Multi-Channel Wireless LAN on a Shared Multi-Processor System-on-Chip

by

R.J.V. Schiedon

Supervisors:
prof. dr. ir. C.H. van Berkel (TU/e)
ir. R.J.M. Nas (NXP Research)

Eindhoven, March 2008
# Contents

1 Introduction .................................................. 1
   1.1 Outline .................................................. 2
   1.2 Glossary .................................................. 2

2 Context ......................................................... 3
   2.1 Software defined radio ....................................... 3
   2.2 NXP platform ................................................. 4
   2.3 Task transaction level interface .............................. 4
   2.4 IEEE 802.11a Wireless LAN ................................... 5

3 Problem description ............................................. 9
   3.1 Problem statement ............................................. 9
   3.2 Related work ................................................. 10
   3.3 Approach .................................................... 10

4 Application model ............................................... 13
   4.1 Definitions ................................................. 13
   4.2 Task-graph .................................................. 15
   4.3 Constraints .................................................. 25
   4.4 Summary .................................................... 29

5 Architecture model ............................................... 31
   5.1 Interconnect ............................................... 31
   5.2 RF front-end ............................................... 32
   5.3 EVP vector processor ....................................... 32
   5.4 Software codec ............................................. 35
   5.5 ARM micro-processor ...................................... 35
   5.6 Summary .................................................... 36

6 Mapping .......................................................... 37
   6.1 Definitions ................................................. 37
   6.2 Interconnect ............................................... 38
   6.3 RF front-end ............................................... 39
   6.4 EVP vector processor ....................................... 40
   6.5 Software codec ............................................. 48
   6.6 ARM micro-processor ...................................... 50
   6.7 Summary .................................................... 52

7 Analysis .......................................................... 53
   7.1 Initial four radio configuration .............................. 53
   7.2 Effects of parameters ....................................... 56
   7.3 Recommended four radio configuration ......................... 60
   7.4 One radio configuration ..................................... 62
7.5 Summary ................................................................. 66

8 Simulation .......................................................... 67
8.1 Implementation ...................................................... 67
8.2 Simulations ......................................................... 68
8.3 All radios detecting ............................................. 69
8.4 One radio receiving a frame .................................. 69
8.5 Incoming data symbols ........................................... 71
8.6 Outgoing data symbols ........................................... 73
8.7 All downloading a frame ........................................ 73
8.8 All downloading a frame at the same time .................. 74
8.9 Summary ............................................................. 74

9 Results ................................................................. 77
9.1 Summary ............................................................. 77
9.2 Assumptions ....................................................... 79
9.3 Conclusion .......................................................... 80

Bibliography ........................................................... 81

A IEEE 802.11a wireless LAN ...................................... 83
A.1 Physical layer ...................................................... 84
A.2 MAC layer .......................................................... 90
Chapter 1

Introduction

Wireless communication is something that will become more and more important in our daily lives. Today a lot of devices already communicate with each other using one of the many existing wireless protocols. Examples are mobile telephones, notebook computers and navigation systems. But already new devices with wireless connectivity are announced, like televisions and storage arrays. This area is evolving rapidly, so the industry has to adapt quickly.

In most devices the wireless communication system is implemented mainly in hardware. Devices that support multiple standards are built with a lot of different chipsets, which make them complex in design. Each new communication standard means an expensive revision to the hardware.

To shorten the implementation time of new standards a more generic platform is being developed with highly reconfigurable components on which an implementation of a wireless communication standard only consists of a set of configuration settings and one or more software applications. Such a platform is called a software defined radio (SDR). An update for a software defined radio can be done in the field; hence support for new standards comes down to installing a new piece of software.

The research department of NXP (formerly known as Philips Semiconductors) is aiming for a multi-processor system-on-chip SDR platform that should support wireless services such as local area networks (LAN), voice communication and digital video broadcasting. A particular wireless LAN standard is the IEEE 802.11a standard, which defines a method for transmitting data between one or more stations using orthogonal frequency-division multiplexing (OFDM).

One of the targets of the NXP platform is to support running multiple services in parallel with each other. This means that one or more instances of various wireless communication standards are active at the same time. This means for a lot of standards that each instance operates at its own communication channel. In other words, the system will support multi-channel implementations, and for IEEE 802.11a in particular: multi-channel wireless LAN. Operating at multiple channels at the same time is not very common. It might be useful for roaming devices that need to switch between different networks while they travel. Multiple channels can also be combined to get a higher overall bandwidth.

IEEE 802.11a is not the only standard that uses OFDM. For instance the IEEE 802.11n wireless LAN standard, which is expected to become the next major data communication protocol, is basically a combination of multiple 802.11a channels bundled together. Other standards like IEEE 802.16 (also known as WiMAX) and terrestrial digital video broadcasting (DVB-T) are also based on OFDM. So although 802.11a has not been very popular yet, it is a precedent of many other coming standards.

Whether or not the multi-processor platform will be able to support multiple IEEE 802.11a wireless LAN radios remains to be seen. At the moment of writing the platform of NXP is not completely designed yet and an implementation of the wireless LAN application for this platform is unavailable. In other words, verifying the feasibility of running multi-channel wireless LAN on the proposed platform will have to be done theoretically.
CHAPTER 1. INTRODUCTION

1.1 Outline

This document is divided into three main parts. The first part consists of two chapters that describe the context of the subject (Chapter 2) and the problem statement (Chapter 3). It is followed by three chapters that explain the chosen approach step-by-step (Chapters 4, 5 and 6). The last part consists of an analysis of the results (Chapter 7), a simulation (Chapter 8) and ends with a conclusion (Chapter 9). The included appendix contains a more detailed description of the IEEE 802.11a WLAN standard.

1.2 Glossary

ACK acknowledgement
ADC analog-to-digital converter
ARM advanced RISC machine
BPSK binary phase-shift keying
CASSE camellia system simulation environment
CRC cyclic redundancy check
CSMA/CA carrier sense multiple access with collision avoidance
CTS clear to send
DAC digital-to-analog converter
DCF distributed coordination function
DIFS DCF interframe space
DSSS direct sequence spread spectrum
EIFS extended interframe space
EVP embedded vector processor
FEC forward error correction
FFT fast fourier transformation
FHSS frequency-hopping spread spectrum
GI guard interval
IEEE institute of electrical and electronics engineers
IEEE 802 IEEE LAN/MAN standards committee
IR infrared
ISM industrial scientific medical frequency band
ISR interrupt service routine
ITCP inter-task communication protocol
LTS long training symbol
MAC medium access control layer
MPSoC multi-processor system-on-chip
NAV network allocation vector
NoC network-on-chip
OFDM orthogonal frequency-division multiplexing
PCF point coordination function
PHY physical layer
PIFS PCF interframe space
QAM quadrature amplitude modulation
QPSK quadrature phase-shift keying
RF radio frequency
RTS request to send
SDR software defined radio
SIFS short interframe space
STS short training symbol
SWC software codec
TDMA time division multiple access
TTL task transaction level interface
VLIW very long instruction word
WLAN IEEE 802.11a wireless LAN
Chapter 2

Context

In today’s communication devices such as cellular phones, services like voice communication, wireless local-area network and global positioning are implemented with application specific hardware. Each individual protocol is implemented with a separate chipset. Support for multiple communication protocols will make these devices quite complex.

Furthermore, the world of communicating devices is evolving rapidly. Every several years a new protocol pops up that promises even better performance. In order to keep up with this market the hardware of a device must be updated frequently, which is a costly operation.

2.1 Software defined radio

To make the development process more flexible a highly reconfigurable platform is needed on which current and future services can be implemented with the same hardware. Such a system is called a software defined radio (SDR). For these systems changes to a protocol or support for a new service can be implemented by running a new piece of software. Software radios offer great benefits for military and mobile consumer devices, because they have to serve a variety of rapidly changing services.

The concept of software defined radio is rather new. The first running system was the SPEAKeasy Phase-1 developed by the United States Military and it was demonstrated at the Joint Warrior Interoperability Demonstration (JWID) of 1995. It was a very big system, consisting of a 19” rack with a Sun Sparc 10 workstation and additional hardware such as DSPs. It was just a programmable base band radio, so not an actually software radio. The first “field ready” package was the SPEAKeasy Phase-2, which was successfully demonstrated at the TF-XXI AWE on March 1997. This was an in the field programmable radio that could operate on the HF, VHF and UHF bands and was build around a combination of FPGAs and 40 MHz TI C40 DSPs. The design was made to be an open software and hardware architecture.

Unfortunately, these systems are not suited for commercial use. They are big, heavy and do not support protocols that are used by commercial communication services, which are more multimedia oriented. For these high-speed communication links at radio frequencies well above 2 GHz a more powerful system is required. And when several of these links are running simultaneously, even more processing power is needed. It is also necessary that the system is small and energy efficient, so it could be used in mobile devices like cellular phones or PDAs.

Today several companies are developing software defined radios. Some of them are united in the Software Defined Radio Forum, where they exchange information. There is even an open-source project called GNU radio.
2.2 NXP platform

The research department of NXP Semiconductors is developing a software defined radio platform, which is designed as a multi-processor system-on-chip (MPSoC) [1]. Such a system consists of a single chip on which a set of heterogeneous processor cores are connected with each other using a flexible on-chip communication network.

This platform is still in development. There are no concrete implementations available yet, so most of the platform specifics in this thesis are based on what the platform is expected to look like. The platform will be chosen such that several properties hold. First of all, it should be highly reconfigurable, so implementing a new protocol should not be a problem. Furthermore, it should be able to run multiple standards (i.e. protocols) simultaneously.

To achieve this, the architecture is designed around at least the following four processor cores:

- **Radio-Frequency (RF) front-end**: A set of hardware components responsible for sampling an analog signal from an antenna into digital samples, and vice versa.
- **EVP**: A vector processor optimized for digital signal processing.
- **Software Codec**: A specifically designed processor for coding and decoding blocks of data.
- **ARM**: A general purpose processor for controlling the incoming and outgoing flow of data bits.

Each processor has a block of local memory that contains the software, configuration parameters and communication channel administration (like tables and buffers). These memory blocks can also be accessed by other processors using the on-chip communication network. An overview is depicted in Figure 2.1. The processors will be discussed in detail later on.

![Architecture](image)

**Figure 2.1: Architecture**

2.3 Task transaction level interface

Designing and verifying software for a MPSoC is a difficult task [1]. Each processor core has its own instruction set with accompanying compiler which makes developing a complete system difficult. Furthermore, communication interfaces between the cores has to be chosen carefully. For interoperability each core must use the same interface as the cores it wants to communicate with. For that last problem a new interface has been introduced by Philips Research, which is called the Task Transaction Level (TTL) interface [4]. This task-level interface is an abstract set of functions that can be used to design a parallel application model of communicating tasks, and can also be used as a platform interface for integrating hardware and software tasks on top of a platform infrastructure. The interface consists of a set of classes and functions, which allow you to specify tasks with communication ports and channels connecting these ports. By implementing this interface for a certain platform the designed model can be mapped on a specific hardware architecture. So TTL combines specification, design and integration into one modeling process.

An application model using the TTL interface consists of a graph of tasks that are connected to each other via channels. Over these channels data is transmitted using tokens. Channels are directional, so only one task can produce tokens for one or more consuming tasks.
During the design of an application model it is common practice to simulate and analyze everything before it is implemented on the hardware. Also when the hardware platform itself is not available already it may be required to simulate the model on a virtual testing platform. For these purposes some environments that implement the TTL interface have been developed. One of them is the Camellia System Simulation Environment (CASSE) developed by Victor Reyes and others [5].

CASSE uses the Inter-Task Communication Protocol (ITCP) to implement the TTL interface primitives. It is implemented using the SystemC library and accepts a model written in the C++ programming language. Besides running the model on the host platform, CASSE can also simulate the model on a virtual multi-processor hardware platform that contains different computation units (for example an ARM micro-processor). During the simulation CASSE can give statistics about the behavior of the tasks, the communications via the various channels and the usage of the processing and network elements.

2.4 IEEE 802.11a Wireless LAN

One of the protocols that the NXP platform is going to support is the IEEE 802.11a Wireless LAN protocol [11]. This section will only explain the basic principles of this standard. Appendix A contains a more detailed explanation. The standard is an amendment to the original 802.11 WLAN protocol [10], using orthogonal frequency-division multiplexing (OFDM) in the physical layer. The medium access control (MAC) layer is largely the same.

Like normal frequency-division multiplexing the channel via which the data is transmitted is split into a number of narrowband sub-channels, each at a separate frequency. With OFDM these frequencies are chosen such that the carrier waves are orthogonal to each other, which eliminates cross-talk between the sub-channels.

Like most other protocols most of the processing is being done in the physical layer. With the IEEE 802.11a WLAN physical layer data is transmitted in blocks of 4 $\mu$s called OFDM symbols. It starts with a cyclic guard interval of 0.8 $\mu$s and is followed by a block of 3.2 $\mu$s. This block is sampled and put through a 64-point fast fourier transformation (FFT) that produces 64 sub carriers of which only 48 are used for actual data transportation. Each of the sub carriers contain a modulated signal that defines up to 6 bits per symbol, so per OFDM symbol at most 288 bits can be transmitted. The resulting bits are put in a single sequence and passed through a decoder, which produces a block of at most 216 bits per OFDM symbol. This means that the highest achievable data rate is 216 bits per 4 $\mu$s or 54 Mb/s. These bits represent the actual payload and are passed to the MAC layer.

The OFDM symbols are part of a physical layer frame. Such a frame starts with a preamble of ten short training symbols (STS) and two long training symbols (LTS) to help detecting and synchronizing to the start of a frame. They are followed by one OFDM symbol that represents the header information. The header symbol is modulated and coded with the lowest possible data rate and contains, amongst other things, the length of the payload in bytes and the data rate (i.e. the modulation scheme and coding rate) at which it is transmitted. From this the number of data OFDM symbols that follow can be computed.

The MAC layer controls the flow of incoming and outgoing frames using carrier sense multiple access with collision avoidance (CSMA/CA). Before a station can transmit a frame it senses the channel and waits until it is not occupied by another station. It will then wait for a random time before starting. This reduces the chance that it starts simultaneously with another station. Once a frame has been transmitted it needs to be positively acknowledged by the receiving station after a certain period, called the short interframe space (SIFS). If it fails to do so the sending station will retransmit the frame. Other stations will wait a bit longer after a frame transmission has been detected, so that the acknowledgements can be transmitted without interference. Unlike the physical layer, the MAC layer does not perform any processing on the data itself.

The short interframe space is the shortest and most crucial interval in the protocol. If a system cannot deal with a frame and transmit the acknowledgement within this time then it cannot work
modern properly.

2.4.1 State diagram

A number of states can be defined in which an IEEE 802.11a WLAN radio can be in. These states
tell the radio what kind of operations it has to do. The following seven states can be defined.

- **Frame Detection** The radio is waiting for a frame to arrive.
- **Gain Control** A frame has been detected and the signal amplifier is adjusted to increase
  the reception quality.
- **Time Synchronization** A frame is detected but the location within it is unknown; therefore
  the radio tries to synchronize to the short training symbols in order to find the start of frame.
- **Frequency Synchronization** The start of a frame is known and the long training symbols
  are used to synchronize the radio in the frequency domain. This will compensate the
  frequency errors that a wireless transmission introduces.
- **Header Processing** The preamble of a frame has been processed, so the first OFDM symbol
  can be received or transmitted. This symbol contains the information about how the payload
  OFDM symbols are transmitted. It is processed like an ordinary OFDM symbol coded and
  modulated at the lowest rates.
- **Data Processing** The payload of the frame is processed. It consists of a number of OFDM
  symbols that are transmitted with the data rate defined in the preceding header symbol.
  For incoming OFDM symbols the samples are passed through a FFT, demodulated, decoded
  and send to the MAC layer. For outgoing symbols the data from the MAC layer is coded,
  modulated and combined with an inverse FFT before they are transmitted through the air.
- **Preamble Transmission** For outgoing frames the radio needs to transmit the preamble
  before the header OFDM symbol can be transmitted.

These states are connected as depicted in Figure 2.2. Initially a radio is in the Frame Detection
state. If the MAC layer notifies that it wants to transmit a frame then the radio goes to the
Preamble Transmission state. Once the preamble has been transmitted the first header symbol
can be processed in the Header Processing state. The actual data delivered by the MAC layers is
then processed in the Data Processing state. When the frame has been transmitted entirely the
radio goes back to the idle state and continues with detecting frames.

If a frame is detected in the Frame Detection state then the radio first adjusts the gain level
in the Gain Control state, finds the start of a frame in the Time Synchronization state and
compensates frequency errors in the Frequency Synchronization state. It can then finally process
the first incoming OFDM symbol in the Header Processing state and the following symbols in the
Data Processing state. Again the radio goes back to the Frame Detection state if the frame is
completely received. If during the processing of the preamble something goes wrong (for instance
a false detection took place) then the radio will go back to Frame Detection.
Figure 2.2: Wireless LAN state diagram
Chapter 3

Problem description

NXP is developing a software-defined radio platform with a number of goals. First of all, it should be capable of running several wireless communication standards. Secondly, these standards should be able to run in parallel without disturbing each other. The platform should also be future proof, such that several upcoming standards might be implemented on it without changing the platform (very much). All this should fit on a single multi-processor system-on-chip.

Running multiple standards simultaneously on one system demands quite some processing power. To find out what the possibilities are of the platform in mind, several wireless communication standards are studied by NXP. One of the more challenging standards is the IEEE 802.11a wireless LAN standard. This standard is based on techniques that are also used in multiple other standards, including future ones like IEEE 802.11n. The IEEE 802.11a wireless LAN is challenging because it is has various real-time requirements for the throughput and latencies of a radio. It is also an uncoordinated packet based protocol, which means that data can arrive at any given time and the radio should use the incoming data to synchronize to the network.

Having one IEEE 802.11a wireless LAN radio running on a shared platform is nice, but running multiple instance makes it even more interesting. It gives some insights on how multiple standards can influence the performance of others. It can also show that if a platform is capable of running multiple WLAN radios simultaneously, then it would probably have enough processing power for supporting future standards as well. The goal of NXP is to support as much channels simultaneously, but this thesis will restrict this to at most four.

As with most wireless LAN radios, the IEEE 802.11a standard consist of basically three parts: a radio frequency front-end, a physical layer and a medium access control layer. On top of this an application is running that uses the radio. The complexity of the three parts is not divided equally. The behavior of the RF front-end is very static and is therefore often implemented in hardware. The MAC layer is also not that complex, since it does not involve difficult data processing. It only analyses the content of a physical layer payload and passes part of that on to the application (and vice versa). The physical layer, on the other hand, demands quite some processing power. This layer has to deal with, amongst other things, digital signal processing and digital error correction of frames that are going in and out at rather high rates.

3.1 Problem statement

The architecture of the platform that NXP has in mind is chosen outside this thesis, so the number and type of processor cores available can not be changed. Hence, verifying whether or not this platform is capable of running multiple wireless LAN radios basically comes down to finding a mapping of the WLAN application and selecting appropriate scheduling policies. A good solution to this will have to make sure that the following is satisfied.

The platform should at least support four WLAN radios running simultaneously while satisfying the following conditions.
- Each running WLAN radio should always be able to process the incoming and outgoing data streams such that its throughput requirements are met; even when it is disturbed by other radios.

- Each running WLAN radio should always respond to events quickly enough such that its latency requirements are met; even when it is disturbed by other radios.

- (optionally) The platform is shared with other less critical applications.

These conditions must be satisfied in all possible situations (including worst-case). However, since there is no actual platform available, this will have to be proven theoretically.

### 3.2 Related work

Currently the most common way to analyze the performance of running an application on a certain architecture is by simulating the corner cases and verifying if in these situations the resulting performance meets the given constraints. The number of corner cases that need to be simulated depends mainly on the amount of resource sharing (processor and network). More resource sharing introduces more runtime dependencies, which will increase the number of corner cases. Defining and simulating all these corner cases quickly becomes unmanageable for MPSoC architectures. As a result, designing and verifying these systems tends towards more conservative static resource allocations, which make systems more predictable at the cost of efficiency and flexibility.

A more formal approach of performance verification is introduced by Marco Bekooij and others [6]. They describe a method based on synchronous dataflow models, for which the minimal throughput and resource budget can be derived formally. It takes into account the processing times of the tasks, the communication times of the interconnect and the arbitration policies. However, this method is designed for streaming applications and cannot deal with data dependent executions. The latter problem can be resolved by using predictable dynamic dataflows, but this still applies to streaming applications only. Unfortunately, a WLAN application is not streaming, because data is not received or transmitted continuously.

Another method is introduced by Kai Richter and others [7]. They propose a method that transforms the incoming and outgoing events of processors in such a way that local analysis per processor can be applied. This simplifies the overall analysis. However, it mainly focuses on the event model and not on the application itself. It is also not yet supported by many tools, making it difficult to use.

### 3.3 Approach

Because of the limitations of current verification methods, the choice has been made not to use a generic method. Instead, a method is designed for the WLAN application in particular.

This method is based on the Y-chart principle (see Figure 3.1). It basically consists of four steps. First a model of the WLAN application will be designed. This model consists of two parts, namely a task-graph and a set of constraints. The task-graph divides the work of the application in separate tasks and describes the data dependencies between the tasks. Each task in the graph has a number of properties like processing- and response times. For these properties a number of constraints are defined that can be used to verify if the throughput and latency requirements of the application are met.

Another model is designed for the architecture that will execute this application. This model gives a number of parameters that define the performance of the architecture. Then a mapping of the application model on this architecture model is defined, which will result in a set of equations that depend on the architecture parameters and provide the task-graph properties.

The resulting properties can then be tested against the constraints given by the application model. This will eventually show whether or not a given platform configuration meets the wireless
LAN requirements. By analyzing the effects that the platform variables have on these constraints, a configuration can be found that is capable of satisfying the requirements in all situations. Finally, this configuration will be tested in a simulated environment.

Since the physical layer of a WLAN radio is the most complex part, it mainly defines the performance of the radio. Therefore, the focus of this thesis will be mostly on the physical layer. The other layers and the applications that use the radio will not be dealt with in detail. And since this platform is not finished yet, there are some parts not defined yet. For instance, the processor cores of this architecture are connected with each other via an on-chip interconnect. How this interconnect is implemented and what its communications delays are is still unclear. Finding realistic numbers for this is out of the scope of this thesis.
Chapter 4

Application model

A WLAN radio, like most applications, consists of a number of tasks that each do a part of the work. A task receives some input data, processes it and sends out the result. The data that tasks receive can come from other tasks. Communicating the data between them can be done in various ways. In this thesis the task transaction level (TTL) interface will be used, which is based on transmission of data with tokens. Tokens are communicated via channels that run between tasks. These dependencies connect the tasks with each other in a graph. In this chapter such a task-graph will be constructed for the WLAN standard.

The tasks themselves are defined as so called actors. Actors are event driven and are only activated when a certain input condition holds, e.g. a specific number of input tokens are available. They also work in a well described manner. They consume tokens in the beginning of their execution, process them accordingly and produce the resulting output tokens. After this they are terminated and only executed again when the input condition holds again.

4.1 Definitions

A task-graph is a visual representation of the tasks that define an application and the dependencies between them. Each task in this graph does a certain amount of work in a certain amount of time. The complexity of the work can depend on the content of its input tokens, so processing it can take a variable amount of time. This means that the task has an average, best-case and worst-case processing time. This study want to verify if a system is able to deal with all possible situations and therefore the worst-case times will be used.

The processing time of a task is defined as the time between the consumption of the input tokens and the production of the output tokens. The longest possible gap between these moments is the worst-case processing time, which is indicated with the variable $P_x$, where $x$ is the name of the task. This includes the worst possible amount of time the task can be preempted by other tasks. In other words, it is measured according to the "wall-time".

The processing time of a task is defined as the time between the consumption of the input tokens and the production of the output tokens. The longest possible gap between these moments is the worst-case processing time, which is indicated with the variable $P_x$, where $x$ is the name of the task. This includes the worst possible amount of time the task can be preempted by other tasks. In other words, it is measured according to the "wall-time".

The processing time of a task is defined as the time between the consumption of the input tokens and the production of the output tokens. The longest possible gap between these moments is the worst-case processing time, which is indicated with the variable $P_x$, where $x$ is the name of the task. This includes the worst possible amount of time the task can be preempted by other tasks. In other words, it is measured according to the "wall-time".

A task is activated as soon as its input condition holds. However the time between this activation and the actual execution of the task might not be the same. For instance, a processor might be working on something more important which has to be finished before it can start on this task. This delay is defined as the startup time of a task and indicated with the variable $S_x$.

The sum of these times, i.e. the time between the activation of the task and the actual completion of it, is defined as the response time and indicated with the variable $R_x$. It is computed according to Equation 4.1.

$$ R_x = S_x + P_x \quad (4.1) $$

The channels that transport the data between the tasks also need some time for doing this. If a channel uses a shared communication medium then the time it takes to transmit a specific token...
might depend on the environment. The worst possible communication time is indicated with the variable $C_y$ with $y$ the type of the token.

Most of the tasks in this model depend on tokens that arrive periodically. This means that although the tasks are event driven they are also periodic. A task must be capable of processing its tokens at the same rate as they arrive, so the arrival rate defines the rate that the task must be executed with. In other words, a task has a maximum period in which it should be executed one time. In other words, it is the maximum time between two consecutive executions of the same tasks. This will be indicated with $T_x$ where $x$ is the name of the task. In order to guarantee a certain throughput, this $T_x$ must be less than the arrival rate of the input tokens.

This does not only hold for tasks, but also for channels. The tokens that are communicated over it are produced at a certain rate. The channel must be capable of transferring them at the same rate. The maximum time it takes to communicate a token one time is also defined by the period $T_y$ where $y$ is now the type of the token that is transmitted.

A summary of these variables is given in Table 4.1.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_x$</td>
<td>The worst-case startup time of task $x$, which is the time between activation of the task (i.e. the arrival of the input tokens) and the actual execution of a task.</td>
</tr>
<tr>
<td>$P_x$</td>
<td>The worst-case processing time of task $x$, which is the largest possible gap between the actual execution and completion of the task. This includes the time that the task is preempted by other tasks.</td>
</tr>
<tr>
<td>$R_x$</td>
<td>The worst-case response time of task $x$, which is the time between the activation of the task and the time that the task is completed.</td>
</tr>
<tr>
<td>$C_y$</td>
<td>The worst-case time required to transmit a token of type $y$ over a channel.</td>
</tr>
<tr>
<td>$T_x$</td>
<td>The maximum time between two consecutive executions of task $x$ or two consecutive transmissions of token $x$.</td>
</tr>
</tbody>
</table>

### 4.1.1 Legend

In this thesis the graphs will be depicted with three different symbols, which are shown in Figure 4.1.1. Tasks are defined by an oval with the name of the task inside. The channels are depicted with a directed arrow with a label that shows the type of the token it transports.

![name](a) Task | ![type](b) Channel | ![dashed](c) Trigger

There are also tasks that do not use input tokens to activate them. Normally these tasks will run indefinitely since their input condition will always be satisfied. If these tasks are not supposed to do this and need to be activated by another task then they can be connected with that task with a trigger line. This is depicted as a dashed line. It can be compared to a normal channel that transmits an empty token. If a task is activated by a trigger then it will run until it suspends itself. Hence, a task can execute for a number of periods and then wait for another trigger. The absence of input channels and trigger lines indicates that such a task will run continuously.
4.2 Task-graph

For each of the states a WLAN radio can be in, a number of tasks can be defined that need to be executed before the next state can be triggered. For some states the tasks depend on whether a frame is being transmitted or received. Table 4.2 shows a list of the seven states of a WLAN radio and the directions they apply to. Only the Header and Data Processing states are applicable to both incoming and outgoing frames.

<table>
<thead>
<tr>
<th>State</th>
<th>Down</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Detection</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Gain Control</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Time Synchronization</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Frequency Synchronization</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Header Processing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data Processing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Preamble Transmission</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

In the rest of this section each of these states will be analyzed and in the process a task-graph will be build incrementally. The result will be a single task-graph that represents the work to be done by a WLAN radio. For each task the maximum period will also be defined, which will basically define the throughput of the entire graph.

4.2.1 Data processing

The first and most important state of a WLAN radio is the actual processing of the payload, which is done in the Data Processing state. An overview of the required processing is already given by the standard in Figure 4.1. It depicts two linear pipelines, one for processing incoming OFDM symbols and another one for outgoing. Several of the smaller components can be combined in a single task. For example, all the tasks from the antenna up to the AFC clock recovery help in converting an incoming analog signal into a stream of digital samples. They can therefore be grouped into a single task called Rx. This can also be done for the other components in the overview, which results in the following list of tasks for the incoming data OFDM symbol processing.

- **Rx** receives analog signals at a specified frequency band and converts them into digital samples.
- **Demod** removes the guard interval of the OFDM symbol, extracts the sub carriers using a FFT, demodulates the samples of each sub carrier into soft bits and finally serializes them in the correct sequence.
- **Decode** de-interleaves the received soft bits and decodes them into data bits.
- **MACi** checks whether the frame was received correctly and transmits an acknowledgement frame when required.

For the outgoing data OFDM symbol processing the following tasks are defined.

- **MACo** constructs the MAC layer frame and sends the bits into the outgoing pipeline.
- **Code** uses a forward error correction scheme to code the bits. It also interleaves them.
- **Mod** distributes the coded bits over the sub carriers and modulates them into digital samples. The sub carriers are then passed through an Inverse FFT to get the time-domain samples. Finally the guard interval is added to construct a complete OFDM symbol that is ready to be transmitted over the air.
• **Tx** reads the digital samples, converts them into analog signals and transmits them at a specified base frequency.

Figure 4.1: 802.11a overview

**Rx and Tx**

The Rx task is responsible for converting the analog signal received by the antenna into digital samples. Digital samples are complex numbers, and they are modeled with a token of type SAMPLE. The sampling rate of a WLAN radio is 20 million samples per second, so the Rx task should produce one SAMPLE token every 50 ns. In order to sustain this throughput a system that implements the Rx task should make sure that the period of the task does not exceed this time. The Tx task does the reverse and consumes SAMPLE tokens that are converted into analog signals and transmitted via the antenna. The maximum periods of the tasks are defined as follows.

\[
\begin{align*}
T_{Rx} &\leq 50 \text{ ns} \\
T_{Tx} &\leq 50 \text{ ns}
\end{align*}
\]  

(4.2)

**Demod and Mod**

The Demod and Mod tasks are responsible for converting the incoming blocks of samples (i.e. OFDM symbols) into blocks of bits and vice versa. An OFDM symbol consists of 80 samples; hence the Demod task consumes 80 SAMPLE tokens. Depending on the modulation scheme it produces between 48 and 288 bits. These bits are so called soft bits, which represent a single bit with more information that can be used to improve the decoding process. This set of bits is encapsulated in one token of type SOFTBITS that can contain at most 288 soft bits. The Demod task produces one such token every period. The Mod task performs the reverse actions and consumes one SOFTBITS token and produces 80 SAMPLE tokens. Both tasks have to process OFDM symbols, which have a period of 4 \(\mu\)s, so the throughput these tasks have to support is at least 250,000 OFDM symbols per second.

\[
\begin{align*}
T_{Demod} &\leq 4 \text{ \(\mu\)s} \\
T_{Mod} &\leq 4 \text{ \(\mu\)s}
\end{align*}
\]  

(4.3)

These constraints seem a bit strict. After all, there is the short interframe space of 16 \(\mu\)s following a frame, which gives some extra time for processing the frame. If, for instance, the
throughput of the system is too little, then a number of symbols will pile up during the reception of the entire frame. When the last symbol is received the system still has 16 µs before it has to transmit the acknowledgement. In this time it can process all the remaining data symbols that couldn’t be processed in time. On the average, this gives a bit more processing time per data symbol.

Unfortunately, the largest possible frame has to be taken into account. According to the specifications a MAC layer frame can be at most 2.346 bytes long (see Figure A.7). With the lowest data rate of 24 bits per symbol this results in:

$$\left\lfloor \frac{22 + (2.346 \times 8)}{24} \right\rfloor = \left\lfloor \frac{18.790}{24} \right\rfloor = 783 \text{ OFDM symbols}$$

Per data symbol this adds $16/783 = 0.0204 \mu s = 20.4 \text{ ns}$ to the average processing time. At clock frequencies around 400 MHz this is barely 8 clock cycles, which is negligible. Taking into account that preparing an acknowledgement frame also takes time it is safe to claim that the throughput of at least 250,000 OFDM symbols per second must be guaranteed.

### Decode and Code

Like the other tasks the Decode and Code tasks are also the reverse of each other. The Decode task converts the soft bits of an OFDM symbol into actual data bits. Hence, it consumes one SOFTBITS token and produces one BITS token. The output token will, depending on the coding rate and the number of soft bits in the input token, contain between 24 and 216 bits.

The Code task does the reverse and consumes a BITS token and produces a SOFTBITS token. It actually does not produce soft bits but just normal bits, because the information of a soft bit is only available during demodulation. However, the same token type is used to make the model more symmetrical. Both tasks also have a period of $4 \mu s$, so they have the same throughput constraints as the Demod and Mod tasks.

$$T_{\text{Decode}} \leq 4 \mu s$$
$$T_{\text{Code}} \leq 4 \mu s \quad (4.4)$$

### MACi and MACo

The MACi and MACo tasks are respectively the last and first task in the pipelines. The MACi task consumes BITS tokens from the Decode task and computes their CRC checksum. If the checksums of all tokens in a frame correspond to the received checksum then the frame is valid and should be acknowledged.

In the other direction the MACo task produces the BITS tokens for the Code task. Each token contains part of the complete frame. The MACo task can be activated by either a higher layer application or the MACi task. The MACi task will trigger it when an acknowledgement frame must be transmitted.

The BITS tokens arrive at a rate of 250,000 per second, so the MACi task should have a period of at most $4 \mu s$. The MACo task should produce the BITS tokens at the same rate, so it has the same throughput constraint.

$$T_{\text{MACi}} \leq 4 \mu s$$
$$T_{\text{MACo}} \leq 4 \mu s \quad (4.5)$$

### Graph

The above tasks can be put into a graph. Based on their input dependencies some tasks are connected with each other via channels. The resulting task-graph is depicted in Figure 4.2. The
pipeline structures in both directions are clearly visible. The MACi and MACo tasks are connected with a trigger channel.

The channels in this graph must be capable of dealing with the throughput of the tasks. There are only three types of tokens communicated. SAMPLE tokens are produced at a rate of one every 50 ns. The SOFTBITS and BITS tokens have a throughput of one every 4 $\mu$s. The periods of communicating these tokens are therefore bounded as follows.

\[
\begin{align*}
T_{\text{SAMPLE}} &\leq 50 \text{ ns} \\
T_{\text{SOFTBITS}} &\leq 4 \text{ $\mu$s} \\
T_{\text{BITS}} &\leq 4 \text{ $\mu$s}
\end{align*}
\] (4.6)

**4.2.2 Header processing**

Before the data processing can start, the tasks need to know the data rate and length of the frame. All tasks except for the Rx and Tx tasks require this information. The information is defined in the header OFDM symbol that precedes the data OFDM symbols. The processing of such a symbol is similar to that of a normal data symbol coded and modulated with the lowest data rate, so the same tasks can be reused in this state.

For incoming frames the Demod task knows that the first set of samples it consumes is the header symbol. It will demodulate this symbol and outputs a SOFTBITS token. The first SOFTBITS token the Decode task receives is also processed as a header symbol and it will produce an RXVECTOR token that contains the extracted information. This token is a trigger that tells the MACi task that data is coming. Besides the MACi task the token is also needed by the Demod task. Both tasks use it to compute the number of symbols in a frame, so they know when the last symbol has been processed and know when to stop.

In the opposite direction, the MACo task will produce a TXVECTOR token when it wants to transmit a frame. The Code task will use this token to construct a SOFTBITS token that represents the header symbol. Since this is the first symbol the Mod task will receive it knows that it is the header symbol and will modulate it accordingly. The following symbols are modulated using the information in the TXVECTOR token. After this token has been produced the MACo task will start producing the BITS tokens.

No extra tasks are added during this state, only the functionality of existing tasks is changed. Since processing a header symbol is virtually the same as processing a data symbol, the constraints are also the same as in the Data Processing state. Only some extra channels for communicating the RXVECTOR and TXVECTOR tokens have to be added to the task-graph. The result is depicted in Figure 4.3.

There is a limit to the number of frames that can be received and send per unit of time. A frame has a minimal length of one header symbol and one data symbol. Together with a preamble this takes 24 $\mu$s to transmit. After this period a station has to wait at least two SIFS periods and one ACK frame before it can send another frame. The minimum length of an ACK frame is also 24 $\mu$s, so the time between two frame transmissions will be at least $24 + 16 + 24 + 16 = 80$ $\mu$s. This means that TXVECTOR tokens can be produced at a rate of at most 12,500 per second.
However, for an observing station frames can arrive at a higher rate. It receives both the data and acknowledgment frames in the described situation. Between these frames is a minimum gap of 40 $\mu$s; hence RXVECTOR tokens are produces at a rate of at most 25,000 per second. The throughput of the channel must support this throughput, so the periods of the token transmissions are bounded as follows.

\[
T_{RXVECTOR} \leq 40 \mu s \\
T_{TXVECTOR} \leq 80 \mu s
\] (4.7)

### 4.2.3 Synchronization

For correct reception of a frame the system must be synchronized in the time and frequency domain. Two tasks are introduced for this: TSync for the time-synchronization and FSync for the frequency-synchronization.

**TSync**

The TSync task aligns the system to short training symbols and tries to find the end of the short training symbol block. Every period a short training symbol of 16 samples is validated. Hence, every period 16 SAMPLE tokens are consumed. If it is a STS then the TSync task assumes it is still processing a preamble and waits for another STS. On the other hand, if is not a STS then there are two choices. First of all, the task might be executing for the first time after activation, so it can be fairly certain that the detection was a false positive. However, if the previous executions succeeded, then it assumes that it reached the end of the STS-block and has processed the first part of the guard interval of the long training symbol block. At this point it will put the system in the Frequency Synchronization state by triggering the FSync task.

The number of times this task gets executed before it reaches the end of the STS-block depends on the location within the block where the preceding stages ended. It will, however, never exceed 11 because there are only 10 short training symbols and the last check processes the beginning of the guard interval. Furthermore, during a successful run it is executed at least two times. Once to verify that it is in a STS-block and once to verify the end of the block.

Since the task processed short training symbols its period will be equal to this, i.e. 0,8 $\mu$s.

\[
T_{TSync} \leq 0,8 \mu s
\] (4.8)
CHAPTER 4. APPLICATION MODEL

The FSync task uses the subsequent long training symbol block to synchronize in the frequency domain. It uses the same principles as data-symbol demodulation. It extracts the sub carriers and the resulting values are used to correct the frequency errors. One long training symbol is 64 samples or 3.2 µs long. The block consists of two long training symbols and is preceded by a guard interval of 1.6 µs. In total the entire block is 160 samples long, thus the FSync task consumes 160 SAMPLE tokens over a period of 8 µs.

FSync has to process two long training symbols. On average it has 4 µs per symbol, which is the same as the Demod task. The task can process these two symbols in two ways. It can wait for both symbols to arrive, combine them to eliminate some noise and do a single computation on the result. This implies that the task only runs once, and has to wait for the entire long training symbol block before it can start, which is 8 µs long. It will have to consume 160 SAMPLE tokens in one period.

On the other hand, it can also do one computation with the first symbol, adjust the system parameters and fine-tune them with a computation on the second symbol. The first symbol arrives 1.6 + 3.2 = 4.8 µs after the end of the short training symbol block. However, the guard interval is twice the size of the guard interval a data OFDM symbol and is constructed by taking the last 1.6 µs (or half) of a long training symbol. So after only 3.2 µs a complete symbol can be constructed by using the guard interval and the first half of the long training symbol. This also holds after 4.0 µs which is exactly the same as the period of the Demod task; resulting in a guard interval of 0.8 µs. Because this second method makes sure the task is executed earlier, less buffering is required.

In this model the second approach is used. The task is executed twice and the input data size is 80 SAMPLE tokens per period. The processing time of the FSync task is bounded by the period of 4 µs. After the task is finished for the second time it puts the system in the Header Processing state by triggering the Demod task.

\[ T_{FSync} \leq 4 \mu s \] (4.9)

The task-graph with both the TSync and FSync tasks and their dependencies added is depicted in Figure 4.4. The channel from the Rx task uses a de-multiplexer, since more than one task is exclusively consuming the SAMPLE tokens, which will be discussed later on.
4.2. TASK-GRAPH

4.2.4 Gain control
Before any processing of a frame can start the strength of the incoming signal must be adjusted to get the highest accuracy out of the hardware. This is done in three incremental steps. For this the Gain task is introduced. During each step the samples of one short training symbol are analyzed and the gain level of the analog amplifier is adjusted accordingly. The processing of the short training symbols is mostly the same as during the detection phase.

Although the gain is set in three steps the task only has to be executed twice because the first step can be completed with the information from the last detection step. Only two additional short training symbols need to be processed by the Gain task. It consumes 16 SAMPLE tokens each period of 0.8 µs. After two periods, when the gain is adjusted three times, the Gain task puts the system in the Time Synchronization state by triggering the TSync task.

The quality of the gain control depends highly on the performance of the task. When a set of samples has arrived the gain should be calculated as soon as possible, because new samples arrive while calculating this value. These newly arrived samples are still based on the old gain level, so the longer the delay the more samples with the old gain level will be used.

This effect is illustrated in Figure 4.5. In Figure 4.5(a) you see the status of the buffer when just enough samples have arrived for the first computation. The task will start calculating a new gain level. This is finished at $\Delta t = 0.4$ (Figure 4.5(b)), when already 8 new samples have arrived using the old gain level. The new gain level is now set and the next samples will use it. Then at time $\Delta t = 0.8$ enough new samples have arrived to calculate the next value. As displayed in Figure 4.5(c), half of the samples are based on the old value. Hence the newly calculated gain value is half as accurate is it should be.

![Figure 4.5: Buffer behavior during AGC](image)

It is clear that the Gain task cannot afford any delay, and therefore needs to be executed as soon and as fast as possible. To increase accuracy the startup time $S_{Gain}$ must be minimized and the processing time must stay well below 0.8 µs.

$$T_{Gain} \leq 0.8 \, \mu s$$  \hspace{1cm} (4.10)

4.2.5 Frame Detection
The WLAN system requires a detection mechanism for discovering whether or not a frame is coming. This is done by the Detect task. This task reads the samples of one short training symbol (16 SAMPLE tokens) each period of 0.8 µs. It verifies if it is a short training symbol and triggers the next state accordingly.
The Detect tasks must be capable of reading all the incoming samples but it does not need to process them all. The more samples it uses and thus more checks are done, the faster it gets a positive result. For instance, if the task checks every short training symbol, then it would get a positive result at the first check after the first 16 samples of the preamble have arrived. In a worst-case situation the previous check might be right before this moment, which is for example when the first 15 samples of the preamble have been received. This check will fail because it misses one sample. The next check will use the next 16 samples and succeeds. Hence, with this granularity the latest time a positive result will occur is after the first two symbols (given that the signal is good enough). This is visualized in Figure 4.6(a). If, on the other hand, the check is done every two short symbols (e.g. every 32 samples), then the worst-case situation gives a positive result at the third STS as visualized in Figure 4.6(b).

\[ T_{Detect} \leq 4.8 \, \mu s \] (4.11)

However, the algorithm of the Gain task is for the largest part similar to that of the Detect task. The Gain control task has to complete its calculations as soon as possible and within 0.8 \( \mu s \) (see Equation 4.10). Since the Detect task will have about the same processing time, it probably never comes close to its maximum allowed processing time. This means the longer the period between the checks the more idle time the system gets.

The Detect task is the last task that needs to be added for the downstream side of the model. The stages that deal with detecting a frame, synchronizing and processing it are all modeled in the graph depicted in Figure 4.7.

4.2.6 Preamble transmission

Until now the focus was mainly on the downstream side. It is, in contrast to the upstream side, more complex and has more tasks associated. This section will highlight the states relevant for the upstream side. Table 4.2 shows that these states are: Header Processing, Data Processing and Preamble Transmission.

The first two states are already elaborated in the previous sections and can be summarized as follows. A frame transfer is initiated by the MACo task, which produces a TXVECTOR token. As a result a SOFTBITS token is produced by the Code task, which is converted by the Mod task into a set of 80 SAMPLE tokens. This is followed by the Data Processing state, during which
the MACo task produces BITS tokens. The Code task will consume these and will again produce a SOFTBITS token, which is consumed by the Mod task that produces another stream of 80 SAMPLE tokens. The SAMPLE tokens are consumed by the Tx task that transmits them over the air. The MACo task will continuously send out BITS tokens until the frame is completed and afterwards put the system back into the Frame Detection state.

However, frames must be prepended by a preamble. This preamble contains no data depending on the content of the frame, so it can be precomputed once and transmitted unchanged for each frame. This is done in the Preamble Transmission state, which is triggered by the MACo task when it produced the TXVECTOR token. A new Preamble task consumes this TXVECTOR token each period and as a result produces a stream of 320 SAMPLE tokens, which are consumed by the Tx task.

For the throughput it does not matter much how long it takes to transmit all the 320 samples as long as they are all produced within 16 $\mu$s. For the latency however it is important, which will be discussed later on. Fortunately, the data the Preamble task produces is static and independent on the frames content, so there is not much processing required. This means that the worst-case processing time will be very low and probably never come close to this constraint.

$$T_{Preamble} \leq 16 \, \mu s$$  \hspace{1cm} (4.12)

### 4.2.7 Final task-graph

The entire task-graph is now complete. Figure 4.8 shows all the tasks required for a WLAN radio including the communication channels. The main part of the graph is rather similar for incoming and outgoing symbols. The major difference is in the handling of the preamble. For incoming frames the preamble is used by a number of tasks, while for outgoing frames it is not processed at all and just statically transmitted.
Figure 4.8: Final task-graph
4.3. CONSTRAINTS

TTL limitations

There are two channels in the graph that are connected to either multiple consumers or multiple producers. One of them comes from the Rx task and the other goes to the Tx task. These channels are bit different than ordinary channels.

Normally, when multiple consumers are reading from the same channel, then a token only gets removed from the channel after all consumers have processed the token. If some of the consumers are inactive, i.e. waiting for a token on another channel, the arriving tokens will not be removed and eventually the channel can get full.

In this task-graph only one of the tasks connected to the Rx task will be active exclusively. The other task will not be running and therefore not read tokens from the channel. With a normal channel the buffer will get full in no time. Therefore this channel should behave in such a way that the tokens are only transmitted to a limited number of tasks. This is also called a multi-cast channels in contrast to the normal broadcast channels. The TTL interface currently does not support such a channel.

One way to support this can be with a virtual de-multiplexer task that consumes the tokens from the producer task and only produces tokens for the tasks that need to receive the tokens. It will look like the graph depicted in Figure 4.9. Similarly a multiplexer task can be implemented to do the reverse, where only a selected number of tasks can inject tokens on the channel.

These multiplexer tasks should be purely virtual. If they would be implemented in a real application then they would have to be constantly active and thus consume extra resources. Instead, they should be implemented more intelligently by changing the transmission logic of the channel itself. In the coming analysis it is assumed that such an implementation is possible.

4.3 Constraints

Now that the task-graph is defined, constraints can be added to the model that will make sure the system that will execute this model can support the WLAN standard. There are two quantities that are relevant to this, namely the throughput and the latency.
4.3.1 Throughput

The throughput is already discussed in the previous sections, which resulted in some constraints on the periods of the tasks. In short it means that the Rx and Tx tasks, which handle samples, have a maximum period of 50 ns. The other tasks can be divided based on their input tokens. The Detect, Gain and TSync tasks deal with short training symbols which have a period of 800 ns. The Detect task has some slack and can have a maximum period of 4.8 $\mu$s. The other tasks have OFDM symbols as input, so their periods are bounded at 4 $\mu$s. This is summarized in Equation 4.13.

\[
\begin{align*}
T_{Rx} & \leq 50 \text{ ns} \\
T_{Tx} & \leq 50 \text{ ns} \\
T_{Detect} & \leq 4.8 \mu s \\
T_{Gain} & \leq 0.8 \mu s \\
T_{TSync} & \leq 0.8 \mu s \\
T_{FSync} & \leq 4 \mu s \\
T_{Demod} & \leq 4 \mu s \\
T_{Mod} & \leq 4 \mu s \\
T_{Preamble} & \leq 16 \mu s \\
T_{Decode} & \leq 4 \mu s \\
T_{Code} & \leq 4 \mu s \\
T_{MACi} & \leq 4 \mu s \\
T_{MACo} & \leq 4 \mu s
\end{align*}
\]  
\text{(4.13)}

Besides the processing times of the tasks, the communication channels are also relevant for the throughput of the system. They have to transmit the tokens that are produced at a certain rate. There are five tokens defined, with several rates. The SAMPLE tokens are produced at a rate of 20 million per second. This means the transmission period must be below 50 ns. The SOFTBITS and BITS tokens are produced every 4 $\mu$s and the RXVECTOR and TXVECTOR every 40 and 80 $\mu$s respectively. These constraints are listed in Equations 4.14.

\[
\begin{align*}
T_{SAMPLE} & \leq 50 \text{ ns} \\
T_{SOFTBITS} & \leq 4 \mu s \\
T_{BITS} & \leq 4 \mu s \\
T_{RXVECTOR} & \leq 40 \mu s \\
T_{TXVECTOR} & \leq 80 \mu s
\end{align*}
\]  
\text{(4.14)}

4.3.2 Latencies

In a task-graph several tasks can be connected to each other in a line. The arrival of a token at the first task will result, after a certain amount of time, in an output token at the last task. This delay is also called latency. For WLAN a number of such latencies can be defined, which have to be below a certain value in order to guarantee correct behavior of the radio.

Short interframe space latency

The most important latency for WLAN is the short interframe space (SIFS). It states that the transmission of the first sample of an acknowledgement frame (i.e. the preamble) must start 16 $\mu$s after the last sample of the corresponding frame has arrived at the station. In order words, the Tx task must be finished processing the first SAMPLE token of the acknowledgement frame 16
μs after the Rx task starts processing the last SAMPLE token of the incoming frame. In order to guarantee that this deadline is met, a global constraint is added to the model.

Several tasks are involved in the SIFS latency. For the downstream part these are Rx, Demod, Decode and MACi and for the upstream part MACo, Preamble and Tx. The entire flow is highlighted in Figure 4.10.

The total latency of this flow is the sum of the response times of the involved tasks and the communication times of all the transmitted tokens. This latency must be less than 16 μs in order to meet the SIFS deadline. It can be formulated as follows.

\[
R_{Rx} + C_{SAMPLE} + R_{Demod} + C_{SOFTBITS} + R_{Decode} + C_{BITS} + R_{MACi} + R_{MACo} + C_{TXVECTOR} + R_{Preamble} + C_{SAMPLE} + R_{Tx} \leq 16 \mu s
\] (4.15)

If a system that implements the WLAN protocol cannot satisfy this constraint then it cannot guarantee that all frames are acknowledged in time and therefore the performance of the system is unpredictable.

### Header-loop

Another latency that has an indirect influence on the SIFS latency is the header-loop latency, which is the time between the processing of the header OFDM symbol and the moment the contained information is available. This information must be decoded and available before the Demod task begins on the first data OFDM symbol or otherwise the Demod task has to wait before it can start, which introduces an extra delay in the overall SIFS latency. This is a big problem when a frame is only one symbol long.

The tasks involved in this are highlighted in Figure 4.11. It shows a feedback loop between the Demod and Decode tasks. In the Header Processing state the Demod task produces a SOFTBITS
token containing the demodulated data of the header symbol. The Decode task consumes this
token and produces an RXVECTOR token that contains the decoded information, like the length
and data rate. This information is needed by the Demod task before it can start processing the
following OFDM symbol which is one period of 4 µs later. This can be formulated as follows.

\[ P_{\text{Demod}} + C_{\text{SOFTBITS}} + R_{\text{Decode}} + C_{\text{RXVECTOR}} \leq 4 \mu s \] (4.16)

If a system cannot satisfy this constraint then the SIFS latency must be increased with the
extra delay this causes. A consequence of that can be that the SIFS latency crosses its limit and
also fails its constraint.

**Gain control accuracy**

There are also smaller, less critical latencies in the task-graph that have impact on the performance
of the radio. In the Gain Control state SAMPLE tokens from the Rx task must be processed by
the Detect and Gain tasks as soon as possible in order to minimize the delay between setting the
gain level and actually processing samples with that new level. The longer this delay, the less
accurate the gain levels will be.

The first gain level is set by the last Detect task. The worst-case latency between the processing
of a sample by the Rx task and setting of the gain level based on that sample by the Detect task
cannot exceed two STS periods. Otherwise, the Rx task will have processed too much samples
with the old gain level that the following two Gain tasks use old values, making them useless.
This also holds for the Gain task. It has to be ready processing a sample within two STS periods
or else the first TSync execution uses samples that are not using a fine tuned gain level. This can
be formulated as follows.

\[ R_{\text{Rx}} + C_{\text{SAMPLE}} + R_{\text{Detect}} \leq 1.6 \mu s \]
\[ R_{\text{Rx}} + C_{\text{SAMPLE}} + R_{\text{Gain}} \leq 1.6 \mu s \] (4.17)

**Acknowledgement frame**

The last constraints involve the outgoing acknowledgement frames. The latency between activating
a frame transmission and the actual start of the preamble is already covered by the SIFS latency
constraint. However, the samples of the header OFDM symbol must be ready for the Tx task
once it has consumed all the preamble tokens.

The TXVECTOR triggers both the Preamble and Code tasks. While the Preamble task is
producing the samples of the preamble, the Code task will create a SOFTBITS token based on the
information in the TXVECTOR token. This token is consumed by the Mod task which produces
the SAMPLE tokens. These must be ready before the Tx task has consumed the last sample of
the preamble. This can be formulated as follows.

\[ R_{\text{Code}} + C_{\text{SOFTBITS}} + R_{\text{Mod}} + C_{\text{SAMPLE}} \leq R_{\text{Preamble}} + C_{\text{SAMPLE}} + R_{\text{Tx}} + 319 \times 50 \text{ ns} \] (4.18)

Of course, after the header symbol has been consumed completely by Tx, the samples of the
first data symbol must be ready for transmission. This means a second part for this constraint,
which is formulated as follows.

\[ R_{\text{MACo}} + C_{\text{BITS}} + R_{\text{Code}} + C_{\text{SOFTBITS}} + R_{\text{Mod}} + C_{\text{SAMPLE}} \leq R_{\text{Preamble}} + C_{\text{SAMPLE}} + R_{\text{Tx}} + 399 \times 50 \text{ ns} \] (4.19)
4.4 Summary

In this chapter a model of a WLAN radio is defined. It consists of a task-graph that divides the work into 13 tasks (see Figure 4.8). Each task in this graph has a number of worst-case properties, namely the processing time \( P_x \), startup time \( S_x \), response time \( R_x \) and period \( T_x \). The channels that communicate the tokens between the tasks also have a worst-case communication time \( C_y \) and period \( T_y \).

Based on these properties a set of constraints are defined that limit the throughput and latencies such that the wireless LAN requirements are met. The throughput constraints are presented in Equations 4.13 and 4.14. The latency constraints are presented in Equations 4.15, 4.16, 4.17, 4.18 and 4.19. The most important of these is the latency constraint on the short interframe spacing.
Chapter 5

Architecture model

The WLAN application will be executed by the platform that NXP is developing. This platform has a number of processor cores and on each of them a number of tasks will be running.

However the platform contains only four processors, while task-graphs normally contain more tasks. The WLAN task-graph for instance already has 13 tasks. This implies that multiple tasks will have to share a single processor. When multiple instances of an application are running, then the processor will be shared by even more tasks.

Sharing a processor with multiple tasks will require some form of multi-tasking. This is often regulated by a scheduler according to some policy. In this chapter the scheduling policies of the four processors will be highlighted.

However, a few variables must be defined first. First of all, some processors in the platform can execute software programs. The tasks mapped on them will consist of a sequence of instructions. The number of instructions a task needs is denoted by the variable $I_y$, with $y$ the name of the task. The frequency at which these processors execute their instructions is denoted by the variable $F_p$, where $p$ is the processor identifier. The speed of the network will also be denoted by the variable $F$, but then the unit will be bits per second instead of instructions per second.

5.1 Interconnect

The various processor cores of an MPSoC have to communicate with each other to actually do something useful. There are various ways to connect processors with each other. A bus is very common and has been used for many years, but is unfortunately not very scalable.

Today more and more systems use a packet-switched network-on-chip (NoC). This can be compared with an IP network implemented on silicon. Data between processors and other blocks is transmitted in packets that are transported by routers to the correct destination. This kind of interconnect is very scalable and there are even implementations possible that can guarantee the latency of the network.

It is expected that the NXP platform will utilize a NoC. However, at this moment it has not been decided yet, so there are no actual performance numbers available. However the network of this platform will probably be able to guarantee a certain latency, so it can be perceived as if the processors are all connected to each other with direct point-to-point connections that are not influenced by the activities on other processors. This makes the analysis of the performance later on much simpler.

The tokens that are communicated between tasks are transported by channels. How these channels are actually implemented falls out of the scope of this thesis. It is assumed that all channels will be handled by the interconnect, so there is no difference in whether or not tasks are mapped on the same processor. Furthermore, multiple instances of the same channel will be mapped on the same logical channel. In other words, if a channel is occupied by one radio then it can stall the communications over the same channel in other radios.
The speed at which the logical channels transport the data will be equal for all channels and is denoted with the parameter $F_{\text{Network}}$.

5.2 RF front-end

The first processor core of the architecture is the Radio Frequency (RF) front-end, which is a set of hardware components capable of sampling analog signals via an antenna into digital numbers. The antenna picks up the radio signals from the air, which are amplified and passed through several analog filters before they are eventually converted to digital samples by an analog-to-digital converter (ADC). Of course the reverse direction is also possible.

At which frequency band the RF front-end needs to sample depends on the currently active radios. Most wireless applications can communicate via a range of predefined channels at various carrier frequencies. This will be one of the configurable options of the front-end. Furthermore the level of amplification should be variable to allow the full utilization of the ADC spectrum.

In order to support multiple radios simultaneously, several things will be required. First of all, there are at least two antennas needed, since some radios might be receiving data while others are transmitting; this cannot be done with one antenna. The easiest but most expensive implementation is to use a separate analog circuit for each radio. So with two radios there are two ADCs, two DACs and accompanying filters. This is obviously too expensive and not scalable when support for more radios is requested.

Another option is to sample all the individual channels as one big channel. Most wireless applications have a range of channels that are allocated next to each other, so a single circuit can be implemented that samples the entire frequency band that covers all the channels. The resulting stream can be split into the separate channels using digital processing (like fourier transformations). This way multiple channels can be received with only one antenna and one (rather advanced) ADC. Of course this also applies to the transmitting part. Such a solution is quite complex, but gives the most flexibility. In this thesis it is assumed that this is the way it will be implemented.

5.3 EVP vector processor

The digital signal processing on the platform is performed by a vector processor called EVP (embedded vector processor), which is developed by the DSP Innovation Center of Philips Semiconductors. It is the productized version of the CVP [2] that was designed by Philips Research for supporting 3G standards. The EVP also incorporates capabilities for OFDM standards taken from the OnDSP processor developed by Philips Semiconductors. The EVP has been proven to be highly versatile in other areas as well.

The EVP has a powerful instruction set for vector manipulations. Its 256-bit vector registers can (depending on the used instruction) be interpreted as 8 scalar elements of 32 bits, 16 scalar elements of 16 bits or 32 scalar elements of 8 bits. The registers can be manipulated with two types of instructions, namely inter- and intra-vector instructions. With inter-vector operations one or more vectors are treated as a whole, like addition of one vector to another. Intra-vector instructions are operations on the elements within a vector, like the sum of the elements or the maximum and minimal values in a vector. Besides vectors it also supports a large number of scalar instructions.

The processor has a very long instruction word (VLIW) architecture, making instruction level parallelism possible. For example, multiple scalar and vector instructions can be executed simultaneously and even registers can be loaded from or saved to memory in parallel with arithmetic instructions.

Because of the large vectors and the parallelism the processor is very suitable for converting the digital samples of the OFDM symbols into usable data. The corresponding algorithms can be
implemented efficiently on the EVP. For instance, a 64-point fast fourier transformation has been implemented in only 65 instruction cycles [3].

The frequency at which this processor processes its instructions is not defined yet. In a 90 nm CMOS process, the EVP core can run at 300 MHz. The core alone then dissipates around 0.5 mW/MHz and including a typical memory configuration 1 mW/MHz is dissipated. It is estimated that with a newer production process (e.g. 65 nm CMOS) the maximum frequency can be increased further. For this study the frequency is a parameter of the platform that can be changed later on and will be denoted with $F_{EVP}$.

### 5.3.1 Context switching

Running multiple tasks on a processor often implies the usage of context switching. However, like most digital signal processors, the EVP does not have features for supporting an advanced operating system, so it misses instructions to store and load the state of all registers efficiently. On this processor this has to be done manually, which makes context switching quite expensive.

The EVP has a lot of registers that need to be saved: at least 32 vector registers, 32 scalar registers and a number of control registers. If during one instruction cycle only one register can be stored or loaded, then it would take around 80 instructions for a full context store or reload. Luckily scalar and vector registers can be stored in parallel, which reduces the instruction count to an estimate of 40 instructions for a full store or reload. Hence, preempting one task and resuming another takes at least $2 \times 40 = 80$ instructions. This number will be denoted by $I_{preempt}$.

$$I_{preempt} = 80 \text{ instructions}$$

This is quite much. For example, if a context switching overhead of at most 10% is allowed, then with a clock frequency of 400 MHz this will result in a maximum context switch rate of:

$$\frac{400 \text{ MHz}}{I_{preempt}/10\%} = \frac{400 \text{ MHz}}{80 \text{ instructions}} = 500,000 \text{ context switches per second}$$

This is equal to a preemption every period of 2 $\mu$s. So the processor cannot switch tasks at the rate of short-training symbols (e.g. 800 ns) without having an overhead of more than 10%.

However, tasks can also be preempted temporarily by interrupt service routines. When an interrupt signal is received the processor will pause the current task and immediately start the ISR. Once the ISR is finished the tasks is resumed again. The storing and loading of the context is the responsibility of the ISR. This can help in the performance, because at compile time it is known what registers are used by the ISR, so only those will have to be stored and loaded. This reduces the number of instructions for context switching. For an average sized ISR this is estimated at 15 instructions for storing or loading a task’s context. In total an ISR will have an overhead of 30 instructions, which will be denoted by $I_{interrupt}$.

$$I_{interrupt} = 30 \text{ instructions}$$

With an overhead percentage of 10 and a processor frequency of 400 MHz this means that a task can be interrupted every 750 ns.

$$\frac{400 \text{ MHz}}{I_{interrupt}/10\%} = \frac{400 \text{ MHz}}{300 \text{ instructions}} = 1,333.333 \text{ context switches per second}$$

### 5.3.2 Scheduling Policy

In the model tasks are activated by the arrival of input tokens; hence the tasks are data-driven. This means that upon arrival of a token a task might get activated that has a higher priority than the currently running task. It is therefore expected that the operating system reevaluates its scheduling at that moment. This can be done in various ways.
A simple approach is to immediately pause the current task when a token arrives, enable the corresponding task and preempt the current running task if the priority is higher. However, with a high token arrival rate, like the SAMPLE tokens, it will be quite inefficient.

Another approach is to check for new tokens periodically. If a token has arrived since the last check then the corresponding task needs to be activated and the current task preempted if the priority is higher. With longer periods the system gets less responsive to events, but will also require less processing power.

In this study the second approach is used for which a timer is set with a specific interval. This will trigger an interrupt every period. As a result the currently running background task will be paused and its full context stored. A background task is any application that utilizes the idle time of the processor. This can be anything from a simple power saving idle-application to another less critical protocol.

Once this task’s context is stored the scheduler gets executed. This scheduler checks whether or not the input condition holds for all the tasks it serves. The tasks are checked in order of priority, with the highest priority first. Once a task’s input condition holds it will be executed immediately. If it is finished the next task will be checked until all tasks are processed. The previously preempted background task will then be resumed again. This can be described in pseudo code as follows.

```plaintext
preempt();
for each task i {
    if check(task[i]) {
        execute(task[i]);
    }
}
resume();
```

Before each task is executed its context does not have to be loaded, since they will be implemented as actors, which finish their execution every period. In other words they have no registers or such that need to be loaded from a previous execution.

There is only one requirement to the tasks that are handled by this scheduler, namely that they need to have limited worst-case execution time. Every time they are executed they will have to finish in a limited time, otherwise they may still be executing when the next timer interrupt occurs. This will be no problem as long as the worst-case processing time of the tasks that can be active in one period is less than the timer interval. This is something that should be guaranteed by the throughput constraints of the WLAN application model.

It is estimated that checking a task’s input condition requires 5 instructions. This means that if the scheduler has to deal with $X$ tasks then the number of instructions its needs to check all of them is at least equal to:

$$I_{\text{check}} = 5 \text{ instructions}$$

$$I_{\text{preempt}} + X \times I_{\text{check}}$$

(5.1)

Higher priority tasks

It might, however, be necessary for some tasks to be executed at the highest priority and with a shorter period. If these tasks are short enough then they can also be encapsulated in a separate ISR that is triggered by another timer (with a shorter period). This ISR behaves almost the same as the scheduler.

It will also pause the current task, which can now be the background task or a normal priority task, and stores its context. However not the full context is stored, since the tasks the ISR is responsible for will be small and so will the number of registers that they use. This means that
overhead of this ISR will be less than that of the normal scheduler. It is equal to the following, where \( X \) is the number of tasks it is responsible for.

\[
I_{\text{interrupt}} + X \cdot I_{\text{check}}
\]  \hspace{1cm} \text{(5.2)}

The tasks that can be handled by such an ISR must be able to execute with a limited amount of registers.

5.4 Software codec

The software codec is a processor that is responsible for the coding and decoding of blocks of bits, according to some algorithm. The processor will be designed totally from scratch, with the throughput and latency constraints of various standards like IEEE 802.11a in mind. The processor is not finished yet, but some of the algorithms that it will support are: Convolutional Viterbi decoding or the more advanced turbo (de)coder, block (de)interleaving and UMTS rate matching.

The architecture of the processor will probably consist of several configurable application specific hardware components each implementing a part of the coding and decoding process. For instance, a convolutional coder, with a configurable length and generator polynomial, or a shuffle unit that reads and writes memory blocks in configurable patterns. The components will be connected to each other in such a way that the output of one component can be configured as input for another. A chain of various components will result in the execution of a specific algorithm. Hence, each task mapped on this processor will be defined by a set of routing settings and component configurations.

Multiple standards will be implemented on this processor. There is however a maximum worst-case processing time that each task can have. For tasks that process incoming data (e.g. decoders) this maximum is indicated with \( P_{\text{decoder}} \) and for tasks in the other direction (e.g. coders) this is denoted with \( P_{\text{coder}} \).

5.4.1 Scheduling policy

The processor can deal with only one task at a time and does not support preemption. Tokens for tasks are stored in a single queue, which is processed on a first-come-first-serve basis. For each token the corresponding task is executed. Because of the throughput and latency constraints the WLAN tasks get a higher priority than the other tasks. This means that when a token for a WLAN task arrives in the queue it will be processed as soon as the current task is finished, before any other waiting token. It can be perceived like two separate queues, one with higher priority than the other.

5.5 ARM micro-processor

The last processor core in the architecture is an ARM micro-processor. It is responsible for controlling the incoming and outgoing flow of data bits. Most of this control belongs to the medium access control (MAC) layer. The operations done by this layer are relatively simple and do not include complex data manipulations. A standard general purpose processor is perfectly suitable for this work, so for this platform an ARM based processor is chosen.

This tasks that will be running on this processor will be at the end or beginning of a protocol pipeline. This often means that the rate at which tokens arrive is much lower than for those of lower level tasks (e.g. physical layer tasks). It is therefore possible to check and activate tasks every time a token arrives.

The processor will execute the instructions of its tasks at a certain frequency. What this will be is also uncertain for the moment. It will therefore be a parameter of the platform and denoted with \( F_{\text{ARM}} \).
5.5.1 CRC checksum acceleration

For many radios a checksum is included for the MAC layer payload such that a receiving station can verify if all bytes have been received correctly. These checksums are most of time cyclic redundancy check (CRC) checksums. Optimized implementations of this algorithm are available for a lot of processors. For ARM-based processors a 32-bit CRC checksum can be computed with an average of 1 bit per instruction. This means that the checksum of a block of 256 bits can be computed by a 300 MHz ARM processor in a little more than 850 ns.

However, a CRC computation is something that can easily be implemented in (or at least accelerated by) hardware. As described in [8] a speedup of more than 20 times faster can be achieved by extending the instruction set of the processor. It is assumed that for this architecture such an extension is possible and will result in a speedup factor of 16. This means that the CRC checksum of a block of 256 bits will be computed in around 16 cycles. At 300 MHz this means only 54 ns, which is much better than before.

5.6 Summary

In this chapter the architecture of the platform that NXP is aiming for is defined. It contains four processors that each have a specific way of dealing with the tasks they execute.

The RF front-end can sample a number of channels at once. The EVP will be executing software tasks. It has a limited scheduler that deals with tasks in a predefined order. The SWC executes its tasks based on the tokens in its input queues. And the ARM processor will execute the MAC layer tasks in software. It is extended with a number of instructions that accelerate the computation of CRC checksums.

There are also several parameters introduced that can change the behavior of the architecture. These are:

- $F_{EVP}$: The speed of the EVP processor in instructions per second.
- $F_{ARM}$: The speed of the ARM processor in instructions per second.
- $F_{Network}$: The speed of the network in bits per second.
- $P_{decoder}$: The worst-case processing time of a decoder task.
- $P_{coder}$: The worst-case processing time of a coder task.

These parameters will influence the properties of the tasks mapped on the various processors. How exactly will be dealt with in the next chapter.
Chapter 6

Mapping

Mapping a task-graph on an architecture is generally not a simple task. There is first of all the choice to implement a task in hardware or software. Implementing a task fully in hardware will make its execution more efficient, but less flexible. Having it executed by a (general purpose) processor will result in the exact opposite. Since the target is to make a software-defined radio, the aim is to have the most flexibility in the most efficient way.

Fortunately, the platform of NXP is designed in such a way that choosing a proper mapping is not that difficult. Each of the four processors is designed for a specific application domain, so for each task it is already clear which processor suits best.

The mapping is as follows. The RF front-end is capable of converting analog signals into digital samples and vice versa. In the model the Rx and Tx tasks are responsible for this, so these will be processed by the RF front-end. The embedded vector processor used in the architecture has powerful instructions for digital signal processing. The tasks that need this kind of processing are the Detect, Gain, TSync, FSync, Demod, Mod and Preamble tasks, so these will be implemented in software and executed by the EVP. The software codec is specialized in coding algorithms. Only the Code and Decode tasks do this, so these will be mapped on it. And the last two tasks, the MACi and MACo tasks, are rather simple tasks that mainly deal with controlling the incoming and outgoing streams of bits. The ARM micro-processor is perfectly suited for this.

The resulting mapping is depicted in Figure 6.1. The effects that each processor has on the properties of its tasks will be discussed in the remainder of this chapter.

![Task mapping diagram](image)

Figure 6.1: Task mapping

6.1 Definitions

The platform will be executing a number of radios simultaneously. The total number of WLAN radios running on the system will be designated with the variable $N$. The final goal is to support
four radios so $N$ will often be equal to 4. Sometimes it is necessary to differentiate between radios that are downloading or uploading a frame. The number of radios that are in a downloading state are denoted with $Nd$ and the number of radios in an uploading state are denoted with $Nu$. Radios that are in the frame detection state are counted as downloading. The sum of these two numbers should add up to $N$.

$$N = Nd + Nu$$

Finally, there are two static periods used by the standard, namely the lengths of short training symbols and OFDM symbols. These are respectively 800 ns and 4 $\mu$s long. These will be denoted with the variable $T_{short}$ and $T_{long}$.

$$T_{short} = 800\text{ns}$$
$$T_{long} = 4\mu\text{s}$$

### 6.2 Interconnect

The interconnect of the platform is responsible for transporting data between the processor cores. This means that the channels between tasks will be processed by this interconnect. Channels are mapped to logical channels. As explained, multiple instances of the same channel are mapped on the same logical channel. Each channel can transmit only one type of token. The speed at which these tokens can be transported (i.e. the throughput of the logical channels) is indicated with $F_{Network}$, for which a minimum can be computed based on the sizes of the tokens.

#### 6.2.1 Token sizes

In the model there are five different tokens communicated between the tasks, namely SAMPLE, SOFTBITS, BITS, RXVECTOR and TXVECTOR tokens. Tokens consist of a payload and a header. The header defines at least the type and size of the token. The size of the header is estimated at 32 bits. The size of the payload depends on the token type.

The SAMPLE token contains one digital sample, which is represented by a 16-bit complex number, i.e. its real and imaginary parts are both 8 bits wide. The SOFTBITS token contains at most 288 soft bits, which are represented by 4 "real" bits. A BITS token has a payload of most 216 bits. The RXVECTOR and TXVECTOR tokens both contain at least the length and data rate of the frame, which is estimated to be 32 bits in total. The various payload sizes and the corresponding token sizes (including the header) are listed in Table 6.1.

<table>
<thead>
<tr>
<th>Token</th>
<th>Payload size (bits)</th>
<th>Token size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLE</td>
<td>16</td>
<td>48</td>
</tr>
<tr>
<td>SOFTBITS</td>
<td>1,152</td>
<td>1,184</td>
</tr>
<tr>
<td>BITS</td>
<td>216</td>
<td>248</td>
</tr>
<tr>
<td>RXVECTOR</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>TXVECTOR</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

#### 6.2.2 Minimum network throughput

The communication times of these tokens are bounded by the throughput constraints given by the application model. Each token is produced at a certain rate, which results in a minimum required throughput of the logical channels that transport them. This minimum throughput is equal to $N$ times the minimum for a single instance, because multiple instances of a channel are mapped to the same logical channel, so tokens are transmitted sequentially.
It is also assumed that the speed at which these logical channels transmit their data is the same for all of them. This means that the throughput of each logical channel must be equal to the highest required throughput. For one radio this is equal to the value computed in Equation 6.1.

\[
\text{max} \left( \frac{\text{sizeof}(\text{SAMPLE})}{T_{\text{SAMPLE}}}, \frac{\text{sizeof}(\text{SOFTBITS})}{T_{\text{SOFTBITS}}}, \frac{\text{sizeof}(\text{BITS})}{T_{\text{BITS}}}, \frac{\text{sizeof}(\text{RXVECTOR})}{T_{\text{RXVECTOR}}}, \frac{\text{sizeof}(\text{TXVECTOR})}{T_{\text{TXVECTOR}}} \right) = \text{max} \left( \frac{48 \text{ bits}}{50 \text{ ns}}, \frac{1.184 \text{ bits}}{4 \mu s}, \frac{248 \text{ bits}}{4 \mu s}, \frac{64 \text{ bits}}{40 \mu s}, \frac{64 \text{ bits}}{80 \mu s} \right) = \text{max} (960 \text{ Mb/s}, 296 \text{ Mb/s}, 62 \text{ Mb/s}, 1.6 \text{ Mb/s}, 0.8 \text{ Mb/s}) = 960 \text{ Mb/s}
\]

Obviously, the highest throughput is needed for the SAMPLE tokens. This means that the minimum throughput of all logical channels must be equal to at least 960 Mb/s per radio. For \( N \) radios the minimum network throughput is as described in Equation 6.2.

\[
F_{\text{Network}} = N \times 960 \text{ Mb/s}
\]

6.2.3 Communication times

The worst-case time it takes to transmit a token depends only on the this network speed and cannot be disturbed by other channels. In other words, the worst-case communication time of a token can be described as follows.

\[
C_t = \frac{\text{sizeof}(t)}{F_{\text{Network}}}
\]

However, tokens that are produced at the same time and have to be transported via the same logical channel will have a longer worst-case time, because the tokens will have to wait for other tokens to be transmitted first. The last token in such a sequence will have to wait the longest and will be at most \( N \) times the time of one token. As will be shown later on, this will only happen to the SAMPLE token; hence the worst-case communication time of the SAMPLE tokens is as described by Equation 6.3.

\[
C_{\text{SAMPLE}} = N \times \frac{\text{sizeof}(\text{SAMPLE})}{F_{\text{Network}}}
\]

The worst-case periods of these transmissions can also be easily defined. The time between two consecutive transmissions of a token is equal to \( N \) transmissions of that same token (one for each radio), so this can be described as follows.

\[
T_t = N \times \frac{\text{sizeof}(t)}{F_{\text{Network}}}
\]

Obviously the worst-case period of the SAMPLE token is equal to the worst-case communication time.

6.3 RF front-end

The Rx and Tx tasks are implemented by the RF front-end. The Rx task produces SAMPLE tokens every period of 50 ns. The tokens are used by several tasks to which they need to be
communicated. Similarly, the Tx task does the reverse; it consumes SAMPLE tokens coming from either the Mod task or Preamble task.

The RF front-end can sample several channels at the same time, so from the outside it will look like every Rx or Tx task that is mapped on this processor runs fully in parallel with the others. The processing time per task is therefore the same as the total processing time for all tasks, which is at most the sample period of 50 ns. And since all channels are processed simultaneously, the startup time of these tasks is zero. This is summarized in Equation 6.5.

\[
P_{Rx} = T_{Rx} = 50 \text{ ns}  \\
S_{Rx} = 0  \\
P_{Tx} = T_{Tx} = 50 \text{ ns}  \\
S_{Tx} = 0
\]  

(6.5)

Because all these tasks finish at the same time, it means that the SAMPLE tokens will also be produced at the same time and will have to wait a little before they are transmitted (see Equation 6.3).

6.4 EVP vector processor

The EVP vector processor is responsible for the digital signal processing of the WLAN radio. This means the tasks Detect, Gain, TSync, FSync, Demod, Mod and Preamble will be processed by this core. The tasks on this processor consist of a sequence of instructions. The number of instructions required for each task is not known at the moment and will have to be estimated. Based on these numbers the minimum required operating frequency of the EVP and the worst-case processing and startup times can be computed.

6.4.1 Scheduling

The WLAN tasks mapped on the EVP can be divided into two sets based on their periods. Four tasks, Detect, Gain, TSync and Preamble, have a short period of 800 ns \((T_{short})\). The other three, FSync, Demod and Mod, have a long period of 4 \(\mu\)s \((T_{long})\).

Long-period tasks

The long period tasks will be handled by the scheduler, which is triggered by a timer. This timer is set to \(T_{long}\). When triggered, the scheduler will check three tasks for all the \(N\) available radios. This means the number of instructions required by the scheduler (see Equation 5.1) is equal to:

\[I_{preempt} + N \times 3 \times I_{check}\]

For \(N = 4\) this results in \(80 + 4 \times 3 \times 5 = 140\) instructions or 350 ns when using a clock frequency of 400 MHz. At this speed it is almost 9% of \(T_{long}\).

However, this can be improved. Of all the tasks belonging to one radio only one will be active at a time. For instance, a radio cannot have both the Demod and Mod tasks running at the same time, because it cannot receive and transmit at the same time. In other words, for each radio only the single task that can be active needs to be checked. This reduces the administrative overhead for the scheduler to:

\[I_{preempt} + N \times I_{check}\]

For \(N = 4\) this equals an overhead of 100 instructions; an improvement of almost 30%.
6.4. EVP VECTOR PROCESSOR

Short-period tasks

The short period tasks have a higher priority than the long period tasks. As explained, the context switching overhead of the scheduler is quite big. So, because these short-period tasks are quite small, it is chosen to implement them in an interrupt service routine triggered by a timer set at $T_{short}$.

This ISR will (like the scheduler) only check the tasks that can be active. This means that the number of instructions required by this ISR (see Equation 5.2) to check all radios is equal to:

$$I_{interrupt} + N \cdot I_{check}$$

For $N = 4$ this equals an overhead of $30 + 4 \cdot 5 = 50$ instructions. This takes 125 ns at an operating frequency of 400 MHz, which is almost 16% of the entire period.

6.4.2 Instruction cycle count

Estimating the number of instructions required by the tasks can be difficult. Fortunately, the capabilities of the EVP are partly based on the OnDSP and for that processor an IEEE 802.11a implementation is available. This can be used as a reference for the estimations.

A direct port of that implementation on the EVP is assumed to perform much better. First of all, the vector registers of the EVP are twice as wide as the OnDSP (16 words instead of 8) and, secondly, it has more support for parallelism. It is therefore assumed that the required number of instructions can be reduced by a factor two. Combined with a clock frequency that can be twice as high the EVP would theoretically perform four times faster. This assumption can be used to estimate the instruction cycle count and the processing times of the tasks mapped on the EVP.

Besides the tasks themselves some processing is required for the TTL primitives that handle the transmission of tokens. For the EVP these are implemented mainly in software. The primitives are based on pointers in a channel buffer, so whether a task uses one token or a batch of tokens in one command does not make it more complex. Reading or writing a token (or a batch of tokens) is estimated both at 10 instructions.

$$I_{ttlread} = 10 \text{ instructions}$$
$$I_{ttlwrite} = 10 \text{ instructions}$$

Demod, Mod and FSync tasks

In the reference implementation the processing of OFDM symbols is done by several modules. In Table 6.2 the instruction cycle counts of those modules are listed (see [3] and [9]). Apparently, processing an incoming symbol with the OnDSP requires 435 instruction cycles and outgoing symbols, which are easier to process, require 336 instruction cycles.

<table>
<thead>
<tr>
<th>Module</th>
<th>Incoming</th>
<th>Outgoing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol (de)mapping</td>
<td>40</td>
<td>23</td>
</tr>
<tr>
<td>Pilot generation</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>Pre/post scaling</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>Tracking</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Channel correction</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Phase-error correction</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>OFDM (de)mapping</td>
<td>30</td>
<td>34</td>
</tr>
<tr>
<td>AFC (i)FFT</td>
<td>169</td>
<td>156</td>
</tr>
<tr>
<td>GI insertion</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>Control</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>435</strong></td>
<td><strong>336</strong></td>
</tr>
</tbody>
</table>
As estimated, an EVP implementation will be able to do this with half the instruction cycles, so around 215 cycles for incoming symbols and 165 cycles for outgoing symbols. In this model incoming OFDM symbols are processed by the Demod task and outgoing symbols by the Mod task. Both tasks have one stream of input tokens and one of output tokens. This results in the total instruction count for the Demod and Mod tasks.

\[
I_{\text{Demod}} = I_{\text{ttlread}} + 215 + I_{\text{ttlwrite}} = 235 \text{ instructions}
\]
\[
I_{\text{Mod}} = I_{\text{ttlread}} + 165 + I_{\text{ttlwrite}} = 185 \text{ instructions}
\]

The required processing for the FSync task is similar to the Demod task without some modules like symbol de-mapping. The number of instructions is estimated at 220 instructions in total.

\[
I_{\text{FSync}} = 220 \text{ instructions}
\]

**Detect, Gain, TSync and Preamble tasks**

For the short-period tasks the estimated numbers of instructions are based on the used algorithms. The Detect task correlates a block of input samples with a reference short training symbol. This can be done with matrix multiplications, which works as follows.

Samples are stored as 16-bit complex numbers, so a full short training symbol of 16 samples fits in one 256-bit EVP vector register. This vector is first multiplied with a 16x16 matrix that contains shifted reference short training symbols. This requires 16 vector multiplications and 16 vector sums. A vector sum and multiplication can run in parallel so at least 16 instruction cycles are required for the matrix multiplication. Of the resulting vector the absolute values of its elements are computed and squared, which gives the correlated values. Then the mean and maximum value of this vector is computed. This requires at least another 4 extra vector operations. The resulting scalar values are compared with a threshold, which is estimated to require another 10 instructions. Furthermore, an estimated 10 instruction cycles are spend on overall control operations. The Detect task needs one block of input tokens and does not produce output tokens, so in total it needs \(16 + 4 + 10 + 10 + I_{\text{ttlread}}\) = 50 instructions.

The Gain and TSync tasks are similar to the Detect task. They also perform a correlation on the incoming samples with a reference short training symbol. However, they can use a single vector instead of a matrix, since the detection has already aligned the system to the start of a short training symbol. This greatly simplifies the implementations. The Gain task only uses the mean value of the resulting vector to compute the new gain level. The TSync task just needs the total correlation to verify whether the received samples are actual short training symbols. Both tasks are similarly complex and are estimated to require only 10 instructions. Because these tasks are smaller they need less functional overhead, which is estimated at 5 instructions. In total both tasks will require \(10 + 5 + I_{\text{ttlread}}\) = 25 instructions.

The last task that is mapped on the EVP is the Preamble task. When activated this task only has to send out a stream of 320 predefined tokens \(I_{\text{ttlwrite}}\) instructions followed by an estimated 25 instructions for functional overhead. In total this is 35 instructions.

**Summary**

The resulting instructions cycles of all the EVP tasks and the extra parameters are summarized in Table 6.3.

**6.4.3 Minimal operating frequency**

In order to support a specific number of WLAN radios the EVP must have a minimum amount of processing power, which basically comes down to a minimum operating frequency. It depends
6.4. EVP VECTOR PROCESSOR

Table 6.3: Instruction cycle counts

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>preempt</td>
<td>80</td>
</tr>
<tr>
<td>interrupt</td>
<td>30</td>
</tr>
<tr>
<td>check</td>
<td>5</td>
</tr>
<tr>
<td>Idle_read</td>
<td>10</td>
</tr>
<tr>
<td>Idle_write</td>
<td>10</td>
</tr>
<tr>
<td>IFSync</td>
<td>220</td>
</tr>
<tr>
<td>IMod</td>
<td>185</td>
</tr>
<tr>
<td>IDetect</td>
<td>50</td>
</tr>
<tr>
<td>IGain</td>
<td>25</td>
</tr>
<tr>
<td>ITSync</td>
<td>25</td>
</tr>
<tr>
<td>IPreamble</td>
<td>35</td>
</tr>
</tbody>
</table>

on the number of instructions required for executing the tasks in a specific time unit added with the administrative overhead discussed earlier.

Since the short-period tasks are implemented as an ISR, they will delay all other tasks including the long-period tasks. A long period of 4 µs will be interrupted 5 times (every 0.8 µs) for at least the administrative overhead of the ISR. This means that per long period the number of available instructions will be reduced with at least:

\[5 \times (I_{interrupt} + N \times I_{check}) = 150 + N \times 25\]

Besides this the EVP will need a number of instructions for the scheduler that handles the long-period tasks (see Equation 5.1). So for the administrative overhead in total a minimum frequency as defined in Equation 6.6 is required. For \(N = 4\) this is at least 87.5 MHz.

\[F_{EVP,\text{overhead}} = \frac{(I_{preempt} + N \times I_{check}) + 5 \times (I_{interrupt} + N \times I_{check})}{T_{long}} = \frac{230 + N \times 30}{4 \, \mu s}\] (6.6)

If the processing of the tasks is included then there are two extremes. In one situation all the radios have an active long-period task of which the Demod task has the largest instruction count (see Table 6.3). The corresponding minimum operating frequency is listed in Equation 6.7, which results in 322.5 MHz for \(N = 4\).

\[F_{EVP,\text{long}} = \frac{(I_{preempt} + N \times (I_{check} + I_{Demod})) + 5 \times (I_{interrupt} + N \times I_{check})}{T_{long}} = \frac{230 + N \times 265}{4 \, \mu s}\] (6.7)

In the other extreme all the radios have an active short-period task of which the Detect state requires the most instructions. Equation 6.8 presents the formula of the minimum operating frequency for this situation. For \(N = 4\) this means \(F_{EVP}\) must be at least 337.5 MHz.

\[F_{EVP,\text{short}} = \frac{(I_{preempt} + N \times I_{check}) + 5 \times (I_{interrupt} + N \times (I_{check} + I_{Detect}))}{T_{long}} = \frac{230 + N \times 280}{4 \, \mu s}\] (6.8)

The overall minimum required operating frequency is defined by the maximum of these three formulas (see Equation 6.9). With the defined number of instruction this will be equal to \(F_{EVP,\text{short}}\), where all radios have an active short-period task. Unfortunately, if a wireless LAN is
not used intensively, then a radio will be in the detect state for most of the time. Hence, the load
will then be high for most of the time.

\[ F_{EVP} = \max (F_{EVP, \text{overhead}}, F_{EVP, \text{long}}, F_{EVP, \text{short}}) \]
\[ = F_{EVP, \text{short}} \]  \hspace{1cm} (6.9)

### 6.4.4 Processing times

The processing times can be defined now that the instruction counts and operating frequencies
are known. The worst-case processing times of a task depend on the number of times it can be
preempted. Only long-period tasks can be preempted by the short-period tasks. The short-period
tasks cannot be preempted since they are executed in an ISR.

#### Short-period tasks

Short-period tasks cannot be preempted at all, so they can be computed quite easily. Per task
only one check is done and then the task’s instructions are executed. This results in the worst-case
processing times listed in Equation 6.10.

\[ P_{\text{Detect}} = \left\lceil \frac{I_{\text{check}} + I_{\text{Detect}}}{F_{EVP}} \right\rceil = \left\lceil \frac{55 \text{ instructions}}{F_{EVP}} \right\rceil \]
\[ P_{\text{Gain}} = \left\lceil \frac{I_{\text{check}} + I_{\text{Gain}}}{F_{EVP}} \right\rceil = \left\lceil \frac{30 \text{ instructions}}{F_{EVP}} \right\rceil \]
\[ P_{\text{TSync}} = \left\lceil \frac{I_{\text{check}} + I_{\text{TSync}}}{F_{EVP}} \right\rceil = \left\lceil \frac{30 \text{ instructions}}{F_{EVP}} \right\rceil \]
\[ P_{\text{Preamble}} = \left\lceil \frac{I_{\text{check}} + I_{\text{Preamble}}}{F_{EVP}} \right\rceil = \left\lceil \frac{40 \text{ instructions}}{F_{EVP}} \right\rceil \]  \hspace{1cm} (6.10)

In one short period \( N \) instances of a short-period task can be active. Besides the executions
of the tasks there are also a number of instructions spend on context switching. The worst-case
period is defined for each task in Equation 6.11.

\[ T_{\text{Detect}} = \left\lceil \frac{I_{\text{interrupt}} + N \cdot (I_{\text{check}} + I_{\text{Detect}})}{F_{EVP}} \right\rceil = \left\lceil \frac{30 + N \cdot 55}{F_{EVP}} \right\rceil \]
\[ T_{\text{Gain}} = \left\lceil \frac{I_{\text{interrupt}} + N \cdot (I_{\text{check}} + I_{\text{Gain}})}{F_{EVP}} \right\rceil = \left\lceil \frac{30 + N \cdot 30}{F_{EVP}} \right\rceil \]
\[ T_{\text{TSync}} = \left\lceil \frac{I_{\text{interrupt}} + N \cdot (I_{\text{check}} + I_{\text{TSync}})}{F_{EVP}} \right\rceil = \left\lceil \frac{30 + N \cdot 30}{F_{EVP}} \right\rceil \]
\[ T_{\text{Preamble}} = \left\lceil \frac{I_{\text{interrupt}} + N \cdot (I_{\text{check}} + I_{\text{Preamble}})}{F_{EVP}} \right\rceil = \left\lceil \frac{30 + N \cdot 40}{F_{EVP}} \right\rceil \]  \hspace{1cm} (6.11)

#### Long-period tasks

The worst-case processing times of the long-period tasks are a bit more complex. These tasks can
be preempted by the short-period tasks, so the amount of processing per time unit that can be
spend on the long-period tasks depends greatly on the short-period tasks. The more a long-period
task gets preempted by a short-period task, the longer the total processing time gets.

When only one long-period task is observed then in the worst-case scenario it will be running
in parallel with \((N - 1)\) short-period tasks (of which the Detect task is the most complex). This
way the long-period task gets interrupted the longest, and thus amount of time spend on it will
be minimal.
This can best be explained using an example system that has four radios active \((N = 4)\) and an EVP with an operating frequency of 400 MHz \((F_{EVP} = 400 \text{ MHz})\). On this system there is a single Demod task running in parallel with 3 Detect tasks. Every long period \(I_{\text{check}} + I_{\text{Demod}} = 5 + 235 = 240\) instructions are spend on that single Demod task. Meanwhile, every short period the 3 Detect tasks are executed. In total this requires \(I_{\text{interrupt}} + N \ast I_{\text{check}} + (N - 1) \ast I_{\text{Detect}} = 30 + 4 \ast 5 + 3 \ast 50 = 30 + 20 + 150 = 200\) instructions per short-period.

As a result the currently running task is interrupted every 800 ns for 200 instructions. In one short period a total of 800 \(\ast F_{EVP} = 320\) instructions can be executed, so every short period only 120 instructions can be spend on the long period and any background task. For the Demod task this means that it will be interrupted \(\left\lceil \frac{240}{120} \right\rceil = 2\) times. In other words, before the Demod task is completely executed a total of 240 + 2 \(\ast 200 = 640\) instructions are spend, which would take \(\left\lceil \frac{640}{400 \text{ MHz}} \right\rceil = 1.6\mu s\).

This can generally be written as follows. The number of times a block of instructions (called \(\text{task}\)) is interrupted by another block of instructions (called \(\text{int}\)) equals:

\[
\left\lceil \frac{I_{\text{task}}}{T_{\text{short}} \ast F_{EVP} - I_{\text{int}}} \right\rceil
\]

This would add the following amount of time to the processing time of \(\text{task}\).

\[
\left\lceil \frac{I_{\text{task}}}{T_{\text{short}} \ast F_{EVP} - I_{\text{int}}} \right\rceil \ast \frac{I_{\text{int}}}{F_{EVP}}
\]

So the total worst-case processing time of \(\text{task}\) interrupted every short period by \(\text{int}\) is as presented in Equation 6.12.

\[
\text{Interrupted}(I_{\text{task}}, I_{\text{int}}) = \left[ \frac{I_{\text{task}}}{F_{EVP}} \right] + \left[ \frac{I_{\text{task}}}{T_{\text{short}} \ast F_{EVP} - I_{\text{int}}} \right] \ast \frac{I_{\text{int}}}{F_{EVP}}
\]  \hspace{1cm} (6.12)

This can be used to define the processing times of the long-period tasks, which are described in Equation 6.13.

\[
P_{\text{Demod}} = \text{Interrupted}(I_{\text{check}} + I_{\text{Demod}}, I_{\text{interrupt}} + N \ast I_{\text{check}} + (N - 1) \ast I_{\text{Detect}})
\]

\[
P_{\text{Mod}} = \text{Interrupted}(I_{\text{check}} + I_{\text{Mod}}, I_{\text{interrupt}} + N \ast I_{\text{check}} + (N - 1) \ast I_{\text{Detect}})
\]

\[
P_{\text{FSync}} = \text{Interrupted}(I_{\text{check}} + I_{\text{FSync}}, I_{\text{interrupt}} + N \ast I_{\text{check}} + (N - 1) \ast I_{\text{Detect}})
\]  \hspace{1cm} (6.13)

Using the same principle the worst-case period can be computed (see Equation 6.14). In this situation \(N\) long-period tasks get preempted a number of times by the short-period interrupt service routine, which only performs the checks and not execute any short-period tasks.

\[
T_{\text{Demod}} = \text{Interrupted}(I_{\text{preempt}} + N \ast (I_{\text{check}} + I_{\text{Demod}}), I_{\text{interrupt}} + N \ast I_{\text{check}})
\]

\[
T_{\text{Mod}} = \text{Interrupted}(I_{\text{preempt}} + N \ast (I_{\text{check}} + I_{\text{Mod}}), I_{\text{interrupt}} + N \ast I_{\text{check}})
\]

\[
T_{\text{FSync}} = \text{Interrupted}(I_{\text{preempt}} + N \ast (I_{\text{check}} + I_{\text{FSync}}), I_{\text{interrupt}} + N \ast I_{\text{check}})
\]  \hspace{1cm} (6.14)
CHAPTER 6. MAPPING

For the example system with four radios and an EVP at 400 MHz, this would mean the following processing times:

\[
P_{\text{Demod}} = \text{Interrupted}(5 + 235, 30 + 4 \times 5 + (4 - 1) \times 50)
= \text{Interrupted}(240, 200) = \left\lceil \frac{240}{400 \text{ MHz}} \right\rceil + \left\lceil \frac{240}{320-200} \times 200}{400 \text{ MHz}} \right\rceil
= 600 \text{ ns} + 1000 \text{ ns} = 1.6 \mu s
\]

\[
P_{\text{Mod}} = \text{Interrupted}(190, 200) = \left\lceil \frac{190}{400 \text{ MHz}} \right\rceil + \left\lceil \frac{190}{320-200} \times 200}{400 \text{ MHz}} \right\rceil
= 475 \text{ ns} + 1000 \text{ ns} = 1.475 \mu s
\]

\[
P_{\text{FSync}} = \text{Interrupted}(225, 200) = \left\lceil \frac{225}{400 \text{ MHz}} \right\rceil + \left\lceil \frac{225}{320-200} \times 200}{400 \text{ MHz}} \right\rceil
= 563 \text{ ns} + 1000 \text{ ns} = 1.563 \mu s
\]

The worst-case periods would then be:

\[
T_{\text{Demod}} = \text{Interrupted}(80 + 4 \times (5 + 235), 30 + 4 \times 5)
= \text{Interrupted}(1.040, 50)
= \left\lceil \frac{1.040}{400 \text{ MHz}} \right\rceil + \left\lceil \frac{1.040}{320-50} \times 50}{400 \text{ MHz}} \right\rceil
= \left\lceil \frac{1.040}{400 \text{ MHz}} \right\rceil + \left\lceil 4 \times 50}{400 \text{ MHz}} \right\rceil
= 2.600 + 500 = 3.1 \mu s
\]

\[
T_{\text{Mod}} = \text{Interrupted}(840, 50) = \left\lceil \frac{840}{400 \text{ MHz}} \right\rceil + \left\lceil 4 \times 50}{400 \text{ MHz}} \right\rceil
= 2.100 + 500 = 2.6 \mu s
\]

\[
T_{\text{FSync}} = \text{Interrupted}(980, 50) = \left\lceil \frac{980}{400 \text{ MHz}} \right\rceil + \left\lceil 4 \times 50}{400 \text{ MHz}} \right\rceil
= 2.450 + 500 = 2.95 \mu s
\]

6.4.5 Startup times

The startup times of the tasks are defined by the worst possible delay between the arrivals of the input tokens and the actual execution of the task. In a worst-case situation the last of the required tokens can arrive right after the check of the corresponding task has been done. That check will fail, since not enough tokens have arrived yet, so the task will not be executed this period. The next period, the check will succeed and the task gets executed.

This scenario is also visualized in Figure 6.2. The startup time is divided into two parts. Part (a) denotes the worst-case remaining time in the first period after the last required token have arrived. Part (b) denotes the time needed in the next period before the corresponding task is actually executed.
6.4. EVP VECTOR PROCESSOR

Figure 6.2: Worst-case startup time

**Short-period tasks**

The worst possible time left in part (a) happens when no tasks are executed and only the checks are executed. For short-period tasks this means that the remaining time of this period is as presented in Equation 6.17. The worst-case time that the last token can arrive is right after the check, so even before the final resume action. Therefore, only half the instructions of $I_{\text{interrupt}}$ are used.

$$T_{\text{short}} = \left\lceil \frac{\frac{I_{\text{interrupt}}}{2} + N * I_{\text{check}}}{F_{\text{EVP}}} \right\rceil$$

(6.17)

Then, before part (b) begins, all radios activate a short-period task. There are only two short-period tasks that can be activated first, namely the Detect task or Preamble task, of which the Detect task has the longest processing time. In other words, before the last short-period task can be executed, $(N - 1)$ Detect tasks need to be executed first, which results in the following equation for (b).

$$\left\lceil \frac{\frac{I_{\text{interrupt}}}{2} + N * I_{\text{check}} + (N - 1) * I_{\text{Detect}}}{F_{\text{EVP}}} \right\rceil$$

(6.18)

The summation of Equations 6.17 and 6.18 defines the worst possible time between the arrival of all required input tokens and the actual execution of the corresponding task, i.e. the startup time. This holds for all short-period tasks, so their startup times are equal to Equation 6.19.

$$S_{\text{Detect}} = S_{\text{Gain}} = S_{\text{TSync}} = S_{\text{Preamble}} =$$

$$T_{\text{short}} = \left\lceil \frac{\frac{I_{\text{interrupt}}}{2} + N * I_{\text{check}}}{F_{\text{EVP}}} \right\rceil + \left\lceil \frac{\frac{I_{\text{interrupt}}}{2} + N * I_{\text{check}} + (N - 1) * I_{\text{Detect}}}{F_{\text{EVP}}} \right\rceil$$

(6.19)

**Long-period tasks**

For the long-period tasks the same scenario can happen, where the period is 4 $\mu$s. For downstream processing the first long-period task that can get activated is the FSync task, and for upstream the Mod task. The FSync task has the longest processing time. However, the execution of long-period tasks gets interrupted, which makes their computation more difficult.

Fortunately, the time they get interrupted in this situation is minimal. The FSync is activated when 80 samples are received. This takes quite some time, so when the last of these samples have arrived no short-period task will be active. This means that only the checks of the short-period tasks will be executed. The resulting time for part (a) is presented in Equation 6.20.

$$T_{\text{long}} - \text{Interrupted} \left( \left( \frac{I_{\text{preempt}}}{2} + N * I_{\text{check}} \right), (I_{\text{interrupt}} + N * I_{\text{check}}) \right)$$

(6.20)

And the time of for part (b) equals the following.

$$\text{Interrupted} \left( \left( \frac{I_{\text{preempt}}}{2} + N * I_{\text{check}} + (N - 1) * I_{\text{FSync}} \right), I_{\text{interrupt}} + N * I_{\text{check}} \right)$$

(6.21)
Adding Equation 6.20 and 6.21 together results in the startup times of the long-period tasks (see Equation 6.22).

\[ S_{\text{Demod}} = S_{\text{Mod}} = S_{\text{FSync}} = \]
\[ T_{\text{long}} - \text{Interrupted} \left( \frac{I_{\text{preempt}}}{2} + N \ast I_{\text{check}}, I_{\text{interrupt}} + N \ast I_{\text{check}} \right) \]
\[ + \text{Interrupted} \left( \frac{I_{\text{preempt}}}{2} + N \ast I_{\text{check}} + (N - 1) \ast I_{\text{FSync}}, I_{\text{interrupt}} + N \ast I_{\text{check}} \right) \]

(6.22)

For the example system with four radios and \( F_{\text{EVP}} = 400 \text{ MHz} \) this would mean that the long-period tasks have the following startup time.

\[ S_{\text{Demod}} = S_{\text{Mod}} = S_{\text{FSync}} = 4.000 - \text{Interrupted}(80, 30 + 4 \ast 5) \]
\[ + \text{Interrupted}(220, 30 + 4 \ast 5) \]
\[ = 4.000 - \text{Interrupted}(60, 50) + \text{Interrupted}(720, 50) \]
\[ = 4.000 - (150 + 125) + (1.800 + 375) \]
\[ = 4.000 - 275 + 2.175 = 5.9 \mu s \]

6.5 Software codec

The software codec is designed for coding and decoding blocks of bits. In the WLAN model the Code and Decode tasks are responsible for this, so these tasks will be mapped on this processor.

6.5.1 Processing times

It is assumed that Code and Decode tasks are the most complex tasks mapped on the SWC and that all other tasks will have a lower worst-case processing time. This means that the Decode and Code tasks of the WLAN radio will be equal to the maximum worst-case processing time of the SWC.

\[ P_{\text{Decode}} = P_{\text{decoder}} \]
\[ P_{\text{Code}} = P_{\text{coder}} \]

(6.23)

Furthermore, the Code task is less complex than the Decode task, because the Decode task has to take error correction into account. The worst-case processing time of the Code task will therefore be less than the Decode task.

\[ P_{\text{Code}} \leq P_{\text{Decode}} \]

The worst-case periods are easily computed. The processor does not support preemption, so the time between two consecutive executions of the same task is at most \( N \) times the length of one executions (one execution for each radio instance).

\[ T_{\text{Decode}} = N \ast P_{\text{Decode}} \]
\[ T_{\text{Code}} = N \ast P_{\text{Code}} \]

(6.24)
6.5. SOFTWARE CODEC

6.5.2 Startup times

The startup time of a task on the SWC is the time between the arrival of a token in the processor’s input queue and the actual processing of it. So it mainly depends on the number of tokens already in the queue. The number of tokens that can arrive is bounded. With $N$ active radios at most $N$ tokens will arrive within a period of $4 \, \mu s$. A number of situations have to be taken into account when defining a formula for the maximum number of tokens that are in the queue on the moment a specific token arrives.

- Do SOFTBITS tokens arrive faster than they can be processed?
- Do BITS tokens arrive faster than they can be processed?
- Do SOFTBITS tokens arrive faster than BITS tokens?
- How many streams are in the upstream state and how many in the downstream state.

There are a lot combinations possible, so defining a generic formula will be difficult. To simplify this, another assumption is added: the Decode task has a higher priority than all other tasks. This means that if there are tokens for the Decode task in the queue they are processed right after the current task is finished, even if there are tokens for the Code task waiting.

**Decode task**

With this new assumptions the startup time of the Decode task only depends on the number of previously arrived SOFTBITS tokens. These tokens are produced one at a time by the Demod task and have an average arrival period of $T_{\text{Demod}}/N$.

![Diagram of SOFTBITS tokens arriving faster than they can be processed](image)

![Diagram of BITS tokens arriving faster than they can be processed](image)

Figure 6.3: Response times of Decode task.

The worst-case response time of the Decode task can be defined for two situations. If $P_{\text{Decode}}$ is less than $T_{\text{Demod}}/N$, i.e. tokens are consumed faster than they arrive, then the startup time is bounded to at most one execution of an arbitrary task. This is also visualized in Figure 6.3(a).

The processing time of an arbitrary task is less than that of the Decode task ($P_{\text{Decode}} = P_{\text{decoder}}$), so the worst-case startup time of the Decode task in this situation will be no more than $P_{\text{Decode}}$, which is also specified in Equation 6.25.

$$S_{\text{Decode}} = P_{\text{Decode}}$$  \hspace{1cm} (6.25)
In the other situation SOFTBITS tokens arrive faster than they are consumed, so $P_{\text{Decode}}$ is greater than $\frac{T_{\text{Demod}}}{N}$. Here the startup time grows when more tokens arrive, which is the largest for the last token that arrived in a period. This situation is depicted in Figure 6.3(b). Each period at most $N$ tokens can arrive. When the first of these tokens arrives the SWC might be working on an arbitrary task, which has to be finished first. So the time it takes before the last of the tokens can be processed is equal to $P_{\text{Decode}} + (N - 1) * P_{\text{Decode}}$. The time at which that token arrived is $(N - 1) * \frac{T_{\text{Demod}}}{N}$ after the first token. The worst-case startup time is the difference between these times. This is described in Equation 6.26.

\[
S_{\text{Decode}} = P_{\text{Decode}} + (N - 1) * P_{\text{Decode}} - (N - 1) * \frac{T_{\text{Demod}}}{N}
\]

In both equations the first element of the equation is the same and if Equation 6.26 is filled in for a situation where $P_{\text{Decode}} < \frac{T_{\text{Demod}}}{N}$ then the last element in the sum will be negative. This can be used to merge the two formulas into one. The resulting worst-case startup times of the Decode task is described in Equation 6.27.

\[
S_{\text{Decode}} = P_{\text{Decode}} + \max((N - 1) * (P_{\text{Decode}} - \frac{T_{\text{Demod}}}{N}), 0)
\]

**Code task**

With the new assumption the Code task has to wait a lot longer, because any pending Decode task will be executed first. The Decode task has the longest processing time, so the worst-case situation for the Code task would be that it has to wait for an arbitrary task followed by $(N - 1)$ Decode tasks. Since the processing time of the arbitrary task will not exceed that of the Decode task, it will result in a maximum startup time as described in Equation 6.28.

\[
S_{\text{Code}} = P_{\text{Decode}} + (N - 1) * P_{\text{Decode}} = N * P_{\text{Decode}}
\]

This is equal to one period of the Decode task (see Equation 6.24), which should not be larger than 4 $\mu$s because of the throughput constraint.

### 6.6 ARM micro-processor

The ARM micro-processor is responsible for the MAC layer tasks. Only two tasks of the WLAN application model do this, namely the MACi and MACo tasks, so these will be mapped on this processor. These tasks control the incoming and outgoing flow of bits that are encapsulated in BITS tokens. The MACo task only sends out a given block of bits; no special computation is required. The MACi task, on the other hand, needs to verify if the received bits are correct and trigger the transmission when a frame is completely received.

#### 6.6.1 Processing times

The MACi task checks whether the data is correct by computing a CRC checksum on the payload of the physical layer frame. It compares the result with the given CRC which is appended at the end of a frame. As explained, it is estimated that the ARM processor of this platform can compute the CRC checksum at an average rate of 16 bits per instruction cycle. Since the maximum size of one BITS token is 216 bits, at least $\lceil \frac{216}{16} \rceil = 14$ instructions are needed by the MACi task for CRC check alone. Besides this around 25 instructions are required for control, TTL primitives and functional overhead. In total it will need 40 instructions for the entire MACi task.
The MACi task needs to be executed on the arrival of a BITS token from the Decode task. Since the MACi task is at the end of the pipeline the token arrival rate is quite low. To be precise, tokens arrive at a rate of 250,000 per second per radio. Tasks can thus be executed right when a token arrives. Token arrival is triggered with an interrupt, so once a token arrives an interrupt service routine is executed which includes the MACi task. The interrupt routine has some administrative overhead (like storing and loading the registers), which is estimated at a total of 20 instructions. Adding this to the MACi task results in a total of 60 instructions per execution. This number is defined as $I_{MACi}$.

$$I_{MACi} = 60 \text{ instructions}$$

The CRC check is only required for incoming frames. Outgoing frames are controlled by the MACo task, which only has to transmit a block of bits (one BITS token) every 4 $\mu$s. No additional processing on the data itself is required, so the task is much simpler. Once triggered by the MACi task or the application it will be activated by a timer every 4 $\mu$s. The task has no input tokens. The total processing for the MACo task is estimated at 30 instructions. Preemption adds another 20 instructions overhead to a total of 50 instructions.

$$I_{MACo} = 50 \text{ instructions}$$

Now the worst-case processing times can be computed, which are specified in Equation 6.29. The worst-case periods are specified in Equation 6.30.

$$P_{MACi} = \left\lceil \frac{I_{MACi}}{F_{ARM}} \right\rceil$$
$$P_{MACo} = \left\lceil \frac{I_{MACo}}{F_{ARM}} \right\rceil$$

$$T_{MACi} = N \times P_{MACi}$$
$$T_{MACo} = N \times P_{MACo}$$

(6.30)

### 6.6.2 Minimum operating frequency

The MACi task apparently has the longest processing time. In order to support $N$ instances of this task the ARM will need a minimum required operating frequency of:

$$F_{ARM} = \frac{N \times \max (I_{MACi}, I_{MACo})}{T_{long}}$$

(6.31)

### 6.6.3 Startup times

The MACi tasks are activated by the arrival of a BITS token and the MACo tasks are activated by a timer. When multiple radios are running simultaneously it can happen that multiple activations occur at the same time. Already running WLAN tasks will not be preempted since they have the same priority, so the tasks will be processed one after the other. The task with the worst-case startup time is the last one in such a sequence, which can be at most after $N - 1$ other tasks. This is defined in Equation 6.32.

$$S_{MACi} = S_{MACo} = (N - 1) \times P_{MACi}$$

(6.32)
6.7 Summary

In this chapter the task-graph of the WLAN model is mapped on the platform architecture. As a result, the properties of the tasks (like processing- and startup times) could be defined with formulas that only depend on the parameters of the platform and the number of radios running on it.

The properties of the tasks mapped on the RF front-end and ARM micro-processor are computed easily. For the EVP and SWC these are, however, quite complex. For the EVP this is mainly caused by the possibility that tasks can be preempted or interrupted by other tasks. The worst-case processing time of a task is affected by this.

Of the tasks mapped on the SWC only the startup time is complex. This is because they depend on the arrival rate of the input tokens. To simplify the computation a new assumption has been added, which states that the Decode task gets the highest priority of all tasks.

With these formulas the constraints given by the application model can now be filled in for various configurations of the platform.
Chapter 7

Analysis

The goal of this study is to find out what the requirements are of the NXP platform in order to support multiple (preferably four) WLAN radios simultaneously. In the previous chapters an application model has been defined and mapped on a model of the architecture. This resulted in a number of formulas that compute the task properties for a given platform configuration. With these properties the throughputs and latencies of the WLAN application can be computed and tested against the constraints given by the application model.

The architecture that is responsible for executing the task-graph can be changed with a number of parameters that each affect the application model differently. The parameters $F_{EVP}$, $F_{ARM}$, $P_{decoder}$ and $P_{coder}$ influence the processing, startup and response times of the tasks. The communication times of the tokens can be computed using the $F_{Network}$ parameter.

A recommended configuration for these parameters can be found by analyzing the effects they have on the throughput and latencies of the WLAN model. This will be done in several steps. First an intuitively chosen configuration for four radios will be analyzed and improved. Then the parameters are changed in various ways, such that the effect of each individual parameter is clarified. This eventually results in a recommended configuration that supports four WLAN radios. An additional one radio configuration will also be highlighted.

7.1 Initial four radio configuration

As a starting point an initial configuration for four radios is needed that can be fine tuned in the right direction. This means that an initial value for the five architecture parameters must be defined. In the previous chapter minimal required values for most of these parameters have been defined that make sure the throughput requirements are satisfied. These can be used as a good starting point.

The minimum required network speed ($F_{Network}$) is the easiest to compute. From Equation 6.2 can be concluded that for every radio on the system 960 Mb/s is needed. So for $N = 4$ this is at least 3,84 Gb/s. This is rounded up to 4 Gb/s. The resulting communication times and periods are listed in Table 7.1.

\[ F_{Network} = 4 \text{ Gb/s} \]

The minimum operating frequency of the EVP can be computed with Equation 6.9. For $N = 4$ it gives a minimum of 338 MHz, which is rounded up to 350 MHz (see Equation 7.1).

\[ F_{EVP} = F_{EVP,\text{short}} = \frac{230 + 4 \times 280}{4 \mu s} = 337.5 \text{ MHz} \approx 350 \text{ MHz} \quad (7.1) \]

The minimum operating frequency of the ARM processor for $N = 4$ should be at least 60 MHz
CHAPTER 7. ANALYSIS

Table 7.1: Communication times (in ns) with a network speed of 4 Gb/s

<table>
<thead>
<tr>
<th>Token</th>
<th>C</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLE</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>SOFTBITS</td>
<td>296</td>
<td>1.184</td>
</tr>
<tr>
<td>BITS</td>
<td>62</td>
<td>248</td>
</tr>
<tr>
<td>RXVECTOR</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>TXVECTOR</td>
<td>16</td>
<td>64</td>
</tr>
</tbody>
</table>

(see Equation 6.31), which is rounded up to 75 MHz

\[ F_{\text{ARM}} = \frac{4 \times 60}{4 \mu s} = 60 \text{ MHz} \approx 75 \text{ MHz} \]

The last two parameters are the maximum allowed worst-case processing times of the tasks mapped on the SWC (i.e. decoder and coder tasks). As explained, these numbers are not available yet, since the processor has not been completely developed yet. However, the values should satisfy the throughput constraints, so they should at least be lower than \( \frac{T_{\text{long}}}{N} \). For \( N = 4 \) this means they cannot exceed 1 \( \mu s \). For the moment the processing times are estimated at \( P_{\text{decoder}} = 900 \text{ ns} \) and \( P_{\text{coder}} = 800 \text{ ns} \).

**Task properties**

Using these parameters the processing, startup and response times of the tasks can be computed with the formulas given by the mapping. The results are listed in Table 7.2. As expected, the throughput requirements are satisfied, since for every task the worst-case period between two executions of the same task (\( T_x \)) does not exceed the constraint (see Equation 4.13). However, the startup times (and therefore the response times) are quite high for some tasks. Especially the long-period tasks on the EVP have a response time of more than twice their period. This will probably result in high latencies.

**Latencies**

The task properties can now be filled into the formulas of the application model that compute the latencies. The results are listed in Table 7.3. Unfortunately, the latencies of the SIFS and header-loop are both too high. The SIFS latency should actually be 4 \( \mu s \) higher. Because the header-loop latency is too high, the header information will not be available at the Demod task when the first data symbol arrives. This task will therefore have to wait another period of 4 \( \mu s \) before it can start on that symbol, which increases the worst-case startup time of the Demod task and therefore the SIFS latency.

The two gain latencies are both within their bounds. However, they are quite high. It takes for instance almost 1.5 \( \mu s \) before the detect task makes changes to the received samples. During this time already 30 new samples have arrived using the old gain level, making the following two Gain tasks useless.

Luckily, the acknowledge frame latencies show no problems. Even though the Code task has a lower priority than the Decode task, and will have to wait a lot longer, the ACK header symbol is still ready at least 4 \( \mu s \) before the Tx task is finished with the preamble. This also holds for the first ACK data symbol. And since transmitting an ACK frame is no problem it will be no problem to transmit other outgoing frames either.

Obviously, the high SIFS and header-loop latencies are for the most part caused by the high response times of the Demod, Decode, MACi and MACo tasks. These should be lowered by changing some parameters.
7.1. INITIAL FOUR RADIO CONFIGURATION

Table 7.2: Task times (in ns) for $N = 4$

<table>
<thead>
<tr>
<th>Task</th>
<th>$P$</th>
<th>$S$</th>
<th>$R$</th>
<th>$T$</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Tx</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Detect</td>
<td>158</td>
<td>1.229</td>
<td>1.387</td>
<td>715</td>
<td>4.800</td>
</tr>
<tr>
<td>Gain</td>
<td>86</td>
<td>1.229</td>
<td>1.315</td>
<td>429</td>
<td>800</td>
</tr>
<tr>
<td>TSync</td>
<td>86</td>
<td>1.229</td>
<td>1.315</td>
<td>429</td>
<td>800</td>
</tr>
<tr>
<td>Preamble</td>
<td>115</td>
<td>1.229</td>
<td>1.344</td>
<td>543</td>
<td>16.000</td>
</tr>
<tr>
<td>Demod</td>
<td>2.401</td>
<td>6.315</td>
<td>8.716</td>
<td>3.687</td>
<td>4.000</td>
</tr>
<tr>
<td>Mod</td>
<td>2.258</td>
<td>6.315</td>
<td>8.573</td>
<td>2.972</td>
<td>4.000</td>
</tr>
<tr>
<td>FSync</td>
<td>2.358</td>
<td>6.315</td>
<td>8.673</td>
<td>3.515</td>
<td>4.000</td>
</tr>
<tr>
<td>Decode</td>
<td>900</td>
<td>900</td>
<td>1.800</td>
<td>3.600</td>
<td>4.000</td>
</tr>
<tr>
<td>Code</td>
<td>800</td>
<td>3.600</td>
<td>4.400</td>
<td>3.200</td>
<td>4.000</td>
</tr>
<tr>
<td>MACi</td>
<td>800</td>
<td>2.400</td>
<td>3.200</td>
<td>3.200</td>
<td>4.000</td>
</tr>
<tr>
<td>MACo</td>
<td>667</td>
<td>2.400</td>
<td>3.067</td>
<td>2.667</td>
<td>4.000</td>
</tr>
</tbody>
</table>

Table 7.3: Latencies (in ns) for $N = 4$

<table>
<thead>
<tr>
<th>Latency description</th>
<th>Latency</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Interframe Space (Equation 4.15)</td>
<td>18.697</td>
<td>16.000</td>
</tr>
<tr>
<td>Header-loop (Equation 4.16)</td>
<td>4.513</td>
<td>4.000</td>
</tr>
<tr>
<td>Gain Control by Detect (Equation 4.17)</td>
<td>1.485</td>
<td>1.600</td>
</tr>
<tr>
<td>Gain Control by Gain</td>
<td>1.413</td>
<td>1.600</td>
</tr>
<tr>
<td>ACK Header (Equation 4.18)</td>
<td>13.317</td>
<td>17.392</td>
</tr>
<tr>
<td>ACK Data (Equation 4.19)</td>
<td>16.446</td>
<td>21.392</td>
</tr>
</tbody>
</table>

7.1.1 Increasing performance

In order to reduce the latencies several parameters can be altered. First of all, two of the tasks responsible for the high SIFS latency are mapped on the ARM, namely MACi and MACo. The response times of these tasks depend only on the operating frequency of the processor. Increasing this will therefore have a positive effect on the SIFS latency. Since the ARM will probably be running more than just the WLAN control and much higher frequencies are not uncommon, it is increased to 250 MHz.

$$F_{ARM} = 250 \text{ MHz}$$

Unfortunately the header-loop latency is not affected by this. It only depends on the times of the Demod and Decode tasks. The response time of the Demod task can go down by increasing the EVP frequency. Unfortunately this frequency is already quite high, for the moment it is increased to its maximum of 400 MHz.

$$F_{EVP} = 400 \text{ MHz}$$

The response time of the Decode task is something that depends on the implementation of the processor. The processor is developed with a certain throughput in mind. To increase the performance the maximum allowed worst-case processing time of the SWC is reduced to 700 ns for both coding and decoding tasks.

$$P_{decoder} = P_{coder} = 700 \text{ ns}$$

The corresponding task times are listed in Table 7.4. What can be observed is that the response times of the long-period EVP tasks have all gone down with more than one microsecond, the response time of the Decode task with almost half a microsecond and the response times of the ARM tasks have dropped with more than two microseconds.
CHAPTER 7. ANALYSIS

Table 7.4: Task times (in ns) for $N = 4$ with better performance

<table>
<thead>
<tr>
<th>Task</th>
<th>P</th>
<th>S</th>
<th>R</th>
<th>T</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Tx</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Detect</td>
<td>138</td>
<td>1.175</td>
<td>1.313</td>
<td>625</td>
<td>4.800</td>
</tr>
<tr>
<td>Gain</td>
<td>75</td>
<td>1.175</td>
<td>1.250</td>
<td>375</td>
<td>800</td>
</tr>
<tr>
<td>TSync</td>
<td>75</td>
<td>1.175</td>
<td>1.250</td>
<td>375</td>
<td>800</td>
</tr>
<tr>
<td>Preamble</td>
<td>100</td>
<td>1.175</td>
<td>1.275</td>
<td>475</td>
<td>16.000</td>
</tr>
<tr>
<td>Demod</td>
<td>1.600</td>
<td>5.900</td>
<td>7.500</td>
<td>3.100</td>
<td>4.000</td>
</tr>
<tr>
<td>Mod</td>
<td>1.475</td>
<td>5.900</td>
<td>7.375</td>
<td>2.600</td>
<td>4.000</td>
</tr>
<tr>
<td>FSync</td>
<td>1.563</td>
<td>5.900</td>
<td>7.463</td>
<td>2.950</td>
<td>4.000</td>
</tr>
<tr>
<td>Decode</td>
<td>700</td>
<td>700</td>
<td>1.400</td>
<td>2.800</td>
<td>4.000</td>
</tr>
<tr>
<td>Code</td>
<td>700</td>
<td>2.800</td>
<td>3.500</td>
<td>2.800</td>
<td>4.000</td>
</tr>
<tr>
<td>MACi</td>
<td>240</td>
<td>720</td>
<td>960</td>
<td>960</td>
<td>4.000</td>
</tr>
<tr>
<td>MACo</td>
<td>200</td>
<td>720</td>
<td>920</td>
<td>800</td>
<td>4.000</td>
</tr>
</tbody>
</table>

This has of course a positive effect on the latencies. The newly calculated values are shown in Table 7.5. It shows that the SIFS latency is now well below its threshold and the header-loop latency is still high but also below its threshold. The gain latencies have not gone down that much, but they only depend on the EVP speed and that has not gone up much.

The ACK latencies have both gone down dramatically. The header symbol is processed 2 $\mu$s earlier, and the first data symbol is processed even 5 $\mu$s earlier than before. This large difference is caused by the much lower response time of the MACo tasks, on which the ACK data symbol latency is depending. Again it shows that transmitting outgoing frames pose no problem at all. The header and data symbols are processed well before the preamble has been fully transmitted.

Table 7.5: Latencies (in ns) for $N = 4$ with better performance

<table>
<thead>
<tr>
<th>Latency description</th>
<th>Latency</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Interframe Space</td>
<td>12.625</td>
<td>16.000</td>
</tr>
<tr>
<td>Header-loop</td>
<td>3.312</td>
<td>4.000</td>
</tr>
<tr>
<td>Gain Control by Detect</td>
<td>1.411</td>
<td>1.600</td>
</tr>
<tr>
<td>Gain Control by Gain</td>
<td>1.348</td>
<td>1.600</td>
</tr>
<tr>
<td>ACK Header</td>
<td>11.219</td>
<td>17.323</td>
</tr>
<tr>
<td>ACK Data</td>
<td>12.201</td>
<td>21.323</td>
</tr>
</tbody>
</table>

With this configuration the various processors have a surplus of processing power. The ARM only needs a minimum of 60 MHz but is running at 250 MHz. This means that 190 MHz is available for background task and the WLAN radios are only using 24 % of the capacity. The EVP needs 338 MHz but is running at 400 MHz, which is a load of 84.5 %. The SWC is in a worst-case situation spending 2.8 $\mu$s on the WLAN tasks every period of 4 $\mu$s, which means a load of 70 %. In other words, the EVP has the highest load, which is expected, since it has to deal with most of the complex computations.

7.2 Effects of parameters

It became clear that changing some parameters have more effect than others. Therefore, several parameters are now changed systematically in order to find out their exact effect on the overall performance.

7.2.1 EVP speed

The EVP frequency is currently set at 400 MHz. If this is changed up- and downwards while keeping the other parameters the same, then the latencies are changing is well. The latencies for
an EVP speed between 300 and 800 MHz are depicted in Figure 7.1 (excluding the ACK latencies). Note that the 300 MHz is below the minimum frequency for the throughput constraints, but it can be used for this analysis.

Figure 7.1: Effect of EVP frequency on latencies

As it appears, increasing the frequency upwards from 400 MHz will not help very much. Only at around 550 MHz a small dip is observed in the SIFS and header-loop latencies; after that the latencies stay almost the same. This dip exists because at exactly 550 MHz the maximum number of interruptions of the Demod task is reduced from 2 to 1. There are two more of these points; at 350 MHz the maximum number of interruptions of the Demod task goes down to 3 and at 400 MHz it goes down to 2. Less than 1 interruption is not possible (there will always be a worst-case scenario that a short period interrupts a long period task), so at frequencies higher than 550 MHz the latencies do not decrease very fast. Only at frequencies less than 375 MHz the header-loop latency rises above its limit of 4 \( \mu s \).

The gain latencies, on the other hand, hardly seem affected by the processor speed. Even at frequencies less than the minimum required for the throughput they stay below their limit of 1.6 \( \mu s \). They are less affected by the frequency because the largest part consists of static values. 800 ns is part of the startup time of the Detect or Gain task and another 98 ns is spent on the processing and transmission of a SAMPLE token. The remaining 500 ns are directly depending on the EVP frequency. Since this is only a small part, the latencies do not change that much. At the end they will converge at around 900 ns, so the worst-case situation will always have 18 or more samples using the old gain level.

The two ACK latencies show a similar curve as the SIFS latency (see Figure 7.2), but will never pose a threat. Even at frequencies below the minimum the latencies are still well below their limits. They go down as the frequency increases while their limits stay nearly the same. At very high frequencies the latencies are at less than half of their limit.

In other words, an operating frequency of 400 MHz seems to be a near optimal solution. The SIFS and header-loop latencies are near their lowest values. The other latencies are less affected, but always within their bounds.

### 7.2.2 ARM speed

Changing the ARM speed will only have an effect on the SIFS and ACK data latency. The other latencies are not depending on tasks running on the ARM processor.

The latencies corresponding to frequency between 50 and 550 MHz are depicted in Figure 7.3. As expected most of the lines are flat except for the SIFS latency, which is quite smooth. It is smooth because the response times of the MACo and MACi tasks are not depending on anything
else but the frequency. As long as the frequency is equal to or greater than 100 MHz the SIFS latency will stay below the 16 $\mu$s (of course the other parameters should remain the same).

The response times of the two MAC tasks are both hyperbolic functions. This means that the SIFS latency increases more for lower frequencies and converges to a constant value at around 10,8 $\mu$s for higher frequencies.

### 7.2.3 Decoder processing time

Changing the speeds of the ARM processor does not affect the header-loop latency. This latency only depends on the Demod and Decode tasks. The Decode task is equal to the maximum worst-case processing time allowed for decoder tasks on the SWC. The influence of this parameter is visualized in Figure 7.4. It shows the latencies for processing times between 100 and 1,100 ns.

Both the SIFS and header-loop latencies seem to increase at the same rate. At around 750 ns they start to rise a bit faster. This point is exactly at 775 ns, which is equal to the average arrival rate of the SOFTBITS tokens from the Demod task (for an EVP frequency of 400 MHz). A decode time less than this will mean that SOFTBITS tokens are decoded faster than they arrive; hence the startup time of the Decode task will be minimal. A time larger than 775 ns will make
the startup time depend on the difference between the arrival rate and the processing rate (see Equation 6.27). This difference which will increase when the decode time gets larger, so the overall worst-case response time will also increase more.

Furthermore, between 850 and 900 ns the header latency rises above the allowed threshold of 4 $\mu$s. The SIFS latency does not reach its limit even when the processing time is larger than the 1 $\mu$s that the throughput constraint dictates.

### 7.2.4 Network speed

Until now the network speed has not been changed. It stayed constantly at 4 Gb/s. Changing this both up- and downwards between 3 and 5,5 Gb/s will result in the latencies depicted in 7.5. It shows that the network speed has almost no effect on any latency. Although the minimum speed that guarantees the throughput should be 3,84 Gb/s, at lower speeds the SIFS latency still does not change very much.

This can be used to make an estimation about the effects of a proper network model. In this thesis the network is assumed to be a set of point-to-point connections. If a token is transmitted it does not have to wait for the channel to be free and no extra time is spent on arbitrating the
CHAPTER 7. ANALYSIS

communication, which is not realistic. In for example a packet switched network-on-chip each packet needs to be analyzed to find out what the destination is. This introduces an extra latency at each node (router).

By introducing an extra latency for each token type the network will perform worse than currently modeled. However, the graph shows that even at a throughput of 3 Gb/s the latencies are within bounds. A SOFTBITS token transmitted at 4 Gb/s takes 296 ns, while transmitting at 3 Gb/s it would take 395 ns. In other words an extra latency of 100 ns per SOFTBITS token will not make the latencies change very much. This also holds for the other tokens.

7.3 Recommended four radio configuration

The five main parameters that defined the architecture can now be fine tuned to a recommended configuration that should be able to support four WLAN radios running simultaneously on a system that has other (background) tasks running.

As it appears the EVP frequency of 400 MHz is quite optimal so it is recommended to have it running at that speed. The performance of the software codec has a lot of influence on the header-loop latency. It will reach its maximum if the processing time of the Decode task is around 850 ns. It is advised to have it running no longer than 800 ns, so that the header-loop latency is not too high for further adjustments to the architecture. An ARM processor running at 250 MHz is a bit overdone. The minimum should be at least 100 MHz or otherwise the SIFS latency will get too high. Having it operate at 150 MHz should be enough. The network speed does not influence the latencies that much. It should be at least 3,84 Gb/s, but it is advised to have some extra room and set it at 4 Gb/s. Summarized it means that the parameters are recommended to be:

\[
\begin{align*}
F_{EVP} & = 400 \text{ MHz} \\
F_{ARM} & = 150 \text{ MHz} \\
F_{Network} & = 4 \text{ Gb/s} \\
P_{decoder} & = 800 \text{ ns} \\
P_{coder} & = 800 \text{ ns}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Task</th>
<th>P</th>
<th>S</th>
<th>R</th>
<th>T</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Tx</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Detect</td>
<td>138</td>
<td>1.175</td>
<td>1.313</td>
<td>625</td>
<td>4.800</td>
</tr>
<tr>
<td>Gain</td>
<td>75</td>
<td>1.175</td>
<td>1.250</td>
<td>375</td>
<td>800</td>
</tr>
<tr>
<td>TSync</td>
<td>75</td>
<td>1.175</td>
<td>1.250</td>
<td>375</td>
<td>800</td>
</tr>
<tr>
<td>Preamble</td>
<td>100</td>
<td>1.175</td>
<td>1.275</td>
<td>475</td>
<td>16.000</td>
</tr>
<tr>
<td>Demod</td>
<td>1.600</td>
<td>5.900</td>
<td>7.500</td>
<td>3.100</td>
<td>4.000</td>
</tr>
<tr>
<td>Mod</td>
<td>1.475</td>
<td>5.900</td>
<td>7.375</td>
<td>2.600</td>
<td>4.000</td>
</tr>
<tr>
<td>FSync</td>
<td>1.563</td>
<td>5.900</td>
<td>7.463</td>
<td>2.950</td>
<td>4.000</td>
</tr>
<tr>
<td>Decode</td>
<td>800</td>
<td>875</td>
<td>1.675</td>
<td>3.200</td>
<td>4.000</td>
</tr>
<tr>
<td>Code</td>
<td>800</td>
<td>3.200</td>
<td>4.000</td>
<td>3.200</td>
<td>4.000</td>
</tr>
<tr>
<td>MACi</td>
<td>400</td>
<td>1.200</td>
<td>1.600</td>
<td>1.600</td>
<td>4.000</td>
</tr>
<tr>
<td>MACo</td>
<td>333</td>
<td>1.200</td>
<td>1.533</td>
<td>1.333</td>
<td>4.000</td>
</tr>
</tbody>
</table>

Table 7.6: Task times (in ns) for recommended $N = 4$ configuration

The task times corresponding to these parameters are listed in Table 7.6. From this the load of the system can be computed. The EVP will have a maximum load of around 84%, so around 62 MHz is always available for other tasks. The SWC has to spend 3,2 $\mu$s on WLAN tasks every 4 $\mu$s in a worst-case situation. This means that the load will be at most 86%. The ARM has the lowest load. Its maximum load will be only 40%.
7.3. RECOMMENDED FOUR RADIO CONFIGURATION

The corresponding latencies of the task-graph are listed in Table 7.7. It shows that the SIFS latency is only 1.8 $\mu$s below its maximum. The header-loop latency is even 0.4 $\mu$s below its limit. The gain latencies are still high. In a worst-case situation there will already be 28 SAMPLE tokens ready before the Detect task has set the gain level. The ACK latencies are like before well below their maximum. In other words, transmitting an outgoing frame will not be a problem.

<table>
<thead>
<tr>
<th>Latency description</th>
<th>Latency</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Interframe Space</td>
<td>14.153</td>
<td>16.000</td>
</tr>
<tr>
<td>Header-loop</td>
<td>3.587</td>
<td>4.000</td>
</tr>
<tr>
<td>Gain Control by Detect</td>
<td>1.411</td>
<td>1.600</td>
</tr>
<tr>
<td>Gain Control by Gain</td>
<td>1.348</td>
<td>1.600</td>
</tr>
<tr>
<td>ACK Header</td>
<td>11.719</td>
<td>17.323</td>
</tr>
<tr>
<td>ACK Data</td>
<td>13.314</td>
<td>21.323</td>
</tr>
</tbody>
</table>

7.3.1 Network latencies

As explained, the network model that is used is not very realistic. Fortunately, the recommended configuration leaves some room for improving this. In the SIFS flow five tokens are transmitted, namely two SAMPLE tokens, a SOFTBITS token, a BITS token and a TXVECTOR token. Since the SIFS latency is 1.8 $\mu$s below its maximum, one could conclude that this would allow an average extra latency of at most 360 ns per token.

It is however not that simple. First of all, the header-loop latency is a bit stricter. It has just around 400 ns left before it reaches its limit. In its flow two tokens are transmitted (a SOFTBITS and a RXVECTOR token), so on average this means that a network latency of at most 200 ns can be added to these tokens instead of the 360 ns.

Furthermore, the gain latencies have only 150 ns left, which is not much. The problem with these latencies is that the higher they get the less accurate the results of the Gain task will be, so this number should stay as low as possible. It is therefore not recommended that the latency between the Rx and Detect task, hence the SAMPLE token latency, is increased any further. This might not be such a simple task when the tokens are transmitted via a shared network. It is therefore advised that the SAMPLE tokens are transmitted between the RF front-end and the EVP processor via a separate link that is not shared with other channels.

Because the extra latencies of these tokens are restricted, more time is available for other tokens. Of the 1.8 $\mu$s left in the SIFS latency at most 1.4 $\mu$s can be averaged over the BITS and TXVECTOR tokens, which is 700 ns per token.

In total, the SAMPLE tokens are recommended to have no extra latency. The SOFTBITS and RXVECTOR tokens can have an extra latency of at most 200 ns and the BITS and TXVECTOR token at most 700 ns. The corresponding task-graph latencies with the new worst-case communication times are listed in Table 7.8. As expected both the SIFS and header-loop latencies are reaching their limit. The other latencies are (almost) the same.

<table>
<thead>
<tr>
<th>Latency description</th>
<th>Latency</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Interframe Space</td>
<td>15.753</td>
<td>16.000</td>
</tr>
<tr>
<td>Header-loop</td>
<td>3.987</td>
<td>4.000</td>
</tr>
<tr>
<td>Gain Control by Detect</td>
<td>1.411</td>
<td>1.600</td>
</tr>
<tr>
<td>Gain Control by Gain</td>
<td>1.348</td>
<td>1.600</td>
</tr>
<tr>
<td>ACK Header</td>
<td>11.919</td>
<td>17.323</td>
</tr>
<tr>
<td>ACK Data</td>
<td>14.214</td>
<td>21.323</td>
</tr>
</tbody>
</table>

In a realistic network this distribution of extra latencies can also be used to define the priorities of tokens. SAMPLE tokens should have the highest priority and BITS and TXVECTOR tokens...
the lowest. Using priorities will automatically introduce latencies, and as long as these extra latencies are within the bounds described above, then there will be no problems.

7.4 One radio configuration

Instead of multiple radios the system can also be running with only one WLAN radio. This better represents normal usage, since most of the time only a single network connection is used. The simplest way to analyze this is to change the value of \( N \) to 1 and leave all other parameters the same. The resulting task times are listed in Table 7.9.

The most dominant changes are the startup times of the tasks. The startup times of most tasks depend on the number of other radios that have to be processed before the current task can be executed. However in this situation there are no other tasks. For the tasks mapped on the EVP this means that the startup time only depends on the period of the timers. For the tasks on the software codec it means that the WLAN tasks only have to wait for a single background task that might be running when they get activated. This time will, as state before, not exceed the longest WLAN task mapped on the SWC.

Because these startup times have gone down so much the corresponding latencies should have gone down dramatically. The latencies are listed in Table 7.10 and depicted in Figure 7.6, where they are put next to the \( N = 4 \) numbers. As expected the latencies have all dropped quite a bit. The SIFS latency has gone down with 5.7 \( \mu \)s and is now at almost half of what it may be. The header-loop latency has decreased with almost 1 \( \mu \)s.

The short-period tasks are also affected now, so the gain latencies have decreased as well with around 400 ns. This means that when the Detect task is finished at most 20 samples will be waiting with the old value. This is half a short-symbol less than before.

The ACK latencies are now more than safe since they both decreased with more than 5 \( \mu \)s. Their limits have, like the gain latencies, decreased with around 400 ns, which is not much relatively. So the ACK latencies are now at less than 40 % of their maximum.

7.4.1 System load

Of course, running only one radio makes the load on the system quite low. The minimum required values for the platform parameters can be used to compute this load. For \( N = 1 \) the minimum required operating frequency of the EVP (Equation 6.9) is only 128 MHz, while it is running at 400 MHz. This means it is loaded for only 32 %.

\[
F_{EVP} = F_{EVP,\text{short}} = \frac{230 + 1 \times 280}{4 \, \mu s} = \frac{510}{4 \, \mu s} = 127.5 \, \text{MHz}
\]

The ARM micro-processor is doing the least amount of work. Its required frequency (see Equation 6.31) is only 15 MHz but it is running at 150 MHz, so the load is a mere 10 %.

\[
F_{\text{ARM}} = \frac{1 \times 60}{4 \, \mu s} = 15 \, \text{MHz}
\]

The load of the software codec is equal to 20 %, because it is at most performing one Decode or Code task every 4 \( \mu \)s, which takes only 0.8 \( \mu \)s. Finally, the minimum required network speed is 960 Mb/s (see Equation 6.2), but it is running at 4 Gb/s, so it is loaded for only 24 %.
Table 7.9: Task times (in ns) for $N = 1$

<table>
<thead>
<tr>
<th>Task</th>
<th>$P$</th>
<th>$S$</th>
<th>$R$</th>
<th>$T$</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Tx</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Detect</td>
<td>138</td>
<td>800</td>
<td>938</td>
<td>213</td>
<td>4.800</td>
</tr>
<tr>
<td>Gain</td>
<td>75</td>
<td>800</td>
<td>875</td>
<td>150</td>
<td>800</td>
</tr>
<tr>
<td>TSync</td>
<td>75</td>
<td>800</td>
<td>875</td>
<td>150</td>
<td>800</td>
</tr>
<tr>
<td>Preamble</td>
<td>100</td>
<td>800</td>
<td>900</td>
<td>175</td>
<td>16.000</td>
</tr>
<tr>
<td>Demod</td>
<td>688</td>
<td>4.000</td>
<td>4.688</td>
<td>975</td>
<td>4.000</td>
</tr>
<tr>
<td>Mod</td>
<td>563</td>
<td>4.000</td>
<td>4.563</td>
<td>763</td>
<td>4.000</td>
</tr>
<tr>
<td>FSync</td>
<td>651</td>
<td>4.000</td>
<td>4.651</td>
<td>938</td>
<td>4.000</td>
</tr>
<tr>
<td>Decode</td>
<td>800</td>
<td>800</td>
<td>1.600</td>
<td>800</td>
<td>4.000</td>
</tr>
<tr>
<td>Code</td>
<td>800</td>
<td>800</td>
<td>1.600</td>
<td>800</td>
<td>4.000</td>
</tr>
<tr>
<td>MACi</td>
<td>400</td>
<td>0</td>
<td>400</td>
<td>400</td>
<td>4.000</td>
</tr>
<tr>
<td>MACo</td>
<td>333</td>
<td>0</td>
<td>333</td>
<td>333</td>
<td>4.000</td>
</tr>
</tbody>
</table>

Table 7.10: Latencies (in ns) for $N = 1$

<table>
<thead>
<tr>
<th>Latency description</th>
<th>Latency</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Interframe Space</td>
<td>8.419</td>
<td>16.000</td>
</tr>
<tr>
<td>Header-loop</td>
<td>2.600</td>
<td>4.000</td>
</tr>
<tr>
<td>Gain Control by Detect</td>
<td>1.000</td>
<td>1.600</td>
</tr>
<tr>
<td>Gain Control by Gain</td>
<td>937</td>
<td>1.600</td>
</tr>
<tr>
<td>ACK Header</td>
<td>6.471</td>
<td>16.912</td>
</tr>
<tr>
<td>ACK Data</td>
<td>6.866</td>
<td>20.912</td>
</tr>
</tbody>
</table>

Figure 7.6: Latencies (in ns) for $N = 4$ and $N = 1$
7.4.2 Slower SWC

Obviously, the platform is not loaded very much. This leaves some room for changing some parameters. In particular the processing times of the SWC, since it will be a tough job to design a software codec that is fast enough to deal with the Decode task in only 800 ns. For only one WLAN radio the load of the SWC with this processing time is quite low, so it can clearly be loosened. The only two things to take into account are the SIFS and header-loop latencies. The other latencies will not be a big problem.

The processing times of the Decode and Code tasks can even be increased to 1.4 µs without problems. The corresponding latencies are listed in Table 7.11 and put next to the original one radio configuration in Figure 7.7.

The gain latencies are not affected by this change, since the EVP performance stayed the same. The other latencies all increased with 1.2 µs, which are caused by the increase in response times of the Decode and Code tasks. Both the startup and the processing times have increased with 600 ns and the response time is the sum of that.

These numbers are still within bounds. The only latency of concern is the header-loop, which is only 200 ns below its limit. Any big change to for example the communication network and the header symbol will be too late. However in this situation it might not be such a big problem. Although the SIFS latency is increased with 4 µs when this happens, the result will still be less than the 16 µs available. The processing time can even be increased to 2.5 µs. The SIFS latency will then be around 15.8 µs including the extra header-loop cycle.

<table>
<thead>
<tr>
<th>Latency description</th>
<th>Latency (ns)</th>
<th>Constraint (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Interframe Space</td>
<td>9.619</td>
<td>16.000</td>
</tr>
<tr>
<td>Header-loop</td>
<td>3.800</td>
<td>4.000</td>
</tr>
<tr>
<td>Gain Control by Detect</td>
<td>1.000</td>
<td>1.600</td>
</tr>
<tr>
<td>Gain Control by Gain</td>
<td>937</td>
<td>1.600</td>
</tr>
<tr>
<td>ACK Header</td>
<td>7.671</td>
<td>16.912</td>
</tr>
<tr>
<td>ACK Data</td>
<td>8.066</td>
<td>20.912</td>
</tr>
</tbody>
</table>

7.4.3 Disabling timers

For WLAN radios only one task will be active on the EVP per radio. This means that when the radio has a long-period task active, its short-period timer will have nothing to do, this also holds
vice versa. The timers will however still be running and add extra load on the EVP. If there is a
way that the timers are disabled and enabled when needed then several things can go down.

First of all the minimum required operating frequency will be defined by Equation 7.2. The
minimum for this situation is 106.25 MHz, which is almost 22 MHz less than before. This will
reduce the maximum load of the EVP with another 5 %. Furthermore, the processing times will
go down, which will result in lower latencies. These are only minor changes. For example, the
SIFS latency will go down to 8.331 ns, which is only 88 ns less than before.

Because of these reductions the frequency of the EVP can either be lowered to safe energy, or
the system can be used more by other tasks.

\[
F_{EVP,\text{long}} = \frac{I_{\text{preempt}} + N \times (I_{\text{check}} + I_{\text{Demod}})}{T_{\text{long}}}
\]

\[
F_{EVP,\text{short}} = \frac{5 \times (I_{\text{Interrupt}} + N \times (I_{\text{check}} + I_{\text{Detect}}))}{T_{\text{long}}}
\]

\[
F_{EVP} = \max (F_{EVP,\text{long}}, F_{EVP,\text{short}})
\]

(7.2)

\[
= \max \left( \frac{80 + 1 \times (5 + 235)}{4 \ \mu s}, \frac{5 \times (30 + 1 \times (5 + 50))}{4 \ \mu s} \right)
\]

\[
= \max (80 \ MHz, 106.25 \ MHz)
\]

= 106.25 MHz

7.4.4 Minimum parameters

A minimum configuration for a platform that will be capable of running a single radio WLAN
radio on a shared platform can be defined by using a combination of the above changes.

As indicated, the minimum required parameters are much lower than currently used, so the
frequencies of both the EVP and ARM can be lowered to 150 MHz. The network speed is kept
at 4 Gb/s but the extra latencies as discussed for the 4 radio configuration are added. The SWC
maximum processing times are set at 1.4 µs. It is also assumed that the timers of the EVP can
be disabled as described. This configuration can be summarized as follows.

- \( F_{EVP} = 150 \ MHz \)
- \( F_{ARM} = 150 \ MHz \)
- \( F_{Network} = 4 \ Gb/s \)
- \( P_{\text{decoder}} = P_{\text{coder}} = 1.4 \ \mu s \)
- EVP timers can be disabled
- All tokens, except for SAMPLE tokens, have a worst-case communication time increased
  with 200 ns.

The latencies corresponding to this configuration are listed in Table 7.12. The result is that
all latencies except for the header-loop latency are within their bounds. Since the header symbol
will be too late, the worst-case SIFS latency will be 4 µs higher. This means it is actually 15.3
µs, which is (although quite high) still below the maximum allowed.

The load of this configuration is as follows. The minimum required frequency of the EVP is
106.25 MHz (see Equation 7.2), so its load is \( \frac{106.25}{150} = 71 \ % \). The load of the SWC is equal to
\( \frac{1.4 \ \mu s}{4 \ \mu s} = 35 \ % \) and for the ARM it is still 10 %. The network is loaded for only \( \frac{960 \ Mb/s}{4 \ Gb/s} = 24 \ % \).
Table 7.12: Latencies (in ns) for minimal $N = 1$ configuration

<table>
<thead>
<tr>
<th>Latency description</th>
<th>Latency</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Interframe Space</td>
<td>11.298</td>
<td>16.000</td>
</tr>
<tr>
<td>Header-loop</td>
<td>5.112</td>
<td>4.000</td>
</tr>
<tr>
<td>Gain Control by Detect</td>
<td>1.229</td>
<td>1.600</td>
</tr>
<tr>
<td>Gain Control by Gain</td>
<td>1.062</td>
<td>1.600</td>
</tr>
<tr>
<td>ACK Header</td>
<td>8.575</td>
<td>17.079</td>
</tr>
<tr>
<td>ACK Data</td>
<td>9.170</td>
<td>21.079</td>
</tr>
</tbody>
</table>

7.5 Summary

The analysis done in this chapter is done in several steps. First a configuration of the platform was defined as a starting point, which was based on the minimum required parameters that meet the throughput constraints. However, the latency requirements were not satisfied, so the effects of the parameters have been analyzed. As it appears the performance of the EVP and ARM have the greatest impact on the latencies. Changing the network performance on the other hand, did not show any considerable effect.

Based on these results a recommended platform configuration is defined for supporting four radios while it is shared with other tasks. The parameters in this configuration are as follows.

\[
F_{EVP} = 400 \text{ MHz} \\
F_{ARM} = 150 \text{ MHz} \\
F_{Network} = 4 \text{ Gb/s} \\
P_{decoder} = 800 \text{ ns} \\
P_{coder} = 800 \text{ ns}
\]

The most important latencies, namely SIFS and header-loop, are respectively $1.8 \mu s$ and $0.4 \mu s$ below their maximum. The load of the platform will be at most 84 % for the EVP, 80 % for the SWC and 40 % for the ARM. This gives enough room for improvements, like a more realistic network model.

It is additionally shown that a single instance of the WLAN application running on this configuration will have a very good performance. The load of the platform will then be at most 32 % for the EVP, 20 % for the SWC and a mere 10 % for the ARM. The latencies in this situation are very low, in some cases nearly 50 % of their maximum. This allows a reduction of the platform performance with the SWC in particular.
Chapter 8

Simulation

The performed analysis only deals with the theoretical possibility of a multi-channel wireless LAN implementation. In this chapter some simulations will be performed to verify that it is actually working. The simulations are performed using the CASSE environment.

8.1 Implementation

An attempt has been made to implement the model directly in CASSE. However, CASSE is still in development and therefore misses some features that are needed for the correct implementation. As a result the implemented model has a few major differences with the actual model.

First of all, communications are controlled directly by the processor and not by a separate controller. This means that while a processor is transmitting a token it has to write the content of the token in the memory block of the other processor. Any delay that the network introduces makes the processor wait longer. Meanwhile the task is blocked until the operation is finished. This can easily be off-loaded to a separate controller that transmits tokens to a remote processor. However CASSE does not easily support this. To overcome this limitation the network is implemented to not use any time at all. This is similar as a network speed $F_{\text{Network}}$ equal to infinity. The resulting latencies will therefore not be as accurate. Afterwards conclusions should be made what the effects will be of delayed communications.

CASSE also does not support advanced schedulers on the processors. The recommended scheduling policy for the EVP, which is based on two different timers, can for instance not be implemented with CASSE itself. As a workaround two tasks are created, EvpLong and EvpShort. EvpShort deals with the short-period tasks and is connected to an interrupt line. A timer triggers this interrupt regularly at 800 ns intervals. When EvpShort is executed it simulates a context switch, checks the input channels and executes the corresponding tasks if applicable. The other task, EvpLong, simulates the long-period tasks. It is running continuously and starts by waiting for the start of a period. It will then also simulate a context switch, check the channels and execute the tasks. When it is finished processing the tasks it starts waiting again for the next period. The time it is waiting should be perceived as idle time that can be utilized by a background task.

Something similar is done with the tasks of the SWC and ARM processors. A single task called Codec is created that deals with the Code and Decode tasks and executes an idle task if no work is available. For the ARM a task called Mac is created that either waits indefinitely for BITS tokens and executes the MACi task accordingly or, when put in the correct state, it will activate the MACo task. The Rx and Tx tasks are also put into one task, called Rf, which is instantiated four times and mapped on the RF processor. A time division multiple access (TDMA) scheduler makes sure that each instance is executed exactly one time each period.

All these CASSE tasks simulate the WLAN tasks of all radios together. This means that all the input and output channels of all corresponding tasks are connected to these single instances. For example, with N radios, the Codec task has N incoming and outgoing SOFTBITS channels,
N incoming and outgoing BITS channels, N outgoing RXVECTOR channels and N incoming TXVECTOR channels. In total it is connected to N * 6 channels. This also holds for the other tasks. The entire model is visualized in Figure 8.1. Obviously this gets quite unmanageable for high numbers of N.

![Figure 8.1: CASSE simulation model](image)

Each CASSE task prints some statistics about what it is doing and how long it took. For instance, the time it takes to execute each (model) task is printed. There are also several moments in time recorded, like the time that the first sample of an incoming or outgoing frame arrives. These numbers can afterwards be used to compute the achieved throughputs and latencies.

Finally, there is a difference with a real implementation in the handling of the SIFS latency. Normally the SIFS deadline is quite strict. A station should transmit the ACK frame after 16 µs (± 10%) after the end of the incoming frame. So when the samples are available earlier, the station should postpone transmission until the correct moment. This is not done in these simulations; the samples are processed as soon as possible. This will make it easier to show whether the system can deal with the SIFS latency.

### 8.2 Simulations

The simulations can be configured with some parameters. The processors are configured according to the recommended configuration with four radios. Hence, the EVP is running at 400 MHz, the ARM at 150 MHz, the Code and Decode tasks have a processing time of 800 ns and the network has a throughput of 4 Gb/s.

Other parameters define the initial state a radio can be in. The simulation will go through the state diagram and at the end return to the initial state (Frame Detection). When the radio is not transmitting a frame the Rx task will number the samples it generates. During a simulation each radio can receive a frame at a certain sample index. The indexes of the samples that the Detect task processes are compared to the configured index and it will trigger a frame detection if they are equal.
8.3. ALL RADIOS DETECTING

The MACo task is executed when a radio is in the correct state. This state is either set initially or by the MACi task when it has received the last symbol of a frame. When activated the MACo task sends out a frame of static length, representing an ACK frame.

The data in a frame will not be inspected, since the actual processing in the tasks is simulated with a static delay. The information in the header symbols will therefore be statically defined. The length will also not indicate the length in bytes, but in number of data OFDM symbols.

All simulations are run for a period of 200 $\mu$s. This will fit 250 short periods and 50 long periods. However, the timer for the short-period tasks sends the first trigger 800 ns after the start and not at timestamp 0. As a result only 249 short-periods are recorded.

Each period the EvpLong and EvpShort tasks simulate the context switches and checks with a static delay. For EvpShort this is $I_{\text{interrupt}} + N \cdot I_{\text{check}} = 50$ instruction cycles and for EvpLong $I_{\text{preempt}} + N \cdot I_{\text{check}} = 100$ instruction cycles. Since the EVP is running at 400 MHz this means that every period the EvpShort task spends 125 ns on context switching and checks. For the EvpLong task this is 250 ns. The total overhead during the entire simulation will be $249 \cdot 50 I_{\text{EVP}} = 31.125$ ns and $50 \cdot 100 I_{\text{EVP}} = 12.500$ ns respectively.

8.3 All radios detecting

The first situation that is simulated will have all radios in the idle state the entire time. Hence, all radios only have the Rx and Detect tasks active. For the other tasks no time is recorded. The SWC only executes its background task and the ARM will be idle the entire time.

As explained, the overhead of the EvpShort task already requires a total processing time of 31.125 ns. In addition, four Detect tasks are executed every period of EvpShort. This means that the Detect task in general is executed 996 times, which should result in a total processing time of $996 \cdot I_{\text{Detect}} I_{\text{EVP}} = 124.500$ ns. The simulation recorded this exact number.

Including the overhead, the EvpShort task spends 155.625 ns in total. Per period this is on average 625 ns, which equals exactly to 250 instructions; 50 instructions overhead and four Detect task executions of 50 instructions each.

Every time the Detect task is executed, a number of tokens are left in the channel that have not been processed yet or have arrived during the execution. If the Detect task would set the gain level then this number would also be the amount of samples using the old gain level. So the number of remaining tokens can be viewed as the first gain latency. During this simulation this number varied between 2 and 11 tokens, which corresponds to a latency between 100 and 550 ns. The maximum is computed to be 1.411 ns, so this is well within the limits. The number of remaining tokens cannot be lower than 2 since the processing time of the Detect task is 125 ns and samples arrive every 50 ns.

Beside the EvpShort, the EvpLong task has also been running. This task was only performing the checks and not executing any tasks. The overhead of this task is computed to be 12.500 ns. However, the simulation recorded 58.750 ns in total. This is 1.175 ns per period instead of the 250 ns as computed. This is entirely caused by the preemptions of the EvpLong task by the EvpShort task. On average it is preempted $\frac{1.175}{250} = 1.48$ times per period.

The EvpShort and EvpLong tasks together spend a total of 155.625 + 12.500 = 168.125 ns in a period of 200 $\mu$s, which is a load of 84.1 %. This is a bit less than the 84.4 % load as computed with the minimum required operating frequency of the EVP. However, since the simulation records only 249 periods instead of 250 periods the total processing time of the EVP is actually 625 ns less than it should in such a period. With one period the load will be 84.4 %.

8.4 One radio receiving a frame

Next, the effect of receiving a frame is simulated. All the radios are still in the frame detection state. However, for radio one an incoming frame of 16 data OFDM symbols starts at sample 500. This will make the Detect task trigger the other tasks and eventually the MACo task will start
transmitting an acknowledgement frame. This ACK frame will contain 6 data OFDM symbols. Once the samples of the ACK frame are transmitted completely by the Tx task the Frame Detection state starts again.

**Short period tasks**

The simulation recorded that the Detect task is executed 827 times in total, which is 169 times less than when no frame is received. In other words, there is a gap between the moment that the Detect task stops and the moment it starts again of \(169 \times 800 = 135,200\) µs.

After the Detect task found a frame the Gain task is executed twice. The TSync task needs 7 times to find the start of the long training symbols. For each of these tasks the expected processing time can be computed. These are listed in Table 8.1 together with the recorded values. For this simulation there is a slight difference between them. This is caused by a rounding error. The simulation records timestamps in nanoseconds, however one clock cycle of the EVP is 2.5 ns long. When printed the decimals after the comma are discarded, which results in an error. So the times of the simulation are correct. Per short period an average of 543 ns is spend by the EvpShort task. The minimum time is 500 ns which corresponds to a short period where three Detect tasks are executed. The maximum is 625 ns which is a period with four Detect executions.

<table>
<thead>
<tr>
<th>Task</th>
<th>Count</th>
<th>Expected Time</th>
<th>Recorded Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect</td>
<td>827</td>
<td>103.375 ns</td>
<td>103.375 ns</td>
</tr>
<tr>
<td>Gain</td>
<td>2</td>
<td>125 ns</td>
<td>124 ns</td>
</tr>
<tr>
<td>TSync</td>
<td>7</td>
<td>438 ns</td>
<td>435 ns</td>
</tr>
<tr>
<td>Preamble</td>
<td>1</td>
<td>88 ns</td>
<td>87 ns</td>
</tr>
<tr>
<td>Overhead</td>
<td>249</td>
<td>31.125 ns</td>
<td>31.125 ns</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>135.150 ns</strong></td>
<td><strong>135.146 ns</strong></td>
</tr>
</tbody>
</table>

**Long period tasks**

The FSync task that follows is executed two times. Then the header symbol and 16 data symbols are processed by the Demod task. The expected and recorded times of the long period tasks are listed in Table 8.2. The recorded times are much higher than the expected times, which is caused by the preemptions of the EvpLong task by the EvpShort task.

Since the three other radios are in the Frame Detection state EvpShort task will need at least 500 ns every short period. This effectively leaves 300 ns for the EvpLong task.

The Demod task is the longest task with 588 ns of processing time. Including the overhead this means 838 ns for the EvpLong in total. This will be interrupted at least 2 times by the EvpShort task making the total processing time at least 1.888 ns. However, in a worst-case situation the complete EvpLong task gets interrupted 3 times making it effectively 2.338 ns long. This is also recorded by the simulation as the maximum processing time of the EvpLong task.

The difference between the expected and recorded times is 49 µs which is caused by preemptions of 500 ns, meaning that the EvpLong task has been preempted exactly 98 times. On average this is almost two times per period.

<table>
<thead>
<tr>
<th>Task</th>
<th>Count</th>
<th>Expected Time</th>
<th>Recorded Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSync</td>
<td>2</td>
<td>1.100 ns</td>
<td>3.100 ns</td>
</tr>
<tr>
<td>Demod</td>
<td>17</td>
<td>9.988 ns</td>
<td>26.994 ns</td>
</tr>
<tr>
<td>Mod</td>
<td>7</td>
<td>3.238 ns</td>
<td>6.738 ns</td>
</tr>
<tr>
<td>Overhead</td>
<td>50</td>
<td>12.500 ns</td>
<td>38.999 ns</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>26.825 ns</strong></td>
<td><strong>75.831 ns</strong></td>
</tr>
</tbody>
</table>
8.5. INCOMING DATA SYMBOLS

Processor loads

In total the two EVP tasks require $135.150 + 26.825 = 161.975$ ns of processing. This is an average load of 81%.

The other two processors, SWC and ARM, do not show any dramatic situations. The SWC has to deal with 17 Decode tasks and 7 Code tasks, which would require a total of $19.2 \mu s$ of processing. This is exactly recorded by the simulation. This means the SWC is only loaded for 9.6% over the entire period.

The ARM is even loaded less. It has to deal with 17 MACi tasks and 7 MACo tasks, which in total require only $9.1 \mu s$ of processing time. Over the full period this is a load of 4.6%.

Latencies

The simulation recorded several moments in time that can be used for the latency calculations. The first one is the SIFS latency which is between the moments that the last sample of the incoming frame is processed by Rx and the first sample of the ACK frame is processed by Tx. It is recorded at only $7.9 \mu s$, which is much less than the computed worst-case of $14.2 \mu s$.

The second important latency, the header-loop, is not recorded explicitly, but it can be verified from two other facts. First of all, the time between the starts of the first and second execution of the Demod task is equal to $4 \mu s$, meaning that it did not have to wait for the RXVECTOR token. Furthermore, the time that the first Decode task is finished is only $2.57 \mu s$ after the start of the corresponding Demod task. After this only the RXVECTOR token has to be transmitted back to the Demod task, which (in this case) takes no extra time.

The gain latencies are as before recorded by the number of tokens left in the channel after the Detect or Gain task finishes. The gain latency with the Detect task is here between 100 and 850 ns and with the Gain task it equals 650 ns. All well below the maximum of $1.4 \mu s$.

After the frame is received the MACo task sends out a TXVECTOR token which makes the Preamble transmit its 320 SAMPLE tokens. The moment the first of those samples is processed by the Tx task is recorded. The Code task will simultaneously construct a SOFTBITS token for the Mod task, resulting in the first OFDM symbol. The moment that these samples are processed is also recorded. The time between these moments should not exceed $16 \mu s$, which would indicate that the samples came too late. In this simulation this is not the case. The Mod task actually produces the SAMPLE tokens $11.4 \mu s$ before the preamble has been processed completely.

This header symbol is followed by the first data symbol, which should be available before the header symbol has been processed by the Tx task. In this case, it is again produced $11.4 \mu s$ in advance. In other words, the ACK latencies are, as predicted in the analysis, well within the limits.

In this simulation a frame is received of 16 data symbols including the preamble and the header symbol. This would take a total of $16 + (16 + 1) \cdot 4 = 84 \mu s$ to receive. Transmitting the ACK frame would take another $16 + (6 + 1) \cdot 4 = 44 \mu s$. Together with the SIFS latency this would add up to a period of $84 + 7.9 + 44 = 135.9 \mu s$ during which the Detect task is not executed. This corresponds to at least 169 periods of 800 ns which is equal to the recorded times the Detect task is not executed.

8.5 Incoming data symbols

The previous simulation involved both receiving and transmitting a frame. It would also be interesting to see what the load of the system is when all radios are receiving a frame. So in this simulation all four radios have a frame of 999 data symbols starting at sample 0.

This would result in a Detect task that stops after its first execution and never again executed during the entire simulation. The other short period tasks are also executed only a few number of times and then never executed again. The Preamble will not be executed at all, since the frame is longer than the simulation period and therefore no outgoing frame will be produced.
However, only the symbol processing is relevant here, so the preamble processing should be ignored. This is done by shifting the start and end times of the inspected period of the simulation. Instead of analyzing all data, only data between two points are used. In this case the period will start at the beginning of the eighth long period and ends 40 periods later, which is a period of 160 μs long.

During this simulation only the Demod task is active on the EVP. This means that the EvpShort task is not executing any tasks and is only performing its regular checks. This still requires 125 ns each period. The task is executed 200 times, so the overhead this causes needs a total processing time of 25 μs. Per short period 675 ns is left for the EvpLong task.

Long period tasks

At each execution of the EvpLong task it processes four Demod tasks. This means a total of 160 executions of the Demod task. One Demod task needs at least 587.5 ns of processing time. Including the overhead the EvpLong needs at least $4 \times 587.5 + 250 = 2.580$ ns per period. However, only 675 ns of processing is available each short period, which means it gets preempted on average 3.9 times. This should result in a minimum processing time of $2.580 + 375 = 2.975$ ns and a maximum of $2.580 + 500 = 3.100$ ns. The latter corresponds to the worst-case period computed during the analysis.

The simulation shows the same numbers. The total expected and recorded execution times are listed in Table 8.3. The task is preempted by the EvpShort task for 17.5 μs, which means a total of 140 times. On average this is exactly 3.5 times. As expected, the minimum recorded processing time of the EvpLong task is 2.975 ns and the maximum 3.100 ns.

<table>
<thead>
<tr>
<th>Table 8.3: Long period execution times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>Demod</td>
</tr>
<tr>
<td>Overhead</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

Processor loads

In total the EVP spends $25 + 104 = 129$ μs on WLAN radio processing. On a period of 160 μs this means a load of 80.6 %, which corresponds exactly to the result of Equation 6.7.

Like the Demod task the Decode task is also executed 160 times. This means that each long period the SWC has to execute four Decode tasks that take at most 800 ns. And since the Decode task is the longest task on the SWC the background task will not exceed the processing time of the WLAN tasks, at least one background task can be executed in the remaining time. This is also recorded by the simulation. Each period exactly four Decode and one other task is executed. This means the WLAN radios cause a load of $\frac{4}{5} = 80$ %.

The ARM only has to deal with the MACi task, which takes 400 ns per execution. In total it is executed 160 times, resulting in 64 μs of processing. This is a load of 40 %.

Latencies

During this simulation no complete frame is received and no ACK frame is returned, so none of the latencies apply. However what can be observed is the time between the start of the Demod task and the completion of the corresponding MACi task. On average this is only 2.743 ns and the maximum is 2.935 ns. This means that every symbol is actually completely processed within one symbol period. This means the latency is very minimal.
8.6 Outgoing data symbols

Instead of processing incoming symbols it is also interesting to see the load of continuously processing outgoing symbols.

This is simulated by letting all radios transmit a frame of 999 data symbols right from the beginning of the simulation. Again only a period of 40 long periods is recorded where only data symbols are processed. No preamble processing is included.

The processing of the EvpShort task is still the same. It spends 125 ns on functional overhead every short period, which leaves 675 ns for the EvpLong task. In total it again needs 25 µs.

The EvpLong task is now only executing the Mod task, which needs at least 462.5 ns per execution. In total the EvpLong task would need $4 \times 462.5 + 250 = 2.100$ ns of processing per period. This means it can get preempted at least 3 times and at most 4 times, which results in an actual processing time between 2.475 and 2.600 ns. The latter again corresponds to the computed values from the analysis.

The expected and recorded times are listed in Table 8.4. Again the task is preempted for 17.5 µs, which means 3.5 preemptions per period. The minimum and maximum total processing times of the EvpLong task show the same values as computed above.

<table>
<thead>
<tr>
<th>Task</th>
<th>Count</th>
<th>Expected Time</th>
<th>Recorded Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mod</td>
<td>160</td>
<td>74.000 ns</td>
<td>84.080 ns</td>
</tr>
<tr>
<td>Overhead</td>
<td>40</td>
<td>10.000 ns</td>
<td>17.420 ns</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>84.000 ns</td>
<td>101.500 ns</td>
</tr>
</tbody>
</table>

Processor loads

The EVP spends a total of 25 + 84 = 109 µs on processing WLAN radios. With a period of 160 µs it is a load of only 68.1 %. This means that processing outgoing symbols instead of incoming the load of the EVP is reduced with 3.1 % per radio.

The load of the SWC stays the same, since the processing time of the Code task is the same as the Decode task. So the load is still 80 %.

The ARM load is reduced a bit too. The MACo task only needs 333 ns of processing per execution. In total it spends 53.3 µs on WLAN tasks, which is a load of only 33.3 %.

Latencies

The latencies do also not apply to this situation. Only the time between the start of the MACo task and the completion of the corresponding Mod task can be inspected. During this simulation it varied between 8,290 ns and 8,995 ns, with an average of 8.674 ns. This is much higher than in the other direction. One cause for this is the startup of an outgoing frame. The first data symbol is produced by the MACo task right after it transmitted the TXVECTOR token. This means that before the Code task even constructed and produced the SOFTBITS token of the header, there is already a BITS token waiting. Luckily there is enough time left since the Tx task only starts on the first symbol after it has completely processed the 16 µs preamble.

8.7 All downloading a frame

The latencies involved with incoming frames can be simulated with another situation. Again all radios start in the detection state. However, this time each radio will detect a frame after a 30, 115, 63 and 500 samples respectively. These frames are 8, 13, 10 and 1 data symbols in length, so in total 4 header and 32 data symbols need to be processed. Each radio returns an ACK frame of 6 data symbols long, so in total 4 outgoing header and 24 outgoing data symbols are processed.
During the simulation, the time between processing the last incoming sample and the first outgoing sample, i.e. the SIFS latency, varied between 5.6 µs and 6.6 µs. This means that it is less than half the computed worst-case 14.2 µs.

The header-loop latency shows no problems either. First of all, for all radios the first and second Demod executions follow each other without delay. And secondly, the time that the Decode task finishes processing the first SOFTBITS token is at most 2.5 µs after the corresponding Demod task started on the samples. In other words, for all radios the RXVECTOR is available in time.

The gain latencies vary quite much. The number of tokens that are still in the channel the moment the Detect task finishes is between 3 and 24, which corresponds to a latency of 150 and 1.200 ns. It should be remembered that this simulation does not count any communication time, so the actual simulated value should be a bit higher. This makes it come close to the computed maximum of 1.411 ns. The gain latency for the Gain task itself varies between 150 and 700 ns, which is much better.

The ACK latencies are well within the limits. For all four radios the Tx task can start processing the samples of the header symbol right after it finishes the preamble. The first data symbol is also available before it finishes the header symbol. This can also be observed by the times between the start of the MACo task and the end of the corresponding Mod task. In this simulation this time varied between 5.6 and 9.8 µs with an average of 8.5 µs. As explained before, this is mainly caused by the start of a frame transmission. Still, these numbers are well below the 16 µs it takes the Tx to process the preamble.

### 8.8 All downloading a frame at the same time

The latencies might be a problem when all radios are receiving a frame at the same time. So it in this simulation all four radios start in the detection state and simultaneously find a frame of 12 data symbols at sample 500. This means that a total of \( (1 + 12) \times 4 = 52 \) symbols need to be processed for the incoming frames and again 36 symbols for the outgoing ACK frames.

This time the SIFS latency should be higher, since the MACi and MACo tasks of the radios are interfering more with each other. This can also be observed in the recorded times. The SIFS latencies are between 7.1 µs and 10.4 µs, which is at most 3.8 µs higher than in the previous situation. Although closer to the maximum, they are still at least 3.7 µs below it.

The other latencies also show no problems. All Demod tasks are executed after each other, without having to wait for RXVECTOR tokens. The Decode task is also ready no later than 2.5 µs after the Demod task started. In other words, the header-loop latency is 1 µs below its maximum.

The gain latencies are also in order. The Detect task has between 2 and 17 tokens left in the channel when it finishes, which corresponds to a gain latency between 100 and 850 ns, well below the limit. The Gain task leaves between 13 and 18 tokens in the channel; a bit higher, but still below the maximum.

Finally the ACK latencies; they also show no problems at all. The Tx task can start on the header and first data symbols at the correct time, so the samples are ready in time. Furthermore, the latencies between the start of the MACo task and the end of the corresponding Mod task are no greater than 9.6 µs, which is well below the duration of the preamble.

### 8.9 Summary

After several simulations it appears that the computed worst-case situations are not easily achieved. Only a few measured units approach their theoretical maximum. There are several causes for this. First of all the model is not simulated realistically. The communication network does not have any delay, which will make the latencies lower than they should actually be.

The SIFS latency, for instance, has five tokens to transmit. According to the theoretical values this will need at least 500 ns. Therefore in these simulations the worst SIFS latency of 10.4 µs
should actually be 10,9 µs. This is still 4 µs lower than the computed maximum.

The other latencies will also be influenced by this. The header-loop latency did not show any problem so far. In every situation the maximum delay between the processing of the header OFDM symbol by the Demod task and the reception of the corresponding RXVECTOR token did not exceed 2,5 µs. According to the analysis the communication of the tokens should at least take 300 ns, which will make the simulated maximum still 0,7 µs lower than its maximum of 3,5 µs.

The gain latencies have not been recorded directly. It is estimated according to the number of SAMPLE tokens left in the input channel after the Detect or Gain task finishes. The highest number recorded is 24 SAMPLE tokens. This comes very close to the computed maximum of 1,411 ns (28 tokens). When the communication time of a token is added, then it is only 3 tokens below that maximum.

In the analysis the ACK latencies never posed any problems, which was also the case in the analysis. Every time the header symbol and first data symbol were ready before the Tx task started processing them. The worst case time between the start of the MACo task and the end of the corresponding Mod task never exceeded 9,8 µs. This still stays well below the computed 13,3 µs if an extra communication time of 350 ns is included.

8.9.1 Loads

On the other hand, the loads of the system have been simulated quite accurately. The maximum recorded periods of EvpLong and EvpShort correspond well to the computed maximum. For instance when all the radios are processing an incoming symbol, then the EvpLong task needs at most 3,1 µs which was also computed during the analysis.

The maximum loads of the various processors depend on the state of the radios. If all radios are in the Frame Detection state the load of the SWC and ARM processors will be zero. The EVP on the other hand will be loaded the most with 84,4 %.

Once a frame is being received and the data OFDM symbols are being processed the load of the other processors go up. Per radio this is 20 % for the SWC and 10 % for the ARM processor. The load of the EVP is reduced slightly in this situation with about 1 % per radio.

When all radios are transmitting a frame, then the load of the EVP will be the lowest at only 68,1 % in total. Per radio this means that compared to the detection state the EVP load is reduced with 4 % for each radio that is transmitting a frame. The load of the SWC is the same as during reception of symbols, since both its WLAN tasks are equally complex. The ARM is loaded slightly less with in total only 33,3 %.

In other words, the load of the EVP will always be between 68,1 % and 84,4 %. The load of the SWC will be between zero and 80 %. The ARM will only be loaded for at most 40 %.

8.9.2 Improvements

A more realistic implementation of the recommended platform can only be achieved when several of the CASSE limitations are resolved. This involves a more realistic communication model. The representations of the operating systems running on the processors must also be simulated better. The schedulers are too limited. And finally the processing times of the task should be implemented more realistically. In many situations the complexity depends on the actual data being processed. A more dynamic representation will give more accurate results.
Chapter 9

Results

This thesis described the analysis of running multiple IEEE 802.11a wireless LAN radios on a shared multi-processor system-on-chip. The platform that is analyzed is given by NXP and consists of four heterogeneous processor cores that are connected with each other via an on-chip interconnect. The architecture of this platform is rather static. Only how these processors are shared by multiple tasks, i.e. their scheduling policies, and what tasks are executed by which processor (the mapping) can be defined.

9.1 Summary

Instead of using existing generic analysis methods it has been chosen to define a method specifically for the WLAN radio. This was because the existing methods are either designed for different type of applications (e.g. streaming), cannot deal with data dependent executions or focus on a different part of the model. The used method is based on the Y-chart principle, where an application model is mapped on an architecture model and then analyzed.

9.1.1 Application model

The first step consisted of creating a model of a WLAN radio. This model consists of a task-graph and a set of constraints. The task-graph is designed with the TTL interface in mind. This implies that between tasks data is transmitted with tokens that are transmitted over directed channels. The constructed graph (see Figure 4.8) contains 13 tasks. Most of them are on the downstream side of a WLAN radio which requires far more processing that the upstream side, especially on the synchronization part.

Each task in this graph has a number of worst-case properties, namely the processing time ($P_x$), startup time ($S_x$), response time ($R_x$) and period ($T_x$). The channels that communicate the tokens also have a worst-case communication time ($C_y$) and period ($T_y$). Based on these properties a set of constraints are defined that limit the throughput and latencies such that the wireless LAN requirements are met. The most important of these latency requirements is the short interframe spacing of 16 $\mu$s.

9.1.2 Architecture model

The application is executed by the platform, so a model is defined for the architecture of that platform as well. This resulted in five parameters that can alter the performance of the platform. These are the frequency of the EVP and ARM processors ($F_{EVP}$ and $F_{ARM}$ respectively), the throughput of the network ($F_{Network}$) and the maximum processing times of the SWC ($P_{decoder}$ and $P_{coder}$).
9.1.3 Mapping and analysis

The application model is then combined with the architecture model by mapping the tasks of the task-graph on the processors of the platform. The mapping was pretty straightforward, because the processors of the platform have been designed for a particular application domain; hence the type of tasks they can process best is predetermined. This resulted in a set of equations that describe the properties of the task-graph with variables of the platform.

These properties are used in the constraints given by the application model, so the constraints depend on the platform performance as well. Therefore, the effects that the platform parameters have on the throughput and latencies are analyzed in detail.

This analysis shows that the minimum required values for the platform parameters that satisfy the throughput requirements do not satisfy the latency requirements. The performance of the EVP and ARM seem to have the most impact on that. The network throughput did not show any significant effect. By changing these values in the right direction a recommended configuration is defined with the following parameters.

\[
F_{EVP} = 400 \text{ MHz} \\
F_{ARM} = 150 \text{ MHz} \\
F_{\text{Network}} = 4 \text{ Gb/s} \\
P_{\text{decoder}} = 800 \text{ ns} \\
P_{\text{coder}} = 800 \text{ ns}
\]

It is shown that this configuration supports four wireless LAN radios simultaneously because the requirements are all satisfied. The throughputs of the tasks and network are well within the constraints (see Table 7.6). The latencies (see Table 7.7) are also below their maximum. The worst-case SIFS latency is almost 1,9 \( \mu \text{s} \) below the maximum of 16 \( \mu \text{s} \). The worst-case header-loop latency is a little more than 0,4 \( \mu \text{s} \) below is maximum. The gain control and acknowledgement frame latencies never showed any real problem. The load that this configuration puts on the platform is computed to be at most 84 \% for the EVP, 80 \% for The SWC and 40 \% for the ARM.

The results show some room for improvements: in particular the network latencies used in the model. For the network a lot of assumptions have been made, since the actual implementation of the network falls out of the scope of this thesis. This resulted in an optimistic view on the latencies of the network. The analysis showed that for most of the channels an additional latency can be introduced without problems.

In addition a configuration for supporting only one WLAN radio has been defined (see Table 7.12).

9.1.4 Simulation

The recommended configuration has also been simulated with the CASSE environment. This showed that the corner cases that define the worst-case values used in the analysis are hard to simulate. For instance, the highest recorded value for the SIFS latency stayed 4 \( \mu \text{s} \) below the computed maximum. Only the gain latencies approached their computed limits.

Part of these shortcomings can be blamed on the CASSE implementation. The CASSE version that has been used did not support complex scheduling policies or network models. As a result the implementation of the task-graph differed quite a bit and the network throughput in the simulation was equal to infinity.

However, the worst-case loads of the platform have been simulated successfully. In cases where all radios are detecting frames the load of the EVP was at its highest value of 84,4 \%. The lowest value of 68,1 \% was recorded when all radios were transmitting a frame. The load of the SWC was never higher than 80 \%, which happened when all radios were receiving a frame. The load of the ARM in that situation was at most 40 \%.
9.2. Assumptions

During the analysis several assumptions have been made on the available platform, which are listed below.

- The RF front-end is expected to execute the WLAN tasks fully in parallel. The hardware that should support this is quite complex. It might not be feasible to implement it like this, which will make the performance of the RF front-end go down.

- The EVP is assumed to execute a program four times faster than the OnDSP processor. This highly depends on the fact that the required instruction cycles can be optimized with a factor of two by making more use of parallelism. This might not be possible for all tasks, which will make the processing times go up and possible fail the latency constraints.

- In all cases the processors are expected to be fully available for the WLAN radios when needed and that they can not be interrupted by any higher priority tasks. This assumption is especially critical for the tasks mapped on the SWC. If the SWC has other higher priority tasks to process then the startup times of the Code and Decode tasks go up. This will also increase when there are tasks mapped on the SWC that have longer processing times than the WLAN tasks.

- It is assumed that the ARM processor has extensions that accelerate the computation of the CRC algorithm. If these are not available then the number of required instructions of the MACi task will be more than four times higher.

These assumptions made the analysis easier. If one or more of these assumptions are not feasible in a real implementation then the latencies might change dramatically.

However, the biggest assumption is made about the implementation of the network. During the entire analysis the network is not modeled in detail. It is assumed that transmitting a token is not influenced by other transmissions and that the transmission itself does not require extra latency. However, in a more realistic implementation the network needs time to process a frame. For instance in a packet-switched network a router needs time to find the destination of a packet. In case of a bus network the network might be occupied by another processor, so one has to wait before it can transmit.

The increase in communication latency has been analyzed a little, but not realistically enough. Once the characteristics of the network are known a more thorough analysis should be made with correct numbers.

9.2.1 Future perspectives

Besides the communication network there are other things that need to be investigated in more detail. First of all, the EVP spends quite some time on context switching. The amount of cycles
that each context switch takes is estimated, so a correct implementation should be investigated. If the processor is extended with better context switching capabilities then the minimum required operating frequency can go down significantly.

Furthermore, the current analysis was done for a platform that has one processor of each type. One can imagine that introducing a second processor for special purposes might reduce the latencies even more. For instance one extra EVP can be used for dealing with the short-period tasks while the other is focusing only on the long-period tasks. This way the worst-case execution times of the long-period tasks will go down, since they are preempted less often. Other possibilities are the usage of hardware shells that implement the TTL primitives. This can reduce the overhead of transmitting a token and further reduce the load of the system.

The system might also be capable of supporting the new IEEE 802.11n standard. This standard is basically a combination of four 802.11a channels bundled together. Since the platform supports four 11a radios it should in theory be capable of running a single 11n radio. However, an implementation of 802.11n also requires more processing on other places, so it should be investigated in more detail.

9.3 Conclusion

At the end it became clear that running multi-channel wireless LAN radios on a single system requires a lot of processing power, in particular the physical layer. However, with the right configuration, the MPSoC platform that NXP is aiming for should be able to process four IEEE 802.11a wireless LAN radios simultaneously while it is also used by other non-critical tasks.

This thesis gives an insight in the many factors that influence the performance of an application running on a multi-processor system. The results presented here can be the basis for a new platform that, for instance, has a better performance or gives more processing power to other tasks. Furthermore, the method presented in this thesis might also be used to analyze the performance of other communications standards as well.
Bibliography


and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band, The Institute of Electrical and Electronics Engineers, New York, September 1999


Appendix A

IEEE 802.11a wireless LAN

In 1997 working group 11 of the Institute of Electrical and Electronics Engineers LAN/MAN Standards Committee (IEEE 802) approved its first Wireless LAN standard [10]. The standard consists of two layers; namely the physical (PHY) layer and the Medium Access Control (MAC) layer. The physical layer deals with the transmission of frames using infrared (IR) signals or modulated radio signals at the Industrial Scientific Medical (ISM) frequency band at 2.4 GHz. For modulation either frequency-hopping spread spectrum (FHSS) or direct sequence spread spectrum (DSSS) can be used. The MAC layer coordinates the use of the communication channel using Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). With these techniques the standard supports data rates of 1 or 2 Mb/s.

Soon after the approval the working group began working on a successor. In 1999 two amendments to the original specification were approved; namely IEEE 802.11a-1999 [11] and IEEE 802.11b-1999 [12]. Both use the same MAC layer protocol, but specify a different physical layer protocol. The 'b' standard is an extension to the original DSSS modulation method. It uses the same 2.4 GHz ISM band and supports data rates of up to 11 Mb/s. The 'a' standard, on the other hand, uses a completely different method, namely orthogonal frequency-division multiplexing (OFDM) [14]. It uses the 5 GHz ISM band and supports data rates of up to 54 Mb/s, which is much higher than the 11b standard.

The two amendments were approved at the same time. Since the 11b standard is only an extension of the already existing protocol it was very easy for manufacturers to update their hardware. It also uses the same frequency band, so the required hardware components were easily available. The 11a standard, on the other hand, uses the 5 GHz frequency band. At that time this frequency band had not been used that often and therefore the hardware components were hard to get and more expensive. At the end, the 11b standard became more popular.

In order to deal with this problem the 802.11g amendment was approved in 2003 [13]. This standard was basically a merge of the two other amendments; hence it supports both DSSS and OFDM modulation on the 2.4 GHz band. For the higher data rates the OFDM modulation is used and DSSS for the lower data rates. Currently more and more products are supporting all three amendments.

The 802.11a standard does have some advantages over the others. First of all, the 5 GHz band is not used much by other products, in contrast to the 2.4 GHz band that is shared with microwave ovens, cordless phones and Bluetooth devices. One can imagine that in crowded areas where a lot of devices are close together a lot of interference can occur. Furthermore, 11a is also more resilient to multi-path interference, which happens when the original signal is reflected by objects like walls and trees. A reflected signal can take a longer path before it reaches the same destination and cause interference with the signal of a shorter path. The 11a standard has methods for dealing with this, which increase the performance of the channel.

Next to IEEE 802.11a, more standards are using OFDM. For instance, the IEEE 802.11n standard, which is also known as WiMAX, can basically be seen as several 11a channels bundled together and terrestrial digital video broadcasting (DVB-T) is a broadcasting protocol using
OFDM over 2048+ sub carriers. IEEE 802.11a can thus be viewed as a precedent for future wireless communication standards.

A.1 Physical layer

The principle of orthogonal frequency-division multiplexing is based on frequency-division multiplexing, where a communication channel is split into a number of narrowband sub-channels, each at a separate frequency. With OFDM these frequencies are chosen such that the carrier waves are orthogonal to each other, which means that cross-talk between the sub-channels is eliminated.

The 802.11a standard has a channel bandwidth of 20 MHz, which is split into 64 sub-channels of \( \frac{20 \text{ MHz}}{64} = 312.5 \) kHz wide. The period of each carrier wave is thus 3.2 \( \mu \)s. Only 48 of these sub-channels are used for data transport; 4 are used for pilot signals and 12 are at the borders of the spectrum and are not used at all.

Because of the orthogonality of the carrier signals a fourier transformation can be used to separate the individual channels. There are 64 sub-channels so a 64-point fast fourier transformation (FFT) is used for which 64 input samples are required. The full channel is 20 MHz wide, hence the sample period is 50 ns. In order to get all the inputs for one FFT calculation a period of \( 50 \times 64 = 3,200 \) ns = 3.2 \( \mu \)s must be sampled, which is exactly the same as the period for one sub-channel. The result of the FFT is a sample for each of the individual sub-channels; hence the sequence in the temporal domain is converted to the frequency domain. For the opposite direction (transmission) an inverse FFT is used to transform the sub-channels from the spectral domain back to the temporal domain.

Over each of the 48 data sub-channels a data bitstream is modulated using either binary phase-shift keying (BPSK), quadrature phase-shift keying (QPSK), 4-bit quadrature amplitude modulation (16-QAM) or 6-bit quadrature amplitude modulation (64-QAM), which can modulate 1, 2, 4 or 6 bits per symbol respectively. So each period at most \( 6 \times 48 = 288 \) bits can be transmitted.

Since low-rate modulation schemes are used, OFDM is less sensitive to inter-symbol interference which can be caused by multi-path interference. In order to reduce this effect even more, a cyclic guard interval (GI) of 0.8 \( \mu \)s is prepended to each period of 3.2 \( \mu \)s. This interval is a repetition of the last 0.8 \( \mu \)s of the sample set, which equals 16 digital samples. This is depicted in Figure A.1. The set of samples required for one FFT including this guard interval is called one OFDM symbol and is 4.0 \( \mu \)s long or 80 samples. During this period between 48 and 288 raw data bits can be transmitted, resulting in a minimum raw data rate 12 Mb/s and a maximum raw data rate of 72 Mb/s.

A.1.1 Error recovery

Unfortunately, radio signals are inherently sensitive to noise generated by external sources, such as natural background radiation and other devices operating in the same frequency spectrum. This noise alters the original signal and if it is changed too much then the receiver will interpret it wrongly, resulting in incorrect data bits.
A.1. PHYSICAL LAYER

In order to deal with such errors some redundant data is be added to the original bit stream that helps in recovering the altered bits. The stream is encoded using a so-called forward error correction (FEC) code. Several of these codes exist, one of these is the convolutional code that transforms each block of $m$ input bits into a block of $n$ output bits using information of $k$ previous bits. The rate of such a coder is defined by $R = \frac{m}{n}$. The 802.11a standard uses a coder with $m = 1$ input bits, $n = 2$ output bits and depth $k = 7$. The rate of this coder is $R = \frac{1}{2}$. The structure is depicted in Figure A.2.

![Convolutional coder (k=7)](image)

Because of the rather high coding rate of $\frac{1}{2}$, the raw data rate is reduced in half. Often the coding rate can be relaxed when the channel does not have that much noise. The standard only defines one coder with one coding rate. In order to support other rates a technique called puncturing is applied. Basically this just removes some bits from the output stream of the coder such that the number of redundant bits per block goes down. This way the standard defines two addition coding rates, namely $\frac{3}{4}$ and $\frac{2}{3}$. The former removes 6 bits from a block of 18 coded bits, which results in 12 coded bits for the original 9 input bits ($\frac{9}{12} = \frac{3}{4}$). The latter removes 3 out of 12 bits, resulting in 9 coded bits for 6 input bits ($\frac{6}{9} = \frac{2}{3}$).

When combining these three coding rates with the four earlier mentioned modulation schemes, a large amount of different data rates can be defined. The list of supported data rates is listed in Table A.1. With lower bitrates less bits are encoded and modulated per symbol, making it easier to recover errors.

One of the properties of a convolutional code is that it is good in recovering single errors in a stream, but it performs rather badly with burst errors where several consecutive bits are incorrect. In order to deal with such errors the output of the coder is rearranged by a block interleaver using a block size equal to the number of bits per OFDM symbol ($N_{CBPS}$). It mainly distributes the bits such that adjacent coded bits are mapped onto non-adjacent sub-channels.

All this only defines how bits are prepared for transmission. Luckily, the reverse process is quite straightforward. First the bits from the demodulator are de-interleaved with the same block size. Then a number of dummy bits are inserted to undo the puncturing. Finally the bits are decoded by a decoder based on convolutional codes. Several of these exist, but the standard recommends the Viterbi decoder [15].

The recovery of errors in a stream becomes more accurate when information is available on how much a demodulated bit deviated from its constellation point. This is included as a weight
Table A.1: Data rates

<table>
<thead>
<tr>
<th>Data rate (Mbits/s)</th>
<th>Modulation</th>
<th>Bits per sub-carrier ($N_{BPS}$)</th>
<th>Coded bits per symbol ($N_{CBPS}$)</th>
<th>Coding rate ($R$)</th>
<th>Data bits per symbol ($N_{DBPS}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>BPSK</td>
<td>1</td>
<td>48</td>
<td>1/2</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>BPSK</td>
<td>1</td>
<td>48</td>
<td>3/4</td>
<td>36</td>
</tr>
<tr>
<td>12</td>
<td>QPSK</td>
<td>2</td>
<td>96</td>
<td>1/2</td>
<td>48</td>
</tr>
<tr>
<td>18</td>
<td>QPSK</td>
<td>2</td>
<td>96</td>
<td>3/4</td>
<td>72</td>
</tr>
<tr>
<td>24</td>
<td>16-QAM</td>
<td>4</td>
<td>192</td>
<td>1/2</td>
<td>96</td>
</tr>
<tr>
<td>36</td>
<td>16-QAM</td>
<td>4</td>
<td>192</td>
<td>3/4</td>
<td>144</td>
</tr>
<tr>
<td>48</td>
<td>64-QAM</td>
<td>6</td>
<td>288</td>
<td>2/3</td>
<td>192</td>
</tr>
<tr>
<td>54</td>
<td>64-QAM</td>
<td>6</td>
<td>288</td>
<td>3/4</td>
<td>216</td>
</tr>
</tbody>
</table>

with each bits coming from the demodulation process. One such "soft bit" is then represented by multiple real bits, typically 3 bits.

An overview of the modulation and coding process is depicted in Figure A.3.

A.1.2 Frame structure

As with most communication protocols the physical layer encapsulates the data given by the MAC layer in a frame. This is used to distinguish the start and end of a data block and to supply additional information like the data rate of the OFDM symbols. The structure of the physical layer frame is depicted in Figure A.4. As you can see a header and a trailer are put before and after the payload (PSDU). The header consists of several fields. The most important are the RATE field, which specifies the data rate of the data symbols, and the LENGTH field that specifies the length in bytes of the payload. The length is given by a 12-bit number; hence the payload can be at most 4095 bytes. The 16 bits of the SERVICE field are mainly reserved for future purposes and should be set to 0.

To be certain that the header is received with the lowest possibility of errors it is transmitted using the lowest data rate, i.e. BPSK modulation and a coding rate of $\frac{1}{2}$. At this data rate one OFDM symbol can only contain 24 bits. The header is larger than this and therefore the SERVICE field is combined with the data OFDM symbols.

After the payload six tail bits are appended to make sure several components (such as an optional scrambler) return to their initial state. These are all supposed to be zero. In total the data to be transmitted after the header OFDM symbol is the size of the payload plus 22 bits.
Because each OFDM symbol contains a certain number of bits ($N_{DBPS}$) it is necessary to pad the data block to a multiple of this in order to full the last symbol completely.

### A.1.3 Frame detection

The WLAN standard is inherently unsynchronized. A frame can arrive at any moment and a receiver does not know when this happens. In order to make detection of a frame possible a preamble is appended at the beginning. This preamble consists of a series of repeated signals, designed to generate high correlation with each other.

The structure of this preamble is depicted in Figure A.5. As you can see it is divided into two equally sized parts. The total preamble is 16 $\mu$s long. The first part consists of 10 identical short training symbols (STS) of 0,8 $\mu$s. The second part consists of two identical long training symbols (LTS) of 3,2 $\mu$s preceded by one guard interval of 1,6 $\mu$s, and will be explained in the next section.

There are several methods of detecting the short training symbols. However, they are chosen such that they generate high correlation with each other, so it is logical to use some correlation algorithm. A recommended algorithm correlates a window of 16 samples with a reference symbol. This signal is then normalized using the average incoming signal strength and compared to a threshold. If it crosses this threshold it can be assumed that a short training symbol has been received. It is summarized in Equation A.1, where $c$ indicates the correlation of the input signal $x$ with the reference signal $r$. The average power of the last symbol is denoted by $p$. The value of $m$ can be compared with a predefined threshold.
The highest correlation occurs at the start of a STS. So the threshold will only be exceeded at those moments. However, the receiver does not know where the start of a STS is, so it has to compute the correlation for each newly arrived sample. This can be simplified by shifting the reference signal instead of the input window. The reference signal is static so a matrix can be constructed in advance that (when multiplied with the incoming signal) checks at all the locations. The equation of $c_n$ in A.1 can then be rewritten using vectors and matrices as depicted in Equation A.2. The resulting vector can then be checked for the highest result and at that index is the start of a STS.

$$
\begin{align*}
  c_n &= \sum_{k=0}^{L-1} x_{n+k} r_k^* \\
  p_n &= \sum_{k=0}^{L-1} x_{n+k} x_{n+k} = \sum_{k=0}^{L-1} |x_{n+k}|^2 \\
  m_n &= \frac{|c_n|^2}{p_n^2}
\end{align*}
$$

(A.1)

So after the detection has occurred two things have to be done before it can proceed. A signal can get weaker when it has to travel longer distances, so first of all the incoming analog signal has to be normalized such that it the entire range of the AD-converter is used. This can be done by inspecting the current signal strength and adjusting the gain of the RF front end accordingly. After it is adjusted for the first time the signal strength of the next set of samples should be inspected to fine-tune the gain. For the most accurate gain level it is advised to do a 3-step adjustment.

After the gain is set the receiver still does not know for sure where in the preamble it is. Due to noise some of the first training symbols might be missed, and it is even possible that the detection generated a false positive because of this noise. So to find out where in the frame the detection ended, each STS that arrives has to be verified. For this a new correlation is computed for each of the 16 new samples that come in. If the first correlation fails one can safely assume that the detection was a false positive. If it fails after a number of successes, then one can assume that the guard interval of the long training symbols has been reached and fine-tuning in the frequency domain can start.

### A.1.4 Frequency error correction

Sampling an analog signal is controlled by an oscillator working at the frequency of channel. These oscillators are not perfect. They have a certain constant and dynamic frequency offset that causes a difference between the receiver and the transmitter. OFDM modulation depends heavily on the orthogonality of the carrier signals and that can be disturbed by this oscillator difference. The offsets can be corrected using certain training symbols, which specify a certain value at each sub-channel. So to deal with this problem the preamble of 11a includes two identical Long Training Symbols (LTS) of 3.2 µs, e.g. one block of 52 modulated sub-channels. The sub-channels each contain a predefined BPSK modulated value. The differences between these predefined values and the values of the various carriers of the received LTS can be used to find the correct frequency offsets.
Like OFDM data symbols the long training symbols are also sensitive for multi-path fading, therefore a single guard interval of $1.6 \, \mu s$ is put in front of them to make sure the short and long training symbols do not interfere with each other. There are two long training symbols in the preamble to improve the accuracy of the frequency error estimation. The two symbols can either be averaged to remove any noise, or they can be processed individually for an incremental adjustment.

The long training symbols can be used to correct the initial constant offset. However, oscillators tend to deviate in time, the dynamic offset. This dynamic behavior is often quite slow and can be tracked easily by inspecting several so-called pilot signals. For 11a WLAN 4 of the 52 sub-channels are reserved for this. So during processing of the data OFDM symbols the pilot channels have to be inspected and the offset corrected accordingly.

Several other effects can still change the characteristics of the channel. For instance, a signal will not arrive at the original transmitted strength. This can deviate for each individual frequency, resulting in a total channel response (like depicted in Figure A.6). Fortunately, this response is changing slowly and is assumed constant for the duration of a frame. It can also be computed using the long training symbols in the preamble.

A.1.5 Summary

The entire physical layer is responsible for transmitting a given block of bits over a communication channel. The procedure is as follows. First the block of bits is appended with a header and a footer, containing information about the length of the block and the data rate at which it should be transmitted. This block is split into smaller parts that are transmitted sequentially.

Per part, the bits are first passed through a coder and an interleaver to improve the error resistance. The coder supports 3 different rates, depending on the required data rate. The output of the coder is distributed over a set of 48 individual channels and modulated using one of four different schemes. The resulting signal is appended with four pilot signals and passed through an inverse fast fourier transformer to generate the time-domain signal of $3.2 \, \mu s$. This signal is extended with a cyclic guard interval of $0.8 \, \mu s$, completing the so called OFDM symbol.

The first of these OFDM symbols, containing the header bits, is coded and modulated using the lowest data rate possible, to make sure it arrives correctly. The other symbols use the given data rate. Each OFDM symbol can contain between 24 and 216 bits of uncoded data, resulting
in an effective data rate between 6 and 54 Mbits/s.

To make detection of these frames possible the sequence of OFDM symbols is preceded by a preamble of 16 μs. It consists of ten short training symbols of 0.8 μs length each and two long training symbols of 3.2 μs, separated by a guard interval of 1.6 μs.

**A.2 MAC layer**

The previous section explains how data is transmitted from one station to another. However, in a wireless network multiple stations share the same channel and can interfere with each other if not regulated correctly. For this task the medium access control (MAC) layer is designed. It provides functions to manage the communication between stations over a shared channel.

It can use any IEEE 802.11 physical layer to transport a frame from one station to another. This can be the 802.11b physical layer, or the previously discussed 802.11a physical layer.

**A.2.1 Frame format**

As with the physical layer the MAC layer transports data units in frames. The global frame format is depicted in Figure A.7. It has a lot of fields. Not all of them are used by all frame types. Only the Frame Control, Duration/ID, Address 1 and FCS field are present in all types, thus the minimal size of a frame is 14 bytes (112 bits).

The first seven fields define the header of the frame. It is followed by one big field containing the actual payload. The last 4 bytes are the Frame Check Sequence and represent the checksum of a 32-bit cyclic redundancy check (CRC-32) on the frames content. The Frame Control field contains information about the rest of the frame, for example what type the frame is. There are a lot of different frame types, however only a few are relevant, which will be discussed in the following sections. The address fields contain the unique MAC addresses of the source and/or destination station, which are similar as the ethernet MAC addresses. The body of a MAC layer frame can contain a maximum number of bytes, namely 2312 bytes. Larger chunks of data have to be fragmented over multiple frames.

![General frame format](image.png)

**Figure A.7: General frame format**

**A.2.2 Access coordination**

Access to the medium is regulated by the MAC layer. The MAC layer supports two access methods, namely a distributed coordination function (DCF) and a point coordination function (PCF). The distributed coordination function is mandatory and makes sure that stations can communicate with each other without the assistance of a central arbitrator. In this case the carrier sense multiple access with collision avoidance (CSMA/CA) protocol is used as DCF.

The optional point coordination function is defined to allow time-bounded delivery of data frames. Here a point coordinator grants exclusive access to stations that wish to transmit. Since the PCF is optional and barely supported by current hardware, it is not treated any further in this document. More information about the PCF can be found in the standard documentation [10].
A.2.3 Carrier sense multiple access with collision avoidance

The medium used by the WLAN standard (i.e. the air) is a shared medium that can cause collisions of frames. Since a station cannot have its receiver and transmitter turned on at the same time, a station is unable to sense frame collisions. The CSMA/CA protocol is designed to minimize the possibility of a collision.

Before a station can attempt to send a frame it will first sense the medium. If it is busy, e.g. another station is currently transmitting, then the station will wait until the medium is free again. When the other station has stopped transmitting the station will wait a period known as the DCF interframe space (DIFS), which is 34 $\mu$s long. After this period all the stations that want to send enter a so called contention window in which each station backs off for a random time. At the end of this back off period the medium is sense one more time. If it is still free it will immediately start transmitting. The station with the shortest back off time will start transmitting first. The other stations will have to wait another turn. Using a random back off time minimizes the chance that multiple stations start transmitting at the same time, reducing the number of collisions.

Once a data frame is transmitted the sending station needs to know if it has arrived correctly at the destination station. A positive acknowledgement mechanism is used to accomplish this. For every data frame that is correctly received, i.e. the checksum is correct, an acknowledgement (ACK) frame is returned to the sender. If the sender does not receive a correct acknowledgement frame after a certain time it assumes the frame has not reached the destination and it retransmits the frame.

Acknowledgement frames should have precedence over data frames, so a receiving station has to send its acknowledgement frame before any other station starts transmitting. Therefore a smaller gap of 16 $\mu$s which is called the short interframe space (SIFS), is used between these frames instead of the DIFS. This space is also used when multiple frames are transmitted in a burst. A station can start transmitting the next frame of a burst SIFS time after the acknowledgement frame is send.

Since the SIFS is shorter than the DIFS other stations will not contend for the medium, because it is still active when the DIFS period should end. Once a station is finished transmitting and the last (or only) acknowledgement frame is send the DIFS timer will end, causing the other stations to start contending for the medium.

The complete mechanism is depicted in Figure A.8. Next to the DIFS and SIFS there are also two additional spaces, namely PCF interframe space (PIFS) and extended interframe space (EIFS). PIFS will be ignored, since PCF is not treated in this document and the EIFS is only relevant when a frame is not correctly received, which is out of the scope of this document.

![Figure A.8: CSMA/CA example](image)

**Carrier sensing**

There are two ways of sensing the medium: physically and virtually. Physical sensing is provided by the physical layer that monitors the actual radio signals. If the energy level is above a certain threshold it is assumed that the medium is being used by another station.

With virtual sensing a station uses the Duration/ID field in the frames header to keep track of the current and announced usage of the medium. Every time a station receives a frame it can use this field to determine for how long the medium will be occupied by the sending station including
the returning acknowledgement. This time is stored in the network allocation vector (NAV), which is basically a counter. For the duration of the NAV the medium is reserved for the sending station and transmission can only start once the NAV has reached zero.

However, wireless networks suffer from a problem known as the "hidden transmitter problem", which occurs when not all stations can receive the frames from the other station in the network. The easiest way to explain this is with Figure A.9.

![Figure A.9: Hidden transmitter problem](image)

The figure shows four stations A, B, C and D. Stations B and D are in the range of station A, so they can receive frames from station A. Station C is out of range but still in range of station B. If station A wants to send a frame to station B, it cannot sense whether station C is transmitting and thus does not know if the medium is busy at station B. If both stations A and C start transmitting at the same time, the signals will collide at station B. Resulting in two useless frames and both stations will have to retransmit their data.

To overcome this problem, two additional frames are defined that make sure communicating stations and their surroundings know from each other that the medium will be busy for a certain period. The communication flow is depicted in Figure A.10. The sending station (A) starts by sending out a request to send (RTS) frame, which includes a duration field representing the complete time it takes to transmit the data and acknowledgment frames. All the surrounding stations of A (i.e. B and D) receive this and adjust their NAV accordingly. Station B, which is the destination, sends back a clear to send (CTS) frame, also including a duration field. Station C, which was unaware of the transmissions until now, receives this CTS frame and also adjusts its NAV accordingly. Now both stations A and B and their surrounding stations know that the medium will be occupied by station A and abstain from sending themselves. Once the NAVs expire the stations know that the medium is free again and will wait DIFS time before contending for the medium again.

### A.2.4 Fragmentation

Wireless transmissions are inherently more sensitive to errors than wired systems. On average in a wired system $1 \times 10^{-2}$ bits is wrong. In wireless systems it is between $1 \times 10^{-5}$ and $1 \times 10^{-6}$ bits. Since this number is so low it is possible that large packets approach the number of bits where the probability of an error is 100%.

In order to deal with this, frames can be fragmented into smaller pieces. Each piece is then individually send and acknowledged, making retransmission less expensive.
Figure A.10: RTS/CTS communication