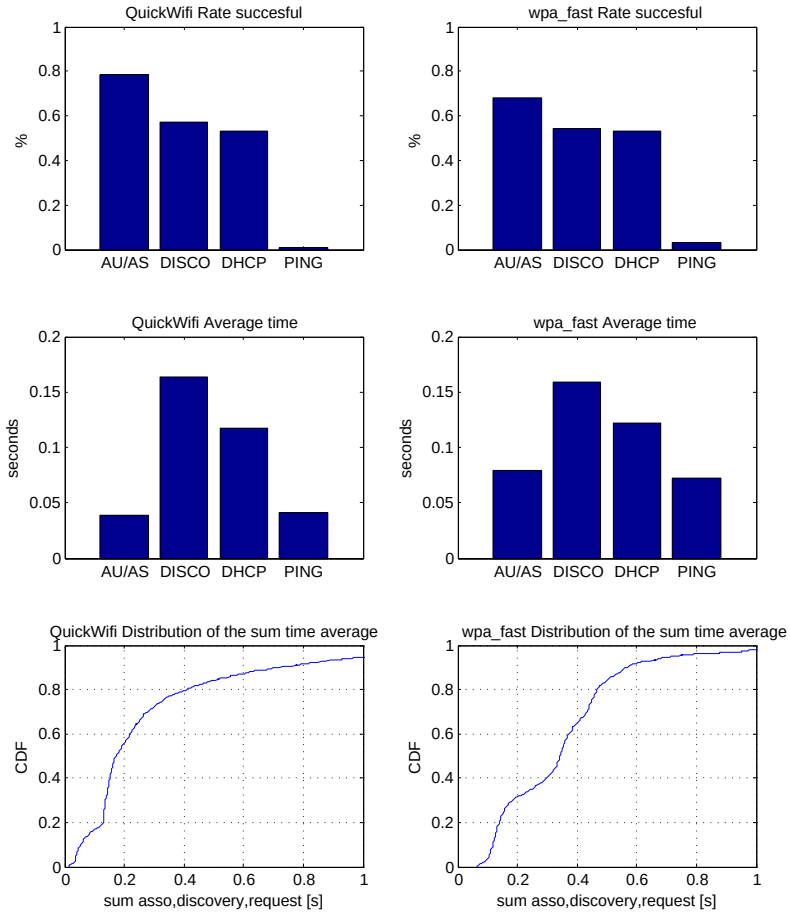



Figure 8.3: Comparison between **wpa_fast** and QuickWifi

The two plot in the center of the figure 8.3 show average time spent in each of the phases, for all connections that eventually succeeded. For the PING stage we measure the time to receive the echo response. Also in this plot we have similar result for the stage, DHCP discovery, DHCP request, and PING, in this case we have a time of, 163ms, 117ms for QuickWifi, and 158ms, 121ms for **wpa_fast** . Instead in first phase **wpa_fast** spent the double of the time of QuickWifi, 788ms in oppositin at 387ms, this for the reason explain in the previous part, this phase double long for **wpa_fast** because wait the response for two frame, while QuickWifi only for one.
As mentioned in [53] DHCP dominates the delay incurred. The reason for this delay is that both the DHCP discover and the DHCP request packets are sent to the broadcast address. Such packets do not benefit from link-layer ACKs and retransmissions, which means that this stage takes a long time to complete.
Last row in figure 8.3 has two plots that shows the CDF of connection establishment times for QuickWifi and **wpa_fast** , here we can observe that there are present two main difference, the start of the line is near 15ms for QuickWifi, while 60ms for **wpa_fast** , the second difference is the shape of the line, the **wpa_fast** line present a flat part between 170ms and 270ms, both this difference are due to the reason explain in the previous part. We have the first difference because **wpa_fast** waits the response for the authentication frame, and the second difference because **wpa_fast** uses netlink to manage the authentication/association phase, and it uses a time to retry the frame of 100ms, this value can't be modify. At the end QuickWifi complete the set-up connection in 900ms for the 93% of the time, while **wpa_fast** for the 97% of the time.

To proving the **wpa_fast** perform, we present a plot of a cumulative experiments, the figure 8.4 shows three plot that present the performance of **wpa_fast** such as explain in the previous section, for this result **we realize several experiments in different hours and days in the Los Angeles area**, suburban and urban area, for a total time of 326 minutes and 17097 number of association request. After collect the data, we use MATLAB to analyzed all the trace log together to obtain a cumulative result show in the figure 8.4.
The first plot present the distribution of connection success, we have 0.70%, 0.53%, 0.50% and 0.09%, respectively for Association/Authentication, DHCP discovery, DHCP request and PING stages.

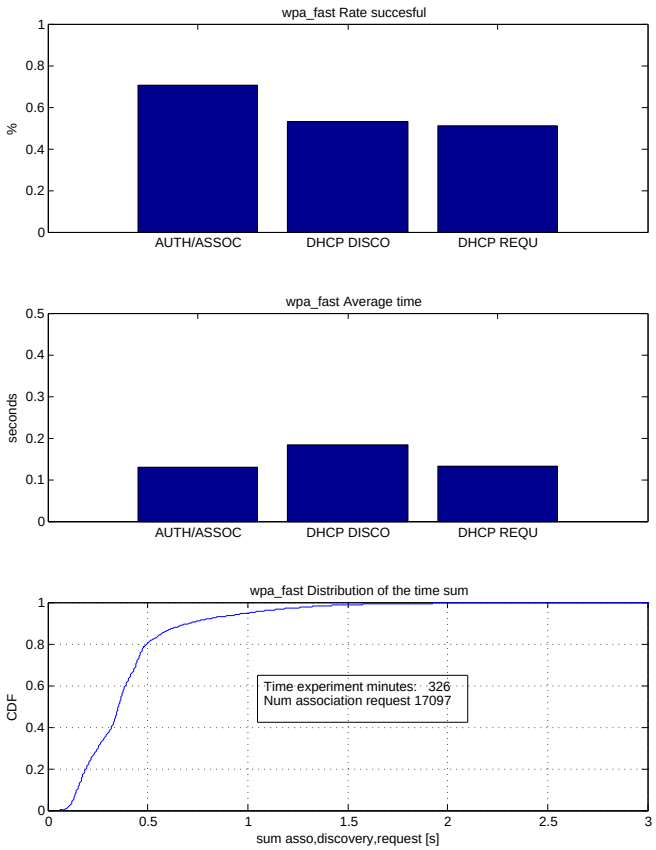



Figure 8.4: Rate succesfull, average time and CDF for cumulative experiments

The second plot of the figure 8.4 show average time spent in each of the connection set-up phases, and in this case we have 129ms, 184ms, 131ms and 49ms, respectively for Association/Authentication, DHCP discovery, DHCP request, and PING stages.
Last plot in figure 8.4 show the CDF of connection establishment times, here we can observe that **wpa_fast** complete the set-up connection in 900ms for the 93% of the time.
How can that in all the plot we make sure we have values similar to those shown in the previous section, that so we prove that **wpa_fast keeps this behaviour for a long time of experiments in different scenarios and areas**.

# Appendix A

# APPENDIX

## A.1 Hardware Deployment

Note that the presented MAC-Engine firmware has been tested on BCM4311 and BCM4318 chipset revisions, using the B43 driver on Linux kernel 3.1.4. The firmware supports all works modes the infrastructure, working as a station or AP (Access Point) and the ad-hoc mode, it is compatible (in terms of protocol timings, frame fields, etc.) with legacy DCF stations in 11b and 11g mode, and it provides throughput performance comparable with the proprietary card firmware when executing the DCF state machine. It does not currently support: the RTS/CTS handshake, the hardware cryptography acceleration (to be used without encryption!) and the dot11 QoS mode (to be disabled when loading the module). Moreover, it has not been tested for working in 11a mode.

This thesis describes the WMP that was developed in according with the CNIT research under the EU project FLAVIA, following the paradigm introduced above, on a specific commercial wireless card designed by Broadcom. The WMP has been implemented by writing a new firmware that replaces the original sofware from Broadcom with a generic state machine executor called **MAC-Engine**: this work has been possible thanks to the availability of a documented open firmware for a specific chipset of the big AirForce54G family of Broadcom wireless NICs, namely the OpenFWWF Project [1].

The combination of the MAC-Engine, the WMP-Editor, the WMP-Compiler, the Byte-Code-Manager and the driver is a complete and cheap tool-chain that allows developing and testing new MAC programs in a very simple, robust and quick way over an ultra-cheap platform. Each component of the toolchain can be found on the site http://wmp.tti.unipa.it, where we provide:

- WMP documentation;
- the MAC-Engine firmware that replaces the original card firmware;
- the graphical editor WMP-Editor;
- many Byte-Code examples (including standard DCF, Time Division Multiple Access and Direct Link);
- the Byte-Code-Manager.

Finally all information and news to WMP are publicized on site [**?**]

## A.2 References and link

Some useful links for integrating this documentation can be found in the following list:

- WMP team: `http://wmp.tti.unipa.it/`
- WMP download tools and firmware: `https://github.com/ict-flavia/Wireless-MAC-Processor`
- OpenFWWF team: `http://www.ing.unibs.it/openfwwf`
- BCM Specs Site: `http://bcm-v4.sipsolutions.net`

[1] OpenFirmWare for WiFi networks, http://www.ing.unibs.it/openfwwf

- B43 information: `http://wireless.kernel.org/en/users/Drivers/b43#firmwareinstallation`
- B43 compilation tools: `https://github.com/mbuesch/b43-tools`

# Bibliography


[1] WMP Project, http://wmp.tti.unipa.it/.

[2] http://en.wikipedia.org/wiki/Media_access_control.

[3] http://en.wikipedia.org/wiki/Finite-state_machine.

[4] http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Linux.Wireless.mac.html

[5] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, "Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware", IEEE INFOCOM'12, Orlando (FL), USA, Mar. 25-30, 2012.

[6] G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, I. Tinnirello, "MAClets: Active MAC Protocols over hard-coded devices", ACM CoNEXT 2012, Nice, France, Dec. 10-13, 2012.

[7] P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, I. Tinnirello, G. Bianchi, "A Control Architecture for Wireless MAC Processor Networking", Future Network and MobileSummit'13, Lisbon, Portugal, Jul. 3-5, 2013.

[8] F. Gringoli, D. Garlisi, P. Gallo, F. Giuliano, S. Mangione, I. Tinnirello, MACE-ngine: A New Architecture for Executing MAC Algorithms on Commodity WiFi Hardware, WINTECH(MOBICOM) Las Vegas 2011

[9] J. Mitola III, G. Q. Maguire, "Cognitive radio: Making software radios more personal", IEEE Personal Communications, vol. 6, no. 4, pp. 13–18, August 1999.

[10] Kramer, M. Dept. of Inf. Technol. Media, Mid Sweden Univ., Sundsvall, Sweden Bader, S.; Oelmann, B. Implementing Wireless Sensor Network applications using hierarchical finite state machines

[11] P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, I. Tinnirello, G. Bianchi, Wireless MAC Processor Networking, A Control Architecture for Expressing and Implementing High-Level Adaptation Policies in WLANs, DECEMBER 2013 | IEEE vehicular technology magazine.

[12] The GNURadio Software Radio, http://gnuradio.org/trac.

[13] USRP. The universal software radio peripheral, http://www.ettus.com.

[14] Open firmware for WiFi networks, http://www.ing.unibs.it/openfwwf.

[15] http://git.bues.ch/gitweb?p=b43-ucode.git;a=summary

[16] Wireless Open Access Research Platform,http://warp.rice.edu/trac.

[17] B43 developer/debug tool, https://github.com/mbuesch/b43-tools.

[18] FLAVIA European Project - FLexible Architecture for Virtualizable future wireless Internet Access, http://www.ict-flavia.eu/.

[19] Gallo, P.; Gringoli, F.; Tinnirello, I., "On the flexibility of the IEEE 802.11 technology: Challenges and directions," Future Network & Mobile Summit (FutureNetw), 2011 , vol., no., pp.1,10, 15-17 June 2011

[20] IEEE 802.11-2007: A new release of the standard that includes amendments a, b, d, e, g, h, i and j. (July 2007)

[21] Keranidis, Stratos and Kazdaridis, Giannis and Passas, Virgilios and Korakis, Thanasis and Koutsopoulos, Iordanis and Tassiulas, Leandros, Online Energy Consumption Monitoring of Wireless Testbed Infrastructure Through the NITOS EMF Framework; WiNTECH '13, Miami, Florida, USA.

[22] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, D. Grunwald, "SoftMAC - Flexible Wireless Research Platform" HotNets, Nov. 2005.

[23] `http://www.cs.berkeley.edu/ananthar/oml.html`

[24] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D.C. Sicker, D. Grunwald, "MultiMAC - An adaptive MAC Framework for Dynamic Radio Networking", IEEE DySPAN 2005, Nov. 2005.

[25] M.H. Lu, P. Steenkiste, and T. Chen, "Using commodity hardware platform to develop and evaluate CSMA protocols", ACM WiNTECH 2008, Sep. 2008, pp. 73-80.

[26] P. Djukic, P. Mohapatra, "Soft-TDMAC: A Software-Based 802.11 Overlay TDMA MAC with Microseconds Synchronization", IEEE Trans. on Mobile Computing, Issue 99, April 2011,

[27] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, H. Balakrishnan, "Airblue: A System for Cross-Layer Wireless Protocol Development", ACM/IEEE ANCS 2010.

[28] FP7 ICT FLAVIA project, "Analysis of state of the art: contention-based extensions", Deliverable D6.1, Oct. 2010; `http://www.ict-flavia.eu`

[29] `http://calradio.calit2.net/calradio1.htm`

[30] A. Di Stefano, G. Terrazzino, C. Giaconia, "FPGA Implementation of a Reconfigurable 802.11 Medium Access Control", Int. Conf. on Wireless Reconfigurable Terminals and Platforms (WIRTEP), Rome, April 2006.

[31] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, G. M. Voelker, "Sora: High Performance Software Radio Using General Purpose Multi-core Processors", NSDI 2009.

[32] G. Nychis, T. Hottelier, Z. Yang, S. Seshan, P. Steenkiste, "Enabling MAC Protocol Implementations on Software-defined Radios", NSDI'09, 2009.

[33] X. Zhang, J. Ansari, G. Yang and P. Mahonen "TRUMP: Supporting Efficient Realization of Protocols for Cognitive Radio Networks", IEEE DySPAN 2011

[34] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova, P. Mahonen, "Decomposable MAC Framework for Highly Flexible and Adaptable MAC Realizations" Proc. of IEEE DySPAN 2010, April 2010, pp.1-2.

[35] Y. Ling, H. Tiansheng, L. Caixing, X. Yue, and Z. Haoen, "A Reprogamming Protocol Based on State Machine for Wireless Sensor Network," International Conference on Electrical and Control Engineering, ICECE, pp. 232-235, 2010.

[36] N. Kothari, T. Millstein, and R. Govindan, "Deriving State Machines from TinyOS Programs Using Symbolic Execution" in Proceedings of the 7th International Conference on Information Processing in Sensor Networks, IPSN, 2008, pp. 271-282.

[37] S. Smolau and R. Beaubrun, "State-Oriented Programming for TinyOS" in Proceedings of the 2007 Summer Computer Simulation Conference, SCSC, 2007, pp. 766-771.

[38] M. Samek, Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems, 2nd ed. Newton, MA, USA: Newnes, 2008.

[39] T.-H. Kim and S. Hong, State Machine Based Operating System Architecture for Wireless Sensor Networks, K.-M. Liew, H. Shen, S. See, W. Cai, P. Fan, and S. Horiguchi, Eds. Springer Berlin / Heidelberg, 2005, vol. 3320.

[40] O. Kasten and K. Röomer, "Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines," in Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, IPSN, 2005.

[41] D. Harel, "Statecharts: A Visual Formalism for Complex Systems" Science of Computer Programming, vol. 8, no. 3, pp. 231-274, Jun. 1987.

[42] T. Hanneforth, "Finite-State Machines: Theory and Applications" Lecture Notes, Aug. 2010. [Online]. Available: tagh.de/tom/wp-content/uploads/FSMUnweigtedAutomata.pdf

[43] IEEE Standard 802.11 - 1999; Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications; November 1999.

[44] http://www.broadcom.com/products/brands/AirForce

[45] http://wireless.kernel.org/en/users/Drivers/b43

[46] Linux kernel mac80211 framework for wireless device drivers. url-http://linuxwireless.org/en/developers/Documentation/mac80211.

[47] debugfs : http://lwn.net/Articles/334546/

[48] Hartenstein, H. et al., "Position-Aware Ad Hoc WirelessNetworks for Inter-Vehicle Communications: The FleetNet Project" MobiHoc '01: Proc. 2nd ACM Int'l. Symp. Mobile Ad Hoc Networking Computing, New York: ACM Press, pp. 259-62, 2001

[49] M. Torrent-Moreno, S. Schnaufer, R. Eigner, C. Catrinescu, andJ. Kunisch, "NoW - Network on Wheels': Project Objectives,Technology and Achievements" 5th International Workshop on Intelligent Transportation (WIT), pages 211 - 216, Hamburg,Germany, March 2008

[50] D. Reichardt, Miglietta, M. Moretti, L. Morsink, P. Schulz,W. "CarTALK 2000: Safe and Comfortable Driving Based upon Inter Vehicle Communication" Intelligent Vehicle Symposium,2002. IEEE, volume 2, pp 545-550, Mar 2003

[51] "Car 2 Car Communiaction Consortium Manifesto" version 1.1, technical report, Aug 2007 Available: www.car-to-car.org

[52] Paul Arana Benefits and Vulnerabilities of Wi-Fi Protected Access 2 (WPA2) INFS 612 - Fall2006

[53] J. Eriksson, H. Balakrishnan, and S. Madden. Cabernet: Vehicular content delivery using wifi. In Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM, pages 199-210, 2008.

[54] V. Bychkovsky, B. Hull, A. Miu, H. Balakrishnan, and S. Madden. A measurement study of vehicular internet access using in situ wi-fi networks. In Proceedings of MOBICOM, pages 50-61, 2006.

[55] Suranga Seneviratne, Aruna Seneviratne, Prasant Mohapatra, Pierre-Ugo Tournoux Characterizing WiFi Connection and Its Impact on Mobile Users: Practical Insights WiNTECH́13, September 30, 2013, Miami, Florida, USA

[56] http://hostap.epitest.fi/wpa_supplicant/.

[57] https://git.fedorahosted.org/git/pump.git.

[58] http://www.l-com.com/wireless-antenna\-24-25-ghz-51-59-ghz-7dbi-rigid-rubberduck-antenna.

[59] https://www.eduroam.org/.

[60] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In Proc. ACM SenSys, Nov. 2006. http://cartel.csail.mit.edu.

[61] http://www.wi-fiplanet.com/tutorials/article.php/1216351