

DAB

Software Receiver Implementation

Andreas Müller

Advisor: Michael Lerjen
Professor: Prof. Dr. Helmut Bölcskei
Spring Term 2008

Preface

Wireless standards are subject to rapid development and frequent changes, making old hardware obsolete at a high rate. On the other hand, the constantly increasing computational power of general purpose computers allows developers to move functionality away from hardware and specialized processors, to high-level software running on arbitrary general purpose processors. This leads to the idea of software defined radios, where as much of the signal processing as possible is done in software. While this idea is not new, CPUs have only recently become fast enough for practical implementations of wireless standards in software.

This semester thesis investigates the possibility of implementing a receiver for the Digital Audio Broadcasting ([DAB](#)) standard [1] in software, by using the GNU Radio toolkit – a free open-source framework for the creation of software defined radios.

Outline

The report is split into the following chapters:

Chapter Introduction presents the project and some background.

Chapter DAB gives an overview of the [DAB](#) standard.

Chapter Software Defined Radio gives a brief introduction into Software Defined Radio.

Chapter GNU Radio and the USRP introduces the GNU Radio toolkit and the Universal Software Radio Peripheral ([USRP](#)).

Chapter Implementation describes the implementation of DAB in GNU Radio.

Chapter Conclusions and Outlook reviews the project and draws some conclusions.

Author: Andreas Müller andrmuel@ee.ethz.ch
Supervisor: Michael Lerjen mlerjen@nari.ee.ethz.ch
Professor: Prof. Dr. Helmut Bölcskei boelcskei@nari.ee.ethz.ch

Acknowledgements

I would like to thank my supervisor, Michael Lerjen, for his continuous support with helpful comments and ideas throughout the project. I would also like to thank the Communication Technology Laboratory ([CTL](#)) for allowing me to pursue this project, despite the fact that the [CTL](#) does currently not make any use of GNU Radio.

I would further like to thank Jens Elsner for helpful tips and for allowing me to gain from his experience from his own experiments with [DAB](#) and GNU Radio. This helped in avoiding some pitfalls, such as the problems with signals from the TVRX daughterboard. Jens also provided the contact to Nicolas Alt, who allowed me to take a look at his complete [DAB](#) receiver implementation in Matlab. This was very helpful, both from a design perspective, and because the comparison with my own implementation allowed me to verify each step.

Furthermore, I would like to thank the ISG, and especially David Schneider, for providing GNU Radio on the Tardis machines and for hosting the Subversion ([SVN](#)) repository used for this project.

Last but not least, I would like to thank the GNU Radio developers, for providing a very interesting toolkit, that allowed me to learn about software radios in a very practical manner and experiment with real-time [DAB](#).

Contents

1	Task Description	1
2	Introduction	6
2.1	Project Idea	6
2.2	Software Framework	6
2.3	Contributions	7
2.4	Some Words on Notation	7
3	DAB	8
3.1	Introduction	8
3.2	DAB Modes	9
3.3	Fast Information Channel (FIC)	9
3.3.1	FIB Assembler	9
3.3.2	Energy Dispersal	9
3.3.3	Convolutional Coding	9
3.4	Main Service Channel (MSC)	10
3.5	Frame Assembly and OFDM Modulation	11
3.5.1	Frames	11
3.5.2	OFDM Modulation	11
3.5.3	Complete Signal	12
3.6	Practical Considerations	14
3.6.1	Availability of DAB in Switzerland	14
3.6.2	Audio Quality	14
4	Software Defined Radio	15
4.1	Ideal Software Radio	15
4.2	Practical Software Radio	16
4.2.1	Analog-Digital Converters	16
4.2.2	Bus Speed	16
4.2.3	Performance of the Processing Unit	17
4.2.4	Latency	17
4.3	Review and Outlook	17
5	GNU Radio and the USRP	18
5.1	GNU Radio	18
5.1.1	GNU Radio Architecture	19
5.1.2	GNU Radio Companion	21
5.2	The USRP	22
5.2.1	USRP Architecture	23
5.2.2	Using the USRP in a GNU Radio Application	25
5.3	Wrap Up	25

6	Implementation	26
6.1	Setup	26
6.2	Physical Layer (OFDM)	27
6.2.1	Input Filtering	28
6.2.2	Time Synchronisation	28
6.2.3	Fine Carrier Frequency Synchronisation	30
6.2.4	OFDM Sampler	33
6.2.5	FFT	34
6.2.6	Coarse Carrier Frequency Synchronisation	34
6.2.7	Phase Differentiation	34
6.2.8	Removal of the Phase Reference Symbol	35
6.2.9	Sampling Frequency Offset Estimation and Resampling	35
6.2.10	Magnitude Equalization	37
6.2.11	Frequency Interleaving	37
6.2.12	Symbol Demapping	38
6.3	Evaluation of the Physical Layer	38
6.3.1	The Test Setup	38
6.3.2	BER in AWGN Channel	38
6.3.3	Effects of Coarse Carrier Frequency Offsets	39
6.3.4	Effects of Fine Carrier Frequency Offsets	41
6.3.5	Effects of Sampling Frequency Offsets	42
6.3.6	Effects of Multipath Propagation	43
6.3.7	Evaluation of the Processing Speed	43
6.3.8	Receiving a Live Signal	45
6.4	Fast Information Channel (FIC)	45
6.4.1	FIC Symbol Selection and Repartitioning	45
6.4.2	Convolutional Coding	46
6.4.3	Energy Dispersal Scrambling	47
6.4.4	FIB Sink	47
7	Conclusions and Outlook	48
7.1	Conclusions	48
7.2	Outlook	48
A	Signal Flow Graphs	50
B	FFTW Speed Evaluation	55
C	Additional Tools	57
C.1	OProfile	57
C.2	Python, IPython and SciPy	57
C.3	Doxygen	58
C.4	Swig	58
C.5	Graphviz, dot and dump2dot	58
C.6	Subversion	58
D	Code Overview	59
D.1	Python Code	59
D.1.1	Quality Assurance	60
D.1.2	Channel Tests	60
D.2	C++ Code	61
D.3	Patches	61

E Installation	62
E.1 Operating System	62
E.2 Packages	62
E.3 External Dependencies	62
E.4 Installation	62
F Presentation	64
G CD-ROM Content Listing	72
H Acronyms	73
Bibliography	76

List of Figures

3.1	Block diagram of a DAB transmitter.	8
3.2	Fast Information Channel according to DAB specification.	9
3.3	Main Service Channel according to DAB specification.	10
3.4	DAB transmission frame.	11
3.5	QPSK constellation used for the PRS.	12
3.6	QPSK constellation used for the other symbols.	12
4.1	Ideal SDR receiver (top) and transmitter (bottom).	16
5.1	Screenshot of the running example application.	22
5.2	GRC with a complete FM broadcast receiver.	22
5.3	The USRP with 4 daughterboards.	23
5.4	High level block diagram of the USRP.	24
5.5	USRP block diagram	24
6.1	Block diagram of the OFDM demodulation.	27
6.2	Null symbol detection	28
6.3	Null symbol detection.	29
6.4	Fine carrier frequency synchronisation.	31
6.5	Fine carrier frequency synchronisation.	32
6.6	Corrected fine carrier frequency offset	32
6.7	Phase jumps because of insufficient fine frequency synchronisation.	33
6.8	No more phase jumps because of averaging.	33
6.9	Plot of partially synchronised symbols.	35
6.10	Plot of the phase over the subcarrier number.	35
6.11	Estimated phase offset per carrier after 10, 50, 100 and 500 symbols, with $\alpha = 0.01$	36
6.12	BER in a noisy channel (tested with 1MB data per transmission).	39
6.13	BER in channel with coarse carrier frequency offset.	40
6.14	BER with coarse carrier frequency offsets (100kB data blocks).	40
6.15	Fine carrier frequency offset without correction (simulated with 10MB data blocks).	41
6.16	Sampling frequency offset – no correction (1MB data packets).	42
6.17	Sampling frequency offset with correction (1MB data packets).	42
6.18	Effect of multipath propagation (simulation runs with 500kB data packets).	43
6.19	Effect of taps with different magnitudes (500kB data packets).	44
6.20	DAB constellation sink with samples from the Swiss DAB ensemble at 227.36 MHz.	46
6.21	Station labels extracted from a DAB signal.	47
A.1	First version of OFDM demodulation.	51
A.2	Revised version of OFDM demodulation: With resampling, magnitude equalization and a single block for fine frequency estimation.	52
A.3	DAB OFDM test bench: Modulation, channel model and demodulation.	53

A.4 Complete receiver with USRP as signal source, OFDM demodulation and FIC decoding.	54
B.1 FFT speed evaluation.	55

List of Tables

3.1	DAB parameters and variables	13
3.2	DAB signal parameters.	13
5.1	Parameters of the D/A and A/D converters in the USRP.	24
6.1	Classes of sample streams used for testing.	27
6.2	Cyclic prefix length.	43
B.1	FFT runtime evaluation results	56

Chapter 1

Task Description

The following pages reflect the task description, as provided by Michael Lerjen.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Communication Technology Laboratory

Communication Theory Group
Prof. Dr. H. Bölcskei
Sternwartstrasse 7
CH-8092 Zürich

Semester Thesis Project in Information Technology and Electrical Engineering

Spring Semester 2008

for
Andreas Müller

DAB receiver implementation

Supervisors: Michael Lerjen (mlerjen@nari.ee.ethz.ch),
Hand-out date: 2008-03-03
Due date: 2008-06-06

Please do not distribute any copies of this project description
before the end of your project

1 Introduction

1.1 DAB

Since many years, analog systems are more and more replaced by their digital successors. This trend also embraces the electronic mass media. As a new digital radio standard, Digital Audio Broadcasting (DAB) has been developed in the EU project Eureka-147 between 1987 and 2000. It was standardized by the European Telecommunications Standards Institute (ETSI) [1] and the broadcast infrastructure is now more and more implemented mainly in Europe but also in Canada and China. A total of 500 mio customers worldwide can already receive DAB signals at their homes.

DAB is based on state-of-the-art techniques like OFDM, convolutional coding, data compression and also offers extra services like program information, traffic information, multimedia object transmission and conditional access.

1.2 GNU Radio

The GNU Radio project [3] develops free hardware and software for a software defined radio (SDR) system. The hardware and the software of this project is based on flexible building blocks and software libraries, which can be configured to implement receivers and transmitters for different radio standards.

2 Project Goals

- Based on the DAB standard [1] and further literature, a concept shall be worked out how the GNU Radio hardware and software together with MATLAB can be employed to build a real-time DAB receiver.
- Identify possible problems and system bottlenecks and analyze possible solutions.
- Implement building blocks for a real-time DAB receiver.

3 Project Tasks

The tasks given represent work items and questions we see at the start of the project. This tentative list is mainly to serve as an orientation and can be modified and extended, depending on the project progress.

- Get familiar with the DAB standard.

- Analyze the performance and complexity of different algorithms for the sub-blocks of the system.
- Implement blocks in Matlab or C++ and compare their performance and processing complexity. Always optimize the system for real-time usability.
- Implement a demonstrator application that can be presented.
- Documentation and Presentation.

4 Project organization

The duration of the thesis is 14 weeks.

- Set up a project plan and track progress continuously.
- Organize weekly meetings with your thesis supervisors. The meetings will be held to evaluate the status of the project and to discuss potential problems.
- Document your work in a thesis report. We recommend to use \LaTeX and to start the documentation at the beginning of the project.

5 End of Project

The due date is Friday, June 6, 2008.

Two copies of the thesis report need to be handed in, containing also a CD/DVD with all relevant files in a clean and documented directory structure. The copies remain property of the IKT.

At the end of the thesis, there will be a short presentation of the project and its results (15 min presentation and 5 min Q&A) to a wider audience.

Please note that the thesis will only be accepted when the keys for the ETZ building are properly returned.

Zürich, April 3, 2008

Prof. Dr. H. Bölcskei

References

- [1] ETSI Standard, "ETSI EN 300 401 V1.4.1," European Standard (Telecommunications series)
- [2] WWW, "WorldDAB Standards and Technical Specifications Homepage," http://www.worlddab.org/technology/standards_specs
- [3] WWW, "GNU Radio homepage," <http://gnuradio.org/trac>

Chapter 2

Introduction

2.1 Project Idea

The goal of this semester thesis is to implement a complete real-time software receiver for **DAB**, a digital radio technology standardized by the European Telecommunications Standards Institute (**ETSI**), which will be introduced in Chapter 3.

DAB is mainly targeted at the Very High Frequency (**VHF**)¹ and Ultra High Frequency (**UHF**)² band; for lower frequencies, a similar technology called Digital Radio Mondiale (**DRM**)³ exists. While several free and open-source **DRM** receiver implementations, such as Dream⁴ or Diorama⁵ exist, no open software implementation of a real-time **DAB** receiver is currently available, even though **DAB** is older than **DRM** [3, 4]. An explanation can be found in the used bandwidth: While **DRM** uses a bandwidth of less than 20 kHz, **DAB** occupies 1.537 MHz. Because of the small bandwidth, **DRM** can be sampled without complicated hardware (a common sound card is in fact sufficient [5]). A **DAB** signal on the other hand can not be sampled without appropriate hardware (section 5.2 will introduce a suitable device), and the higher bandwidth makes real time processing much more difficult.

Although a software implementation is obviously less efficient than the use of dedicated hardware, it will be shown that a software implementation still has several advantages, one of which is its very high flexibility. In the case of **DAB**, the argument for a software implementation is supported by the **DAB+** standard, which replaces the current **DAB** standard (in Switzerland, **DAB+** will be introduced in 2009⁶). While this renders many hardware receivers useless, a software receiver can be updated easily.

2.2 Software Framework

The software framework used during the development of the presented **DAB** receiver implementation is GNU Radio, an open-source Software Defined Radio (**SDR**) toolkit. GNU Radio will be introduced in Chapter 5.

¹30-300 MHz

²300-3000 MHz

³The DRM specification is available at the ETSI website as well [2].

⁴Available at <http://drm.sourceforge.net/>.

⁵Available at <http://nt.eit.uni-kl.de/forschung/diorama/>.

⁶More information is available at <http://www.dab-digitalradio.ch/>.

2.3 Contributions

An inquiry on the GNU Radio mailing list⁷ about Orthogonal Frequency Division Multiplexing (OFDM) and DAB revealed, that Jens Elsner had already created an implementation of the DAB physical layer (OFDM) some time ago. Unfortunately, his implementation is not compatible with some of the newer developments in GNU Radio. Additionally, one goal for the presented implementation was to use independent modular signal processing blocks (which results in simpler blocks with better reusability), whereas Jens' receiver is mainly implemented as one big block (which tends to result in faster code and allows the use of feedback loops). For this reasons, none of the code written by Jens is reused in this thesis. The code was however still interesting as a reference. Jens also provided some DAB samples and shared his experience from experiments with the USRP. Especially his warnings about problems with the Television Receiver (TVRX)⁸ saved me from wasting a lot of time and energy. Additionally, Jens' student thesis [6] provided a lot of interesting and relevant information.

GNU Radio itself also includes some code for OFDM. Although this code is not DAB specific, part of this code was used as a basis for the DAB implementation, namely the code for fine carrier frequency synchronisation and part of the code for coarse carrier frequency synchronisation. Additionally, one OFDM block, the cyclic prefixer, could be used directly without any adjustments. As DAB uses an OFDM scheme that is fundamentally different from the one used by the GNU Radio OFDM code, most of the existing OFDM blocks could however not be reused. On the other hand, many non OFDM-specific blocks from GNU Radio were very useful, such as the Fast Fourier Transform (FFT) block, Finite Impulse Response (FIR) filter blocks, the Trellis module, various blocks for mathematical operations and numerous low-level blocks.

Finally, Jens Elsner also provided a contact to Nicolas Alt, who has implemented a complete DAB receiver in Matlab, which he kindly shared with Jens and me. Although his implementation is designed as an off line DAB receiver, having a working software model and "known to work" samples was highly useful to verify each step of the GNU Radio implementation, especially for the Fast Information Channel (FIC) decoder. Additionally, the idea to use the distance between two Null symbols to get an estimate for the sampling frequency was adopted from Nicolas' code.

2.4 Some Words on Notation

The following notation is used throughout this semester thesis:

- Function names, class names, file names and executable commands are printed in a **monospace** font
- Functions are additionally designated by brackets, e.g. `foo_xy()`

⁷<http://lists.gnu.org/mailman/listinfo/discuss-gnuradio>

⁸The TVRX is a daughterboard for the USRP; more on this will be explained in Chapter 5.2.

Chapter 3

DAB

3.1 Introduction

DAB is a digital radio technology, which was developed between 1981 and 1993 and standardized by ETSI in 1997. DAB makes use of various advanced technologies, such as OFDM, convolutional coding and audio coding. Besides an audio signal, DAB can also be used to transmit additional data, such as program information, traffic information, paging services or even complete HTML pages [3].

This chapter aims to be a basic introduction into the technical details of DAB. The full DAB specification is available for free on the ETSI website [1]. Please note that this introduction describes the transmit path, as does the specification. The chapter Implementation (Chapter 6) on the other hand is written from a receiver perspective.

Figure 3.1, which is a simplified version of the block diagram shown in Section 4 on Page 22 of the DAB specification [1], shows the general design of a DAB transmitter.

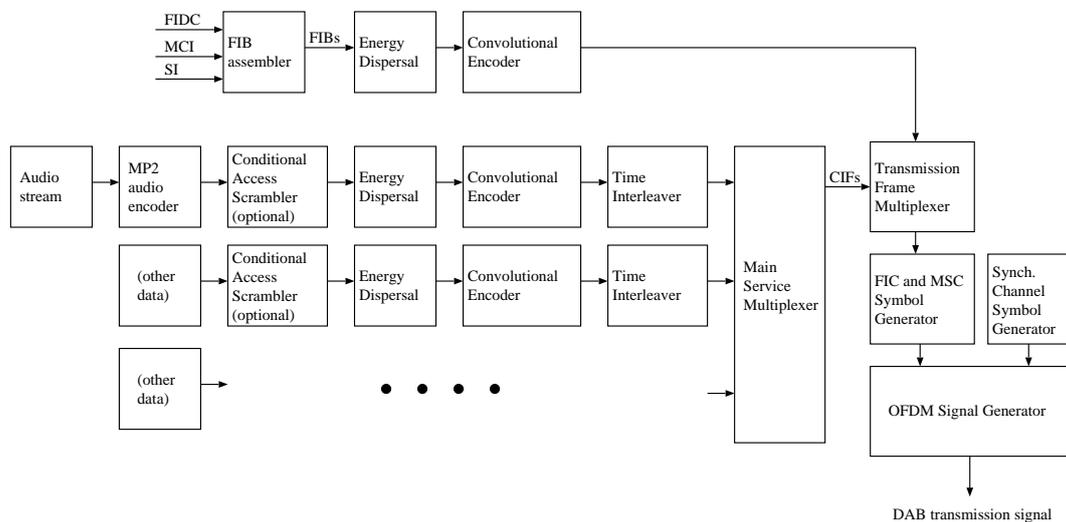


Figure 3.1: Block diagram of a DAB transmitter.

3.2 DAB Modes

The **DAB** specification defines four sets of parameters for four different modes. Although this affects the **FIC** and the Main Service Channel (**MSC**) as well, the main differences between the modes are on the physical layer.

3.3 Fast Information Channel (FIC)

The **FIC** is made up of Fast Information Blocks (**FIBs**), which contain control information, such as Multiplex Configuration Information (**MCI**) (information about the association of data with services), Service Information (**SI**) (such as radio station names) or Conditional Access (**CA**) information (indication of whether and how content is scrambled).

The information carried in the **FIC** is generally information, that is needed by the receiver to be able to interpret the other channels. As this information is particularly important, the **FIC** uses a high level of error protection. The basic scheme of the **FIC** is shown in Figure 3.2.

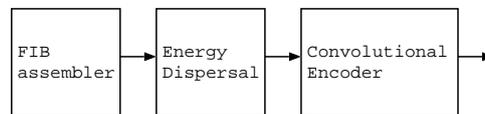


Figure 3.2: Fast Information Channel according to DAB specification.

3.3.1 FIB Assembler

The **FIB** assembler packs up the information of the **FIC** into packets of size 32 bytes, so called **FIBs**. Each **FIB** contains 30 bytes of data and a 16 bit Cyclic Redundancy Check (**CRC**) sum, which allows the receiver to verify correct decoding. The **CRC** is calculated with the polynomial

$$G(x) = x^{16} + x^{12} + x^5 + 1$$

The specification of **FIBs** does of course also include a lot of protocol details, such as information about the different types of **FIBs**, where in the **FIB** which information can be found, etc. For the sake of brevity, this information is completely omitted here. It can be found mainly in Section 5 of the **DAB** standard.

3.3.2 Energy Dispersal

The energy dispersal block asserts that there are no unwanted patterns in the data, as this might result in an output signal with discontinuous energy.

For this purpose, a Pseudo Random Binary Sequence (**PRBS**) generated by the Linear Feedback Shift Register (**LFSR**) with the polynomial

$$P(x) = x^9 + x^5 + 1$$

with initial state of all registers set to one is added modulo 2 (i.e. XORed) to the data.

As the XOR operation is symmetric, the energy dispersal block is exactly the same in the transmit and receive path.

3.3.3 Convolutional Coding

To protect the data against errors, coding is applied. Coding is split into two parts – first, the data is coded and secondly, puncturing is applied.

Convolutional Code

As specified in Section 11.1.1 of the [DAB](#) standard, the encoder consists of a Finite State Machine ([FSM](#)) with 64 states (the code has constraint length 7, i.e. the current bit and 6 past bits are used). The [FSM](#) produces four output bits from one input bit. The bits are created with the following modulo 2 addition of input bits:

$$x_{0,i} = a_i \oplus a_{i-2} \oplus a_{i-3} \oplus a_{i-5} \oplus a_{i-6}$$

$$x_{1,i} = a_i \oplus a_{i-1} \oplus a_{i-2} \oplus a_{i-3} \oplus a_{i-6}$$

$$x_{2,i} = a_i \oplus a_{i-1} \oplus a_{i-4} \oplus a_{i-6}$$

$$x_{3,i} = a_i \oplus a_{i-2} \oplus a_{i-3} \oplus a_{i-5} \oplus a_{i-6}$$

Input bits outside the codeword are zero by definition.

Puncturing

Puncturing is applied to adjust the code rate. The process here is rather simple: Some predefined bits are deleted from the codeword, according to puncturing vectors. For different protection levels, resp. code rates, 24 different puncturing vectors are specified.

3.4 Main Service Channel (MSC)

The [MSC](#) transports Common Interleaved Frames ([CIFs](#)), which contain different Service Components ([SCs](#)). Depending on the service, a packet based or a stream based mode can be used. All services contained in the [MSC](#) are multiplexed according to the information carried in the [FIC](#). The main service contained in the [MSC](#) is stream based audio data.

The scheme of the [MSC](#) is shown in [Figure 3.3](#).

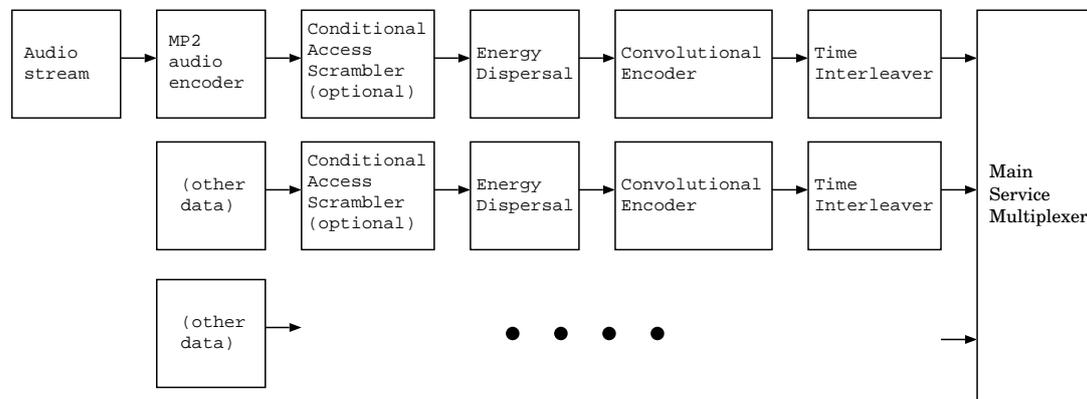


Figure 3.3: Main Service Channel according to DAB specification.

In case of an audio stream, the data is first encoded with the MPEG-1 Audio Layer II ([MP2](#)) codec. The data may then be scrambled, to prevent access by unauthorized users. Energy dispersal and convolutional encoding is then applied in the same way as in the [FIC](#) (though with different parameters).

Differently from the [FIC](#), the information is then interleaved in time. Mixing up the information in time serves to protect the data against short-time interferers across all frequencies.

Finally, the different information blocks from the [MSC](#) are multiplexed according to the information specified in the [FIC](#).

3.5 Frame Assembly and OFDM Modulation

3.5.1 Frames

The information from the FIC and the MSC is split among transmission frames, which have a duration of 96 ms¹. Each frame consists of a Null symbol, a Phase Reference Symbol (**PRS**) and 75 **OFDM** symbols containing **FIC** and **MSC** information. The structure of a **DAB** frame is shown in Figure 3.4.

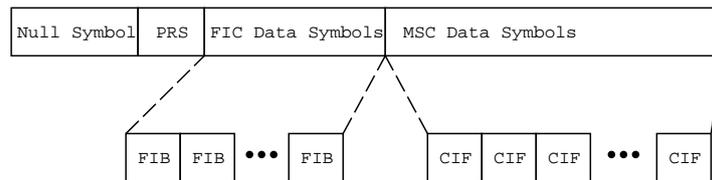


Figure 3.4: DAB transmission frame.

Null Symbol

For the duration of the Null Symbol, no signal is transmitted. This is done to indicate the separation of the frames.

Phase Reference Symbol

The **PRS** (sometimes also referred to as pilot symbol) is a predefined **OFDM** symbol, which serves as phase reference.

Data Symbols

The other symbols in the frame contain either **FIC** or **MSC** data. Please note that there is no one-to-one association between symbols and **FIB**s or **CIF**s.

3.5.2 OFDM Modulation

The **OFDM** signal² consists of 1536 frequency-interleaved subcarriers, each of which is individually modulated with $\frac{\pi}{4}$ Differential Quadrature Phase Shift Keying (**D-QPSK**).

QPSK Mapping

Two different Quadrature Phase-Shift Keying (**QPSK**) constellations are used.

The constellation shown in Figure 3.5 is used for the first symbol only, the **PRS**. For the other symbols, the constellation shown in Figure 3.6 is used.

Differential Modulation

To achieve differential modulation, each symbol is multiplied to the previous symbol before transmission. This means, that the phase of the current symbol is always added to the phase of the previous symbol, which results in output symbols with constellations alternating between

¹For better readability, this section always refers to **DAB** mode I. Parameters for all modes are collected in Table 3.2 on Page 13.

²As many texts about **OFDM** already exist, **OFDM** itself is not discussed here. A good introduction can be found in [7].

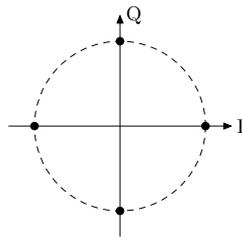


Figure 3.5: QPSK constellation used for the PRS.

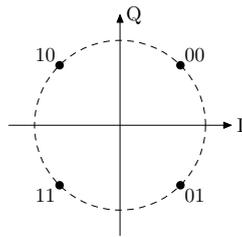


Figure 3.6: QPSK constellation used for the other symbols.

the constellation shown in Figure 3.6 and the one from 3.5. Since each constellation is shifted by $\frac{\pi}{4}$ relative to the previous one, this is called $\frac{\pi}{4}$ D-QPSK.

Frequency Interleaving

The individual OFDM subcarriers are mixed according to a static sequence defined in the DAB standard. Frequency interleaving prevents that too much information of the same codeword is lost if there is a small-bandwidth interferer.

3.5.3 Complete Signal

The complete DAB signal is specified in [1] as

$$s(t) = \Re \left\{ e^{2j\pi f_c t} \sum_{m=-\infty}^{\infty} \sum_{l=0}^L \sum_{k=-K/2}^{K/2} z_{m,l,k} \cdot g_{k,l}(t - mT_F - T_{Null} - (l-1)T_s) \right\}$$

with

$$g_{k,l}(t) = \begin{cases} 0 & \text{for } l = 0 \\ e^{2j\pi k(t-\Delta)/T_U} \cdot \text{Rect}(t/T_S) & \text{for } l = 1, 2, \dots, L \end{cases}$$

and $T_S = T_U + \Delta$.

The parameters and variables, as defined in [1], are listed in Table 3.1.

Please note that $z_{m,l,k} = 0$ for $k = 0$. This means that the central subcarrier is not used. The value $g_{k,l}(t) = 0$ for $l = 0$ on the other hand, represents the Null symbol.

The values for the parameters, which depend on the transmission mode, are given in Table 38 in the DAB specification [1]. For reference, the values are listed in Table 3.2. The elementary time period is $T = 1/2048000$ second.

The variables defined in Table 3.1 will be used throughout this text; as they depend on the used DAB mode, please refer to Table 3.2 for specific values.

Variable	Description
L	Number of OFDM symbols per frame, exclusive Null symbol
K	Number of subcarriers
T_F	Length of the frame without the Null symbol
T_{Null}	Length of the Null symbol
T_S	Length of an OFDM symbol different than the Null symbol
T_U	Inverse of the subcarrier spacing
Δ	Length of the guard interval
$z_{m,l,k}$	Complex D-QPSK symbol of carrier k of OFDM symbol l in transmission frame m
f_c	Central frequency of the OFDM signal

Table 3.1: DAB parameters and variables

Parameter	DAB Mode I	DAB Mode II	DAB Mode III	DAB Mode IV
L	76	76	153	76
K	1536	384	192	768
T_F	$196608T$ $= 96ms$	$49152T$ $= 24ms$	$49152T$ $= 24ms$	$98304T$ $= 48ms$
T_{Null}	$2656T$ $\approx 1.297ms$	$664T$ $\approx 324\mu s$	$345T$ $\approx 168\mu s$	$1328T$ $\approx 648\mu s$
T_S	$2552T$ $\approx 1.246ms$	$638T$ $\approx 312\mu s$	$319T$ $\approx 156\mu s$	$1276T$ $\approx 623\mu s$
T_U	$2048T$ $= 1ms$	$512T$ $= 250\mu s$	$256T$ $= 125\mu s$	$1024T$ $= 500\mu s$
Δ	$504T$ $\approx 246\mu s$	$126T$ $\approx 62\mu s$	$63T$ $\approx 31\mu s$	$252T$ $\approx 123\mu s$

Table 3.2: DAB signal parameters.

The different modes are specified to accommodate different frequency ranges and operating conditions. In Section 15.1 of the DAB specification [1], suitable conditions are described as follows (quote):

- Transmission mode I is intended to be used for terrestrial Single Frequency Networks (SFN) and local-area broadcasting in Bands I, II and III.
- Transmission modes II and IV are intended to be used for terrestrial local broadcasting in Bands I, II, III, IV, V and in the 1452 MHz to 1492 MHz frequency band (i.e. L-Band). It can also be used for satellite-only and hybrid satellite-terrestrial broadcasting in L-Band.
- Transmission mode III is intended to be used for terrestrial, satellite and hybrid satellite-terrestrial broadcasting below 3000 MHz.

One notable feature is the possibility to use Single Frequency Networks (SFNs) in mode I. This means, that multiple geographically separated transmitters broadcast at exactly the same frequency. This requires that the transmitters are accurately synchronised both in time and frequency to avoid interference at the receiver. Even with synchronized transmitters, there may however still be a delay between two signals from different transmitters arriving at the receiver, because of different path lengths. The effects of this are the same as from multipath propagation. The solution in the standard is to require that the maximum distance between

each pair of transmitters in an **SFN**, divided by the propagation speed, must be smaller than the length of the cyclic prefix. Like this, no signal arriving from any secondary transmitter will have a delay longer than the cyclic prefix, and the signal quality is not affected, as long as the magnitude of the delayed signal is not too large (a simulation of these effects is presented in Section 6.3.6).

3.6 Practical Considerations

3.6.1 Availability of DAB in Switzerland

In Switzerland, three **DAB** ensembles exist, each of which is an **SFN** with multiple transmitters. Each of the three ensembles serves one language region; the ensemble for the German speaking part of Switzerland can be found at 227.36 MHz.

As the runtime distance of signals from different transmitters in an **SFN** must not be longer than the cyclic prefix, it would not be possible to use one **SFN** for all of Switzerland – with a cyclic prefix of length 504 samples (mode I), the maximum distance between two transmitters in an **SFN** is

$$d_{max} = l_{cp} * T * c \approx 504 * \frac{1}{2048000} s * 300'000 km/s \approx 73.83 km$$

3.6.2 Audio Quality

Although **DAB** is theoretically superior to Frequency Modulation (**FM**), the audio quality is unfortunately severely limited because many broadcasters use reduced bit rates [3]. Together with an artificially increased loudness³ and dynamic range compression, this can reduce the audio quality severely. A closer look at these problems is beyond the scope of this thesis, but more information can be found in [9].

In the Swiss **DAB** ensembles, besides a very low bit rate, some of the channels are even sent in mono only⁴. Because of this, the audio quality of Swiss **DAB** transmissions is worse than the audio quality of **FM** radio stations.

³Many recording studios and broadcast radio stations artificially increase the loudness of their music for marketing reasons. This has led to the so called loudness war [8].

⁴A list of **DAB** ensembles in Switzerland with the broadcast channels and the transmitted quality can be found under <http://digiradio.ch/dab/programme/ch/index.html>.

Chapter 4

Software Defined Radio

A Software Defined Radio (**SDR**) or just Software Radio (**SR**)¹ is a radio that is built entirely or in large parts in software, which runs on a general purpose computer. A more extensive definition is given by Joseph Mitola, who coined the term Software Radio:

A software radio is a radio whose channel modulation waveforms are defined in software. That is, waveforms are generated as sampled digital signals, converted from digital to analog via a wideband DAC and then possibly upconverted from IF to RF. The receiver, similarly, employs a wideband Analog to Digital Converter (ADC) that captures all of the channels of the software radio node. The receiver then extracts, downconverts and demodulates the channel waveform using software on a general purpose processor. Software radios employ a combination of techniques that include multi-band antennas and RF conversion; wideband ADC and Digital to Analog conversion (DAC); and the implementation of IF, baseband and bitstream processing functions in general purpose programmable processors. The resulting software-defined radio (or "software radio") in part extends the evolution of programmable hardware, increasing flexibility via increased programmability. [10]

This means, that instead of using analog circuits or a specialized Digital Signal Processor (**DSP**) to process a radio signal, the digitized signal is processed by architecture independent high level software running on a general purpose processor. The term radio designates any device, that transmits and/or receives radio waves.

While most modern radios contain firmware that is written in some kind of programming language, the important distinction in a software radio is that it is not tailored to a specific chip or platform, and it is therefore possible to reuse its code across different underlying architectures.

4.1 Ideal Software Radio

In the ideal case, the only hardware that is needed besides a computer is an antenna and an Analog Digital Converter (**ADC**) for the receiver, as well as a Digital Analog Converter (**DAC**) for the transmitter. A **SR** would thus look as depicted in Figure 4.1.

In the receiver, a transmitted radio signal is picked up by an antenna, and then fed into an **ADC** to sample it. Once digitized, the signal is sent to some general purpose computer (e.g. an embedded PC) for processing.

The transmitter looks very similar, except that the signal is sent in the reverse direction, and a **DAC** is used instead of an **ADC**. In a complete transceiver, the processing unit and the antenna may be shared between receiver and transceiver.

¹This two terms are used interchangeably in this semester thesis.

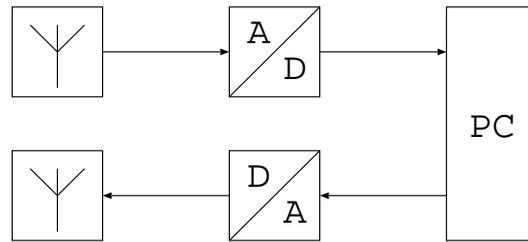


Figure 4.1: Ideal SDR receiver (top) and transmitter (bottom).

4.2 Practical Software Radio

While the approach presented in the previous section is very simple and (in the ideal case) extremely versatile, it is not practical, due to limitations in real hardware. However, various solutions have been suggested, e.g. in [11], to overcome these problems.

In the following sections, we take a quick look at the different hardware limitations. For better readability, only the receiving side is discussed. The transmitting side is symmetrical.

4.2.1 Analog-Digital Converters

The two parameters of interest in [ADCs](#) are sample rate and resolution.

The sample rate limits the maximum bandwidth of the received signal – according to Harry Nyquist and Claude Shannon, the sampling rate must be at least twice as high as the bandwidth.

Current [ADCs](#) are capable of sampling rates in the area of 100 Mega Samples Per Second ([MSPS](#)), which translates to a bandwidth of 50 MHz. While this bandwidth is enough for most current applications, the carrier frequency is usually higher than 50 MHz. In practice, a Radio Frequency ([RF](#)) front-end is therefore usually required, to convert the received signal to an Intermediate Frequency ([IF](#)).

The second parameter, the [ADC](#) resolution, influences the dynamic range of the receiver. As each additional bit doubles the resolution of the sampled input voltage, the dynamic range can be roughly estimated as

$$R = 6dB \cdot n$$

where R is the dynamic range and n the number of bits in the [ADC](#).

As [ADCs](#) used for [SDRs](#) usually have a resolution of less than 16 bits, it is important to filter out strong interfering signals – such as signals from mobile phones – before the wideband [ADC](#)². This is usually done in the [RF](#) front-end.

4.2.2 Bus Speed

Another problem lies in getting the data from the [ADC](#) to the computer. For any practical bus, there is a maximum for the possible data rate, limiting the product of sample rate and resolution of the samples.

The speed of common buses in commodity PCs ranges from a few MBit/s to several GBit/s; as an example, the Peripheral Component Interconnect ([PCI](#)) 2.2 bus has a theoretical maximum speed of 532MB/s.

²The other possibility would be to attenuate the whole signal, such that the interferer fits into the dynamic range of the [ADC](#); as the signal of interest would be attenuated as well, this would however mean that it would no longer be possible to use the whole dynamic range of the [ADC](#) for the signal of interest.

4.2.3 Performance of the Processing Unit

For real-time processing, the performance of the Central Processing Unit (CPU) and the sample rate limit the number of mathematical operations that can be performed per sample, as samples must – in average – be processed as fast as they arrive.

In practice, this means that fast CPUs, clever programming and possibly parallelization amongst different computers is needed. If this does not suffice, a compromise must be found, to use a less optimal but faster signal processing algorithm.

4.2.4 Latency

Since general purpose computers are not designed for real-time applications, a rather high latency can occur in practical SDRs.

While latency is not much of an issue in transmit-only or receive-only applications, many wireless standards, such as Global System for Mobile communications (GSM) or Digital Enhanced Cordless Telecommunications (DECT) require precise timing, and are therefore very difficult to implement in an SDR.

4.3 Review and Outlook

Because of the use of general purpose processing units, an implementation of a given wireless application as an SDR is likely to use more power and occupy more space than a hardware radio with analog filtering and possibly a dedicated signal processor. Because an SDR contains more complex components than a hardware radio, it will likely also be more expensive, given a large enough production volume.

While this looks quite grim for SDRs, they have one distinct advantage: Flexibility. Bringing the flexibility of software to the radio world introduces a number of interesting possibilities. For example, very much the same way as someone may load a word processor or an internet browser on a PC, depending on the task at hand, a SDR could allow its user to load a different configuration, depending on whether the user wants to listen to a broadcast radio transmission, place a phone call or determine the position via Global Positioning System (GPS). A new application may even be added after the device is finished. Since the same hardware can be used for any application, a great reuse of resources is possible.

Even more advantages, such as the rapid development process, are listed and explained in [11], where arguments can also be found, why the cost of an SDR compared to a traditional radio may even be reduced in certain cases.

Another interesting possibility enabled by SDR is the creation of a cognitive radio, which is aware of its RF environment and adapts itself to changes in the environment [12]. By doing this, a cognitive radio can use both the RF spectrum and its own energy resources more efficiently. As a cognitive radio requires a very high degree of flexibility, its creation is much easier, when an SDR is used.

Chapter 5

GNU Radio and the USRP

This chapter introduces the GNU Radio software toolkit and the associated hardware device, the [USRP](#).

5.1 GNU Radio

GNU Radio is a free software toolkit for implementing software defined radios. It is licensed under the GNU General Public License ([GPL](#)), which means that anyone is allowed to use, copy and modify GNU Radio without limits, provided that extensions are made available under the same license.¹

The goal of GNU Radio is to “bring the code as close to the antenna as possible”, and therefore “turn hardware problems into software problems”, according to the projects founder Eric Blossom [13].

To this end, GNU Radio provides numerous building blocks for signal and information processing, as well as a framework to control the data flow between them. By “wiring together” such building blocks, a user can create a software defined radio, similar to how one might connect physical RF building blocks to create a hardware radio. An incomplete list of the functions provided by GNU Radio includes:

- Mathematical operations (add, multiply, log, ...)
- Interleaving, delay blocks, ...
- Filters ([FIR](#), Infinite Impulse Response ([IIR](#)), Hilbert, ...)
- [FFT](#) blocks
- Automatic Gain Control ([AGC](#)) blocks
- Modulation and demodulation (FM, AM, PSK, QAM, GMSK, OFDM, ...)
- Interpolation and decimation
- Trellis and Viterbi support

Apart from signal processing blocks, GNU Radio also provides support for various signal sources and sinks, such as:

- Signal generators

¹A more detailed explanation and the complete license can be found under <http://www.gnu.org/licenses/gpl.html>.

- Noise generators
- Pseudo random number generators
- [USRP](#) source and sink (to transmit/receive signals via [USRP](#))
- Graphical sinks (Oscilloscope, [FFT](#), Waterfall, ...)
- Audio source and sink
- File source and sink (reads/writes samples from/to a file)
- User Datagram Protocol ([UDP](#)) source and sink (to transport samples over a network)

The GNU Radio project was founded by Eric Blossom and is now supported by numerous enthusiasts around the world. While GNU Radio was started on a Linux platform, it now supports various Unixes, and partially even Windows.

More information about GNU Radio can be found on the GNU Radio website [14]. An introduction to GNU Radio and its background can also be found in the wired magazine [15].

5.1.1 GNU Radio Architecture

Flow Graphs

Each GNU Radio application has at its core a flow graph. A flow graph is a directed graph, whose nodes can be either signal sources, sinks or processing blocks. Blocks have one or more ports, where edges can be connected. The edges represent the connections between the blocks, i.e. they determine the data flow. A block can have multiple incoming and/or outgoing ports. Blocks without incoming ports are called sources, and blocks without outgoing ports sinks.

One restriction for flow graphs is, that they may not contain loops. This does not mean, that signal processing algorithms with loops are impossible, but the part that contains a loop in the data flow must be implemented fully inside one block.

Data Format

The most common data format used to represent samples in GNU Radio blocks are complex floats – interleaved floats, of which the first is treated as real, and the second as imaginary part. This means, that each sample occupies 8 bytes. Other data formats are float (4 byte float values), short (2 byte integer values) and char (1 byte integer values).

These data types also exist as vector types, which is useful for blocks that operate on a fixed number of samples at once, e.g. [FFT](#) blocks.

The data format can often be recognized from the name of the block. If a block is named `foo_vXY`, the block operates on vectors (as `v` is present), has the input type `X` and the output type `Y`. The types can be

- `c` – complex interleaved floats
- `f` – floats (4 bytes)
- `s` – short integers (2 bytes)
- `b` – byte integers (1 byte)

For instance, the block `fft_vcc` is an [FFT](#) block that operates on vectors with complex interleaved floats.

Implementation

GNU Radio is implemented in C++ and Python, and uses SWIG² to create an interface between the two. While C++ is used for low-level programming, mainly for the core of the framework, to create low-level signal processing blocks and for hardware support, Python is used on a higher level, to connect signal processing blocks into an application, or the wrap multiple low level blocks into a higher level block. This approach allows the combination of the high speed of C++ and the ease of use of Python.

To create an application, a programmer derives her own class from either the class `top_block` of the module `gr` in the package `gnuradio` for a console application, or from the class `std_top_block` of the module `stdgui2` in the package `gnuradio.wxgui`³ for a GUI based application. Currently, this can only be done in Python; support for pure C++ applications, which may sometimes be preferable (e.g. for embedded systems), is under development.

Signal processing blocks can be either synchronous, which means that there is an integer relationship between the sample rate at input and output ports, or asynchronous. Synchronous blocks are classes that are usually derived from `gr_sync_block` (blocks with an 1:1 sample rate ratio), `gr_sync_interpolator` (1:N) or `gr_sync_decimator` (N:1). Asynchronous blocks are commonly directly derived from `gr_block`. Blocks are usually written in C++; there is however the possibility to wrap up several blocks into a higher level block. This can be done in Python by deriving a class from `gr.hier_block2`.

Once the desired blocks have been instantiated, they can be connected with the `connect` method of the main class in Python (i.e. the class which is derived from `top_block` or `std_top_block`, that represents the flow graph). This is the analog of drawing edges in the graph. To get a runnable graph, all ports of each block must be connected. Signal processing can then be started with the `start` method of the flow graph.

In a running GNU Radio application, the scheduler of the GNU Radio framework manages the blocks and the flow of samples between them, assisted by the `forecast()` methods of the blocks, which can be used by the programmer, to tell the scheduler, how many input samples a block requires to produce a given number of output samples.

The actual signal processing is done in the function `general_work()` (in the case of a general block), resp. `work()` (in the case of a synchronous block). These functions are called by the scheduler and are given a certain number of input samples (which is not fixed, but can be influenced by the programmer to a certain degree). The work function then does its signal processing and reports back, how many input samples were processed and how many output samples produced.

GNU Radio also takes care of buffering the data. While signal processing blocks usually process samples as fast as they come in (or as fast as the CPU allows, depending on which is lower)⁴, some signal sources and sinks provide, respectively require a constant sample rate (e.g. the audio sink). In this case, it is the programmers responsibility to make sure, that data is processed fast enough. If a source delivers more samples than the application can process, then some buffer will eventually overflow, and samples are lost (this is called an overrun). If the sample rate at the input of a sink is too low on the other hand, the buffer will eventually be empty and a glitch in the signal results (an underrun).

Such problems can also occur, if some interface (e.g. the Universal Serial Bus (USB) interface between an USRP and the PC) is too slow, or if data streams with different fixed sample rates are connected without appropriate interpolation or decimation. Obviously, it is crucial to avoid such problems as much as possible for a real-time radio application (this can never be completely guaranteed however, as a PC usually does not run a real-time OS).

²SWIG is an acronym for “Simplified Wrapper and Interface Generator”. Information about SWIG can be found under <http://www.swig.org/>.

³GNU Radio uses the WX Widgets GUI library. Please refer to <http://www.wxwidgets.org/> for more information.

⁴It is however possible to limit the processing rate by using the block `gr.throttle`.

While the principle of connecting blocks with continuous data streams is quite general, it has its limitations, for example when dealing with packet based data. Currently, several extensions to GNU Radio, such as message blocks, which are better suited for handling packet-based data⁵ are under development.

Further information on the GNU Radio architecture and on writing signal processing blocks can be found in the tutorial “How To Write a Block” by Eric Blossom [16].

An illustrative example

The following listing shows a simple and illustrative example of a GNU Radio application:

```
#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio
from gnuradio.wxgui import stdgui2, fftsink2

class audio_fft_graph(stdgui2.std_top_block):
    def __init__(self, frame, panel, vbox, argv):
        stdgui2.std_top_block.__init__(self, frame, panel, vbox, argv)
        audio_sample_rate = 48000
        src = audio.source(audio_sample_rate)
        amp = gr.multiply_const_ff(1000)
        sink = fftsink2.fft_sink_f(panel,
                                   fft_size=1024,
                                   sample_rate=audio_sample_rate)

        self.connect(src, amp, sink)

if __name__ == '__main__':
    app = stdgui2.stdapp(audio_fft_graph, "Real-Time-Audio-Spectrum")
    app.MainLoop()
```

In this example, the PC’s microphone is taken as signal source. The signal is amplified by the factor 1000 and the spectrum of the resulting signal is displayed on the screen.

Contrary to the explanations above, the flow graph is never started with its `start()` method. In this example, this is done automatically, when the `stdapp` is initialised.⁶

Figure 5.1 shows a screenshot of the resulting application.

5.1.2 GNU Radio Companion

GNU Radio Companion (**GRC**) is a graphical front end for GNU Radio. It allows the creation of software radios by wiring together graphical blocks. While **GRC** was originally developed independently of GNU Radio, it can now be found in the official GNU Radio repository. **GRC** is developed by Josh Blum.

Although visual programming is not as flexible as programming in python, programming with visual tools can be much more intuitive. A simple **FM** broadcast receiver can be created with **GRC** in a few minutes, as shown in Figure 5.2.

⁵Please visit <http://gnuradio.org/trac/wiki/MessageBlocks> for information about message blocks.

⁶The `start()` function is actually called from the constructor of the panel, as can be seen in <http://gnuradio.org/trac/browser/gnuradio/trunk/gr-wxgui/src/python/stdgui2.py>.

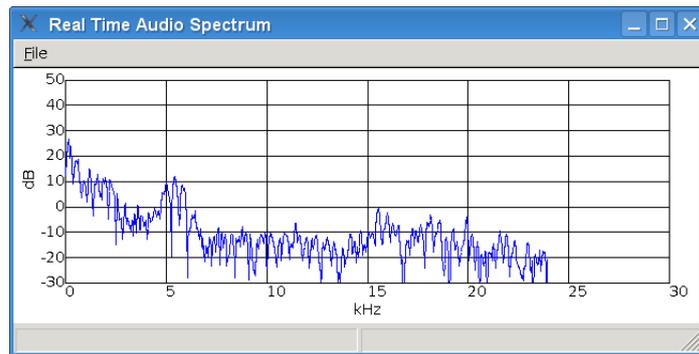


Figure 5.1: Screenshot of the running example application.

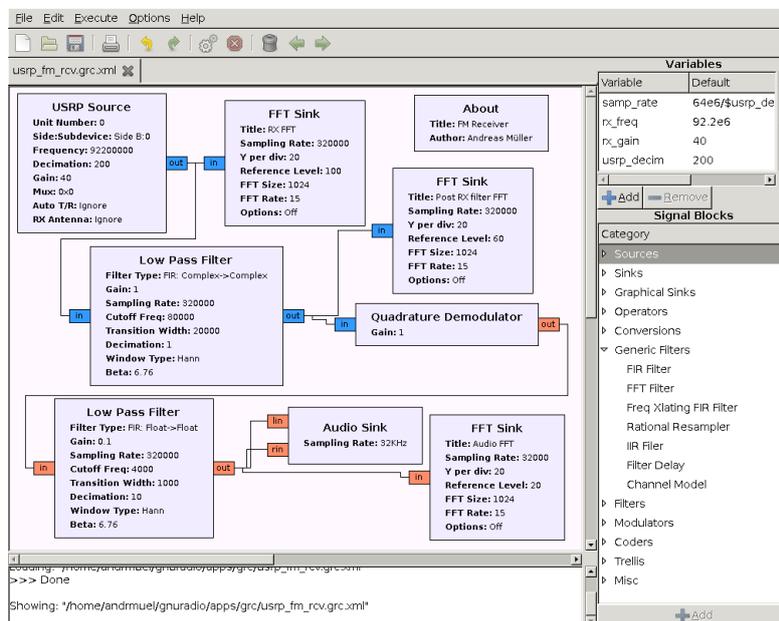


Figure 5.2: GRC with a complete FM broadcast receiver.

5.2 The USRP

The Universal Software Radio Peripheral (**USRP**) is a piece of hardware that can be used with GNU Radio, to build a software radio. It consists of a motherboard and various daughterboards⁷, which can transmit and/or receive in different frequency ranges from 0 to around 2.4 GHz. To allow the **USRP** to work at the desired frequency, the daughterboards can easily be exchanged⁸.

While GNU Radio can be used with other hardware, and the **USRP** with other software, the two are usually used together, because of the ease of use and the relatively moderate price of the **USRP**.

Just like with GNU Radio, all schematics and all code of the **USRP**, including the Verilog code for the Field Programmable Gate Array (**FPGA**), is available for free, under the GNU

⁷The term USRP is also used for the motherboard alone.

⁸A list of available daughterboards can be found on the GNU Radio website under http://gnuradio.org/trac/wiki/List_of_USRP_daughterboards

GPL. The **USRP** is developed and sold by Ettus Research⁹, in collaboration with the GNU Radio developers.

In Figure 5.3, a photograph of an **USRP** with four daughterboards can be seen.

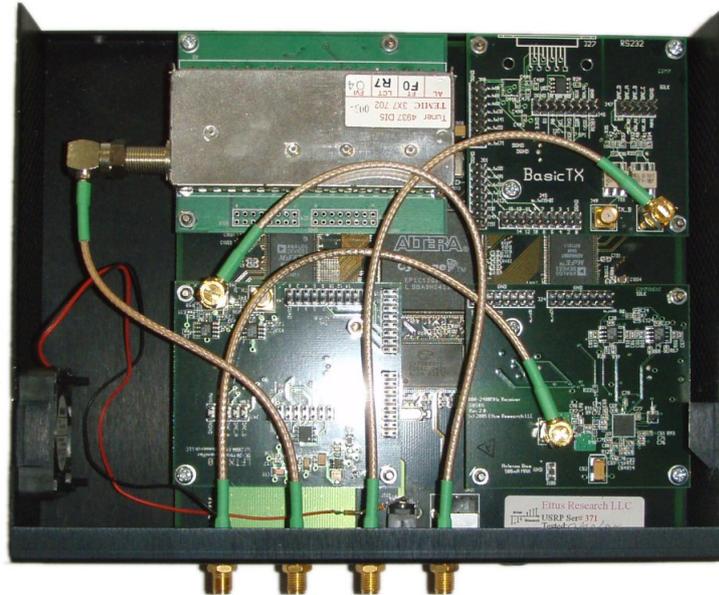


Figure 5.3: The USRP with 4 daughterboards.

5.2.1 USRP Architecture

The **USRP** motherboard's main components are an Altera Cyclone EP1C12 **FPGA**¹⁰, two AD9862 Mixed Signal Front-End Processors¹¹, and a Cypress FX2 **USB** 2.0 interface chip. Figure 5.4 shows a high level block diagram of the **USRP**, where the flow of the data between antennas, **USRP** and the PC can be seen. The daughterboards are mainly responsible for initial filtering and for mixing the signal to/from an intermediate frequency, whereas the motherboard translates between the analog and digital domain, converts the samples to/from baseband, and communicates with the PC.

As each of the two AD9862 chips contains two **DACs** and two **ADCs**, there are four receive and four transmit channels, which can either be used as independent channels (if supported by the daughterboard), or as two IQ channels. The core parameters of the D/A and A/D converters are listed in Table 5.1.

Since all data is transmitted via **USB**, the maximum bandwidth is in practice limited by the speed of the **USB** interface. With **USB** 2.0, a speed of about 32 MB/s is possible with good **USB** chip sets, which results in a maximum bandwidth of around 8 MHz with complex 4-Byte samples (16 Bit signed integers, in-phase and quadrature interleaved) and if only one channel is used. For higher bandwidths, the resolution has to be reduced.

⁹See <http://www.ettus.com> for more info.

¹⁰Information about the FPGA can be found at the Altera website: <http://www.altera.com/products/devices/cyclone/overview/cyc-overview.html>

¹¹Their data sheet can be found under <http://www.analog.com/en/prod/0,,AD9862,00.html>.

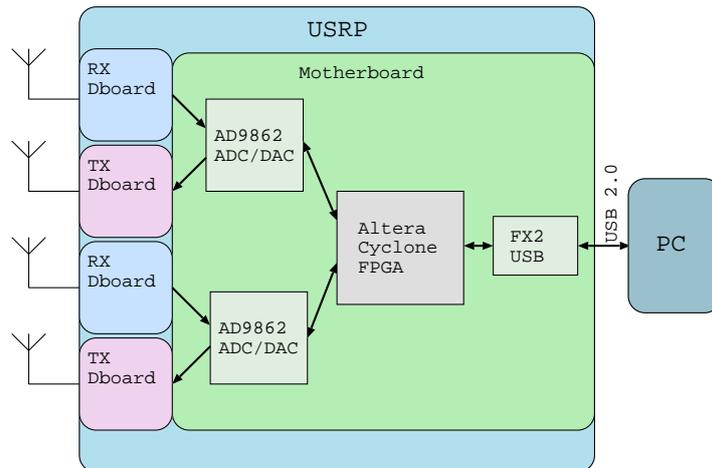
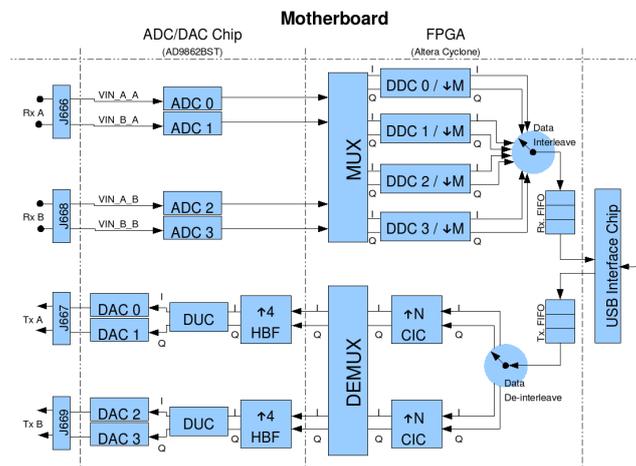


Figure 5.4: High level block diagram of the USRP.

	DA converters	AD converters
Speed	128 MSPS	64 MSPS
Resolution	14 Bit	12 Bit

Table 5.1: Parameters of the D/A and A/D converters in the USRP.

As it is not possible to transmit the data stream at the full sample rate via [USB](#), up- and down-conversion needs to be done in the [USRP](#). Figure 5.5 shows a more detailed block diagram of the [USRP](#) motherboard.

Figure 5.5: USRP transmit and receive paths (Source: GNU Radio Wiki¹²).

On the upper half, the receive chain can be seen. The signals are first converted into digital samples; in the [FPGA](#), the channels are then routed to the Digital Down-Converter ([DDC](#))

¹²<http://gnuradio.org/trac/wiki/UsrpRfxDiagrams>

blocks by a configurable multiplexer. In the DDCs, they are converted to baseband at the desired frequency and decimated. The interior of the DDCs (which is not shown) consists of a CORDIC¹³, which converts the signal to baseband, followed by a Cascaded Integrator Comb (CIC) filter with a programmable decimation and a halfband filter with a fixed decimation of 2. Finally, all data streams are interleaved sample wise, buffered and sent over the USB interface to the PC.

The transmit path is similar, with the exception that the up-conversion is done in the DAC instead of the FPGA. More information can be found in the GNU Radio Wiki [18].

Latency

The whole system, consisting of an application on a general purpose PC, the USB interface and the USRP, has together a latency of several hundred milliseconds. Because PC's are not designed for real-time applications, it is difficult to reduce the latency. This makes the use of the USRP for TDMA systems difficult (i.e. application dependent reconfiguration on FPGA level would be needed). Extensions to overcome such limitations are currently under development.

5.2.2 Using the USRP in a GNU Radio Application

GNU Radio provides the module `gr-usrp`¹⁴ for interaction with the USRP. The USRP and its daughterboards can be fully controlled from Python. For this, the Python module `usrp` provides several classes, such as `source_c`, which uses the USRP as a source of complex baseband samples. These classes provide several useful functions, e.g. to set the RX frequency of the USRP (`tune()`), set the decimation rate (`set_decim_rate()`) or the programmable gain amplifier of the AD converter (`set_pga()`). When the class is instantiated in Python, the appropriate FPGA configuration (bit stream) is automatically loaded (by default, the file `std_2rxhb_2tx.rbf` is loaded, which contains two receive and two transmit paths; other bit streams, such as a bit stream with four receive paths, are available). Other parameters, such as the decimation factor, the multiplexer value and the number of channels can be set during instantiation and adjusted on the fly whenever required.

5.3 Wrap Up

Together, GNU Radio and the USRP provide a powerful tool for creating software defined radios. GNU Radio carefully abstracts lower level details from its users, to allow them to focus on their application. With the growing number of available low-level signal processing blocks, GNU Radio becomes more and more versatile. The USRP is kept as generic as possible. This makes it usable for a wide range of applications; in most cases without the need to write any new Verilog code.

¹³A CORDIC is an efficient way to compute trigonometric functions. CORDIC stands for COordinate Rotation DIgital Computer. More information is available in [17]. The CORDIC implementation for the USRP can be found under http://gnuradio.org/trac/browser/gnuradio/trunk/usrp/fpga/sdr_lib/cordic.v

¹⁴<http://gnuradio.org/trac/browser/gnuradio/trunk/gr-usrp>

Chapter 6

Implementation

In this chapter, the implementation of the **DAB** receiver is presented. After a short explanation of the general experimentation setup, the implementation of the **OFDM** demodulation will be explained (Section 6.2). The **OFDM** part represents the major part of the work for this thesis, and it was also extensively tested. For this purpose, an **OFDM** modulation block was implemented. The results of these tests are discussed in Section 6.3. Finally, in Section 6.4, the implementation of the **FIC** decoder, which was significantly facilitated by the availability of a generic trellis module, `gr-trellis`, is discussed.

6.1 Setup

While it is possible to use GNU Radio to work on a signal that is received live over the air, it is more convenient to record some samples to a file and use this file for experiments. From a file, samples can easily be loaded into GNU Radio or Matlab. Using a file for development also has the advantage that results are reproducible. Once the application is finished, it is easy to switch to a signal received live.

The **USRP** with the **TVRX** daughterboard was therefore used to record about half a minute worth of **DAB** Mode I samples from the Swiss **DAB** Ensemble at 227.36 MHz¹. However, during their experiments, Jens Elsner and Nicolas Alt had discovered, that the **TVRX** disturbs the phase of the received signal strongly, making it very hard to decode the content.² Nicolas was kind enough to provide some of his samples, recorded with a Lyrtech **SDR** platform, which provided a much cleaner signal. The Lyrtech samples on the other hand had a noticeable offset in the sampling rate. This required some additional work to correct the sampling frequency, but did not pose any problems otherwise.

For reference, Jens additionally provided some samples from a mode II **DAB** ensemble, recorded with the **USRP** and the Direct Broadcast Satellite Receiver (**DBSRX**) daughterboard (in Switzerland, there are unfortunately no **DAB** ensembles in the frequency range of the **DBSRX**).

Towards the end of this thesis, samples were additionally recorded with a setup consisting of a spectrum analyzer to receive and filter the signal and an external mixer to move the signal from the IF output of the spectrum analyzer (310.7 MHz) to a carrier frequency of 20 MHz, which is in the range of the **BasicRX**. The samples were then recorded with the **USRP** and the **BasicRX** daughterboard. With this setup, the **FIC** of a Swiss **DAB** signal could successfully be decoded.

¹A list of the available ensembles in Switzerland can be found under <http://digiradio.ch/dab/programme/ch/index.html>.

²Jens noticed, that each symbol had a random phase offset, possibly due to phase noise in the oscillator of the **TVRX** (the **TVRX** is the only daughterboard, which does not use the same clock source as the **USRP** mainboard). Ettus Research is currently developing a new daughterboard for the frequency range of the **TVRX**, which will hopefully solve these problems.

	TVRX samples	DBSRX samples	Lyrtech samples	BasicRX samples
Device	USRP (TVRX)	USRP (DBSRX)	Lyrtech SDR	USRP (BasicRX)
Frequency Band	Band III	L-Band	Band III	Band III
DAB Mode	I	II	I	I
Sampling frequency	2 MSPS	2 MSPS	2.048 MSPS	2 MSPS
Actual sampling frequency	≈ 1999973 SPS	≈ 1999983 SPS	≈ 2047846 SPS	1999971 SPS
Sampling frequency offset	≈ 14 ppm	≈ 9 ppm	≈ 75 ppm	≈ 15 ppm
Carrier frequency offset	≈ 4 kHz	≈ 3 kHz	≈ 74 kHz	≈ 0.04 kHz
Problems	Random phase jitter		Sample rate offset	Add. equipment used
FIC decoding	not successful	successful	successful	successful

Table 6.1: Classes of sample streams used for testing.

Table 6.1 gives an overview over the used samples and lists some challenges that were encountered (Actually, multiple streams of each type of samples were used, to see the effects of different Signal to Noise Ratios (SNRs), etc. The parameters listed in Table 6.1 are valid for all used streams of each type, however.).

All used samples are complex floats. Although floats may be processed slower than integers on certain architectures, they are much easier to work with, as the programmer needs to worry much less about ranges and precision. The use of complex float samples is in fact rather common in GNU Radio.

6.2 Physical Layer (OFDM)

DAB uses OFDM with D-QPSK modulation to transmit data. OFDM requires precise timing and frequency synchronisation. To achieve this, several synchronisation blocks are needed, before the signal can be demodulated. Figure 6.1 gives a coarse overview of the receiver, that has been implemented in GNU Radio. More detailed signal flow graphs, that have been created directly from the code, can be found in Appendix A.

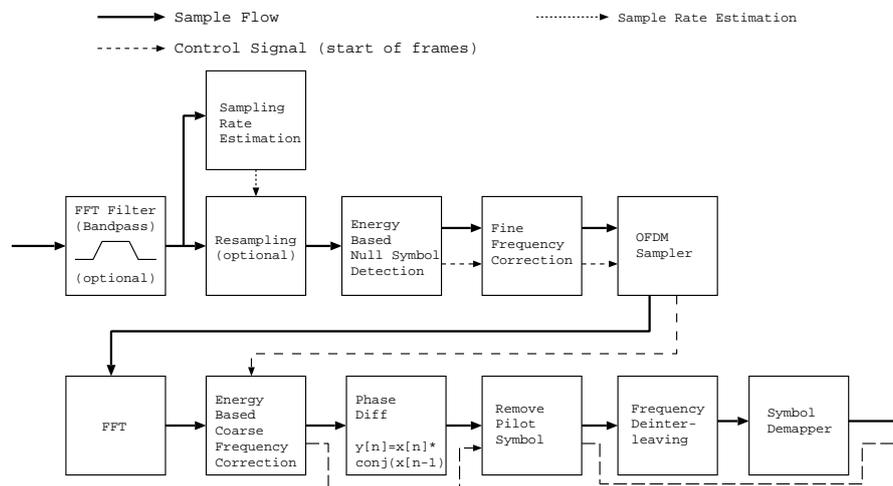


Figure 6.1: Block diagram of the OFDM demodulation.

The following sections explain the individual blocks. Please note that the description of the blocks is written in the order, in which they were developed. This is not always the same order as the order of blocks in the flow graph.

6.2.1 Input Filtering

If the user enables filtering, the samples are first passed through an FFT bandpass filter, which is provided by GNU Radio (`gr.fft_filter_ccc`).

An FFT filter takes the FFT of the signal, multiplies it with the FFT of the taps and transforms the result back into time domain. Although this means, two FFTs are needed³, the convolution of the signal with the taps can be replaced by a multiplication. In the case of steep filters with many taps, this results in quite a speed gain.

While the input filter can help to reduce the impact of noise and nearby interfering signals, care must be taken not to cut away some part of the OFDM signal if there is a carrier frequency offset. The crystal of the USRP is specified to an accuracy of 50 parts per million (ppm); therefore, at 230 MHz, a maximal carrier frequency offset of about 12 kHz can be expected. The TVRX daughterboard is an exception here however, as it contains its own oscillator. According to the data sheet of the TVRX, the frequency offset may be up to 50 kHz. While the actual carrier frequency offset seems to be much smaller in practice (as can be seen in Table 6.1), a generous filter cutoff frequency of 868 kHz (i.e. 100 kHz more than the signal bandwidth) is set as the default value, because of the larger offsets that were seen with Lyrtech samples (such parameters can be easily adjusted – all parameters are collected in the file `parameters.py`).

6.2.2 Time Synchronisation

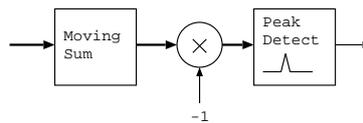


Figure 6.2: Null symbol detection

Time synchronisation is required to find the start of the frames. As many of the other blocks rely on the frame start to process the signal, it is very important that the Null symbol detection is accurate.

Since DAB frames are separated by Null symbols, during which the signal is zero (except for a possible Transmitter Identification Information (TII) signal, which can be ignored, as its energy is comparatively small), time synchronisation can easily and accurately be done by observing the signal energy.

To find the Null symbol, a moving sum of the length of the Null symbol is first calculated, which can be done with an FIR-filter with taps set to one:

$$y[n] = \sum_{k=0}^{T_{Null}-1} x[n-k]$$

with T_{Null} as specified in Table 3.2. The signal is then inverted and a peak detector from GNU Radio, `gr.peak_detector_fb` is used, which is basically a finite state machine with two states. Whenever a given threshold is exceeded, the peak detector starts looking for a maximum, until the signal goes below the threshold again. Null symbol detection is shown in Figure 6.2.

Because the FIR filter is rather inefficient, it has been replaced by an equivalent IIR filter,

$$y[n] = y[n-1] + x[n] - x[n-T_{Null}]$$

While this replacement is mathematically simple, a practical implementation must take care that no error accumulates, because of the limited precision of the calculations, and that there are no

³Technically, even three FFTs, but the transformation of the taps only has to be done once.

problems with new values being rounded away, if the sum is already large in comparison⁴. Experiments showed that this was indeed a problem, and the moving sum was therefore implemented as a new block in C++, with a double precision data type for the sum.

Figure 6.3 shows a part of the received signal (top), the output of the moving sum block (bottom) and the detected start of the DAB frame (red).

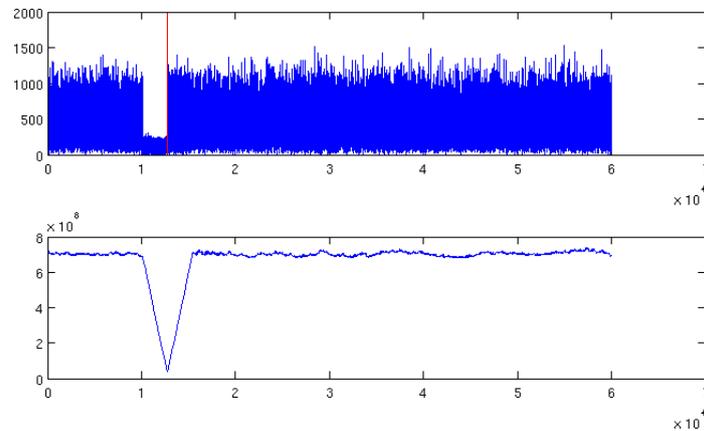


Figure 6.3: Null symbol detection.

Null symbol detection is implemented in the class `detect_null` in the file `detect_null.py` (directory `gr-dab/src/python/`).

Complexity Analysis

While the implementation with an FIR filter has a complexity of $O(n * m)$ (with m being the length of the Null symbol), the rewrite with a delay block and an IIR filter only has blocks with $O(n)$ complexity, and it can therefore be expected to run quite fast.

Although tests with Gaussian noise at a sample rate of 2.048 MSPS first seemed to imply that time synchronisation alone still already uses most of the available processing power (on a laptop with a Pentium Mobile CPU, clocked at 1.6 GHz), OProfile⁵ later showed that actually, about 80% of the resources were spent on generating the noise. Analysis also shows, that with the FIR implementation, most processing time is spent on calculating the moving sum (the two loops belong to a Streaming SIMD Extensions (SSE) optimized assembler function that calculates the complex dot product, which is used by the FIR filter):

```

CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
samples %      image name                symbol name
7768    96.3533  libgnuradio-core.so.0.0.0.0 .loop2
59      0.7318   libgnuradio-core.so.0.0.0.0 gr_fir_fff_simd::filter()
54      0.6698   libgnuradio-core.so.0.0.0.0 .loop1
41      0.5086   libgnuradio-core.so.0.0.0.0 gr_peak_detector_fb::work()
37      0.4589   libgnuradio-core.so.0.0.0.0 .cleanup
31      0.3845   libgnuradio-core.so.0.0.0.0 gr_vector_source_c::work()
19      0.2357   libgnuradio-core.so.0.0.0.0 float_dotprod_sse
16      0.1985   libgnuradio-core.so.0.0.0.0 gr_fir_fff_generic::filterN()
15      0.1861   libgnuradio-core.so.0.0.0.0 gr_complex_to_mag_squared::work()
10      0.1240   libgnuradio-core.so.0.0.0.0 gr_single_threaded_scheduler::main_loop()
8       0.0992   libgnuradio-core.so.0.0.0.0 gr_multiply_const_ff::work()
[...]

```

⁴For illustration: In Matlab, $1e20+1-1e20$ evaluates to zero.

⁵For a brief description of OProfile, please see Appendix C.1.

In the delay block implementation on the other hand, CPU time is much more evenly distributed among the blocks, and the CPU usage is drastically reduced (this is not reflected in the number of samples in the OProfile report, as a different amount of data was processed):

```

CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
samples %      image name                symbol name
2929    45.7871  libgnuradio-core.so.0.0.0 gri_iir::filter()
761     11.8962  libgnuradio-core.so.0.0.0 gr_peak_detector_fb::work()
646     10.0985  libgnuradio-core.so.0.0.0 gr_vector_source_c::work()
549     8.5821   libgnuradio-core.so.0.0.0 gr_sub_ff::work()
508     7.9412   libgnuradio-core.so.0.0.0 gri_iir::filter_n()
261     4.0800   libgnuradio-core.so.0.0.0 gr_complex_to_mag_squared::work()
215     3.3610   libgnuradio-core.so.0.0.0 gr_single_threaded_scheduler::main_loop()
123     1.9228   libgnuradio-core.so.0.0.0 gr_add_const_ff::work()
117     1.8290   libgnuradio-core.so.0.0.0 gr_multiply_const_ff::work()
[...]
```

The implementation of the moving sum block in C++ actually reduces the runtime even a bit more.

6.2.3 Fine Carrier Frequency Synchronisation

Carrier frequency synchronisation is split into coarse and fine carrier frequency synchronisation. The coarse carrier frequency error is defined as the frequency offset in multiples of the subcarrier spacing (in the case of DAB, this is $\frac{1}{T_U}$, with the value of T_U as defined in Table 3.2 on Page 13), whereas the fine carrier frequency error is the remaining error, smaller than the subcarrier spacing. In the presented implementation, fine carrier frequency synchronisation is done first.

As derived in [19], a fine carrier frequency offset actually has two separate effects:

- the amplitude of each symbol is reduced and the phase is shifted (i.e. the constellation is rotated)
- Inter Carrier Interference (ICI) is introduced

Accurate fine frequency correction is therefore rather important; especially in a signal with a small subcarrier spacing and possibly low SNR.

GNU Radio already implements several carrier frequency synchronisation blocks for OFDM. One of them is a preamble correlator, as suggested by Timothy M. Schmidl and Donald C. Cox in [20]. Schmidl and Cox make use of the fact, that if a pilot symbol is used, where all odd subcarriers are zero, the first half of the training symbol is equal to the second half, up to a phase difference, which depends only on the fine carrier frequency error Δf (as the coarse frequency error introduces phase differences only in multiples of a complete period):

$$\phi = \pi T_U \Delta f$$

where T_U is again the inverse of the subcarrier spacing, as defined in Table 3.2. Schmidl and Cox show, that the fine carrier frequency error can therefore be estimated as

$$\widehat{\Delta f} = \hat{\phi} / (\pi T_U)$$

with

$$\hat{\phi} = \angle(P(d)) = \angle \sum_{m=0}^{M-1} r_{d+m}^* r_{d+m+M},$$

where M is the number of complex samples in the first half of the training symbol and r_i denotes a complex baseband sample. Once the frequency error has been calculated, it can be corrected by multiplying the signal with $\exp(-j2t\hat{\phi}/T_U)$.

This method is robust and has linear complexity. Unfortunately, the pilot symbols used in DAB do not have the property that all odd carriers are zero.

Another synchronisation method implemented by GNU Radio is the cyclic prefix correlator, as suggested by Jan-Jaap van de Beek, Magnus Sandell and Per Ola Börjesson in [21]. Instead of comparing the first and the second half of a pilot symbol, this method compares the cyclic prefix of any symbol to the last part of that symbol, whose phase should be identical, unless there is a fine carrier frequency error. The method therefore uses the redundancy introduced by the cyclic prefix, assuming that the channel is Additive White Gaussian Noise (AWGN). In their paper however, they show that this method also works in a time-dispersive channel (although with decreased performance). The existing code using this method was adapted for DAB. The moving sum was implemented in the same way as presented in Section 6.2.2, and the block therefore has linear complexity.

After calculating the fine carrier frequency offset, correction is straightforward: The original signal is multiplied with a signal from a numerically controlled oscillator, whose frequency is as high as the negated offset.

The implementation is in the file `ofdm_sync_dab.py`. Figure 6.4 shows the corresponding block diagram.

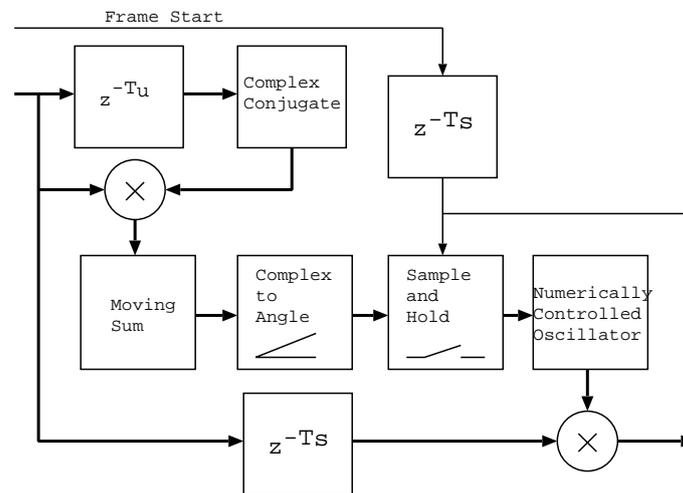


Figure 6.4: Fine carrier frequency synchronisation.

In Figure 6.5, the result can be seen. The plot shows DAB data (blue), the start of the symbols (red), the output of the moving sum over the phase difference (green) and the value after the sample and hold block (magenta).

The result was verified by writing the synchronised signal into a file and feeding it through synchronisation again. In Figure 6.6 it can be seen, that the fine frequency error is indeed corrected. Beginning at the start of the first symbol, the phase difference goes close to zero (the result can be seen at the end of the first symbol, because of the delay introduced by the moving sum and the correlator itself). Please note that the first symbol is corrected with the fine frequency error found by evaluating the phase difference of the cyclic prefix in the first symbol. This can be done by simply delaying the signal by one symbol length, before doing the correction.

Improved Fine Carrier Frequency Synchronisation

While this method generally works, experiments later showed, that there were still some quite abrupt changes in the estimated value for the fine carrier frequency offset, which led to jumping

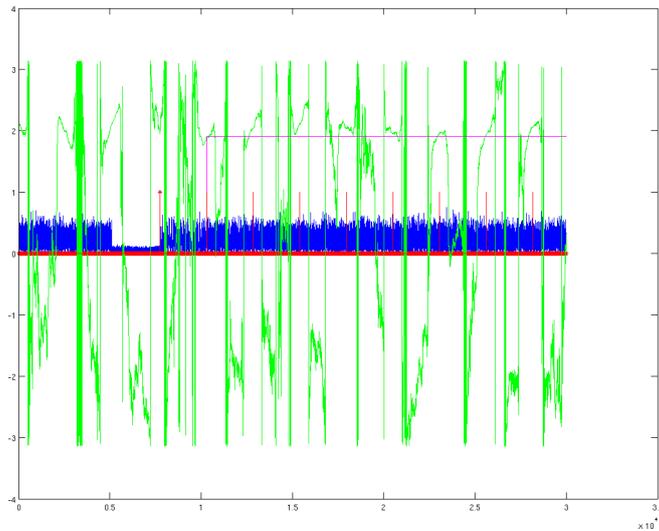


Figure 6.5: Fine carrier frequency synchronisation.

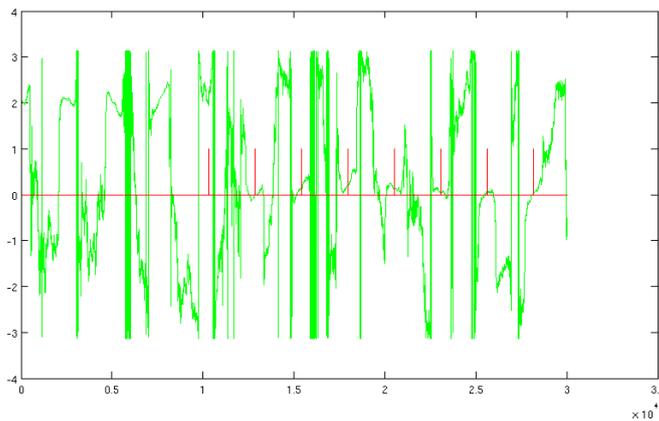


Figure 6.6: Second pass through the fine frequency correction.

phase offsets in the **DAB** symbols, as shown in Figure 6.7 (for this plot, the **DBSRX** samples were used).

One possible way to improve the fine carrier frequency offset estimation is to evaluate more than one symbol per **DAB** frame and use the average value. The trade-off about this is, that an additional delay is introduced. Additionally, if there is a sampling frequency offset, symbols towards the end of the **DAB** frame tend to have larger timing offsets, since the timing is always synchronised at the start of the frame. This means that in the presence of a sampling frequency offset, the symbols at the start of the frame are more reliable for fine carrier frequency offset estimation. For later symbols, the possibility that the sampling is not done exactly at the end of the symbol is increased.

Another possibility is to adjust the value only gradually to changes, e.g. by using the correction value

$$f_c[n] = \alpha \hat{f}_f[n] + (1 - \alpha) f_c[n - 1] \quad 0 \leq \alpha \leq 1$$

based on estimates $\hat{f}_f[n]$ of the fine carrier frequency offset.

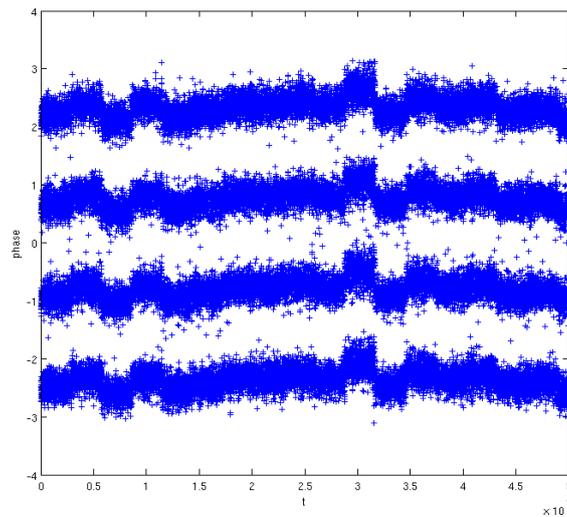


Figure 6.7: Phase jumps because of insufficient fine frequency synchronisation.

These two methods are implemented in the block `ofdm_sync_dab`, which makes use of the C++ block `ofdm_ffs_sample`. The plot in Figure 6.8 shows, that the phase indeed doesn't jump any longer.

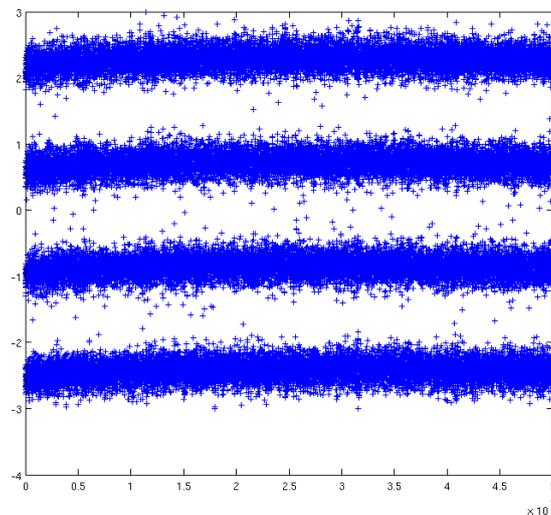


Figure 6.8: No more phase jumps because of averaging.

6.2.4 OFDM Sampler

The **OFDM** sampler is responsible for removing the cyclic prefix and slicing the sample stream into vectors with one **OFDM** symbol each, i.e. each vector has the length of one symbol in time domain. This task is implemented in the C++ block `dab_ofdm_sampler`, using a finite state machine with three states.

The same block also adjusts the Null symbol control signal, such that it indicates the first symbol vector of a **DAB** frame instead of the first sample.

6.2.5 FFT

Once the stream is split into vectors, they can be transformed into the frequency domain, by applying the [FFT](#). This is done with the block `gr.fft_vcc` provided by GNU Radio. This block uses the external Fastest Fourier Transform in the West ([FFTW](#)) library⁶ to perform fast Fourier transformation.

6.2.6 Coarse Carrier Frequency Synchronisation

The goal of coarse carrier frequency synchronisation is to correct the carrier frequency offset in multiples of the subcarrier spacing. As fine carrier frequency synchronisation has already been done at this point, the total carrier frequency offset should be close to zero afterwards.

The effect of a coarse carrier frequency offset is that the subcarriers are moved to the wrong [FFT](#) bins, and therefore interpreted as the wrong symbols. As even a shift of one subcarrier spacing would usually result in complete loss of the information, coarse done frequency must be done with high accuracy (i.e. coarse frequency correction is either done perfectly, or it fails completely).

In this implementation, coarse carrier frequency correction is done after fine carrier frequency synchronisation. In this case, coarse carrier frequency synchronisation can easily be done after the [FFT](#), by looking at the energy of the signal. In the case of [DAB](#), this is supported by the central carrier, which is always zero. The coarse carrier frequency offset estimation is therefore

$$\hat{f}_c = \arg \max_n \sum_{0 \leq i \leq K, i \neq K/2} |X[i+n]|^2 \quad n \in [0, L_F - K)$$

where $X[i]$ is the i -th entry in the [FFT](#) vector, K is the number of used subcarriers and $L_F = T_U/T$ the [FFT](#) length (please refer to [Table 3.2](#) for specific values).

While this correlation has quadratic complexity, the same trick as for the moving sum – only calculating the energy of the first offset and just adding the difference for the other offsets – can be used to get linear complexity. This also makes it unnecessary to restrict the search range for the frequency offset.

The implementation is in the C++ block `dab_ofdm_coarse_frequency_correct`, which also removes the unused carriers and outputs vectors with K (the number of used subcarriers, as specified in [Table 3.2](#)) symbols per vector.

6.2.7 Phase Differentiation

Since [DAB](#) uses [OFDM](#) with [D-QPSK](#), the phase difference of consecutive symbols must be calculated.

This can be done by multiplying each symbol with the complex conjugate of the previous symbol:

$$y[n] = x[n] * \overline{x[n-1]}$$

Although GNU Radio provides a block that does exactly this, a new C++ block, called `diff_phasor_vcc` has been implemented, to allow the processing of complete symbol vectors (while this does not have any algorithmic advantages, it simplifies both programming and scheduling).

An interesting advantage of [D-QPSK](#) is that the absolute value of the phase of the symbols never needs to be known, since only the phase difference from symbol to symbol is of interest. Therefore, no phase equalization is done at any point – the [PRS](#) only serves as a reference to calculate the first phase difference.

⁶The FFTW library is available under <http://www.fftw.org/>.

6.2.8 Removal of the Phase Reference Symbol

Since the PRS is no longer needed, it can be removed. This is done in a small C++ block called `remove_first_symbol_vcc`.

6.2.9 Sampling Frequency Offset Estimation and Resampling

At this stage of the receiver, i.e. after synchronisation, slicing, FFT, phase differentiation and PRS removal, a plot of the symbols should show four separate clouds with symbols belonging to the four constellation points. Unfortunately, this was not the case. Figure 6.9 shows the plot of the symbols (using the Lyrtech samples).

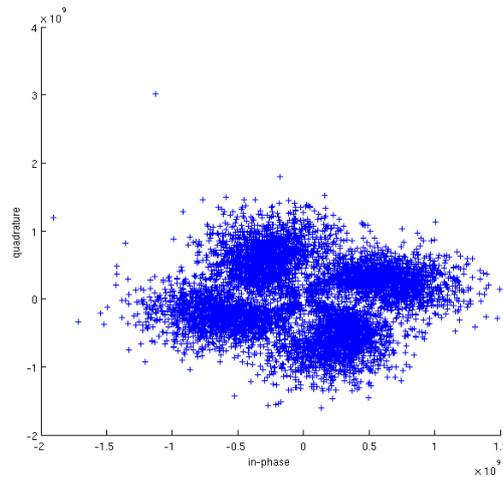


Figure 6.9: Plot of partially synchronised symbols.

More plots with different parameters led to the assumption, that there might be a frequency dependent phase offset introduced by an inaccurate sampling rate. The plot displayed in Figure 6.10, which shows the phase of symbols from different subcarriers, confirmed this assumption.

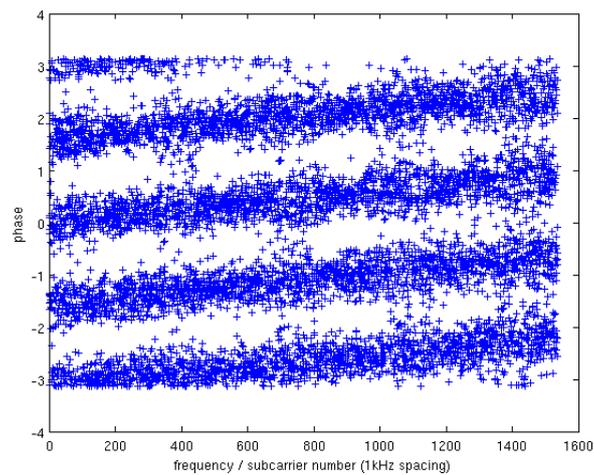


Figure 6.10: Plot of the phase over the subcarrier number.

This problem is also described in [19], where it is shown that an offset in the sampling frequency has the following effects:

- reduction of the amplitude of the symbols
- a phase shift for each symbol, depending on the distance to the central carrier
- inter-carrier interference, as the orthogonality between subcarriers is lost

According to [19], the phase offset on subcarrier i is

$$\Delta\vartheta_i = 2\pi(\varepsilon_s \cdot i + \varepsilon_c)$$

where $\varepsilon_c = \frac{\Delta f_c}{\Delta f}$ is the relative carrier frequency offset (with Δf_c the actual frequency offset and Δf the subcarrier spacing) and $\varepsilon_s = \Delta f_s / f_s$ is the relative sampling frequency offset. In our case, the sampling frequency is 2.048 MSPS with an offset of approximately 75 ppm. The phase offset difference between the lowest and the highest subcarrier should therefore be

$$\Delta\vartheta_i \approx 2\pi * 1536 * 75 * 10^{-6} \approx 0.72$$

A coarse estimation from Figure 6.10 results in a similar value.

The first idea to solve this problem was to estimate the offset of each individual subcarrier and use this estimation to correct the phase of each subcarrier. This idea is implemented in the C++ block `correct_individual_phase_offset_vff`. The experiment showed however, that this is not a good idea. Figure 6.11 shows the estimation of the offset for each subcarrier after 10, 50, 100 and 500 symbols. While the estimation for central carriers with small offsets is quite precise, larger offsets cannot be estimated accurately, because as soon as an offset is larger than $\frac{\pi}{4}$, the symbol is associated to another phase region and the offset estimation gets wrong. In Figure 6.11, this effect can be seen for the carriers from 0 to about 300.

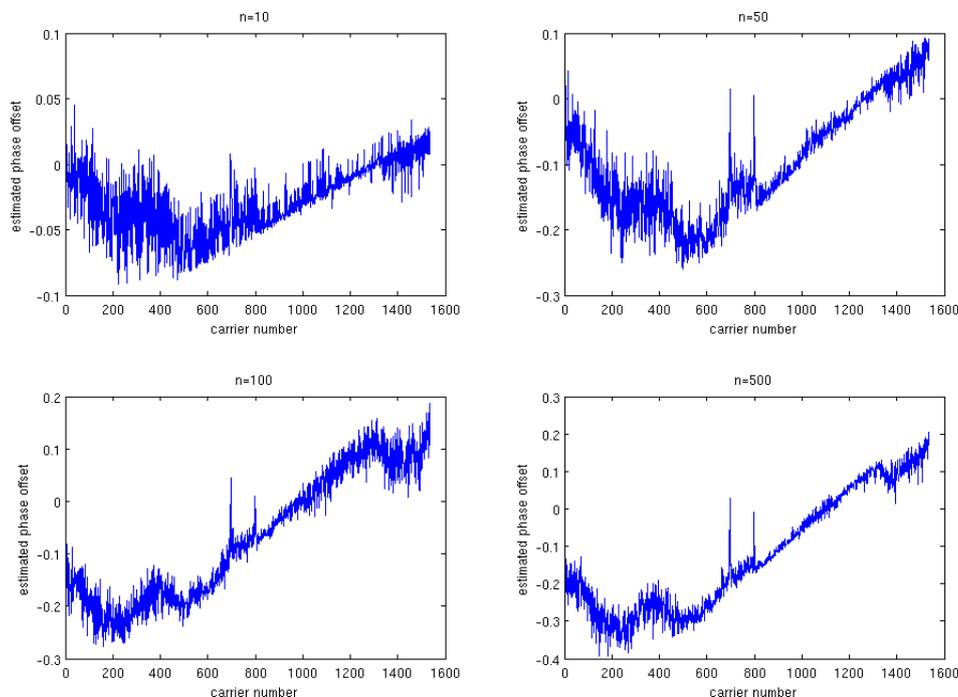


Figure 6.11: Estimated phase offset per carrier after 10, 50, 100 and 500 symbols, with $\alpha = 0.01$.

The approach used by Nicolas Alt in his implementation is to estimate the sample rate by looking at the length of a frame. Since the start of each frame needs to be detected anyways (by looking at the Null symbols), this can be implemented easily.

Once the actual sample rate is known, it can be used to estimate the individual phase offset of each subcarrier, or to resample the signal (another very efficient and beautiful method is described by Maja Sliskovic in [19] – the author basically suggests to adjust the twiddle factors in the FFT to make up for the sampling frequency offset; as this only needs to be done once, it is a very efficient method).

The method of choice for this implementation was to resample the signal. This is done by fractional interpolator from GNU Radio and a separately running thread, which estimates the sampling rate and updates the interpolation factor. As the interpolation ratio is a parameter of the fractional interpolator rather than an input signal, the estimates are read from Python and the parameter is updated accordingly. This approach however leads to the problem that the exact moment, when the interpolation factor is updated, can not be controlled, and this might therefore happen in the middle of a frame. To avoid this, the fractional interpolation from GNU Radio has been extended, such that it ignores a newly set interpolation factor, until a new frame starts.

As the drift of the sampling rate is usually small, even a static interpolation factor will in practice usually suffice – in the case of the USRP, resampling is in fact usually not needed at all. Dynamic resampling is therefore optional and not enabled by default.

6.2.10 Magnitude Equalization

While the original code did not perform any equalization of the magnitude at all (as the information is only carried in the phase in a D-QPSK signal, this would be a waste of precious CPU cycles), a magnitude equalizer was later added to allow the use of soft bits⁷ (for the use of a trellis decoder with a continuous domain for the input values).

Because the distance to the constellation points is used to determine the certainty of a correct detection for soft bits, it is important that all symbols are rescaled to have the same average power. While this seems counterproductive, as the energy of a symbol contains information about the certainty of correct detection, it makes sense, because the noise is also scaled.

To illustrate this, assume that a given OFDM subcarrier is received with an average energy that is five times lower than on the other subcarriers, for instance because of frequency selective fading due to multipath propagation. The symbols on this carrier will be amplified by a factor of five during equalization, but since they will have approximately the same amount of noise as the other carriers, the noise is also amplified and that symbol will likely be further away from the constellation point, resulting in a soft-bit with smaller certainty.

6.2.11 Frequency Interleaving

The frequency interleaver mixes the different subcarriers, according to a fixed sequence specified in the DAB standard. This is done to achieve better protection against narrowband interfering signals. As the bits are spread in the frequency spectrum, a narrowband interferer is less likely to disturb two bits that belong to the same codeword.

Frequency interleaving is implemented in the block `dab_frequency_interleaver_vcc`. The same block can be used for interleaving and deinterleaving, by simple passing an appropriate interleaving sequence.

⁷The term soft bit is used to designate a bit with an associated value for the probability of the correctness of that bit.

6.2.12 Symbol Demapping

Demapping is implemented in a small C++ block, which simply evaluates, whether the real and the imaginary part of a given symbol are positive or negative. Even for soft bits, it is not necessary to calculate the phase (which is rather slow, as the `atan2` operation is quite CPU intensive). As the real and the imaginary part of a given symbol belong to two independent bits, the real and the imaginary part can directly be used as soft bit value.

6.3 Evaluation of the Physical Layer

6.3.1 The Test Setup

To allow testing of the OFDM implementation, a DAB transmitter was developed. As the transmitter needs no synchronisation, frame start detection, etc. this implementation is quite straightforward and was done in a much shorter time than the receiver.

The modulation block and the demodulation block, connected through a channel model, are wrapped up in a test bench (which is implemented in the file `channel_tests/dab.tb.py`). This test bench generates random bits, runs them through modulation, channel and demodulation and estimates the Bit Error Rate (BER).

The channel model, a simple Python block provided by GNU Radio, models the following imperfections:

- AWGN, modeled by a noise source and an adder
- Carrier frequency offset, modeled by a sine source and a multiplier
- Sampling frequency offset, modeled by a fractional interpolator
- Multipath propagation, modeled by FIR taps

While the imperfections are modeled in the channel, they may in practice also be introduced by the receiver or the transmitter.

The complete test bench is shown in Figure A.3 in the Appendix.

6.3.2 BER in AWGN Channel

To evaluate the BER in a noisy channel, the script `plot_snr_ber.py` instantiates a DAB test bench and steps through all DAB modes and a range of signal energies, while the noise energy is fixed at 1. The results are then plotted and written to a file.

The resulting plot can be seen in Figure 6.12.

Some remarks on this figure:

- The energy of the signal is scaled by the FFT block. To solve this problem, the energy is measured in a first run without noise, and an appropriate scale factor is consequently applied. This also has the effect, that the energy that is specified for the signal is the average energy of the signal *including* the Null symbols (without rescaling, the average energy would be slightly lower).
- The energy of the signal is rescaled by its spectrum occupancy ratio (about 0.75), as the signal does not occupy the same bandwidth as the noise.
- If too many frames are lost, the BER is reported as 0.5, as this is the equivalent to not getting any information.

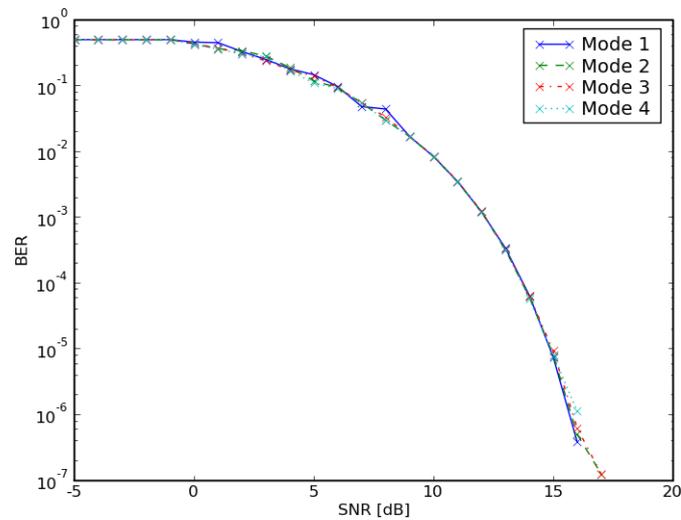


Figure 6.12: BER in a noisy channel (tested with 1MB data per transmission).

- The very first frame can not be detected, as there is no reference signal before the first Null symbol, which makes detection of the frame start impossible. The first frame is therefore ignored (this is no problem for a live receiver; losing the first frame only implicates that the user has to wait 50-100 ms longer before hearing something after turning on the radio).
- A small number of bytes is usually lost at the end. This happens, because the scheduler does not move the last samples through the flow graph, when there is no more input. The lost bytes are ignored for BER calculation (in a live receiver, there is no such problem, as there is an endless stream of input samples).

6.3.3 Effects of Coarse Carrier Frequency Offsets

Although a frequency offset is usually introduced by an inaccurate carrier frequency in the receiver, it can be modeled in the channel. A first test with different coarse carrier frequency offsets (i.e. offsets in multiples of the subcarrier spacing) showed a rather weird and unsatisfactory result, as can be seen in Figure 6.13.

This is surprising for a number of reasons. First of all, the BER should be zero, at least for all small offsets, as it should be no problem to correct them. Secondly, even with completely random data, one would not expect a BER over 0.5. Having a BER of exactly 1 means, that the received signal constellation is probably rotated by 180 degrees. This would lead to the conclusion, that each additional frequency offset of 1 kHz would add a phase offset of $\frac{\pi}{2}$; however, there was one test at 32 kHz, which resulted in a BER of 0.25. This leads to the assumption, that the phase offset must be exactly $\frac{\pi}{4}$ at 32 kHz, such that about half of the symbols are in the correct region (50% of the bits from symbols in the wrong region are still correct, because moving the symbol to a neighboring constellation region only changes one of the two bits carried in the symbol⁸, making a total of 75% correct bits), and the added phase offset would be a little smaller than $\frac{\pi}{2}$.

Further investigation revealed the source of the problem: Cutting away the cyclic prefix (in the `ofdm_sampler`) before doing the coarse frequency correction means, that a phase jump is created, with the phase difference

$$\Delta\phi = \Delta\phi_s \cdot l_{cp} = 2\pi \cdot \frac{\Delta f}{f_s} \cdot l_{cp}$$

⁸The constellation uses a Gray code, as can be seen in the constellation diagram in Figure 3.6 on Page 12.

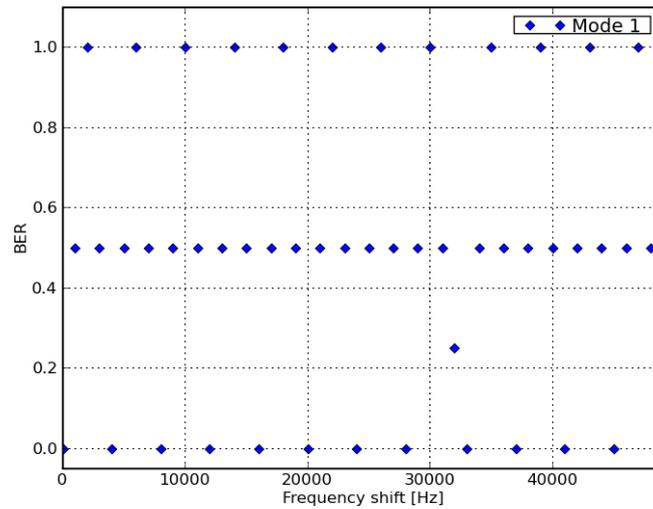


Figure 6.13: BER in channel with coarse carrier frequency offset.

where $\Delta\phi_s$ is the phase difference per sample, l_{cp} is the length of the cyclic prefix, Δf is the carrier frequency offset and f_s is the sampling frequency. If we express the carrier frequency offset as $\Delta f_n = \frac{\Delta f}{f_{sub}}$ in multiples of the subcarrier spacing f_{sub} , which is related to the FFT length l_{fft} by $f_s = f_{sub} \cdot l_{fft}$, we get

$$\Delta\phi = 2\pi \cdot \frac{l_{cp}}{l_{fft}} \cdot \Delta f_n$$

While it would be easy to correct the carrier frequency offset before cutting away the cyclic prefix, by multiplying it with a complex phasor, this would waste precious CPU cycles. Additionally, if the same energy based coarse frequency offset detection was to be used, two FFTs would be required instead of one. The preferred solution to correct the phase offset is therefore to calculate it from the frequency offset and the length of the cyclic prefix and correct it after the FFT.

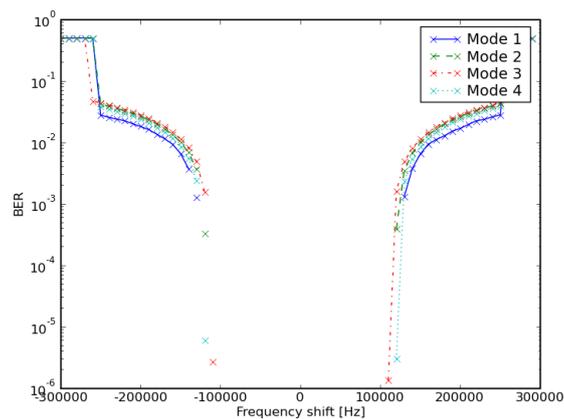


Figure 6.14: BER with coarse carrier frequency offsets (100kB data blocks).

With this problem fixed, a plot of the BER with coarse carrier frequency offsets from -300kHz to +300kHz now looks as depicted in Figure 6.14.

Two different effects are visible here. Firstly, at an offset of about 110 kHz in either direction, the BER starts to increase. This is due to the input bandpass, which in this experiment has a bandwidth of 100 kHz more than the signal bandwidth and a transition width of 50 kHz. With the filter removed, this effect vanishes. Secondly, at about 250 kHz offset in either direction, the BER goes to 0.5, i.e. all information is lost. This happens, because the signal, which has a bandwidth of 1.537 MHz is moved out of the received spectrum area, which has a bandwidth of 2.048 MHz. At an offset of more than 256 kHz, it is therefore no longer possible to perform accurate coarse frequency correction. Such large offsets are however rather unlikely in a practical receiver.

6.3.4 Effects of Fine Carrier Frequency Offsets

A first plot of the BER with fine frequency offsets revealed, that the BER gets quite high if the offset is close to half of the subcarrier spacing. This is not surprising: The fine frequency offset can either be corrected up or down to the nearest subcarrier. With an offset of half the subcarrier spacing, both is equally likely. While this would be okay by itself (coarse frequency correction can cope with it), averaging of the fine frequency offset estimation causes a problem if the estimation oscillates between plus and minus half of the subcarrier spacing, as the estimation is in that case averaged to zero.

This problem can be avoided by detecting, when the fine frequency estimation switches its correction direction. As a carrier frequency drift in both directions is equally likely, it is justified to assume, that if the offset between two frames has changed by more than half of the subcarrier spacing, it is more likely that there was a direction change, than that there is actually such a fast carrier frequency drift. Furthermore, it is also quite unlikely, that such a fast carrier frequency drift should occur – e.g. in DAB mode I with a frame length of roughly 96ms and a subcarrier spacing of 1kHz, a frequency *drift* of more than 5.2kHz per second would have to occur to cause problems (and since the frequency correction values are only updated at the start of the frames, such a fast drift would be problematic in any case).

With this corrected version, the BER is zero for any fine frequency offset. Without noise, the estimation of the fine carrier frequency offset generally deviates less than 0.5 Hz from the actual offset, and even with noise present, the error is less than 10 Hz most of the time, as long as the SNR is better than 10 dB.

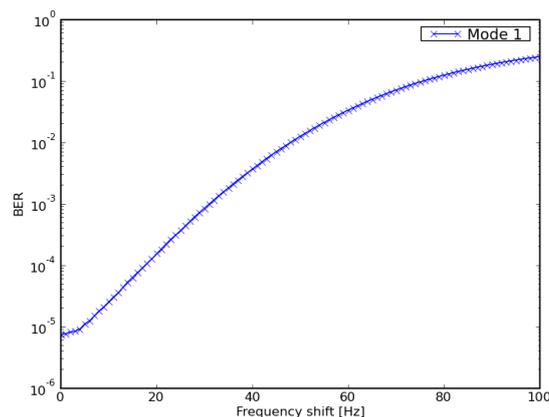


Figure 6.15: Fine carrier frequency offset without correction (simulated with 10MB data blocks).

To illustrate the importance of fine carrier frequency synchronisation, Figure 6.15 shows the

BER of a Mode I DAB signal in a channel with an SNR of 15 dB and a fine carrier frequency offset between 0 and 100 Hz (1/10 of the subcarrier spacing), with correction turned off.

6.3.5 Effects of Sampling Frequency Offsets

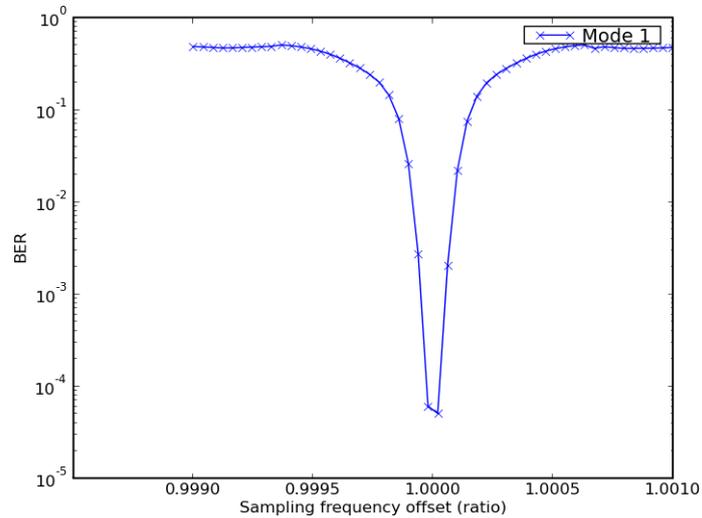


Figure 6.16: Sampling frequency offset – no correction (1MB data packets).

Figure 6.16 shows the effect of a sampling frequency offset without any correction (this test was done with additional noise; the SNR is 15 dB).

Without correction, even small offsets in the area of 50 ppm have a dramatic effect. Since crystal oscillators are often specified to an accuracy of around 50 ppm, this is critical.

With dynamic sampling rate correction enabled, the BER stays below 10^{-2} , for offsets up to 10000 ppm – Figure 6.17 shows the plot (again with SNR 15 dB).

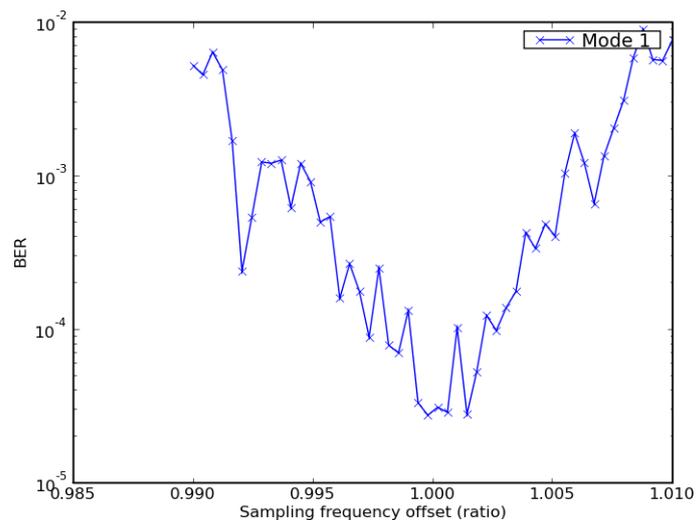


Figure 6.17: Sampling frequency offset with correction (1MB data packets).

The first five frames were ignored here, as the receiver needs some time to adjust to the sample rate offset, which disturbs the result for the first few frames.

6.3.6 Effects of Multipath Propagation

The effect of multipath propagation can be modeled with an FIR filter. To keep things simple, a single tap with variable magnitude and delay was used. The plot in Figure 6.18 shows the result of a simulation run with a delay between 50 and 400 samples and an echo with half the magnitude of the signal.

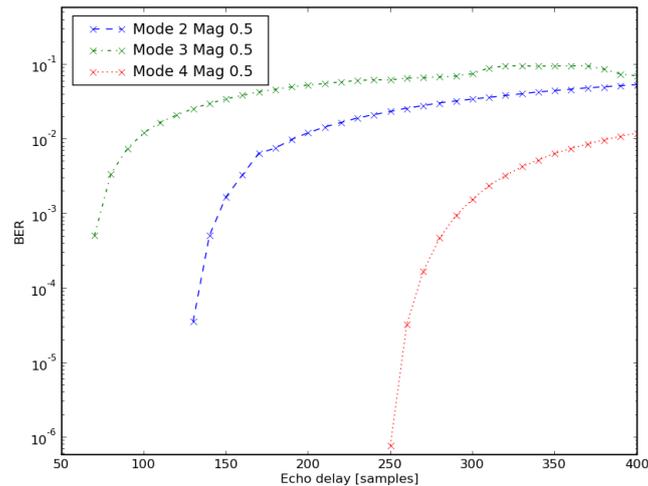


Figure 6.18: Effect of multipath propagation (simulation runs with 500kB data packets).

This plot is particularly interesting, when the lengths of the cyclic prefix for each mode is kept in mind. As a reminder, the lengths are listed in Table 6.2.

DAB Mode	1	2	3	4
Length of the cyclic prefix	504	126	63	252

Table 6.2: Cyclic prefix length.

Comparing these numbers with the plot confirms, that as soon as the delay is longer than the cyclic prefix, the effect of an echo increases drastically. For Mode 1, no errors occurred at all (since zero can not be plotted on a log scale, this is not visible in the plot).

Figure 6.19 shows the effect of taps with magnitudes of 0.3 to 1 (relative to the signal power) for DAB mode 1.

Echoes with magnitudes smaller than 0.2 have little effect, even when the delay is long (magnitude 0.2 is not shown in the plot, as the BER is always zero).

6.3.7 Evaluation of the Processing Speed

After developing the complete OFDM layer without worrying too much about speed, real-time processing was no longer possible. In order to reclaim some of the CPU cycles, a resource usage evaluation with OProfile was therefore needed.

A first evaluation showed, that most of the CPU time was spent for resampling. Resampling from 2 MSPS to 2.048 MSPS used up more than 3/4 of the CPU time. Since the USRP cannot

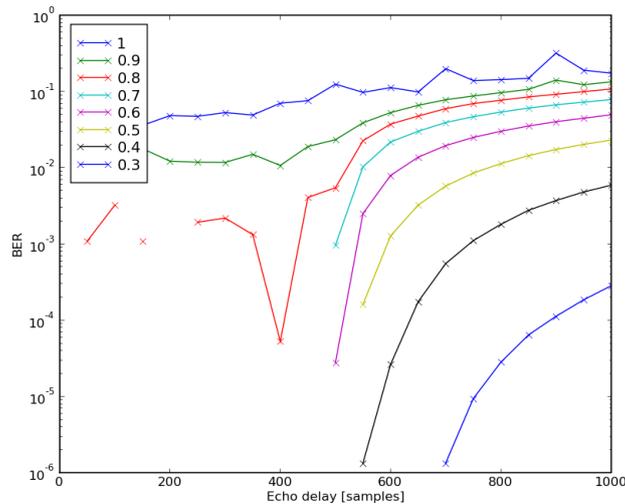


Figure 6.19: Effect of taps with different magnitudes (500kB data packets).

sample with 2.048 **MSPS** directly, and the rational resampler block is obviously too slow, the solution was therefore to adapt the code to work with a sample rate of 2 **MSPS**. The drawback about this is mostly, that the **FFT** length is in that case also 2000. The speed penalty is however acceptable (please refer to Appendix B for an evaluation of the speed of different **FFT** lengths).

The adaptation of the code for arbitrary sample rates only requires the update of a few OFDM parameters. In the source code, this results in the following code fragment:

```
if self.sample_rate != self.default_sample_rate:
    self.T = 1./self.sample_rate
    self.ns_length = int(round(self.ns_length*float(self.sample_rate)/float(self.default_sample_rate)))
    self.cp_length = int(round(self.cp_length*float(self.sample_rate)/float(self.default_sample_rate)))
    self.fft_length = int(round(self.fft_length*float(self.sample_rate)/float(self.default_sample_rate)))
    self.symbol_length = self.cp_length + self.fft_length
    self.frame_length = self.symbols_per_frame * self.symbol_length + self.ns_length
```

With the resampler removed, the block wasting the most **CPU** cycles was the block that calculates the phase for the fine carrier frequency synchronisation algorithm (block 16 in Figure A.1). The `atan2` operation employed by this block requires about 35% of the total **CPU** time inside `libgnuradio`. As there is a sample and hold block right after the `complex_to_arg` block, this is a huge waste of **CPU** cycles, which can however be fixed easily, by moving the phase calculation after the sample and hold block.

With this update, real-time processing was possible again, however only with sampling frequency correction (which uses another interpolation block) turned off. The `OProfile` report looked as follows:

```
15118 17.9274 python2.4
      TIMER:0|
samples|    %|
-----|-----
 5484 36.2746 libgnuradio-core.so.0.0.0
 3100 20.5054 _dab_swig.so
 2804 18.5474 libm-2.6.1.so
 1286  8.5064 libc-2.6.1.so
 1210  8.0037 libfftw3f.so.3.1.2
  817  5.4042 kernel-2.6.24
[...]
```

Obviously, most of the time is still spent inside libgnuradio. The `FFTW` library uses surprisingly few `CPU` time, even with the new `FFT` length 2000. Inside libgnuradio, the statistics look as follows:

```
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
samples %      image name          symbol name
542     20.7583 libgnuradio-core.so.0.0.0 gr_multiply_cc::work()
325     12.4473 libgnuradio-core.so.0.0.0 gr_single_threaded_scheduler::main_loop()
290     11.1069 libgnuradio-core.so.0.0.0 gr_frequency_modulator_fc::work()
215     8.2344  libgnuradio-core.so.0.0.0 gr_sincosf
184     7.0471  libgnuradio-core.so.0.0.0 gr_kludge_copy::work()
183     7.0088  libgnuradio-core.so.0.0.0 gr_peak_detector_fb::work()
142     5.4385  libgnuradio-core.so.0.0.0 gr_fft_vcc_fftw::work()
80      3.0640  libgnuradio-core.so.0.0.0 gr_conjugate_cc::work()
72      2.7576  libgnuradio-core.so.0.0.0 gr_complex_to_mag_squared::work()
72      2.7576  libgnuradio-core.so.0.0.0 min_available_space()
71      2.7193  libgnuradio-core.so.0.0.0 gr_multiply_const_ff::work()
65      2.4895  libgnuradio-core.so.0.0.0 gr_block_detail::input()
[...]
```

One block that uses up a lot of `CPU` time is the frequency modulator (block 20 in Figure A.1), which is used to correct the fine frequency offset. A simple idea to avoid its use is to simply ask the `USRP` to retune the frequency instead. The implementation of this idea showed, that the resource usage is indeed reduced drastically. It should be noted however, that this is a step away from `SDR`. Also, whenever the frequency is retuned, the signal is disturbed.

Another block that uses up many `CPU` cycles is the complex multiplication block. One place where a complex multiplication is done at the full data rate, is the fine frequency estimation (block 14 in Figure A.1). This is especially awkward, because most of these multiplications are unnecessary, as the fine frequency offset is estimated from the first eight (in the case of `DAB` mode I) symbols only, and the values for the other symbols are not needed. To avoid this, a signal would be needed, which tells every block, when it needs to calculate outputs (by default, every block in the flow graph processes all samples). This would require, that every block is extended to be able to use such a control signal. A simpler method is to implement the whole fine frequency estimation (blocks 12-17 in Figure A.1) in one block. Although this approach has the disadvantage that modularity is lost, it is the preferable solution, because it generates the least overhead for control signals and it is easily realizable. This idea is implemented in the C++ block `ofdm_ffe_all_in_one` (block 20 in Figure A.2).

Figure A.2 shows the `OFDM` demodulation with all the code improvements.

6.3.8 Receiving a Live Signal

Figure 6.20 shows a screen shot of the `DAB` constellation sink with samples from the Swiss `DAB` ensemble located at 227.36 MHz.

With all the improvements explained in the previous section, the constellation now looks much better than the one shown in Figure 6.9.

6.4 Fast Information Channel (FIC)

This section describes the process for decoding the `FIC`, which involves the decoding of the convolutional code and reverting the energy dispersal. Although this chapter is written for the `FIC`, the `MSC` is quite similar, as far as convolutional decoding and energy dispersal is concerned.

6.4.1 FIC Symbol Selection and Repartitioning

The first task in the `FIC` decoder is to select the symbols, which belong to the `FIC`. These are the first symbols in the frame (except for the `PRS`). The number of `FIC` symbols depends on the

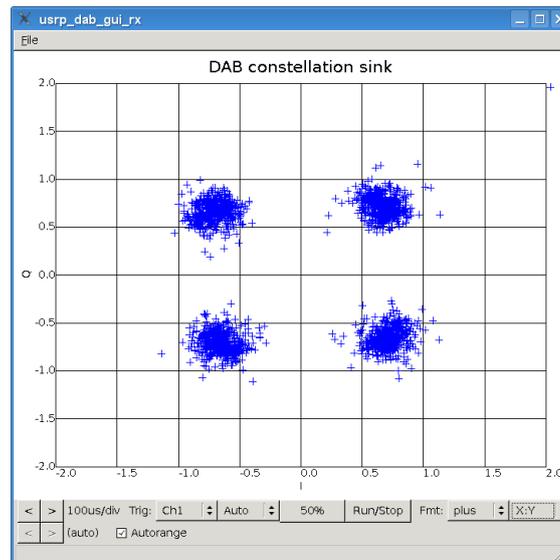


Figure 6.20: DAB constellation sink with samples from the Swiss DAB ensemble at 227.36 MHz.

DAB mode. The block responsible for this task is called `select_vectors` (its implementation is generic, i.e. it can be used to select any number of vectors at the start of the frame or after skipping a given number of vectors).

The next task is to repartition the vectors, such that each vector contains one codeword. In the case of **DAB** mode I for instance, three **OFDM** symbols with 3072 bits each contain four convolutional codewords with 2034 bits each, and therefore three vectors need to be repartitioned into four vectors. This task is done in the block `repartition_vectors`.

6.4.2 Convolutional Coding

Unpuncturing

The bits that have been removed during puncturing (as described in Section 3.3.3) have to be reinserted. As these bits are not known, the block `unpuncture_vff` simply inserts soft bits that are zero. Since a zero is exactly between the two constellation points (real and imaginary part of the symbols are treated separately), it corresponds to a bit that is unknown.

Viterbi Decoder

The module `gr-trellis`, which was written by Achilleas Anastasopoulos, implements a generic **FSM** class and a generic Viterbi decoder. This module makes the decoding process rather simple.

First, an object with the **FSM** described in Section 11.1.1 of the **DAB** specification is instantiated. This can easily be done by calling the appropriate constructor with the generator polynomials and the input and output bit width.

Secondly, a combined block, which first calculates the metrics for the input symbols and then does the decoding is instantiated. This is a bit more complicated. As the metrics calculator considers each four bits generated by one input bit to the **FSM** as one of 16 possible symbols, a constellation table with 64 entries is needed, specifying the 4 constellation points for each of the 16 symbols. The metrics calculator uses the distance from the constellation point to get an estimate for the certainty of correct detection.

With the metrics, the Viterbi decoder then selects the most likely path in the trellis and outputs the corresponding bit sequence.

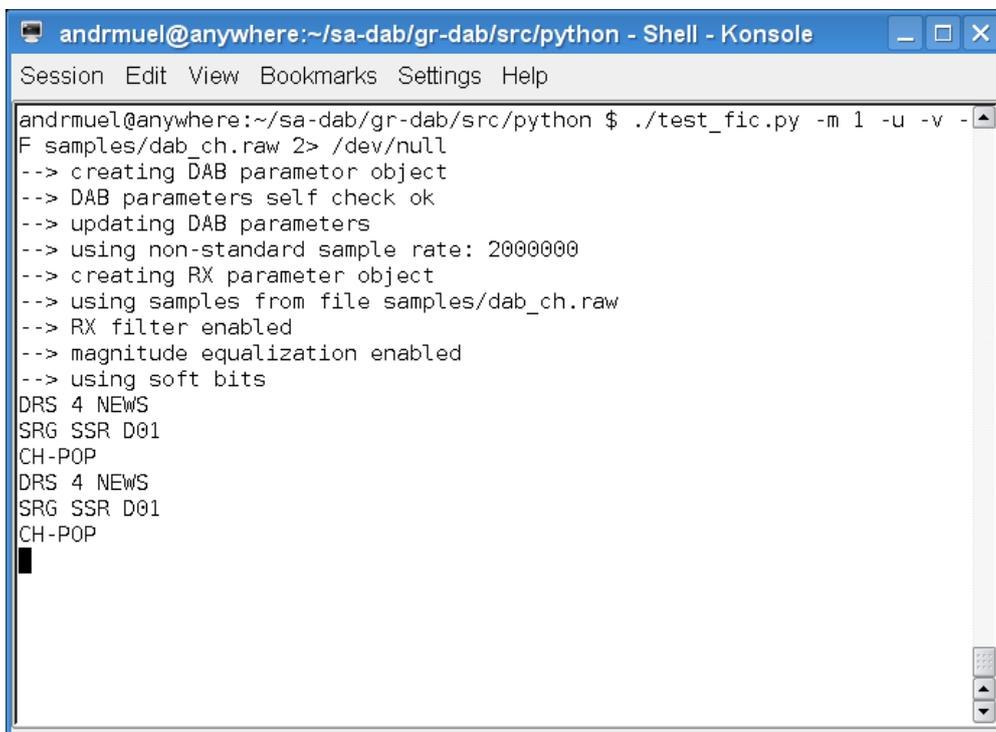
6.4.3 Energy Dispersal Scrambling

As the XOR operation is symmetric, the procedure to undo energy dispersal is exactly the same as the one applied in the transmitter – a PRBS that is generated according to the DAB standard is added modulo 2 to the signal.

6.4.4 FIB Sink

The FIB sink simply collects and sorts the information from the FIBs.

Figure 6.21 shows an application that demodulates a DAB signal from samples in a file and prints out the station labels from the received FIBs.



```
andrmuel@anywhere:~/sa-dab/gr-dab/src/python - Shell - Konsole
Session Edit View Bookmarks Settings Help
andrmuel@anywhere:~/sa-dab/gr-dab/src/python $ ./test_fic.py -m 1 -u -v -
F samples/dab_ch.raw 2> /dev/null
--> creating DAB parameter object
--> DAB parameters self check ok
--> updating DAB parameters
--> using non-standard sample rate: 20000000
--> creating RX parameter object
--> using samples from file samples/dab_ch.raw
--> RX filter enabled
--> magnitude equalization enabled
--> using soft bits
DRS 4 NEWS
SRG SSR D01
CH-POP
DRS 4 NEWS
SRG SSR D01
CH-POP
█
```

Figure 6.21: Station labels extracted from a DAB signal.

Chapter 7

Conclusions and Outlook

7.1 Conclusions

One of the major challenges when implementing a software defined radio is the limited amount of available processing time. The majority of papers about signal processing algorithms is written from a primarily mathematical perspective, and many of these algorithms are not suitable for a software implementation. Therefore, the choice often has to be made in favor of the faster algorithm, rather than the one with the best receiving performance. On the other hand, computers are getting faster and faster. Even though the project of this thesis was developed and tested on a laptop with a single core CPU, CPUs with four and more cores are available today.

With all features (input filtering, frequency correction, magnitude equalisation, ..) except dynamic resampling¹ turned on, the presented implementation of DAB in GNU Radio requires about 95% of the available CPU time to demodulate the complete OFDM signal and decode the FIC in real-time (using a five years old ThinkPad laptop with an 1.6 GHz CPU). Without the FFT filter at the input, even only about 80% of the available processing power is required. Since most of the processing power is spent for OFDM demodulation, a complete real-time software DAB receiver is certainly feasible on current hardware.

A big advantage for implementing an algorithm in software, is the open availability of existing code for a similar problem. Often, only minor adjustments of existing code are needed to reuse it for a new problem. While making such adjustments still requires a complete understanding of the used algorithm, the availability of open code for a similar problem facilitates both the process of learning and implementing the algorithm.

In the case of GNU Radio, all source code, including the Verilog code for the FPGA, is available for inspection and reuse, which was very helpful.

7.2 Outlook

To finally be able to hear an audio signal, the implementation of the MSC is needed. As many blocks required for this have already been implemented for the FIC, this should not be too much work.

Once the receiver is complete, a comparison to hardware receivers would be interesting, to evaluate the receiving performance practically, rather than only with simulations as described in Section 6.3. On the other hand, the existing test bench and simulation code could easily be adapted to do other interesting simulations. For instance, the performance of SFNs could be evaluated by using an appropriate FIR tap for each transmitter.

¹The current fractional resampler is too slow to run in real-time together with the rest of the code; the USRP is however accurate enough to not require sampling frequency correction.

To save some of the required processing power, it would also be interesting to see, whether the receiver can be extended, such that it adapts itself dynamically to the receiving conditions. For instance, if the conditions are good and the channel changes only slowly, an update of the magnitude equalisation factors for each new frame may not be necessary, and it may be done less often. The ultimate idea behind software defined radio is to have a cognitive radio, which is completely aware of its [RF](#) environment, and adapts all its parameters whenever needed.

Finally, it would also be interesting to implement a [DAB](#) transmitter. Since a transmitter does not require such things as frequency synchronisation, its implementation is often much easier than the implementation of the corresponding receiver. As the physical layer has already been implemented and most blocks for the upper layers are already available, the implementation of a [DAB](#) transmitter in GNU Radio should be comparatively easy. With a complete transmitter and receiver implementation, [DAB](#) could also be used to transport other data, for instance to build a wireless data network between two computers equipped with [USRP](#)s operating in the 2.4 GHz [ISM](#) band.

Appendix A

Signal Flow Graphs

The following pages contain flow graphs from various parts of the [DAB](#) code. The content of these blocks and the underlying algorithms are explained in [Chapter 6](#).

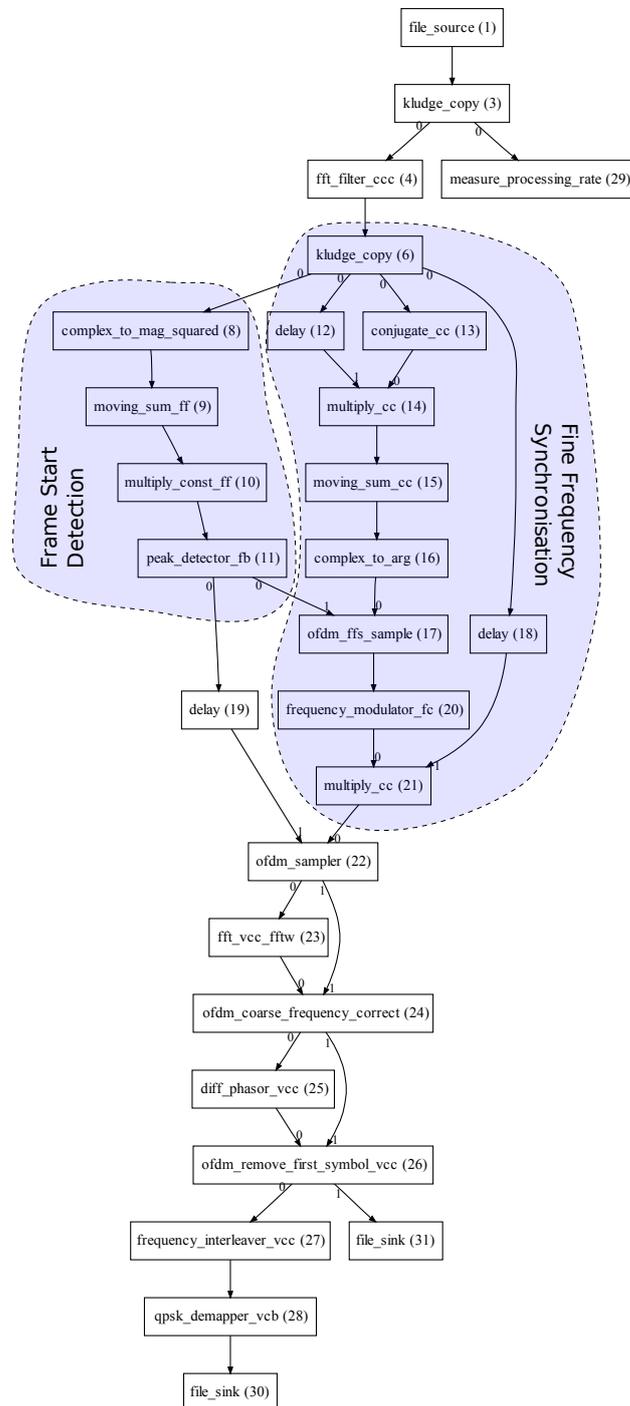


Figure A.1: First version of OFDM demodulation.

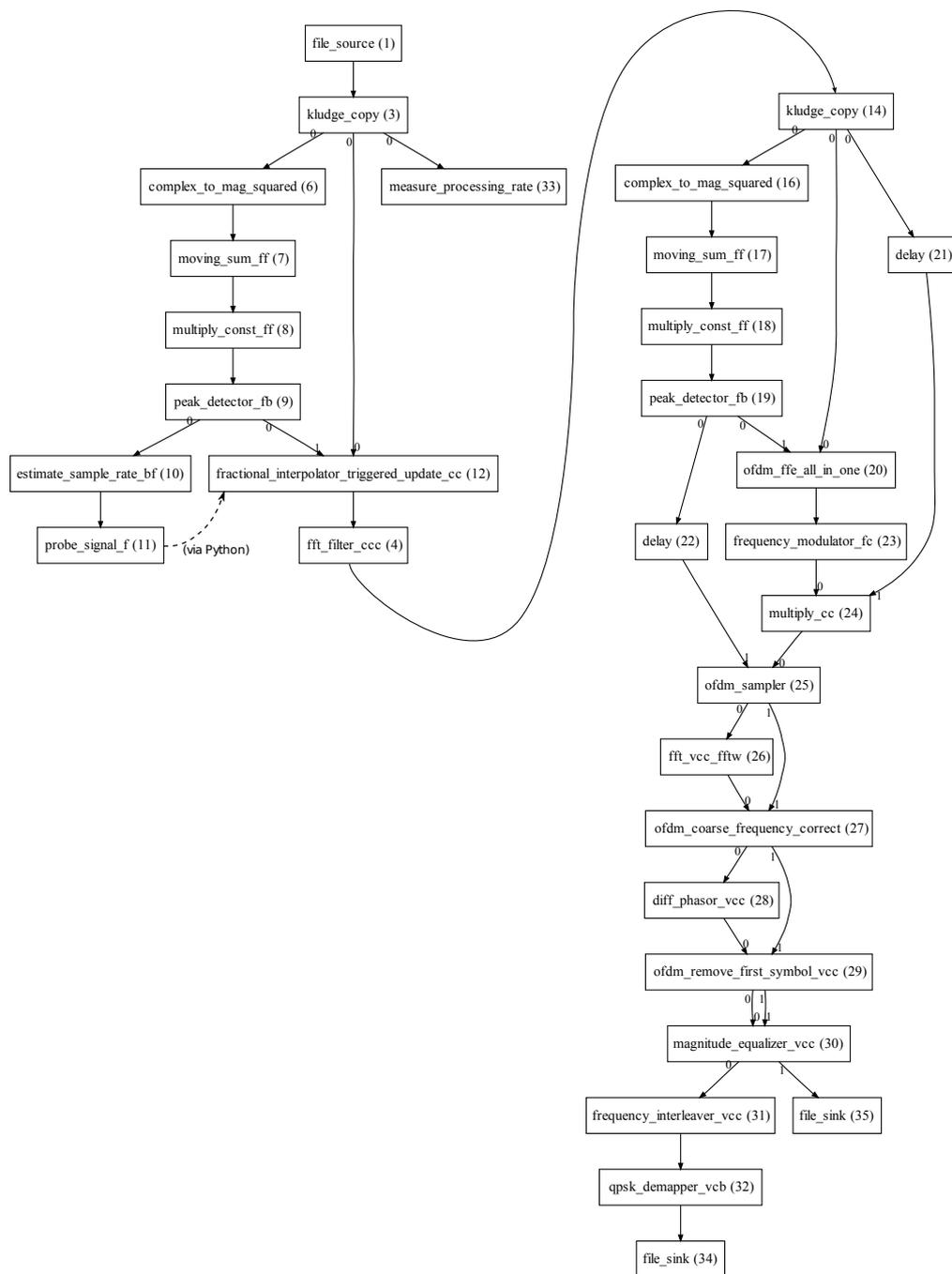


Figure A.2: Revised version of OFDM demodulation: With resampling, magnitude equalization and a single block for fine frequency estimation.

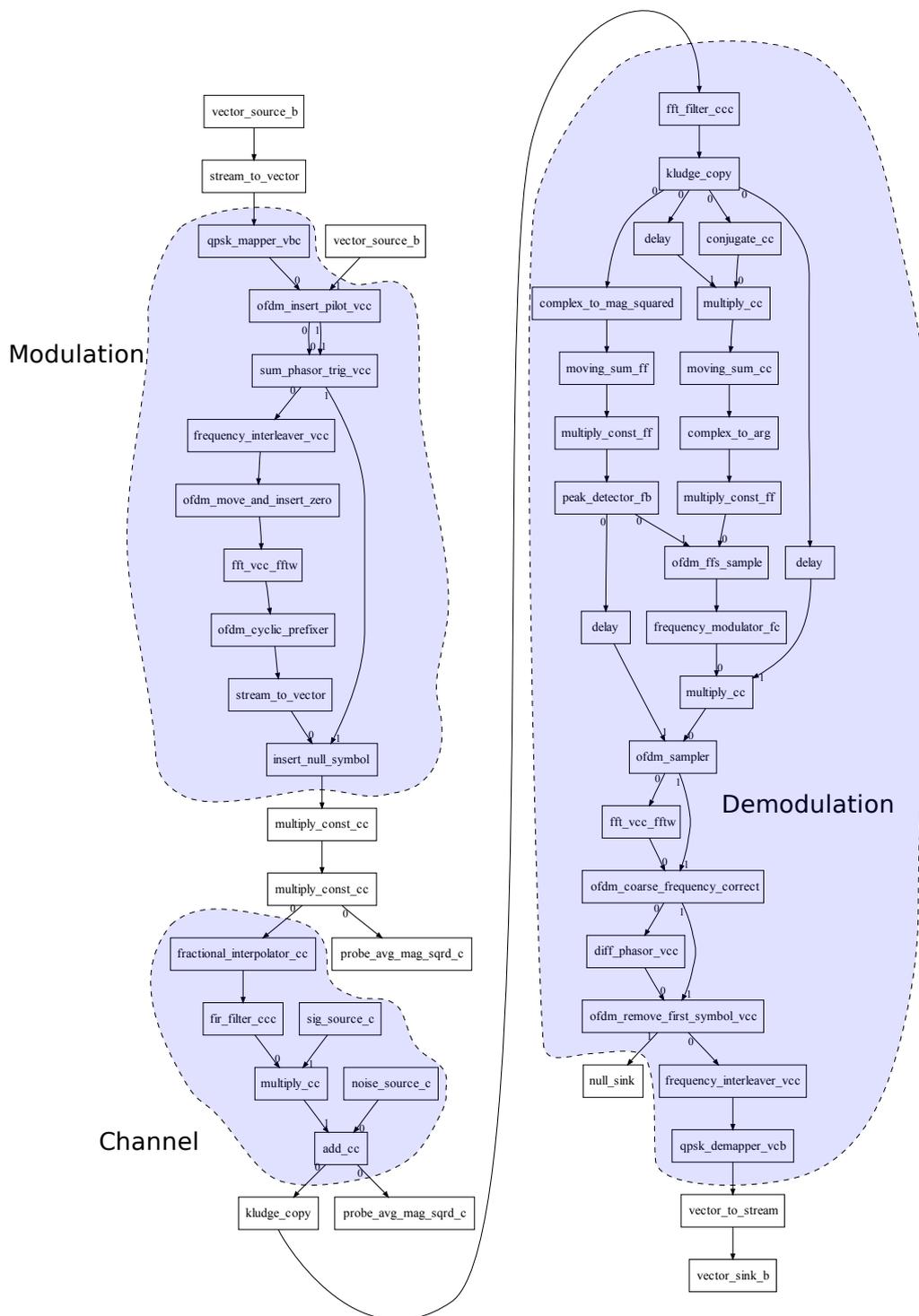


Figure A.3: DAB OFDM test bench: Modulation, channel model and demodulation.

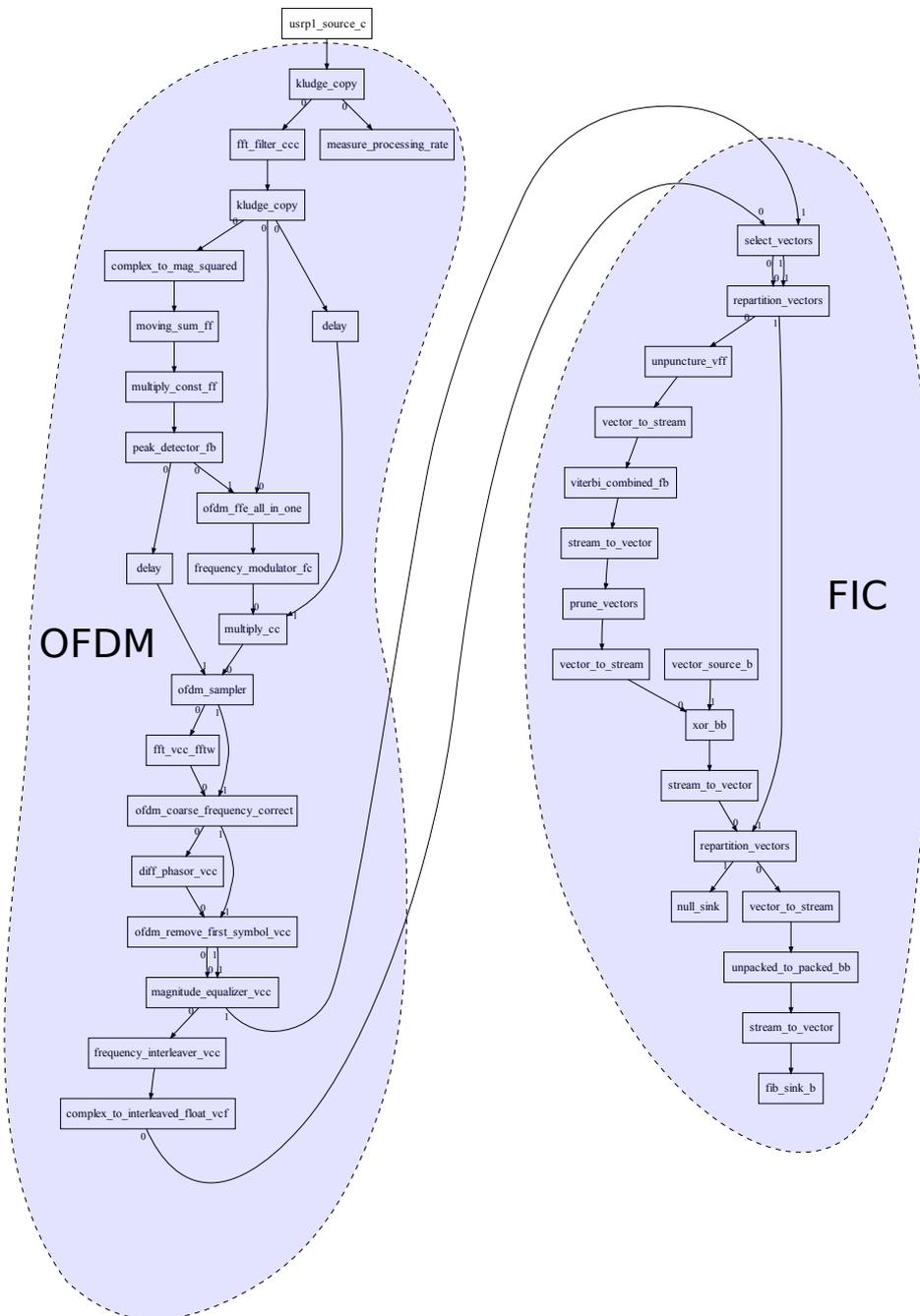


Figure A.4: Complete receiver with USRP as signal source, OFDM demodulation and FIC decoding.

Appendix B

FFTW Speed Evaluation

As the [USRP](#) can not be used to sample directly at 2.048 [MSPS](#) (which is the inverse of the fundamental time unit specified in the [DAB](#) standard), the nearest possible sample rate, 2 [MSPS](#), is used instead. Therefore, the signal must either be resampled, or the code must be able to use any sample rate. The latter option would also require an [FFT](#) of length 2000 instead of 2048.

To answer the question, which option is more computationally intensive, the speed of the [FFT](#) library, [FFTW](#), has to be evaluated. According to the [FFTW](#) website, [FFTW](#) has a complexity of $O(n \log n)$ for any length, even for primes. This does however not mean, that the runtime is completely independent of the [FFT](#) length, but rather that the speed difference is only a constant factor and not some function of the number of processed samples.

To evaluate the speed for the lengths between 1950 and 2050, a small test program in Python was created. The result can be seen in [Figure B.1](#).

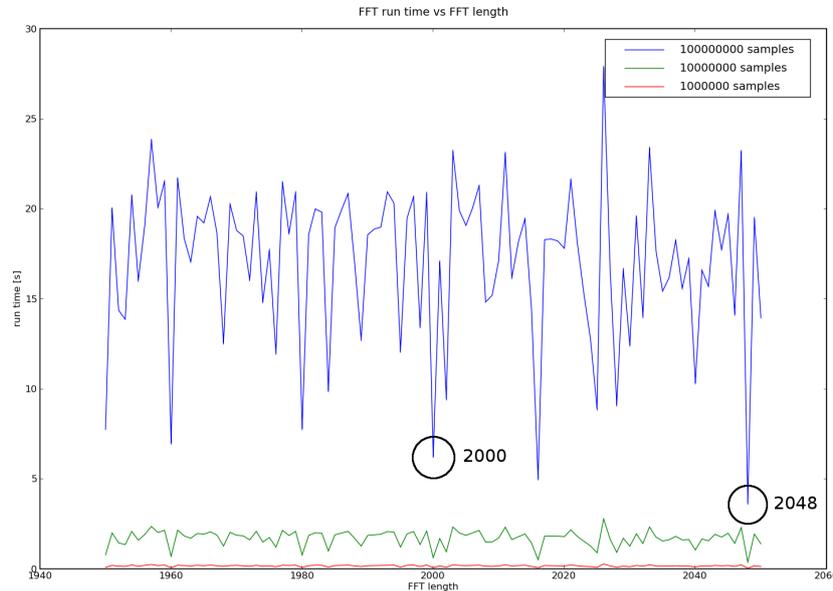


Figure B.1: FFT speed evaluation.

It should be noted that the relative speed between different [FFT](#) sizes is also influenced heavily by the architecture of the computer used (e.g. by the cache block size).

Table [B.1](#) lists some of the run times from a simulation with 100'000'000 samples on two

different PCs,

- a) a ThinkPad with an Intel Pentium M processor clocked at 1.6 GHz (with real-time scheduling enabled and an SSE optimized FFTW library).
- b) a Tardis (a student PC with a Dual Core Intel Pentium 4 CPU clocked at 3.2 GHz, from which only one core was used)

The fastest result is in both cases the one with an FFT of length 2048.

	2048	2000	Average runtime	Longest runtime (length)
ThinkPad	3.66s	6.25s	17.07s	27.84s (2026)
Tardis	18.83s	19.95s	49.79s	191.99s (2039)

Table B.1: FFT runtime evaluation results

Although this shows up to a factor of 10 between the ideal case and the worst case, an FFT of length 2000 is reasonably fast to justify its use. In total, the FFT block still uses less than 10% of the available CPU time, whereas resampling alone would need about 80% (and an FFT of length 2048 would still be required in that case).

Appendix C

Additional Tools

This chapter briefly introduces some of the additional tools used for development and testing. As good information on all of these tools is available online and they are not subject of this thesis, the description is deliberately kept short. This should hopefully still suffice to understand, how each tool fits into the development process.

All software presented in this chapter is open source software, which is freely available.

C.1 OProfile

OProfile is a statistical profiler for Linux, which can be used for performance analysis. OProfile allows the profiling of programs by collecting statistical data with the aid of a kernel module. Based on events, such as timer events, cache misses or memory references, samples (i.e. information about the code which is currently being executed) are recorded.

Once all code has been executed, these samples can be retrieved from the OProfile daemon for further processing. This allows a programmer to quantitatively analyze, how much computing power is spent in some area of her code, relative to other parts. Besides analyzing single program runs, it is possible to get differential information about multiple program runs, e.g. to find out, whether a change in the code resulted in a performance increase.

Because OProfile uses a kernel module and hardware performance counters of the CPU, profiling a program or a library requires no recompilations¹ or alike, which makes profiling very convenient. This is especially useful for a toolkit like GNU Radio, where the mix of Python and C++ code would otherwise complicate performance analysis.

Besides OProfile, standard Unix tools, such as `time` or `top` were used to evaluate the efficiency of the code.

Further information on performance analysis can be found in [22]. Information about OProfile can be found on the OProfile website [23], as well as in [24].

C.2 Python, IPython and SciPy

IPython is an advanced interactive shell for the Python programming language. Although the Python interpreter comes with an interactive shell itself, IPython provides many advanced features, such as completion or history, which are very useful for Python development. For GNU Radio development, IPython can be used to manually test a signal processing block.

¹Although not necessary, recompilation with debug information (i.e. for `gcc` with `-g`) is useful for code annotation, however.

SciPy is a library, which provides many mathematical functions. Together with IPython, this library can be used as an interactive Computer Algebra System (CAS), similar to Matlab, but with Python as its underlying programming language.

More information about Python, IPython and SciPy can be found under <http://python.org/>, <http://ipython.scipy.org/> and <http://scipy.org> respectively.

C.3 Doxygen

Doxygen is a tool, which can be employed to extract comments from the source code of various programming languages and create a code manual in Hypertext Markup Language (HTML), L^AT_EX, or a number of other output formats.

As a good documentation of the source code is important, it is preferable to document the source code and let Doxygen create a code manual, rather than to write a separate manual.

More information on Doxygen is available on <http://www.doxygen.org/>.

C.4 Swig

Simplified Wrapper and Interface Generator (SWIG) is a tool which can automatically create interfaces between two different programming languages. In the case of GNU Radio, SWIG is used to create bindings between Python and C++ blocks.

Information about SWIG can be found at <http://www.swig.org/>.

C.5 Graphviz, dot and dump2dot

Graphviz is a graph visualization software package, which can be used to create visual graphs from a textual representation. Specifically, the program dot from this package creates directed graphs.

GNU Radio provides a function to dump textual information about the structure of a running program. For this thesis, a small Python script called dump2dot was written, which converts this information into a format suitable for dot. This allows the automatic generation of graphical representations of a signal flow graph.

More information about Graphviz and dot can be found under <http://www.graphviz.org/>.

C.6 Subversion

Subversion is a version control system, i.e. it can be used to keep track of changes made in source code or other documents. Whenever the code is checked in, its current state is stored on the Subversion server. This is very helpful if some change in the code breaks the functionality, and it also has the benefit, that a backup is always available.

Subversion is available from <http://subversion.tigris.org/>.

Appendix D

Code Overview

This section gives an overview of the program code. It does however not intend to describe the purpose of the individual blocks and their functions, as this information is available in the [HTML](#) documentation generated by *Doxygen*.

D.1 Python Code

The Python code is located in the directory `gr-dab/src/python`.

The main files are `ofdm.py`, which implements modulation (class `ofdm_mod`) and demodulation (class `ofdm_demod`) for the [DAB](#) physical layer and `fic.py`, which provides the class `fic_decode`, to decode the [FIC](#).

The following executables exist in `gr-dab/src/python/`:

- `dab_estimate_samplerate.py` – estimates the exact sample rate of samples from a file
- `dab_rx_constellation.py` – visual real-time constellation display of samples from the [USRP](#) or from a file
- `usrp_dab_rx.py` – receive live [FIC](#) information with the [USRP](#)
- `test_fic.py` – evaluate [FIC](#) information of samples from a file
- `test_ofdm.py` – demodulate [OFDM](#) layer of samples from a file
- `test_ofdm_sync_dab.py` – test the synchronisation code with samples from a file

All executables have built-in help pages, which can be displayed by calling the program with the parameter `-h`, e.g. for `dab_rx_constellation.py`:

```
$ ./dab_rx_constellation.py -h
usage: dab_rx_constellation.py: [options] <filename>
```

options:

```
-h, --help                show this help message and exit
-m DAB_MODE, --dab-mode=DAB_MODE
                           DAB mode [default=1]
-F, --filter-input        enable FFT filter at input
-c, --correct-ffe         do fine frequency correction
-u, --correct-ffe-usrp    do fine frequency correction by retuning the USRP
```

```

        instead of in software
-e, --equalize-magnitude
        do magnitude equalization
-s RESAMPLE_FIXED, --resample-fixed=RESAMPLE_FIXED
        resample by a fixed factor (fractional interpolation)
-S, --autocorrect-sample-rate
        estimate sample rate offset and resample (dynamic
        fractional interpolation)
-R RX_SUBDEV_SPEC, --rx-subdev-spec=RX_SUBDEV_SPEC
        select USRP Rx side A or B [default=A]
-f FREQ, --freq=FREQ set frequency to FREQ [default=227360000.0]
-r SAMPLE_RATE, --sample-rate=SAMPLE_RATE
        set sample rate to SAMPLE_RATE [default=2000000]
-d DECIM, --decim=DECIM
        set decimation rate to DECIM [default=32]
-g RX_GAIN, --rx-gain=RX_GAIN
        set receive gain in dB (default is midpoint)
-v, --verbose
        verbose output

```

Another important file is `parameters.py`. This file contains two classes, `dab_parameters`, which contains all parameters from the [DAB](#) standard, and `receiver_parameters`, which contains all parameters of the receiver. `dab_parameters` also has many built-in self checks to verify the [DAB](#) parameters.

D.1.1 Quality Assurance

The Python files in the directory `gr-dab/src/python/qa` are unit tests, to verify the correct functioning of the blocks implemented in C++.

This is done in a way similar to how test benches are used in Very Large Scale Integration ([VLSI](#)) design, by specifying vectors with stimuli and expected results. To check a block, the appropriate stimuli are run through it and compared to the expected results.

This code can be executed with the `make check` command, which should always be done between `make` and `make install`, and after changes in the source code.

D.1.2 Channel Tests

The Python files found under `gr-dab/src/python/channel_tests` provide tools to test channel effects and evaluate properties, such as [SNR](#) vs [BER](#). These tests are documented in [Section 6.3](#).

The following four scripts exist, to evaluate different effects:

- `plot_snr_ber.py` – evaluates the effect of noise
- `plot_freq_shift_ber.py` – evaluates the effect of frequency shifts
- `plot_multipath_ber.py` – evaluates the effect of multipath propagation
- `plot_sampling_rate_offset_ber.py` – evaluates the effect of an offset in the sampling frequency

All four scripts use the [DAB](#) test bench from the file `dab_tb.py`, which is depicted in [Figure A.3](#).

D.2 C++ Code

The source code for the C++ blocks is located in the directory `gr-dab/src/lib`. Each block consists of header file (`*.h`) and the corresponding implementation (`*.cc`).

The bindings between C++ and Python are created with Swig. The file `dab.i` in `gr-dab/src/lib` controls, how the bindings for each block are created.

D.3 Patches

The directory `gr-dab/patches` contains some patches for blocks in the GNU Radio framework.

Appendix E

Installation

This chapter gives an overview of how GNU Radio and the code developed in this thesis can be installed. As GNU Radio is a rather complex toolkit, the first installation of GNU Radio and associated code can be cumbersome. The GNU Radio community has however written some good documents on how to install GNU Radio. Rather than reproducing them, they shall be linked here:

- Readme: <http://gnuradio.org/trac/browser/gnuradio/trunk/README>
- Build guide: <http://gnuradio.org/trac/wiki/BuildGuide>

More documents can be found in the GNU Radio Wiki at <http://gnuradio.org/trac/>.

E.1 Operating System

While GNU Radio can be run under many operating systems, including Mac OS X, NetBSD and Windows, Linux tends to be the easiest to use.

E.2 Packages

Some distributions provide packages for GNU Radio. As the [DAB](#) module relies on some of the newer code found in the trunk, an installation from the code in the [SVN](#) repository is required. The packages may however be useful to pull in the required dependencies.

E.3 External Dependencies

GNU Radio already has quite a few dependencies, which are listed in the Readme. The [DAB](#) module does not introduce any new dependencies itself; but the channel testing code requires two additional Python modules:

- Scipy (available from <http://scipy.sourceforge.net>)
- Matplotlib (available from <http://matplotlib.sourceforge.net>)

E.4 Installation

Once all external dependencies are satisfied, the build and install process is – in the ideal case – as easy as

```
$ ./bootstrap
```

```
$ ./configure
```

```
$ make
```

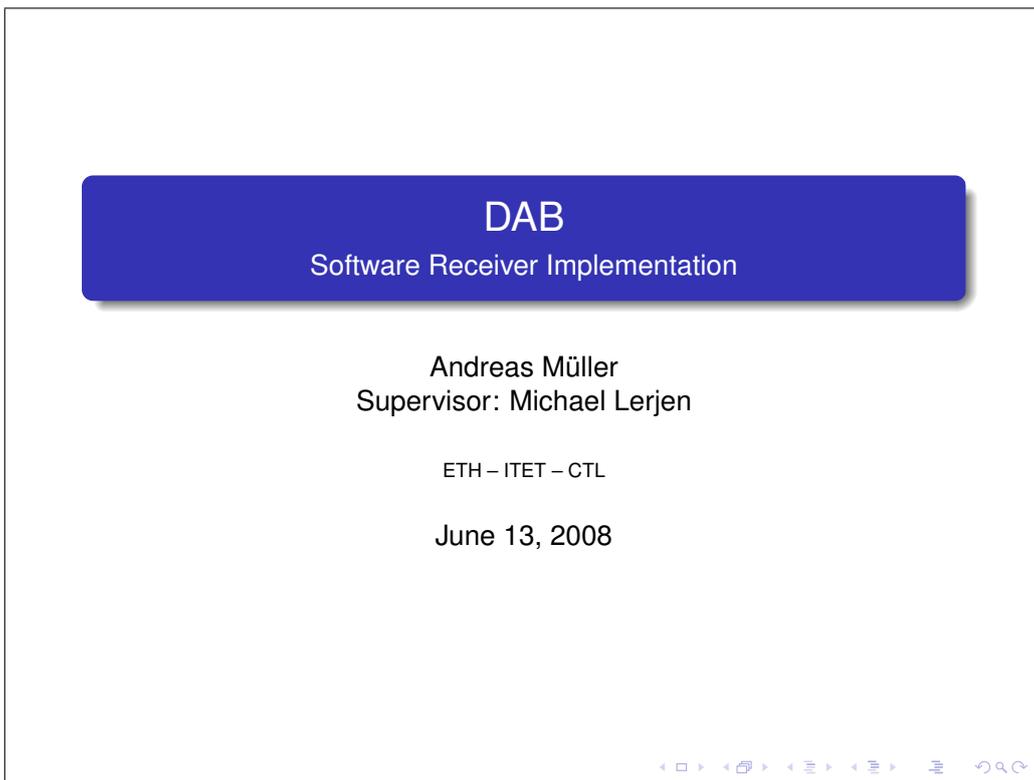
```
$ make check
```

```
$ sudo make install
```

Appendix F

Presentation

On the following pages, the presentation slides are included, which were used for the presentation on June 13th, 2008.



DAB
Software Receiver Implementation

Andreas Müller
Supervisor: Michael Lerjen

ETH – ITET – CTL

June 13, 2008

Navigation icons: back, forward, search, etc.

- 1 Introduction
 - Task
 - Software Defined Radio
 - DAB
 - GNU Radio and USRP
- 2 Implementation
 - OFDM Synchronisation
 - OFDM Demodulation
- 3 Evaluation
 - Test Setup
 - Results
- 4 Conclusions
- 5 Questions



Introduction ●○○○○○ Implementation ○○ Evaluation ○○○ Conclusions Questions

Task

Goal: Implementation of a real-time DAB receiver as SDR

SDR
Software Defined Radio → (almost) all signal processing in software

DAB
Digital Audio Broadcasting → digital radio technology standardized by ETSI

Real-time
Process data as fast as it arrives → 2 MSPS or 16 MB/s

Introduction ●○○○○○ Implementation ○○ Evaluation ○○○ Conclusions Questions

Software Defined Radio

Idea

Digitize the signal and do all the signal processing in (high level, architecture independent) software.

Strengths

- Flexibility
- Reusable code, fast development cycle
- Cognitive radio: Adapts itself dynamically to RF environment → better spectral and power efficiency

Weaknesses

- Limited sample rate and dynamic range of ADCs and DACs → analog front end needed for filtering
- Resource usage, energy consumption, cost

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺ ↻

Introduction ○●○○○○ Implementation ○○ Evaluation ○○○ Conclusions Questions

Digital Audio Broadcasting (DAB) Specification

Modes

Four modes for different frequency ranges and RF characteristics

- Presentation: Mode I (Code: All Modes)

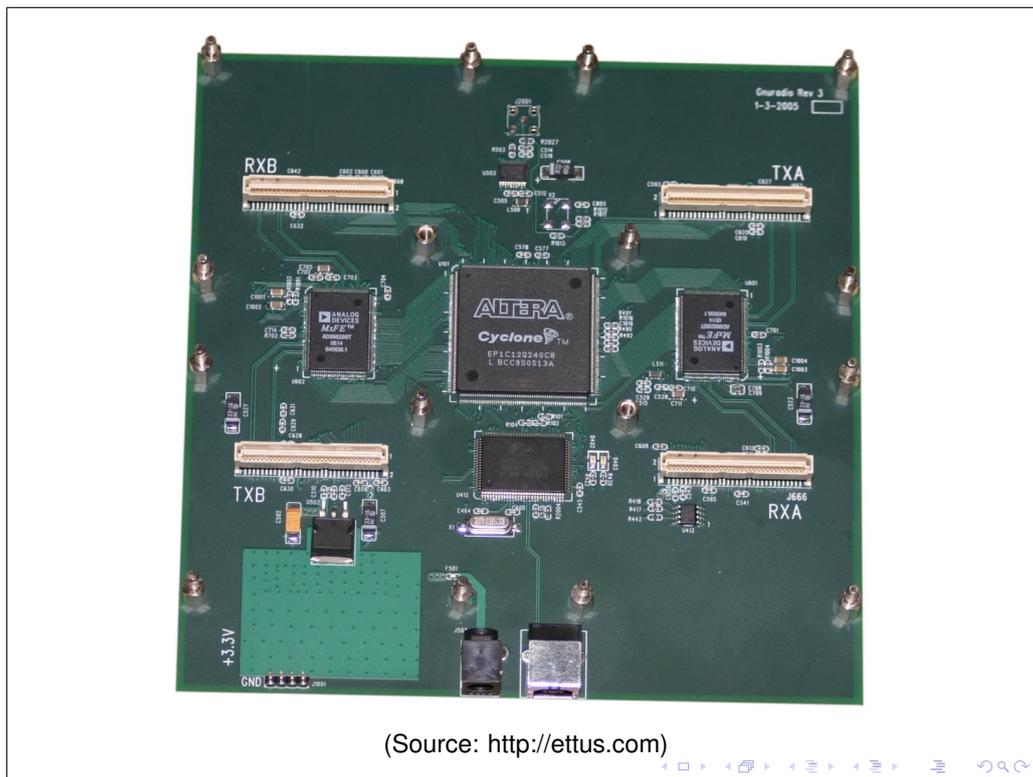
DAB Mode I OFDM signal

- Frames with 76 OFDM symbols (1 pilot, 75 data)
- Null symbols (energy zero) to separate frames
- 1536 subcarriers à 1 kHz & central carrier zero → 1.537 MHz
- D-QPSK modulation for each subcarrier
- Cyclic prefix: 504 samples → SFN with max. TX distance 74 km

Upper Layers

- Punctured convolutional coding
- Energy dispersal, Time interleaving
- MPEG 2 audio coding

🔍 ↺ ↻



Introduction ○○○○○○ Implementation ●○○ Evaluation ○○○ Conclusions Questions

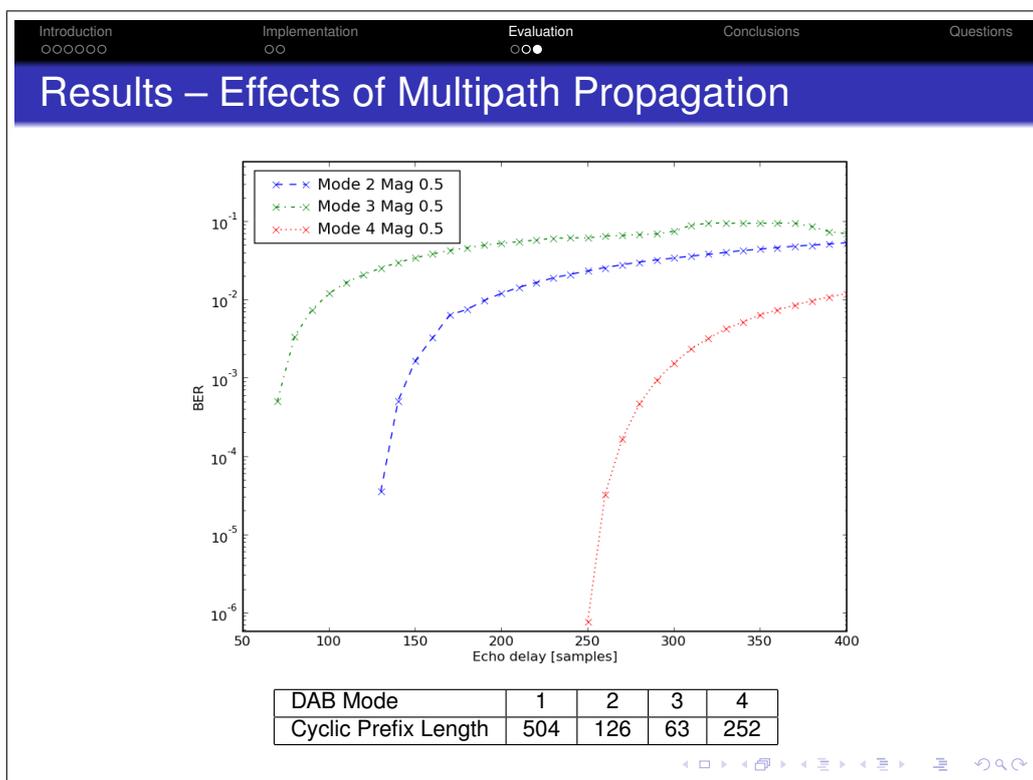
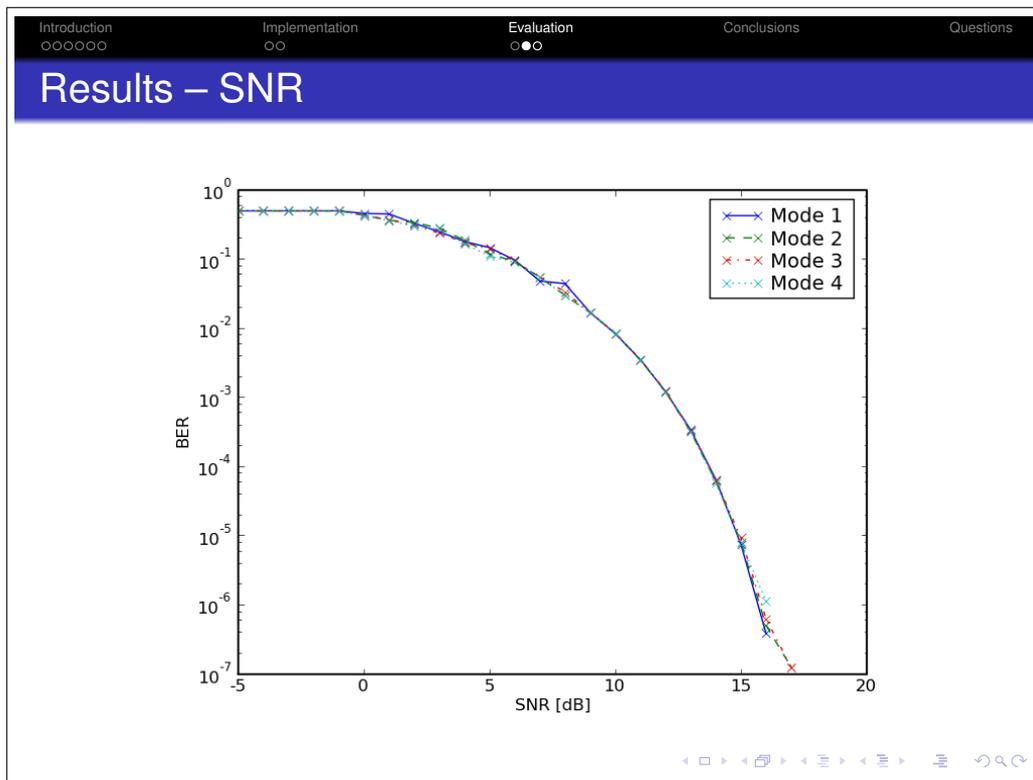
OFDM I – Synchronisation

Time Synchronisation

- Frame start detection must be accurate, as the other blocks depend on it
- Can easily be done by looking at the energy of the signal (Null symbols)
- Implemented with moving sum, inverter and peak detector

Frequency Synchronisation

- Small subcarrier bandwidth → accurate synchronisation needed
- Fine frequency synchronisation (offsets < subcarrier bandwidth)
 - compare cyclic prefix to end of the symbol → fine frequency offset can be estimated from the phase offset
- Coarse frequency synchronisation (offsets > subcarrier bw)
 - done after fine frequency synchronisation and after FFT
 - simply shift signal in the frequency domain → very efficient



Introduction ○○○○○○ Implementation ○○ Evaluation ○○○ Conclusions Questions

Conclusions

Conclusions

- Real-time processing is possible
- FIBs successfully decoded
- No audio yet

Challenges

- Very efficient algorithms and programming needed
- Many signal processing papers are written from a primarily mathematical perspective

Advantages

- Same code for simulation and actual receiver
- Open source code of existing blocks helps understand algorithms
- Existing code can sometimes be adapted for new purposes
- GNU Radio: Large and enthusiastic community



Introduction ○○○○○○ Implementation ○○ Evaluation ○○○ Conclusions Questions

Questions?

Thank you for your attention.



Appendix G

CD-ROM Content Listing

This semester thesis is accompanied by a CD-ROM, which has the following content:

/	
gr-dab Python and C++ code for the DAB GNU Radio module
doc/index.html Source code documentation (generated by Doxygen)
task Task description
thesis This semester thesis as Portable Document Format (PDF) file
presentation Presentation slides as PDF file
tools Additional tools developed for this thesis
samples Some DAB samples
test_results Detailed results (log files) of the tests with the channel model
fft_eval Script and log files from FFT speed evaluation

Appendix H

Acronyms

The following list provides an overview of the acronyms used throughout this thesis.

AAC Advanced Audio Coding

ADC Analog Digital Converter

AGC Automatic Gain Control

ASIC Application Specific Integrated Circuit

AWGN Additive White Gaussian Noise

BER Bit Error Rate

CA Conditional Access

CAS Computer Algebra System

CIC Cascaded Integrator Comb

CIF Common Interleaved Frame

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CTL Communication Technology Laboratory

CU Capacity Unit

DAB Digital Audio Broadcasting

DAC Digital Analog Converter

DBSRX Direct Broadcast Satellite Receiver

DDC Digital Down-Converter

DECT Digital Enhanced Cordless Telecommunications

D-QPSK Differential Quadrature Phase Shift Keying

DRM Digital Radio Mondiale

DSP Digital Signal Processor

DUC Digital Up-Converter

ETSI European Telecommunications Standards Institute

FFT Fast Fourier Transform

FFTW Fastest Fourier Transform in the West

FIB Fast Information Block

FIC Fast Information Channel

FIG Fast Information Group

FIR Finite Impulse Response

FM Frequency Modulation

FPGA Field Programmable Gate Array

FSM Finite State Machine

GNU GNU's Not Unix

GPL General Public License

GPS Global Positioning System

GR GNU Radio

GRC GNU Radio Companion

GSM Global System for Mobile communications

HTML Hypertext Markup Language

ICI Inter Carrier Interference

IF Intermediate Frequency

IIR Infinite Impulse Response

ISM Industrial, Scientific and Medical

LFSR Linear Feedback Shift Register

MCI Multiplex Configuration Information

MPEG Moving Picture Experts Group

MP2 MPEG-1 Audio Layer II

MSC Main Service Channel

MSPS Mega Samples Per Second

OFDM Orthogonal Frequency Division Multiplexing

PCI Peripheral Component Interconnect

PDF Portable Document Format

ppm parts per million

PRBS Pseudo Random Binary Sequence

PRS Phase Reference Symbol

QA Quality Assertion

QPSK Quadrature Phase-Shift Keying

RF Radio Frequency

SC Service Component

SDR Software Defined Radio

SFN Single Frequency Network

SIMD Single Instruction, Multiple Data

SNR Signal to Noise Ratio

SR Software Radio

SSE Streaming SIMD Extensions

SI Service Information

SVN Subversion

SWIG Simplified Wrapper and Interface Generator

TII Transmitter Identification Information

TVRX Television Receiver

UDP User Datagram Protocol

UHF Ultra High Frequency

USB Universal Serial Bus

USRP Universal Software Radio Peripheral

VHF Very High Frequency

VLSI Very Large Scale Integration

Bibliography

- [1] ETSI, “ETSI EN 300 401 V1.4.1: Radio Broadcasting Systems; Digital Audio Broadcasting to mobile, portable and fixed receivers,” 2006. [Online]. Available: <http://www.etsi.org>
- [2] —, “ETSI ES 201 980 V2.3.1: Digital Radio Mondiale (DRM); System Specification,” 2008. [Online]. Available: <http://www.etsi.org>
- [3] Wikipedia, “Digital Audio Broadcasting.” [Online]. Available: http://en.wikipedia.org/wiki/Digital_Audio_Broadcasting
- [4] —, “Digital Radio Mondiale.” [Online]. Available: http://en.wikipedia.org/wiki/Digital_Radio_Mondiale
- [5] V. Fischer, “Software Implementation of a Digital Radio Mondiale (DRM) Receiver, Part I (Framework),” *Institute for Communication Technology, Darmstadt University of Technology*. [Online]. Available: [http://drm.sourceforge.net/papers/Software_Implementation_of_a_Digital_Radio_Mondiale_\(DRM\)_Receiver,_Part_I_\(Framework\).pdf](http://drm.sourceforge.net/papers/Software_Implementation_of_a_Digital_Radio_Mondiale_(DRM)_Receiver,_Part_I_(Framework).pdf)
- [6] J. P. Elsner, *Implementation of the DAB physical layer in software using the GNU Radio framework*. Universität Karlsruhe, 2007.
- [7] O. Edfors, M. Sandell, J. van de Beek, D. Landström, and F. Sjöberg, “An Introduction to Orthogonal Frequency-Division Multiplexing,” 1996.
- [8] Wikipedia, “Loudness war — Wikipedia, The Free Encyclopedia,” 2008. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Loudness_war
- [9] D. J. M. Robinson, “OFCOM: Regulation in digital broadcasting: DAB digital radio bitrates and audio quality; Dynamic range compression and loudness,” *Department of Electronic Systems Engineering, University of Essex*, 2002. [Online]. Available: <http://www.david.robinson.org/commsbill/>

- [10] J. Mitola, “What is a Software Radio?” [Online]. Available: <http://web.archive.org/web/20050315234159/http://ourworld.compuserve.com/homepages/jmitola/whatisas.htm>
- [11] V. Bose, M. Ismert, M. Welborn, and J. Gutttag, *Virtual Radios*. Massachusetts Institute of Technology, 1998.
- [12] J. Mitola, “Cognitive Radio – An Integrated Agent Architecture for Software Defined Radio,” 2000. [Online]. Available: http://www.it.kth.se/~maguire/jmitola/Mitola_Dissertation8_Integrated.pdf
- [13] E. Blossom, “GNU Radio: Tools for Exploring the Radio Frequency Spectrum,” *Linux Journal*, 2004. [Online]. Available: <http://www.linuxjournal.com/article/7319>
- [14] “GNU Radio project.” [Online]. Available: <http://gnuradio.org/trac>
- [15] Q. Norton, “GNU Radio Opens an Unseen World,” *Wired Magazine*, 2006. [Online]. Available: <http://www.wired.com/science/discoveries/news/2006/06/70933>
- [16] E. Blossom, “How to Write a Signal Processing Block.” [Online]. Available: <http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html>
- [17] Wikipedia, “CORDIC — Wikipedia, The Free Encyclopedia,” 2008. [Online]. Available: <http://en.wikipedia.org/w/index.php?title=CORDIC&oldid=201345871>
- [18] G. R. Trac, “USRP FPGA,” 2008. [Online]. Available: <http://gnuradio.org/trac/wiki/UsrcFPGA>
- [19] M. Sliskovic, “Carrier and Sampling Frequency Offset Estimation and Correction in Multi-carrier Systems,” 2001.
- [20] T. M. Schmidl and D. C. Cox, “Robust Frequency and Timing Synchronization for OFDM,” *Signal Processing, IEEE Transactions on Communications*, vol. 45, no. 12, pp. 1613–1621, Dec 1997.
- [21] J. van de Beek, M. Sandell, and P. Borjesson, “ML estimation of time and frequency offset in OFDM systems,” *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, vol. 45, no. 7, pp. 1800–1805, Jul 1997. [Online]. Available: <http://epubl.luth.se/avslutade/0347-0881/96-09/bsb96r.pdf>

-
- [22] Wikipedia, “Performance Analysis — Wikipedia, The Free Encyclopedia,” 2008. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Performance_analysis&oldid=201033250
- [23] J. Levon, “OProfile – A System Profiler for Linux.” [Online]. Available: <http://oprofile.sourceforge.net/>
- [24] W. E. Cohen, “Tuning Programs with OProfile,” *Wide Open Magazine*, 2008. [Online]. Available: <http://people.redhat.com/wcohen/Oprofile.pdf>