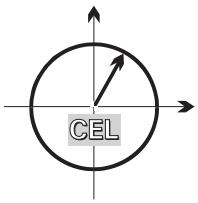
Forschungsberichte aus dem
Institut für Nachrichtentechnik des
Karlsruher Instituts für Technologie



Stefan Nagel

Portable Waveform Development for Software Defined Radios

Band 26

Druck: Frick Digitaldruck
Brühlstraße 6
86381 Krumbach

ISSN: 1433-3821

Forschungsberichte aus dem Institut für Nachrichtentechnik des Karlsruher Instituts für Technologie
Herausgeber: Prof. Dr. rer. nat. Friedrich Jondral

Band 1	Marcel Kohl	**Simulationsmodelle für die Bewertung von Satellitenübertragungsstrecken im 20/30 GHz Bereich**
Band 2	Christoph Delfs	**Zeit-Frequenz-Signalanalyse: Lineare und quadratische Verfahren sowie vergleichende Untersuchungen zur Klassifikation von Klaviertönen**
Band 3	Gunnar Wetzker	**Maximum-Likelihood Akquisition von Direct Sequence Spread-Spectrum Signalen**
Band 4	Anne Wiesler	**Parametergesteuertes Software Radio für Mobilfunksysteme**
Band 5	Karl Lütjen	**Systeme und Verfahren für strukturelle Musteranalysen mit Produktionsnetzen**
Band 6	Ralf Machauer	**Multicode-Detektion im UMTS**
Band 7	Gunther M. A. Sessler	**Schnell konvergierender Polynomial Expansion Multiuser Detektor mit niedriger Komplexität**
Band 8	Henrik Schober	**Breitbandige OFDM Funkübertragung bei hohen Teilnehmergeschwindigkeiten**
Band 9	Arnd-Ragnar Rhiemeier	**Modulares Software Defined Radio**
Band 10	Mustafa Mengüç Öner	**Air Interface Identification for Software Radio Systems**

Forschungsberichte aus dem Institut für Nachrichtentechnik des Karlsruher Instituts für Technologie
Herausgeber: Prof. Dr. rer. nat. Friedrich Jondral

Band 11	Fatih Çapar **Dynamische Spektrumverwaltung und elektronische Echtzeitvermarktung von Funkspektren in Hotspotnetzen**
Band 12	Ihan Martoyo **Frequency Domain Equalization in CDMA Detection**
Band 13	Timo Weiß **OFDM-basiertes Spectrum Pooling**
Band 14	Wojciech Kuropatwiński-Kaiser **MIMO-Demonstrator basierend auf GSM-Komponenten**
Band 15	Piotr Rykaczewski **Quadraturempfänger für Software Defined Radios: Kompensation von Gleichlauffehlern**
Band 16	Michael Eisenacher **Optimierung von Ultra-Wideband-Signalen (UWB)**
Band 17	Clemens Klöck **Auction-based Medium Access Control**
Band 18	Martin Henkel **Architektur eines DRM-Empfängers und Basisbandalgorithmen zur Frequenzakquisition und Kanalschätzung**
Band 19	Stefan Edinger **Mehrträgerverfahren mit dynamisch-adaptiver Modulation zur unterbrechungsfreien Datenübertragung in Störfällen**
Band 20	Volker Blaschke **Multiband Cognitive Radio-Systeme**

Forschungsberichte aus dem Institut für Nachrichtentechnik des Karlsruher Instituts für Technologie
Herausgeber: Prof. Dr. rer. nat. Friedrich Jondral

Band 21	Ulrich Berthold	**Dynamic Spectrum Access using OFDM-based Overlay Systems**
Band 22	Sinja Brandes	**Suppression of Mutual Interference in OFDM-based Overlay Systems**
Band 23	Christian Körner	**Cognitive Radio – Kanalsegmentierung und Schätzung von Periodizitäten**
Band 24	Tobias Renk	**Cooperative Communications: Network Design and Incremental Relaying**
Band 25	Dennis Burgkhardt	**Dynamische Reallokation von spektralen Ressourcen in einem hierarchischen Auktionssystem**
Band 26	Stefan Nagel	**Portable Waveform Development for Software Defined Radios**

Vorwort des Herausgebers

Software Defined Radios (SDRs) werden inzwischen seit zwei Jahrzehnten intensiv untersucht. Dabei stand zunächst die Frage im Vordergrund, wie ein Funkgerät entwickelt werden kann, das mehrere verschiedene Standards unterstützt. Danach wurden in weitergehenden Untersuchungen Verfahren zum Software-Update, auch durch Funkübertragung, implementiert. Dazu gehörten auch Methoden zum Nachweis der Korrektheit der übertragenen Software. Breitflächig durchgesetzt haben sich die SDR Technologien bisher im Wesentlichen in Basisstationsgeräten. Mobile Endgeräte existieren vor allen Dingen als Multiband- (GSM 900, 1800, 1900) und Multistandard- (GSM, UMTS) Transceiver.

Eine weitere wesentliche Eigenschaft, durch die sich SDRs auszeichnen können, wurde bisher zwar in der Literatur breit diskutiert, experimentell jedoch, wohl wegen der dazu notwendigen Hardware, nicht angegangen. Es handelt sich um die **Portabilität** von Wellenformen[1] von einer Hardwareplattform auf eine andere. Durch das Aufkommen finanzierbarer Plattformen wie der Universal Software Radio Peripheral (USRP) von Ettus Research oder der Small Form Factor Software Defined Radio Development Platform (SFF SDR DP) von Lyrtech wird eine experimentelle Untersuchung der Portabilität von Wellenformen durchführbar. Dabei treten Fragen nach einer Entwicklungsstrategie für die SDR Software unter Berücksichtigung der Portabilität sowie nach den Kosten, die eine Portierung verursacht, in den Vordergrund. Um hierfür Antworten finden zu können, müssen, neben einem profunden Wissen über die beteiligten SDR Plattformen, Kenntnisse über Wellenformen bzw. Standards vorliegen. Dazu kommt die Beherrschung der notwendigen Hilfsmittel wie Programmiersprachen von MatLab über Python und C bis hin zu VHDL, Verilog und Assembler sowie der Umgang mit Messmitteln wie Signalgeneratoren und -analysatoren, die

[1] Die Begriffe Standard und Wellenform werden oft als synonym angesehen. Allerdings umfassen Wellenformen, im Gegensatz zu Standards, häufig neben der Definition einer Luftschnittstelle auch Eigenschaften des Empfängers.

für den Nachweis erfolgreicher Arbeit, der die Interoperabilität zwischen auf verschiedenen Hardwareplattformen basierenden Funkgeräten umfasst, gebraucht werden.

Auf der Basis des am KIT Institut für Nachrichtentechnik (Communications Engineering Lab, CEL) eingerichteten Funklabors hat Stefan Nagel die Portabilität verschiedener Wellenformen zwischen den beiden oben genannten Hardwareplattformen USRP und SFF SDR DP untersucht. Mit der Dissertation *Portable Waveform Development for Software Defined Radios* legt er die wesentlichen Ergebnisse seiner Arbeit vor.

Die Arbeit trägt insofern nachhaltig zum Fortschritt von Wissenschaft und Technik bei, als hier erstmalig, basierend auf der Model Driven Architecture (MDA) der Object Management Group (OMG), die Software für die Signalverarbeitung verschiedener Funkübertragungsstandards entwickelt, implementiert und von einer Plattform auf eine andere portiert wurde.

Karlsruhe, im Mai 2011
Friedrich Jondral

Portable Waveform Development for Software Defined Radios

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie

genehmigte

DISSERTATION

von

Dipl.-Ing. Stefan Werner Nagel

geb. in

Sinsheim

Tag der mündlichen Prüfung:	26. Mai 2011
Hauptreferent:	Prof. Dr. rer. nat. Friedrich K. Jondral
Korreferent:	Prof. Jeffrey H. Reed Ph.D.
Korreferent:	Prof. Dr.-Ing. Robert Weigel

Zusammenfassung

Der Grundgedanke bei Software Defined Radios besteht darin, Funkstandards in Software auf rekonfigurierbaren Prozessoren zu verarbeiten. Dieses Konzept für neue Funkgeräte existiert bereits seit mehreren Jahrzehnten. Allerdings ist es erst durch die rasante Entwicklung auf dem Markt der Prozessoren der letzten Jahre möglich, SDR-Plattformen kommerziell zu erwerben und eigene Wellenformen zu implementieren. Dabei ergibt sich das Problem, dass Wellenformen, die auf der eigenen Plattform lauffähig sind, zu anderen Plattform nicht zwangsläufig kompatibel sind. Dieses Problem tritt auch bei einer möglichen Plattformerneuerung auf. Die Wellenformen, die für eine alte Plattform entwickelt wurden, müssen auf eine neue Plattform portiert werden. Dabei stellt sich zwangsläufig die Frage: „Wie können Wellenformen entwickelt werden, die auf beliebigen Plattformen lauffähig sind?"

Ein Erfolg versprechender Ansatz ist die Entwicklung von Wellenformen nach der Model Driven Architecture. Diese beschreibt einen Entwicklungsprozess, der sich von sehr generischen, plattformunabhängigen Modellierungen der Funktionalität bis zum ausführbaren Binärcode erstreckt. In dieser Arbeit wird vorgestellt, wie dieser Prozess auf die portable Entwicklung von Wellenformen angepasst werden kann. Die automatisierte Erzeugung von Quellcode spielt hierbei eine wichtige Rolle. Daher werden Laufzeit- und Speichermessungen vorgestellt, die generierten Code mit nicht optimiertem und optimiertem Code vergleichen und damit einen Einblick in die Effizienz von automatisch generiertem Code erlauben.

Das Ettus USRP und das Lyrtech Small Form Factor SDR gehören zu den kommerziell erfolgreichsten SDR-Plattformen. Daher werden in dieser Arbeit ausführlich ihr Aufbau sowie ihre Fähigkeiten und Limitierungen beschrieben. Weiterhin wird aufgezeit, wie diese Plattformen in den bereits vorgestellten Entwicklungsprozess integriert werden können. Dazu gehören sowohl die Umsetzung der programmierbaren Schnittstellen

und der Bussysteme in die Systemmodellierung als auch die Integration der betreffenden Prozessoren in die Codegenerierung.

Um zu demonstrieren, dass der vorgestellte Entwicklungsprozess auch praktisch anwendbar ist, wurde die Wellenform TETRA für eine Plattform entwickelt und auf eine zweite Plattform portiert. Die Umsetzung der dabei zu realisierenden Modelle werden in dieser Arbeit genauso vorgestellt wie die Verarbeitungsdauer der Algorithmen auf den betreffenden Prozessoren. Um zu gewährleisten, dass die Wellenform standardkonform umgesetzt wurde, kamen Messgeräte zum Einsatz, die sowohl Sende- als auch Empfangspfad gegen die TETRA-Spezifikation getestet haben. Neben TETRA wurden zwei weitere Wellenformen entwickelt und portiert. Die Ergebnisse und Herausforderungen all dieser Entwicklungsvorgänge werden in dieser Arbeit präsentiert.

Abstract

The basic idea of Software Defined Radio is the implementation of radio communication standards with software on reconfigurable processors. This concept for new radio devices was already proposed several decades ago. However, through the rapid development of processors in recent years, it is possible to acquire commercial SDR platforms and build own waveforms. Unfortunately, waveforms that are developed for one platform are not necessarily compatible to other platforms. This problem also occurs with a possible hardware upgrade. The waveforms that were developed for an old platform must be ported to a new platform. Therefore, this work focuses on the question: "How can we build waveforms that can be moved from one platform to another?"

A promising approach is the development of waveforms based on the Model Driven Architecture. It describes a development process that extends from a very generic, platform-independent functionality to the executable binary code. This work presents the adaption of this process to the development of portable waveforms. In this process, the automated generation of source code plays a decisive role. Therefore, measurements of processing time and memory consumption are presented to compare the generated code with non-optimized and optimized code and allow insights into the efficiency of automatically generated code.

The USRP from Ettus and the Small Form Factor SDR from Lyrtech are among the most used commercially SDR platforms. Therefore, their structure as well as their capabilities and limitations are presented in this work. It is furthermore shown, how these platforms can be integrated into the development flow. This includes the implementation of programmable interfaces and bus systems as well as the integration of the processors in the code generation process.

To demonstrate that the used development flow is also applicable in practice, a proof of concept is given with the development and port of a TETRA waveform from one platform to another. Therefore, the

realizations of the waveform for both platforms are presented as well as the processing times for the algorithms on the different processors. To demonstrate the standard compliance, the waveform was tested with measurement equipment against the TETRA air specification. In addition to TETRA, two other waveforms were developed and ported. This work presents the results and challenges of all these waveform developments and ports.

Contents

1 Introduction

1.1 Motivation

1.1.1 Developments in wireless communications

Two fields in electrical engineering formed the evolution of Software Defined Radios: the history of wireless communication principles and the development of digital processing technologies.

The wireless transmission of information started at the end of the 19th century. In 1886, Heinrich Hertz was the first who generated and detected electromagnetic radiation, with which he verified Maxwell's theory of the existence of electromagnetic waves. Nicola Tesla gave a public demonstration of wireless radio communications in 1893 and ten years later Marconi succeeded in the first wireless transmission over the Atlantic. Even if these milestones in the history of wireless communications took place more than hundred years ago, the principles of the transmission of amplitude modulated (AM) analog signals are still existent today: Most broadcast radio receivers provide an AM tuner for the radio frequency (RF) band and the signal for analog broadcast television is also amplitude modulated. In 1933, Edwin Armstrong patented a method to integrate the information in the frequency component of an electromagnetic wave instead in the amplitude. Hence, radios with Frequency Modulation (FM) occurred and are still used in the Very High Frequency (VHF) band for audio broadcast.

The fact that old communication schemes are still used today is also true for digital communications. The Global System for Mobile Communications (GSM) was created 1981 and thirty years later it has a market share of 80.4 % for cellular mobile networks [1]. With the standards that were released in the last two decades like the 802.11 family, Universal Mobile Telecommunications System (UMTS) or Long Term Evolution (LTE), a

normal user faces plenty of complete different wireless communication schemes. It is a fact that more and more communication schemes emerge while the older ones still must be maintained. From a normal user perspective this is not problematic due to the fact that the communication devices are cheap and can be replaced immediately.

From a military perspective this is a huge challenge. The number of radio communication standards in this area are in the range of several hundreds and a legacy radio device supports only a few of them. These devices should additionally operate over decades. New devices should not only be interoperable with the older devices they should also be interoperable with communication schemes from other countries for international operations. The present situation with the need for a radio device can be summarized as follows: There are plenty of radio communication standards and the optimum would be a device that supports present standards and also communication schemes for the future.

1.1.2 History of digital processing technology

The development of programmable devices for data processing started in 1971 with the Intel 4004. This was the first single chip microprocessor on the market with an integrated Central Processing Unit (CPU), a memory unit, as well as input and output control ports [2]. Eleven years later the first commercial microprocessor with a special architecture for multiplying and accumulating was released [3]. With these Digital Signal Processors (DSPs) a new branch of processors intended for digital processing purposes evolved. At the same time, the company Xilinx was founded and developed the first Field Programmable Gate Array (FPGA) for the commercial market [4]. These inventions and the development of the Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC) technology formed the basis of processing radio signals with software.

1.1.3 Evolution of Software Defined Radio

With the development in processing technology in the last thirty years, the architecture of radio devices changed from application specific in-

tegrated circuits to processors and reconfigurable logic. With these programmable devices, the radio standards could be realized by software and could be changed according to the user's need. Although many people contributed in this idea like Ulrich Rohde in 1984 [5], Joe Mitola coined the term "Software Defined Radio" in 1991 [6].

At the same time, the first Software Defined Radio (SDR) project was initiated by the U.S. military to produce an SDR that could be tuned to frequencies between 2 MHz and 2 GHz and provide furthermore interoperability between several different standards of the armed forces. Today, the principle of SDR is present in many fields of wireless communication. In the military environment, SDR projects are launched around the world. In the U.S. military, the project Joint Tactical Radio System (JTRS) should produce devices that are interoperable to many existing military and civilian communication standards and should successively replace existing legacy radios. The German equivalent: "Streitkräftegemeinsame Verbundfähige Funkgeräteausstattung (SVFuA)" is a software defined joint radio system that works in a frequency range between 3 MHz and 3 GHz and should also replace existing legacy devices in the German army.

The advantage of an SDR is the reconfigurability and the interoperability to old radio communication standards. Legacy radios can still be used and the SDR has even the possibility to work as a gateway between existent and new wireless standards [7]. These advantages were also recognized by various European agencies of public and governmental security, which face the problem to replace their old analog communication structure with a new digital system. The European project "WIreless INTeroperability for SECurity (WINTSEC)" proposed SDR as the key enabler for this step [8]. But not only the replacement of legacy to new devices can be accomplished. An SDR can also enable the wireless communication among rescue teams, firefighters and police. It can furthermore establish a connection with rescue teams across a country's borders.

With the company Vanu Inc., SDRs are also existent on the commercial market for mobile communication. Vanu Inc. was the first company that developed base stations with General Purpose Processors (GPPs). The wireless standards are written in high level languages and can be configured and maintained online [9].

1.1.4 Problem of portability

The difference between a legacy radio device and a Software Defined Radio is in principle the distribution of the SDR in a waveform part and in a platform part. It is therefore named: "The waveform/platform paradigm". The platform consists of the hardware with various processing elements, memory and user interfaces. However, it is not only the hardware that comprises a platform, it is also the software connected to it like the operating system, the firmware and Application Programmable Interfaces (APIs). The waveform is an application that is supported by the SDR platform and configures it in accordance to the dedicated radio communication standard. Therefore, the waveform enables the platform to be part of the related radio communication system. The waveform developer is able to build multiple waveforms for his platform.

In the past years several new SDR platforms were released and each waveform developer build applications for his own system. The question evolved: "How can be assured that waveforms can be moved from one platform to another?" This portability problem is not only valid for the exchange of waveforms. It is also existent when SDR platforms are modernized and upgraded with new processors. How can be assured that waveforms running on the old platform can be ported to the new platform with a minimum of effort?

This work focuses on these questions. It proposes a development flow and shows that waveforms can be ported. A proof of concept is given with an exemplary waveform and two platforms. It additionally investigates the overhead of automatically generated code, which is a key term for waveform portability.

1.2 Outline of Work

Chapter 2 gives an overview of existing waveform development flows and compares them with respect to portability issues. The Model-Based Design is introduced for baseband processing and measurements are presented about the overhead of automatically generated codes.

In this work about portability of SDR waveforms also an overview of SDR platforms are given in chapter 3, concerning a general and a

detailed description of two special platforms that are used for waveform porting. These are the USRP and the SFF SDR. Furthermore, chapter 3 describes how the model-based waveform design was integrated in these platforms.

With the knowledge of the waveform development flow and the SDR platforms, the next step is the design and the port of a real world waveform. For this proof of concept, the professional mobile radio waveform for public and governmental security systems was chosen, known as TETRA. Chapter 4 therefore gives an overview of TETRA and describes the development of the waveform according to the design flow, introduced in chapter 2. Furthermore, the port of this waveform from one platform to another is described. It presents results and benchmarks of the generated code for TETRA on both platforms and introduce the tests for evaluating the real time waveform with measurement equipment.

The detailed description of further ports that were realized is not necessary for this work. Therefore, chapter 5 gives an overview of the challenges and results from other waveform ports.

Chapter 6 finally concludes the work and gives an outlook of waveform development in the future.

2 Waveform Development

2.1 Introduction

The hardware for radio devices changed dramatically in the last decades and so did the development of waveforms. In former times, waveform development consisted mainly of designing electronic circuits for specific waveforms. Nowadays, waveform development means programming on different processing units like GPPs, DSPs or FPGAs. In addition the way of programming changed from a platform specific to a platform independent development. These approaches deal with a trade-off between efficient platform specific code on the one hand and a portable platform independent code on the other. This chapter gives an overview of the existing and common ways for waveform development and describes in more detail a new approach of the model-based waveform development under the aspect of portability. It furthermore gives a closer look into the performance overhead of generated code by measuring the processing time as well as the logic resources and compares it with hand-written code.

2.2 Overview of Waveform Development

2.2.1 Platform Specific Development

Implementing applications for a specific SDR platform is the traditional way of waveform development. The code is optimized for the underlying processing element and the platform specific tools are used to maximize performance in sense of speed, size and power. Although this is a high-performance approach, existing optimized codes can hardly be ported to other processing elements. If you additionally think of

porting the code from GPP to FPGA this means a complete rewriting of the algorithms.

Code Development on Processors

For processors, a lot of programming languages exist, but only a few are used for digital signal processing and embedded systems. This is due to the performance constraints in power, speed and code size. Benchmarks show that C and C++ belong to the most powerful languages in that field [10], [11]. However, it has to be mentioned that benchmarks for dedicated programming languages have only limited significance. This is due to the fact that benchmarks for processing times or cycles measure not the language itself but the application behind. Nevertheless, C and C++ are, according to the results from [10] and [11], the choices of the DSP industry. Figure 2.1 shows the DSP vendor market shares in 2006. It is remarkable that development environments from these vendors only support C based and assembly languages. According to this result, this work focuses only on C, C++ and optimized libraries written in Assembler.

Furthermore, a still recent topic is the choice between the languages C, C++ and assembly. The big advantage and the main reason why assembly languages are still used is the direct mapping from language to processor instruction. This mapping gives the programmer any control to optimize the code in the sense of speed and size. Even if the performance constraints are relaxed, small execution times lead to a decrease of the clock frequency and to a lower power consumption. Nevertheless, high-level languages like C became more and more popular in embedded systems due to three main reasons [13]:

- better portability between hardware platforms
- protection of the investment in program code
- reduced development time

In modern systems, there has to be a trade-off between high-level and low-level programming languages. On the one hand the application should be executed as fast as possible while on the other hand the shrinking time to market calls for portable code. Therefore, almost any vendor for integrated development environments provides user guides and

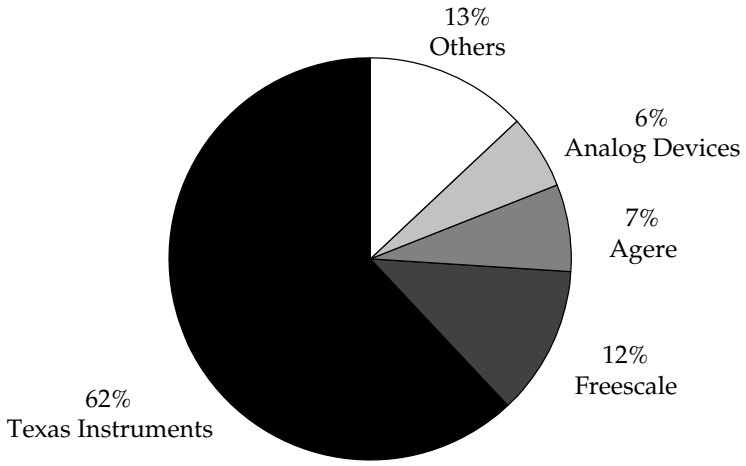


Figure 2.1: DSP vendor market shares in 2006 [12]

tutorials for C optimizations ([14], [15] or [16]). They also claim that well written, efficient and platform independent C code is appropriate for most applications due to platform optimized compilers. The code section represented in listing 2.1 calculates the dot product for two variables in simple C. Listing 2.2 shows the assembly code that is produced by the compiler without optimizations. By enabling the compiler optimizations, the code is reduced to only a few lines, shown in listing 2.3. Due to the fact that the code is written clearly, the compiler is able to optimize it and to use the platform specific architecture. It can calculate two Multiply and Accumulate (MAc) operations and simultaneously load two data elements in one single cycle. This example was written for Analog Devices Blackfin DSP and compiled with VisualDSP++.

Listing 2.1: Example code in C

```
1 for (i=0; i< 150; i++){
2   dotp += b[i] * a[i];
3   sqr += b[i] * b[i];
4   }
```

Listing 2.2: Unoptimized assembler code

```
1  [FP+ -8] = R7;
2  ._ P1L1:
3    R3 = [FP+ -8];
4    R2 = 150 (X);
5    CC = R3 < R2;
6    IF !CC JUMP ._P1L3;
7    R3 <<= 1;
8    P2 = R3;
9    P0 = [FP+ 8];
10   P0 = P0 + P2;
11   R1 = W[P0+ 0] (X);
12   R0 = [FP+ -8];
13   R0 <<= 1;
14   P1 = R0;
15   P2 = [FP+ 12];
16   P2 = P2 + P1;
17   R7 = W[P2+ 0] (X);
18   R7 *= R1;
19   R1 = [FP+ -4];
20   R0 = R1 + R7;
21   [FP+ -4] = R0;
22   R3 = [FP+ -8];
23   R3 <<= 1;
24   P0 = R3;
25   P1 = [FP+ 12];
26   P1 = P1 + P0;
27   R1 = W[P1+ 0] (X);
28   R7 = [FP+ -8];
29   R7 <<= 1;
30   P2 = R7;
31   P1 = [FP+ 12];
32   P1 = P1 + P2;
33   R3 = W[P1+ 0] (X);
34   R3 *= R1;
35   R1 = [FP+ 16];
36   R7 = R1 + R3;
37   [FP+ 16] = R7;
38   R3 = [FP+ -8];
39   R3 += 1;
40   [FP+ -8] = R3;
41   JUMP ._P1L1;
```

Listing 2.3: Optimized assembler code

```
1 LSETUP (._P1L2 , ._P1L3-8) LC0=P1;
2 ._P1L2:
3   A1+= R0.H*R0.H, A0+= R0.L*R0.H (IS)
4       || R0.L = W[I1++]
5       || R0.H = W[I0++];
6
7 ._P1L3:
```

There are several other ways of optimizing C code beside the optimizing abilities of the compiler. However, [17] claims that even an optimum assembler code is merely 10 % faster than a well written C code. In addition the rising speed of processors made it possible to pass by the assembly language in most cases. If the code is still too slow, despite of being well written, the positive impact of the Pareto Principle can be applied. It says that small code portions (10 % - 20 %) are consuming

most (80 % - 90 %) of the system resources [18]. The performance critical code segments can be identified easily and tuned with optimized assembler code. This can be done for example by applying vectorization or parallelization of the underlying processor. However, these code segments can hardly be ported to other platforms.

A similar discussion about the choice between C and assembly language can be observed in the choice between C and C++. C++ is often seen as slow, inefficient and too large for embedded systems. However, these are assumptions that must not be true, due to the fact that obviously C++ is just a superset of the C language. This means that implementing in C and compiling with a C++ compiler should not produce any overhead. Unfortunately that is not really true. In [13] this overhead was investigated by compiling C benchmark programs with the TASKING DSP563xx C and C++ compilers and comparing the speed and size. The result was that the execution speed decreased by no more than 1.6 % while the code size growed by no more than 6.8 %. C++ features are not consuming much more space than elements in C do if they are well designed. The key issue for writing efficient high-level code (independent of C or C++) is that the developer is aware of the functionality at machine code level and the work of the compiler.

Code Development on FPGAs

An FPGA has to be programmed different from a microprocessor or DSP, due to its underlying hardware structure (see section 3.1.2). While processors are implemented in assembly language, which can be mapped to the binary machine code instructions, the FPGA and its underlying logic is represented by an Hardware Description Language (HDL). The HDL is a formal description of the digital logic with reference to the timing behavior. In the beginnings of HDLs only representations for either the logical behavior or the structure existed. This changed in the 1980s when two new languages appeared, representing the logical and structural behavior: Verilog and VHDL. These languages are still dominating the FPGA world, but their distribution is locally different. While Verilog is the most popular HDL at the US west coast and Asia, VHDL is the preferred language in Europe and the US east coast [19]. There are various discussions about the advantages and disadvantages of these two HDLs and several papers compare them based on example applications

[20], [21]. The choice between Verilog and VHDL depends lastly not on technical capabilities but on personal references, tool capability and existing code to reuse.

Portability with platform specific development

Platform specific waveform development is, as indicated by the name, dependent on the underlying processing element. C and C++ code can be ported from one GPP to another without a major decrease in performance. But this is only true under the condition that the operating systems are not changing. The porting of DSP code depends on the underlying architecture. The port from a floating point processor to a fixed point processor forces the compiler to translate every floating point instruction into fixed point instructions with exception handling in case of overflows. Furthermore, a program with processor specific intrinsics can not be ported to a new platform. The sections of the code with the platform specific commands have to be rewritten. Although C and C++ are high level programming languages, there is no assurance that they can be ported without modifications or loss in performance. This is even more dramatic when porting HDL. The choice of language is predetermined by the platform developer and provider of the board support kit. Therefore, a specific language has to be used and code can not be ported if it is available in the wrong language.

2.2.2 GNU Radio

GNU Radio is an open source project for building waveforms for software defined radios [22]. The idea is the interpretation of a waveform as multiple components that are composed together. The components with an input and output port build the signal processing blocks, while blocks with only input or output ports can be named sinks or sources. Therefore, the GNU Radio library supports hundreds of components written in C++. The connection between blocks and the creation of waveforms is accomplished by building a flow graph in Python that holds the components as primitives. The code example 2.4 works as an “Hello World” tutorial for GNU Radio. It generates two sine waves with a frequency of 350 Hz and 400 Hz, adds them and passes the output to the sound card. In lines 13 to 16 the components are defined and

configured. The simplicity of GNU Radio can be seen on line 18 and 19. The components can be connected with the instruction `connect()`. The whole scheduling and necessary inter-process communication is done by GNU Radio itself.

Listing 2.4: Dial tone example in GNU Radio

```
1  #!/usr/bin/env python
2
3  from gnuradio import gr
4  from gnuradio import audio
5
6  class my_top_block(gr.top_block):
7    def __init__(self):
8      gr.top_block.__init__(self)
9
10     sample_rate = 32000
11     ampl = 0.1
12
13     src0 = gr.sig_source_f(sample_rate, gr.
           GR_SIN_WAVE, 350, ampl)
14     src1 = gr.sig_source_f(sample_rate, gr.
           GR_SIN_WAVE, 440, ampl)
15     add = gr.add_ff()
16     dst = audio.sink (sample_rate, "")
17
18     self.connect (src0, (add,0))
19     self.connect (src1, (add,1))
20     self.connect (add, dst)
21
22 if __name__ == '__main__':
23   try:
24     my_top_block().run()
25   except KeyboardInterrupt:
26     pass
```

Despite the comprehensive signal processing library, GNU Radio was only used by few developers through the beginning years. In 2004 Ettus Research LLC released the first version of the USRP as an RF front end that can be connected directly with GNU Radio. Therefore, the community of developers for GNU Radio increased. The next step in the

evolution of GNU Radio was the development of a tool, named as GNU Radio Companion (GRC), which graphically connects and configures components to a flow graph. The equivalent example from listing 2.4 is shown in the GRC environment in figure 2.2. The graphical flow graph can automatically generate the Python code where lower programming experience for SDR developers is demanded.

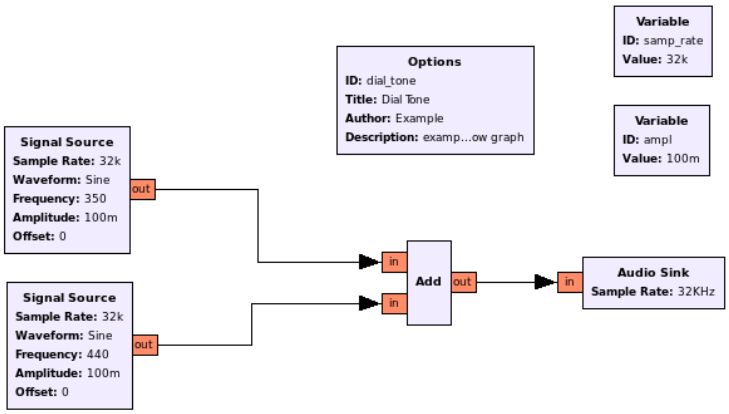


Figure 2.2: Dial tone example in the GNU Radio Companion

GNU Radio composes the waveform with a flow graph and allows the developer to build waveforms that are independent of the underlying hardware. Nevertheless they are not independent of the underlying operating system. In simple terms GNU Radio waveforms are portable between processors that run Linux operating systems. It has to be mentioned that ports of GNU Radio were only successful from a GPP to another GPP. Even if there have been ports to various hardware platforms like Sony's Playstation 3 [23] and the BeagleBoard [24], it has to be mentioned that GNU Radio only ran on the GPP part of these multi core platforms, neither on DSPs nor on FPGAs.

2.2.3 Software Communication Architecture

History of the Software Communication Architecture

Waveform development and portability aspects are also enforced by U.S. military. It maintains hundreds of different legacy radios, which are neither upgradeable nor compatible to each other. In parallel the processor capabilities increased in the last two decades according to Moore's Law [25], [26]. This claims that the number of transistors that can be placed on the same area is doubled every other year. The development in chip technology and in the DSP capability resulted in a shift of the borders between analog and digital domain of a radio towards the antenna as mentioned in section 1.1. Therefore, the JTRS Joint Program Office (JPO) was founded in 1997 to develop a family of the next generation of reconfigurable software-based tactical radios. This system should increase the flexibility and interoperability among the legacy radio systems to reduce the costs for maintenance and to supply and provide the ability to upgrade not only hardware parts but also the software of the radio. The first milestone in this effort was the definition of a common software architecture for these radios: The Software Communications Architecture (SCA). To accomplish the JTRS project goals, the SCA was intended to support portability between SCA compliant implementations, and to use already established frameworks and architectures from the industry. This should result in minimizing the development time due to reuse of waveform components.

Operating Environment

The Operating Environment (OE) builds the interface between a waveform component and the radio [27]. One part of it is the operating system of the radio. To be independent of the underlying hardware and the operating system, the OE provides several interfaces for the waveform components to communicate with its environment: These are the interfaces provided by the Core Framework and CORBA and additionally the Application Environment Profile (AEP). The access points for the application to the operating system are shown in figure 2.3.

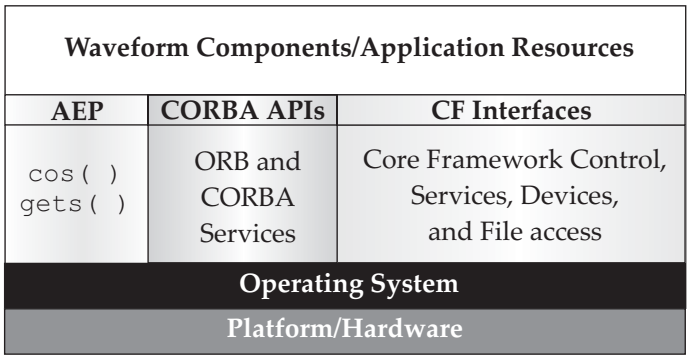


Figure 2.3: Operating Environment with the connections to the platform and the waveform

Core Framework

The Core Framework (CF) provides interfaces to the operating system, which can be used by the waveform or other applications running on the OE. The interfaces give access to various services like the installation, management or configuration of a waveform. Another purpose of the interfaces is to abstract the underlying hardware. Therefore, four classes of CF interfaces are specified:

- Base Application Interfaces
- Base Device Interfaces
- Framework Control Interfaces
- Framework Service Interfaces

The base application interfaces provide access to waveform components like initializing or releasing, configuring or querying the properties of the different components for testing purposes. The software components for access to the hardware resources are called devices, where the basic device interfaces allow interaction with the physical hardware devices on the radio. The framework control interfaces provide a radio wide control of services like installation, deployment or management while the framework service interfaces supply additional services like management of file systems.

CORBA

The Common Object Request Broker Architecture (CORBA) is an open specification for mechanisms calling objects from other processes per remote. The specific characteristic of CORBA is, that these objects can run under different operating systems or on different processor architectures and that they can even be implemented with a different programming language. The remote calls can not be distinguished in the implementation from local calls. This is the main reason of the inclusion of CORBA in the SCA due to the fact that waveform components should exchange information without knowledge of the underlying operating systems and transport mechanisms. Nevertheless, CORBA has also some technical restrictions. The most important is the complexity and the size of its APIs, which are quite larger than necessary [28] and therefore not well suited to an embedded system. To circumvent this, the SCA dictates the use of minimumCORBA, standardized by the Object Management Group. MinimumCORBA is a subset of CORBA designed for systems with limited resources.

Application Environment Profile

The Institute of Electrical and Electronics Engineers (IEEE) and The Open Group specified APIs between application and operating system under the name Portable Operating System Interface (POSIX). The SCA [29] dictates with the AEP a subset of POSIX with 256 APIs. This includes for example mathematical operations like `cos()`, `sin()`, `tan()` or string operations like `gets()`, `getchar()`. Due to the fact that these interfaces are standardized, the access from the waveform component to the underlying operating system, is portable without the use of the CF or CORBA.

Domain Profile

Beside the aspect of portability, an additional requirement of SCA compliant radios is the reconfigurability. Therefore, the Domain Profile provides an essential description of the platform and the waveform application. The Domain Profile consists of several description files written in eXtensible Markup Language (XML). These files describe:

- the various hardware devices
- the waveform components
- the deployment
- the connections between components
- the properties of components and devices

With this information the SCA-compliant radio is able to deploy a waveform, independent of the underlying hardware.

SCA-compliant waveform development

The development of an SCA-compliant waveform requires a good understanding of the SCA CF. However, there are tools that make it easier for waveform developers. Beside professional tools like SCARI [30], Spectra [31] or Component Enabler [32], the Open Source project Ossie [33] builds a complete SCA framework with code generation. It already supports interfaces to the USRP and the USRP2 and was ported to environments including FPGAs [34] and DSPs [35].

2.3 Model-Based Development

2.3.1 OMG's Model Driven Architecture

The Object Management Group (OMG) is a non-profit international computer industry consortium for standardizing distributed object-oriented software systems as well as model-based developments. It created the development and the standardization of modeling architectures like the Unified Model Language (UML) as well as middleware solutions like CORBA, which is described in section 2.2.3. In 2001 OMG adopted the Model Driven Architecture (MDA), which can be seen as an approach for the usage of models in software development to provide portability, interoperability and reusability [36]. These goals can be achieved by the concept of separating the functionality of an application from the capabilities of the underlying platform. The fundamental idea is

to initiate an iterative process in developing models of a specific system and to extend these models in a last step to an application running on a platform. Therefore, the MDA suggests to start the development with a Computational Independent Model (CIM) for the requirements of the system, but which is independent of the way how the system is built. The first step of an implementation is the Platform Independent Model (PIM). This is a model that describes the system but does not show details of its use of any platform. The first implementation with platform details is realized in the Platform Specific Model (PSM). As the name implies, this illustrates how the system uses the underlying platform to fulfill the functionality and the requirements specified in the CIM. The step from one model to another is called transformation. The MDA does not define a specific method for the transformation. However, it gives examples from all manual transformations up to fully automated transformations [36].

In contrast to the MDA, the SCA is also defining its waveforms in a platform independent manner but models the platform itself in a Platform Specific Model with the Domain Profile. The transformation for this PSM to executable code running in real-time on the SDR platform should be done with the Core Framework and CORBA. However, CORBA comes with overhead in processing time and memory usage and is therefore not the best choice for the baseband processing. Due to this, the MDA should be reinterpreted to fit the needs of baseband processing on heterogeneous hardware architectures.

2.3.2 A new interpretation of the MDA

The introduced models CIM, PIM, and PSM build the basis of the MDA and should should be maintained. However, the interpretation should be more application specific concerning the air interface. The CIM that should describe the requirements independent of the implementation is actually the specification of the waveform. While most specifications for industry waveforms are open to public (e.g. specifications of IEEE or European Telecommunications Standards Institute (ETSI)), this is not the case for military waveforms. These waveforms were built for legacy devices where the baseband data were processed with hardware. The specifications are not open and in some cases not even existent. This results in a lot of work for reengineering and creating the CIM when

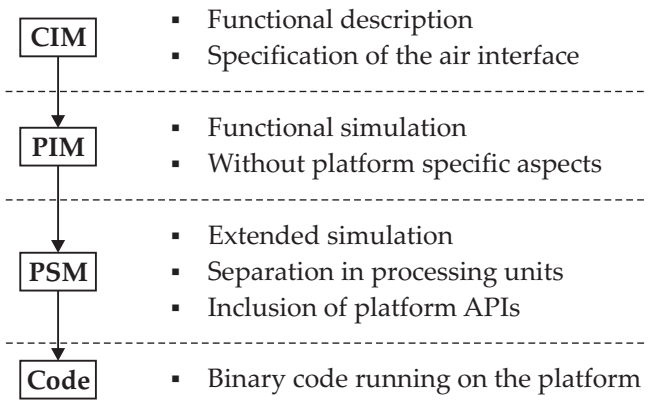


Figure 2.4: Overview of the models, which are defined by the MDA

the waveform should be supported by an SDR. The transformation to the PIM can be accomplished by implementing the waveform without platform specific aspects. Therefore high-level languages can be used, which support good debug and simulation features. The PIM is a model for the implementation of the waveform's functionality. This includes for example the realization of the synchronization schemes or the design of the digital filters. The step to the PSM is done by extending the PIM with platform specific aspects. These can be infrastructural platform aspects like the data buses on the system or the configuration of the RF front ends and ADCs. Further platform specific aspects are processor specific issues like the adaption of algorithms for fixed-point representations. The last transformation from the PSM to an executable code is done automatically with code generation tools like *Real Time Workshop* or *HDL Coder* from the company The Mathworks and processor specific compilers like *Visual C Compiler* from Microsoft, *Code Composer Studio Compiler* from Texas Instruments (TI) or *Intel Parallel Studio*. Another example for this use of the MDA is described in [37] and [38]. Figure 2.4 summarizes the steps from CIM to Code.

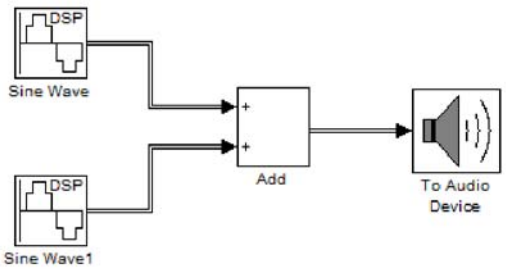


Figure 2.5: Dial tone example in Simulink

2.3.3 Model-Based Design with Simulink

Simulink [39] is a visual programming language, which allows engineers to model dynamic systems. It provides hundreds of blocks, which generate, process and consume data. Furthermore, Simulink supports extensions to generate code in the languages C, C++, Verilog and VHDL. It is therefore used in this work for waveform development. This section should give an overview of the principles of Simulink and its extensions to code generation.

Simulation

In introductions to programming languages the "Hello World" example is used very often. However, this does not make sense for a visual programming language that is used for digital signal processing. A better example is the "dial tone" example from section 2.2.2. The generation and processing of two sine waves gives an overview of the necessary parameters and configurations.

Figure 2.5 shows the dial tone example in a Simulink environment. The blocks require information about the sample time as well as frequency and magnitude of both sine waves. This does not differ from the GRC model described in section 2.2.2. The difference is in the optional handling of the data. While GNU Radio considers the data flow as a stream

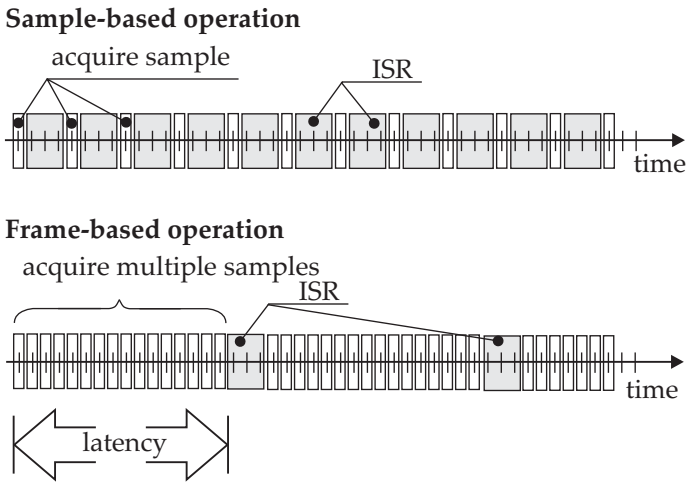


Figure 2.6: Comparison of latency and throughput for sample-based and frame-based processing

without the knowledge of the actual processing, Simulink provides the opportunity to parameterize the size of vectors in the initialization phase. This results in new degrees of freedom. While the processing on single values (sample-based) minimizes latency, the throughput is decreased due to the Interrupt Service Routine (ISR) after each sample. Working on multiple values (frame-based) increases the latency and the memory usage but optimizes the performance with respect to execution time. Figure 2.6 clarifies this relation.

Code Generation from Simulink to C/C++

Real Time Workshop [40] and Real Time Workshop Embedded Coder [41] extend Simulink's functionalities with the ability to generate C and C++ code for real-time systems. For building executable models, it is not only necessary to generate the code for the algorithmic function. This code must also support run time interfaces as well as scheduler and memory transfer. Therefore, the following steps are executed:

- **`ModelInitialize`**: Initializes the code for the run-time interface

and the model. This includes allocation of memory and data that can be calculated prior to the execution of the algorithm.

- **`ModelOutput`:** Calls all blocks in the model that have a sample hit at the current time and produce their output. The output follows a given simulation time step.
- **`ModelUpdate`:** Calls all blocks in the model that have a sample hit at the current point in time and update their discrete states.
- **`ModelTerminate`:** Terminates the program if it is designed to run for a finite time. It deallocates memory and can write data to a file.

The pseudo code in listing 2.5 shows how these functions are integrated in the generated code.

Listing 2.5: Structure of a generated model in pseudo code [40]

```
1  rtOneStep()
2  {
3    //Check for interrupt overflow
4    //Enable "rt_OneStep" interrupt
5    ModelOutputs(); // Major time step.
6    LogTXY();      // Log time, states and root
       outports.
7    ModelUpdate(); // Major time step.
8  }
9
10 main()
11 {
12   ModelInitialize();
13   //including installation of rtOneStep as an
14   //interrupt service routine for a real-time clock
15
16   while(time < final time){
17        rt_OneStep();
18     //Background task
19     }
20   ModelTerminate();
21   //Mask interrupts (Disable rt_OneStep() from
       executing.)
```

```
22  //Complete any background tasks.
23  //Shutdown
24 }
```

The C++ code from the dial tone example shown in figure 2.5 was generated for a better overview of these functions working with a model. Listing 2.6 shows the initialization of this configuration with the step size according to the sample rate of 32 kHz and the memory allocation of the block input and output arrays.

Listing 2.6: Initialization of the model

```
1 dial_tone_initialize(){
2  dial_tone_M->Timing.stepSize = (0.032);
3  dial_tone_M->ModelData.blockIO = ((void *) &
      dial_tone_B);
4  (void) memset(((void *) &dial_tone_B), 0,
5              sizeof(BlockIO_dial_tone));
6  ...
7  }
```

In listing 2.7 the actual signal processing is shown for the `ModelOutput` function, which is in this case the addition of two sine waves with a frame length of 1024 as the iteration size for the loop.

Listing 2.7: Signal processing in the dial tone

```
1 dial_tone_output(){
2  ...
3  for (i = 0; i < 1024; i++) {
4   dial_tone_B.Add[i] = dial_tone_B.SineWave[i]
5                 + dial_tone_B.SineWave1[i];
6  }
7  ...
8  }
```

In the code section from the `dial_tone_update` function in listing 2.8, two parameters are shown. The first is the call of the sound card with the memory address of the adders output. The second is the increment of the step size.

Listing 2.8: Update of the model's states

```
1 dial_tone_update(){
2   ...
3   LibUpdate_Audio(
4          &dial_tone_DWork.
               ToAudioDevice_AudioDeviceLib[0],
5          &dial_tone_B.Add[0], 0, 1024, 0);
6   ++dial_tone_M->Timing.clockTick0;
7   dial_tone_M->Timing.t[0] = dial_tone_M->Timing.
       clockTick0 *
8                dial_tone_M->Timing.stepSize0;
9   ...
10  }
```

The models are finalized by terminating the call of the sound card and freeing all memory space as shown in listing 2.9.

Listing 2.9: Termination of the model

```
1 dial_tone_terminate(){
2   ...
3   LibTerminate(
4          &dial_tone_DWork.
               ToAudioDevice_AudioDeviceLib[0]);
5   LibDestroy(
6          &dial_tone_DWork.
               ToAudioDevice_AudioDeviceLib[0], 1);
7   ...
8        }
```

Code generation from Simulink to Verilog/VHDL

The HDL Coder extension of Simulink [42] provides the ability to generate Verilog or VHDL code for a model. This leads, with the support of C code generation and simulation, to a system that is highly capable of building and transforming models for heterogeneous processing platforms. The code generation shall be introduced by the already known dial tone example. However, the output is not connected to a sound card but to an output port for the dedicated module. Some minor changes

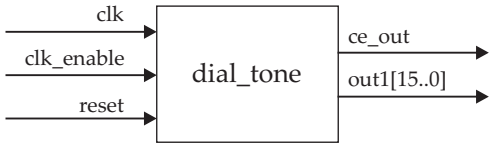


Figure 2.7: Structure of the generated `dial_tone` module

have to be applied to the model to assure that the code can be generated. Due to the fact that FPGAs do not support floating point arithmetic, the sine waves must be configured to generate fixed point output. In difference to DSPs, FPGAs are not working on arrays. This means that the processing changed to be based on samples. With these changes, HDL code can be generated.

Listing 2.10: Extraction of the generated Verilog code

```
1  module dial_tone(...)
2
3  always @(address_cnt_1)
4    ...
5    begin
6      case(address_cnt_1)
7        7'b0000000 : Sine_Wave1_out1 = 16'
             b0000000000000000;
8        7'b0000001 : Sine_Wave1_out1 = 16'
             b0000100000001001;
9        ...
10     endcase
11   end
12   assign Add_add_cast = Sine_Wave_out1;
13   assign Add_add_cast_1 = Sine_Wave1_out1;
14   assign Add_add_temp = Add_add_cast +
       Add_add_cast_1;
15   assign Out1 = Add_add_temp[15:0];
16
17 endmodule // dial_tone
```

Listing 2.10 shows an extraction of the generated Verilog code for the dial tone example. The sine waves are generated by a lookup table and

the output is assigned to the highest 16 bit of the sum of the sine waves. Figure 2.7 shows the structure of the generated Verilog module.

2.4 Overhead of Code Generation

The efficiency of generated code is still highly controversial in the development of radio systems. While some are complaining about the fact that the code is too slow baseband signal processing, others argue that there is no overhead. In this section, the processing time of the generated code was measured on different processing units under varying conditions like compilers or data types. It is furthermore compared with an unoptimized hand written code and an assembly-based optimized code. Parts of these measurements were also published in [43].

2.4.1 Code Generation for GPPs

The benchmarks for generated C/C++ code were executed on an Intel Core 2 Duo CPU (P8400) with a clock frequency of 2.28 GHz. The processor is working with a Windows 7 operating system and three compilers:

- In the following the C/C++ compiler, which is integrated in Visual Studio 2005, is named *VS*. The comparison with newer versions of the Visual Studio Express Edition showed no differences in speed. Therefore only this version is listed in the results.
- The Local C Compiler in the following denoted as *LCC* is a small open source C compiler, where in the benchmarks, version 2.4.1 is used.
- Intel Parallel Studio XE2011 integrates a C/C++ compiler in the development environment, which is especially suited for Intel architectures. The compiler is subsequently denoted as *IPS*.

The C/C++ code was generated with Real Time Workshop version 7.6. For better readability and comparison with different processors, the benchmarks show, beside the time consumption of the evaluated code, also the number of cycles. These are calculated by the processing time

multiplied with the processor frequency. Therefore, it is just a time scaling.

Figure 2.8 shows the overhead that the generated code consumes after every ISR due to scheduling, logging and status updates. This is the time the code remains in the gray boxes in figure 2.6. It can be seen that the choice of the compiler dramatically influences the differences in processing times. The Intel Parallel Studio (IPS) consumes less than a third of the time than the Local C Compiler (LCC). The figure indicates also the influence of the frame size on the generated overhead. While the processing time of the non-functional code in figure 2.8 remains constant, the processing time for code inside the `model_output` function increases with the frame size and leads to a better throughput in total.

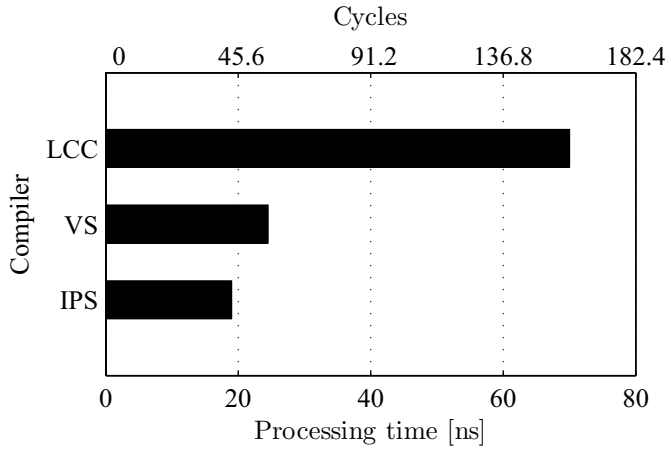


Figure 2.8: Processing times of the generated non-functional code, which was compiled with various compilers

One of the essential signal processing functions in SDRs are Finite Impulse Response (FIR) filters. Figure 2.9 shows the benchmark of an FIR filter compiled with the aforementioned compilers. To present values independent of the number of taps and frame size, the results are normalized to one input sample and one tap, leading in the best case to only one MAc operation. Additionally to the named compilers, the data type (real or complex) and the code (hand written or generated) are

compared. The results for the compilers from figure 2.8 are also valid for this benchmark. The Visual Studio (VS) and IPS produce the fastest code. The generated code consumes always less time than the hand written code. Note that the hand written code is not optimized in any sense except in the implementation of a circular buffer. The source code of the unoptimized hand written FIR filter can be found in appendix A.1. The fastest code is the generated code for an FIR filter with real data and compiled with the IPS. Only three cycles are consumed on the GPP per real tap and real sample. The processing time for a complex operation takes about ten cycles, which is also a good result, taking into account that the complex MAc operation needs four real multiplications and two additions.

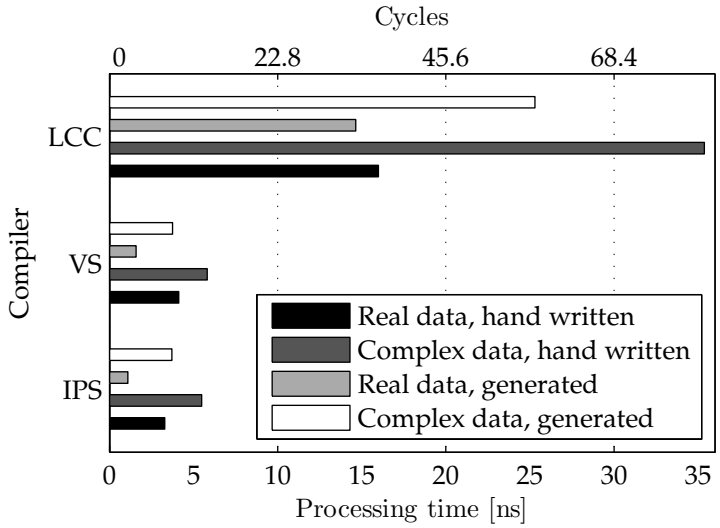


Figure 2.9: Benchmark of an FIR filter on a GPP

Another important algorithm used in a waveform implementation is the Fast Fourier Transformation (FFT). Figure 2.10 compares the processing times of an FFT for *generated code*, *unoptimized hand written code* and an *optimized code*. This optimized code comes from FFTW, which is a C subroutine library for calculating discrete Fourier Transforms. It is supported by an open software platform and is in performance comparable

to machine-specific, vendor-optimized codes. The good performance is achieved by querying the parameters through the operating system and tuning the architecture automatically for the fastest results [44]. The unoptimized code is the straight implementation of the Cooley-Tukey algorithm, which can be found in appendix A.2.

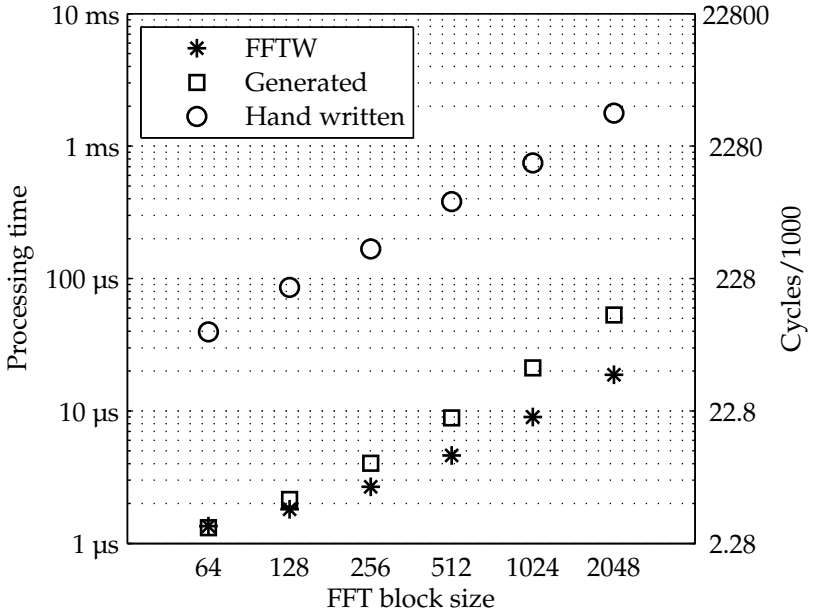


Figure 2.10: Benchmark of the FFT on a GPP

For better comparability the compiler times are not included in the above results. The differences in the times for the compilers are in accordance to the previous results, which are shown in figure 2.8. The input data are complex floating point vectors with a length according to a power of two. The hand written code is about thirty times slower than the generated and optimized code. They are varying over the FFT block size according to the benchmarks of the FFTW in [45]. There, it is shown, that the FFTW implementation achieves the best speed results with FFT sizes between 512 and 2048. The speed of the generated code does not fully reach the optimized FFTW implementation.

2.4.2 Code Generation for DSPs

The benchmarks for generated C code were measured with a C64x+ core on Texas Instrument's DM6446 system on chip, which is described in more details in section 3.3. The processing times and cycles are in accordance with the GPP measurements when taking the scaling factor of the DSP's clock frequency of 594 MHz into account. The C code is compiled with Code Composer Studio (CCS) from the DSP vendor TI. Figure 2.11 shows an overhead of 3.8 µs of the generated non-functional code. Referring to the number of cycles, this is about fifty times more than the non-functional code for the GPP. This is mainly caused by the underlying fixed point processor that has to deal with floating point code.

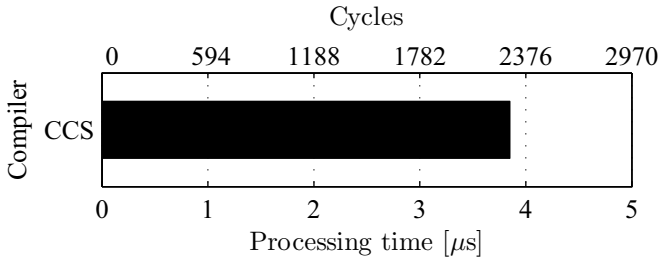


Figure 2.11: Overhead of the non-functional code on a DSP

The performance loss of floating point code on a fixed point processor can also be seen in the benchmark of an FIR filter represented in figure 2.12. While the difference between floating point and fixed point arithmetic with hand written code makes a factor of ten, the generated fixed point code is about hundred times faster than the generated code dealing with floating point data types. The optimized code with a special library for digital signal processing comes from TI and is only available in fixed point arithmetic. The generated fixed point code needs twice the time, which is not a bad result compared with the hand written, non-optimized code.

Figure 2.13 compares the results for the FFT for *hand written code*, *generated code* and the *assembler optimized code* from TI's signal processing library. It has to be noted that the hand written code uses floating point

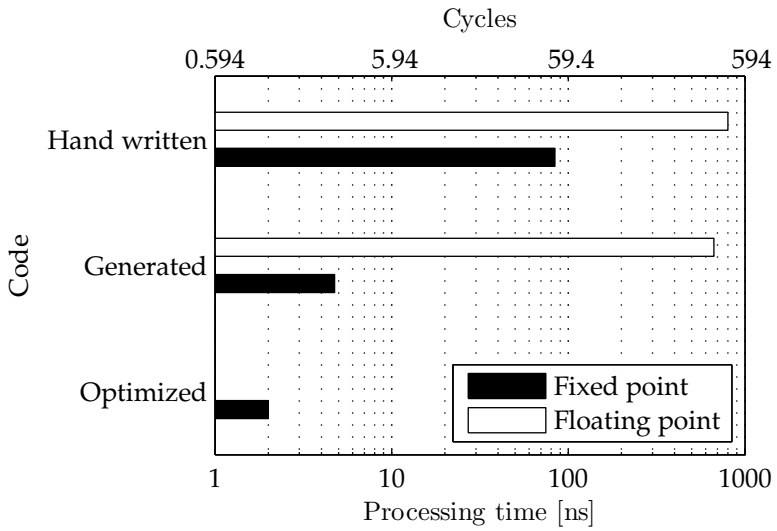


Figure 2.12: Benchmark of an FIR filter on a DSP

arithmetic in contrast to the other code. This leads to the poor results, which are in processing time several decades slower than the optimized codes. The speed up for the optimized code is about four times higher than the generated C code.

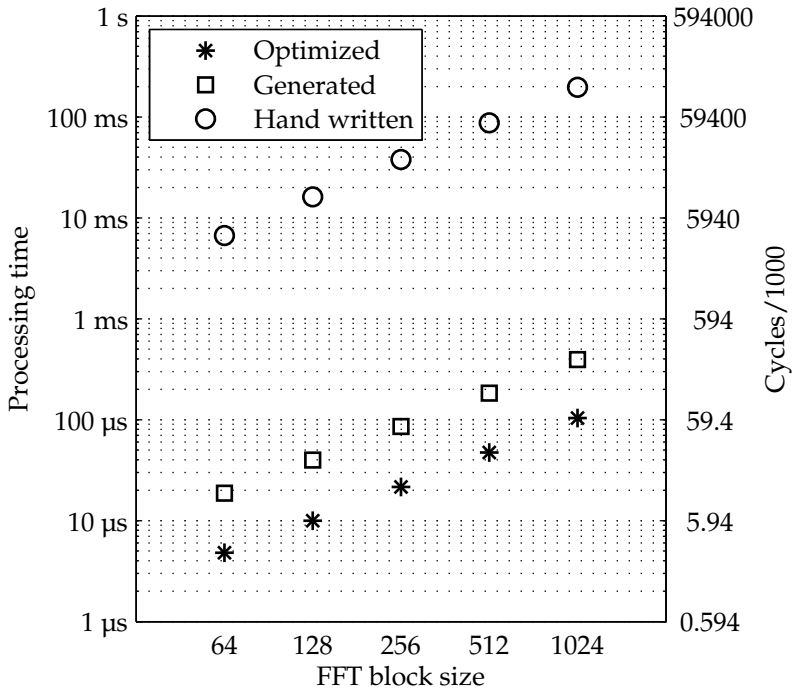


Figure 2.13: Benchmark of the FFT on a DSP

2.4.3 Code Generation for FPGAs

In contrast to the generated codes for DSPs and GPPs, the generated HDL code creates no overhead for the system. The main task of the FPGA on a SDR is the adaptation of the DSP sample rate to the higher data rate of the DAC and vice versa. Therefore, decimation and interpolation filters have to be realized. Figure 2.14 shows the resource usage

of interpolation filters with a Cascaded Integrator-Comb (CIC) structure. The filters were implemented on a Virtex 4 FPGA from Xilinx with VHDL [46] and on a Cylone FPGA from Altera with Verilog [47].

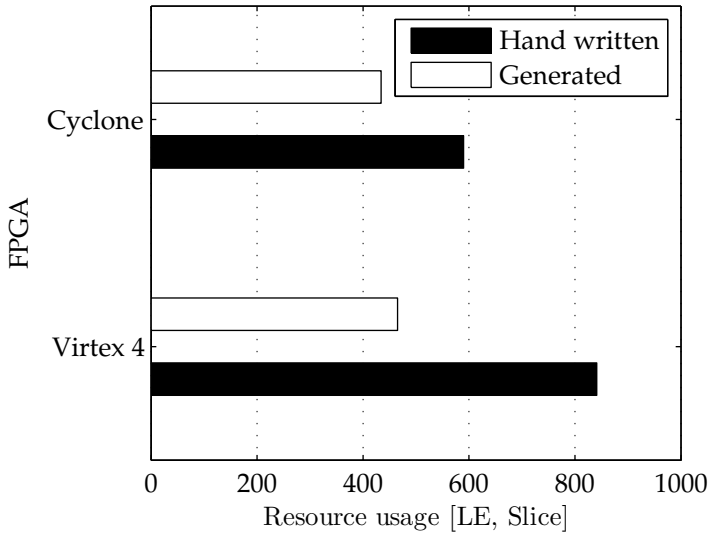


Figure 2.14: Benchmark of a CIC decimation filter on different FPGAs

The interpolation factor was 128 and the bit lengths for input and output were 16. The bit length inside the filter increased to 44 due to bit pruning. The filter was implemented with four cascaded stages. Due to the different hardware architectures, the resource usage between both FPGAs can not be compared. The FPGAs from the company Altera are based on Logic Element (LE), which mainly consists of one programmable register, one carry chain and one Lookup Table (LUT) with four inputs. This is different from the Xilinx approach. The smallest logic block is named “slice”, comprising two LUTs with four inputs, one carry chain and two programmable registers. Therefore, the resource usage of the filter was measured in LEs for the Cyclone and in slices for the Virtex FPGA. The generated code is smaller in the sense of space than the hand written code but the difference is not as dominating as in the previous sections.

Beside CIC filters, also FIR filters, used for decimation in time, are benchmarked. Figure 2.15 shows the difference between an optimized halfband decimation filter in comparison to the generated polyphase decimation filter. The filter length was 31 but due to the coefficient properties, only eight multiplications were needed. The other coefficients were zero. Furthermore, due to the symmetry of the filter some coefficients were equal. The generated filter did not recognize this symmetry and hence the possibilities to use this for optimization.

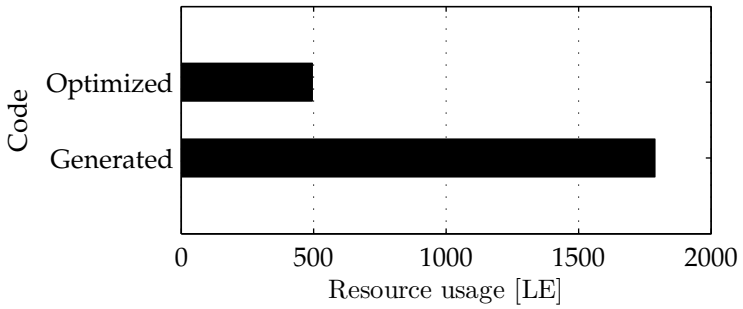


Figure 2.15: Comparison of the resource usage for a 31-tap halfband filter

To get an overview how the number of taps correlate with the resource usage in generated code, Figure 2.16 shows the used LEs of a decimate by two filter. The bit length of input and output signal as well as the filter coefficients was 16 bit, the bit length of the accumulator was 32 bit. The results of figure 2.16 were measured on a Cyclone FPGA without hardware multipliers.

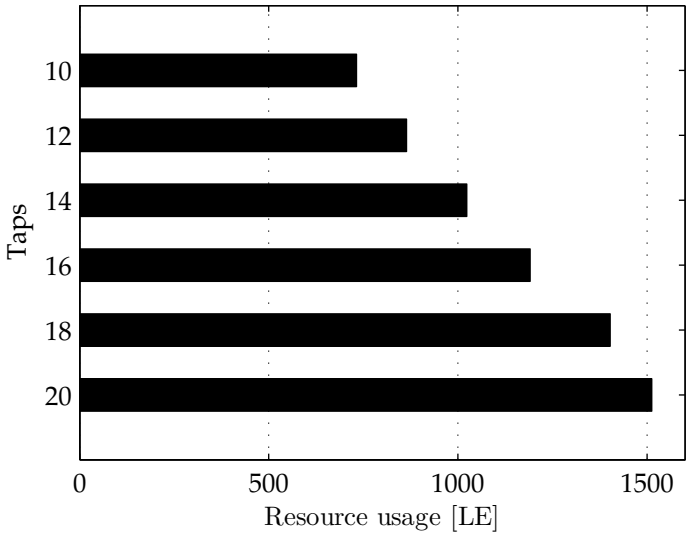


Figure 2.16: Comparison of the resource usage of an FIR decimation filter with various tap lengths

3 SDR Platforms

This chapter gives an overview of the platforms and their components that are used in this work. The chapter starts with a general overview of SDR platforms and presents the different used platforms in detail. Furthermore, the integration in the development flow is shown, which was introduced in section 2.3.

3.1 SDR Platforms in General

Software Defined Radio is associated with digital signal processing, implemented on reconfigurable processors. The interface to the analog world is done by the ADC and DAC, which are as close as possible to the antenna. It should be highlighted that reconfigurable digital signal processing is the core part of any SDR. The RF signals in the physical analog world are on high frequencies at low power and the ADC sampling rate, dynamic range and power consumption are limited [48]. Therefore, the RF front end should only extract a dedicated frequency band of interest and convert it into a signal suitable for the ADC. After AD conversion the digital signal processing is in dependence on the underlying digital hardware. This scheme is represented in figure 3.1.

3.1.1 RF Front Ends in SDR

Receiver Architectures

The receive side in RF front ends is for down conversion of desired signals in a form that the ADC can handle the signals and further to suppress interferences in the band of interest. The most common architecture to achieve this is the superheterodyne receiver [49]. It converts the signal from the desired Radio Frequency (RF) into a fix Intermediate

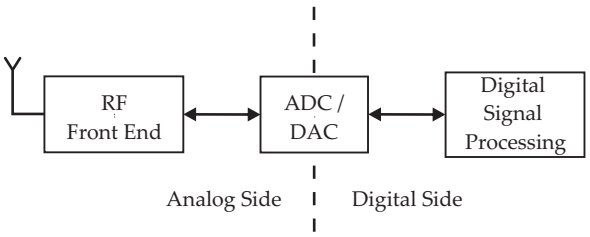


Figure 3.1: Structure of a general SDR platform

Frequency (IF). The IF is lower than the RF but higher than the bandwidth of the signal. This fact is also the origin of the name "superheterodyne" where hetero and dynamis are the greek words for "different" and "power". This means the superposition of different forces: the desired signal on RF and the mixing signal in the distance of IF from RF. There are plenty of commercial components available on the market for standard IFs like 10.7 MHz for FM broadcast. Two examples of superheterodyne topologies are shown in figure 3.16 and 3.17.

A serious problem in superheterodyne receivers is the presence of image spectra, which are symmetrically located above and below the mixing frequency in the distance to the IF. Therefore, image rejection filters are mandatory for superheterodyne architectures. This leads to a tradeoff in the choice of the IF. On high IFs image suppression is not very demanding but channel filtering is more difficult at these frequencies and the same is true vice versa: On relative low IFs the requirements for channel selection are relaxed but the filters for image suppression are more demanding [50]. To circumvent this problem, downconversion is realized over multiple stages with the presence of a filter after each downconversion step. This yields to merely partial channel selection at each step and as a result the relaxation of the quality of the filters.

Another way to circumvent the problem of images is the homodyne architecture, which is also called direct-conversion or zero-IF. The principle is to mix the desired signal with a carrier on its center frequency. This yields to an IF with a frequency of zero and to the circumvention of the problem of image frequencies. Furthermore, high quality IF filters and subsequent downconversion stages can be replaced with baseband amplifiers and low-pass filters, which eases the monolithic integration. An example of such a receiver topology is presented in figure 3.5. Nev-

ertheless, having here the advantages of image rejection and monolithic integration, other problems occur. DC offsets can corrupt the signal due to the baseband location. Another problem is the I/Q phase shift mismatch due to errors in the ninety degree phase shifter.

Transmitter Architectures

The transmit path of an RF front end converts the signal from IF or baseband up to the desired carrier frequency. It furthermore amplifies the signal before transmitting through the antenna, where in-phase and quadrature components are provided by quadrature modulation. Compared to receiver architectures, the requirements for transmitters are lower regarding noise, interference rejection or band selectivity. Two types of transmitters are of interest: the direct-conversion transmitter and the two step transmitter.

Similar to the direct-conversion receiver, the mixing frequency of a direct-conversion transmitter is equal to the transmitted carrier frequency. Examples of this architecture can be found in figures 3.5 and 3.16. The problems described for the direct-conversion receiver remain for the transmitter: leakage of the power amplifier and I/Q phase mismatch. These disadvantages can be circumvented by the two-step transmitter, which is shown in figure 3.17. The quadrature modulation is done digitally and the Power Amplifier (PA) works on different frequencies as the carrier from the Local Oscillators (LOs).

3.1.2 Digital Signal Processing in SDR

The digital signal processing part is the core of an SDR to provide the computational performance. The SDR consists of various processors and reconfigurable logic to fulfill the underlying radio communication standards and baseband processing algorithms. The most common components in this field are: FPGAs, DSPs and GPPs.

Field Programmable Gate Arrays

An FPGA is an Integrated Circuit (IC) consisting of thousands of programmable logic blocks with reconfigurable interconnections to route signals between them [51]. A general logic block includes a LUT for the combinatorial logic, a programmable register to store elements and a multiplexer for internal routing [52]. However, there are slight differences in actual designs from different vendors to this general structure. The two major companies on the world market for FPGAs are Xilinx and Altera. Xilinx's main logic resources are named Configurable Logic Blocks (CLBs), which are connected to switch fabrics. Any CLB itself contains four slices, which are grouped in pairs. The slices are comparable with the general custom logic blocks described above. Each slice contains two LUTs, two programmable registers and one carry chain for multiplexing and adding [46]. This structure is similar to the actual Altera design that comprises several Logic Array Blocks (LABs) grouped into rows and columns on the FPGA. Each LAB contains 16 LEs, which include themselves one LUT, one programmable register and one carry chain [47]. Due to these differences in architecture it is hard to compare measurement results for FPGAs from different vendors. Furthermore, most vendors provide their FPGAs in various configurations: with several multiplication cores and embedded micro controllers or with embedded bus systems. The different structures make it even harder to compare.

Digital Signal Processors

In the evolution of microcomputers special architectures for digital signal processing appeared first in 1982 on the market with TI's TMS32010 [3]. This was the first commercial DSP that made use of specialized hardware for multiplication and accumulation in one single cycle. Today, MAc units are mandatory for modern DSPs but at that time comparable processors required multiple add and shift operations each consuming one cycle. But not only add and shift operations became the bottleneck for high speed, real time processing. In early processors there was generally one single bus interface to the memory, which could only be accessed once per cycle. The Harvard architecture circumvents this by changing the memory structure to use separate buses: one for the data memory and the other for the instructions. This memory architecture

is still used, but the memory access was accelerated over the years. Beside the faster underlying hardware, there are also algorithms that are adapted to the memory structure in DSP applications in order to find memory addresses more quickly. Further, DSPs are mostly working on vector bases in block processing. The high end DSPs of today provide up to eight arithmetic units and support multiple MAc operations in one single cycle. Other features in the present high end processors are specialized arithmetic units like Viterbi accelerators or video codecs [53].

General Purpose Processors

Today, more and more GPPs are used to handle signal processing tasks in the SDR environment. On the one hand, this is due to the possibility to write software with high-level languages and to debug directly on the host instead of using a Joint Test and Action Group (JTAG) device. On the other hand, GPPs made huge steps in their capability to process digital signals. This is due to extensions like Single-Instruction Multiple-Data (SIMD) capabilities, which can process one instruction on multiple data segments in a single cycle. Other reasons are high clock speeds and multiple cores that GPPs became alternatives to DSPs in digital signal processing.

3.2 Universal Software Radio Peripheral

The Universal Software Radio Peripheral (USRP) is not itself a Software Radio Platform, it only provides an interface from a host Personal Computer (PC) to several RF front ends. The real baseband processing is calculated on the PC. The USRP was originally developed by Matt Ettus [54] as a cheap interface for the GNU Radio software, which anyone could afford or even build himself. The schematics and the source code for the firmware can be downloaded for free. Today, the USRP is not only an interface for the GNU Radio community. Users can utilize it with various software radio development tools like OSSIE, MATLAB/Simulink, LabView or they build their applications in C++. The company Ettus Research became the dominant seller of low cost software radio platforms and has been acquired by National Instruments in February 2010.

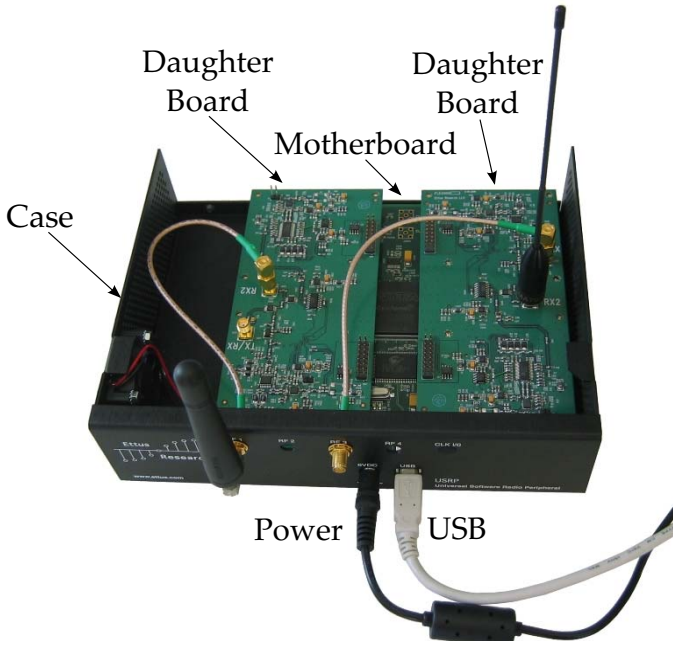


Figure 3.2: The USRP motherboard equipped with two daughter boards

In the following sections, the USRP is named as the motherboard, while the RF front ends are named as the daughter boards. A fully equipped USRP motherboard with two FLEX400 daughter boards is shown in figure 3.2. You can also see at the front side of the case the power connection and the USB cable for high speed data connection to the host.

3.2.1 The USRP Motherboard

The motherboard is equipped with a Cyclone EP1C12 FPGA from Altera [55], which provides the interfaces to the Universal Serial Bus (USB) and to the ADCs and DACs. Two ADCs and two DACs are integrated in a single AD9862 [56] conversion chip. Due to the fact that the board is equipped with two AD9862, the motherboard provides a total of four transmit and four receive paths. The ADCs support a sample rate

of 64 MHz with a quantization resolution of 12 bit. Furthermore they provide Programmable Gain Amplifiers (PGAs), bypassable low pass decimation filters and a bypassable Hilbert filter. The transmit side of the AD9862 provides two 128 MHz DACs with 14 bit resolution as well as interpolation filters and digitally tunable real or complex upconverters.

Figure 3.3 shows the motherboard in more details. The FPGA is located at the center of the board and the two ADC/DAC combinations are in adjacency on the right and left side. The chip, shown in figure 3.3, below the FPGA is a Cypress' FX2 USB microcontroller [57], which supports the connection to the USB peripheral. The white plug-ins over and under the AD9862 conversion chips are the connectors to the daughter boards.

3.2.2 The USRP Daughter Boards

Ettus Research provides a variety of daughter boards for the USRP, working in a frequency range from baseband up to 5 GHz. In this subsection only the Flex400 and Flex2400 daughter boards are described. These front ends were used for the implemented waveforms in chapter 4 and chapter 5.

The Flex400 supports a frequency range from 400 MHz to 500 MHz while the Flex2400 is designed for RF signals between 2400 MHz and 2500 MHz. Other USRP daughter boards are described in [54]. The Flex400 and Flex2400 are designed similarly in a way that they are using a single conversion topology for upconversion and downconversion.

For the Flex400, the received signal is sent to the AD8348 direct quadrature demodulator [58]. The demodulator needs a carrier for mixing the RF signal into baseband. This signal comes from a Voltage Controlled Oscillator (VCO)/Phase Locked Loop (PLL) combination, which is able to tune the frequency in steps of 4 MHz. This does not achieve an exact direct-conversion architecture but results in a low IF topology, where the intermediate frequency varies between baseband and 4 MHz. The last step for downconversion of the in-phase and quadrature components into baseband is done on the motherboard inside the FPGA with a Digital Down Converter (DDC).

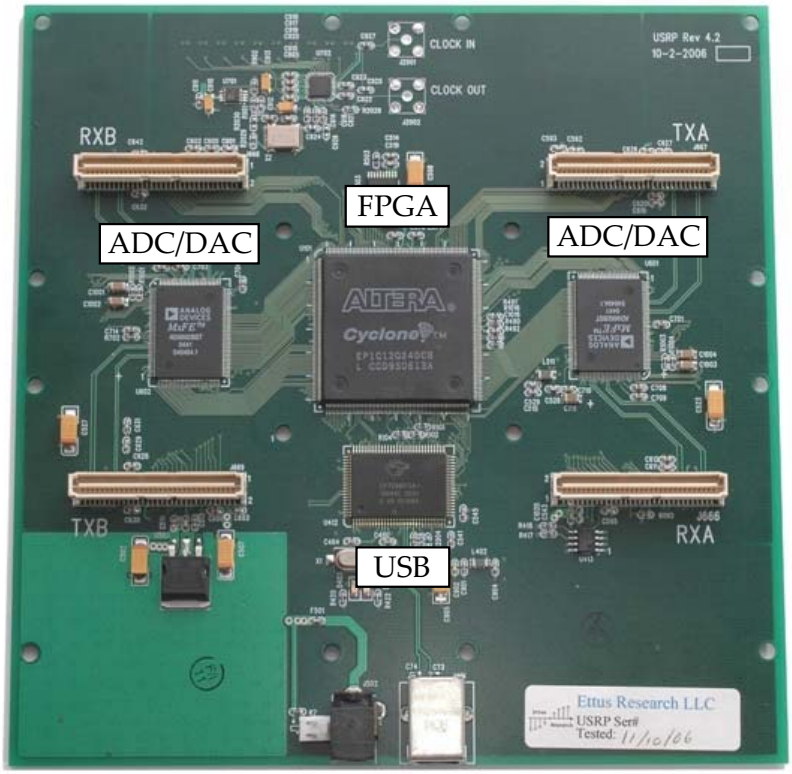


Figure 3.3: The USRP motherboard [54]

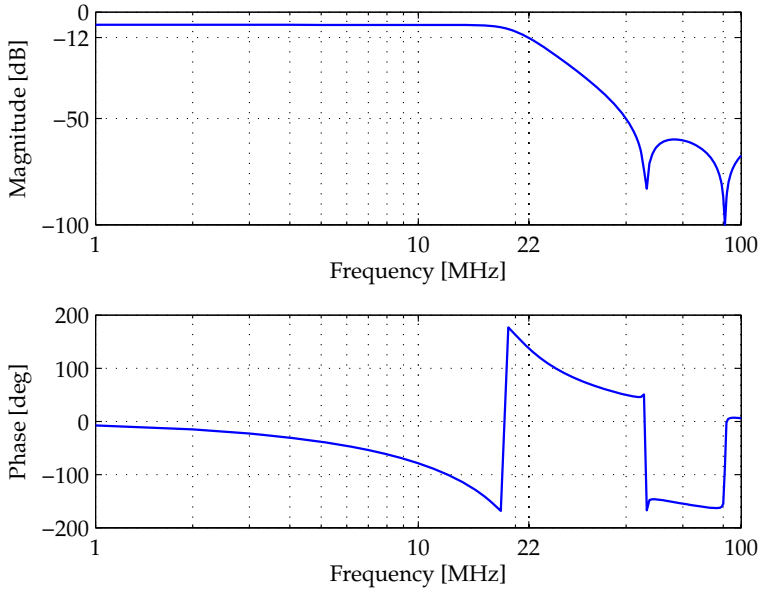


Figure 3.4: Transfer function of the anti-aliasing filter for the Flex400 and Flex2400 daughter boards

The architecture of the Flex2400 is equal but an AD8347 [59] quadrature modulator replaces the AD8348 due to the higher frequency range. The receiver structure is shown in figure 3.5. The Anti-aliasing filter is the same for both daughter boards and comprises resistances, capacitors and inductors with known values due to the open source schematics. The transfer function was calculated and is shown in figure 3.4. The 6 dB cut off frequency is at 22 MHz.

The transmit path is designed analogically to the receive path as shown in figure 3.5. The AD8345 quadrature modulator [60] converts the signals to a radio frequency between 400 MHz and 500 MHz for the Flex400. The local oscillator is the same version as for the receive path, where digital up conversion of the baseband signal to an intermediate frequency between baseband and 4 MHz is necessary. This can be done with the internal Digital Up Converter in the AD9862 [56]. The difference between

EP1C12	Logic Elements	
	12060	100 %
Tx	1214	10 %
Rx	3448	29 %
Tx & Rx	3895	32 %

Table 3.1: Minimum number of Logic Elements used in the FPGA

Flex2400 and Flex400 daughter board is again the different frequency range of the modulator.

3.2.3 Platform Specific Constraints

Figure 3.5 shows a signal flow graph for a software radio platform with a Flex400/Flex2400 front end. As described before, both daughter boards have the same structure but with different modulators and demodulators as well as different VCO/PLL combinations. The USRP motherboard works as a conversion module. Custom signal processing can be implemented either on the FPGA or on the GPP respectively.

Table 3.1 gives an overview how much space the FPGA provides. Without any logic usage, Altera's Cyclone EP1C12 has a total of 12060 LEs. However, all these elements cannot be used for signal processing due to the fact that the interfaces to the AD9862 as well as buffers and interfaces to the USB chip have to be implemented. With the transmit and receive sides, approximately 68 % of the internal logic can be used for signal processing. The space occupancy for a receive only configuration is about three times higher than for a pure transmitter. This is due to the fact that in the receive side a Digital Down Converter (DDC) has to be implemented. This is not the same for the transmit side, since the internal Digital Up Converter (DUC) on the AD9862 can be used. These results are summarized in table 3.1.

Another platform specific limitation is the USB connection between motherboard and GPP. While USB 2.0 supports data rates up to 480 Mbps, the USB controller from Cypress together with the library: *libusb* on the GPP achieves a maximum throughput of 228 Mbps. With this, a complex bandwidth of approximately 7 MHz with 16 bit wide in-phase and

quadrature samples is achieved. To get the lower sample rate in comparison to the conversion rate of the ADCs, a decimation filter has to be implemented with a factor not less than eight on the receive side. On the transmit side an interpolation filter with an upsampling rate not less than four is necessary. These filters have to be implemented for in-phase and quadrature signal components.

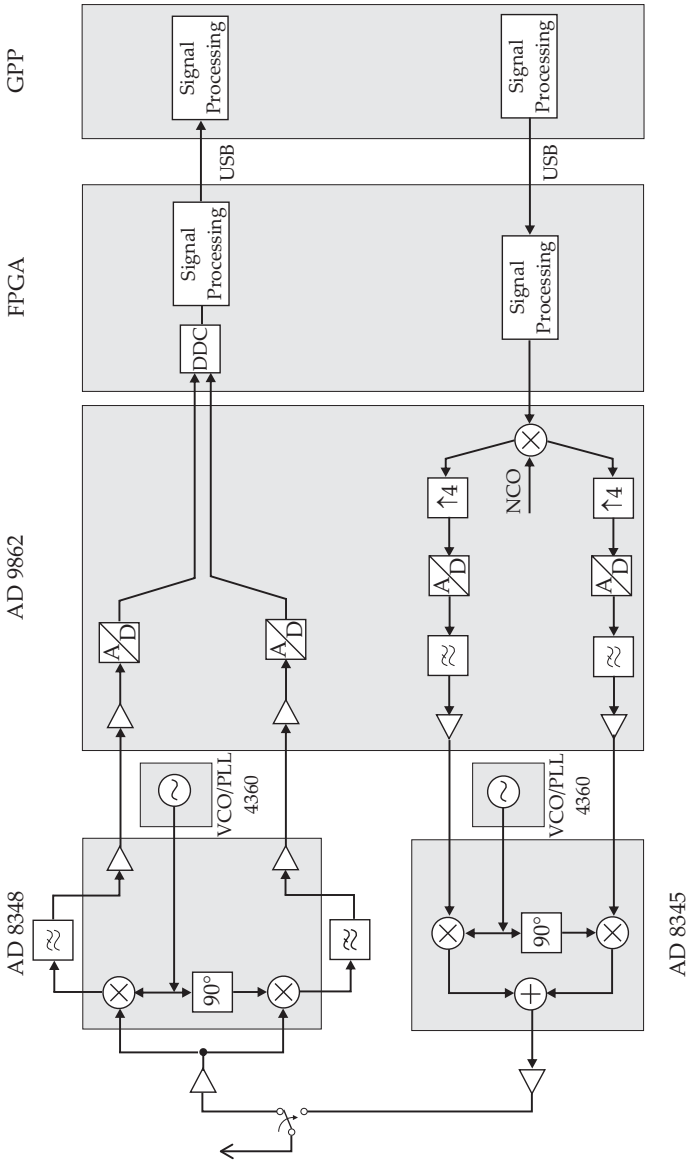


Figure 3.5: Signal flow in a USRP

3.2.4 Integration in the design flow

Integration of the FPGA

Figure 3.6 gives an overview of the FPGA, where it should be highlighted that the baseband data are coming from the ADC and are passing to the DAC with fixed data rates and word lengths. Therefore, on the receive side a DDC is implemented that can be configured through a register by the GPP. As mentioned previously, a DUC is already included at the transmit side on the DAC. The *Rx Chain* and *Tx Chain* blocks are black boxes with interfaces to in-phase and quadrature signal components as well as to the clock. This makes the integration of the generated Verilog code possible. The generated time scopes inside the black boxes are connected automatically with buffers at the transmit or receive side of the USB. In addition, interrupt commands for the GPP are provided in a form that the GPP can handle the sample rate.

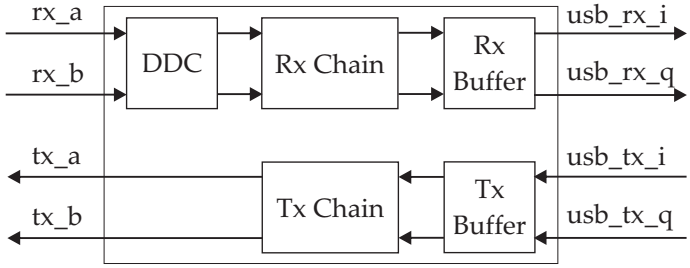


Figure 3.6: Black boxes in the FPGA on the USRP

The transformation from PSM to the raw binary file for the FPGA is presented in figure 3.7. The functional code is generated from the PSM. This is the code that is integrated in the *RX Chain* at the receive path and in the *TX Chain* at the transmit path. A template makefile, written in Tool Command Language (TCL), is doing the synthesizing and mapping of the code. It includes information concerning the FPGA, the synthesizer and mapper as well as the physical connections to the chip. Additional Verilog code for the platform specific functions like buffers are integrated in the development flow. The resulting file can be uploaded to the FPGA.

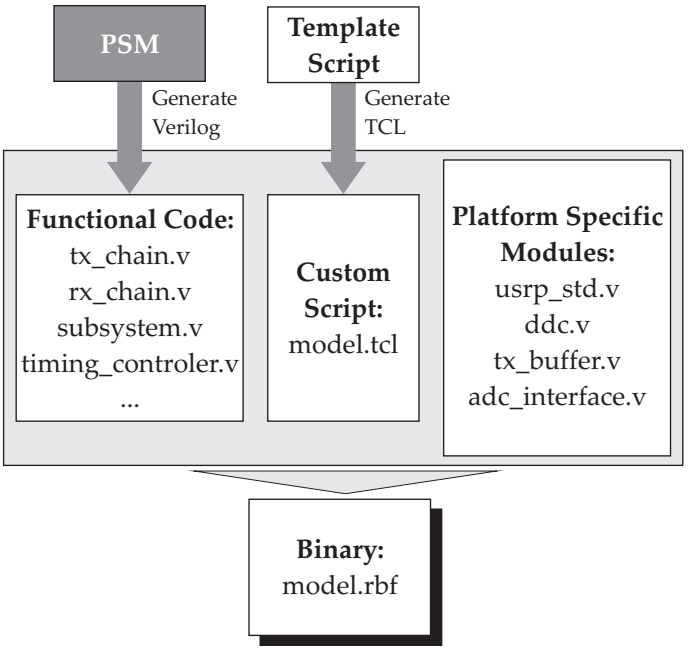


Figure 3.7: Transformation from the PSM to the bitstream on the Cyclone FPGA

Integration of the GPP

The integration of the GPP is done by implementing an interface which is based on the USRP API from GNU Radio. This interface must be extended for code generation and inclusion in Simulink. Therefore two APIs are build: a `usrp_source` object that handles the data transfer from the USRP to the GPP and a `usrp_sink` object that handles the connection from GPP to USRP. Figure 3.8 shows these interfaces in the Simulink environment.

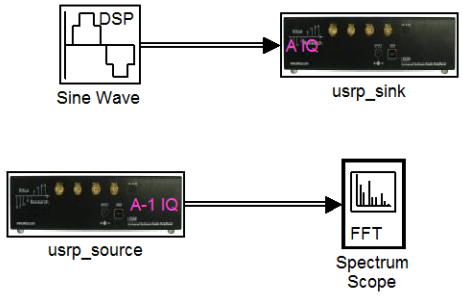


Figure 3.8: Example for the use of the APIs for integration of the USRP in a model

The interfaces are implemented such that for code generation, the platform specific objects are linked to the functional code. According to the example in figure 3.8 the `usrp_sink` receives the data from the sine wave and sends it to the USB buffer and then to the FPGA. The `usrp_source` gets new data from the USB buffer and passes it to the spectrum scope. To maintain real time processing, the USRP sends interrupts when a new frame of data can be accessed in the USB buffer. An error is produced if these data are not used or not passed over before the next interrupt arrives. This ISR controls the timing and the scheduling of the code running on the GPP.

For configuring the data, a Graphical User Interface (GUI) was developed shown in figures 3.9 and 3.10. The parameters for timing and scheduling can be configured via the GUI shown in figure 3.9. The ISR

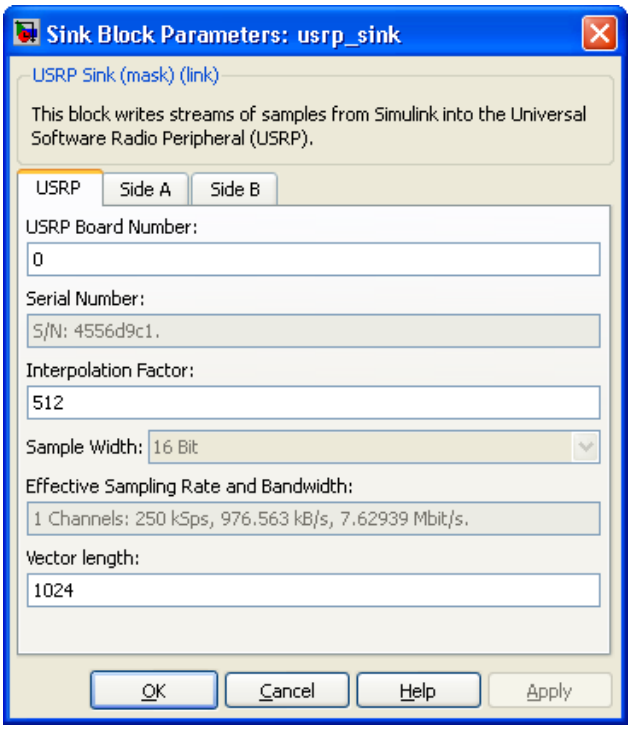


Figure 3.9: GUI for the USRP sink with an example for configuration parameters

time period T_{ISR} can be calculated as follows:

$$T_{\text{ISR}} = \frac{f_{DAC}}{I} \cdot N_{\text{frame}}, \tag{3.1}$$

where f_{DAC} is the sampling rate of the DAC, I is the interpolation factor and N_{frame} is the vector length. This would lead to an ISR time period of 4.096 ms for the configuration shown in figure 3.9.

Figure 3.10 shows the configuration for the USRP daughter boards. The daughter board ID is written to the text field and gives additional information like the maximum tuning frequency and the range of the gain. All these data are written to the APIs. The interfaces for the daughter board expect the tuning of the frequency and the gain as

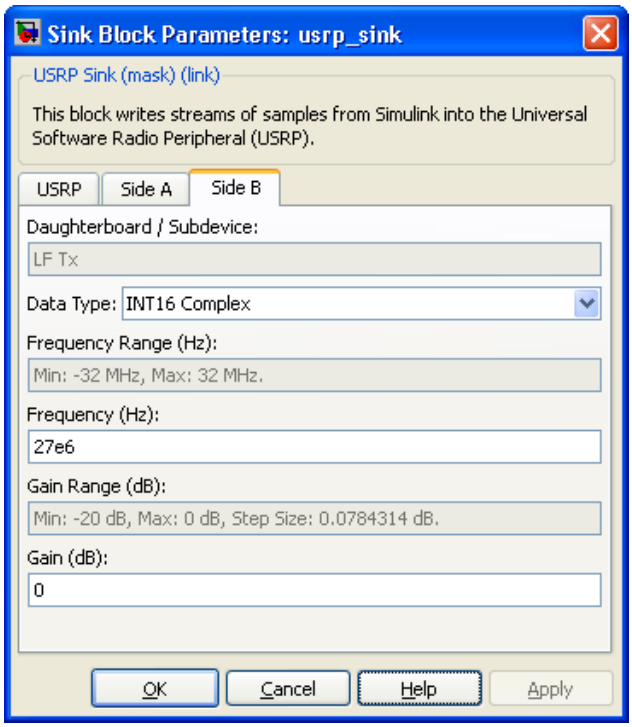


Figure 3.10: GUI for the USRP subdevice, showing configuration values

input data. Additionally, the data type can be configured as well as the selection of the antenna type through the antenna ID. The setting of the configuration can be done for both daughter boards by selecting one of the two tabs: *Side A* or *Side B*.

Figure 3.11 gives an overview of the transformation from PSM to executable code, according to the transformation from PSM to bitstream shown in figure 3.7. As described in section 2.3.3, the functional code is generated in a C++ code. With the integration of the `usrp_sink` or `usrp_source` objects, the functional code can be linked to the source code of the interfaces. The template makefile starts the development environment as well as the compiler and linker. The output is a file that can be executed on any host PC operating with Windows or Linux operating system.

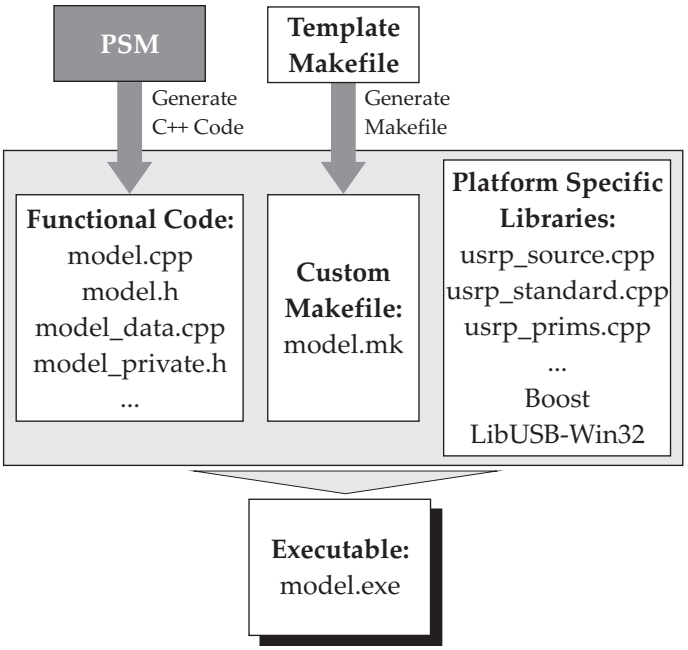


Figure 3.11: Transformation from the PSM to an executable file for the GPP

3.3 Small Form Factor SDR

Lyrtech's Small Form Factor SDR Development Platform (SFF SDR DP) [61] was one of the first embedded development platforms for Software Defined Radios. The platform was originally developed as an evaluation board for Texas Instrument's Da Vinci System on Chip (SoC). Lyrtech used the processing power of the included processors for Software Radio applications and extended it with ADCs, DACs and various RF front ends. The SFF SDR DP consists of three modules, which are shown in figure 3.12. The Digital Processing Module (DPM) is the signal processing layer at the bottom with a GPP/DSP combination in addition to an FPGA. The Data Conversion Module (DCM) is the middle layer and is equipped with an additional FPGA for sample rate conversion as well as ADCs and DACs. The Radio Frequency Module (RFM) finally is the RF front end on the top.

3.3.1 Digital Processing Module (DPM)

As the name indicates, the whole signal processing on the SFF SDR is placed on the DPM. Therefore, the board is equipped with a Virtex 4 SX35 FPGA [46] and TI's TMS320DM6446 SoC [62] as the central processing units. The SoC comprises two processing cores: a C64x+ fixed point DSP and an ARM GPP. An overview of the architecture of the DPM is given in figure 3.13. The exact properties of the components are described in table 3.2. The FPGA and the DSP are connected via the Video Processing Sub-System (VPSS). This is a 16-bit synchronous video data transfer port for the DSP, which was originally designed as a video processing chip. The VPSS was adapted to provide high data rates between both processing units with the Video Processing Front End (VPFE) as the interface towards the DSP and the Video Processing Back End (VPBE) in the opposite direction back to the FPGA. Both interfaces are independent from another. Furthermore, there are two additional connections between FPGA and DSP: The External Memory Interface (EMIF) allows access to read from and to write to registers, while the Audio Serial Port (ASP) is connected to the on board PCM3008 audio stereo codec through the FPGA. Beside the audio codec, the Virtex 4 allows access to several buttons, switches, LEDs and to the DCM over a proprietary bus system. The peripherals of the DSP consists of the RS232

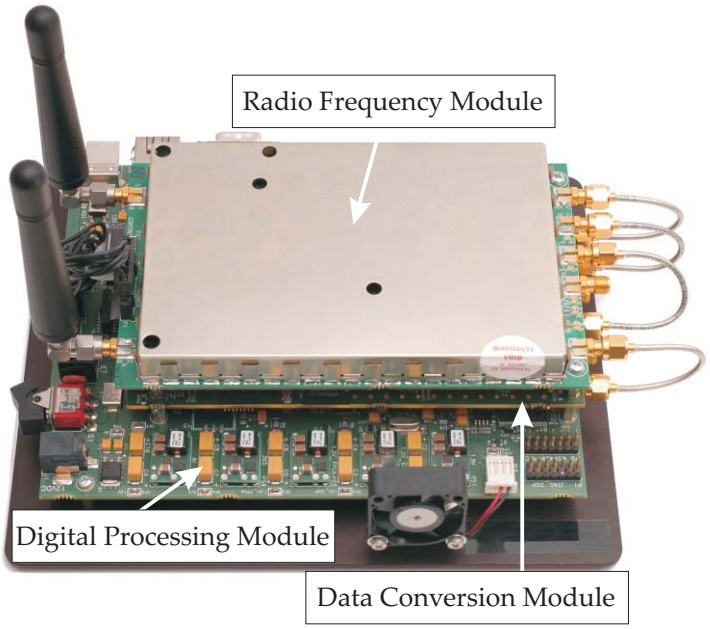


Figure 3.12: Illustration of the SFF SDR DP, showing the three different modules

TMS320DM6446 DMP SoC		
Core Frequency		594 MHz
Peak MMAcs		4752
On-chip L1/SRAM	DSP GPP	112 kB 40 kB
On-chip L2/SRAM	DSP GPP	64 kB 0 kB
ROM	GPP	16 kB
EMIF	EMIFA DDR2	16 bit / 8 bit 32 bit / 16 bit
Timer	GP WD	64 bit 64 bit
Virtex 4 XC4VSX35 FPGA		
CLB array		96x40
Slices		15360
Block RAM bits (#)		3456 (192)
18 x 18 multpliers		192
Digtal Clock Manager		8

Table 3.2: Properties of the digital processing units on DPM

interface to configure the IP-address, the DDR2 RAM external memory with 128 MB and a high speed USB bus, which is not supported by the firmware right now. Further peripheral equipments are an Ethernet connection for board access over a host PC, a slot for an SD card and another 128 MB memory, which stores the bootloader and a kernel image of the operating system and the file system.

3.3.2 Data Conversion Module (DCM)

The DCM is the interface between the digital world of the DPM and the analog RF front end. The DCM is equipped with two ADCs, two DACs, an FPGA and a clock distribution unit. The ADC is a Texas Instruments ADS5500 Analog-to-Digital Converter [63] with a maximum sampling frequency of 125 MHz and a resolution of 14 bit. As shown in figure 3.14, two ADS5500 are needed for a complex baseband signal received from a direct-conversion receiver. This is different to the Radio Frequency

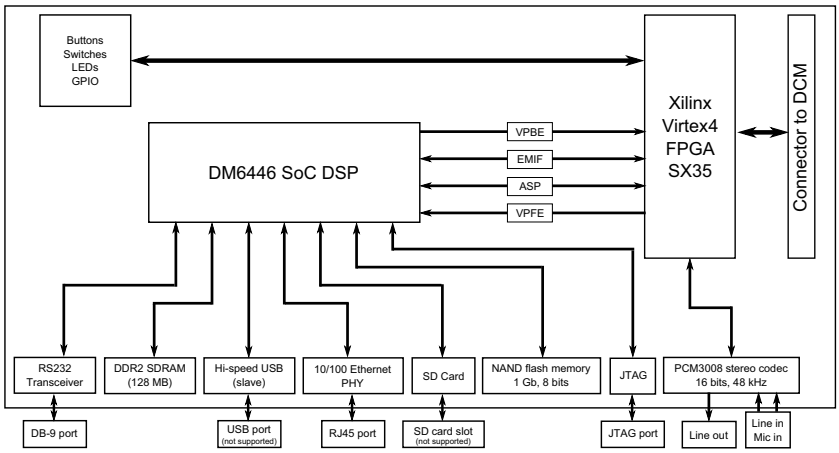


Figure 3.13: Structure of the Digital Processing Module

Modules (RFMs) described in section 3.3.3, where the incoming signal on an IF on 30 MHz or 44 MHz is real. Therefore, the second input port and the second ADC is not used. After sampling, the digitized data are passed to Xilinx's Virtex-4 LX25 FPGA, which is the interface between the conversion chips and the proprietary expansion connector. This FPGA here is not designated for custom signal processing and its logic cannot be changed.

The transmit side of the board consists mainly of TI's DAC5687 Dual-Channel DAC [64]. This chip comprises two DACs with a bit resolution of 16 at a maximum sampling rate of 500 MHz. Furthermore, it provides an interpolation filter for upsampling by a factor of 2, 4 or 8, as well as a complex modulator and a programmable amplifier.

The clock of the DCM controls the sample rate converters. It is based on Analog Device's AD9511 PLL [65]. The reference clock for the PLL is supported from the on-board 10 MHz internal LO or from an external clock. With the limited frequency range from 800 MHz to 1400 MHz of the VCO, the target clock frequency f_{out} can be calculated by

$$f_{\text{out}} = \frac{f_{\text{VCO}}}{\text{div}} = \frac{N}{R \cdot \text{div}} \cdot f_{\text{ref}} \,. \tag{3.2}$$

In equation (3.2), div is the clock divider, which is realized as a 5-bit

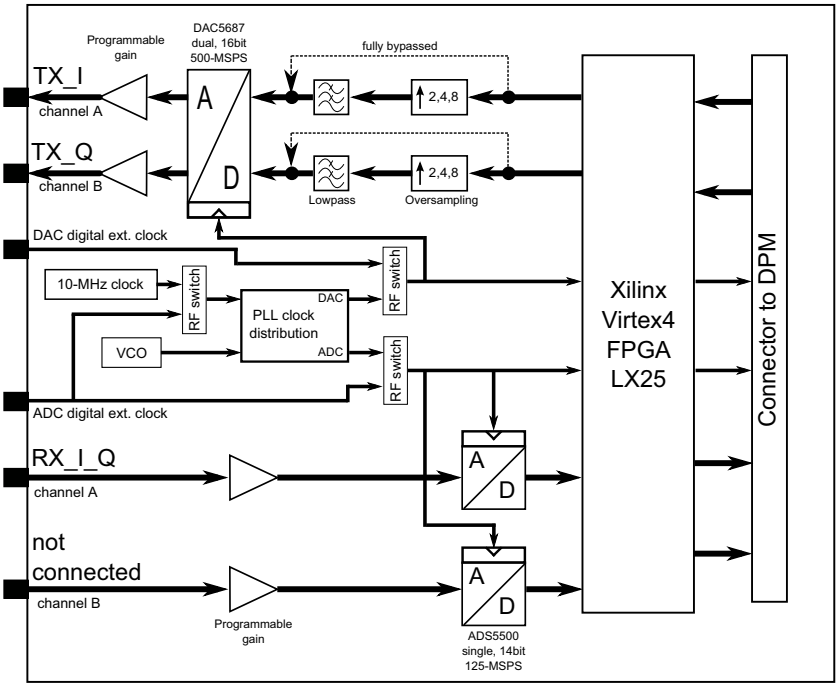


Figure 3.14: Structure of the Data Conversion Module

counter. This leads to a division range of $[1 \dots 32]$. R is the PLL reference divider with a division range between one and 16383 due to the implemented 14-bit counter. The feedback divider N is a combination of a 3-bit prescaler and two counters A (6 bit) and B (13 bit). Through combinations in the prescaler, any integer value for N in the range of $[1 \dots 262112]$ can be selected. Figure 3.15 gives an example for the possible division factors between N and R with a given reference LO at 10 MHz. Due to the limitations of the VCO in the interval between 800 MHz to 1400 MHz, the ratio between feedback and reference divider is limited to

$$\frac{N}{R} = \frac{f_{\text{VCO}}}{f_{\text{ref}}} = [80 \dots 140] \, . \tag{3.3}$$

These are the upper and lower bounds for the ratio, shown in figure 3.15. The curves represent any possible value for the clock divider div. The remaining parameters can finally be chosen as follows:

$$\frac{N}{R} = \frac{P \cdot B + A}{R} \tag{3.4}$$

With this clock distribution circuit any sampling frequency with high accuracy can be achieved. Therefore no more complex resampling algorithms have to be implemented on the FPGA or on the DSP to provide the system's base sampling rate.

3.3.3 Radio Frequency Module

In the following two subsections the Tunable Low Band RF module and the WiMAX RF module are described. These two modules are used in conjunction with the SFF SDR DP.

Tunable RF Module

The Tunable RF module consists of a superheterodyne receiver with three stages, which converts signals between 200 MHz and 1 GHz to an IF of 30 MHz. It is equipped with five analog filters and three local oscillators. The first filter, close to the antenna, is a low-pass filter that suppresses all frequencies above 1 GHz. The following LO is a combination of PLL and VCO, which supports frequencies between 1775 MHz and

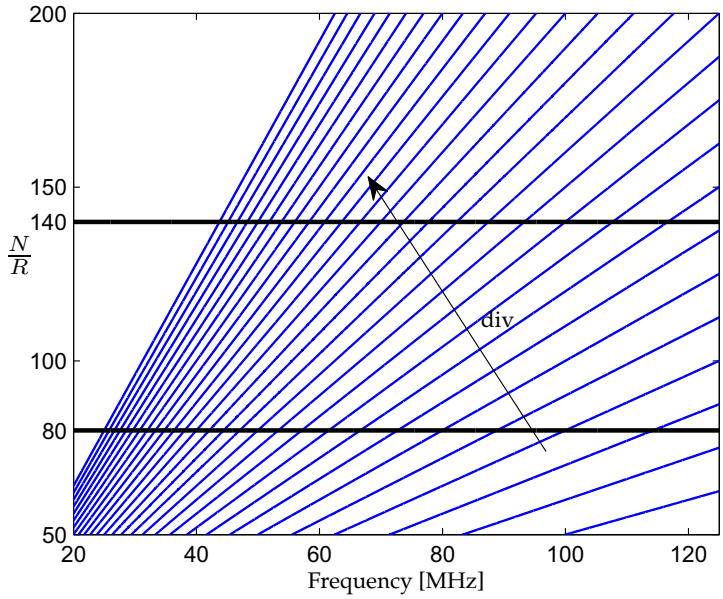


Figure 3.15: Ratios between N and R over the target frequency for different divider values div

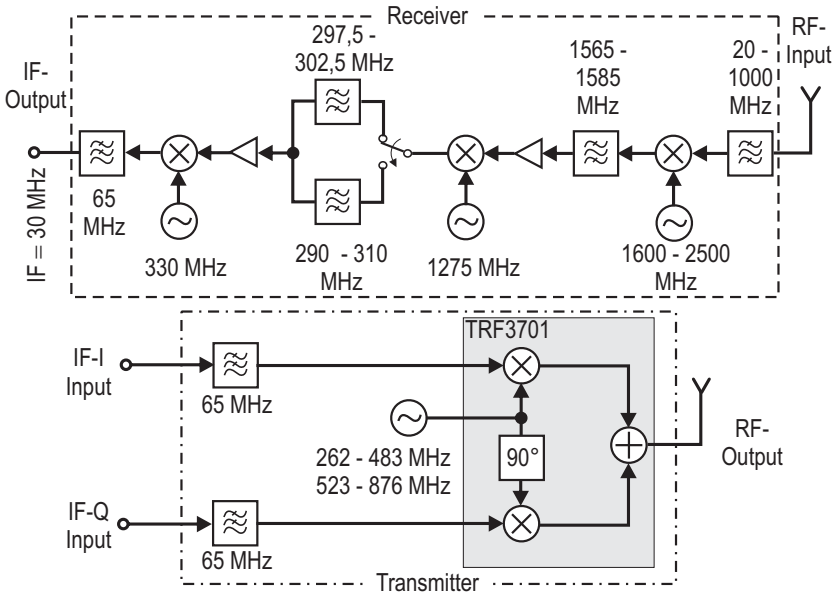


Figure 3.16: Structure of the Tunable RF Module

2575 MHz and converts the RF signal to a fixed frequency of 1575 MHz. This is the center frequency of the second 20 MHz wide bandpass filter. With a switch, two filters with different signal bandwidths of 5 MHz or 20 MHz can be selected. These bandpass filters have a center frequency of 300 MHz. Before the bandpass filters, the signal is down converted with the second LO with a mixing frequency of 1275 MHz. The last step to an IF of 30 MHz is achieved with the third conversion stage of 330 MHz and the final low pass filter for the images.

The transmit side of the RF front end is built according to a direct-conversion transmitter, using TI's quadrature modulator TRF3701 [66]. An in-phase and a quadrature signal are mixed with the incoming signal from the LO, which is composed of a PLL, a VCO and a didivide-by-two prescaler. In case when the prescaler is activated, a frequency of 262 MHz up to 483 MHz is generated while a frequency of 523 MHz up to 876 MHz is generated when the prescaler is deactivated.

WiMAX RF Module

Similar to the Tunable RF module, the WiMAX RF module is a super-heterodyne receiver. The supported frequency range of the front end is from 2.3 GHz to 2.7 GHz. This is also the frequency range for the wireless communication standard WiMAX. However, the WiMAX RF module has no bearing with the WiMAX specification but the frequency range. It converts signals from the RF over two stages to an IF of 44 MHz. As shown in figure 3.17, most components are integrated in ICs. The RF signal is limited by a 400 MHz wide bandpass filter working on 2.5 GHz. The first downconversion is done by TI's TRF1115 [67] low-noise down converter. It mixes the signal with an incoming carrier on a fixed frequency of 456 MHz. On this frequency, the signal is filtered by a bandpass with a bandwidth of 24 MHz. The final conversion on the IF to 44 MHz is done by a combination of PLL and IF down-converter (TRF1112 [68]) and is followed by the last filtering on a bandwidth of 7 MHz or 22 MHz. These last bandpass filters are switchable and controlled by TRF1112.

The transmit side of the WiMAX RF module is designed as a two stage architecture composed of two ICs and two bandpass filters similar to the receive side architecture. The first up-converter (TRF1121 [69]) expects the transmit signal on a 18 MHz IF and mixes it up to a fixed frequency of 325 MHz. Furthermore, it works as a LO distributor for the second

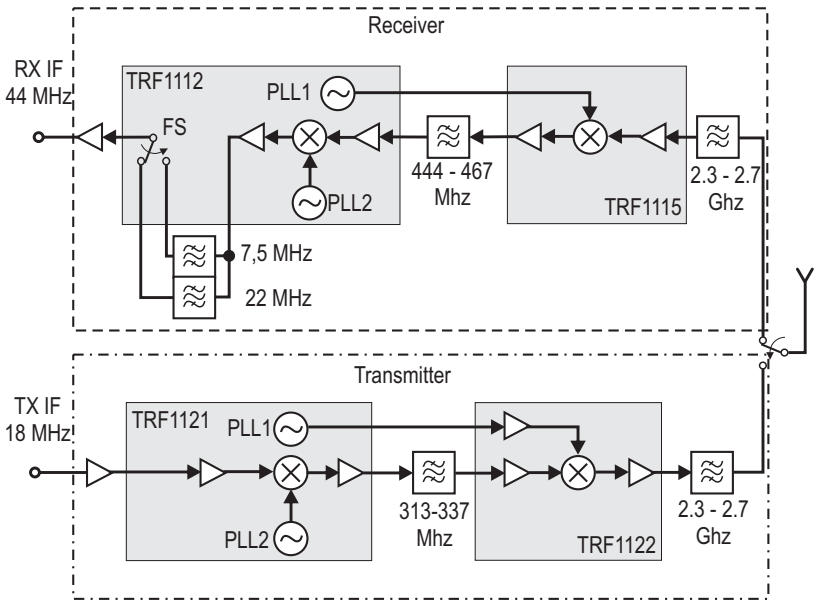


Figure 3.17: Structure of the WiMAX RF Module

upconversion stage, where the mixing on frequencies between 2.3 GHz and 2.7 GHz is done (TRF1122 [70]). The bandpass filter between the upconversion stages limits the signal to a bandwidth of 24 MHz, while the last bandpass filter on 2.5 GHz limits the frequency on a bandwidth of 400 MHz.

3.3.4 Platform Specific Constraints

The C64x+ DSP is one of the processing units for the SFF SDR DP. This is in contrast to the USRP concept, where any GPP with an operating system can be used as a processing unit. With a clock frequency of 594 MHz, the DSP runs very fast, but the memory space for the signal processing code is limited to approximately 176 kB. Code lengths that exceed this size have to be partitioned and moved to the external Synchronous Dynamic Random Access Memory (SDRAM). Although it is dimensioned with a size of 128 MB, the access time for read and write operations takes several cycles. This can slow down the running code until violating the real time constraints.

Another limit for waveform development on the platform is the interface between DSP and FPGA. Originally intended to connect the processor to video peripherals, the VPSS was rebuilt for data transfers from and to the FPGA. With the underlying firmware it is not possible to achieve higher data rates than 64 Mbps. This yields to a bandwidth of 2 MHz assuming 16 bit wide in-phase and quadrature components. Waveforms with higher bandwidths have to be processed on the FPGA.

As shown in table 3.2, the Virtex4 provides 15360 slices and 192 multiplier components. Similar to the platform constraints for the USRP, shown in table 3.1, not all logic cells can be used for waveform development. Table 3.3 summarizes the number of logic cells for different RF modules for the following configurations: single transmitter, single receiver or transceiver. It is remarkable that approximately 15 % of the resources are used for internal logic. This is due to the implementation of several First In, First Out (FIFO) queus for the VPFE, VPBE, ADC and DAC. These buffers are built independently of the actual existence of a transmit or receive side. Further logic resources are used to implement the On-chip Peripheral Bus (OPB) subsystem which provides access to the different peripherals. The differences for transmit and receive paths are due to the archiecture of the connected front end. The receive signal from

Virtex 4 SX35		Slices		Multipliers	
		15360	100%	192	100%
WiMAX RF	TX	1942	12%	7	3%
	RX	1981	12%	7	3%
	TX & RX	2046	13%	9	4%
Tunable RF	TX	1907	12%	5	2%
	RX	1981	13%	7	3%
	TX & RX	2016	13%	7	3%

Table 3.3: Space occupation with different RF modules and transmit receive configurations

the tunable RF module is a real signal on a 30 MHz IF. This makes downconverting and the implementation of a Direct Digital Synthesizer (DDS) and two multipliers necessary. For the transmit side, a quadrature modulator is necessary on the tunable front end. The transmit side for the WiMAX module expects a real signal on 18 MHz IF. In this case upconversion and quadrature modulation have to be implemented on the FPGA.

3.3.5 Integration in the design flow

Integration of the FPGA

Similar as for the USRP flow, the code generated for the PSM is integrated in the existent environment. Figure 3.18 gives an overview of the FPGA environment in the SFF SDR. The custom logic can be seen as a black box for the individual signal processing functions. Therefore, it provides interfaces to the data converters and to the VPSS buses. Additional interfaces are for registers, interrupts and control strobes.

The transformation from PSM to bitstream is shown in figure 3.19. The functional VHDL code is generated such that the interfaces to the data and control ports are connected automatically. The template makefile includes the entities and connects them with the provided netlists for the data buses and the additional platform specific functions. Furthermore physical properties and connections are given in the makefile that finally scripts the synthesizing and mapping of the code to a bitstream.

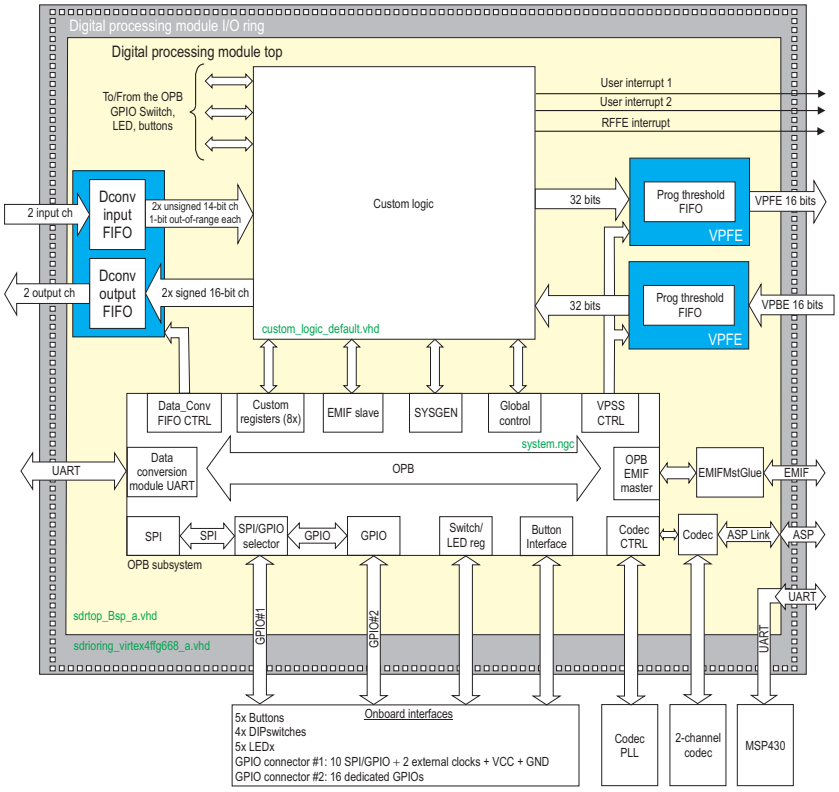


Figure 3.18: Default configuration of the FPGA as in [61]

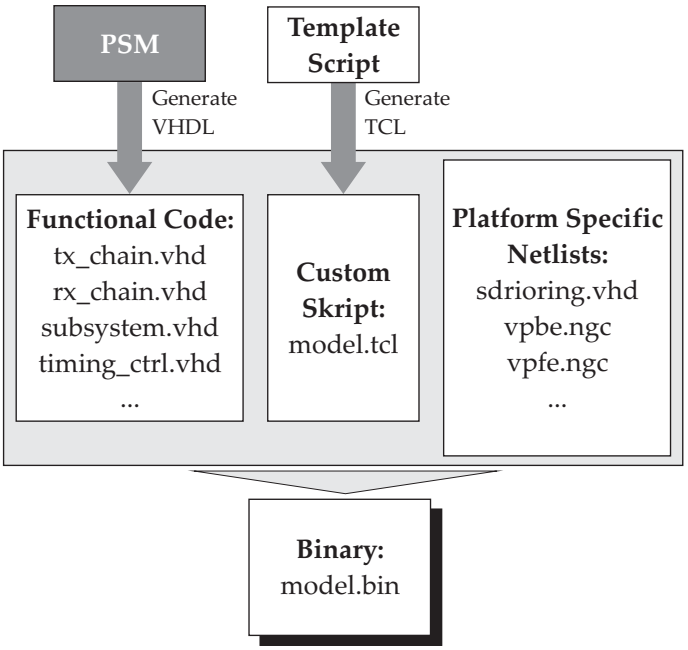


Figure 3.19: Transformation from the PSM to the bitstream on the Virtex4 FPGA

Integration of the DSP

The integration of the DSP in the design flow is done with APIs that connect the model with the data buses. An example of these APIs in the Simulink nevironment is shown in figure 3.20. The interface to the VPSS is implemented as a single block, where in the example of figure 3.20 a block in transmit direction is shown. The VPFE interfaces are linked after code generation and compiling. The other platform specific interfaces are for configuration of the following modules: DSP, DCM and RFM. The configuration of the DSP consists of building properties and arguments for the compiler. In contrast to the data conversion section on the USRP, an own parametrization is needed for the DCM to distribute the clock frequency or to adjust the gain. The configuration of the front end is similar to the USRP: the important parameters are the carrier frequency and the cutting frequencies of the bandpass filters.

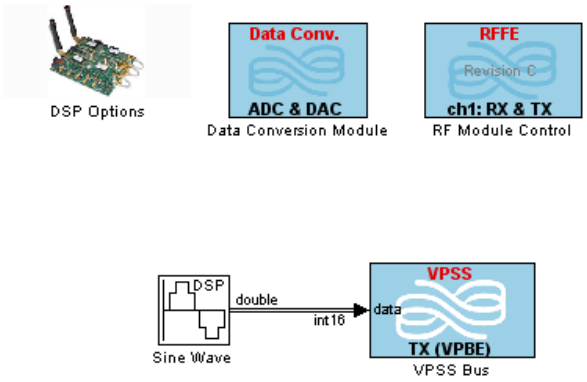


Figure 3.20: Example of the use of the APIs for the integration of the DSP on the SFF SDR DP

Figure 3.21 shows the transformation from PSM to executable code, which is similar to the transformation shown in figure 3.11. In this transformation the C code is generated and linked to the buses and APIs. The resulting code is compiled with the vendor specific development environment (Code Composer Studio) and loaded on the platform.

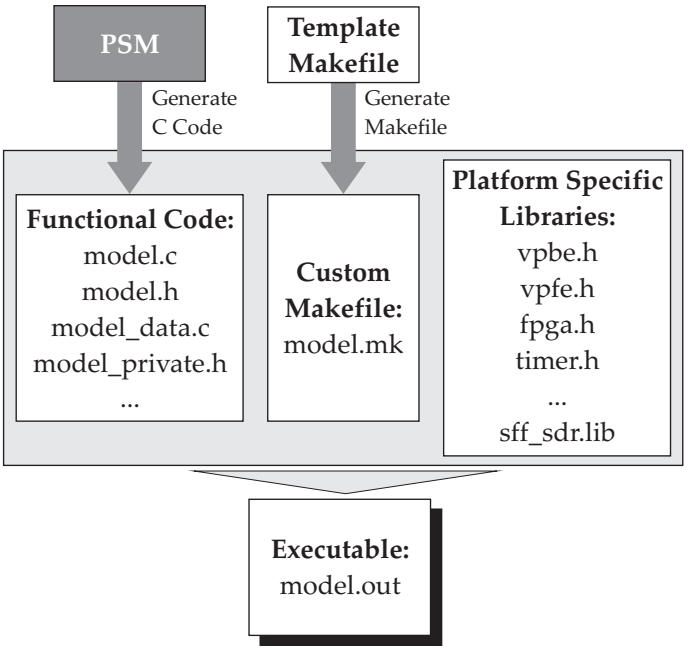


Figure 3.21: Transformation from the PSM to the executable file for the DSP

4 Proof of Concept: Portability of TETRA

4.1 Overview of the TETRA standard

Terrestrial Trunked Radio (TETRA) is an open specification for a digital communication system defined by ETSI in 1995. It is intended as a radio link for public safety agencies like police or fire department and will replace legacy analog radio devices. Therefore, the carrier frequency was specified to be in the already existing VHF and UHF bands for Public Mobile Radio (PMR). With a bandwidth of 25 kHz, two 12.5 kHz wide analog FM channels can be replaced by one channel of the new digital system. Due to the TDMA structure with four time slots per channel, the number of users for each channel can be doubled. Furthermore, new features like multicasting and broadcasting, data transmission and encryption, access to the Internet and a better resistance against interferences are added. Although its first draft was released in 1995, the integration of the system in the current radio communication structure for public mobile radio is still ongoing. The European project WINTSEC proposed SDR as a way for cost efficient introduction of new communication systems by a software based interoperability with the legacy devices.

This chapter describes the software based development of a TETRA waveform under the aspect of portability. Due to the high complexity of the specification, this work focuses on the user plane of the Voice and Data (V+D) air interface protocol stack. Therefore, Physical Layer (PHY) and Media Access Control (MAC) layers are implemented. For completeness, figure 4.1 shows the architecture of the V+D protocol stack for the user plane as well as the higher levels of the control plane. The implemented PHY and MAC layers are highlighted in gray.

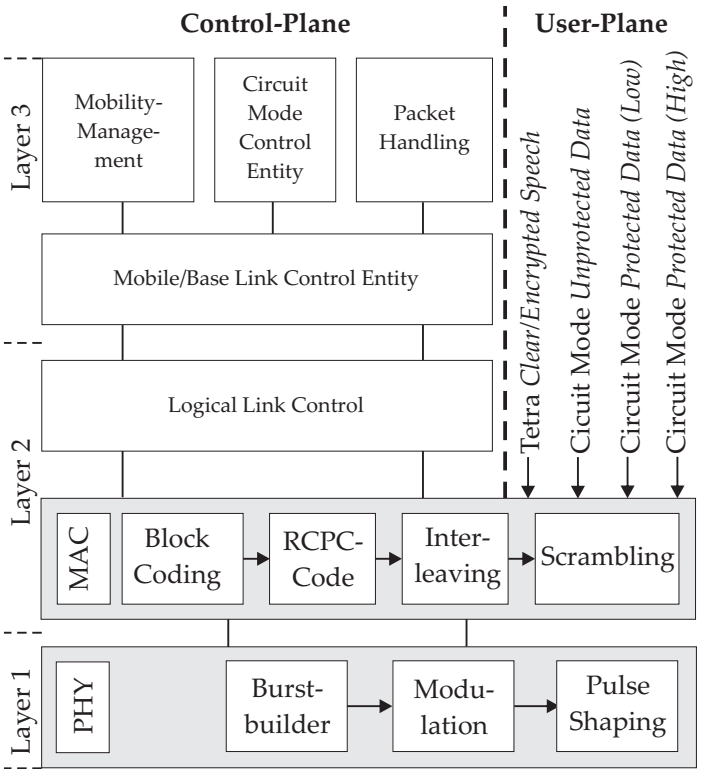


Figure 4.1: Architecture of the V+D protocol stack, based on [71]

The user plane specifies four possible traffic channels for transmitting user data, which are distinguished according to the robustness of channel coding and the data rate as follows:

Traffic CHannel/2.4 (TCH/2.4): Channel for data transmission with high protection. The bit rate is according to the name 2.4 kbit/s

Traffic CHannel/4.8 (TCH/4.8): Channel for data transmission with low protection. The bit rate is according to the name 4.8 kbit/s

Traffic CHannel/7.2 (TCH/7.2): Channel for data transmission without protection. The bit rate is according to the name 7.2 kbit/s

Traffic CHannel/Speech (TCH/S): Channel for speech transmission, due to the bit rate of the underlying voice codec of 4567 bit/s, the TCH/S uses TCH/4.8.

The various traffic channels described above are only logical channels. The mapping from logical to physical channels will be described in the specification of the PHY in section 4.2.2.

4.2 Computational Independent Model

4.2.1 Media Access Control

The MAC layer of TETRA consists of channel coding, interleaving and scrambling according to figure 4.1. These operations are shown in more detail in figure 4.2 with separation in the different traffic channels. For a complete description of the individual coding and interleaving schemes, five planes are introduced that separate the different processing blocks. In this nomenclature $b_x[k]$ is the bit at position k of plane number x. The figure shows furthermore the width of the bit fields, which differs according to the data rate of the different channels. The length of these fields in plane x is named K_x.

To assure that the convolutional encoder ends up in a defined state, the information bits for TCH/2.4 and TCH/4.8 must be extended with four tail bits according to the following clauses:

$$b_2[k] = \begin{cases} b_1[k] & \text{for} \quad 1 \leq k \leq K_1 \\ 0 & \text{for} \quad K_1 < k \leq K_2 \end{cases} \tag{4.1}$$

For the TCH/2.4 the length of the vector is $K_1 = 144$ while for the TCH/4.8 the length is specified to $K_1 = 288$. For these channels a zero padding of four bits is inserted due to the register length of four for the following encoding. Therefore, the length of K_2 can be determined to be $K_2 = K_1 + 4$. Due to the fact that the TCH/7.2 is not protecting its information with channel encoding, neither tail bits nor encoding schemes or interleaving are needed. For the traffic channels TCH/2.4 and TCH/4.8 the convolutional coding is identical. The encoded bits b_3 can be calculated by:

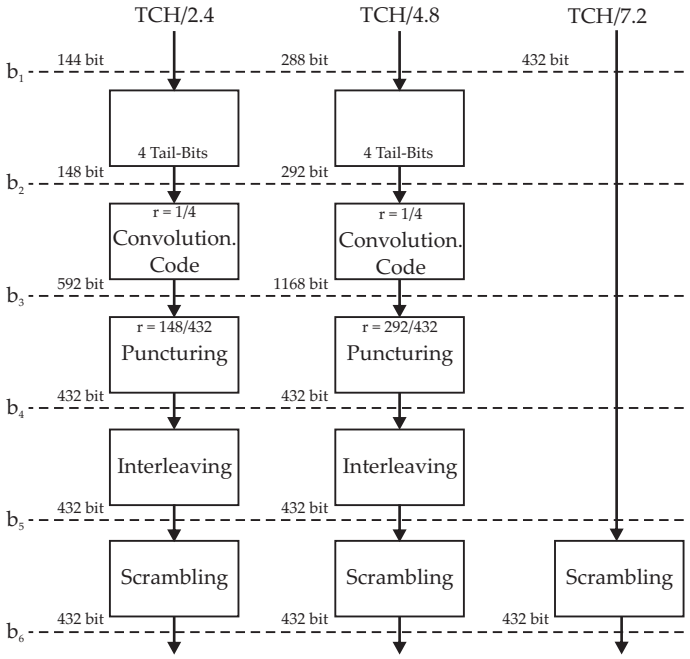


Figure 4.2: Overview of the encoding and scrambling schemes, used in the transmit side of the traffic channel

$$b_3[4(k-1)+i] = \sum_{j=0}^{4} b_2[k-j]g_{i,j} \quad \text{for} \quad \begin{cases} i = 1,2,3,4 \\ k = 1,2,...,K_2\ . \end{cases} \tag{4.2}$$

In this equation, $g_{i,j}$ is the element j at row i of the matrix that can be described by the generator polynomials of the rate $\frac{1}{4}$ mother code:

$$\begin{aligned} G_1(X) &= 1 + X + X^4 &\Rightarrow\quad g_1 &= [11001] \\ G_2(X) &= 1 + X^2 + X^3 + X^4 &\Rightarrow\quad g_2 &= [10111] \\ G_3(X) &= 1 + X + X^2 + X^4 &\Rightarrow\quad g_3 &= [11101] \\ G_4(X) &= 1 + X + X^3 + X^4 &\Rightarrow\quad g_4 &= [11011] \end{aligned}$$

At this point, the traffic channels have different frame lengths. This can also be seen in figure 4.2. The length of the bit field K_3 is 592

for the TCH/2.4, 1168 for the TCH/4.8 and 432 for the TCH/7.2. To achieve the specified frame length of $K_4 = 432$ bit, which is equal for all channels, the redundancy of the encoded channels is reduced with different puncturing schemes. The puncturing can be described as follows:

$$b_4[k] = b_3[j(k)] \tag{4.3}$$

The index j can be calculated dependent on the index k from the output vector by the following equation for the TCH/2.4:

$$j(k) = 8\left(\left\lfloor \frac{k-1+k_{35}}{6} \right\rfloor\right) + P_{2.4}\left(k + k_{35} - 6 \cdot \left\lfloor \frac{k-1+k_{35}}{6} \right\rfloor\right), \tag{4.4}$$

where $\lfloor x \rfloor$ is the floor function that rounds down to the largest integer smaller than x. The variable k_x is a counter that increments for multiples of x. This can be described by

$$k_x = \left\lfloor \frac{k-1}{x} \right\rfloor. \tag{4.5}$$

The mapping of $P_{2.4}$ is depicted in table 4.1. For the TCH/4.8 the index j is calculated with

$$j(k) = 8\left(\left\lfloor \frac{k-1+k_{65}}{3} \right\rfloor\right) + P_{4.8}\left(k + k_{65} - 3 \cdot \left\lfloor \frac{k-1+k_{65}}{3} \right\rfloor\right), \tag{4.6}$$

where the mapping of $P_{4.8}$ is also depicted in table 4.1.

i	$P_{2.4}(i)$	$P_{4.8}(i)$
1	1	1
2	2	2
3	3	5
4	5	
5	6	
6	7	

Table 4.1: Mapping of the puncturing scheme for TCH/2.4 and TCH/4.8

The interleaving mechanism follows the same rule for TCH/2.4 and TCH/4.8 respectively. Therefore the bits are interleaved by

$$b_5[k] = b_4\left[1 + ((103 \cdot k) \bmod (432))\right] \quad \text{for} \quad 1 \leq k \leq 432\,. \tag{4.7}$$

While interleaving is only applied within the convolutional encoded channels, the scrambling mechanism is the same for all traffic channels. It is done by adding a pseudo noise sequence $p[k]$ to the bits:

$$b_6[k] = b_5[k] \oplus p[k] \quad \text{for} \quad 1 \leq k \leq 432 \tag{4.8}$$

The scrambling sequence $p[k]$ is generated by a 32 bit wide linear feedback shift register with a connection polynomial of

$$\begin{aligned} c(x) &= \sum_{i=0}^{32} c_i x^i \\ &= 1 + x + x^2 + x^4 + x^5 + x^7 + x^8 + x^{10} \\ &\quad + x^{11} + x^{12} + x^{16} + x^{22} + x^{23} + x^{26} + x^{32}\,. \end{aligned}$$

Hereby, the k-th bit of the scrambling sequence is given by

$$p[k] = \sum_{i=1}^{32} c_i\, p[k-i]. \tag{4.9}$$

Beside the connection polynomial, the initialization of the register defines the scrambling code. This is done with the EXTENDED COLOUR CODE $e[k]$:

$$p[k] = \begin{cases} e[1-k] & \text{for} \quad -29 \leq k \leq 0 \\ 1 & \text{for} \quad -31 \leq k \leq -30 \end{cases} \tag{4.10}$$

The EXTENDED COLOUR CODE consists of 30 bits and describes the mobile station according to figure 4.3. The first ten bits are defined by the MOBILE COUNTRY CODE which identifies the country where the device is registered. The next 14 bits identify the access net within the country with the MOBILE NETWORK CODE. The last six bits, finally, represent the COLOUR CODE, which identifies the individual mobile [72]. To achieve a register length of 32, two defined bits are added.

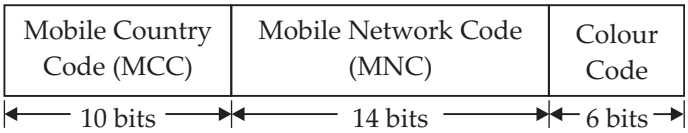


Figure 4.3: Initialization of the scrambling registers with the EXTENDED COLOUR CODE

4.2.2 Physical Layer

Similar to the well known GSM standard, TETRA is specified to use a combination of Time Division Multiple Access (TDMA) and Frequency Division Multiple Access (FDMA) to give multiple users access to the air interface. A TDMA frame in the TETRA system consists of four time slots, for supporting up to four users on a single frequency of 25 kHz bandwidth. Uplink and downlink are separated in time and frequency where the various European regulatory organizations agreed on a distance of 10 MHz between uplink and downlink carrier. To ease the requirements for a mobile station, uplink and downlink are not only separated in frequency, they are further shifted by two slots in time. Figure 4.4 shows the TDMA/FDMA structure with the combination of Frequency Division Duplex (FDD) and Time Division Duplex (TDD). The gray boxes show an example of a physical channel, defined with a pair of frequencies (for uplink and downlink carriers) and a corresponding time slot number [73].

In every occupied time slot one burst is transmitted. TETRA specifies four bursts for the downlink and another three bursts for the uplink. With these bursts, control information, synchronization mechanisms and user data can be transmitted. Due to the fact that this work is focused on the user plane, only the bursts containing user data or synchronization information are considered. A NORMAL UPLINK BURST or a NORMAL DOWNLINK BURST consists of 510 bits and maps the logical traffic channels to the physical channels by extending the coded bits with training sequences, guard intervals and control information. In addition, special synchronization bursts ensure that the mobile station can be synchronized to its frequency and frame structure. Examples of uplink and downlink bursts, specified in the TETRA standard and considered in the CIM, are shown in figure 4.5. To achieve a data transmission, the

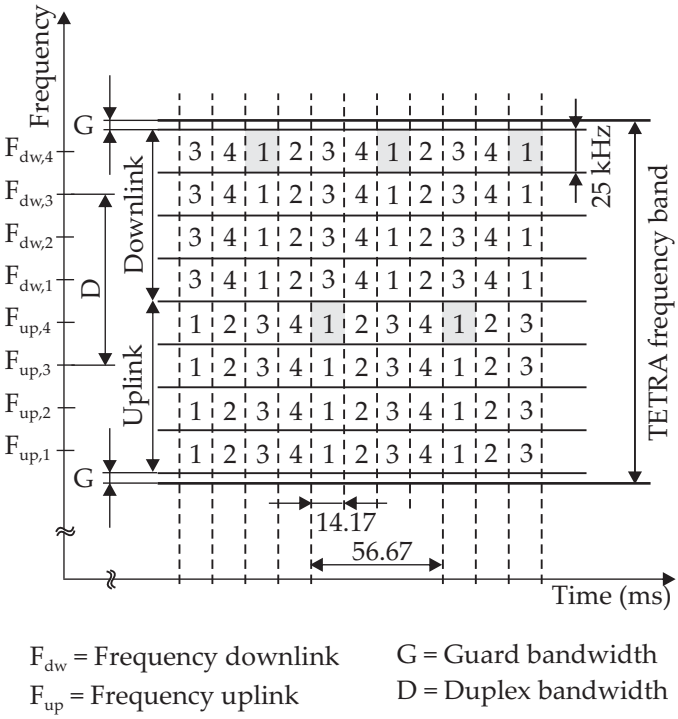


Figure 4.4: Overview of the FDMA/TDMA structure

NORMAL DOWNLINK BURST is transmitted. The two blocks with encoded data comprise of the 432 bit long traffic channel, which can be encoded with the error schemes described in the previous section. The broadcast bits, indicated as BBK in figure 4.5, are control information from the Access Assignment CHannel (AACH). This is a channel that controls the time slot occupation in uplink and downlink. As mentioned previously, this work focuses on the user plane. However, for the sake of completeness the AACH is part of the NORMAL DOWNLINK BURST and has therefore to be implemented. The information for the AACH consists of 14 bits that are encoded with a shortened Reed Muller code, leading to a coded bit length of 30. The generator matrix can be found in [73]. The coded bits are scrambled with the scheme described above and form the BBK bits in the NORMAL DOWNLINK BURST.

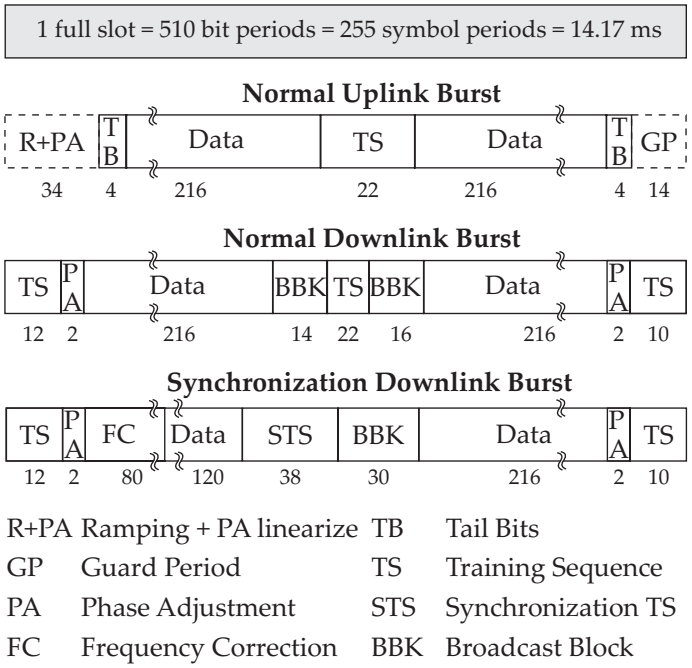


Figure 4.5: TETRA V+D uplink and downlink burst types

The bits of the bursts are modulated with a Differential Quadrature Phase-Shift Keying (DQPSK) scheme with a phase offset of $\pi/4$. Due to the fact that a phase rotation must occur between two adjacent symbols, the current symbol $s[k]$ can be calculated dependent on the previous one by

$$s[k] = s[k-1]e^{j\phi[k]} \quad \text{with} \quad s[0] = 1\,. \tag{4.11}$$

The phase offset $\phi[k]$ depends on the current symbol and can be determined using table 4.2. With these values, no phase shift of $\pm\pi$ occurs, which reduces fluctuations in the magnitude of the complex envelope. This eases the linearization requirements of power amplifiers, especially in mobile devices where power amplifier should be as cheap as possible. The signal trajectory of this modulation scheme is shown on the left hand

side in figure 4.6. Another advantage of this modulation scheme is that it allows non-coherent demodulation at the receiver, due to the fact that information is not transmitted with the current phase but with the phase offset. Even if the Signal-to-Noise Ratio (SNR) has to be doubled to achieve the same bit error rate as QPSK, the ease of the synchronization compensates this loss.

Table 4.2: Phase offset depending on the actual symbol

Bit 2	Bit 1	Symbol	ϕ[k]
1	1	3	- $\frac{3\pi}{4}$
0	1	1	+ $\frac{3\pi}{4}$
0	0	0	- $\frac{\pi}{4}$
1	0	2	+ $\frac{\pi}{4}$

To generate the transmit signal $s_{\text{tx}}(t)$, the discrete time symbols $s[k]$ are filtered with a time continuous pulse shaping filter $g_{\text{RRC}}(t)$. This can be described as follows:

$$s_{\text{tx}}(t) = \sum_{k=0}^{K} s[k] \cdot g_{\text{RRC}}(t - kT_{\text{sym}}), \tag{4.12}$$

where T_{sym} is the symbol duration of 55.56 µs and K is the maximum number of symbols. The impulse response of the pulse shaping filter $g_{\text{RRC}}(t)$ is obtained by the inverse Fourier transform of a square root raised cosine spectrum $G_{\text{RRC}}(f)$ with roll-off factor α defined as

$$G_{\text{RRC}}(f) = \begin{cases} 1, & |f| \leq \frac{1-\alpha}{2T_{\text{sym}}} \\ \cos\left(\frac{\pi T_{\text{sym}}}{2\alpha}\left(|f| - \frac{1-\alpha}{2T_{\text{sym}}}\right)\right), & \frac{1-\alpha}{2T_{\text{sym}}} < |f| \leq \frac{1+\alpha}{2T_{\text{sym}}} \\ 0, & |f| > \frac{1+\alpha}{2T_{\text{sym}}}\ . \end{cases} \tag{4.13}$$

TETRA specifies $\alpha = 0.35$ and offers the possibility to implement a time limited version of $g_{\text{RRC}}(t)$ that fulfills the constraints of modulation accuracy and adjacent channel attenuation. Therefore, the sequence of complex symbols $s[k]$ must be interpolated by a factor of I_{RRC} and furthermore filtered with a discrete time realization of the root raised cosine filter. This is identical to the sampling of the time continuous transmit

function $s_{\text{tx}}(t)$ with sampling rate $I_{\text{RRC}}/T_{\text{sym}}$. Hence, the discrete time signal of the transmit sequence can be described as

$$s_{\text{tx}}\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}}\right) = \sum_{k=0}^{K} s[k] g_{\text{RRC}}\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}} - kT_{\text{sym}}\right). \tag{4.14}$$

The design of the filter $g_{\text{RRC}}(\cdot)$ is platform specific due to the sample rate conversion. Therefore it is part of the PSM. However, it has to be mentioned that the pulse shaping filter results in larger fluctuations of the complex envelope and hence a smaller hole of the signal trajectory in the complex plane. This effect is shown on the right hand side of figure 4.6.

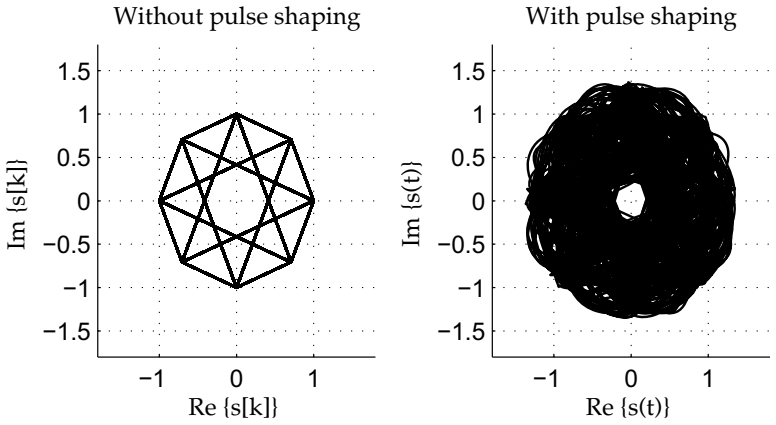


Figure 4.6: Signal trajectory of the $\frac{\pi}{4}$-DQPSK modulation scheme

Table 4.3 summarizes the key parameters of the TETRA system.

4.3 Platform Independent Model

The Platform Independent Model of TETRA is the implementation of the transmit path as described in the CIM in the previous section 4.2.

Table 4.3: Overview of the TETRA system parameter

Parameter	Value
Carrier frequency	400 MHz
Bandwidth	25 kHz
Media access	TDMA/FDMA
Duplex mode	TDD/FDD
Users per carrier	4
Modulation	$\frac{\pi}{4}$-DQPSK
Channel coding	RCPC
Symbol rate	18 kBaud/s
Bits per slot	510
Frame duration	56.67 ms
Bursts per frame	4
Burst duration	14.167 ms
Pulse shaping	RRC with 0.35 rolloff
Data rate	up to 28.8 kbit/s

Furthermore, the receive side is implemented with the synchronization of time, frequency and frame, which is not described in the CIM.

4.3.1 Transmitter

The input of the transmitter is a vector with random bits generated by a Pseudo Noise (PN) sequence and is used as information data. The length of this array depends on the traffic channel and can vary from 144 bits for the TCH/2.4, over 288 bits for the TCH/4.8 to 432 bits for the TCH/7.2. The encoding depends also on the traffic channel and assures an output length of 432 bits, independent of the channel's data rate. The scrambling with the EXTENDED COLOUR CODE is similar for all channels. It is assumed that the initialization of the register is already known at transmit and receive side.

Another input for the transmitter is the 14 bit wide vector for the control channel (AACH). It has to be mentioned that this channel also transmits random bits. Therefore, no control or configuration is applied from the information of the broadcast channel. The control channel is encoded with a shortened Reed Muller code and scrambled with the same pseudo

noise sequence as the traffic channel. The burst builder shown in figure 4.7 integrates the encoded data fields in a NORMAL DOWNLINK BURST as defined by the TETRA specification and adds a SYNCHRONIZATION DOWNLINK BURST after 18 regular bursts. This synchronization burst is needed for the frequency synchronization in the receiver. The frame size for the implementation is according to the TETRA burst length set to 510. These bits are modulated with the described $\frac{\pi}{4}$-DQPSK modulation scheme to 255 complex symbols. The pulse shaping filter that must fulfill the root raised cosine spectrum finalizes the transmit side.

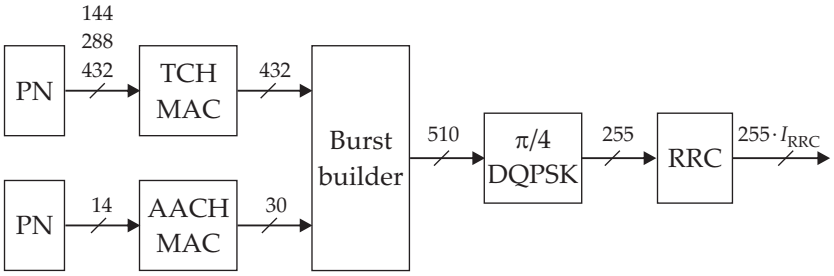


Figure 4.7: PIM of the TETRA transmit path

The realization of the Root Raised Cosine (RRC) transmit filter as an FIR filter can be determined by two parameters: the interpolation factor I_{RRC} and the group delay D_{RRC}. The third parameter, the roll-off factor is already specified in the standard to $\alpha = 0.35$. According to the symbol rate of 18 kBaud/s the symbol time can be determined to $T_{\mathrm{sym}} = 55.56\,\mu\mathrm{s}$. The impulse response of an RRC filter is according to [74]

$$g_{\mathrm{RRC}}(t) = 4\alpha \frac{\cos\left((1+\alpha)\pi\frac{t}{T_{\mathrm{sym}}}\right) + \frac{\sin\left((1-\alpha)\pi\frac{t}{T_{\mathrm{sym}}}\right)}{4\alpha\frac{t}{T_{\mathrm{sym}}}}}{\pi\sqrt{T_{\mathrm{sym}}}\left(1-\left(4\alpha\frac{t}{T_{\mathrm{sym}}}\right)^2\right)}\,. \tag{4.15}$$

The discrete time realization of this filter response can be described with

the interpolation factor I_{RRC} as follows:

$$g_{\mathrm{RRC}}\left(i \cdot \frac{T_{\mathrm{sym}}}{I_{\mathrm{RRC}}}\right) = 4\alpha \frac{\cos\left((1+\alpha)\pi \frac{i}{I_{\mathrm{RRC}}}\right) + \frac{\sin\left((1-\alpha)\pi \frac{i}{I_{\mathrm{RRC}}}\right)}{4\alpha \frac{i}{I_{\mathrm{RRC}}}}}{\pi\sqrt{T_{\mathrm{sym}}}\left(1 - \left(4\alpha \frac{i}{I_{\mathrm{RRC}}}\right)^2\right)} \tag{4.16}$$

Due to the fact that this filter is defined for infinite values of i, a time limited version of $g_{\mathrm{RRC}}(\cdot)$ must be applied and the impulse response must be delayed to achieve a causal filter. The group delay D_{RRC} describes the number of symbol durations the impulse response must be shifted and is therefore the implicit parameter of the rectangular window function $w[i]$ with

$$w[i] = \begin{cases} 1 & |i| \leq D_{\mathrm{RRC}} \cdot I_{\mathrm{RRC}} \\ 0 & |i| > D_{\mathrm{RRC}} \cdot I_{\mathrm{RRC}}. \end{cases} \tag{4.17}$$

The time limited realization of the filter can be described as follows:

$$g_{\mathrm{RRC,w}}\left(i \cdot \frac{T_{\mathrm{sym}}}{I_{\mathrm{RRC}}}\right) = g_{\mathrm{RRC}}\left(i \cdot \frac{T_{\mathrm{sym}}}{I_{\mathrm{RRC}}}\right) \cdot w[i] \tag{4.18}$$

With the multiplication of the window function, the filter length can be determined to $2D_{\mathrm{RRC}}I_{\mathrm{RRC}} + 1$ and is therefore proportional to the group delay. The performance loss of this filter due to the windowing can be evaluated by the relative energy loss that can be calculated as follows:

$$E_{\mathrm{rel}} = \frac{E_{\mathrm{RRC}} - E_{\mathrm{RRC,w}}}{E_{\mathrm{RRC}}}, \tag{4.19}$$

where the energy of an RRC impulse response can be easily calculated with the impulse response of a raised cosine filter $g_{\mathrm{RC}}[i]$:

$$\begin{aligned} E_{\mathrm{RRC}} &= g^*_{\mathrm{RRC}}[-i] * g_{\mathrm{RRC}}[i] \quad |_{i=0} && (4.20) \\ &= \frac{I_{\mathrm{RRC}}}{T_{\mathrm{sym}}} \cdot g_{\mathrm{RC}}[i] \quad |_{i=0} && (4.21) \\ &= \frac{I_{\mathrm{RRC}}}{T_{\mathrm{sym}}} && (4.22) \end{aligned}$$

The relation of the energies can be determined to:

$$E_{\text{rel}} = 1 - \sum_{i=-D_{\text{RRC}} \cdot I_{\text{RRC}}}^{D_{\text{RRC}} \cdot I_{\text{RRC}}} \left| 4\alpha \frac{\cos\left((1+\alpha)\pi \frac{i}{I_{\text{RRC}}}\right) + \frac{\sin\left((1-\alpha)\pi \frac{i}{I_{\text{RRC}}}\right)}{4\alpha \frac{i}{I_{\text{RRC}}}}}{\pi\sqrt{T_{\text{sym}}}\left(1 - \left(4\alpha \frac{i}{I_{\text{RRC}}}\right)^2\right)} \right|^2 \tag{4.23}$$

Figure 4.8 shows the relative energy loss E_{rel} in relation to the group delay D_{RRC}. To achieve an energy loss less than 0.01 % a group delay of six was chosen in combination with an interpolation factor of $I_{\text{RRC}} = 64$. The chosen parameters result in 769 taps for one filter and the signal processing period for one frame is about 1.5 ms on an Intel P8400 processor with a clock frequency of 2.28 GHz. Even if this filter length is not feasible for a real time implementation, the combination between filter accuracy and simulation time is acceptable.

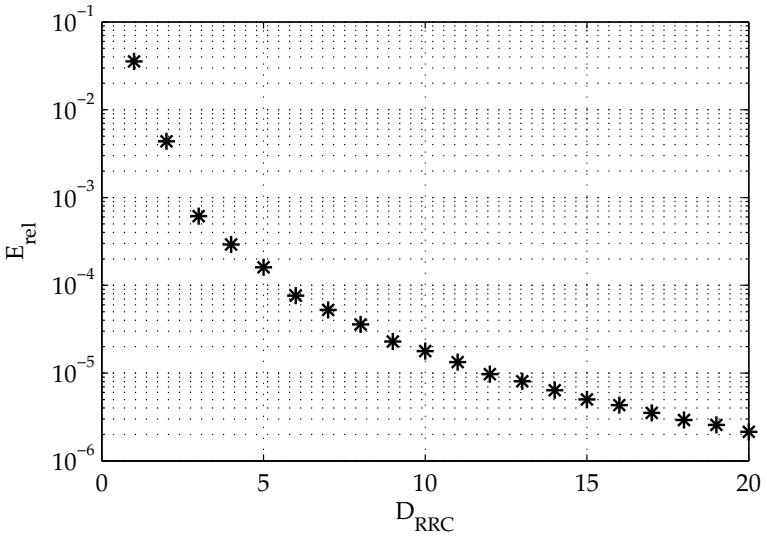


Figure 4.8: Relative energy loss of a windowed realization of an RRC in dependence of the group delay D_{RRC}

4.3.2 Channel

For the evaluation of the synchronization algorithms, a channel must be simulated. This channel behaves like a virtual front end and adds a white Gaussian noise signal $n(t)$. The input of the channel consists of a vector comprising the interpolated and filtered symbols $s_{\text{tx}}(t)$ with the TETRA burst length multiplied with the interpolation factor I_{RRC}. To introduce channel delay, a fixed latency of N symbols is applied. Hence, a vector comprises two partial TETRA bursts and the receiver has to detect the beginning of a burst. To simulate the different clock frequencies of transmitter and receiver, another delay ϵ is inserted. In contrast to the first delay, the time shift is not fixed and varies over time. With these operations, the channel can be described as follows:

$$r_{\text{rx}}(t) = s_{\text{tx}}\left(t - NT_{\text{sym}} - \epsilon(t)\right) + n(t) \tag{4.24}$$

Due to the discrete time realization of the PIM, the received signal is given by

$$r_{\text{rx}}\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}}\right) = s_{\text{tx}}\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}} - NT_{\text{sym}} - \epsilon\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}}\right)\right) + n\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}}\right). \tag{4.25}$$

Another effect of the asynchronous local oscillators on the transmitter and receiver is the offset of the carrier frequency ν and the phase offset φ. These RF impairments are simulated by the multiplication with a complex harmonic wave. Under the assumption that no time and symbol offset occurs, the receive signal $r_{\text{rx}}(\cdot)$ can be determined by

$$r_{\text{rx}}\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}}\right) = s_{\text{tx}}\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}}\right) \cdot e^{j2\pi\left(\nu \cdot i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}} + \varphi\right)} + n\left(i \cdot \frac{T_{\text{sym}}}{I_{\text{RRC}}}\right). \tag{4.26}$$

A more detailed description of this channel as a virtual front end can be found in [75].

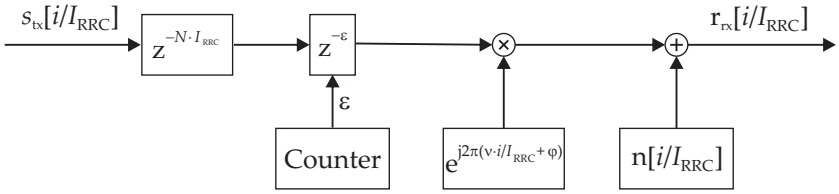


Figure 4.9: Structure of the channel, which works as a virtual front end

4.3.3 Time Synchronization in TETRA

To minimize the influence of the additive white Gaussian noise $n(\cdot)$, the received signal $r_{\mathrm{rx}}(\cdot)$ is filtered with the root raised cosine filter, which is also the matched filter to

$$r\left(i \cdot \frac{T_{\mathrm{sym}}}{I_{\mathrm{RRC}}}\right) = r_{\mathrm{rx}}\left(i \cdot \frac{T_{\mathrm{sym}}}{I_{\mathrm{RRC}}}\right) * g_{\mathrm{RRC}}\left(i \cdot \frac{T_{\mathrm{sym}}}{I_{\mathrm{RRC}}}\right). \tag{4.27}$$

Prior to downsampling, the timing error ϵ must be determined and corrected. This is done by a square timing error detector as proposed by Oerder in [76]. By calculating the square of the absolute value of the received and filtered signal $r(\cdot)$, there are linear distortions leading to spectral peaks at multiples of the system's symbol rate, which include the delay information. The squared receive signal $x[i]$ can be expressed by

$$x[i] = \left| r\left(i \cdot \frac{T_{\mathrm{sym}}}{I_{\mathrm{RRC}}}\right) \right|^2. \tag{4.28}$$

Figure 4.10 shows the spectrum of the signal $x[i]$. The mentioned spectral lines at multiples of 18 kHz indicate the symbol rate. The time delay in the symbol rate is now transformed into a phase shift. Therefore, the delay can be determined by calculating the phase rotation in the spectral domain at the frequency of the symbol rate. By assuming that the timing error is constant over L symbols, the Fourier transform can be applied over $L \cdot I_{\mathrm{RRC}}$ samples. Due to the fact that only the Fourier coefficient at the symbol rate is needed for the timing error, it can be calculated as follows:

$$X[m] = \sum_{i=mLI_{\mathrm{RRC}}}^{(m+1)LI_{\mathrm{RRC}}-1} x[i]e^{-j2\pi i/I_{\mathrm{RRC}}} \tag{4.29}$$

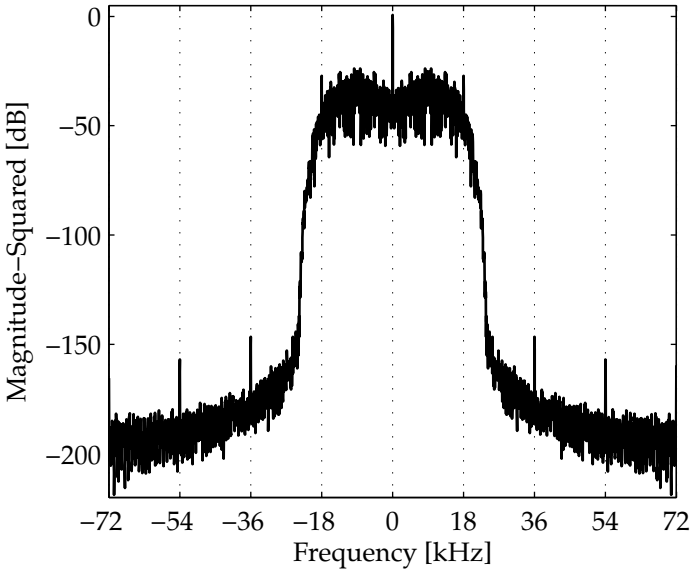


Figure 4.10: Spectrum of the squared magnitude of the receive signal

Therefore, the timing error ϵ can be determined by

$$\epsilon[m] = -\frac{1}{2\pi} \arg\{X[m]\} \ . \tag{4.30}$$

It has to be mentioned that the choices of several parameters such as the interpolation factor I_{RRC} or the assumption that the timing error is constant over L symbols, depends on the underlying platform. Therefore, these parameters will be discussed in the PSM.

4.3.4 Frequency Synchronization

The frequency offset is determined by using the 19 symbol wide SYNCHRONIZATION TRAINING SEQUENCE and the 40 symbol wide FREQUENCY CORRECTION FIELD as shown in figure 4.5. According to the specification, the SYNCHRONIZATION DOWNLINK BURST is sent after 18 NORMAL DOWNLINK BURSTS. To detect the SYNCHRONIZATION DOWNLINK BURST the incoming symbols are correlated with the following two training sequences: the FREQUENCY CORRECTION FIELD and the SYNCHRONIZATION TRAINING SEQUENCE. However, by assuming a frequency offset, the peak of the correlation would disappear in the noise. A possibility to circumvent this is to build the product of the incoming symbol with the conjugate complex of the previous symbol. This transforms the frequency offset to a phase offset. The signal is then correlated with the training sequences and the peak remains detectable for frequency offsets as shown in figure 4.11.

The position of the peak leads to the position of the FREQUENCY CORRECTION FIELD, which consists of the following bits:

$$FC = [\underbrace{1,1,...,1,1}_{8},\underbrace{0,0,...,0,0}_{64},\underbrace{1,1,...,1,1}_{8}]$$

The zero bits in the middle of the field lead to a phase offset of $\pi/4$ between two symbols. With this information, the frequency offset can be evaluated by

$$r(k \cdot T_{\text{sym}}) \stackrel{!}{=} r((k-1) \cdot T_{\text{sym}}) e^{j\pi/4} \quad \text{for} \quad k = FC_5 \ldots FC_{36} \,. \tag{4.31}$$

In this equation, FC_x represents the position of symbol number x in the FREQUENCY CORRECTION FIELD inside the symbol stream. With the following equation:

$$\hat{v} = \frac{1}{2\pi} \cdot \left(\frac{1}{32} \sum_{k=\text{FC}_5}^{\text{FC}_{36}} \arg\left\{ r(k \cdot T_{\text{sym}}) \right\} - \arg\left\{ r((k-1) \cdot T_{\text{sym}}) \right\} - \frac{\pi}{4} \right), \tag{4.32}$$

a frequency offset in the range of -9 kHz to 9 kHz can be evaluated. Higher frequency offsets lead to errors due to phase ambiguities. Therefore, frequency offset acquisition algorithms have to be implemented. However, this depends on the platform specific offset and is therefore not taken into account in the PIM. Figure 4.11 shows the correlation with the known sequences for finding the FREQUENCY CORRECTION FIELD. The input signal has a signal to noise ratio of SNR = 20 dB and a frequency offset of ν = 1 kHz.

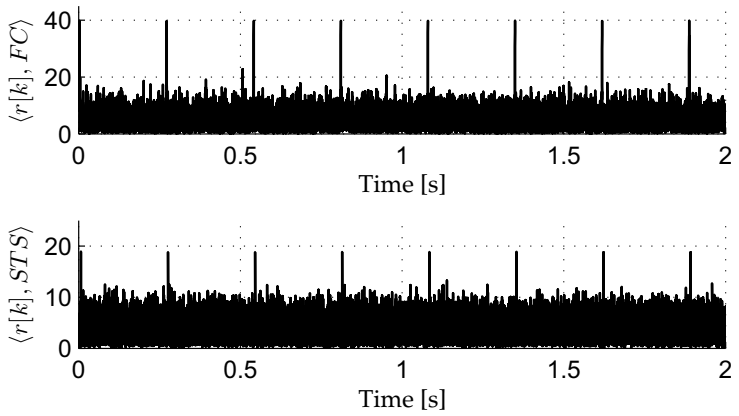


Figure 4.11: Correlation of the receive signal $r[k]$ with the synchronization sequences: SYNCHRONIZATION TRAINING SEQUENCE (STS) and FREQUENCY CORRECTION FIELD (FC)

The frequency offset signal is an error compensation signal in the demodulation. Due to the incoherent demodulation of the signal, the angle of the actual signal is subtracted from the angle of the last symbol. This means that the frequency offset can be regarded as an angle offset between two symbols, leading to another subtraction of the phase offset. With this method, a complex multiplication can be replaced by a simple real addition.

A phase synchronization is not necessary. The differential modulation scheme includes the information on the phase difference between two adjacent symbols. Therefore, no knowledge of the absolute phase position is needed.

4.3.5 Frame Synchronization

The output of the demodulator is a bit stream without information about the embedded bursts. Therefore, the frame synchronization extracts the traffic and control channels from the bit stream. To find the logical channels inside the stream, every burst includes training sequences as shown in figure 4.5. When receiving data from the NORMAL CONTINUOUS DOWNLINK BURST, the frame synchronization searches the bit stream for the TRAINING SEQUENCE 1 (TS1) and TRAINING SEQUENCE 3 (TS3). These fields indicate the start and stop of the traffic and the control channel. The output is a 432 bit long vector, comprising the Traffic CHannel (TCH) and a 30 bit long vector with the AACH. The receiver for the physical layer is shown in figure 4.12 with a matched filter, time synchronization, frequency synchronization and frame synchronization.

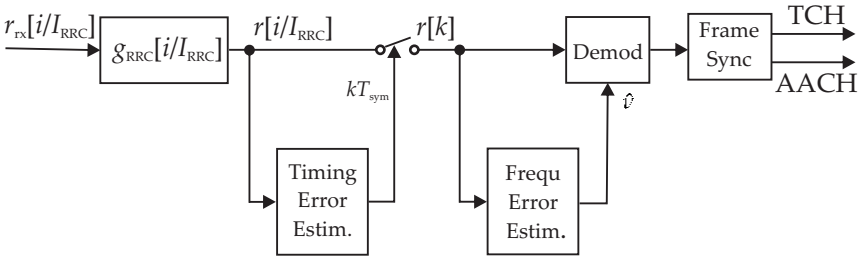


Figure 4.12: Overview of the PIM of the receiving PHY

4.3.6 MAC receiver

The input of the MAC receiver comprises two vectors: the 432 bit long TCH and the 30 bit long AACH. Since both channels are encoded differently, two receivers have to be used that work in parallel. The first operation in the TCH receiver is descrambling. Due to the fact that an additive scrambler was used in the transmit path, the descrambling can be achieved by a second addition with the pseudo noise sequence $p[k]$. To generate the same sequence as in the transmitter, the initialization of the register with the EXTENDED COLOUR CODE must be known at the

receiver. The descrambling can be described by

$$\hat{b}_5[k] = \hat{b}_6[k] + p[k] \quad \text{for} \quad 1 \leq k \leq 432\,, \tag{4.33}$$

where $\hat{b}_x[k]$ represents the approximated bit at position k of the plane x as indicated in the transmit side by figure 4.2.

In case of the TCH/7.2, the descrambled bits are the information bits with

$$\hat{b}_1[k] = \hat{b}_5[k]\,. \tag{4.34}$$

No decoding is applied, however, the bits of the other traffic channels must be descrambled. According to the interleaving scheme, which was described in equation (4.7), the inverse element of 103 has to be determined. It can be shown that 151 is the inverse element to 103 when holding the condition

$$(151 \cdot 103) \mod 432 = 1\,. \tag{4.35}$$

The de-interleaving can be applied as follows:

$$\hat{b}_4[k] = \hat{b}_5\left[1 + (151\,(k-1) - 1) \bmod (432)\right] \quad \text{for} \quad 1 \leq k \leq 432 \tag{4.36}$$

The time shifts are necessary because the range of values have to be mapped from the modulo 432 interval $[0, 431]$ to the interval $[1, 432]$. The now de-interleaved data are decoded, according to the dedicated traffic channel. The lengths of 592 for the TCH/2.4 and 1168 for the TCH/4.8 can be achieved by inserting random bits at the positions where information data were dropped, according to the scheme described in equation (4.3) and table 4.1. Due to the fact that there is no information about the inserted data, values of 0.5 are inserted.

The decoding is achieved with a Viterbi algorithm. Since the PIM is not a real-time implementation, the traceback length of the Viterbi decoder can be set to the frame length, leading to the best error correction performance. However, due to the applied scrambling of the data, the Viterbi decoder works in a hard decision mode. The only bits with reliability values are the punctured bits, which are included with values of 0.5. An advantage of the hard decision decoding is the reduced memory usage. This yields in a loss of approximately 2.5 dB in the SNR compared to soft decision decoding [77].

The 30 bit wide data field of the AACH is descrambled with the same sequence used for the traffic channels. The decoding is achieved with the parity check matrix, which is used to calculate the syndrome vector. Since the syndrome depends only on the error vector and not on the code word, it can be mapped to an error pattern. However, due to the fact that there are 2^{16} different syndromes, only pattern errors with one bit error are searched. This eases the implementation and is a valid expectation. The probability that more than one bit is wrong in the 30 bit long AACH receive vector can be calculated with the bit error rate of the uncoded data P_b to

$$P_{\text{AACH}} = 1 - \left((1 - P_b)^{30} + 30\,(1 - P_b)^{29} \cdot P_b \right) . \tag{4.37}$$

Assuming a bit error rate of $P_b = 0.01$ on the channel, the probability that the AACH would be decoded incorrectly is 3.6 %.

4.4 Platform Specific Model for the USRP

4.4.1 Separation on different processing elements

As described in section 3.2, the platform consists of an FPGA on the USRP and a GPP. Due to the fact that the FPGA is relatively small and without hardware multipliers, most of the signal processing should be located on the GPP. The sample rate conversion from symbol rate to DAC rate and vice versa should be done in the FPGA. Another limitation of the FPGA is the fixed clock frequency of 64 MHz, leading to sampling rates that are only integer divisions of this rate. In the following the resampling for the transmit side is described. As introduced in section 4.3, the symbol time interval is denoted by T_{sym}. The sampling rate of the DAC is given by $f_s = 1/T_s$. The TETRA symbol rate is according to table 4.3 $f_{\text{sym}} = 18\,\text{kHz}$. The input rate of the DACs is $f_s = 32\,\text{MHz}$. With these data, a resampling factor R can be determined with the following factors of interpolation and decimation:

$$R = \frac{f_s}{f_{\text{sym}}} = \frac{32\,\text{MHz}}{18\,\text{kHz}} = \frac{2^7 \cdot 5^3}{3^2} \tag{4.38}$$

Due to the architecture of the platform, the FPGA is only able to work with an upsampling factor of integer numbers. This leads to a resampling over two stages: the first resampling stage is realized in the GPP, while the second stage is done in the FPGA with an interpolation to achieve the DAC rate of 32 MHz. This yields the factors

$$R = \frac{I_{\mathrm{GPP}}}{D_{\mathrm{GPP}}} \cdot I_{\mathrm{FPGA}} = \frac{I_{\mathrm{FIR}}}{D_{\mathrm{FIR}}} \cdot I_{\mathrm{CIC}} \,. \tag{4.39}$$

The FPGA provides no multipliers and should perform a high sampling rate conversion. Therefore, a CIC interpolation filter as described in [78] is needed. CIC filters are a class of digital filters that achieve decimation and interpolation without any multipliers. The hardware efficient implementation is furthermore favored by the highly symmetrical structure, comprising cascaded integrator and comb filter pairs [79]. However, CIC filters have two major disadvantages. The first is that the internal word width increases with the number of stages. However, due to the fact that the FPGA is able to work on data words with arbitrary lengths, this is not a problem in this realization. The second disadvantage is the passband of the filter, which is not flat. To circumvent this problem, an FIR filter has to be designed on the GPP that compensates the CIC frequency response in the desired band. The design of these filters is described in the next section.

Figure 4.13 shows the separation of the resampling on the USRP. The remaining part of the transmitter on the GPP is not shown. The resampling on the receive side has two differences in comparison to the concept of the transmit side. The decimation factors on the receiver are represented with the letter I, while the interpolation factors are described with the letter D. This is due to the fact that the same interpolation and decimation rates are used on the transmit and receive side. Another difference is the missing decimation at the last block of the resampling, since the decimation has to be triggered by the timing error detector. Therefore, it is not shown in this figure.

4.4.2 Resampling

The resampling on the transmit side is done in two stages, one interpolation on the FPGA and one resampling on the GPP. The following

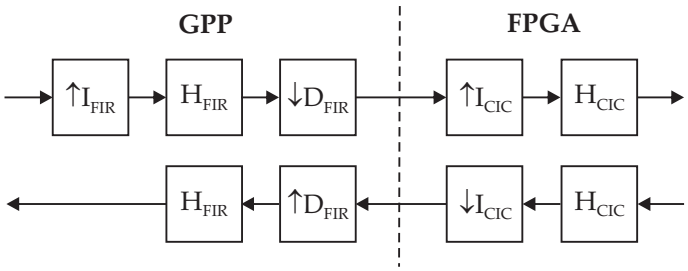


Figure 4.13: Overview of the resampling on the USRP

objectives have to be achieved:

- The sampling rate must be converted from f_{sym} to f_s.
- The pulse shaping of the transmitter must be equal to the frequency response of a root raised cosine filter.
- The conditions for the TETRA spectrum mask must be maintained.

As indicated in the previous section, the interpolation filter on the FPGA can be realized as a CIC filter with a frequency response as follows:

$$G_{\text{CIC}}(f) = \left[\frac{\sin\left(\pi f T_{\text{sym}} \frac{D_{\text{FIR}}}{I_{\text{CIC}}}\right)}{\sin\left(\pi f T_{\text{sym}} \frac{D_{\text{FIR}}}{I_{\text{FIR}} \cdot I_{\text{CIC}}}\right)} \right]^{N_{\text{CIC}}} \tag{4.40}$$

The parameters for the resampling are already given, therefore the number of stages N_{CIC} remains as the only parameter to configure this filter. To ensure that the overall frequency response of the system is equal to a root raised cosine, a CIC compensation filter has to be implemented on the GPP. The frequency response for this filter G_{FIR} on the GPP can be calculated by

$$G_{\text{FIR}}(f) = \frac{G_{\text{RRC}}(f)}{G_{\text{CIC}}(f)}\,, \tag{4.41}$$

where $G_{\text{RRC}}(f)$ is the frequency response of an RRC filter as described in (4.13) and $G_{\text{CIC}}(f)$ is the frequency responses for the CIC filter as described in (4.40).

A root raised cosine filter is also a low pass filter where the pass band depends on the roll-off factor α. Therefore, the relation between interpolation and decimation on the FIR filters should fulfill the following condition:

$$\frac{I_{FIR}}{D_{FIR}} \geq 1 + \alpha \tag{4.42}$$

The filters are realized digitally on reconfigurable logic or processors. From the discrete time impulse response, the frequency response can be calculated for the CIC filter as follows:

$$H_{\mathrm{CIC}}(f) = \sum_{n=-\infty}^{\infty} G_{\mathrm{RRC}}\left(f - n\frac{I_{\mathrm{FIR}} \cdot I_{\mathrm{CIC}}}{D_{\mathrm{FIR}} \cdot T_{\mathrm{sym}}}\right) \tag{4.43}$$

The frequency response of the discrete time filter $H_{\mathrm{FIR}}(f)$ is given by

$$H_{\mathrm{FIR}}(f) = \sum_{n=-\infty}^{\infty} G_{\mathrm{FIR}}\left(f - n\frac{I_{\mathrm{FIR}}}{D_{\mathrm{FIR}}}T_{\mathrm{sym}}\right)\mathrm{si}\left(\pi f(2N_{\mathrm{FIR}}+1)T_{\mathrm{sym}}\right) \ . \tag{4.44}$$

The decimation after filtering is included in the frequency response $H_{\mathrm{FIR}}(f)$. The factor N_{FIR} defines the length of the FIR filter, i.e. the number of filter taps. When combining these filters, the TETRA spectrum mask has to be considered in a way that the tolerance schemes, as given in table 4.4 showing the maximum noise levels, are fulfilled .

Frequency Offset	Maximum Noise Level
25 kHz - 50 kHz	−55 dBc
50 kHz - 100 kHz	−70 dBc
100 kHz - 250 kHz	−75 dBc
250 kHz - 5 MHz	−80 dBc
> 5 MHz	−100 dBc

Table 4.4: TETRA spectrum mask with the maximum noise levels in dependence of the frequency offset

There are several ways of choosing the filter parameters to fulfill the above mentioned objectives. However, due to the fact that the CIC filters are the only signal processing elements on the FPGA, the interpolation factor of the CIC was chosen as high as possible to lower the interpolation factor on the GPP. The decimation factor D_{FIR} was already defined in equation (4.38) to $D_{\mathrm{FIR}} = 9$. As follows, the smallest integer value of I_{FIR} that holds the criteria (4.42) is $I_{\mathrm{FIR}} = 16$. This determines the interpolation factor on the FPGA to $I_{\mathrm{CIC}} = 1000$. A CIC filter with $N_{\mathrm{CIC}} = 10$ stages suppresses the aliases of the RRC spectrum to be compliant with the tolerance scheme of the spectrum mask. With higher values of the filter length N_{FIR}, the spectrum gets closer to the RRC spectrum. According to section 4.3, the filter length was chosen as $N_{\mathrm{FIR}} = 6$. These parameters are summarized in table 4.5.

Parameter	Value
N_{CIC}	10
I_{CIC}	1000
M_{CIC}	1
I_{FIR}	16
D_{FIR}	9
N_{FIR}	6

Table 4.5: Parameters for the Resampling on the USRP

The frequency response of this system with RRC, CIC compensation and CIC filters is shown in figure 4.14 with reference to the TETRA spectrum mask and an optimum RRC filter.

It has to be mentioned that the impulse response of the filters for the receive path and the transmit path are identical. While the filter on the transmit path is the pulse shaping filter, the filter on the receive path is the matched filter. The values for the interpolation of the transmit side are now the values for the decimation on the receive side and vice versa, except for the decimation in the CIC filter which is double the interpolation factor on the transmit side. This is due to the different ADC and DAC rates. Another difference is that the filters on the receive side do not apply a decimation. The decimation in the last cascade is shifted to the timing synchronization, which is described in the following section.

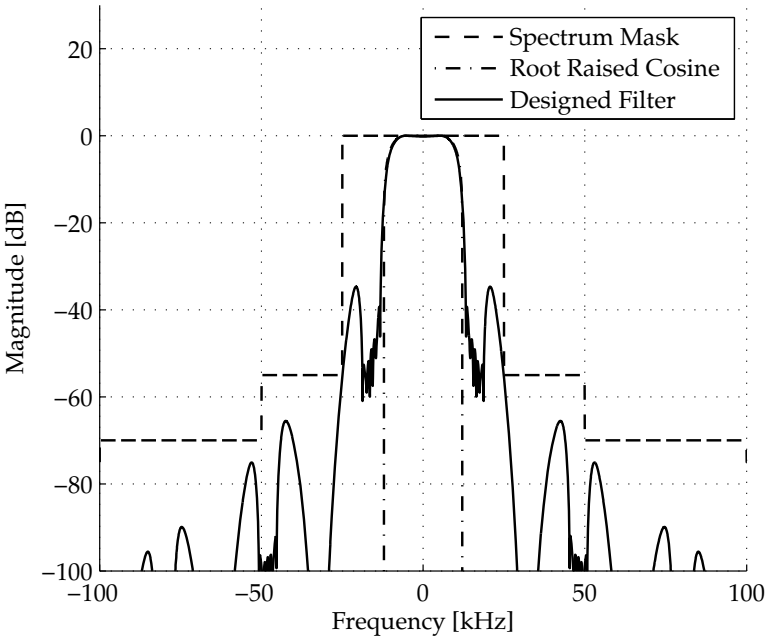


Figure 4.14: Designed filter, which fulfills the tolerance scheme of the TETRA spectrum mask

Table 4.6 shows the usage of LEs on the USRP's FPGA. It can be seen that 92 % of the provided space is used. Due to the fact that the decimation rate on the CIC receive filter twice the factor of the CIC transmit side, the usage for the receive side is higher. The usage of the overhead for buffers, Digital Down Converter or interfaces is given in table 3.1.

4.4.3 Timing Synchronization

The timing synchronization is described in section 4.3 but two parameters have still to be defined according to the underlying platform. These are the oversampling factor I and the number of symbols that have a constant time offset L. The oversampling factor is given by the resampling

Cyclone	Logic Elements	
	12060	100 %
Overhead	3895	32 %
CIC Rx	5374	45 %
CIC Tx	1871	16 %
Sum	11140	93 %

Table 4.6: Number of Logic Elements used in the FPGA

filter design, which leads to

$$I = I_{\mathrm{FIR}} = 16 \, .$$

The factor L can be approximated by the timing error, caused by the oscillator on the USRP. According to section 3.2, the frequency tolerance of the local oscillator is ± 20 ppm. This leads to a maximum time shift of approximately one symbol interval in a three second period. Working with a frame length of $L = 255$ symbols, as done in the PIM, leads to a time shift of 0.5 % of the symbol time period after any frame. Therefore, no adaptation of the PIM has to be taken into account and the timing error can be approximated as constant over one burst. Figure 4.15 shows the normalized time offset for the USRP. The dashed lines are the maximum and minimum theoretical time offsets from the oscilator's instability while the solid line shows measured values on the USRP over one second.

4.4.4 Frequency Synchronization

The frequency offset is detected with the FREQUENCY CORRECTION FIELD in the SYNCHRONIZATION DOWNLINK BURST as described in section 4.3. However, due to the 20 ppm inaccuracy of the oscillator the resulting frequency offset from the Flex400 daughter board can vary from ± 8 kHz to ± 10 kHz, depending on the carrier frequency. This value exceeds the maximum detectable frequency of the tracking which is 9 kHz. Therefore, prior to tracking, a carrier acquisition scheme has to be applied on the USRP. This scheme is according to the dual filter detector as proposed in [80]. The structure of this acquisition method

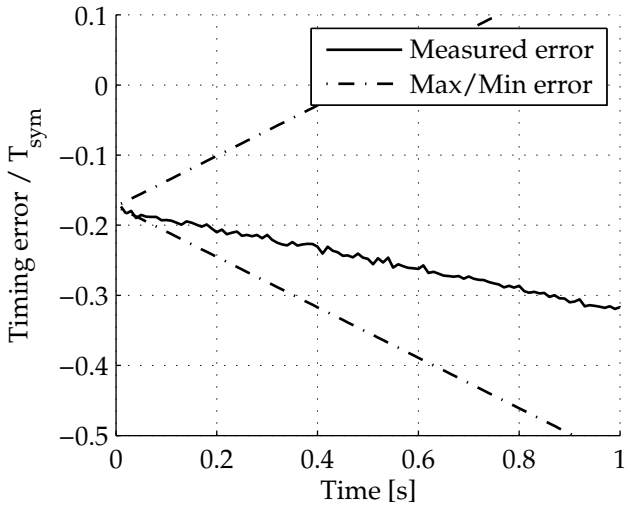


Figure 4.15: Measured time offset with maximum and minimum conditions for the timing error

is shown in figure 4.16. The band pass filters are placed symmetrically around the carrier frequency. If the energy of the signal filtered with the upper band pass is equal to the energy of the signal with the lower band pass, no frequency offset occurred. The two band pass filters and the spectrum of the TETRA signal are depicted in figure 4.17. The magnitude square of the filter output can be seen as the energy of the filtered signal, whereas the error signal e can be obtained by the subtraction of these two energies.

The filters are designed as low pass filters and then transformed to band pass filters. For the low pass filters the window design was chosen in combination with a Hamming window. To ensure that the error signal has its highest values between 8 kHz (frequency offsets lower than this value can be synchronized by the tracking) and 10 kHz (this should be the highest frequency offset due to the oscillator inaccuracy), the center frequency was chosen to be 9 kHz. Combined with the sampling rate of 288 kHz at the input of the GPP, this leads to a tap length of multiples of 16. Figure 4.17 shows the frequency response of the band pass filters with the frequency response of the root raised cosine pulse shaping filter

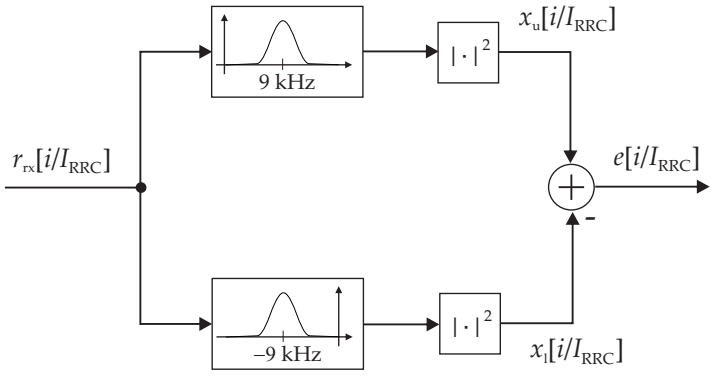


Figure 4.16: Block diagram of the frequency acquisition

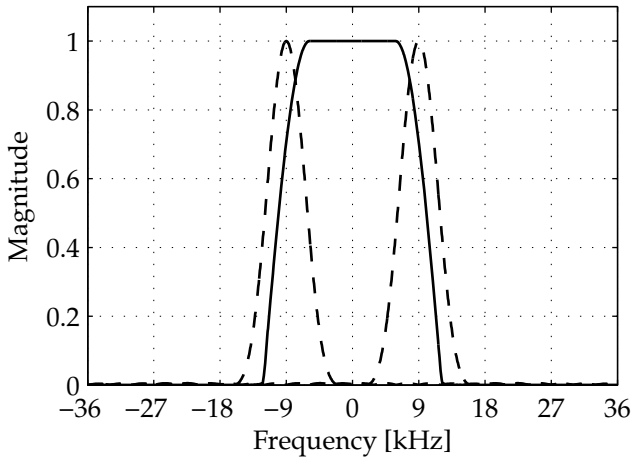


Figure 4.17: Illustration of the band pass filters and the raised cosine spectrum for the frequency acquisition

designed above. Figure 4.18 shows the error signal $e(\nu)$ in relation to the frequency offset ν.

Large frequency offsets like ± 10 kHz must only be synchronized at the

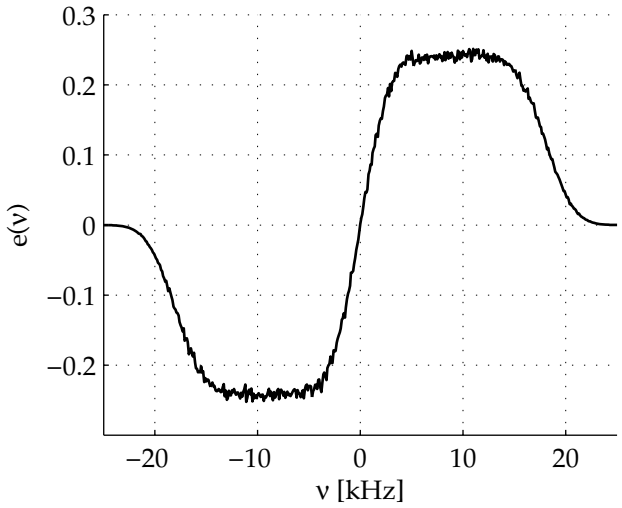


Figure 4.18: Error signal $e(\nu)$ of the dual filter detector in dependence on the frequency offset ν

start of the transmission. Therefore, the acquisition is only applied in the initialization phase of the receiver. The frequency synchronization as described in the PIM controls that the frequency offset does not exceed the maximum value. The measurement of the current offset is performed for each received SYNCHRONIZATION DOWNLINK BURST and the approximated frequency offset is saved until the next SYNCHRONIZATION DOWNLINK BURST is detected. The following demodulation can be implemented as a subtraction of the previous phase value and the phase offset, which was evaluated by the frequency synchronization, from the current phase value.

4.4.5 MAC receiver

The whole frame synchronization as well as the receiver for the traffic channel and the AACH are taken from the PIM without any adaptations due to the signal processing. Changes that are made for measuring

the processing time (varying of the data types, hand written versus generated code) are described in section 4.5.

4.5 Benchmarks for the waveform on the GPP

The PSM for the USRP that was described in section 4.4 is transformed into C++ code and the execution time on a GPP was measured. The processor is an Intel Core 2 Duo CPU (P8400) with a clock frequency of 2.28 GHz. It was introduced in section 2.4 for the evaluation of the code generation overhead. The C++ code is compiled with VS and executed on a Windows 7 operating system. All measurement results apply to one NORMAL DOWNLINK BURST, which comprises 255 symbol periods.

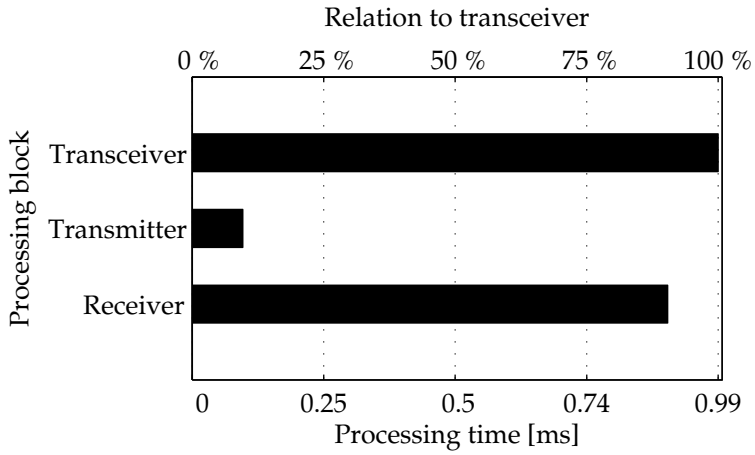


Figure 4.19: Comparison of the processing time for the transceiver, transmitter and receiver

Figure 4.19 shows the processing time for the complete transceiver, which is 0.99 ms. With the timing constraint of 14.17 ms for one burst, only 7 % of the processor capacity is used. In this case the code was not optimized for the platform, except for the compiler flags. Figure 4.19 shows the processing time for one burst on the lower x-axis. To

highlight the influence of the measured code segment on the whole system, the x-axis on the top is the relation of the processing time to the transceiver time. This form of representation is maintained for the following figures. In the transceiver measurements, the relation for the processing time from transmitter to receiver can be seen. Due to the increasing complexity for synchronization and decoding, the receiver consumes approximately ten times more processing time than the transmitter.

In figure 4.20 the processing times of the different blocks in the transmitter are shown. They are separated in the MAC blocks of the traffic channel (MAC TCH) and the broadcast channel (MAC AACH), the physical layer (PHY) and finally the sample rate conversion (SRC). The fact that the broadcast channel consumes only 15.5 µs while the traffic channel consumes 33 µs can be explained by the different frame lengths of both channels.

The processing times of the functions inside the traffic channel are shown in figure 4.20.

The broadcast channel MAC AACH consists of the following processing blocks: encoding and scrambling. The encoder is a Reed Muller encoder as described in section 4.3 with a processing time of 15 µs. This is due to the slow realization of the matrix multiplication for binary values. However, this comprises only 1.5 % of the processing time for the transceiver, therefore no optimized version of the Reed Muller encoder was realized. The encoded data are scrambled with the same scrambling sequence used in the MAC TCH. This operation takes less than 1 µs.

The PHY consists of the burst builder, which inserts training sequences and synchronization bursts in the data stream. It furthermore comprises the modulator for mapping the bits from the two logical channels (AACH and TCH) to symbols in the complex plane. With a processing time of 36 µs this is the block with the highest time consumption in the transmitter. It can also be seen that the sample rate conversion can be applied very efficiently with the polyphase structure of the multirate filter. With the processor's clock frequency of 2.28 GHz and the processing time for the Sample Rate Conversion (SRC) of 11.3 µs, approximately hundred clock cycles are used for the filtering and resampling of one complex symbol. This is a good value for a processor that is not optimized for digital signal processing.

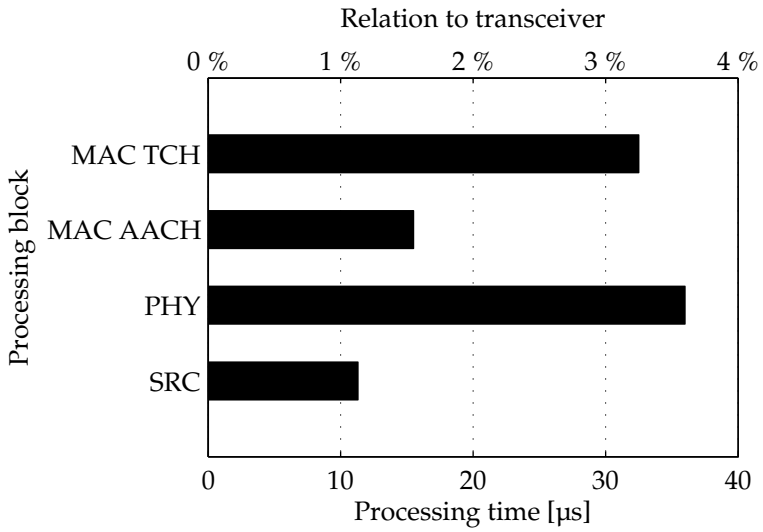


Figure 4.20: Comparison of the processing times in the transmitter

Figure 4.21 provides more details about the processing times of the signal processing blocks in the MAC TCH. The used traffic channel was the TCH/4.8 with an encoding rate $r = 292/432$. The Rate-Compatible Punctured Convolutional (RCPC) code needs 20 µs and is the block with the highest time consumption. However, even if this block comprises puncturing and encoding, it uses only 2 % of the whole transceiver. The interleaving and the scrambling need 5 µs and 7 µs. This is due to the fact that both blocks are implemented regarding time and not memory. For the interleaving, an array is saved that provides mapping from the incoming data to the interleaved data. This vector is calculated at the initialization phase and has to be provided at run time. The implementation of the scrambling is similar. The scrambling sequence is calculated in the initialization phase and saved as a constant array during run time i.e. it is merely an exclusive disjunction of the incoming data with a constant array.

Figure 4.22 shows the processing times of the various blocks in the receiver. It can be seen that the processing times are larger than the results

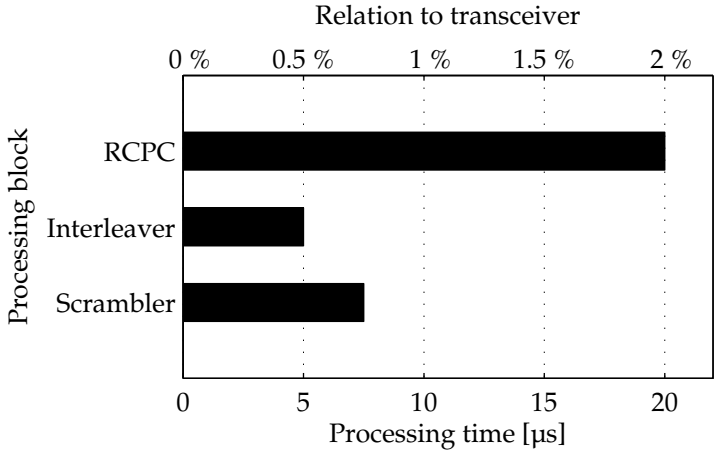


Figure 4.21: Comparison of the processing times within the MAC TCH in the transmitter

for the transmitter in figure 4.20. The receive side of the AACH receiver consists of a syndrome decoder and descrambling. The relatively short time for this processing block is due to the relaxed non-optimal implementation that corrects only one error and the short frame length. The SRC takes, with 110 µs, about eight times more than the SRC in the transmitter. This is due to the fact that no decimation is applied after filtering. The interpolated values are sent to the timing error detector, which triggers the downsampling. Similar to the SRC, the PHY on the receive side consumes with 368 µs about ten times more than the physical layer in the transmitter. In this case, this is due to the various synchronization schemes that must be applied.

In figure 4.23, the processing times of the blocks in the PHY are shown. While the transmitting part of the PHY consists only of a burst builder and a modulator, the receiving part of the PHY must correct time, frequency and symbol offsets. In these synchronization schemes, the timing error correction is the most simple regarding the processing time of 50 µs. The frequency error correction applies two correlations to find the synchronization burst and is therefore more complex with 146 µs. The frame synchronization realizes two buffers that are filled when training se-

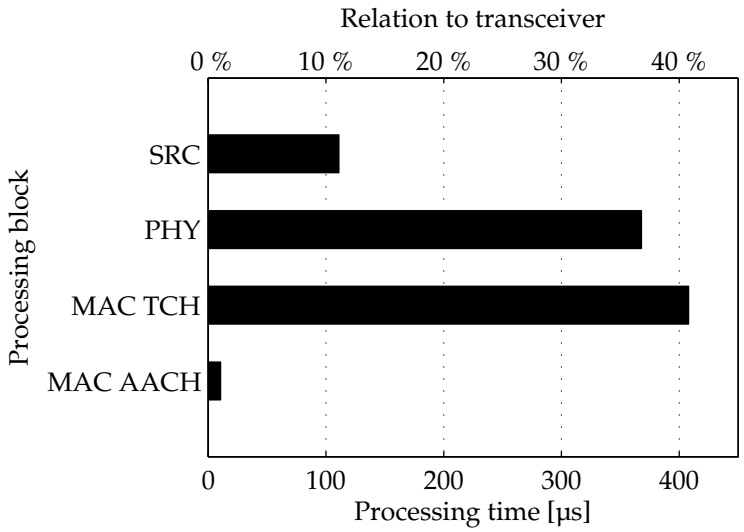


Figure 4.22: Comparison of the processing times in the receiver

quences are found inside a vector. These buffers are transmitted to the TCH and AACH receiver.

The measurements of the processing time for the MAC TCH receiver are presented in figure 4.24. It is obvious that the Viterbi decoding is the bottleneck for the transceiver with 388 µs, which makes about 40 % of the processing time for the complete system. In relation to that, the descrambling and the de-interleaving can be almost ignored. The same block is used for descrambling as for scrambling, therefore the times are equal. The same is true for the de-interleaver, except for the different entries in the mapping vector.

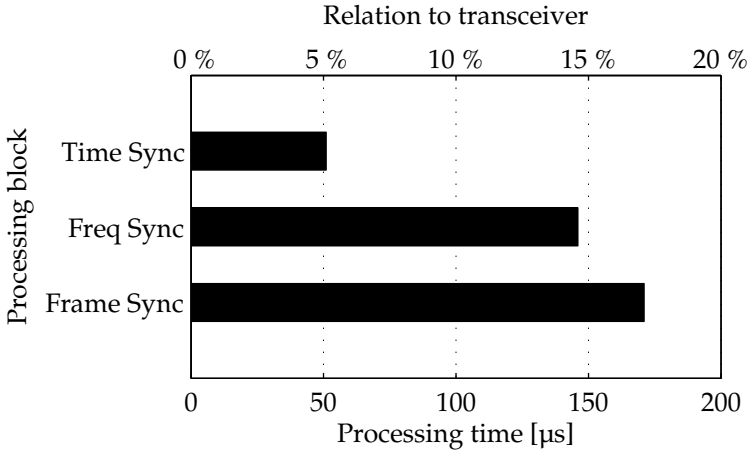


Figure 4.23: Comparison of the processing times within the PHY in the receiver

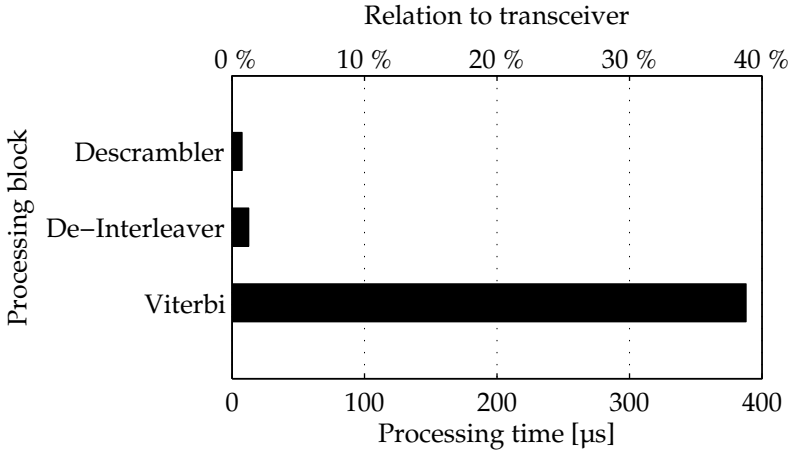


Figure 4.24: Comparison of the processing times within the MAC TCH in the receiver

4.6 Platform Specific Model for the SFF SDR

4.6.1 Separation on different Processing Units

The processing performance of the SFF differs from the USRP in so far as the FPGA on the SFF provides more logical resources and additional hardware multipliers. This leads to the possibility to shift more signal processing to the FPGA, as done in the PSM for the USRP. Another difference of this platform to the USRP is the DSP. Even if it provides an optimized hardware architecture for digital signal processing, its clock frequency of 594 MHz is approximately four times lower than the GPP used in combination with the USRP. Further, the cache size makes with 172 kB only 2 % of the cache size on the GPP. The third major difference is the possibility of the clock distribution on the DCM. This means that the sampling rate of the data converters can be configured in a range between 25 MHz and 125 MHz as described in section 3.3.

The choice of the sampling rate depends on two parameters: the symbol rate f_{sym} that should be achieved and the IF f_{IF} that is needed for downconversion. The sampling rate can be calculated as follows:

$$f_{\text{DAC}} = k f_{\text{IF}} = I_{\text{FIR}} I_{\text{CIC}} f_{\text{sym}} \,, \tag{4.45}$$

where k is an integer not smaller than two, and I_{FIR} and I_{CIC} are the upsampling factors of two cascaded filters, similar to the resampling filters designed in the PSM for the USRP. The CIC filter is processing the high upsampling rates while the FIR filter compensates the non exact flat pass band in combination with the RRC pulse shaping. Due to the fact that the clock can be adjusted to a multiple of the symbol rate no more decimation is needed.

An ideal relation between IF and clock rate would be a factor of four, where the IF would then only consist of additions and subtractions. With this configuration the IF of 30 MHz on the SFF would lead to a clock rate of 120 MHz. Even if the clock distribution circuit is able to handle this frequency, it is not a multiple of the symbol rate of 18 kHz. This would demand a resampling filter as designed in the PSM for the USRP. The advantage of the relaxed implementation of the down-conversion would be on the cost of an additional filter for the resampling in the DSP.

A good compromise would be a sampling rate of 90 MHz. With this clock frequency the down-conversion can be implemented by the multiplication of the signal with three constants for the in-phase component and another three constants for the quadrature component which can be saved easily in a LUT. This sampling rate would also define the upsampling factor of the system as follows:

$$I_{\mathrm{CIC}} \cdot I_{\mathrm{FIR}} = \frac{f_{\mathrm{ADC}}}{f_{\mathrm{sym}}} = \frac{90\,\mathrm{MHz}}{18\,\mathrm{kHz}} = 5000 \tag{4.46}$$

The sample rate conversion and pulse shaping on the FPGA does not only reduce the load of the DSP but also the load of the bus between DSP and FPGA. To ensure that the bus on the SFF runs at the same speed for transmit and receive path, the time synchronization has to be implemented in the FPGA. This leads to a separation as shown in figure 4.25.

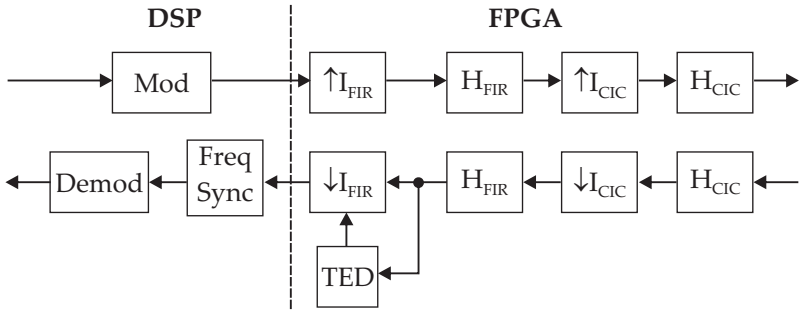


Figure 4.25: Separation of the processing function in the SFF between DSP and FPGA

4.6.2 Sample Rate Conversion

The conversion from symbol rate to sampling rate of the data converters consists only of interpolation filters, running on the FPGA. Similar to the USRP model, two interpolation filters should be designed that hold the conditions of the TETRA spectrum mask in a cascaded mode. In the receive path only one decimation filter shall be applied. After the adherent matched filter, the oversampled data are sent to the timing

error detector, which implements the final decimation regarding the calculated timing error. Therefore, the filters are designed in a way that, in combination with the timing synchronization, the FPGA resources are used to the maximum. The filter parameters are listed in table 4.7. The frequency response of the interpolation filter is shown in figure 4.26. For a reference also the TETRA spectrum mask and an ideal root raised cosine transmit filter is represented.

Parameter	Value
N_{CIC}	6
I_{CIC}	625
M_{CIC}	6
I_{FIR}	8
N_{FIR}	4

Table 4.7: Parameters for the interpolation on the SFF

4.6.3 Timing error synchronization

Due to the shift of the timing error synchronization from the GPP to the FPGA, the design has to be adapted from a floating point, frame based implementation to a fixed point, sample based design. The sine and cosine functions, needed for the calculation of the Fourier coefficients, must be implemented with a lookup table. This table must only hold I_{FIR} values. With the results from the previous section, this would mean eight values for the sine and another eight values for the cosine. This can be optimized with only one lookup table and shifted accesses for the sine and the cosine waves. The Fourier coefficients can be calculated by the multiplication of the squared magnitude with the sine waves. Figure 4.27 gives an overview of the timing error detector on the FPGA.

It is assumed that the timing error is constant over L symbol intervals. To avoid random peaks in the timing error signal a moving average filter with length $L \cdot I_{\mathrm{RRC}}$ is implemented. By realization of the filter in an FIR form, the number of taps grows with the length of $L \cdot I_{\mathrm{RRC}}$. To circumvent this, the FIR filter can be transformed to an IIR filter with the use of the geometric series. The z-transformation of a moving average

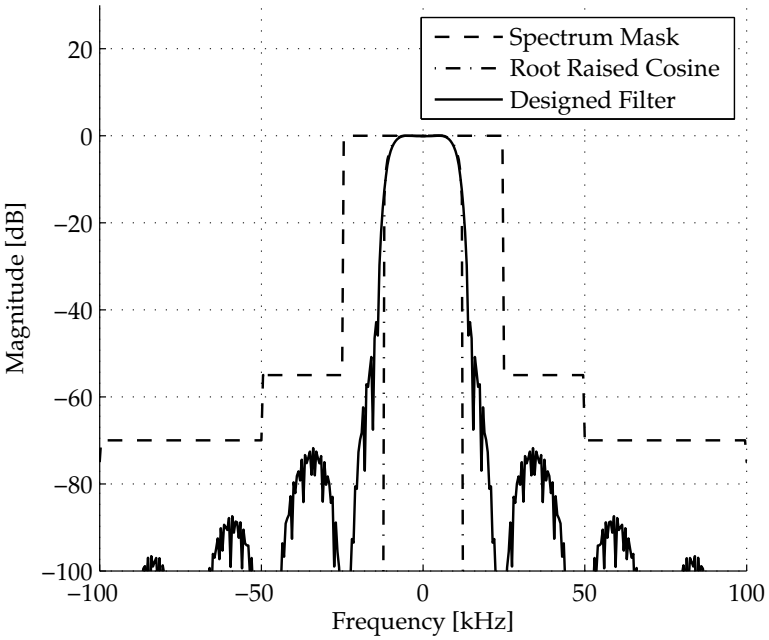


Figure 4.26: Frequency response of the interpolation filter on the SFF with respect to the TETRA spectrum mask and an ideal RRC filter

filter can simply be described as follows:

$$H(z) = \sum_{i=0}^{L \cdot I_{\mathrm{RRC}}-1} z^{-i} \tag{4.47}$$

$$= \frac{1 - z^{-L \cdot I_{\mathrm{RRC}}}}{1 - z^{-1}} \tag{4.48}$$

The increasing number of taps for the FIR filter is now transformed in a delay component. The resulting filter can be designed efficiently in hardware with only two adders and two delay lines. The block diagram of the efficient calculation is shown in figure 4.28.

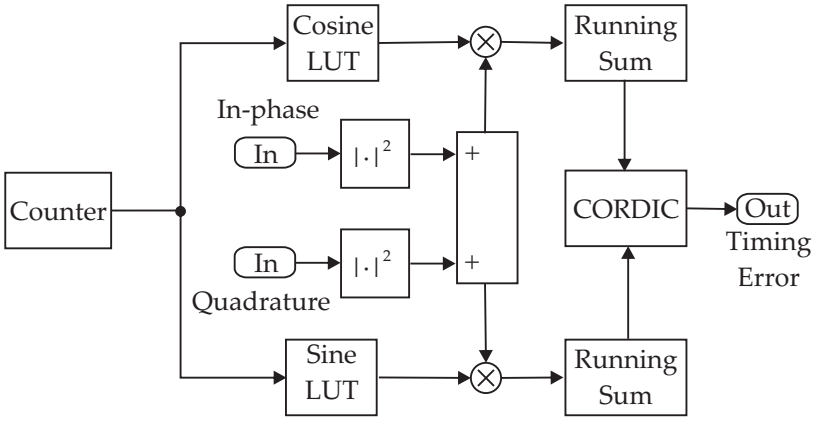


Figure 4.27: Overview of the fixed point realization of the timing error detector for the FPGA

The last block in the timing error detector is the transformation from real (x) and imaginary (y) parts of the symbol to its angle (φ). Therefore, an implementation of an inverse tangent function can be applied. This is done by a COordinate Rotation DIgital Compute (CORDIC) algorithmic element as described in [81]. Due to the fact that real and imaginary parts of the complex symbol are given, the vectoring computing mode is used. Therefore, a vector of N_φ angle values $\theta[i]$ is saved in the initialization phase to rotate the phase:

$$\theta[i] = \operatorname{atan}\left(2^{-i}\right) \quad \text{for} \quad 1 \leq i \leq N_\varphi \tag{4.49}$$

As follows, the iterative process to calculate the phase φ can be described by

$$\varphi[i+1] = \varphi[i] + \operatorname{sgn}(y[i]) \cdot \theta[i] \,, \tag{4.50}$$

$$x[i+1] = x[i] + \operatorname{sgn}(y[i]) \cdot 2^{-i} y[i] \,, \tag{4.51}$$

$$y[i+1] = y[i] - \operatorname{sgn}(y[i]) \cdot 2^{-i} x[i] \,. \tag{4.52}$$

This process assumes a complex value in the first quadrant of the complex plane. If this is not the case, the quadrant can be defined by the sign

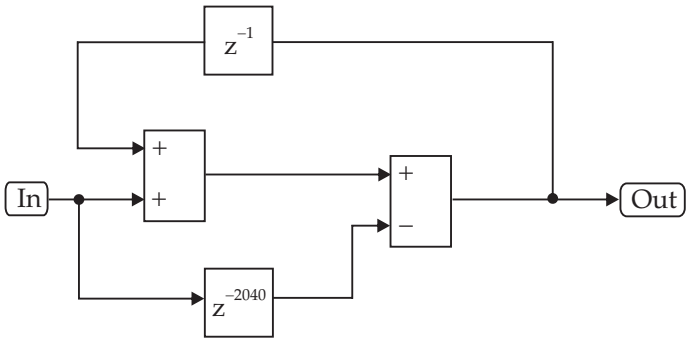


Figure 4.28: Hardware efficient implementation of a running sum

bits of the real and imaginary part and a phase offset can be added. Another difference to an original CORDIC algorithm is the missing scaling factor but this is no problem since only the resulting angle is of interest. The implementation of the sign function consist of an extraction of the sign bit of the imaginary part which triggers an adder. Furthermore, the multiplication by the factors of 2^{-i} are only bit shifts of i. The upper bound of N_φ is given by the used word length of 16.

The timing offset ϵ can be calculated with the evaluated phase through scaling and rounding to the nearest integer:

$$\epsilon[k] = \left\lfloor \frac{\varphi[k]}{2\pi} \cdot I_{\mathrm{FIR}} + 0.5 \right\rfloor \tag{4.53}$$

The maximum error of the CORDIC algorithm occurs if the correct phase is found at the step $i = N_\varphi - 1$. Then, the phase rotates with the maximum angle, which can be described by

$$\theta[N_\varphi - 1] = \operatorname{atan}\left(2^{-\left(N_\varphi - 1\right)}\right) \ . \tag{4.54}$$

Due to rounding, the accuracy of the phase with the CORDIC algorithm should be small against the following decimation, which can be described by the following condition:

$$\frac{\theta[N_\varphi - 1]}{2\pi} \cdot I_{\mathrm{FIR}} \ll 1 \tag{4.55}$$

Figure 4.29 shows the error value from the left hand side of inequality (4.55) in dependence to the number of steps in the CORDIC algorithm N_φ. It can be seen that seven iterations for the CORDIC algorithm already fulfills this condition.

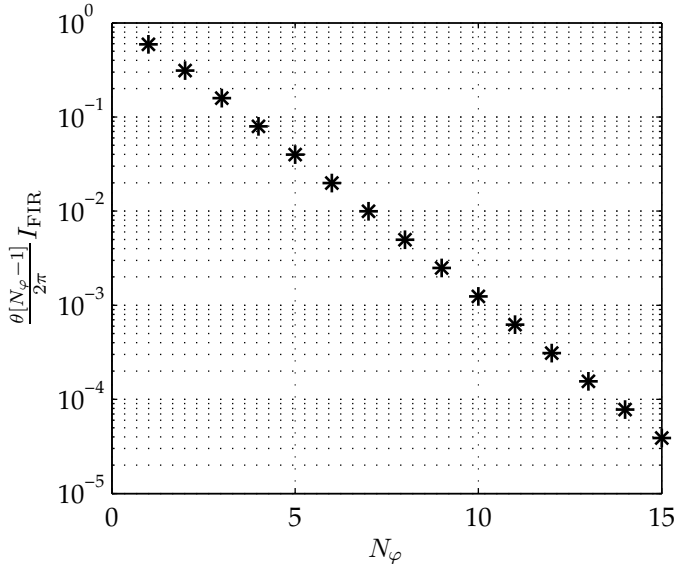


Figure 4.29: Error for the phase calculation in relation to the number of steps N_φ, assuming an interpolation factor of $I_{\mathrm{FIR}} = 8$

The resulting phase triggers the downsampling. The symbols with the corrected timing offset are sent to the DSP at the TETRA symbol rate of 18 kHz.

4.6.4 PSM on the DSP

The porting of the waveform from USRP to SFF should be done with a minimum effort of rewriting code, the majority of the PSM for USRP's GPP shall be used at the DSP on the SFF. The differences are, as already mentioned in the previous sections, the sample rate conversion and the time synchronization, which was shifted to the FPGA. In chapter

4.7 the operations are described that must be tuned in order to get an operating waveform. No frequency acquisition has to be applied due to the precise oscillator with an accuracy of 1 ppm, which leads to a maximum frequency offset of ±500 Hz.

4.7 Benchmarks for the waveform on the DSP

The PSM for the USRP can be ported to the SFF by shifting the time synchronization and the pulse shaping to the FPGA. This means these blocks have to be deleted from the DSP side. By measuring the processing time of the waveform, one single burst is in the range of seconds. It could be shown that the processing time of the code causes less problems than the memory allocation. Every memory section of the code must be placed on the external SDRAM, when the PSM, originally intended for a GPP is inserted. This needs several read and write operations, for example: to load the section of the SDRAM into the internal cache, to process these data and to write it back to the external memory.

	Generated	Hand-written
Processing time	120 ms	248 µs
Level 1 memory	0 kB	1.04 kB
Level 2 memory	0 kB	11.1 kB
SDRAM	54.8 kB	0 kB

Table 4.8: Comparison of processing times and memory allocation for generated and hand-written part of the transmitter

The difference between processing on the SDRAM and processing on the L1 and L2 memory sections can be seen with the following example. The MAC section of the transmitter has been rewritten in C code and compared with generated code. While the generated code was executed on the external SDRAM, the hand written code was executed on the internal cache of the processor. The results are shown in table 4.8. It can be seen that the generated code needs approximately 500 times more processing time than the hand-written code. This is mainly caused by the slow read and write accesses of the external memory.

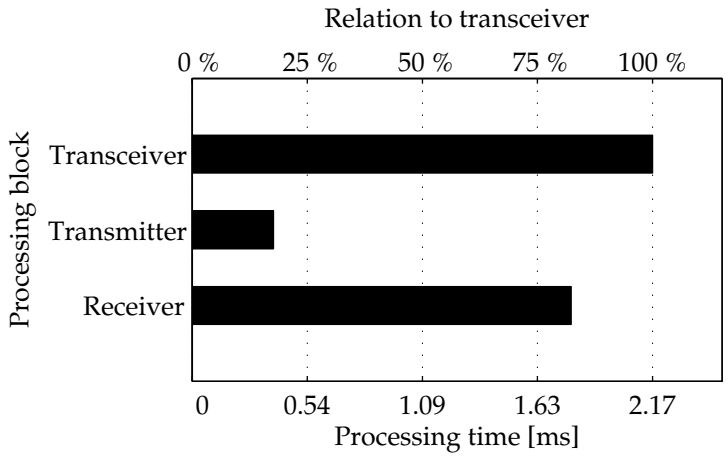


Figure 4.30: Comparison of the processing times of the transmitter, receiver and transceiver for the DSP

The generated code creates a lot of arrays for the signal processing where especially for the puncturing an array with a length of 1168 is created. In this array for puncturing the positions of the coded bits that can be dropped are contained. This is the same for the interleaving, where an array is created with the position of the interleaved data. While these methods work really fast, they consume a lot of memory. For a GPP with an internal cache of 8 MB this is not a problem. The DSP however, has to shift these sections to the external SDRAM, where it needs much more time. To compare the memory usage, the hand-written code was optimized for memory consumption but not for speed. This means the punctured channel encoder and the interleaver have been combined in one function. No bits are encoded that would be dropped due to the puncturing, and the encoded bits are immediately shifted to the right position as specified through the interleaving scheme. The code is shown in appendix A.3. This was also done for the scrambling of the data. The code for the optimized scrambling is shown in appendix A.4. Due to to the register length of 32, the whole register can be calculated with one single integer value that has a size of four bytes. The used memory of both functions is also shown in table 4.8. Compared to the memory

usage of the generated code with 54.8 kB, the optimized code consumes 12.1 kB. The reason why the generated code was placed on the SDRAM is the better comparability to the whole transceiver. By generating the code for the transceiver, it has to be placed on the external memory. This is not the same for the hand written code. Due to the optimization options, it is possible to place the code for the complete transceiver on the internal cache.

Figure 4.30 shows the relation of the processing times between transmitter and receiver. The receiver consumes approximately 80 % of the processing time for the complete transceiver. This is similar to the results for the GPP from section 4.5 although the time synchronization as well as the resampling was moved to the FPGA. However, the processing time for the complete transceiver is with 2.17 ms approximately twice the time of the code running on the GPP.

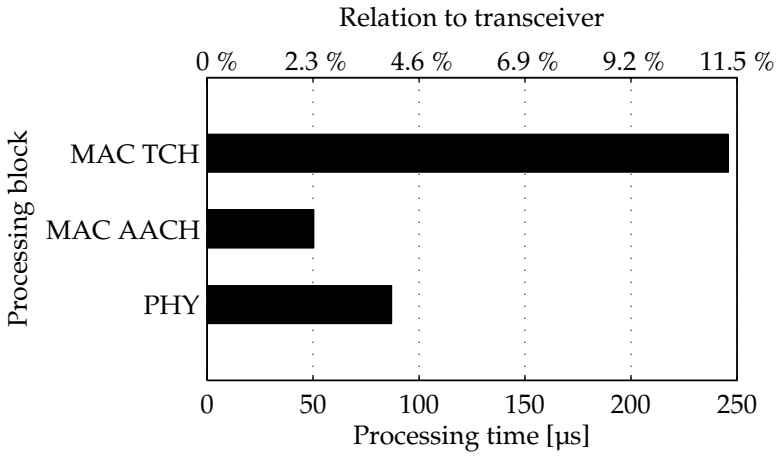


Figure 4.31: Comparison of the processing times within the transmitter on the DSP

The transmitter is split in three parts as shown in figure 4.31: The PHY, the traffic channel MAC TCH and the broadcast channel MAC AACH. The processing time for the TCH is already known from table 4.8. Compared to the results from figure 4.20 the MAC TX block needs more time. This is due to the fact that the code is now optimized for memory and

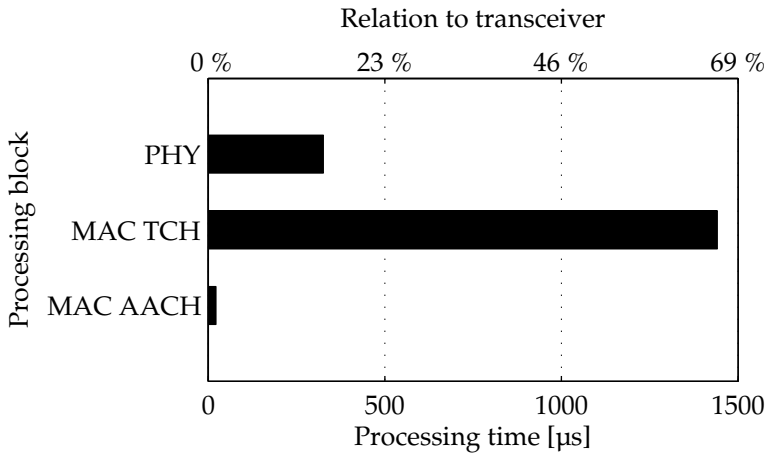


Figure 4.32: Comparison of the processing times within the receiver on the DSP

therefore slower than a code that is optimized for speed. The processing blocks PHY and MAC AACH were not changed.

Figure 4.32 shows the processing times of the functions on the receive side. It can be seen that the traffic channel is the block that consumes the most processing time. The difference to the other blocks have been increased in comparison to figure 4.22. This is due to the fact that the timing synchronization as well as the decimation at the receive side was realized on the FPGA.

The both remaining blocks on the physical layer need with 191 µs for the frequency synchronization and 135 µs for the frame synchronization approximately similar processing times as on the GPP. These processing times are shown in figure 4.33.

Figure 4.34 shows the reason for the large processing time of the MAC receiver. The Viterbi algorithm consumes approximately 65 % of the processing time of the whole transceiver. To circumvent this problem, modern DSPs provide hard wired Viterbi accelerators. Unfortunately the DM6446 does not provide those accelerators. Therefore, the decoder

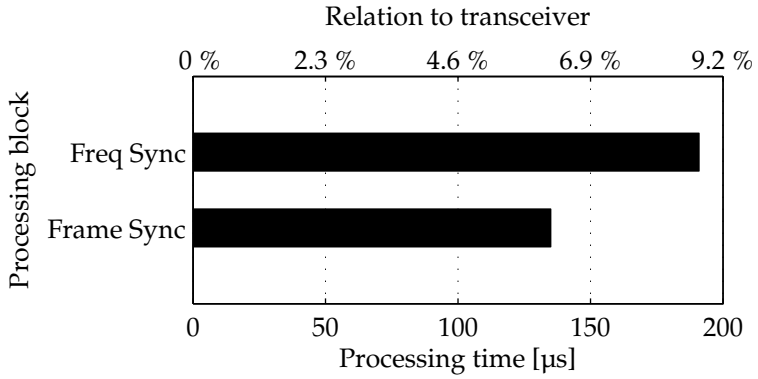


Figure 4.33: Comparison of the processing times within the PHY in the receiver

has to be implemented in software and remains the bottleneck of the complete system as already shown in the results for the GPP.

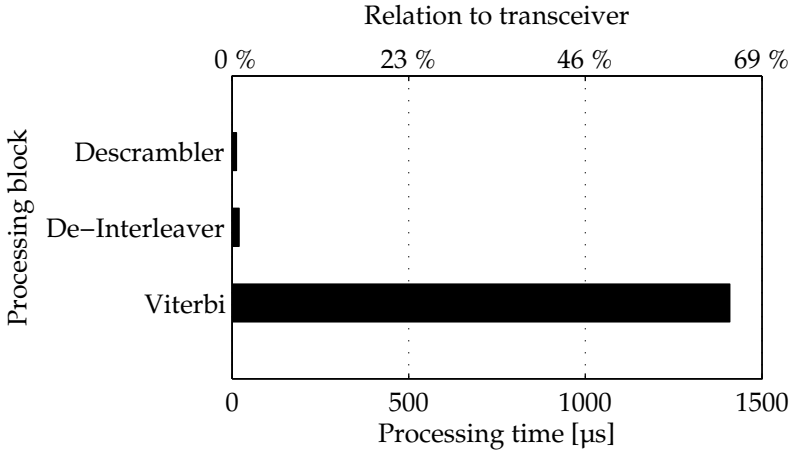


Figure 4.34: Comparison of the processing times within the MAC TCH in the receiver

4.8 Interoperability Tests

A signal generator and a analyzer were used as a reference to prove that the transmitting and receiving behavior of the generated waveforms for USRP and SFF are like TETRA signals. To test the transmit function, the signals from USRP and SFF were evaluated with a Rohde & Schwarz FSQ8 signal analyzer. As shown in figure 4.35, the signals are processed with a special software that shows the training sequence as well as the signal constellation and the modulation accuracy.

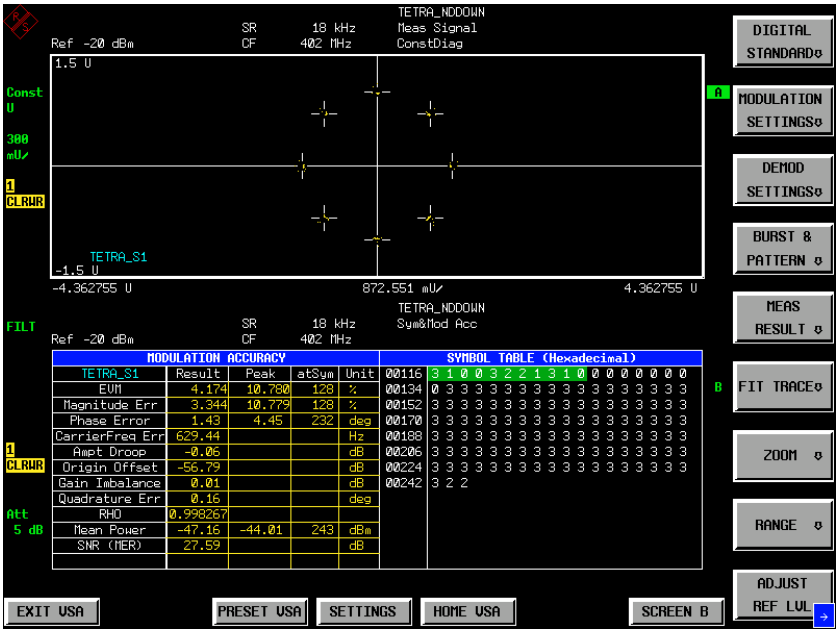


Figure 4.35: Received TETRA signal from the SFF, evaluated with an FSQ8

To test the receive path on USRP and SFF SDR, TETRA signals were generated with a Rohde & Schwarz SMU200 signal generator. USRP and SFF were tuned on a carrier frequency of 402 MHz and decoded the NORMAL DOWNLINK BURST correctly. To ensure that the transmissions from USRP to SFF and vice versa works correctly, the transmit and receive functions were tested in both directions and the decoded data

were compared for errors. However, no errors occurred due to the close distance between transmitter and receiver. The setup for the demonstration of the waveform is shown in figure 4.36. The USRP is on the left hand side with a laptop for the signal processing. The SFF is on the right hand side of the picture. The desktop is only used for configuration and does not process any radio signals. The measurement equipment is in the center of the picture. The lower device is the FSQ8 signal analyzer and the upper device is the SMU200A signal generator.

Figure 4.36: Measurement setup for the TETRA waveform testing

5 Portability aspects of further waveform implementations

5.1 Family Radio Service

5.1.1 Overview of the standard

Family Radio Service (FRS) is a communication system that is used for speech transmission within a range of less than one mile. Therefore, the transmitters have a maximum output power of 500 mW. The signal is modulated with an FM scheme, where the envelope of the complex baseband signal $s_{\text{tx}}(t)$ can be described as follows:

$$s_{\text{tx}}(t) = \exp\left(j2\pi f_{\Delta} \int_0^t v(t')dt'\right), \tag{5.1}$$

where $v(t)$ is the audio signal that should be transmitted and f_{Δ} describes the frequency deviation. FRS is a narrowband FM system and is intended for channels with 12.5 kHz bandwidth. Therefore, f_{Δ} is set to 2.5 kHz while the maximum frequency of the audio signal $v(t)$ should not exceed 3.5 kHz. The Federal Communications Commission (FCC) specified 14 channels between 462.5625 MHz and 467.7125 MHz. This means that the carrier frequency is in the same range as the TETRA waveform and can be achieved with the same RF front ends.

5.1.2 Digital signal approximation for the portability

In a digital representation, the integral in equation (5.1) can be replaced by a sum of the values coming from the sound card. These values are scaled with the frequency deviation factor and synthesize the angle for

the complex base band signal. The receiver works vice versa. The angle of the incoming in-phase and quadrature component is determined and scaled. The audio signal can be retrieved by subtraction of the previous angle from the current angle.

5.1.3 Results

Due to the narrow bandwidth of 12.5 kHz, a high resampling has to be applied. However, the signal has just to be resampled to a data rate that the sound card supports. In case of the USRP this is 32 kHz. Together with the ADC rate of 64 MHz, this leads to a decimation rate of 2000 in the FPGA. The audio system on the SFF can be coupled to the FPGA data rate. Therefore, the same rates as for the TETRA waveform was applied leading to a reuse of the decimation, interpolation and downconversion.

Since the transmitter comprises only an integration and the receiver consists only of a differentiator beside the resampling filters, the whole PIM can be shifted to the GPP, DSP or even FPGA without any modifications. This shows that if the processing power of the underlying platform is amply dimensioned for the complexity and bandwidth of the waveform, the concerned waveform here can be ported without any knowledge of the underlying hardware.

5.2 Wireless LAN in the version IEEE 802.11g

5.2.1 Overview of the standard

Since 1980, the IEEE releases specifications about Local Area Networks (LANs) in its project 802. In 1997, the first specification of a Wireless Local Area Network (WLAN) was published within the working group eleven. This was the beginning of the standard IEEE 802.11, which specifies MAC and PHY for local radio networks. In the original version, data rates up to 2 Mbit/s were reached with spread spectrum techniques in the 2.4 GHz Industrial, Scientific and Medical (ISM) radio band. Just about two years later, in 1999, the standard was extended with two annexes. IEEE 802.11a specified an Orthogonal Frequency Division

Multiplex (OFDM) transmission in the 5 GHz ISM band with data rates up to 54 Mbit/s and 802.11b extended the transmission on 2.4 GHz to data rates up to 11 Mbit/s. The most common extension till the present day is annex g, which was released in 2003 and specified the OFDM transmission with up to 54 Mbit/s in the 2.4 GHz range [82]. This is a frequency range that is supported by the Flex2400 daughter board for the USRP and the WiMAX module RF front end for the SFF as described in sections 3.2 and 3.3.

5.2.2 Challenges for the portability of the waveform

The major challenge in implementing and porting the 802.11g waveform is its large bandwidth. The symbol duration of one OFDM symbol is specified to 3.2 µs. This results in a bandwidth of 20 MHz with 64 subcarriers. Even if the data converters on both platforms can support this bandwidth, the buses between DSP/GPP and FPGA are not fast enough. By assuming 16 bit in-phase and quadrature components, data rates of 640 Mbit/s must be achieved, while the maximum data rate between GPP and FPGA on the USRP is 228 Mbit/s and the maximum data rate on the SFF SDR DP is 52 Mbit/s. This means that the complete waveform should be implemented on the FPGA. However, both platforms provide not enough logic to achieve this. Due to the fact that no real time implementation is possible with the given platforms, the maximum achievable bandwidths should be evaluated. Therefore, a PIM was realized with the channel from section 4.3. To achieve a synchronization in time and frequency, an algorithm from Schmidl and Cox [83] was realized that uses training sequences in the beginning of each transmission to find the exact position of an OFDM symbol in the data stream coming from the ADCs.

5.2.3 Results

To fulfill the real time constraints for 802.11g, the time for one OFDM symbol with the guard interval must not exceed 4 µs. Measurements on the Intel P8400 showed that the processing time for the transmitter is approximately 2.78 µs. Without the limitation due to the bus between GPP and FPGA, it is possible to implement the transmit side in software and support the real time constraints of a 20 MHz bandwidth. However,

due to the fact that the maximum data rate of the bus is at 228 Mbit/s, only a bandwidth of 6.4 MHz can be achieved. The spectrum of the transmitted signal was recorded with a Rohde & Schwarz FSQ8 signal analyzer and is shown in figure 5.2(a). To detect the 802.11g frame, a training sequence was transmitted where its spectrum is shown in figure 5.2(b). The representation of the spectrum shows the power in a logarithmic scale.

The processing time for the receiver is 58.9 µs. Therefore, with the used processor, a whole system comprising of transmitter and receiver with a USRP can only be realized with a complex bandwidth of up to 1.3 MHz. This is due to the timing and error synchronization as well as the Viterbi decoding. These two processing blocks consume 75 % of the processing time for one OFDM symbol.

When porting the code for the transmitter from the USRP to the SFF SDR a processing time of 600 µs was measured for one OFDM symbol. As already mentioned in section 4.7, this is due to the floating point implementation. The data section of the code becomes too large for the internal memory and has to use the slow external SDRAM. The usage of external memory can be circumvented with a fixed point design and the use of optimized libraries as presented in section 2.4. With these changes the processing time for an OFDM symbol decreases to 18.9 µs.

The transformation of the PIM for the receiver yields similar results. The measurements of the floating point code lead to a processing time of 222 ms. Even with a fixed point implementation of the model, the memory size exceeds the internal cache. Therefore, the timing and frequency synchronization must be shifted to the FPGA. The sample based processing on this reconfigurable logic requires a completely new modeling of the synchronization algorithm. Especially the peak detection of the correlation must be redesigned. Nevertheless, the processing time for one OFDM symbol in the receiver could be decreased to 12.4 ms. This time could be further reduced with an optimized version of the Viterbi algorithm. Similar to the results on the USRP, this is the processing block that consumes most of the time.

The realization of the PSMs for the USRP and the SFF SDR are shown in figure 5.1. The porting process for this waveform is described in more details in [84] and [85].

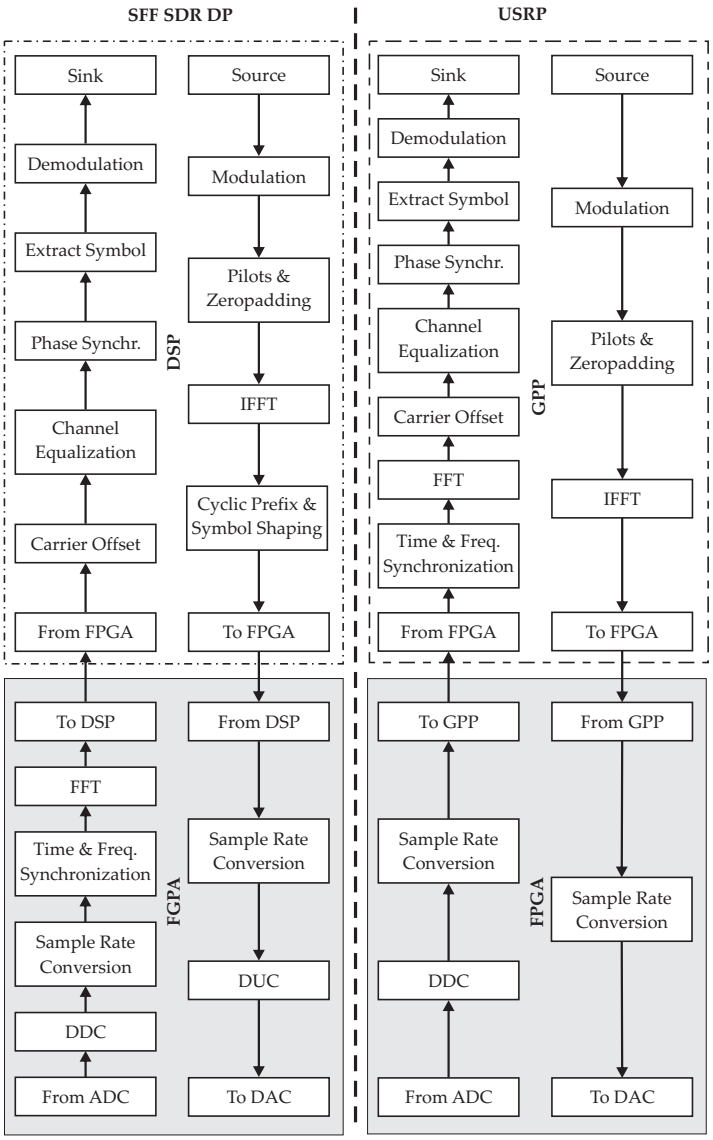
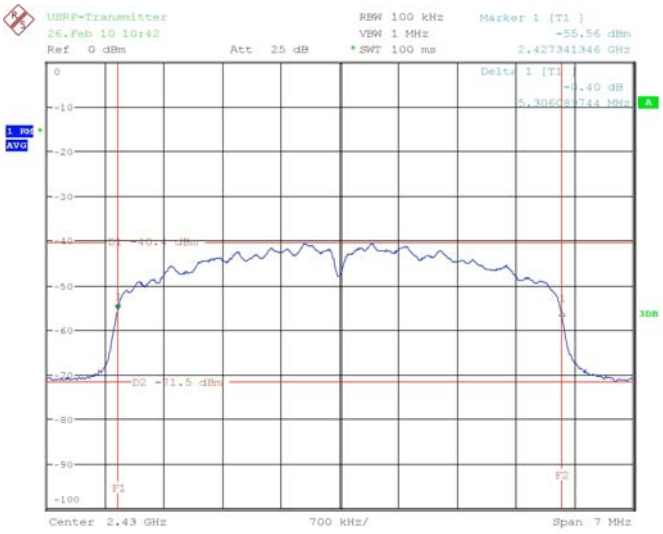
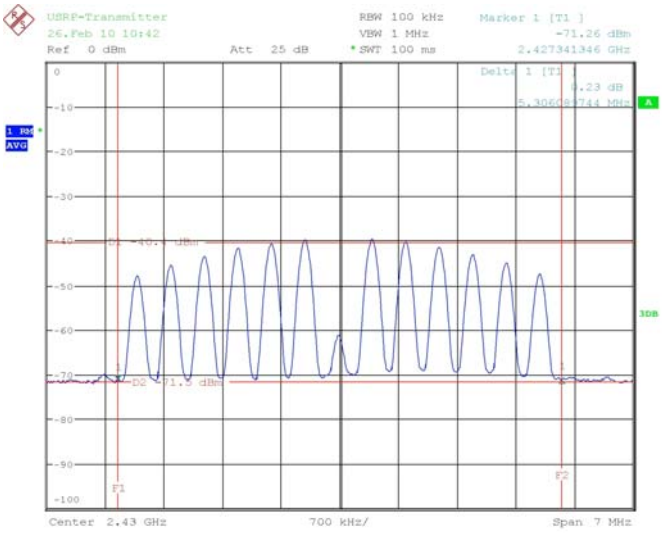


Figure 5.1: Separation of the waveform for the processing elements on USRP and SFF SDR



(a) Power spectrum of the waveform data



(b) Power spectrum of the training sequence

Figure 5.2: Measured power spectrum in dBm of the OFDM waveform

6 Conclusions

6.1 Contributions

Chapter 2 presented different possibilities of waveform development with all their assets and drawbacks for waveform portability. A design flow based on the Model Driven Architecture was proposed as the best choice for portable waveform development in combination with generated codes for various processing elements like DSPs, GPPs and FPGAs. Due to the fact that code generation is an important part of this concept, the overhead of generated code was measured in the sense of processing time and number of used logic elements. Especially for processing components that are used often like digital filters, there is no large overhead in the generated code. It could be seen that hand-written code is often much slower than the generated code. Even in relation to highly optimized vendor specific libraries or components, there is no tremendous loss in performance of generated codes.

Two platforms used for waveform development were introduced in chapter 3: the USRP and the SFF SDR. The possibilities and limitations of both platforms regarding the speed of interfaces, the clock frequencies as well as the processing capabilities are shown. Furthermore, the integration of the design flow based on the MDA is presented for both platforms.

A proof of concept is given in chapter 4 to demonstrate that the proposed design flow works. A TETRA waveform was realized and ported from the USRP to the SFF SDR. Results of the porting process are given as well as processing times for the generated code in comparison to an optimized code.

Chapter 5 describes the challenges and results for the port of further waveforms, which were also ported from USRP to SFF. Since a detailed

description would go beyond the scope of this work, only an overview of the results is given.

6.2 Outlook

One of the major questions in the Model Driven Architecture with code generation is: "Can we generate the waveform for our platform without hand written modifications?". The results from this work show that this depends on the waveform and on the platform. The FRS waveform can easily be ported without modifications from one of the presented platforms to another. Unfortunately, this is not possible for more complex waveforms like the PSM for TETRA on the SFF SDR. Parts of this waveform had to be rewritten in C to fulfill the memory constraints of the processor.

The code generation for DSPs faces two problems: the low cache size and the fixed point architecture. Nevertheless, code generation and platform independent models in floating point arithmetic are the right direction in waveform development. This can also be confirmed with the current developments on the DSP market. TI releases a new DSP with an arithmetic to use fixed and floating point algorithms [86]. The internal memory of this new branch is with approximately 8 MB in the same range as GPPs running on a desktop. With these changes in DSP architecture, code generation for DSPs will result in lower processing times. The code size and memory usage will not decrease but due to the rising cache size, the processor is able to handle this.

One trend that can be seen in the development of new processors (DSPs as well as GPPs) are multiple cores. Regarding portability of waveforms it has to be verified that the same code can be executed on single-core and multi-core processors. It furthermore must be checked that the code on multi-core processors runs faster with the number of cores. This can be achieved by the integration of OpenMP. OpenMP is an API that supports multi-platform shared-memory parallel programming in C/C++ on all architectures. The advantage is that the directives are ignored from single-core systems. Therefore, source codes can be generated that can be applied in multi-core systems but also be compatible for single-core processors.

The role for FPGAs will be the interface between the data converters and the processors. Therefore, sample rate conversion, pulse shaping, matched filtering and time synchronization will be the major tasks. Even if FPGAs are the faster processing units for digital signal processing, the configuration of the data is too complex and needs too much development time. One example for this is the OFDM transmitter that was evaluated in this work. The FPGA is able to handle the IFFT very efficiently, but the inclusion of the guard interval with cyclic prefix can only be accomplished with multiple FIFO buffers. This diminishes the advantage of the efficient IFFT processing. This is also valid for the Viterbi algorithm, which is the most time consuming process in each evaluated waveform. The shift of the Viterbi algorithm back to the FPGA would bring a tremendous speedup for the system. However, it would overload the bus between the processing elements. Solutions for this problem are already presented by the industry. The high performance DSP chips from TI for baseband processing provide hardware accelerators for time consuming functions like the Viterbi algorithm [87] or a rake receiver.

A Source Code

A.1 Source Code of the unoptimized FIR filter

```
1  static void mdlOutputs(SimStruct *S, int_T tid)
2  {
3    int_T         i, j;
4    int_T         width;
5
6    creal_T *input = (creal_T *) ssGetInputPortSignal(
       S, 0);
7    creal_T *output = (creal_T *)
       ssGetOutputPortSignal(S, 0);
8
9    width = ssGetOutputPortWidth(S,0);
10
11   for (i=0; i<width; i++) {
12
13     buffer[position] = input[i];
14     output[i].re = 0;
15     output[i].im = 0;
16
17     for(j = 0;j<NUM_COEF;j++){
18       output[i].re +=     filter_coef[j].re*buffer[(
           position+j)%NUM_COEF].re -filter_coef[j].im*
           buffer[(position+j)%NUM_COEF].im;
19       output[i].im +=  filter_coef[j].re*buffer[(
           position+j)%NUM_COEF].im +filter_coef[j].re*
           buffer[(position+j)%NUM_COEF].im;
20     }
21     position = (position+1)%NUM_COEF;
22   }
23 }
```

A.2 Source Code of the unoptimized FFT

```
1 std::complex <float> * my_fft(int length, std::
    complex <float>* input)
2 {
3  using namespace std;
4  complex <float> *output = new complex <float> [
     length];
5  complex <float> *even_in = new complex <float> [
     length/2];
6  complex <float> *odd_in = new complex <float> [
     length/2];
7  complex <float> *even_out = new complex <float> [
     length/2];
8  complex <float> *odd_out = new complex <float> [
     length/2];
9  complex <float> odd_factor;
10 int i;
11
12 if(length == 1)
13   output = input;
14 else
15   {
16   for(i = 0;i<length/2; i++){
17     even_in[i] = input[2*i];
18     odd_in[i] = input[2*i+1];
19     }
20   even_out = my_fft(length/2,even_in);
21   odd_out = my_fft(length/2,odd_in);
22
23   for(i = 0;i<length/2; i++){
24     odd_factor = polar(1.0,-2*PI*i/length) ;
25     odd_factor = odd_out[i]* odd_factor;
26     output[i] =     even_out[i] + odd_factor;
27     output[i+length/2] = even_out[i] - odd_factor;
28   }
29 }
30 return output;
31 }
```

A.3 Source Code of the optimized RCPC for the TCH/4.8

```
1 void tch4_8_rcpc(unsigned char inVar[432], unsigned
    char outVar[255])
2 {
3   unsigned char *state = malloc(292+4);
4   unsigned char init[] = {0U,0U,0U,0U};
5   short k,j,i;
6
7   memcpy(state,init,4);
8   memcpy(state + 4, inVar, 292);
9
10  j = 0;
11  i = 0;
12  for (k=0; k < 146; k++){
13    outVar[j+i] = (state[2*k+4] + state[2*k+3] +
        state[2*k])%2;
14    outVar[j+i+1] = (state[2*k+4] + state[2*k+2] +
        state[2*k+1] + state[2*k])%2;
15    if((i+2)!=65){
16      outVar[j+i+2] = (state[2*k+5] + state[2*k+4] +
          state[2*k+1])%2;
17      i+=3;
18      }
19    else{
20      i=0;
21      j+=65;
22      }
23    }
24 }
```

A.4 Source Code of the optimized scrambling

```
1 void tch_scrambling(unsigned char inVar[432],
     unsigned char outVar[432])
2 {
3   unsigned int init = 0x4183E8B7;
4   unsigned int new;
5   int k;
6
7   for(k=0;k<432;k++){
8     new = 0x80000000&(init ^ (init << 1) ^ (init <<
          3) ^ (init << 4) ^ (init << 6) ^ (init << 7)
          ^ (init << 9) ^ (init << 10) ^ (init << 11) ^
           (init << 15) ^ (init << 21) ^ (init << 22) ^
           (init << 25) ^ (init << 31));
9     init = init >> 1 | new;
10    outVar[k] = inVar[k] ^ (init >> 31);
11    }
12 }
```

Acronyms

AACH	Access Assignment CHannel
ADC	Analog-to-Digital Converter
AEP	Application Environment Profile
AGC	Automatic Gain Control
AM	Amplitude Modulation
API	Application Programmable Interface
ASP	Audio Serial Port
BPSK	Binary Phase-Shift Keying
CCS	Code Composer Studio
CF	Core Framework
CIC	Cascaded Integrator-Comb
CIM	Computational Independent Model
CLB	Configurable Logic Block
CORBA	Common Object Request Broker Architecture
CORDIC	COordinate Rotation DIgital Compute
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DCM	Data Conversion Module
DDC	Digital Down Converter
DDS	Direct Digital Synthesizer
DPM	Digital Processing Module

DQPSK	Differential Quadrature Phase-Shift Keying
DSP	Digital Signal Processor
DUC	Digital Up Converter
EMIF	External Memory Interface
ETSI	European Telecommunications Standards Institute
FCC	Federal Communications Commission
FDD	Frequency Division Duplex
FDMA	Frequency Division Multiple Access
FFT	Fast Fourier Transformation
FIFO	First In, First Out
FIR	Finite Impulse Response
FM	Frequency Modulation
FPGA	Field Programmable Gate Array
FRS	Family Radio Service
GPP	General Purpose Processor
GRC	GNU Radio Companion
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HDL	Hardware Description Language
IC	Integrated Circuit
IFFT	Inverse Fast Fourier Transformation
IPS	Intel Parallel Studio
ISM	Industrial, Scientific and Medical
ISR	Interrupt Service Routine
IEEE	Institute of Electrical and Electronics Engineers
IF	Intermediate Frequency
JTAG	Joint Test and Action Group

JTRS	Joint Tactical Radio System
JPO	Joint Program Office
LAB	Logic Array Block
LCC	Local C Compiler
LE	Logic Element
LMR	Land Mobile Radio
LNA	Low Noise Amplifier
LO	Local Oscillator
LTE	Long Term Evolution
LTS	Long Training Sequence
LUT	Lookup Table
MAc	Multiply and Accumulate
MAC	Media Access Control
MDA	Model Driven Architecture
MSPS	Mega Samples Per Second
OE	Operating Environment
OFDM	Orthogonal Frequency Division Multiplex
OMG	Object Management Group
OPB	On-chip Peripheral Bus
OSSIE	Open Source SCA Implementation: Embedded
PA	Power Amplifier
PC	Personal Computer
PGA	Programmable Gain Amplifier
PHY	Physical Layer
PIM	Platform Independent Model
PLCP	Physical Layer Convergence Protocol
PLL	Phase Locked Loop

PMR	Public Mobile Radio
PN	Pseudo Noise
POSIX	Portable Operating System Interface
PPDU	Physical Protocol Data Unit
PSDU	Physical layer Service Data Unit
PSM	Platform Specific Model
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase-Shift Keying
RCPC	Rate-Compatible Punctured Convolutional
RF	Radio Frequency
RFM	Radio Frequency Module
RRC	Root Raised Cosine
SCA	Software Communications Architecture
SDR	Software Defined Radio
SDRAM	Synchronous Dynamic Random Access Memory
SFF SDR DP	Small Form Factor SDR Development Platform
SIMD	Single-Instruction Multiple-Data
SNR	Signal-to-Noise Ratio
SoC	System on Chip
SRC	Sample Rate Conversion
STS	Short Training Sequence
SVFuA	Streitkräftegemeinsame Verbundfähige Funkgeräteausstattung
TCH	Traffic CHannel
TCL	Tool Command Language
TDD	Time Division Duplex
TDMA	Time Division Multiple Access

TETRA	Terrestrial Trunked Radio
TI	Texas Instruments
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
V+D	Voice and Data
VCO	Voltage Controlled Oscillator
VHDL	Very High Speed Integrated Circuit (VHSIC) HDL
VHF	Very High Frequency
VHSIC	Very High Speed Integrated Circuit
VPBE	Video Processing Back End
VPFE	Video Processing Front End
VPSS	Video Processing Sub-System
VS	Visual Studio
WINTSEC	WIreless INTeroperability for SECurity
WLAN	Wireless Local Area Network
XML	eXtensible Markup Language

Bibliography

[1] GSM Association and Europe Technologies, "GSM world coverage 2009," http://78.46.55.237/pubs/GSM_WorldPoster2009A.pdf, This is an electronic document. Date retrieved: March 10, 2011.

[2] Mary Bellis, "Inventors of the Modern Computer: Intel 4004 - The World's First Single Chip Microprocessor," http://inventors.about.com/od/mstartinventions/a/microprocessor.htm, This is an electronic document. Date retrieved: December 31, 2010.

[3] Noam Levine David Skolnick, "Why Use DSP? Digital Signal Processing 101," *Analog Dialogue*, vol. 31, no. 1, 1997.

[4] Xilinx, "Our History," http://www.xilinx.com/company/history.htm, This is an electronic document. Date retrieved: December 31, 2010.

[5] Ulrich Rohde, "Digital HF Radio: A Sampling of Techniques," *HAM Radio Magazine*, April 1985.

[6] Joseph Mitola, "The Software Radio," in *IEEE National Telesystems Conference*, 1992.

[7] Stefan Nagel, Friedrich Jondral, "Software Defined Radio in Public and Governmental Security Systems," *Micromaterials and Nanomaterials*, , no. 10, pp. 94–97, 2009.

[8] Stefan Nagel, Volker Blaschke, Friedrich Jondral, Eric Nicollet, Dominique Ragot, "Wireless Interoperability for Security - WINTSEC," in *Proceedings of the Software Defined Radio Forum Technical Conference*, Denver CO, November 2007.

[9] Steve Muir, "Trends in the Evolving Software Defined Radio (SDR) Market Landscape," http://www.vanu.com/documents/technology/trends-in-evolving-software-defined-radio.pdf, This is an electronic document. Date retrieved: December 31, 2010.

[10] Lutz Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, October 2000.

[11] "The Computer Language Benchmark Game," http://shootout.alioth.debian.org/fastest-programming-language.php, 2010, This is an electronic document. Date retrieved: October 8, 2010.

[12] Will Strauss, "DSP/Wireless market analyses," http://www.dsp-fpga.com/articles/id/?2547, 2006, This is an electronic document. Date retrieved: October 8, 2010.

[13] Robert Jan Ridder, "Programming digital signal processors with high-level languages," *DSP Engineering*, Summer 2000.

[14] Alan Anderson, "Programming and optimizing C code," http://www.eetimes.com/design/automotive-design/4017021/Programming-and-optimizing-C-code-part-1, 2007, This is an electronic document. Date of publication: February 20, 2007. Date retrieved: October 7, 2010.

[15] Texas Instruments, *TMS320C6000 Optimizing C Compiler User's Guide*, 2010, http://focus.ti.com/lit/ug/spru187q/spru187q.pdf.

[16] AMD, *Software Optimization Guide for AMD64 Processors*, 2005, http://support.amd.com/Embedded_TechDocs/25112.pdf.

[17] Michael E. Lee, "Optimization of Computer Programs in C," http://leto.net/docs/C-optimization.php, 1997, This is an electronic document. Date retrieved: October 14, 2010.

[18] Jim Xochellis, "The impact of the Pareto principle in optimization," http://www.codeproject.com/Articles/49023/The-impact-of-the-Pareto-principle-in-optimization.

aspx, January 2010, This is an electronic document. Date retrieved: October 14, 2010.

[19] Uwe Meyer-Baese, *Digital Signal Porcessing with Field Programmable Gate Arrays*, Springer, 3rd edition, 2007.

[20] Douglas J. Smith, "VHDL & Verilog Compared & Contrasted Plus Modeled Example Written in VHDL, Verilog and C," http://www.angelfire.com/in/rajesh52/verilogvhdl.html, 2003, This is an electronic document. Date retrieved: October 25, 2010.

[21] Anke Kamp, Frank Schmidt, Florian Thiem, "VHDL vs Verilog," 2007/2008.

[22] Eric Blossom, "GNU Radio: Tools for Exploring the RF Spectrum," *Linux Journal*, vol. 122, June 2004.

[23] GNU Radio, "Installation on the Play Station 3," http://gnuradio.org/redmine/wiki/gnuradio/BuildGuide, This is an electronic document. Date retrieved: January 5, 2011.

[24] Philip Balister, "Installation on the Play Station 3," http://www.opensdr.com/, This is an electronic document. Date retrieved: January 5, 2011.

[25] Gordon E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April 1965.

[26] Gordon E. Moore, "Excerpts from A Conversation with Gordon Moore: Moore's Law," ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf, 2005, This is an electronic document. Date retrieved: October 26, 2010.

[27] Carlos R. Aguayo González, Carl B. Dietrich, and Jeffrey H. Reed, "Understanding the software communications architecture," *Comm. Mag.*, vol. 47, no. 9, pp. 50–57, 2009.

[28] Michi Henning, "The rise and fall of CORBA," *Communications of the ACM*, vol. 51, no. 8, pp. 52–57, 2008.

[29] "Software Communications Architecture Specification Version 2.2.2," http://sca.jpeojtrs.mil/, May 2006, This is an electronic document. Date retrieved: October 26, 2010.

[30] Communications Research Center, CRC, "SCARI Software Suite," http://www.crc.gc.ca/en/html/crc/home/research/satcom/rars/sdr/products/scari_suite/scari_suite, This is an electronic document. Date retrieved: October 26, 2010.

[31] Prismtech, "SPECTRA," http://www.prismtechnologies.com/spectra, This is an electronic document. Date retrieved: October 26, 2010.

[32] Zeligsoft, "Component Enabler," http://www.zeligsoft.com/tools/zeligsoft-ce, This is an electronic document. Date retrieved: October 26, 2010.

[33] Wireless@Virginia Tech, "OSSIE: Open Source SCA Implementation - Embedded," http://ossie.wireless.vt.edu/, This is an electronic document. Date retrieved: October 27, 2010.

[34] M. Carrick, S. Sayed, C. Dietrich, and J. Reed, "Integration of FPGAs into SDR via Memory-Mapped I/O," in *Proceedings of the SDR Forum Technical Conference 2009*, Dec 2009.

[35] Philip Balister, "A Software Defined Radio Implemented using the OSSIE Core Framework Deployed on a TI OMAP Processor," M.S. thesis, Virginia Tech, Wireless@VT, VA, Dec 2007.

[36] "MDA Guide Version 1.0.1," http://www.omg.org/mda/, June 2003, This is an electronic document. Date retrieved: October 28, 2010.

[37] T. Langguth and H. Schober, "SDR based Waveform Development," in *Proc. 5th Karlsruhe Workshop on Software Radios*, 2008, pp. 109–114.

[38] S. Nagel, D. Epple, and F. K. Jondral, "Implementing the TETRA Physical Layer on Lyrtech's SFF SDR," in *Proc. SDR Forum Technical Conference*, 2008.

[39] "Simulink User's Guide," 2010, http://www.mathworks.com/help/toolbox/simulink/.

[40] "Real-Time Workshop User's Guide," 2010, http://www.mathworks.com/help/toolbox/rtw/.

[41] "Real-Time Workshop Embedded Coder User's Guide," 2010, http://www.mathworks.com/help/toolbox/ecoder/.

[42] "HDL-Coder User's Guide," 2010, `http://www.mathworks.com/help/toolbox/slhdlcoder/`.

[43] Stefan Nagel, Michael Schwall, Friedrich K. Jondral, "Performance Overhead of High-Level Waveform Development," in *Proceedings of the SDR Forum 2010 European Reconfigurable Radio Technologies Workshop*, Mainz, June 2010.

[44] M. Frigo and S.G. Johnson, "Fftw: an adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, May 1998, vol. 3, pp. 1381 –1384 vol.3.

[45] "FFTW Benchmark Results," `http://www.fftw.org/speed/`, This is an electronic document. Date retrieved: December 03, 2010.

[46] "Virtex 4 FPGA User Guide," `http://www.xilinx.com`, December 2008, This is an electronic document. Date retrieved: December 03, 2010.

[47] "Cyclone 2 Device Handbook, Volume 1," `http://www.altera.com`, 2008, This is an electronic document. Date retrieved: December 03, 2010.

[48] Jeffrey H. Reed, *Software Radio, A Modern Approach to Radio Engineering*, Prentice Hall, 2002.

[49] Anne Wiesler Friedrich Jondral, Ralf Machauer, *Software Radio, Adaptivität durch Parametrisierung*, J. Schlemmbach Fachverlag, 2002.

[50] Behzad Razavi, *RF Microelectronics*, Prentice Hall, 1998.

[51] André Dehon Scott Hauck, *Reconfigurable Computing: The theory and practice of FPGA-based computation*, Morgan Kaufmann Publishers, 2008.

[52] Bob Zeidman, *Designing with FPGAs and CPLDs*, CMP Books, 2002.

[53] "TMS320C6414, TMS320C6415, TMS320C6416, Fixed-Point Digital Signal Processors Data Sheet," `http://www.ti.com`, May 2005, This is an electronic document. Date retrieved: December 03, 2010.

[54] Matt Ettus, "Ettus research llc," `http://www.ettus.com`, March 2010, This is an electronic document. Date retrieved: December 03, 2010.

[55] "Cyclone FPGA Family Data Sheet," http://www.altera.com, March 2010, This is an electronic document. Date retrieved: December 03, 2010.

[56] "Data Sheet for the Mixed Signal Front-End Processor for Broadband Communications," http://www.analog.com, March 2010, This is an electronic document. Date retrieved: December 03, 2010.

[57] "Data Sheet for the CY7C68013 EZ-USB FX2 USB Microcontroller, High-Speed USB Peripheral Controller," http://www.cypress.com, March 2010, This is an electronic document. Date retrieved: December 03, 2010.

[58] "Data Sheet for the 50 MHz to 1000 MHz Quadrature Demodulator AD8348," http://www.analog.com, 2005, This is an electronic document. Date retrieved: December 03, 2010.

[59] "Data Sheet for the 0.8 GHz to 2.7 GHz Direct Conversion Quadrature Demodulator AD8347," http://www.analog.com, 2005, This is an electronic document. Date retrieved: December 03, 2010.

[60] "Data Sheet for the 140 MHz to 1000 MHz Quadrature Modulator AD8345," http://www.analog.com, 2005, This is an electronic document. Date retrieved: December 03, 2010.

[61] "Small Form Factor SDR Evaluation Module/ Development Platform User's Guide," http://www.lyrtech.com, October 2007, This is an electronic document. Date retrieved: December 03, 2010.

[62] "TMS320DM6446 Digital Media System-on-Chip Data Sheet," http://www.ti.com, March 2008, This is an electronic document. Date retrieved: December 03, 2010.

[63] "Datasheet for the 14-bit, 125 MSPS Analog-to-Digital Converter ADS5500," http://www.ti.com, February 2007, This is an electronic document. Date retrieved: December 03, 2010.

[64] "Datasheet for the 16-bit, 500 MSPS 2x–8x Interpolating Dual-Channel Digital-to-Analog Converter DAC5687," http://www.ti.com, September 2006, This is an electronic document. Date retrieved: December 03, 2010.

[65] "Datasheet for the 1.2 GHz Clock Distribution IC AD9511 with PLL Core, Dividers, Delay Adjust, Five Outputs," `http://www.analog.com`, June 2005, This is an electronic document. Date retrieved: December 03, 2010.

[66] "Datasheet for the TRF3701 0.6 GHz to 1.0 GHz Quadrature Modulator," `http://www.ti.com`, February 2003, This is an electronic document. Date retrieved: December 03, 2010.

[67] "Datasheet for the TRF1115 2.5 GHz, High Dynamic Range, Low-Noise Down-Converter," `http://www.ti.com`, September 2006, This is an electronic document. Date retrieved: December 03, 2010.

[68] "Datasheet for the TRF1112 Dual VCO/PLL Synthesizer with IF downconversion," `http://www.ti.com`, December 2005, This is an electronic document. Date retrieved: December 03, 2010.

[69] "Datasheet for the TRF1121 Dual VCO/PLL Synthesizer with IF Up-Converter," `http://www.ti.com`, December 2005, This is an electronic document. Date retrieved: December 03, 2010.

[70] "Datasheet for the TRF1122 2.5 GHz Integrated Up-Converter," `http://www.ti.com`, September 2006, This is an electronic document. Date retrieved: December 03, 2010.

[71] Martin Steppler, *Leistungsbewertung von TETRA Mobilfunksystemen durch Analyse und Emulation ihrer Protolle*, Ph.D. thesis, Department of Communication Networks (ComNets), Faculty 6, RWTH Aachen University, 2002.

[72] James Irvine John Dunlop, Demessie Girma, *Digital Mobile Communications and the TETRA System*, Wiley, 1 edition, 1999.

[73] ETSI, *Terrestrial Trunked Radio (TETRA); Voice plus Data; Part 2: Air Interface (AI)*, European Telecommunications Standards Institute, 3.2.1 edition, 2007.

[74] Karl-Dirk Kammeyer, *Nachrichtenuebertragung*, Teubner, 3 edition, 2004.

[75] Jens Elsner, Martin Braun, Stefan Nagel, Kshama Nagaraj, Friedrich K. Jondral, "Wireless Networks In-the-Loop: Software Radio as the Enabler," in *Proceedings of the Software Defined Radio Forum Technical Conference*, Washington DC, December 2009.

[76] M. Oerder and H. Meyr, "Digital filter and square timing recovery," *Communications, IEEE Transactions on*, vol. 36, no. 5, pp. 605 –612, May 1988.

[77] Martin Bossert, *Kanalcodierung*, Teubner, 2 edition, 1998.

[78] E. Hogenauer, "An economical class of digital filters for decimation and interpolation," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 29, no. 2, pp. 155 – 162, Apr. 1981.

[79] Altera, "Understanding CIC Compensation Filters," *Application Note*, vol. 455, Apr. 2007.

[80] T. Alberty and V. Hespelt, "A new pattern jitter free frequency error detector," *Communications, IEEE Transactions on*, vol. 37, no. 2, pp. 159 –163, Feb. 1989.

[81] Jack Volder, "The cordic computing technique," *Managing Requirements Knowledge, International Workshop on*, vol. 0, pp. 257, 1959.

[82] IEEE 802.11 Working Group, "IEEE Standard for Information technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Standard 802.11*, 2007.

[83] T.M. Schmidl and D.C. Cox, "Robust frequency and timing synchronization for ofdm," *Communications, IEEE Transactions on*, vol. 45, no. 12, pp. 1613 –1621, Dec. 1997.

[84] Stefan Nagel, Michael Schwall, Friedrich K. Jondral, "Porting of waveforms: Principles and implementation," *Frequenz*, vol. 64, no. 11-12, pp. 218 –223, nov/dec 2010.

[85] Stefan Nagel, Michael Schwall, Friedrich K. Jondral, "Portable Waveform Design," in *Proceedings of 20th Virginia Tech Symposium on Wireless Communications*, Blacksburg VA, June 2010.

[86] Arnon Friedmann, "TI's new TMS320C66x fixed and floating-point DSP core conquers the 'Need for Speed'," `http://focus.ti.com/lit/wp/spry147/spry147.pdf`, November 2010, This is an electronic document. Date retrieved: March 10, 2011.

[87] Jelena Nikolic-Popovic, "Using TMS320C6416 Coprocessors: Viterbi Coprocessor (VCP)," `http://focus.ti.com/lit/an/spra750d/spra750d.pdf`, September 2003, This is an electronic document. Date retrieved: March 10, 2011.

Supervised Theses

Dennis Epple	*Aufbau eines Demonstrators zum echtzeitfähigen TETRA-Empfang* (Diplomarbeit)
Michael Schwall	*Aufbau einer Übertragungsstrecke mit einem Small Form Factor SDR* (Studienarbeit)
Jan Herzmann	*Aufbau eines Senders nach dem IEEE 802.11a Standard auf einem Small Form Factor SDR* (Studienarbeit)
Patrick Schuster	*Erweizerung einer echtzeitfähigen TETRA-Übertragungsstrecke* (Studienarbeit)
Michael Schwall	*Echtzeitfähige Implementierung einer Zeit- und Frequenzsynchronisation für eine OFDM-Übertragung* (Diplomarbeit)
Enno Klasing	*Realisierung eines UMTS-Empfängers in Simulink* (Studienarbeit)
Simon Meier	*Vergleich von High-Level Entwicklungsverfahren für FPGAs* (Studienarbeit)
Hendrik Brunst	*Umsetzung einer TETRA-Wellenform in Fixed Point Arithmetik* (Studienarbeit)
Enno Klasing	*Untersuchung des Optimierungspotentials von automatisch generiertem Code in der DSV* (Diplomarbeit)

Index

Sponsorship

The research visit at the Mobile and Portable Radio Research Group (MPRG) at the Virginia Polytechnic Institute and State University was supported in part by the Karlsruhe House of Young Scientists (KHYS) of the Karlsruhe Institute of Technology (KIT).

Curriculum Vitae

Personal Details

Date of birth:	September 6th, 1980
Place of birth:	Sinsheim, Germany
Citizenship:	German

Education

07/2006 - 06/2011:	Research Associate, Karlsruhe Institute of Technology
08/2009 - 10/2009:	Visiting Researcher, Virginia Tech
06/2001 - 06/2006:	Dipl.-Ing., Universität Ulm
09/1991 - 06/2000:	Anna Essinger Gymnasium, Ulm