HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Bachelor Thesis, Institute for Software

Data over DAB

University of Applied Sciences Rapperswil

Spring Semester 2016
February 24th, 2016

Supervisor:	Prof. Dr. Farhad Mehta
Authors:	Felix Morgner & Tobias Stauber
Technical Advisor:	Matthias Brändli
Partner Company:	Opendigitalradio
Duration:	02/24/2016 - 06/17/2016
Extent of work:	360 Hours, 12 ECTS per student
Website:	`http://thesis.dab.arknet.ch`

Abstract

Digital Audio Broadcasting (DAB) is a radio broadcast technology widely used in Europe and asian regions. Despite the popularity, there is no open-source tool that allows the transmission of arbitrary data over DAB.

Most of the currently existing professional Software Defined Radio (SDR) transceivers are capable to encode and pack raw DAB data in hardware like Field Programmable Gate Arrays (FPGAs). Affordable DAB receivers that do not support hardware decoding produce just complex Orthogonal Frequency-Division Multiplexing (OFDM) samples. There exists therefore a large gap for third party applications that want to use DAB to transmit and receive arbitrary data.

The aim of this thesis is to find ways in which third party applications can easily use DAB to communicate. Apart from a review of possible solutions, we provide an open-source tool set that can be used for this purpose. It was especially challenging to obtain the performance necessary for this application.

Management Summary

Motivation

There are currently no solutions that allow the transmission and reception of arbitrary data over DAB or DAB+. Therefore Opendigitalradio asked us to develop a viable way to do this in the scope of this thesis. The goals of the desired open-source solution were:

- The transmission and reception of arbitrary data over DAB must use existing protocols and mechanisms.
- Being a broadcast medium each receiver must be able to selectively receive data that is relevant for itself.
- Its interface must be easy to use for third-party applications.
- Its interface must be stable and compact.
- The software must be extendable, for example to support encryption.

Approach

A large part of the work done during this thesis consisted of studying the voluminous DAB standard and analyzing existing applications in order to decide on the best way to achieve our goals. We then designed and implemented a solution that is highly modular and easily extendable.

Results

The result of this thesis is a detailed analysis on how the DAB protocol could be used to transmit and receive arbitrary data, as well as an open-source software solution that third-party applications can easily use to take advantage of this mode of communication. The software solution consists of endpoint daemons and several libraries that are programmed in the C++ programming language (C++). Composing these components in different ways allows for versatile and efficient handling of DAB packets. The delivery also includes a pre-built environment that couples the components in a ready-to-use setup. All that has to be done by the third-party application is to configure the Internet Protocol (IP) addresses and the DAB parameters. The usability of the software solution has been demonstrated using a demo application.

Contents

Part I

Introduction

Motivation

With this chapter we legitimate this work, explain why this work is valuable and how the products of this labour can be brought to use for good.

1.1 Present situation

1.1.1 Digital Audio Broadcasting

Digital Audio Broadcasting (DAB) is a radio transmission technology that allows for digital radio broadcasts. Although the name Digital Audio Broadcasting includes "Audio" there is no limitation to audio. There is more, much more that can be achieved using DAB. Beginning with a slide show of CD Cover images displayed on the radio to lyrics and news-feeds through to weather forecasts. But even that is not even close to the limits. Further details about DAB are located in chapter 3.1 on page 14.

1.1.2 Problem

There are currently no open source solutions that allow the transmission and reception of arbitrary data over DAB or Digital Audio Broadcasting Plus (DAB+) broadcasts. Most solutions used to transmit in professional applications are based on professional Software Defined Radios (SDRs) like the Universal Software Radio Peripheral (USRP) B200, the SDR mostly used by Opendigitalradio (ODR). SDRs are highly optimised hardware components designed to run an industrial radio application.

1.2 Goals

In this thesis we will develop and specify a protocol that allows for unidirectional transmission of arbitrary data over DAB. Subsequently we will design and implement applications to demonstrate the transmission and reception of data using the specified protocol.

A convenient solution consists of two endpoints. These endpoints resemble an Internet Protocol (IP) tunnel over DAB. When an IP datagram is sent to the sender-side endpoint daemon, it will pop out on the receiver-side endpoint. Figure 1.1 gives a first impression about how the finished product works.

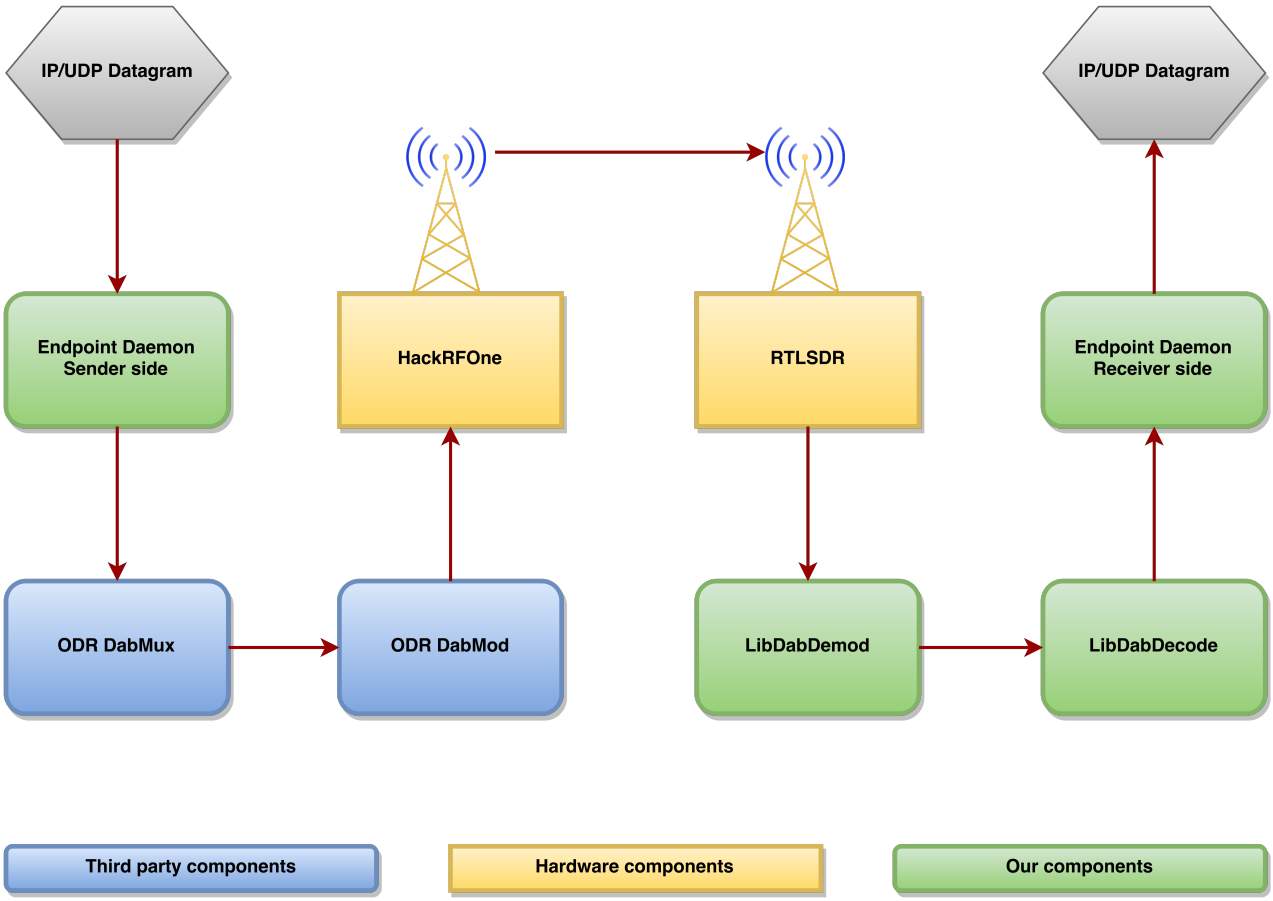


Figure 1.1: Data-flow outline

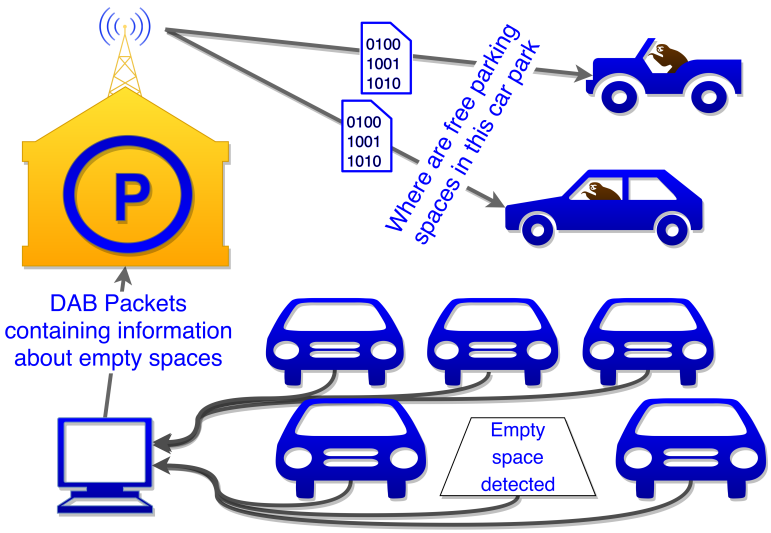
1.3 Possible applications

In the course of this semester, we thought a lot about possible applications of our work. Among the practical ideas are car park guidance systems, systems to distribute current market prices of products in less developed regions, digital bus-stop signs in remote locations or even firmware updates for cars.

1.3.1 Car park guidance system

Most of us were once in this situation, we take tours through the car park and we can simply not find an empty space.

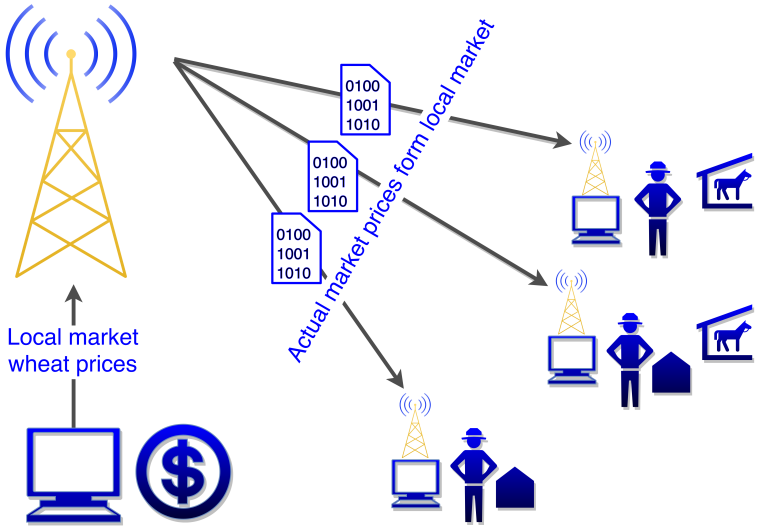
What if the car park had sensors in every lot? These sensors would be connected to a server in the car park. The server uses a low-power local DAB transmitter to send a map of all the available lots to all the cars in the building. When the parking spaces are numbered, it is even possible to just transmit the numbers of the free ones.



1.3.2 Distribution of actual market prices

Another application is a system to broadcast the current market prices for grain and the like in regions, with poor cellular network coverage.

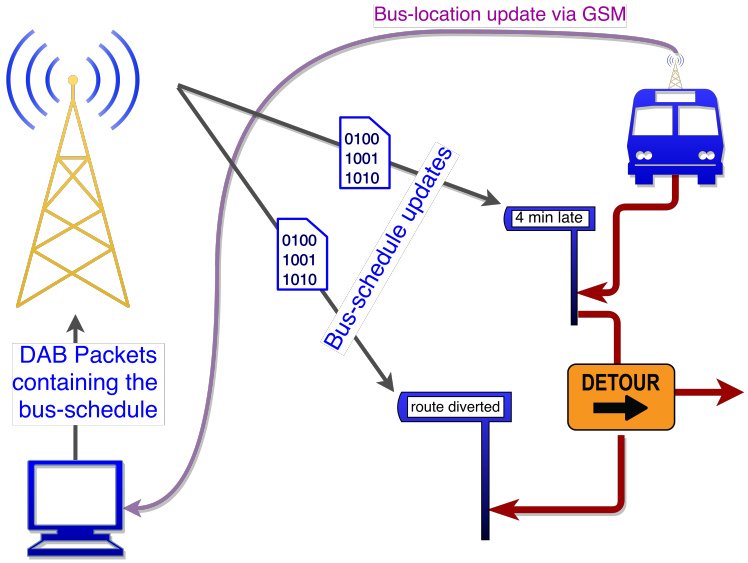
In many regions in Africa this could help farmers to optimise their profit. All they need to receive the newest market prices is a cheap computer, a DAB-USB-Stick and a monitor. This could help the economy in poor regions flourish.



1.3.3 Digital bus-stop sings

Bus lines in the Grison mountains can often reach into regions without mobile network. So if one would like to place digital bus-stop signs along these lines, the question remains how to update these signs.

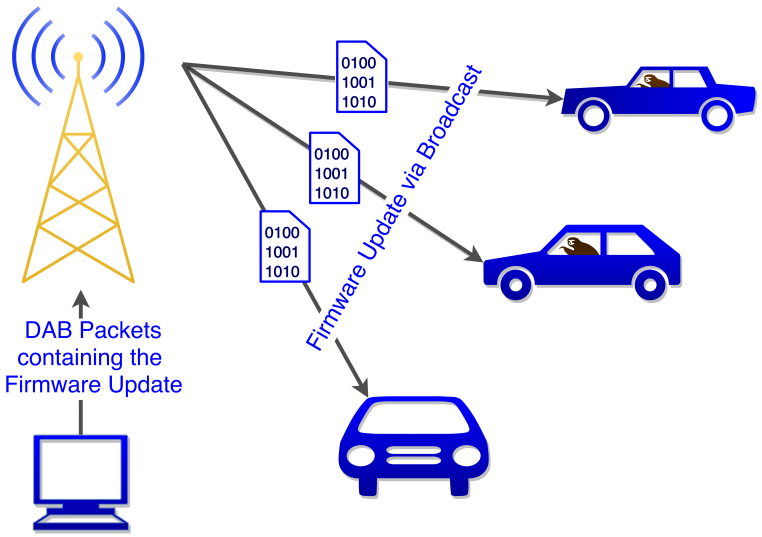
One simple and economic solution would be to equip the signs with small computers and DAB-receivers to receive schedules and updates via DAB.



1.3.4 Firmware updates

Today, many modern cars are able to update their own firmware over Wireless LAN (WLAN). The question is, does the car really need to be in the reach of a WLAN? With DAB the firmware update could be distributed to many cars at the same time, without the need for them to be in reach of a WLAN.

A DAB broadcast carrying the update could be sent many times in a week, so every car would be in reach at least once to receive the update.



Goals and tasks

In this chapter we write about our goals, the problems we had to face and how we finally satisfied the mentioned goals.

2.1 Goals

The goals we initially agreed on to achieve with Prof. Dr. Farhad Mehta and Matthias Brändli are documented in the original task-description in appendix F on page 71. The goals mentioned in this chapter are the revised goals. Details about why we had to do these changes are mentioned in the next section.

2.1.1 Difficulty

While studying the DAB standard we realized that the effort to specify a protocol in DAB would be enormous. The detailed process for standardising a new protocol is explained in section 4.2.1 on page 21. In short, the efforts and spendings to accomplish this would exceed the scope of a bachelor thesis by far. That was, when we realised that if we wanted to complete our goals, we had to find another way, a way that uses already existing mechanisms.

2.1.2 Revised goals

Caused by the mentioned difficulties we had to change the goal from "Writing a protocol specification" to "Researching a method to achieve transmission of arbitrary data". The original wording of the task description can still be found in appendix F.

The revised goals are the following:

Goal 1: It shall present a stable and compact core-specification.

Goal 2: It shall provide means to be extended for application specific usages like cryptography and do this without breaking the core-specification.

Goal 3: It shall offer the receiving party a way to decide whether a message is intended for them or not.

Goal 4: It shall provide a generic interface to third party applications, enabling them to transmit data via a stable interface.

Goal 5: It shall be released under an open source licence.

Goal 6: All demonstration software should be written in C++ [1].

Goal 7: Rather than developing an exact protocol specification as defined in the original task-description (Appendix F), all efforts should be focused on researching ways to transmit arbitrary data using existing mechanisms.

2.1.3 Purpose of this thesis

What remains the same is that the search for a practical solution to transmit arbitrary data over DAB is the main accomplishment in this thesis. Thereby the core achievement is the research, studying and understanding the standards and documentation of this process and its decisions.

Furthermore we invested a lot of work in the tool set which by now is a lot more sophisticated than what one would possibly expect from ”demo-applications”. So we consider these tools too as being part of our core achievements.

2.2 Satisfaction

Satisfaction 1: The stable and compact core-specification requested in Goal 1: is given by the fact that the mechanisms used to transmit data are already defined in the DAB standard.

Satisfaction 2: Every protocol which can be transported over User Datagram Protocol (UDP) can be used. So extensions like cryptography are easily implementable.

Satisfaction 3: We use IP, so every addressing scheme provided by IP is usable. This includes simple IP addresses but also subnets or even IP version 6 (IPv6) multicast groups. The receiver-side daemon only drops those IP packets in the Kernel, whose destination IP address matches his configured address.

Satisfaction 4: Most applications are able to communicate over ethernet. This enables most of the applications that are available today to use our system to communicate with each other.

Satisfaction 5: We release all the productive code under the BSD 3-Clause License.

Satisfaction 6: We wrote all the productive software in C++. For the development of prototypes we used python3, as of the development cycles are a lot faster.

Satisfaction 7: Rather than developing an exact protocol specification as defined in the original task-description, we focused our efforts on researching ways to transmit arbitrary data over existing mechanisms.

[1] Although this was not mentioned in the task description, ODR proposed to use the C++ programming language (C++) in the project. All of the currently existing solutions from ODR are written in C++, so all code produced in this thesis will blend in smoothly. We decided to integrate it into our goals.

Part II

State of the Art & Design

State of the art

In this chapter, we provide an overview of the currently available standards, and on how they apply to our work. We discuss the high-level concepts of DAB and its extension standard DAB+. Additionally we give an introduction into IP, UDP and TCP.

3.1 Digital Audio Broadcasting

DAB and DAB+ are both digital radio technologies that allow for the transmission of a diverse set of digital information. In addition to audio services, DAB was designed with different kinds of data services at its core. Some of these data services are designed to accompany the transmitted audio services while others carry data that is not associated to the transmitted audio programme.

DAB+ is an extension of DAB that enables a broadcaster to provide more audio programmes in a single ensemble. This is made possible by using a different audio codec known as MPEG-4 Advanced Audio Codec (MPEG-4 AAC). Through modern approaches, this codec produces an equivalent audio quality as the older MPEG-2 codec used in DAB while requiring a much lower data rate. Field tests in the United Kingdom and Australia also revealed that the new codec produces a slightly larger area of coverage than the older one. The use of Reed-Solomon-Coding provides better protection against reception errors (17, sec 2.2.3.2 and sec. 3.4.2.3).

3.1.1 Ensembles

Contrary to classical Amplitude Modulation (AM) or Frequency Modulation (FM) radio, DAB combines multiple "radio stations" into groups called ensembles. At the highest level, ensembles are composed of different services. Each service might represent a classical radio station or transport different types of data. An overview of the basic structure of an ensemble and its three main abstractions is shown in figure 3.1[1].

As mentioned above, services in a DAB ensemble can represent audio stations as well as data stations. The actual payload of a service is carried in service components. One major advantage of DAB is that the broadcaster can reuse existing service components for multiple services. This allows for structures like the one shown in the figure below. One application of this technique might be that the broadcaster transmits an hourly news segment on all audio services, without the need for retransmitting the actual audio data. Doing this allows the broadcaster to conserve bandwidth and thus provide more services in a single ensemble.

[1] In reality there would be one additional layer called the sub-channels. We left this layer out intentionally because it is not required to understand the concept of a DAB ensemble. We will introduce the concept of sub-channels when it actually becomes meaningful.

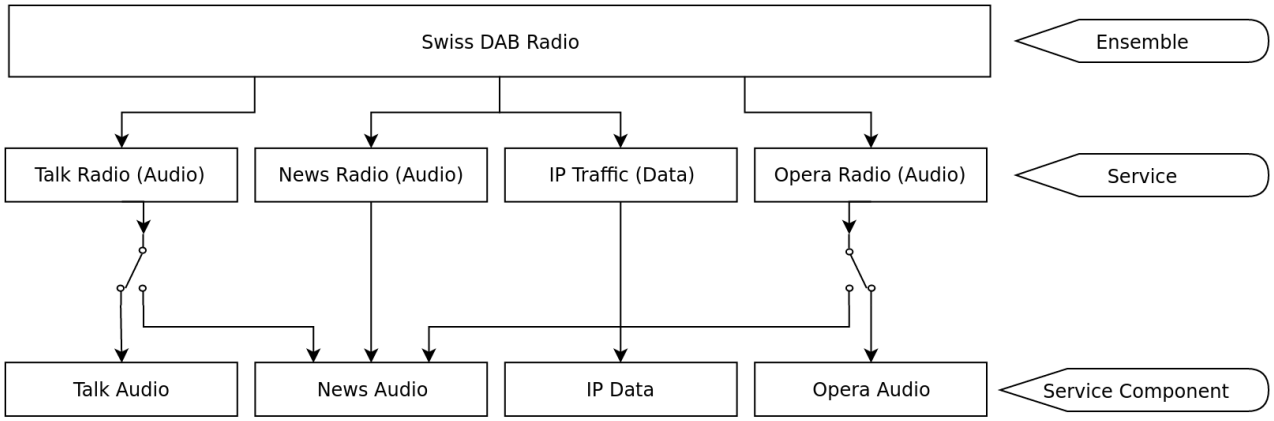


Figure 3.1: Structure of a DAB ensemble (adapted from Hoeg and Lauterbach (17, sec. 2.3.6)).

Each service has at least one component. This guarantees that there is always one primary component in each service which in turn defines the type of the service. In addition, each service can have multiple other components transporting either data or audio payloads regardless of the service type. This enables receivers that are capable of processing multiple service components to make use of advanced features of DAB like displaying album covers, weather forecasts, or traffic information.

3.1.2 Transmission Frames

A DAB transmission is split up into a sequence of so-called transmission frames. Each transmission frame carries a part of the transmitted data and has a basic structure that is common across all DAB transmission modes. This general structure can be seen in figure 3.2.

As the name implies, the Synchronisation Channel (SC) is mostly used to synchronize the receiver to the transmitter and to mark the start of a frame. This is done by sending a null symbol as well as a phase reference symbol. Despite being called the null symbol, the first symbol of a frame might convey the Transmitter Identification Information signal (TII). This signal can be used to uniquely identify each transmitter in a DAB network.

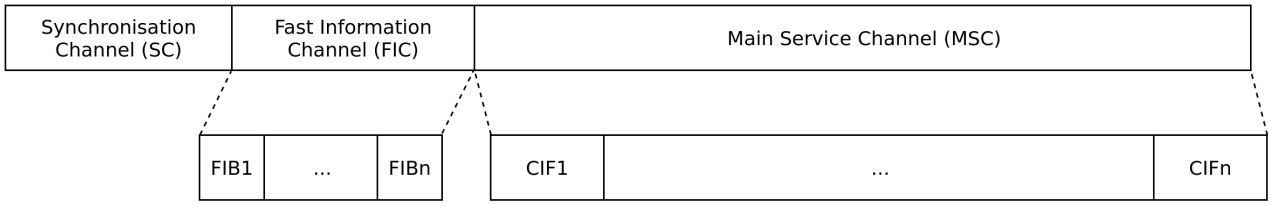


Figure 3.2: Structure of the DAB transmission frame (8, sec. 5.1).

As depicted above, the second part of the DAB transmission frame is known as the Fast Information Channel (FIC). This channel contains all the information required to describe the structure of the DAB ensemble. Examples of the information transported in this channel are the name of the ensemble, the services carried within the ensemble, and their associated labels. In contrast to the Main Service Channel (MSC), the FIC uses a fixed coding scheme (as defined in (8, sec. 11.2.1)) and does not employ time interleaving (8, sec. 12). The FIC is split up into multiple Fast Information Blocks (FIBs), with the actual number of FIBs depending on the actual transmission mode. An FIB is a 32 byte block of data carrying the actual payload of the FIC.

Most of the transmission frame is allotted to the MSC. This channel carries the actual data of the

services provided by the ensemble. Similarly to the FIC, the MSC is divided up into several Common Interleaved Frames (CIFs). A CIF carries the payload of the MSC and each service is assigned to a segment of each CIF. Each service transported in the MSC is time interleaved, hence the name Common Interleaved Frame. This interleaving is applied to evenly distribute bit errors introduced by fading and other disturbances in the transmission channel (17, sec. 2.2.4). As a result of this interleaving, there exists a decoding delay since the receiver must first accumulate the relevant sections of 16 CIFs to be able to reverse the time interleaving.

3.1.3 Transmission Modes

DAB is designed to be transmitted on different media and frequencies. One consequence of this design is that there exist four different transmission modes in DAB. Each transmission mode specifies its own coding and signalling parameters. For a more detailed explanation of the reasoning behind the choice of the individual parameters as well as an overview table of these parameters we refer to Hoeg and Lauterbach (17, sec. 2.2.1). Most of our work has been done in transmission mode one, but we designed our implementation to be as agnostic as possible to the choice of mode.

3.1.4 Programme Associated Data

Programme Associated Data (PAD) allows the broadcaster to transmit data that contains information that supplements an audio programme. The PAD is divided into two different kinds, the Fixed PAD (F-PAD) and the Extended Programme Associated Data (X-PAD).

The F-PAD is present at the end of every audio frame and has a length of two bytes. This type of PAD is designed to be used for control signals like Dynamic Range Control (DRC) or for indicating whether an audio programme carries music or speech (adapted from ETSI (8, chap. A.4)). Due to the fixed size of the F-PAD, the bit rate is directly dependent on the sampling frequency of the transported audio. This limits the maximum bandwidth to 0.667 kbit/s (8, chap. 7.4).

In contrast to the F-PAD, the X-PAD may or may not be present. Additionally, the X-PAD can be either a short X-PAD, which is always four bytes in length (8, chap. 7.4), or a variable size X-PAD, with the length varying between 4 and 48 bytes (8, chap. 7.4.4.2). This allows for a much larger bandwidth and thus the transportation of more data than the F-PAD. Examples for applications that make use of the greater bandwidth are dynamic labels and Multimedia Object Transfer (MOT).

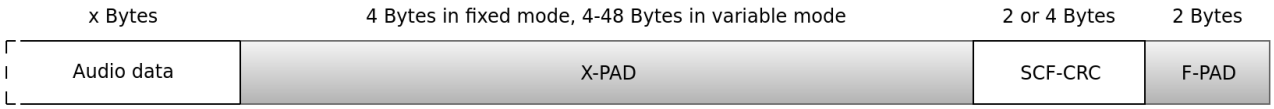


Figure 3.3: Location of the F-PAD and the X-PAD in a DAB audio frame (adapted from ETSI (8, chap. 7.4))

3.1.5 Non Programme Associated Data

DAB also allows the transmission of so-called Non Programme Associated Data (NPAD). As the name suggests, NPAD does not necessarily have a relation to any transmitted audio programme. Like PAD, NPAD services can also be used to transfer MOT applications. Non Programme Associated Data services are also useful to transport applications like Journaline® or enhanced traffic information.

One of the most distinguishing features of NPAD in contrast to PAD is that the transmission of data in the form of NPAD does not require an associated audio service. As a result, the bandwidth available for NPAD is much higher.

3.2 Internet Protocol

IP builds the basis of current computer networks as it allows addressing of data packets to specific recipients. The basic principle of both of the currently used versions, version 4 and version 6, is the same. An IP packet provides information about the sender and the receiver as well as the possibility to encapsulate further data. This basic idea of addressing packets to specific recipients stands in stark contrast to broadcast systems like DAB.

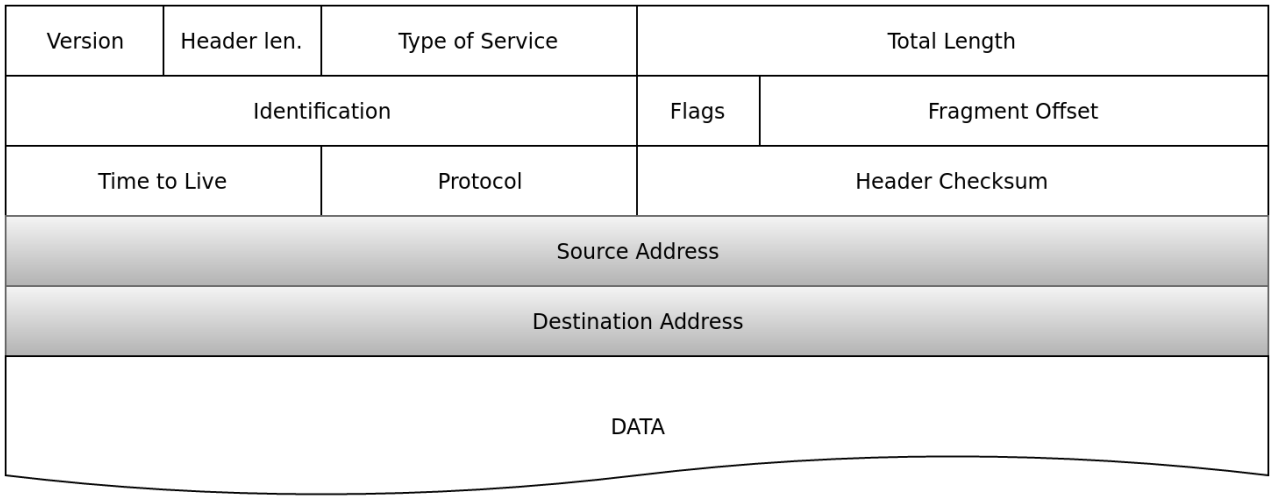


Figure 3.4: Structure of an IP version 4 (IPv4) datagram (20, sec. 3.1)

While addressing of single recipients, also known as unicast, is the most widely known mode of operation for IP there also exists methods to address groups of recipients.

The first of these addressing schemes is called IP broadcasting. This method is only supported by IPv4 and is not even part of the original specification as found in (20). With the advent of larger and more dynamic networks, it was recognised that broadcasts might be beneficial for automatic discovery of hosts in a network or to supply information to a whole network all at once (26, sec. 3). Broadcasting is achieved by using special addresses as the destination for an IP datagram.

The second addressing scheme is known as multicasting. This method is supported in version 4 as well as version 6 of IP and allows the sender to address specific groups within a network. While multicasting was also a later addition to IPv4 (2), it is an integral part of IPv6 as defined in (16). Similar to broadcast addressing, multicast addressing uses special IP addresses to select a specific group of hosts on a network as recipients.

The field *DATA* as seen in figure 3.4 can carry arbitrary data in the form of bytes. While it would be possible to transport the payload directly in this field, in most cases an additional protcol like UDP or TCP is used to encapsulate and address the data further.

3.3 User Datagram Protocol

The User Datagram Protocol (UDP) is a network protocol that adds another layer of addressing to a network transmission. This layer is comprised of so-called ports. Ports allow to further specify the source and destination of a network datagram by providing an additional $2^{16} - 2$ endpoint addresses in the range $[1, 65535]$. In the case of source ports, the value might also be 0 (29).

An interesting property of UDP, when compared to other transport protcols like the Transmission Control Protocol (TCP), is that it is unreliable. In the context of network protocols, unrealiable means that there are no guarantees for delivery or duplicate protection. This allows for the usage of UDP without the existence of a return channel, since no communication between both endpoints is required to ensure the state of the transmission. This property makes for a perfect fit of UDP in broadcast systems like DAB.

3.4 Transmission Control Protocol

Similarly to UDP, the Transmission Control Protocol (TCP) allows to further specify the source and destination of a data transmission. It uses the same concept of ports to provide an additional layer of addressing over the basic addresses present in IP.

However, as mentioned above, Transmission Control Protocol (TCP) is a reliable protocol. This means that TCP, or rather a network stack implementing TCP, provides the means necessary to ensure and verify the delivery of the datagram. In order to achieve this, TCP makes use of the fact that hosts in a regular computer network possess a bidirectional communication channel between each other. The reliability of TCP is owed to the fact that the protocol makes use of sequence numbers and acknowledgements (21, sec. 2.6). This requires the sender and recipient to be able to both communicate with each other, since the recipient has to acknowledge the sequence number of the received packets to the sender. This property makes TCP virtually unusable in broadcast systems like DAB since there is no return path from the receiving device to the broadcast station.

Methods and research

In this chapter, we provide an overview of the currently available standards and on how they apply to our work. We discuss the high-level concepts of DAB and its extension standard DAB+. Additionally we give an introduction into IP, UDP and TCP.

4.1 Research

Soon after we started the thesis, we realised that we know almost nothing about DAB and we therefore will have to gather a lot of knowledge before we can start. We used different sources to learn what we needed to know.

Most of our knowledge and insights into DAB we produced through studying the DAB standard, its technical specifications and pertinent literature. The standard itself, consists of more than twenty different documents that supply a tremendous width and depth of knowledge. The book Digital Audio Boradcasting helped us to deepen our understanding enough, to understand the DAB standard.

4.1.1 ETSI DAB-Standard

The European Telecommunications Standards Institute (ETSI) DAB standard is a wide collection of ETSI Standards (ESs), European Standards (ENs), ETSI Technical Specifications (TSs) and ETSI Technical Reports (TRs). In total, there are more than fifty unique documents in the list of ETSI standards for DAB. We had to figure out which of these documents will be relevant to our work.

We mostly used following documents:

- EN 300401 Digital Audio Broadcasting to mobile, portable and fixed receivers. (8)
- ES 201735 Internet Protocol datagram tunnelling. (3)
- TS 101756 Registered Tables. (15)
- TS 101735 Internet Protocol datagram tunnelling. (4)
- TR 101495 Guide to DAB standards; Guidelines and Bibliography. (12)

Other documents we used to research the best way for transmitting arbitrary data are:

- EN 301234 Digital Audio Broadcasting Multimedia Object Transfer protocol. (9)
- TS 103177 Filecasting; User application specification. (14)
- TS 102979 Journaline; User application specification. (11)
- TS 102978 IPDC Services; Transport specification. (10)

- TS 101499 MOT SlideShow; User Application Specification. (13)
- TR 101497 Rules of Operation for the Multimedia Object Transfer Protocol. (5)
- TS 101759 Data Broadcasting - Transparent Data Channel (TDC). (7)

Difficulties

What complicated our process was the fact that the standards are hard to dig through. Some of the documents reference to chapters like "see chapter 6.2.1.8". But when we wanted to take a look, all we could see was: "This chapter was moved to a separate document". Then we dug through the standard guide (12) to find the document that the chapter was moved to, just to realise that the internal structure of the document was totally different and chapter 6.2.1.8 leads nowhere. We then searched an older version of the document, just to get an idea about what we should search in the document.

Another example of the complications that arose from the fragmentation of the standard documents is the definition of Fast Information Groups (FIGs). Even though most of FIGs were documented in EN 300401 (8), some of those specifications explicitly mentioned bit combinations that are reserved for future use. During the testing of our DAB decoding library exactly those combinations came up in certain publicly broadcasted services. This led us to believe that our implementation was flawed but we were unable to find the error. Only after digging through additional standard documents, we figured out that these combinations have in fact already be defined for use in MPEG-4 AAC audio programmes. It would have been very helpful if EN 300401 (8) – as the defacto central document – would have mentioned the use associated with these bit combinations.

Also most of the illustrations in the documents are very old (e.g. from 1995 (8)) and accordingly pixelated and blurry. That means we had to redraw all the graphics and structure diagrams. On the other hand this saved us the effort of requesting permission to copy the graphics from the document.

4.1.2 Digital Audio Broadcasting Book

At the kick-off meeting Matthias Brändli lent us the book Digital Audio Broadcasting (17). The book helped us a lot getting into DAB. We could get an idea about the different mechanisms used to transport data in DAB. Soon we discovered that this book is also available at Google-Books (18) which was very helpful when we needed to search for a specific term.

4.1.3 ODR tool-set

Opendigitalradio maintains a tool-set to multiplex different services into a DAB ensemble, and to create a sample stream for use with SDR devices. We used these tools to create a custom ensemble to test our software during development. The tool-set is released under the terms of the GNU General Public License version 3 (GPLv3).

4.1.4 Software receivers

During the course of our project, we analysed different existing software solutions for receiving DAB. A major help were SDR-J-DAB and dab-rpi. These two applications are written and maintained by Jan van Katwijk and are released under the term of the GNU General Public License version 2 (GPLv2). While the former is a DAB reception application for desktop and mobile computers, the latter was specifically designed and tuned to work on the Raspberry Pi 2 Model B (RasPi2).

We used these applications as reference implementations to verify our own system's functionality. Additionally, we reused some parts of dab-rpi in our demodulation library.

Additional applications that were extremely helpful – especially in the beginning of our project – are GNU Radio and gr-dab. GNU Radio is an open source platform for SDR applications, while gr-dab is a set of extensions for GNU Radio. We used GNU Radio and the gr-dab modules to get an initial idea of how DAB transmissions work. The gr-dab modules provide building blocks to demodulate the DAB data stream and process the FIC. Having the ability to dump real-world dab data into a file helped us tremendously during the development of our DAB decoding library.

4.2 Methods to transmit data over DAB

As stated in the original task-description (appendix F) we started research to find out how a protocol is specified in DAB. We mentioned in section 2.1.1 on page 11 that the hurdles to specify an own protocol are too high. In this section we go into details about the process of specifying an own protocol.

4.2.1 Own protocol

While studying the DAB standard we realized that the effort to specify a protocol in DAB would be enormous. This efforts would include writing an ETSI compliant specification, covering all facets of the protocol. This specification will then be proposed to WorldDAB. If everything is compliant and does not interfere with other parts of the DAB standard, it will be accepted and proposed to the WorldDAB Steering Board. If the WorldDAB Steering Board accepted the proposal, it will be offered for standardisation to the international standards body (i.e. ETSI, or European Committee for Electrotechnical Standardization (CENELEC)). If the proposal is accepted by the European Broadcast Union (EBU)/ETSI/CENELEC Joint Technical Committee, it will be included in the next version of the DAB standard (40).

Obviously, the efforts and spendings to accomplish this would exceed the scope of a bachelor thesis by far. We had to reorient our efforts to developing a way to reach our goals by using existing mechanism of DAB.

4.2.2 Multimedia Object Transfer

The MOT standard was the first data-transfer service available in DAB. It was introduced to enable the same features that Radio Data Service (RDS) offers in FM radio in DAB. A "MOT slide show" application (13) can display the album cover to the currently playing song. Or it could display news headlines during the news show. In fact it can transmit any image that is not bigger than 50kB and available in PNG or JFIF (JPEG) (13).

An extension we looked at was the "filecasting" user application. As the name suggests, it is designed to transport files. We realised that the supported formats are limited (14, sec. 5.5) and are not sufficient for our use-cases.

MOT can also be used for other services than "slide show" or "filecasting". All the transportable formats are defined in (15, table 17). Due to the limitations in data formats, it can not be used for arbitrary data and we had continue our search.

4.2.3 Transparent Data Channel

Transparent Data Channel (TDC) was the next mechanism we analysed. Here the raw-data is packed in MSC data groups (MSCdgs) (for details see section 9.4 on page 54). The MSCdg is then split up and packed into DAB Packets (DABps) (for details see section 9.3 on page 51). The DABps are then passed to the SDR.

The overhead is rather small at approximately 4.3%. This small overhead comes with the cost of no forward error correction. Therefore the delivery of data is best effort. This means, it can not be assumed that every sent packet is delivered successfully. Therefore every implementation using TDC must be fault-tolerant to corrupted data, lost data and additional data.

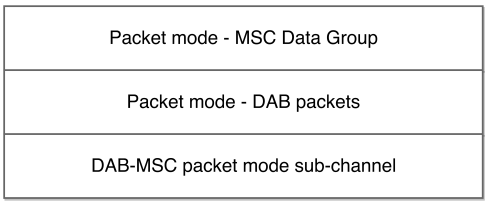


Figure 4.1: TDC stack (7, sec. 4.1.2).

This service is able to transmit arbitrary data which satisfies goal 7. Satisfying the goals 3 (receiver can decide if a message is meant for him) and 4 (the implementation offers a generic interface for third parties) will be a lot more difficult. But still, we kept TDC in our pool of viable solutions.

4.2.4 Internet Protocol

We discovered two mechanisms that transport IP datagrams. With a unidirectional protocol like UDP running on top, IP satisfies every demand we had to the transport method.

Is it extensible for application specific usages?

Yes it is. One can use every type of application one wishes. The transmitted data can be encrypted, it can be signed, everything that can be done on unidirectional media is possible. In addition, any other unidirectional protocol like FLUTE (27) can be used on top of the UDP layer.

Does it offer the receiving parties means to decide if the message was meant for them?

Yes it does. IP addresses are an ideal means of addressing. Using an IP address one can simply assign every receiver an address and send the packets to this address. In IPv6 it is even possible to use multicast groups for simple multicasts. Using IPv4 a similar functionality could be realised using the multicast addresses (224.0.0.0 to 239.255.255.255).

Is it offering a generic interface to third party applications?

Yes it is. Most modern applications are able to communicate over IP. Therefore we consider an IP interface as sufficient to say that we offer a generic interface to third party applications.

Can received data-packets easily be redistributed?

Yes they can. IP packets can be easily addressed. After reception the IP datagrams will be handed to the Kernel. From there the IP datagrams can be handled like any other IP datagram. They can be rerouted, they can be filtered using a firewall or they can be consumed by the system.

4.2.5 IP Data Casting

One of the two ways is IP Data Casting (IPDC).

IPDC is the same concept as in Digital Video Broadcasting - Handheld (DVB-H). This is, because IPDC was designed with the goal of a uniform IP data casting system over DAB, Multimedia Broadcast Multicast Service and DVB-H in mind.

IPDC is based on a DAB Enhanced Stream Mode. Enhanced means there are some additional mechanisms that allow for error correction. The detailed difference is visible in figure 4.2.

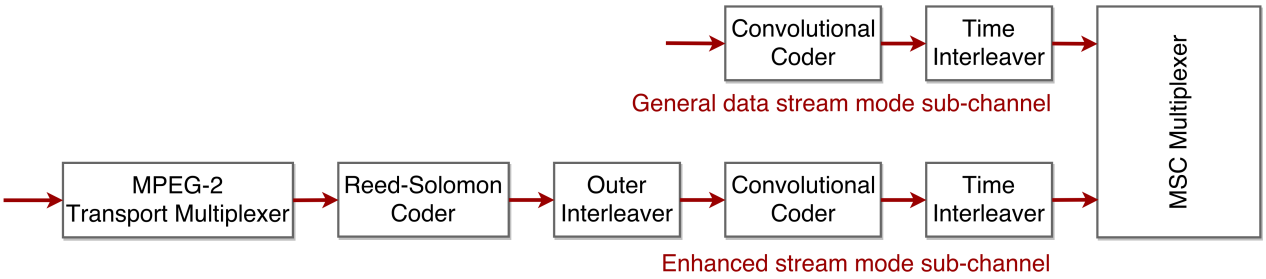


Figure 4.2: Difference of stream modes (adapted from Hoeg and Lauterbach (17, Fig. 2.15))

The protocol stack in IPDC is more complicated than the ones of TDC and IP Datagram Tunneling (IPDT) (see figure 4.3). Instead of a packet mode sub-channel this service is based on an enhanced stream mode sub-channel with an MPEG-2 transport stream on top.

The next layer is provided by Multiprotocol Encapsulation (MPE) and the programme information (Programme Specific Information (PSI) and Service Information (SI)).

MPE is a data link layer protocol initially designed for Digital Video Broadcasting (DVB) (6). It provides means to transport any packet oriented protocol over an MPEG-2 transport stream. In this case IP with UDP is used.

To stay compatible with the protocol stacks used in the IPDC implementations of DVB-H and Multimedia Broadcast Multicast Service, there are one or two additional layers on top.

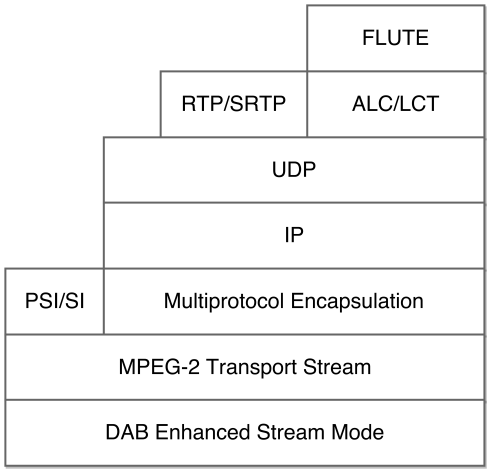


Figure 4.3: IPDC stack (reproduced from ETSI (10, chap. 4.2))

Either the next layer is supplied by RTP/Secure RTP (SRTP) or by ALC/LCT with FLUTE on top. RTP and SRTP are mostly used for delivery of audio or video over IP networks. In our eyes, using a packet mode protocol on top of an MPEG-2 stream to transmit an audio or video stream is rather questionable.

FLUTE is used for file delivery on unidirectional media. According to RFC 3926 it is particularly suited for multicast networks. This is the reason why there is an ALC layer beneath. ALC is a protocol that supports massive multicasts with congestion control.

4.2.6 IP Datagram Tunneling

The second of the two ways to make use of IP is IPDT.

The stack of IPDT is similar to the stack of TDC. An IPDT service also uses a DAB-MSC packet-mode sub-channel with a network layer and a transport layer in packet-mode.

On top of said layers are three additional layers. There is an IP layer, a layer with an unidirectional protocol like UDP and then a layer with an application protocol. We like the idea of an application able to support any protocol that a user could wish for.

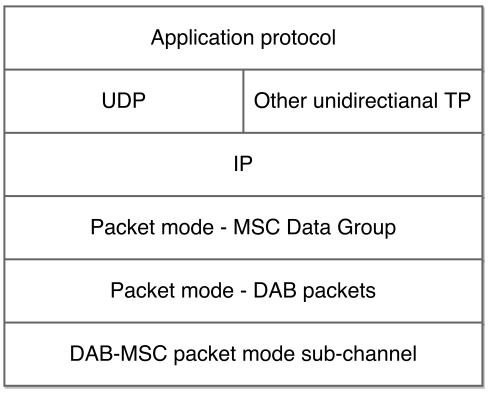


Figure 4.4: IPDT stack (reproduced from ETSI (4, chap. 4))

Overhead

We asked ourselves, how much overhead this service would carry. The minimal overhead can only be achieved by using IP datagrams with a Maximal Transmission Unit (MTU) of size 8191 bytes. These 8191 bytes are owed to the maximal data field size of the surrounding MSCdg. Assuming that we use UDP/IPv4 there are 20 bytes IP header and 8 bytes UDP header, this leaves 8163 bytes of payload per MSCdg. Including the MSCdg header and the Cyclic Redundancy Check (CRC), 8163 bytes of payload require a total of 8195 bytes. The 8195 bytes must be packed in the data-fields of 90 DABps of size 96 bytes and one of size 24 bytes. The total size of the DABp will be 90 packets $\times$ 96 bytes + 1 packet $\times$ 24 bytes = 8664 bytes.

This ratio of 8664 transmitted bytes to 8163 bytes payload leads to an overhead of approximately 6%. We deem this to be acceptable.

Reliability

Like a TDC service, IPDT does not guarantee for delivery and is only best effort. Therefore every implementation must be fault-tolerant to corrupted data, lost data and additional data.

Only the receiving party can decide, if a data-group was transmitted completely and without corruption. This could be done by transmitting a hash-checksum that can be compared to the checksum of the received data-group. The down side of this method is the possibility of false negatives, when the checksum gets corrupted during transmission.

A further measure provided by MSCdgs is the repetition of the whole data group as described in TS 101735 (4, sec. 5.3.2).

Further details

More detailed information about how IPDT is used and our implementation can be found in chapter LibDabIp on page 49.

4.3 Decisions

We decided to use UDP/IP because it brings a lot of advantages, simplifies our work and provides a stable and well established foundation. What remained was the decision which approach to pursue, IPDC or IPDT.

4.3.1 IPDC

TS102978 says IPDC is optimised for devices limited in computational resources and battery and that it is primarily used for transportation of multiple audiovisual services based on H.264/AVC video coding.

We could not figure out where this optimisation for limited devices comes from. The parts that are rather time consuming, like demodulation and decoding, still had to be done. In addition there is more overhead caused by the mandatory protocols on top of UDP.

IPDC is more complicated to implement than IPDT. This is due to the error correction mechanisms in enhanced stream mode (see figure 4.2). Furthermore, there are questions about MPEG-2. Are some of the parts of MPEG-2 used in these transport streams not yet free to use? Would we have to pay royalties? With time getting short we could not expand our studies to get clear with MPEG-2 and its patents.

4.3.2 IPDT

We soon understood the protocol stack of IPDT and realised its simplicity. In contrast to IPDC we spotted good chances being able to implement a solution with IPDT in the course of this semester.

We gave IPDT preference to IPDC because of IPDCs flaws and difficulties rather than for IPDT having great advantages. The only drawback we saw in this choice is that IPDC does support forward error correction. Still, the libraries designed in this thesis are modular enough for being brought to use in an IPDC setup, once the necessary parts to handle the MPEG-2 stream are implemented.

Results

This chapter gives an overview of the provided "out of the box solution", and our common library.

5.1 LibDabCommon

While we designed the libraries we realised that we often used identical constants, literals, and types. At this point we decided to extract similarities into the LibDabCommon. We extracted both, parts of the public Application Programming Interface (API) like `byte_vector_t`, `sample_queue_t`, or `parse_status` and internal parts.

To compile our libraries, LibDabCommon must be provided. To simplify the compilation process, we decided to integrate LibDabCommon as a git submodule in the other libraries.

5.2 Ready-to-use solution

We designed a simple solution to establish an unidirectional IP tunnel that can be set up and managed without in-depth knowledge of DAB, and the used transport mechanisms. Our solution consists of two daemons, one for the sender-side and one for the receiver-side. Our daemons are implemented using our own libraries which are connected using a thread-safe queue designed for high performance applications.

5.2.1 Endpoint daemons

The sender-side endpoint daemon is responsible for the translation of standard IP datagrams into DAB packets that can be transported in a service. To fulfil its purpose, the daemon provides a virtual network interface and listens on a user defined port. The user can then use any application capable of sending UDP packets to provide data for the daemon to encode. To configure the sender-side daemon, the user must pass the source and target IP addresses, the UDP port, the desired DAB packet address and the name of the destination file as command line parameters. The produced packet file can be used as a data source for ODR-Dabmux.

The receiver-side endpoint daemon lives on the side of the transmission channels. Its main responsibility is to receive and process the DAB signal, and extract the transmitted data. It creates a virtual network interface and uses this to passe the received IP datagrams to the operating system's kernel for further processing. To consume the received data, the user can use any UDP enabled application. They must open a listening UDP socket on the desired port and address. To configure the receiver-side daemon, the must pass the desired IP and DAB packet addresses as command line parameters.

5.2.2 Queue

We required a way to connect our libraries to each other. Since they are designed to run in a multi-threaded environment, the channel between the libraries must also be thread safe. After researching different ways to create a data connection between two threads, we decided to make use of a thread-safe queue. To keep our dependencies to a minimum, we decided to make use of *readerwriterqueue* by Cameron Desrochers. This queue is designed to be lock free and released under the terms of the Simplified BSD License.

5.2.3 Detailed data flow

In this section we present a detailed schematic of the dataflow in our components. On the sender-side three applications must be run.

Sender-side

First the *input handler* must be run. The handler creates a virtual network interface and waits for data. As soon an IP datagram is available on the virtual network interface, it will be packed and written into the file specified as the input in the ODR-DabMux configuration. As the ODR-DabMux IPDT service configuration must specify an input source and the only possible input source supported for data services at the moment is file input, the input handler must be stopped when enough data was collected.

Now ODR-DabMux and ODR-DabMod must be configured and run. Each service must be configured for it to work correctly. For our data service, we select the file written by the input handler as input source. While ODR-DabMux multiplexes different services into a DAB stream, ODR-DabMod modulates the DAB stream to samples which can be transmitted using the SDR.

A detailed outline of the sender-side transmission can be seen in figure 5.1 on page 28.

Receiver-side

The receiver-side is mostly implemented by us. It consist of one application which makes use of all the libraries.

Each step in the process chain is done on its own thread and run at the same time. The steps resemble the path that the data takes through the process chain.

First the *Device Thread* uses the LibDabDevice to acquire samples from the provided DAB-Stick. These samples are then enqueued into the sample-queue.

Next the *Demodulation Thread* reads samples from the sample-queue and demodulates them into Orthogonal Frequency-Division Multiplexing (OFDM) symbols using LibDabDemod. The demodulated symbols are enqueued into the symbol-queue.

Now the *Decode Thread* reads these symbols from the symbol-queue and decodes them. The resulting DAB packets are feed to a callback that is provided by the next thread, the *User Thread*.

The *User Thread* does three things. It tells the *Decode Thread* which service should be decoded by the LibDabDecode. It provides a callback to the *Decode Thread*. In our application this callback uses the LibDabIp to parse MSC data groups from the supplied DAB packets and feeds the MSC data groups to the MSC data group parser to get the reassembled IP datagrams. The callback enqueues

the datagrams into the ASIO network IO in the *User Thread*. The network IO sends every enqueued datagram to the kernel using a virtual network interface.

In figure 5.2 on page 29 we provide a detailed overview on the threading, and the dataflow.

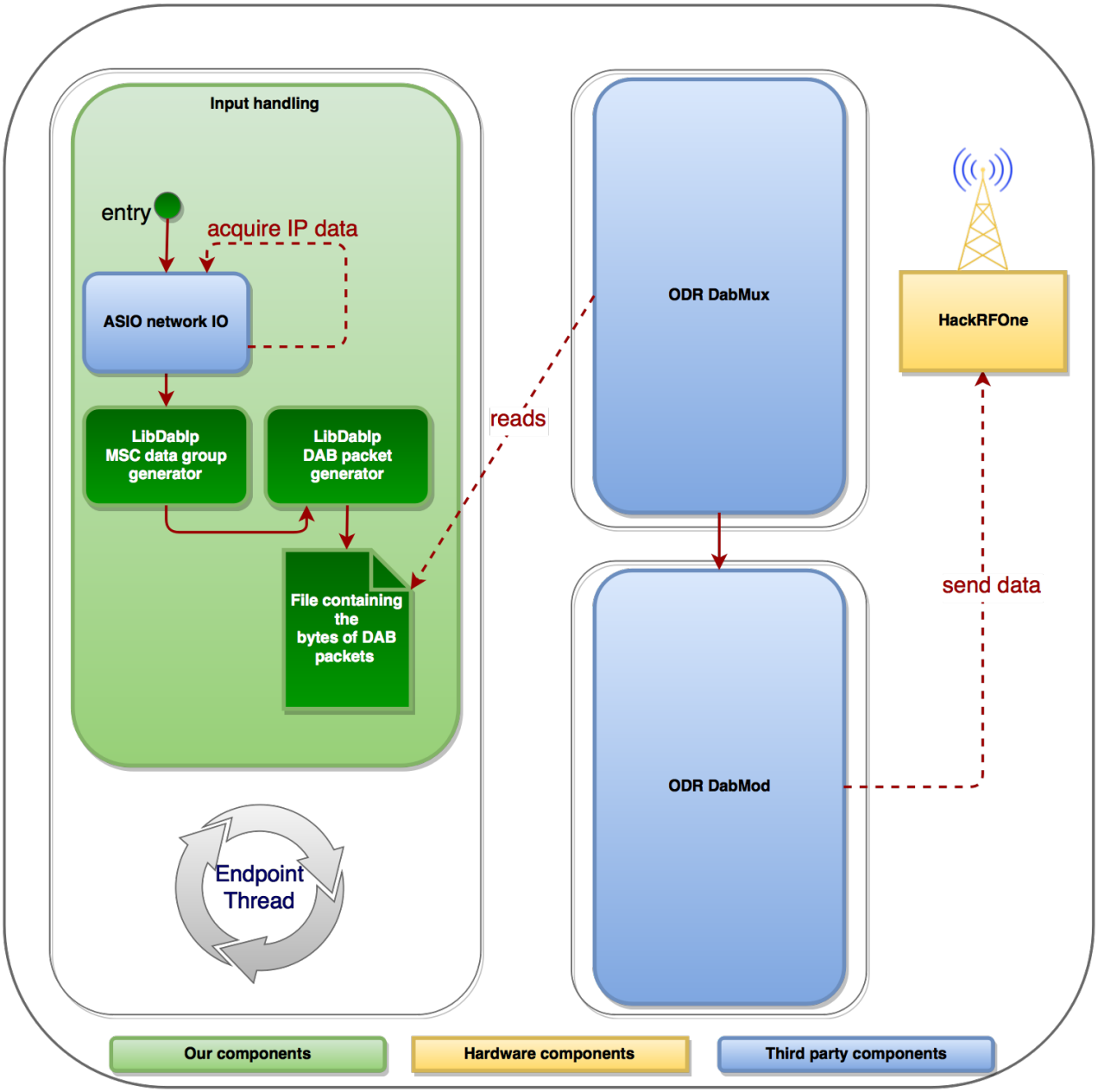


Figure 5.1: Detailed transmission flow on the sender-side

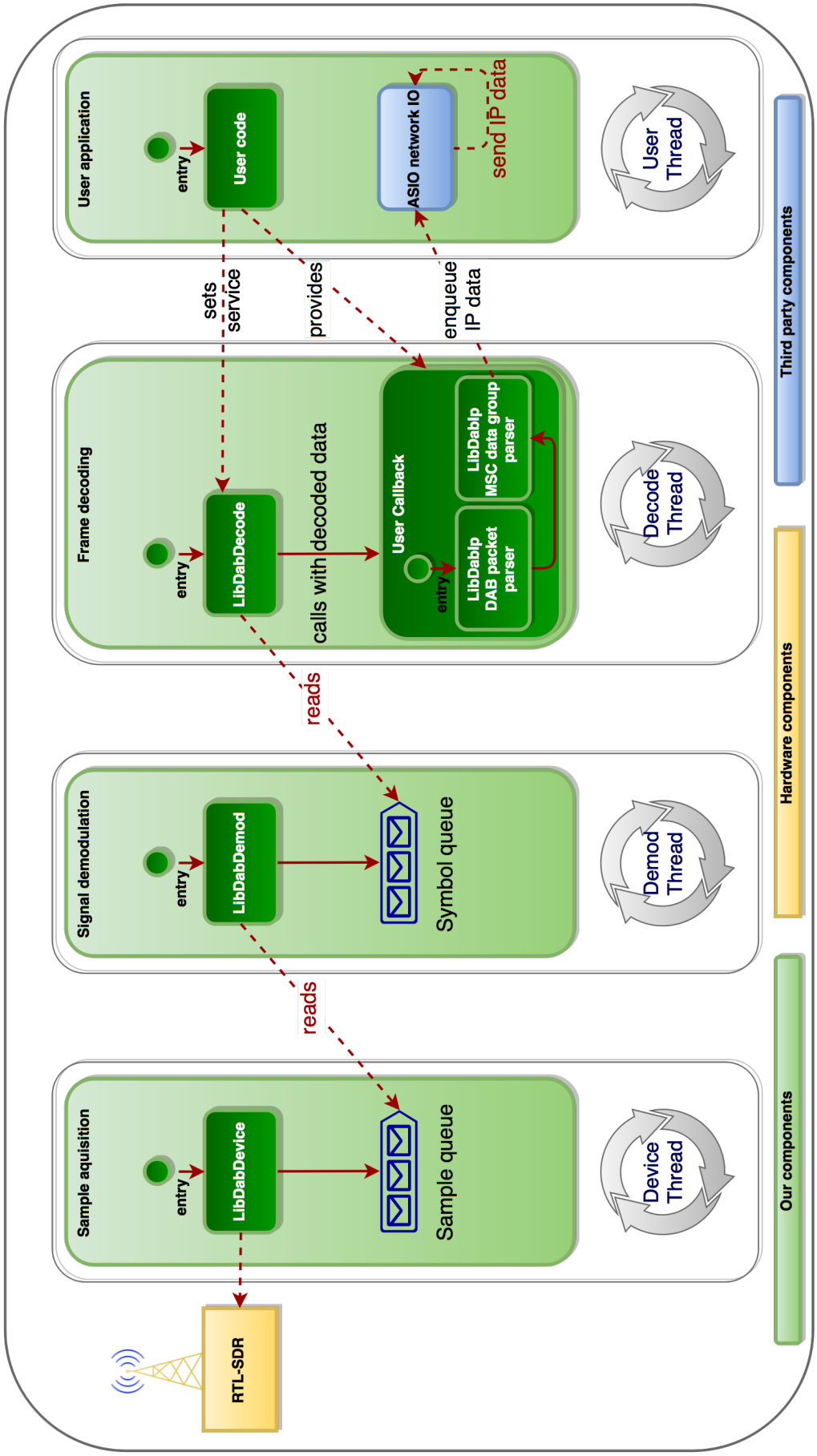


Figure 5.2: Detailed transmission flow on the receiver-side

Part III

Libraries

LibDabDecode

In this chapter, we provide an introduction into the API presented by our DAB decoding library. We also present the design decisions, and an architectural overview.

6.1 Introduction and Design

DAB is a very flexible technology for the transmission of audio and data services. This flexibility comes with the price tag of a high complexity in its specifications. Since the original release of the DAB core specification in 1995, the system has continually been extended to accommodate a wider variety of services. Each extension brought new features and also new challenges to the table. Our main goal with the development of LibDabDecode was to reflect this approach of extensibility while keeping a clean and easy to maintain design.

To simplify future work on the decoding process, we decided to implement it in a separate library. This fosters a clear separation of the decoding process from the demodulation process, thus allowing to work on both part separately.

We designed a very simple data input interface, so that developers are as free as possible when choosing an input channel for OFDM symbol input. This even allows developers to run other parts of their application on a different computer. One could imagine the demodulator running on a more powerful machine, while processing the DAB data on a lower end system.

In addition to a simple input interface, we designed a very simple data extraction interface. We decided on an architecture that exposes the bare minimum required to the user. Our main abstraction is the *ensemble* class. It allows the user to access ensemble information and services in an intuitive way. We achieved this by allowing the enumeration of the available services while hiding complex inner workings like the FIC and DAB sub-channels. Later in this chapter, we show how easy it is to get up and running using our API.

6.1.1 Goals

During the design phase of the library we found the following requirements that had to be tackled by an acceptable implementation:

- The implementation must not unnecessarily expose the complicated structure of a DAB ensemble.
- Due to our focus on DAB transmission mode one, the complete processing of the ensemble must not take longer than 96ms. This includes extraction of service data.
- Owed to the fact that DAB is still in motion and future extensions must be expected, the internal design must be modular and easily extensible.

- Considering that the implementation will be used on embedded systems, the implementations memory requirements shall be as low as possible. The implementation shall not leak memory.
- Decoding and processing of DAB data are computationally expensive operations. The implementation must be able to run in isolation, to enable the user to employ multi-threading to make optimal use of the systems resources.

6.1.2 Difficulties

The main difficulty we faced, was the high complexity of the DAB transmission. Even though it might seem fairly flat and simple on first glance, further studies of the standard revealed that a DAB ensemble can be highly dynamic. This leads to a large number of extensions, which in turn complicates the implementation process.

Another difficulty is related to processing power. DAB uses several mechanisms to enhance the resiliency of the transmission against errors. One of these mechanisms is convolutional coding. On a very high level, this coding scheme adds additional bits – so-called parity bits – to the bit-stream. This is done by processing an input plus some of the bits that came before it. The number of parity bits and their values depend on several parameters that have to be defined by the designer of the application using the convolutional code. For example, DAB generates four output bits from the input bit and the six bits that preceded it. Implementing the encoding process is straightforward and the algorithm runs in linear time with respect to the input. Decoding on the other hand is not as simple. As can easily be seen, there are multiple seven-bit combinations that can be mapped onto the same 4-bit combination. Since physical transmission channels are subject to noise, the stream of parity bits will contain errors. The goal of the decoder is to find a sequence of possible input bits which would be encoded into the received parity bits as closely as possible. Therefore, the decoder must track multiple possibilities during the decoding process. For an excellent discussion on convolutional codes and decoding algorithms we refer to two lectures note of the MIT(24)(25).

6.1.3 The ensemble abstraction

The main interface exposed to a user of the library is the `struct ensemble`. Our goal with this abstraction was to expose a useful interface to the structure and components of a DAB ensemble. Due to the high complexity and flexibility of DAB, we decided to leave out parts of the standard that do not apply to our project. Examples for omissions are audio services, enhanced streaming services and Programme Associated Data (PAD).

The ensemble manages the lifetime of service, service component, and sub-channel objects. The actual relationships of services to service components as well as service components to sub-channels however is not managed by the ensemble. We decided that this approach reflects the actual structure of a DAB ensemble as closely as possible. Considering the structure of a DAB ensemble – as seen in figure 3.2 on page 15 – the ensemble itself clearly owns the services. The ownership of the service components however is not as clear. Service components and services however are not in a one-to-one relationship. Each service can be associated with more than one service component while each service component can also be associated with multiple services. A service component might move between different services, which would require moving the ownership of the object. We came to the conclusion that it thus would be logical to move the ownership of the service component to the ensemble and let the services manage their relation to the service components. Since on an abstract level, sub-channels are part of the transmitted ensemble, we decided to give the ensemble ownership of the sub-channels.

6.1.4 Processing of the Fast Information Channel

The first part of interest in a DAB transmission frame is the Fast Information Channel (FIC) as introduced in section 3.1.2 on page 15 of this document. As described earlier, this part of the transmission frame carries information on the structure of ensemble. It is comprised of a transmission mode dependent number of Fast Information Blocks (FIBs). Each FIB in turn contains one or more FIGs.

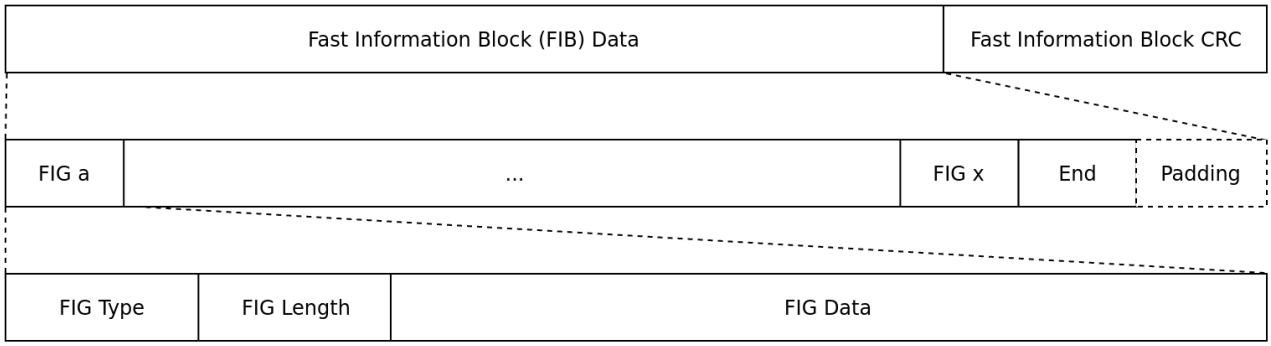


Figure 6.1: Structure of a FIB (adapted from ETSI (8, sec. 5.2.1)).

Figure 6.1 shows a breakdown of the structure of an FIB. Each FIB has a fixed length of 32 bytes. 30 of these bytes contain the data of the FIGs, possibly including an end marker and some padding bytes. The remaining two bytes contain a bit-wise inverted CRC of the transported data. The end marker is implemented as a special FIG with type seven, length 31 and no data field. If the actual transported data occupies exactly 30 bytes, the end marker and padding are omitted. If the data occupies exactly 29 bytes, only the end marker but no padding is present while in any other case, the remaining bytes, after the end marker, are filled with zeros (8, sec. 5.2.1).

Our first approach was to design an object oriented parser that exactly resembled the structure of the FIC. After a first try on implementing this approach, we realised that despite the structural beauty of such an implementation, extending this structure became harder with every feature we implemented. Subsequently we reworked our design and came up with a simple functional-style parser that takes a reference to the bytes that make up the FIC.

Since the information contained in the FIC does not only apply to the ensemble itself but also to its parts, we wrapped the parser into a class. This class is a `friend`[1] of the ensemble as well as its parts. This enabled us to hide the complete parsing process, including the required APIs, from the library user. We decided that this benefit rectifies the breaking of internal encapsulation.

Our implementation supports the following FIG extensions:

FIG 0/0	General ensemble information like country, id, etc.
FIG 0/1	Sub-channel organisation (IDs, addresses and protection levels)
FIG 0/2	Service organisation (ids, related sub-channels and types)
FIG 0/3	Packet mode service components (IDs, related sub-channels and addresses
FIG 1/0	Basic ensemble label
FIG 1/5	Basic data service labels

Table 6.1: FIG extensions supported by our implementation and their meaning according to (8))

[1] The C++ keyword **friend** allows class A, which is not in an inheritance relationship with class B, to access B's **protected** and **private** members

6.1.5 Services and sub-channels

Services represent the second and final part of the public API. At the same time, they are the gateway for the user to gain access to the transmitted data. To gain access to the services of an ensemble, the user can iterate over the the result of the `services()` member function of an ensemble. This call returns a map of service IDs to the actual services. After finding the desired service, either by id, type, or label, the user passes the service and a data processing callback to the `activate(...)` member function of an ensemble. The supplied callback function will then be called every time the services primary component has new data present.

From an internal perspective, services only manage metadata like their label, type and id as well as their relation to their components. Figure 6.2 shows a slightly simplified description of the related classes.

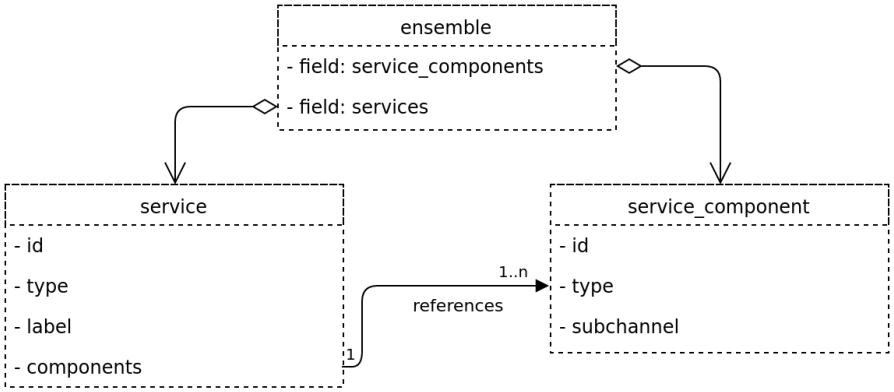


Figure 6.2: Relation of services to service components

Service components are indirectly exposed to the user through the `primary()` member function of a service. This allows the user to ask a service component, what kind of data is being transported by the component. For example, the data type for IPDT is specified to be 59 (15, sec. 5.3).

As mentioned earlier, figure 3.2 on page 15 is missing one additional layer. This layer is called the sub-channels. DAB sub-channels describe the location and the protection scheme that has been applied to a service component's payload. In our design, these descriptors are reflected in a separate class called `subchannel`. Objects of this class handle the processing of the received data and extract the payload from the encoded bit-stream. Since service components and sub-channels posses a one-to-one relationship to each other, we decided to not expose the concept of sub-channels to the user.

6.1.6 Undoing of the convolutional encoding

As already mentioned in section 6.1.2, decoding a convolutional encoding is not a trivial task. We therefore decided to reuse a well known and proven implementation of a decoder. We opted to use the decoder implementation present in GNU Radio. This specific decoder has been developed by experts in the field of signal processing and is used throughout a wide variety of GNU Radio based applications. The source code of GNU Radio as well as the source code of this decoder are released as open source software under the terms of the GPLv3.

6.2 Usage

In this section, we will introduce the API of LibDabDecode with some short code examples. In order to keep the examples as short as possible, we assume a general frame of reference as seen in listing 6.1. Each of the code samples we present here will not compile on its own, but full examples can be found in the source of the endpoint daemons which is included on the CD.

Listing 6.1: Common definitions for the examples

```
// The input queue containing OFDM symbols
dab::symbol_queue_t symbols{};
// The selected transmission mode
auto transmission_mode = dab::transmission_modes::kTransmissionMode1;
// A convenience type alias for a vector of bytes
using byte_vector_t = std::vector<std::uint8_t>;
```

6.2.1 Ensemble initialisation

To get started with LibDabDecode, the user must first instantiate an object of the ensemble class as seen in listing 6.2. The first argument to the constructor call of the ensemble class is the queue that contains the demodulated OFDM symbols. The second argument specifies the transmission mode of the ensemble. This is very important since the exact frame structure depends on several parameters defined in the four different DAB transmission modes.

Listing 6.2: Instantiating an ensemble

```
// Instantiate a new ensemble
auto ensemble = dab::ensemble{symbols, transmission_mode};
```

After creating an ensemble object, the object still contains no data. Thus the next step is to update the ensemble and bring it into a valid state. This process can be seen in listing 6.2. Calling the `update()` member function of the ensemble extracts the symbols making up a transmission frame from the symbol queue and parses them. It updates the ensembles internal service structure, label, ID, etc. An ensemble is considered valid as soon as a label for it has been found.

Listing 6.3: Initializing an ensemble

```
// To check whether the ensemble is valid, it can be casted to bool
while(!ensemble) {
  // Process symbols and initialise the internal structures.
  ensemble.update();
}
```

As soon as the programs control-flow leaves this loop, we know that the ensemble has successfully been initialised. This means that we can now check what services are present in the ensemble by iterating over them. Listing 6.4 shows how this iteration process can be used to print some information about the service to the standard output. Iterating the services returns a pair that associates the service's id to a pointer to the actual service.

Listing 6.4: Iterating an ensemble's services

```
// Iterate over the ensemble's services
for(auto const & serviceDescriptor : ensemble.services()) {
  // Print each service's name
  std::cout << "Service name: " << serviceDescriptor.second->name()
            << "Service id: " << serviceDescriptor.second->id() << '\n';
}
```

After the user has decided which service they would like to consume, the service must be activated in the ensemble. Listing 6.5 shows an example of the activation. The user must pass the pointer, and a callback function to the service. The callback function must return `void` and take exactly one argument of type `byte_vector_t`. As shown in the example, the call also accepts C++ lambda expressions. Our application currently only supports one active service at a time and activating a different service deactivates any currently active service.

Listing 6.5: Activating a service

```
// Extract the service from the descriptor.
auto service = serviceDescriptor.second;

// Activate the service
ensemble.activate(service, [&](byte_vector_t data){
  std::cout << "Received " << data.size() << "bytes!\n";
});
```

The supplied callback will be called as soon as the service has data available. We would like to point out that the callback function will be called on the same thread the `update()` function is being called on. The user should take care to extract long running processing functions into a separate thread and use the callback to enqueue a work item for the processing task.

LibDabDemod

In this chapter, we provide an introduction into the API presented by our demodulation library. We discuss the difficulties, and the decisions that resulted from those.

7.1 Introduction

To transmit data via physical channel, communication technologies like DAB use a technique called modulation. In a nutshell, modulation encodes information on top of a carrier signal by manipulating some of the parameters that define this carrier signal. DAB uses a combination of OFDM and Phase-shift Keying (PSK). OFDM, provides multiple carrier signals which are spaced in a very specific way in order to minimise cross-talk. These carriers are then modulated using PSK. PSK is a modulation technique, that changes the phase of the carrier signal to transport digital data via an analogue channel. For more information on PSK and OFDM, we refer to (39), (28), (38) and (30).

A receiver must reverse this modulation. This process is called demodulation and requires a precise frequency synchronisation between sender and receiver. If the receiver-side aggregates too large of a frequency offset to the sender-side, it can happen that the receiver misidentifies the modulated carriers. This would make it impossible for the receiver to extract the original data stream from the received signal. To enable the receiver to synchronise itself to the sender, a special OFDM symbol – known as the phase reference symbol – is transmitted at the start of each transmission frame. This symbol carries standardised data and thus allows to receiver to correct its frequency offset until it receives exactly the correct data.

Because the data is transmitted as a stream, time synchronisation is also required. To allow the receiver to identify the start of a transmission frame, another special symbol called the null symbol is prepended to the phase reference symbol. This symbol is designed in such a way, that its signal is (almost) equal to zero (17, sec. 2.2.2). This allows the receiver to coarsely synchronise to the start of a transmission frame by searching for a dip in received signal power. For fine time synchronisation, the phase reference symbol is used (17, sec. 2.2.2).

7.1.1 Goals

During the design phase of the library we found the following requirements an implementation must fulfil to be of use:

- Since the data stream coming from a device will come in at a fixed speed, the implementation must be at least as fast as the incoming stream.
- DAB uses a fixed timing between frames. In order to allow for real-time processing of DAB, the implementation must not take longer than one frame to process the incoming signal. Since

we focused on transmission mode one, this means that processing one frame worth of symbols must not take longer than 96ms.

- Due to the fact that a missing frame would cause the loss of transmitted data, the implementation must not lose synchronisation to the sender.
- Since demodulating the signal is a computationally expensive process, the implementation must be able to run in isolation. This allows the user to employ multi-threading to make optimal use of their systems resources.

7.1.2 Difficulties

The major difficulty we faced was the fact that both of us had little to no experience in digital signal processing. At the same time, demodulating an analogue signal is a complex task. At first we were confident that we could create a working implementation ourselves. It turned out however, that even though we understood the basic working principles of OFDM and PSK, implementing a solution that can keep synchronisation and at the same time process the vast amount of incoming data in real-time was way more involved than we though.

7.1.3 Decisions

Because of the aforementioned problem, we – in accordance with Prof. Dr. Farhad Mehta – decided to reuse an existing open source implementation. We therefore adapted the demodulator implementation of Jan van Katwijk's *dab-rpi* to suit our infrastructure. *dab-rpi* is released as open source software under the terms of the GPLv2. This software is able to play DAB audio streams in real-time on a RasPi2.

7.1.4 Design

The user visible part of LibDabDemod is designed to be as minimalist as possible. The only class that is directly exposed to the user is the `struct demodulator`. This class contains the main algorithm to synchronise the receiver to the sender. The demodulator exposes only three functions which allow the user to create a demodulator as well as start and stop the processing of complex samples.

Internally, LibDabDemod uses a couple of abstractions to help with demodulating the received samples into OFDM symbols. These abstraction include classes that wrap the creation and storage of lookup-tables that cache the results of expensive computations and statically defined mappings. This allows the demodulator to translate some of the calculation into simple array accesses which in turn reduces the processing power required during demodulation. Another internal class is used to move the reordering and processing of the carrier signals onto a separate thread, to free the time critical section of the algorithm from having to deal with the high amount of copy and scaling operations.

7.2 Usage

In this section, we will introduce the API of LibDabDemod with some short code examples. In order to keep the examples as short as possible, we assume the code in listing 7.1 as a general frame of reference. The examples we present here will not compile on their own and may require additional libraries. Complete examples can be found in the source of the endpoint daemons which is included on the CD.

Listing 7.1: Common definitions for the examples

```
// The input queue containing complex samples acquired from a device
dab::sample_queue_t samples{};
// The output queue for the demodulated OFDM symbols
dab::symbol_queue_t symbols{};
// The selected transmission mode
auto transmission_mode = dab::transmission_modes::kTransmissionMode1;
```

7.2.1 Demodulator initialisation

The first step in demodulating the signal of a DAB transmission, is to instantiate an object of the `struct demodulator`. Listing 7.2 shows the required constructor call to create a new demodulator. After creating the demodulator, it will be ready to start processing of the acquired complex samples.

Listing 7.2: Creating a demodulator instance

```
// Instantiate a new demodulator
dab::demodulator demod{samples, symbols, transmission_mode};
```

We would like to point out that it is of utmost importance to select the correct transmission mode. Failing to do so will prevent the demodulator from successfully acquiring symbols from the received signal, as the structure and timing of the individual symbols depends on the parameters defined by each transmission mode.

7.2.2 Controlling the demodulation process

With the instance of the demodulator created, the two final remaining tasks are starting and stopping the the demodulation process. Listing 7.3 below shows how to start the demodulation process on a separate thread. As soon as the demodulator is started, it will begin acquiring samples from the queue that was specified at construction time.

Listing 7.3: Starting symbol demodulation

```
// Start demodulation on a separate thread
auto demodRunner = std::async(std::launch::async, [&]{
  demod.run();
});
```

The call to the `run()` member function will block until `stop()` is called on the object. We therefore recommend to start the processing of the samples on its own thread. As shown in listing 7.4, stopping the demodulation process can be achieved by a single function call.

Listing 7.4: Creating a demodulator instance

```
// Stopping the demodulation process
demod.stop();
```

LibDabDevice

In this chapter, we provide an introduction to the API presented by our device abstraction layer. We also present the design decisions, and an architectural overview.

8.1 Introduction and Design

Software based processing of digital wireless transmissions is impossible without any means to acquire the data off the air. For this reason, various vendors created chip-sets that allow access to the transmitted data. These chips often differ in resolution, sampling rate, frequency range, etc. This variety in devices is encompassed by a variety of APIs. This makes it unnecessarily complicated to implement software that can use any specific device. We therefore decided to create a simple abstraction layer that provides a consistent and stable API for developers.

To make the abstraction layer as easy to extend as possible, we decided to extract it as a separate header-only library. This approach minimises the dependencies required to make use of the implementation. For example, if a user is not interested in using device A, they should no be forced to install the required vendor libraries. Instead they can simply include just the header for their device of choice and link their program against the vendor libraries.

We designed a generic data extraction interface based on a thread-safe queue. This allows even developers that do not want to work on DAB to make use of our API. We decided to do so in order to foster creativity amongst our future user base. In addition to the data extraction interface, we designed a generic API that enables the user to achieve common tasks like tuning to a specific frequency. Later in this chapter, we give an overview of the actual API.

The user will receive the extracted data in the form of complex numbers. This design decision is the result of the technique used in DAB transmission. Like other digital wireless technologies, DAB uses Phase-shift Keying to encode the transmitted data. This modulation scheme encodes data as phase shifts of the carrier signal. To recover the amplitude as well as the phase at the same time, a sampling technique known as I/Q-sampling is used. Contrary to simple real sampling, I/Q-sampling produces two values per sample, the so-called in-phase and quadrature components. For a detailed introduction into I/Q-sampling and the associated benefits, we refer to (23), (22), and (32). In order to accommodate for the different resolutions and formats with which different devices capture the sample data, we decided to linearly scale the sample data to the range $[-1.0, 1.0]$ by convention. This means that the implementer of a new device should also apply this scaling in order to be compliant with the current implementation.

8.1.1 Goals

During the design phase of LibDabDevice we found the following requirements our implementation should fulfil:

- The implementation must provide the acquired data in a unified format.
- The implementation should incur minimal overhead over the existing vendor APIs.
- All devices must expose the same API.
- Since the acquired samples will most likely be used in complex processing steps, the implementation shall be thread-safe. This minimises the likelihood of lost samples due to other long running tasks blocking the sample acquisition.
- The implementation should also provide dummy devices that read pre-acquired samples from files. This allows a developer to use known-good data when developing a complex application.

8.1.2 Common infrastructure

Each device type supported by the library inherits from the abstract `struct device`. This class defines the common interface for all devices. In addition to the common interface, this class also provides some data members that are common to all devices, like a reference to the target sample queue. This means that developers wanting to add a new device implementation can make use of a generic underlying infrastructure.

8.1.3 RTL implementations

During our project, we worked with devices based on the *RTL2832U* chipset designed by Realtek Semiconductor. Even though this chipset was originally designed for devices used for receiving Digital Video Broadcasting - Terrestrial (DVB-T), it can also be used in SDR applications. This feature was discovered by several different people in the open source community (31). A major advantage of devices based on this chipset is their low price point when compared to more professional SDR equipment. For example, an *RTL2832U* based USB stick can be bought starting around $20 while the more professional HackRF-One starts at around $299. This huge price difference has made *RTL2832U* based devices very popular and well supported in the open source SDR community.

During the course of our project, we developed two concrete implementations of the `device` interface. The first one being an implementation for an actual hardware device while the second one is a virtual device that can be used to replay recorded sample streams.

Device based implementation

Devices based on this chipset provide the acquired samples as an interleaved stream of unsigned eight bit data. Each samples I-component is directly followed by its Q-component. The open source library *librtlsdr* provides 2 different methods to receive the data. The first method is synchronous sample acquisition. This mode requires the user to provide a buffer for the samples and the function call blocks until the requested number of samples has been acquired or an error occurred. One obvious drawback of this mode of operation is that during the processing step of the samples - for example enqueuing them into a another buffer - no samples can be acquired. Therefore we decided to use the second acquisition mode, called the asynchronous mode. In this mode too, the function call to start the sample acquisition is blocking. In contrast to the synchronous mode however, this mode allows

us to specify a callback function which is called when the requested number of samples has been acquired. The call of the second function is asynchronous, and during the time it takes to handle the data, the devices continues to acquire samples.

File based implementation

To complement the device based implementation, we also implemented a file based implementation. This implementation is especially useful when testing the performance and correctness of other system components. The implementation accepts the path to a file containing a dump of samples acquired from an actual *RTL2832U* based device. To acquire dumps of the samples, we used the *rtl_sdr* utility which ships as part of *librtlsdr*.

8.2 Usage

In this section, we introduce the API exposed by LibDabDevice using short code examples. We will demonstrate how to use our device and file based implementations. We would like to point out, that for any functionality involving a real device, *librtlsdr* is required. To keep the samples as short as possible, we assume the following common context:

Listing 8.1: Common context for the examples

```
// Include support for real devices
#include <device/rtl_device.h>
// Include support for virtual file based devices
#include <device/rtl_file.h>
// The target queue for the complex samples
dab::sample_queue_t samples{};
```

8.2.1 Device initialisation

The first step in using LibDabDevice is to create an instance of the desired device as demonstrated in the listing below.

Listing 8.2: Creating device instances

```
// Create a virtual file device that replays the samples in 'rtl_dump.raw'
dab::rtl_file fileDevice{samples, "rtl_dump.raw"};

// Create a 'real' device
dab::rtl_device realDevice{samples};
```

As the above example demonstrates, the only difference between creating a virtual device and a real device is that the virtual device's constructor takes an additional argument. This argument is the

path to the file that should be used by the device. All relative paths, like in the example, are relative to the current working directory when starting the application. If the file can not be opened or the file does not contain any samples, an exception of type `std::ios::failure` will be thrown by the constructor. For real devices, the constructor will throw an exception of type `std::runtime_error` when one of the following happens:

- No device is connected
- A device is connected but could not be opened
- The default sample-rate of 2.048 MSps could not be set.

After successfully creating a device instance, the user has the opportunity to toggle device specific options. At the moment there are only the two options `dab::device::option::loop` and `dab::device::option::automatic_gain_control` available. The latter only makes sense when used with a hardware device while the former only applies to file devices. As shown in listing 8.3, trying to enable or disable an unsupported option will cause the call to `enable(...)` or `disable(...)` to return false.

Listing 8.3: Setting options on devices

```
// Enable file looping
if(fileDevice.enable(dab::device::option::loop)) {
  std::cout << "Successfully enabled looping!\n";
}

// Enable the AGC of the RTL USB stick
if(realDevice.enable(dab::device::option::automatic_gain_control)) {
  std::cout << "Enabled the device's AGC!\n";
}

// Trying to set an unsupported option will fail
if(!realDevice.enable(dab::device::option::loop)) {
  std::cout << "Failed to enable looping on the real device!\n";
}

// Trying to disable an unsupported option will fail too
if(!realDevice.disable(dab::device::option::loop)) {
  std::cout << "Failed to enable looping on the real device!\n";
}
```

The next step and final initialisation step, tuning to a desired frequency, applies only to hardware devices. Listing 8.4 demonstrates this. The member function `tune(...)` expects an argument of type `dab::frequency`. We designed this type as a simple wrapper for an unsigned 32 bit value, to make it clear, what the argument represents. In addition to the type, we provide user defined literals for common frequency units like Hz and kHz. Using these literals makes code much easier to read without incurring any performance or memory overhead.

Listing 8.4: Tuning to a specific frequency

```
// Tune to a specific frequency
if(realDevice.tune(218640_kHz) {
  std::cout << "Successfully tuned to channel 11B!\n";
}
```

8.2.2 Managing sample acquisition

After successful tuning, it is now possible to start receiving samples from the device. Listing 8.5 shows how to do this using the hardware device, but of course this also applies to virtual devices. The call to `run()` blocks until sample acquisition is aborted. For this reason, we recommend to call `run()` on a separate thread.

Listing 8.5: Starting sample acquisition

```
// Start sample acquisition on a separate thread
auto deviceRunner = std::async(std::launch::async, [&]{
  realDevice.run();
});
```

The acquisition of samples can be stopped as shown in listing 8.6. Calling the `stop()` member function will cause the prior call to `run()` to return after the device cleaned up any asynchronous operations.

Listing 8.6: Stopping sample acquisition

```
// Stop the acquisition run-loop
realDevice.stop();
```

LibDabIp

In this chapter, we introduce the API presented by our IP - DAB conversion library. Furthermore we explain the design decisions and provide an architectural overview.

9.1 Introduction and Design

IP datagram tunnelling (4) requires the user to wrap each transported IP datagram in an MSCdg. This MSCdg will then be split into parts small enough to fit inside a DAB packet. This results in 1 to 91 packets that must be reassembled to an MSCdg at the receiver-side. Finally the IP datagram must be extracted from the MSCdg.

This requires a two tier design. The sender-side builds MSCdgs from IP datagrams, these MSCdgs are then used to build DAB packets. The receiver-side does the same conversion, but in reversed order. This means only those part of the library that will be used must be compiled and linked.

To ensure high modularity and reusability we designed the generating and parsing process in two steps. One handles MSCdgs, the other DAB packets. This way future applications, where for example no MSCdgs are needed, can easily be implemented by using only the DAB packet generator and parser.

To keep the API simple, parsing and generating are done with just one function-call.

9.1.1 Goals

During the design phase of this library we identified the following requirements that need to be covered by an acceptable implementation:

- The implementation should hide all the complicated processes, the splitting, the packaging and the CRC generation as well as the validity evaluation, the parsing and the reassembling.
- While parsing, parsed MSCdgs shall only be returned, if no errors were detected.
- The parser must be able to signal its current state.
- The library must be modular. This means packing and parsing for MSCdgs and IP datagrams must be two separated function calls.
- The generating and parsing processes should be efficient. This means, the processing time per DAB packet shall not exceed 96ms on one core with 2GHz.

9.1.2 Difficulties

- ETSI TS101735 (4) specifies that each IP datagram shall be represented by exactly one MSC data group. As specified in Request For Comments (RFC) 791 (20) an IP datagram has a minimal length of 576 bytes. The DAB packet comes in four sizes (24 bytes, 48 bytes, 72 bytes, 96 bytes). So every MSC data group must be split into multiple DAB packets. Therefore the DAB packet parser must be able to collect as many packets as needed to reassemble the whole MSC data group.

- In the course of this work, we only implemented those features that were required by our use-cases. The features we did not implement include the MSCdgs session functionality, DAB Conditional Access (CA) functionality and command-packets.

9.2 Common architecture

The core components of the LibDabIp are designed as four classes.
But first some types (listing 9.1), that will help understanding the code snippets featured in this chapter.

Listing 9.1: libdabip usings

```
using byte_vector_t = std::vector<std::uint8_t>;
using pair_status_vector_t = std::pair<parse_status, byte_vector_t>;
```

A `pair_status_vector_t` is returned by parsers. The `pair.first` value contains a `parse_status`. If the value of `first` is `parse_status::ok` the `byte_vector_t` in `pair.second` contains the parsed bytes.

The type `parse_status` is defined by an `enum-struct` which offers the parse statuses available to the parsers. It is defined as shown in listing 9.2.

Listing 9.2: Enum parse_status

```
enum struct parse_status : std::uint8_t
    {
    invalid_crc,
    invalid_address,
    incomplete,
    segment_lost,
    ok
    };
```

9.3 DAB packets

As required by the ETSI standard about IP datagram tunnelling (3, sec. 4) we use a packet mode sub-channel for data transmission. This sub-channel type requires everything sent to be wrapped into DAB packets. These DAB packets are created and parsed by the LibDabIp.

9.3.1 Difficulties

The biggest difficulty we had to face on packet-level is the fact that DAB can not guarantee that a packet will be received. This raises the question, how to detect dropped packets and how to react in this case.

The detection mechanism for lost packets is implemented by a modulo-4 counter in the packet header (see table 9.1). If a packet was dropped, a number in the counter will be skipped. This detection is highly limited by the size of the counter. The modulo-4 counter can not detect, if a multiple of 4 packets have been lost. Considering this, DAB is not able to guarantee integrity of the parsed data.

If a continuity index (see continuity index in table 9.1) was skipped, we drop all packets until a new packet group starts. This means, every packet will be dropped until one occurs that has the First-flag set. To avoid passing incomplete data groups we do not return any data after any kind of error occurred.

9.3.2 Structure

The structure of a DAB packet is shown in detail in figure 9.1. The key to the figure can be found in table 9.1.

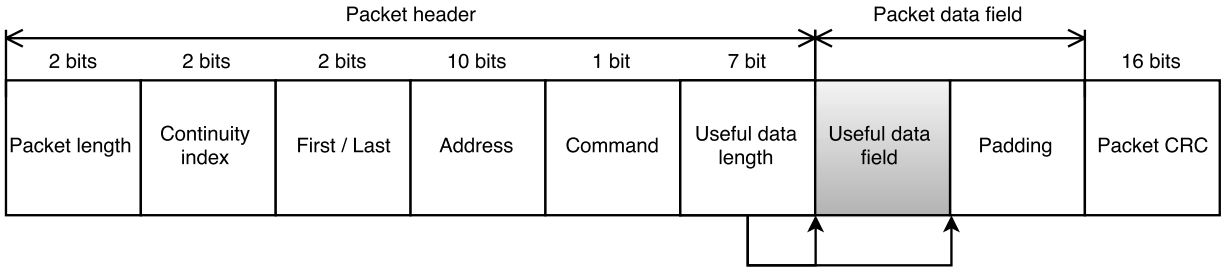


Figure 9.1: DAB packet structure (key in table 9.1) (adapted from ETSI (8, chap. 5.3.2.1))

Packet length	Represents the length of the packet including the header and the CRC. Values 00 to 11 are mapped to a size using the following formula $(packetlength + 1) * 24$.
Continuity index	A 2-bit modulo 4 counter that is incremented by one for each successive packet in with the same address.
First / Last	These two flags indicate the packets position in the MSC data group. If there is only one packet, it is 11 for first and last, if it is an intermediate packet it is 00. First packet is 10 and the last 01.
Address	The address of the service component in the sub-channel. This allows distinguishing between upto 1023 service components.
Useful data length	The length of the Useful data field.
Useful data field	The field that carries the actual payload.
Padding	The packet must be one of the four sizes allowed by the packet length. If the Useful data field is not long enough, there must be padding.
Packet CRC	A 16-bit CRC word generated over the packet header and the Useful data field. It must be calculated using the polynomial $x^{16} + x^{12} + x^5 + 1$ while the registers are initialised to 1. The CRC must be bitwise inverted before transmission.

Table 9.1: DAB packet fields (adapted from ETSI (8, chap. 5.3.2))

9.3.3 DAB packet generator

The `packet_generator` will be instantiated by the user of the LibDabIp to generate DAB packets from an MSC data group. On instantiation of the `packet_generator`, a service-address (see address in table 9.1) must be passed. There must not exist more than one `packet_generator` with the same service-address per sub-channel. If this is rule violated, a correct order in the DAB packets can not be guaranteed, therefore successful unpacking at the receiver-side can not be guaranteed. The service-address must be an integer value in the range $[1, 1023]$.

Listing 9.3: Instantiation of a DAB packet generator

```
// Address is an unsigned 16-bit int.
std::uint16_t address = ...;

// Instantiates a new packet generator.
auto generator = packet_generator(address);
```

To build the DAB packets from an MSC data group, the user needs only one function call. As shown in listing 9.4, `build(msc_data_group)` takes a `byte_vector_t` as argument and returns a `byte_vector_t`. The returned `byte_vector_t` contains all the bytes of the DAB packets that are required to carry the bytes of the `msc_data_group`. This means `dab_packets` contain the bytes representing 1 to 91 DAB packets, depending on the size of the MSC data group. This process allows wrapping of any kind of data that can be represented using bytes.

Listing 9.4: Usage of the DAB packet generator

```
// The variable msc_data_group contains bytes representing an MSC data group.
byte_vector_t msc_data_group{...};

// This call generates the bytes for the DAB packets from msc_data_group.
byte_vector_t dab_packets = generator.build(msc_data_group);
```

The generator is designed as a class. A class, because it must keep a state to continuously increase the index number in the generated DAB packets. Furthermore, there is a member variable holding the service-address.

9.3.4 DAB packet parser

The `packet_parser` will be instantiated by the user of the LibDabIp to parse MSC data groups from DAB packets. Like the `packet_generator`, the `packet_parser` must be constructed with a service-address. The parser can only use packets with the same service-address it was created with, all other packets will be dropped. If one works on a sub-channel that features multiple services with different addresses, one parser per address must be created.

Listing 9.5: Instantiation of the DAB packet parser

```
// Address is an unsigned 16-bit int.
std::uint16_t address = ...;

// Instantiates a new DAB packet parser.
auto parser = packet_parser(address);
```

Using the `packet_parser` is an iterative process. As shown in listing 9.4, `parse(dab_packets)` takes a `byte_vector_t` as its sole argument and returns a `pair_status_vector_t`. The returned `pair_status_vector_t` contains the `parse_status` as `first`. If `parse_status` is `parse_status::ok`, the bytes representing an MSC data group are available via `second`.

The `packet_parser` returns `parse_status::incomplete` until all the DAB packets, necessary to complete the actual MSC data group, are parsed or until the parsing had to be aborted. An abortion can be caused by a packet with invalid CRC or by a dropped packet.

In a future iteration, we plan to add a feature that enables the parser to consume a `byte_vector_t` that may contain the bytes of more than one DAB packet, but may not contain DAB packets of different MSC data groups.

Listing 9.6: Usage of the DAB packet parser

```
//The variable dab_packet contains the bytes of a DAB packet.
byte_vector_t dab_packet{...};

//This call parses an MSC data group from a DAB packet.
pair_status_vector_t msc_data_group = parser.parse(dab_packet);
```

9.4 MSC data group

One of the main structures in DAB is the MSC data group. As required by ETSI ES 201735 (3, sec. 5.1) and ETSI TS 101735 (4, sec. 5.4.1), we wrap each IP datagram in a MSC data group before packaging.

9.4.1 Difficulties

The biggest difficulty we faced on the MSCdg level is the complexity of the session headers. Although we do not support these headers, we must be able to parse them to evaluate the size of the headers. What makes this difficult is the fact that the session header is variable in size (see Figure 9.2). This kind of best-effort parsing turned out to be hard to implement efficiently.

9.4.2 Structure

Figure 9.2 shows the detailed structure of an MSC data group. Table 9.2 explains the function of the fields we used.

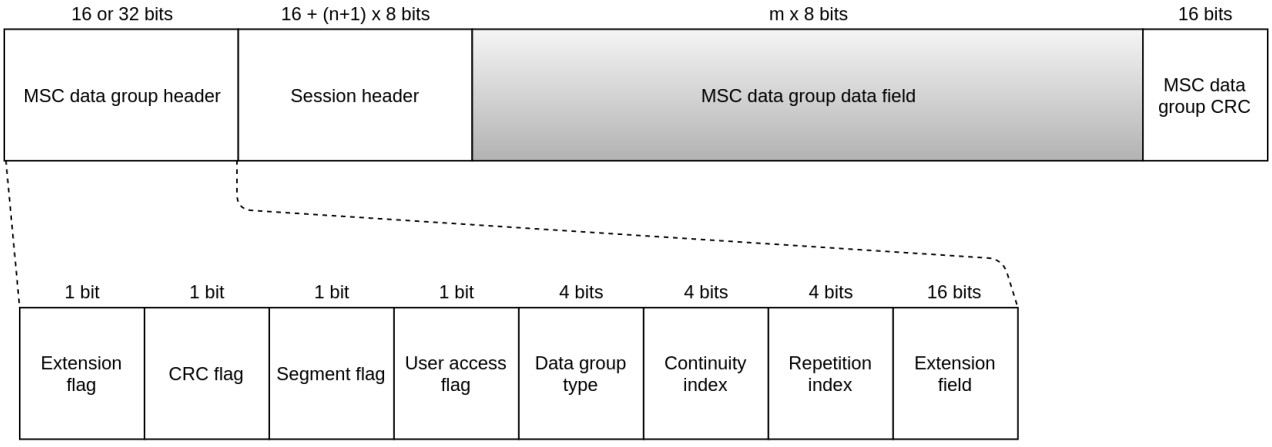


Figure 9.2: Shortened MSC data group structure (adapted from ETSI (8, chap. 5.3.3.1))

CRC flag	Indicates, if the MSC data group has a CRC attached.
Segment flag	Defines, if the MSC data group is segmented. If this field is set to 1, a session header containing a "Segment field" is required. Support for segmentation is prepared but not functional yet.
Data group type	Is set to 0000 in our application. 0000 means General data.
Continuity index	Will be incremented each time when a MSC data group, with a content different from the last sent MSC data group of the same type, is sent.
Repetition index	Indicates the remaining repetitions of the same MSC data group. This means, the Continuity index must stay the same, while the Repetition index counts to 0000. The value 1111 means, the MSC data group will be repeated for an undefined period.
MSC data group data field	Contains an integral number of bytes, with a maximum of 8191 bytes. This field carries the payload.
MSC data group CRC	Will contain a CRC word generated over the packet header and the Useful data field if the CRC flag is set. The CRC must be calculated using the polynomial $x^{16} + x^{12} + x^5 + 1$ while the registers are initialised to 1. The CRC must be bitwise inverted before transmission.

Table 9.2: MSC data group fields (adapted from ETSI (8, chap. 5.3.3))

9.4.3 MSC data group generator

The `msc_data_group_generator` will be instantiated by the user of the LibDabIp to generate MSC data groups from IP datagrams. The `msc_data_group_generator` is default constructible.

Listing 9.7: Instantiation of the MSC data group generator

```
// Instantiates a new MSC data group generator.
auto generator = msc_data_group_generator();
```

To build an MSC data group, the user only needs one function call. As shown in listing 9.8, `build(ip_datagram)` takes a `byte_vector_t` as its argument and returns a `byte_vector_t`. The returned `byte_vector_t` contains a MSC data group with the bytes of `ip_datagram` as payload. This process allows wrapping of any kind of data representable by bytes. ETSI TS101735 (4) requires to wrap exactly one IP datagram in one MSC data group, but there are other uses, where other type of data must be packed into MSC data groups.

Listing 9.8: Usage of the MSC data group generator

```
// The variable ip_datagram contains bytes resembling an IP datagram
byte_vector_t ip_datagram{...};

// This call generates the bytes for the MSC data group.
byte_vector_t msc_data_group = generator.build(ip_datagram);
```

The `msc_data_group_generator` needs to keep a state to continuously increase the continuity index in the generated MSC data groups. Therefore we designed it as a class.

9.4.4 MSC data group parser

Like the `msc_data_group_generator`, the `msc_data_group_parser` is default constructible.

Listing 9.9: Instantiation of the MSC data group parser

```
// Instantiates a new MSC data group parser.
auto parser = msc_data_group_parser();
```

The parsing is simple. All that has to be done is to call `parse(msc_data_group)` and if no error occurred, `second` contains the bytes for an IP datagram. Possible causes for errors are invalid CRCs and missing segments, even if support for segmentation is not fully implemented yet.

Listing 9.10: Usage of the MSC data group parser

```
// The variable msc_data_group contains bytes resembling an MSC data group
byte_vector_t msc_data_group{...};

// This call parses an IP datagram from a byte_vector_t containing a
↪  msc_data_group
pair_status_vector_t ip_datagram = parser.parse(msc_data_group);
```

Part IV

Conclusion

Conclusion

This chapter contains our conclusion to our decisions, our lapses and the whole project.

10.1 Decisions

IP

We are confident that IP was the right choice. Still it brings networking into play and networking can not work out-of-the-box. Networking must be configurable, flexible, and modular.

IP Data Casting vs IP Datagram Tunneling

IPDT is a good approach. But still we can not say if we would have obtained better results with IPDC. For sure our implementation proves that it is possible to do this in software, and if necessary it can be remodelled to work with IPDC. This means the libraries can be used as foundation for future extensions.

10.2 Lapses

Documentation

When we came closer to the deadline we realised we should have started writing earlier. We still doubt that it would have been possible for Felix Morgner to join the writing earlier, as he was busy with testing and bug-fixing. Although we were short on time we still managed to ask three people to review our thesis concerning orthography.

Embedded systems

In the middle of the thesis we received the request to enable our solution to be run on an embedded systems like RasPi2. During the final test-run it turned out that the demodulator is not able to consume the acquired samples fast enough. We used the *perf* utility to evaluate how much time was spent in which part of the application and it seemed like the extraction of samples out of the queue uses about one third of the process time.

In short, the RasPi2 ran out of memory soon and terminated the application. More about possible countermeasures in section 11.3.

10.3 The whole project

Making DAB easy to use

DAB is not designed to be simple to use. There are many legal limitations which bring some difficulties. For example, running a DAB transmitter requires the user to be authorised to do so by BAKOM.

Then there are technical limitations. For example, the user must know the service-address of the data-packets in order to configure their DAB packet parser correctly. The configuration of the endpoint daemons requires the user to specify the target IP address of the packets of interest.

All in all our solution is easy to use, but requires some expertise. An IT specialist is not required to know a lot about the mechanisms in DAB. All they must know is how to configure ODR-DabMux, ODR-DabMod, and on which channel and in which transmission mode they want to transport their data.

Separation of responsibilities

We are quite happy with the used separation. Each of the authors could employ his strengths where they were most effective. And even if not both did the same amount of programming or documentation, we consider both contributions as being about the same size and extent.

Future work

This chapter covers what still remains to be done and where there is room for improvement.

11.1 Endpoint daemons

The endpoint daemons are fully functional but in a next iteration they should be enabled to read their configuration from a file. By now the endpoint are configured with command-line arguments. The command-line argument handling can also be improved by allowing dynamic argument orders. Additionally, the error handling and status output provide a lot of room for improvements.

11.2 Support for services

Our implementation currently only supports the processing of DAB data services. This limitation is the result of our decision to focus our efforts on the goal of our thesis. Future projects might thus include adding support for extended stream mode and audio programmes as well as additional service types.

11.3 Performance issues

When we first tested our "ready-to-use solution" on a RasPi2 it turned out that the samples in the sample-queue were not consumed fast enough. The queue grew in size and the operating system terminated our application. We are not quite sure about which part was not fast enough, but we suspect that one of the following parts is the bottleneck.

We analysed the application with *perf* and it seemed like the acquisition of samples was not fast enough. It is also possible that the demodulation process is to slow. The queue we use at the moment can only deliver one sample at the time and even if the throughput is remarkable, the thousands of copy-operations might just not be fast enough.

We recommend trying to replace the queue with a bounded buffer or the like that can deliver more than one sample at a time. Maybe this reduction of copy operations is sufficient.

If the queue is not the bottleneck but more performance measurements have to be taken to narrow down the cause of the problem.

11.4 ODR-DabMux Zero-MQ

Zero-MQ is a message queue which allows for easy message passing between two processes.

Today ODR-DabMux supports data delivery using Zero-MQ only for some selected service types. Packet services are not yet enabled to use input from a Zero-MQ. As it only uses file input, we write the packets that we want to transmit in a file which will be read by ODR-DabMux. Once Zero-MQ support for packet services is implemented in ODR-DabMux, delivering raw-packets to the multiplexer will be a lot simpler, and faster.

Part V

Appendix

Documentation LibDabDecode

This appendix has been moved to the separate document ”Data over DAB Appendix”.

Documentation LibDabDemod

This appendix has been moved to the separate document ”Data over DAB Appendix”.

Documentation LibDabDevice

This appendix has been moved to the separate document ”Data over DAB Appendix”.

Documentation LibDabIp

This appendix has been moved to the separate document ”Data over DAB Appendix”.

Documentation LibDabCommon

This appendix has been moved to the separate document ”Data over DAB Appendix”.

Task description

This chapter shows the scanned original of the task-description.
Therefore the design might be inconsistent with the other parts of the thesis.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Task Description – Bachelor Thesis
"Data over DAB"
FS 2016

1. Client & Supervisor

- *Client:* opendigitalradio.org
- *Client Contact:* Matthias Brändli, opendigitalradio.org
- *Supervisor:* Prof. Dr. Farhad Mehta, HSR Rapperswil

2. Students

- Felix Morgner
- Tobias Stauber

3. Setting

Opendigitalradio is a non-profit association that is developing a software suite, called the ODR-mmbTools, that can be used to broadcast any kind of data using DAB+. These tools are used by different organizations throughout Europe to provide digital terrestrial broadcasts in a very cost-efficient way.

DAB and DAB+ are intrinsically digital technologies which, besides digital audio broadcasting, allow for the transmission of arbitrary data. However, the currently available tools are tailored to digital audio and associated auxiliary data. There is currently no support for a generic top-level protocol that allows for arbitrary data to be sent by DAB/DAB+ stations or received by digital receivers.

DAB and DAB+ are broadcast technologies, meaning that there is no return route for receivers to provide feedback to the sender. Some examples for systems that could be implemented using such a generic data transmission protocol are information signs for public transportation stations – like bus and tram stations – or signs that show the amount of free spaces available on different parking lots throughout a city.

4. Goals

The client would like to have a solution with the following attributes:

1. It must have a stable and compact core-specification.
2. It must provide means to be extended for application specific usages like cryptography and this without breaking the core-specifications.

Prof. Dr. Farhad Mehta ▪ farhad.mehta@hsr.ch

HSR Hochschule für Technik Rapperswil ▪ Oberseestrasse 10 ▪ CH-8640 Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

3. It should provide ways for a receiving party to decide whether a message is intended for them or not.
4. The solution has to provide a generic interface to applications to enable a diverse set of applications to communicate via a stable API.

The students will have to develop and write an exact specification of the protocol. Additionally, they will have to implement demonstration applications for both, the sender as well the receiver side. All of the products created during this thesis shall be released under an Open-Source-Software (OSS) license.

5. Guidelines

The students and the supervisor will plan weekly meetings to check and discuss progress. The student will schedule meetings with the client as and when required (recommendation: 1 meeting per week of 1 hour duration).

All meetings are to be prepared by the students with an agenda. The agenda will be sent at least 24h prior to the meeting. The results will be documented in meeting minutes that will be sent to the supervisor.

A project plan must be developed at the beginning of the thesis to promote continuous and visible work progress. For every milestone defined in the project plan, the temporary versions of all artefacts need to be submitted. The students will receive a provisional feedback for the submitted milestone results. The definitive grading is however only based on the final results of the formally submitted report.

6. Documentation

The project must be documented according to the regulations of the Computer Science Department at HSR (see https://www.hsr.ch/Allgemeine-Infos-Bachelor-und.4418.0.html). All required documents are to be listed in the project plan. All documents must be continuously updated, and should document the project results in a consistent form upon final submission. All documentation and work artefacts have to be completely submitted in three copies on CD/DVD (one copy each for the client, university, and supervisor). Three printed copies of the report need to be submitted (one copy each for the client, external examiner, and supervisor).

Prof. Dr. Farhad Mehta ▪ farhad.mehta@hsr.ch
HSR Hochschule für Technik Rapperswil ▪ Oberseestrasse 10 ▪ CH-8640 Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

7. Important Dates

Please refer to https://www.hsr.ch/Termine-Diplom-Bachelor-und.5142.0.html.

8. Workload

A successful bachelor thesis project results in 12 ECTS credit points per student. One ECTS points corresponds to a work effort of 30 hours.

All time spent on the project must be recorded and documented.

9. Grading

The HSR supervisor is responsible for grading the master thesis. The following table gives an overview of the weights used for grading.

Facet	Weight
1. Organisation, Execution	1/6
2. Report	1/6
3. Content	3/6
4. Final Presentation & Examination	1/6

The effective regulations of the HSR and Department of Computer Science apply (see https://www.hsr.ch/Ablaeufe-und-Regelungen-Studie.7479.0.html).

Rapperswil, 21.02.2016
Prof. Dr. Farhad Mehta

Prof. Dr. Farhad Mehta ■ farhad.mehta@hsr.ch

HSR Hochschule für Technik Rapperswil ■ Oberseestrasse 10 ■ CH-8640 Rapperswil

List of Figures

List of Code-samples

Glossary

ALC

Asynchronous Layered Coding. 27

AM

Amplitude Modulation. 18

API

Application Programming Interface. 30, 37, 39–41, 43, 44, 47–49, 53, 85, *Glossary:* Application Programming Interface

Application Programming Interface

An interface, that allows communication between libraries and programs. 30, 85

BAKOM

Bundesamt für Kommunikation (Federal Office of Communications). 63

BSD 3-Clause License

The license we release the work of this thesis under. See (1). 14

C++

the C++ programming language. 3, 14, 16, 39, 42, 68

CA

Conditional Access. 54, 85, *Glossary:* Conditional Access

CENELEC

European Committee for Electrotechnical Standardization. 25

CIF

Common Interleaved Frame. 20

Coded Orthogonal Frequency Division Multiplex

Modulation method (division into small units, chronological nesting, transmission on various carrier frequencies). COFDM is used in DAB, DAB+ and DMB. A synonym for OFDM. Reproduced from World DAB (41). 85, 89

COFDM

Coded Orthogonal Frequency Division Multiplex. 85, 89, *Glossary:* Coded Orthogonal Frequency Division Multiplex

Conditional Access

Mechanism by which the user access to service components can be restricted. Reproduced from World DAB (41). 54, 85

CRC

Cyclic Redundancy Check. 28, 39, 53, 56, 57, 59, 60, 86, *Glossary:* Cyclic Redundancy Check

Cyclic Redundancy Check

A type of function that takes as input a data stream of any length and produces as output a value of a certain fixed size. Can be used as a checksum to detect accidental alteration of data during transmission or storage. Reproduced from World DAB (41). 28, 86

DAB

Digital Audio Broadcasting. 1, 3, 5–7, 10–15, 18–25, 27, 28, 30, 31, 37–41, 43–45, 47, 53–58, 63, 64, 66, 68, 81, 83, 86, 88, *Glossary:* Digital Audio Broadcasting

DAB+

Digital Audio Broadcasting Plus. 10, 18, 23, 86, *Glossary:* Digital Audio Broadcasting Plus

DABp

DAB Packet. 26, 28

Digital Audio Broadcasting

A digital radio broadcasting technology using MPEG-2 Streams. 1, 5, 10, 18, 86

Digital Audio Broadcasting Plus

The next iteration of the DAB technology. Mostly remarkable change is the use of MPEG-4 AAC as a replacement for MPEG-2. 10, 86

DRC

Dynamic Range Control. 20

DVB

Digital Video Broadcasting. 27

DVB-H

Digital Video Broadcasting - Handheld. 27, *see* DVB

DVB-T

Digital Video Broadcasting - Terrestrial. 48, *see* DVB

EBU

European Broadcast Union. 25

EN

European Standard. 23, *see* ETSI

ES

ETSI Standard. 23, *see* ETSI

ETSI

European Telecommunications Standards Institute. 6, 23, 25, 54, 55, 58, 59, 87, *Glossary:* European Telecommunications Standards Institute

European Telecommunications Standards Institute

Standardization organization in the telecommunications industry in Europe. 23, 87

F-PAD

Fixed PAD. 20, 81

FIB

Fast Information Block. 19, 39, 81

FIC

Fast Information Channel. 6, 19, 20, 25, 37, 39

Field Programmable Gate Array

Hardware, that can be reprogrammed to a specific purpose. 1, 87

FIG

Fast Information Group. 24, 39

FLUTE

File Delivery over Unidirectional Transport. 26, 27

FM

Frequency Modulation. 18

FPGA

Field Programmable Gate Array. 1, 87, *Glossary:* Field Programmable Gate Array

GNU Radio

GNU Radio is an open source platform to design software defined radio applications and process real-world signals in real-time. More information can found at `http://gnuradio.org`. 25, 40

GPLv2

GNU General Public License version 2. 24, 44

GPLv3

GNU General Public License version 3. 24, 40

IP

Internet Protocol. 3, 5, 6, 10, 14, 18, 21–23, 26–28, 30, 31, 53–55, 58–60, 62, 63, *see* IPv4 & IPv6

IP version 4

IPv4 uses 32 bit addresses. Today the available IPv4 addresses are rare. Adapted from Information Sciences Institute (20). 21, 88

IP version 6

IPv6 uses 128 bit addresses. Therefore there are more addresses than in IPv4. Adapted from Information Sciences Institute (20). 14, 88

IPDC

IP Data Casting. 6, 27–29, 62, *see* IP

IPDT

IP Datagram Tunneling. 6, 27–29, 31, 40, 62, *see* IP

IPv4

IP version 4. 21, 26, 28, 81, 88, *Glossary:* IP version 4

IPv6

IP version 6. 14, 21, 26, 88, *Glossary:* IP version 6

Journaline®

is a technology developed at the Frauenhofer IIS that is designed to transport structured trees of text information with very low bandwidth requirements. Adapted from IIS (19). 20

Kernel

The kernel is a computer program that constitutes the central core of a computer's operating system. Reproduced from Wikipedia (34). 14, 26

LCT

Layered Coding Transport. 27

Main Service Channel

A time-interleaved data channel divided into a number of sub-channels which are individually convolutionally coded, with equal or unequal error protection. Each sub-channel may carry one or more service components. Used to carry audio and data service components. Reproduced from World DAB (41). 19, 89

MOT

Multimedia Object Transfer. 6, 20, 25, 88, *Glossary:* Multimedia Object Transfer

MPE

Multiprotocol Encapsulation. 27

MPEG-2

MPEG-2 (aka H.222/H.262 as defined by the ITU) is a standard for "the generic coding of moving pictures. It describes a combination of lossy video compression and lossy audio data compression methods, which permit storage and transmission of movies using currently available storage media and transmission bandwidth. While MPEG-2 is not as efficient as newer standards such as H.264 or H.265/HEVC, backwards compatibility with existing hardware and software means it is still widely used, for example in over-the-air digital television broadcasting (DVB-T) and in the DAB standard. Adapted from Wikipedia (36). 18, 27, 29, 86

MPEG-4 AAC

MPEG-4 Advanced Audio Codec. 18, 24, 86, 89, *Glossary:* MPEG-4 Advanced Audio Codec

MPEG-4 Advanced Audio Codec

AAC has been standardized by ISO and IEC, as part of the MPEG-2 and MPEG-4 specifications. Part of the AAC known as High Efficiency Advanced Audio Coding (HE-AAC) which is part of MPEG-4 Audio is also adopted into digital radio standards like DAB+ and Digital Radio Mondiale, as well as mobile television standards DVB-H and ATSC-M/H. Adapted from Wikipedia (33). 18, 89

MSC

Main Service Channel. 7, 19, 20, 26, 28, 54, 56–60, 81, 89, *Glossary:* Main Service Channel

MSCdg

MSC data group. 26, 28, 53, 54, 58

MTU

Maximal Transmission Unit. 28

Multimedia Broadcast Multicast Service

A point-to-multipoint interface specification for existing and upcoming 3GPP cellular networks, which is designed to provide efficient delivery of broadcast and multicast services, both within a cell as well as within the core network. Reproduced from Wikipedia (35). 27

Multimedia Object Transfer

MOT (EN 301 234) is the transmission protocol used in DAB for transferring file oriented data in DAB audio or data channels. It is tailored to the needs of multimedia services and the specific constraints given by the broadcasting characteristics of the DAB system. It supports two different transport modes, MOT header mode and MOT directory mode. Reproduced from World DAB (41). 6, 20, 25, 88

NPAD

Non Programme Associated Data. 5, 20, 21, *see* PAD

ODR

Opendigitalradio. 10, 14

OFDM

Orthogonal Frequency-Division Multiplexing. 1, 31, 37, 41, 43, 44, 85, 89, *Glossary:* Orthogonal Frequency-Division Multiplexing

Orthogonal Frequency-Division Multiplexing

Used as a synonym for Coded Orthogonal Frequency Division Multiplex (COFDM). Reproduced from World DAB (41). 1, 31, 89

PAD

Programme Associated Data. 5, 20, 21, 38, 87, 89, *Glossary:* Programme Associated Data

Programme Associated Data

Data that is transmitted in association with an audio service. This data can further describe the audio service or provide additional media like album covers, lyrics or slideshows. Adapted from World DAB (41). 5, 20, 38, 89

PSI

Programme Specific Information. 27

PSK

Phase-shift Keying. 43, 44, 47

Radio Data Service

Transmission of additional text data via FM broadcast. Reproduced from World DAB (41). 25, 90

Raspberry Pi 2 Model B

A credit card sized single board computer with a 0.9 GHz quad-core ARM Cortex-A7 and 1 Gb memory. Adapted from Wikipedia (37). 24, 90

RasPi2

Raspberry Pi 2 Model B. 24, 44, 62, 64, 90, *Glossary:* Raspberry Pi 2 Model B

RDS

Radio Data Service. 25, 90, *Glossary:* Radio Data Service

Request For Comments

A type of publication from the Internet Engineering Task Force. 54, 90

RFC

Request For Comments. 54, 90, *Glossary:* Request For Comments

RTP

Real-time Transport Protocol. 27

SC

Synchronisation Channel. 19

SDR

Software Defined Radio. 1, 10, 24–26, 31, 48, 90, *Glossary:* Software Defined Radio

SI

Service Information. 27

Software Defined Radio

Hardware to build a radio that is configured using software. 1, 10, 90

SRTP

Secure RTP. 27, *see* RTP

TCP

Transmission Control Protocol. 5, 18, 21–23

TDC

Transparent Data Channel. 6, 26–28

TII

Transmitter Identification Information signal. 19

TR

ETSI Technical Report. 23, *see* ETSI

TS

ETSI Technical Specification. 23, *see* ETSI

UDP

User Datagram Protocol. 5, 14, 18, 21–23, 26–30

Universal Software Radio Peripheral

A highly optimized hardware component, designed to run an industrial radio application. 10, 91

USRP

Universal Software Radio Peripheral. 10, 91, *Glossary:* Universal Software Radio Peripheral

WLAN

Wireless LAN. 12

WorldDAB

A global forum for digital radio. 25

X-PAD

Extended Programme Associated Data. 20, 81, *see* PAD

Bibliography

[1] BSD. BSD 3-Clause License, May 2016. URL `https://opensource.org/licenses/BSD-3-Clause`. [Online; last accessed 16-June-2016].

[2] Stephen Deering. Host Extensions for IP Multicasting. RFC 1112, Stanford University, 1989.

[3] ETSI. ETSI ES 201735 Internet Protocol (IP) datagram tunnelling. Etsi standard, ETSI, 2000.

[4] ETSI. ETSi TS 101735 Internet Protocol (IP) datagram tunnelling. Technical specification, ETSI, 2000.

[5] ETSI. ETSI TR 101497 Rules of Operation for the Multimedia Object Transfer Protocol. Technical report, ETSI, 2002.

[6] ETSI. ETSI EN 301192 DVB specification for data broadcasting. Etsi norm, ETSI, 2004.

[7] ETSI. ETSI TS 101759 Data Broadcasting - Transparent Data Channel (TDC). Technical specification, ETSI, 2005.

[8] ETSI. ETSI EN 300401 Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers. Etsi norm, ETSI, 2006.

[9] ETSI. ETSI EN 301234 Digital Audio Broadcasting (DAB) Multimedia Object Transfer (MOT) protocol. Etsi norm, ETSI, 2006.

[10] ETSI. ETSI TS 102978 IPDC Services; Transport specification. Technical specification, ETSI, 2008.

[11] ETSI. ETSI TS 102979 Journaline; User application specification. Technical specification, ETSI, 2008.

[12] ETSI. ETSI TR 101495 Guide to DAB standards; Guidelines and Bibliography. Technical report, ETSI, 2012.

[13] ETSI. ETSI TS 101499 MOT SlideShow; User Application Specification. Technical specification, ETSI, 2013.

[14] ETSI. ETSI TS 103177 Filecasting; User application specification. Technical specification, ETSI, 2013.

[15] ETSI. ETSI TS 101756 Registered Tables. Technical specification, ETSI, 2015.

[16] Stephen Deering , Robert Hinden. Internet Protocol, Version 6 (IPv6) - Specification. RFC 2460, Cisco and Nokia, 1998.

[17] Wolfgang Hoeg and Thomas Lauterbach. *Digital Audio Broadcasting*. Wiley, 2009.

[18] Wolfgang Hoeg and Thomas Lauterbach. Google Books Digital Audio Broadcasting, June 2016. URL `https://books.google.ch/books/about/Digital_Audio_Broadcasting.html?id=T0KSa6w3qH4C&redir_esc=y`. [Online; last accessed 16-June-2016].

[19] Frauenhofer IIS. Journaline, June 2016. URL http://www.iis.fraunhofer.de/de/ff/amm/prod/digirundfunk/digirundf/journaline.html.

[20] Information Sciences Institute. Internet Protocol. RFC 791, Information Sciences Institute University of Southern California, 1981.

[21] Information Sciences Institute. Transmission Control Protocol. RFC 793, Information Sciences Institute University of Southern California, 1981.

[22] National Instruments. What is i/q data?, March 2016. URL http://www.ni.com/tutorial/4805/en/. [Online; last accessed 16-June-2016].

[23] Mikael Q Kuisma. I/q data for dummies, 2016. URL http://whiteboard.ping.se/SDR/IQ. [Online; last accessed 16-June-2016].

[24] MIT. Lecture 8 – convolutional coding. Lecture notes, Massachusetts Institute of Technology, 2010.

[25] MIT. Lecture 9 – viterbi decoding of convolutional codes. Lecture notes, Massachusetts Institute of Technology, 2010.

[26] Jeffrey Mogul. Broadcasting Internet Datagrams. Rfc, Computer Science Department Stanford University, 1984.

[27] Paila, et al. FLUTE - File Delivery over Unidirectional Transport. RFC 6726, IETF, Nokia, Qualcomm Technologies, Inc., 2012.

[28] Ian Poole. What is psk, phase shift keying. URL http://www.radio-electronics.com/info/rf-technology-design/pm-phase-modulation/what-is-psk-phase-shift-keying-tutorial.php. [Online; last accessed 16-June-2016].

[29] Jon Postel. User Datagram Protocol. RFC 768, Information Sciences Institute University of Southern California, 1980.

[30] Louis Litwin , Michael Pugel. The principles of ofdm. [Online; last accessed 16-June-2016], April 2016. URL http://research.microsoft.com/en-us/um/people/pcosta/cn_slides/ofdm.pdf.

[31] roklobsta. History and discovery of rtlsdr, April 2016. URL http://rtlsdr.org/#history_and_discovery_of_rtlsdr. [Online; last accessed 16-June-2016].

[32] Wikipedia. In-phase and quadrature components — wikipedia, the free encyclopedia, 2015. URL https://en.wikipedia.org/w/index.php?title=In-phase_and_quadrature_components&oldid=695796542. [Online; last accessed 16-June-2016].

[33] Wikipedia. Advanced audio coding — wikipedia, the free encyclopedia, May 2016. URL https://en.wikipedia.org/w/index.php?title=Advanced_Audio_Coding&oldid=721304877. [Online; last accessed 16-June-2016].

[34] Wikipedia. Kernel — wikipedia, the free encyclopedia, May 2016. URL https://en.wikipedia.org/w/index.php?title=Kernel_(operating_system)&oldid=723188006. [Online; last accessed 16-June-2016].

[35] Wikipedia. Multimedia broadcast multicast service — wikipedia, the free encyclopedia, May 2016. URL https://en.wikipedia.org/w/index.php?title=Multimedia_Broadcast_Multicast_Service&oldid=720525901. [Online; last accessed 16-June-2016].

[36] Wikipedia. Mpeg-2 — wikipedia, the free encyclopedia, May 2016. URL https://en.wikipedia.org/w/index.php?title=MPEG-2&oldid=722085203. [Online; last accessed 16-June-2016].

[37] Wikipedia. Raspberry pi — wikipedia, the free encyclopedia, May 2016. URL https://en.wikipedia.org/w/index.php?title=Raspberry_Pi&oldid=723063614. [Online; last accessed 16-June-2016].

[38] Wikipedia. Orthogonal frequency-division multiplexing — wikipedia, the free encyclopedia, May 2016. URL https://en.wikipedia.org/w/index.php?title=Orthogonal_frequency-division_multiplexing&oldid=719531917. [Online; last accessed 16-June-2016].

[39] Wikipedia. Phase-shift keying — wikipedia, the free encyclopedia, May 2016. URL https://en.wikipedia.org/w/index.php?title=Phase-shift_keying&oldid=722307327. [Online; last accessed 16-June-2016].

[40] World DAB. Standardization, May 2016. URL https://www.worlddab.org/about-worlddab/committees-task-forces. [Online; last accessed 16-June-2016].

[41] World DAB. Glossary, May 2016. URL https://www.worlddab.org/glossary. [Online; last accessed 16-June-2016].