DIGITAL AUDIO BROADCASTING

JAN VAN KATWIJK

Lazy Chair Computing
F. W. van Stoetwegensingel 9
2642 BX Pijnacker, The Netherlands

(received: 5 June 2021; revised: 22 August 2021;
accepted: 18 August 2021; published online: 30 November 2021)

Abstract: Processing digital radio - either on the transmitter or the receiver side - requires a significant amount of digital processing. A receiver for digital radio usually consists of two parts, a "hardware" part, handling the conversion from an analog antenna signal to a stream samples, and a "software part", a decoder, decoding the samples and generating audio, text, images and video. In this paper some aspects of the design and implementation of Qt-DAB, a software decoder for Digital Audio Broadcasting (DAB and DAB+), is discussed. The Qt-DAB decoder runs on a variety of hardware platforms, hardware as small as creditcard sized computers as the Raspberry PI 2, and home computers. In the design performance and flexibility were key. The design is such that it is easy to interface to different hardware SDR devices and easy to add new features. While the core of the Qt-DAB software is formed by the signal processing part, interpreting the incoming sample stream and generating audio, text and images, by far the largest part of the software is handling user interaction and user comfort. Qt-DAB provides a large amount of options, options to select a device, to inspect the signal, to store signals, and options to set the configuration. All in all, it shows that about three quarters of the amount of code is involved is the non-signal processing part.

Keywords: DAB, C++, Software Defined Radio, DAB, DAB+, Qt

DOI: https://doi.org/10.34808/tq2021/25.3/c

1. Introduction

Whether we like it or not, digital communication is the future. We all use computers to zoom, our smartphone uses digital communication, and our TV is digital. The (almost nostalgic) era of AM radio transmissions with home-built radios with glooming tubes are gone, and within years most of the FM transmissions will disappear as well and be replaced by digital radio.

In Europe and Australia (Digital Audio Broadcasting (DAB, DAB+) [1] is the system of choice, other countries and other continents use other systems. Some countries use Digital Radio Mondiale (DRM) [2] for transmissions in shortwave, and some are experimenting with DRM+, similar to DRM, but for transmissions

in the FM band. The US has its own system of digital radio, a hybrid form with possibilities for digital and analog signals in the same transmission.

DAB transmissions are in the old TV Band III (app 170 - 230 MHz), DRM is mainly transmitted in shortwave, and DRM+ in the FM band.

In this paper some elements of Qt-DAB, an open source software DAB (DAB+) decoder[1] are discussed. In section 2 a brief introduction to DAB (DAB+) and the underlying OFDM technique is given, in section 3 we present briefly the Qt-DAB decoder, in section 4 we briefly discuss some aspects of its design, in section 5 we discuss synchronization of a DAB transmission in the DAB decoder, using well known techniques, and in section 6 we discuss aspects of the project.

2. OFDM, DAB and DAB+

DAB, Digital Audio Broadcasting, and DAB+ is a form of digital[2] radio developed in the late 90-ies and revised in the first decade of this century. DAB+ differs from DAB by the way the audio is encoded, the term DAB will be used for both.

Other than AM or FM, where a transmission ususally is restricted to deliver a single audio stream, a DAB transmission may contain a number of audio and data services. As an example, the NPO (Dutch public radio) delivers a transmission with 13 services, 12 services of which are audio, one is a data service, transmitting the radio guide.

2.1. OFDM

The underlying technique for transferring bits is OFDM, *Orthogonal Frequency Division Multiplexing*. OFDM is a technique with which large amounts of digital data can be encoded and transmitted over a radio signal. Anything that can be expresses using bits can be transmitted, either audio, video, or whatever data. A DAB audio service often carries - next the the digitized audio content - one or more pictures, Unfortunately, for understanding DAB decoding, a glimpse of OFDM understanding is required.

With OFDM based techniques [4] data is modulated on N carriers, carriers with a minimal distance between their frequencies. The minimal distance is determined by the modulation speed on these carriers. For DAB, carrier distance is 1 kHz.

We consider modulated carriers as a sequence of complex samples, rather than as an analog signal. An Inverse Fast Fourier Transform (IFFT) operation maps a group of values from consecutive carriers from the frequency domain onto the time domain. Feeding these time domain samples subsequently into a Digital-Analog (DA) converter and shifting the frequency of the signal will give the analog DAB signal.

---

1. For this and other software, see [3]

2. Of course the term ”digital” is easily misunderstood, the actual signal that is being transmitted and entering the radio device is analog.

In practice, there are two extensions to this scheme. First of all, the group of carrier values that is input to the IFFT operation is extended at both sides with dummies, null carriers before applying the IFFT operation, second, the resulting segment in the time domain is prefixed with a segment, conisting op copies of the last M values in the segment.

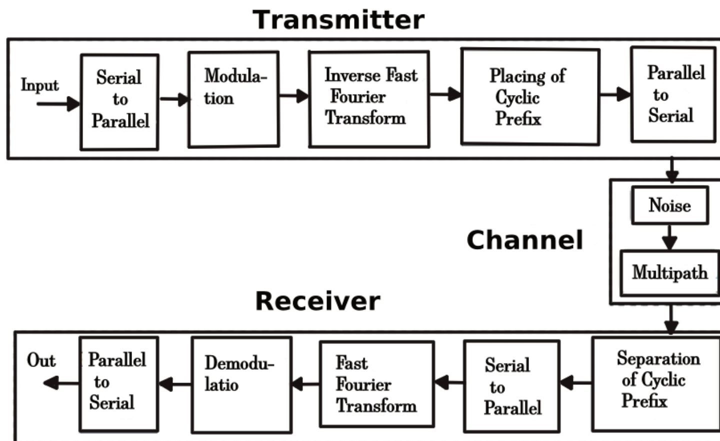

Figure 1. ODFM scheme

2.2. DAB

In DAB the data is encoded in segments of 1536 complex carrier values. Prior to applying an IFFT operation, each segment is extended with 512 null carriers, 256 at each side.

Applying an IFFT operation on such segments, each with 2048 carrier values, leads to segments of 2048 complex samples in the time domain. In the time domain, each such segment is prefixed with a *cyclic prefix* of 504 samples copied from the end of the segment, the *guard*. So, the 1536 complex carrier values used as input in the transmission end up as 2552 complex samples in the time domain.

In DAB these samples are sent through a DA converter - with a rate of 2048000 samples per second - resulting in a signal with an IF of 0 Hz and a bandwidth of app 1536 kHz (the inserted carriers with null values do not contribute to the bandwidth). The signal is mixed with a complex oscillator signal in the range 170 .. 230 MHz, the result is amplified and send to an antenna system.

On the receiver side the process is reversed. First the analog signal arriving through the antenna is mixed using some oscillator and shifted to an IF of 0 Hz, after which an Analog-Digital (AD) converter transforms the analog signal into a sequence of samples - with a rate of 2048000 samples/second - in the time domain. The receiver collects groups of 2552 samples, removes the 504 samples from the cyclic prefix, and feeds the resulting 2048 samples through a Fast Fourier

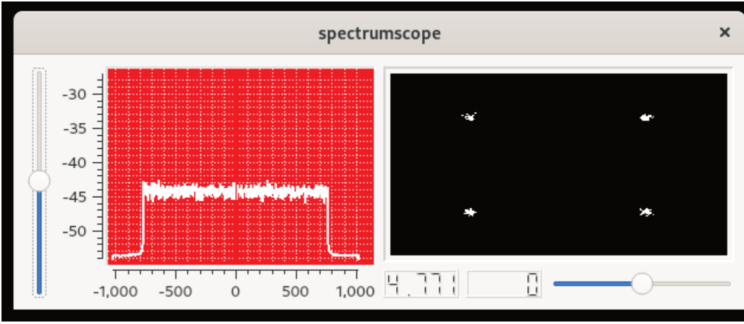

Figure 2. Spectrum and signal constellation

Transform (FFT) processor. From the resulting 2048 carriers values, 1536 carrier values with useful data are extracted.

A DAB transmission is organized in *frames*. In the transmission, a DAB frame takes 199608 samples (again, with a rate of 2048000 samples per second), just over 10 frames per second.

Such a DAB frame starts with 2656 samples *with almost no amplitude*, a so-called null period. Then 76 data blocks follow, each consisting of 2552 samples (504 + 2048). The null period makes it easy to get an idea of where the data of a DAB frame starts. The first data block of the DAB frame contains predefined data, correlating the data found in the incoming samplestream with the predefined values gives the exact position of the first sample of the first data block of a DAB frame.

The next 3 data blocks from which carriers are extracted in the frequency domain contain so-called FIC data (Fast Information Channel). With the decoded FIC data, directory information can be built up describing the content of the decoded data of the remaining blocks (the MSC, Master Service Channel).

In DAB the complex values in the carriers are used to encode the bits. The carrier values in the first data block of a DAB frame are used as reference for the bits encoded in the second data block. In general, decoding is in two steps. If $carrier_{i,j}$ represents the value of carrier j in data block i, we extract two bits (b_j, b_{1536+j}) from first computing $phase = carrier_{i,j} * conj(carrier_{i-1,j})$ and defining $b_j = real(phase)$ and $b_{1536+j} = imag(phase)$. In reality the values for the bits are scaled in the range -127 ... 127, just for practical reasons. In Figure 2 the spectrum and the constellation of a synthetic DAB signal is shown.

Furthermore, at the transmitter side, the carriers were interleaved, to increase the possibility of recovering from spurious frequency errors during transmission. Of course, the de-interleaving takes place before extracting the bits.

The OFDM technique used by DAB makes it possible for a receiver to distinguish between interfering data streams with the same content from different transmitters (and of course due to reflection). This makes it possible to transmit DAB through a so-called SFN [5] a *single frequency network*, a group of transmitters transmitting the same signal. The obvious advantage is that rather

than a single high power transmitter a number of transmitters with a much lower transmission power can be used.

Each of the transmitters in such an SFN can add some data to the *null* period, and encoding identifying the transmitter, the so-called Transmitter Identicaiton Information (TII) data.

Most current transmissions use DAB+ rather than DAB. While the underlying transmission technique is the same, DAB+ uses a different approach to store the (audio) data in the MSC. While in DAB audio is encoded in MP2[3], audio in DAB+ services is encoded in He-AAC, in Qt-DAB decoded by libfaad [7]. A modern DAB receiver is therefore able to decode DAB and DAB+ audio, a first generation receiver cannot decode DAB+ services.

To add redundancy, at the transmitter side, the bits were fed through a convolutional *encoder* before being combined into a complex number used as input for the transmission process.

Due to channel effects, the phases of the carriers resulting from the FFT operation in the decoding process most likely differ from the phases that were presented as input. The resulting values were therefore decoded to *soft bits*, i.e. values in the range -127 .. 127. These soft bits are fed into a convolutional decoder (a Viterby decoder [8]) to obtain the final bits whenever needed. These resulting bits are then used for building the directory structure if coming from the first few blocks, or used for data or audio. CRC checks are executed to validate the resulting data.

2.3. DAB+

Often the term DAB is used when DAB+ is meant. DAB itself dates from the late 90-ies, DAB+, a revised version from the first decade of this century. While the underlying mechanisms for DAB and DAB+ are the same, there are fundamental differences in the encoding of the audio data.

In DAB the MP2 encoded audio segments were stored directly in subsequent blocks in DAB frames, one segment per frame. The audio handling was simple, extract the right segments from the frame and feed them into the MP2 decoder.

In DAB+ audio is encoded in He-AAC. Multiple AAC franes are stored in so-called *superframes*, where a superframe was built from 5 segments in consecutive DAB frames. The data for building up a superframe is - in segments of 120 bytes - passed through a Reed-Solomon decoder [9], that could detect and repair up tp 5 errors and delivers - if the number of errors not greater than 5 - 110 error free bytes[4].

The superframe, when built, contains a description of where to find the - other than the MP2 segments - differently sized HE-AAC segments.

3. The MP2 decoder used was written by [6]

4. This is of course not completely true, the parity bytes themselves could contain errors, therefore an additional CRC check is performed on the HE-AAC segments

3. Qt-DAB

Qt-DAB is a program for decoding terrestrial DAB signals. The software is written in C++ and uses the Qt framework [10], hence the name. The program takes samples from an SDR input device or a file, and generates PCM samples for the computer's soundcard or for a small server, for sound over IP.

Common devices that are supported are DABsticks, AIRspy devices, SDR-play devices HACKrf devices, LIME devices and Adalm Pluto devices. In a few experiments, support software was written for some old devices only supporting frequencies up to 30 or 60 Mhz, using subsampling.

The software is developed under Linux and cross compiled for Windows. It runs on a PC under Linux, under Windows and on some ARM based boards, such as the Raspberry Pi 2, 3 and 4.

Qt-DAB is developed with the idea that the user is in full control, i.e. the GUI of Qt-DAB has an abundant amount of controls and displays. The sourcetree for Qt-DAB contains a second version, *dabMini*, a version with an minimal GUI, used for just for listening to a service.

Further family members are a so-called *DAB library*, a library providing all functionality for decoding DAB, and *Terminal-DAB*, a DAB decoder for running DAB without a GUI and a few scanner programs, for scanning the channels in the band.

Most members of the family support the same variety of input devices, varying from a simple DABstick to the more elaborate SDRplay devices.

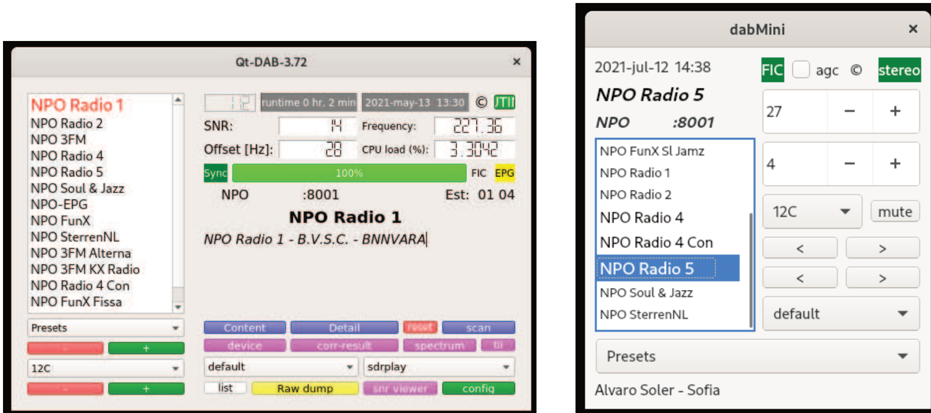

Figure 3. Two DAB decoders

Qt-DAB always shows a main widget (Figure 3) and - under user control - up to 7 additional widgets for control and for displaying information can be made visible. The main widget contains general information, obviously the list of services in the transmission, buttons for making additional displays and widgets visible and buttons for device and audio selection. It also contains number displays

for showing the frequency of the selected DAB channel, the total CPU load, the SNR of the signal and the detected frequency offset of the incoming signal.

The widget further shows a label, showing whether or not time synchronization is successfull (dark green), and a progress indicator (green) telling the quality of the decoding. Below these indicators, the ensemble is shown, together with the transmitter identification, the name of the selected service and the so-called dynamic label, the text transmitted with the audio.

An esthetical extension was coloring of buttons and displays. Based on user requests an extremely flexible scheme was chosen, after all, color selection is personal. The solution was that a user can dynamically set the colors of buttons and displays with a few simple mouse clicks to his or hers likings. These settings are - obviously - maintained between program invocations.

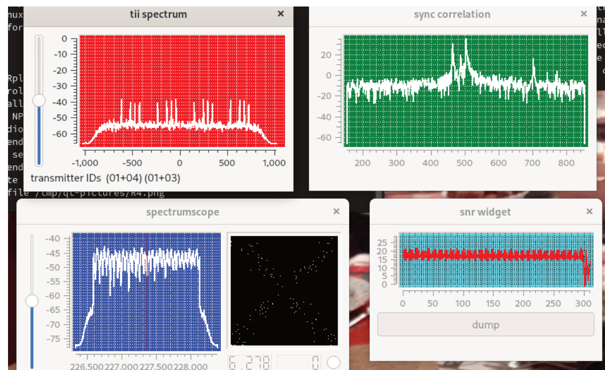

Figure 4. Qt-DAB: some other widgets

Some of the other widgets are shown in Figure 4.

- at the top left, the spectrum of the null period is shown. The null period contains the encoding of the transmitter identification information, the TII. Apparently, data from transmitters (1, 4) and (1, 3) is received, which makes sense, since these two transmitters are within 20 Km of my home.
- at the top right, the correlation is shown. Correlation is used to synchronize the receiver with the strongest signal. The picture shows 4 peaks, the two strongest ones show apparently the correlation with the signals of the transmitters (1, 4) and (1, 3).
- at the bottom left, the spectrum is shown, it is easy to see that the width is app 1.5 MHz. Furthermore, the constellation of the received data is displayed in that widget. It is clear from the picture that the signal is affected by interference from the other transmitters. The two numbers on the widget give a quality indication (the 6.2...) on a scale to 10, resp an indication of the clock error (0), i.e. the number of samples off in 10 frames of 199608 samples.
- at the bottom right, the development of the SNR over time is shown.

Other widgets that can be made visible are (a) a widget for device control, (b) a widget showing technical data of the selected audio service (if any), and (c) a configuration widget for some additional settings.

4. Qt-DAB architecture and design issues

4.1. Qt-DAB architecture

Processing DAB is basically transforming an inputstream of 2048000 samples per second into an outputstream (PCM samples) of 48000 samples per second. Processing involves (a.o.) executing over 800 FFT operations per second on segments of 2048 complex samples, and performing Viterbi decoding on the data in the FIC blocks and on selected services, as well as Reed-Solomon decoding on selected DAB+ services, all operations that are quite resource intensive.

While modern PC's and laptops have more than enough CPU power to run a decoder (see Figure 3, the picture shows a processor load of a few percent), running the software in a *single* thread on something like a Raspberry Pi 2 overloads the processor core that is being used.

The different program elements are therefore mapped on different objects, objects that *can* run each on its own thread. In configuring the software a choice can be made to have each such object run as task in its own thread or not. Maximal concurrency in the Qt-DAB implementation shows an average load of 50 to 60 percent on the 4 cores of an Raspberry Pi 2 and well below 50 percent on an Raspberry Pi 3.

Elements to be considered are:

- *Input handling*. In order to handle different input devices, a simple interface had to be defined, support for a device requires developing a driver program implementing the interface. The interface is implemented as a class with a handful of virtual functions. Only three functions are essential for the decoder: startChannel on a given frequency, *stopChannel* and *getSamples*. Setting or altering gain is done through the device control widget. In configuring a user can select which devices to include.
- *OFDM handling* takes the input samples from the interface, takes care of *time* and *frequency* synchronization (see section 5) and passes on the datablocks from the DAB frame to either the FIC handler or the backend handler. What should be realized is that the final check whether or not the synchronization is OK, is that the FIC handling can decipher the FIC blocks and show names and attributes of services. FIC handling therefore is intimately coupled to OFDM handling and forms, with other functions with OFDM handling a single object.
- *backend controller* is the interface between the OFDM handling and the actual backends. Partitioning FFT operations over more than a single thread was the objective, therefore the backend controller collects the *time domain samples* for the datablocks in the MSC, applies the FFT

transformation and stores the result into a buffer. The backend controller provides interface functions, to be used by the GUI handler, for creating and stopping a service by allocating (or deallocating) a backend. The backend controller maintains a set of allocated (i.e. active) backends, its implementation does not limit the number of active backends. After reading the data of a DAB frame, the backend controller passes selected segments of the MSC data to the appropriate backend.

- *backends* For the processing of the data for each selected service, a separate *backend* object will be allocated. Two category backends are the *audio* and the *data* backends. The first category has two members, a backend for MP2 (classical DAB) and a backend for He-AAC (DAB+). The second category supports Multimedia Object Transfer (MOT), Internet Protocol (IP) handling, Electronix Program Guide (EPG) and Transport Protocol Expert Group (TPEG) handling and (untested) journaline handling. Whether a backend is implemented as running in its own thread or in the caller's thread is element of the configuration.
- *GUI and controller*. The GUI handling is implemented using Qt. It is responsible for handling the user interaction and - thereby - displaying information on a variety of widgets.

4.2. Flexibility of the design, some examples

The current version of Qt-DAB differs - especially wrt the GUI - considerably from earlier versions, and - most likely - future versions will differ from the current one. From the very start, the design has been such that changes and extensions could be made easily. Two extensions are mentioned here.

4.2.1. Presets

One of the additions in a rather late stage of the project was the addition of *presets*. In an earlier light-weight variant the choice was made to scan - on start-up - a (user-indicated) subset of channels for building a list of reachable services. Most users do not want such a long list but want to be able to specify some services as *presets* for easy access. While selecting a preset service within the *currently selected* channel is fairly trivial, it is slightly more complex if the service is located in a different channel. Then a whole sequence of operations has to be performed, stopping the channel, changing the frequency, waiting to get data from that channel, and finally, when the descriptive data for the requested service is available, selecting the service. Having defined some basic operations in the control part, such as *stopService*, *stopChannel*, *startChannel*, etc, made that a relatively simple addition.

4.2.2. Continuous scanning

One option - not mentioned so far - is that Qt-DAB provides a *scanning* function. People - especially DX-ers - like to continuously scan through the band and see what they can receive on the different channels. The result of the scan,

the description of what is detected in the different channels, the time and the SNR, is stored as text file, readable by a spreadsheet program such as LibreCalc. Of course, it is often known beforehand that some channels will definitely not contain any form of DAB signal, so an option was created to create so-called *skip* files, files with a description which channels to use and which to skip. A second point, noted by some users, was that during such a scan, different channels needed different gain settings, also dependent on the connected device, so an extension was made to maintain for each channel, for each device, the gain settings between program invocations.

5. Synchronization issues

Looking at the structure of a DAB decoder such as Qt-DAB, it becomes obvious that as soon as the incoming samples are mapped upon (soft) bits, further processing requires computing the ”hard” bits using Viterbi decoding and applying consistency checks, but is basically just playing with bits.

The interesting part in the software is the OFDM handling and synchronization. Synchronization implies knowing which sample in the input stream fits in which position in the DAB frame that is being read, but synchronization also has to deal also with correcting a frequency offset, if any. Recall that in the decoding process a translation from samples in the time domain to samples in the frequency domain takes place. An offset in the ocillator frequency of the SDR device may have a disastrous result, a carrier value in the result of the FFT, seen at position i should have appeared on position j. Of course, decoding then is impossible.

Note that for simple devices, such as a DABstick, the frequency offset of the oscillator in the device can be tens of kHz (note we are talking about frequencies around 200 MHz), while for more advanced ones, e.g. the various SDRplay devices, offsets of only a few Hz are measured.

5.1. Time synchronization

DAB provides us with a *null* period of 2656 samples as start of a DAB frame, the null period is helpful in detecting the start of the data blocks of the DABframe.

On starting up a channel, we just look at the incoming samples for the null period. Once found, the data in the first data block is correlated with predefined data to identify the sample in the inputstream that is the first sample in the DAB frame.

The time synchronization takes the following steps:

- Compute some moving average amplitude value for all incoming samples, i.e. $L_g = 0.00001 * abs(sample) + 0.99999 * L_g$, and ensure that prior to taking the next steps, at least 100000 samples were read.
- Compute for each incoming sample S_i for a value N of app 50 $L_l = (\sum_{j=i-N+1}^{i} abs(S_j))/N$, i.e. the average value over the last N incoming samples.

- As soon as $L_l < 0.5 * L_g$ it is reasonable to assume that we have the start of the null period detected (of course, if for a long time this relation is not seen, it is most likely not a DAB data stream);
- Continue to compute L_l and L_g until either
  - we had well more than 2656 samples (i.e. the length of the null period), in which case there was probably no DAB signal, or
  - we found that $L_l >= 0.8 * L_g$, in which case it seems we reached the end of the null period.
- In the second case we then collect 2048 samples in a vector V, i.e. data of the first data block of the DAB frame, and we compute the correlation vector $V = IFFT(FFT(V) * conj(P))$ where P is a vector with predefined values, given by the DAB standard. We look for $max(V[i])$ as the index in the vector V of the start of the first data block.

Of course, computing a moving average for each subsequent DAB frame, while we know we are synced, is not needed. We just process the data in the DAB frame, set the input pointer 199608 samples further and restart for the next round with the correlation, and skip the detection of the null period.

5.2. Frequency correction

Offsets in the oscillator frequency that are small, up to half the carrier distance, i.e., for DAB 500 Hz, can be corrected while decoding continues. For larger offsets there is a serious problem, the output value at is actually data that should be the output value for a different carrier. The result is disastrous, no decoding is possible.

Therefore we distinguish between *coarse correction* and *fine correction*. Note that - as mentioned earlier - simple devices like RT2832 based DABsticks show frequency offsets of tens of kHz, so a decent approach to coarse correction is required.

5.2.1. Determining coarse frequency offset

Note that in many OFDM based applications datablocks in the frequency domain contain pilot carriers with predefined phases and increased amplitudes. These pilots can be used to reconstruct the originally transmitted data, and are very useful as markers to detect and compute frequency offsets. In DAB, the first datablock contains predefined data, used - as mentioned earlier - to synchronize and as a basis for decoding the next block.

The best results with detecting a coarse frequency offset were onbtained by looking at the correlation of the phase differences between successive carriers over a region of - in our case app 40 carriers -

$$\sum_{i=a}^{b}\sum_{j=c}^{d}(refC_i * conj(refC_{i+1}) * conj(testC_{i+j} * conj(testC_{i+1+j}) \qquad (1)$$

is maximal where $refC$ denotes the (complex) carrier values as defined in the DAB standard, and $testC$ the value in the first data block after transforming it into the frequency domain.

The ultimate test is of course seeing that decoding is possible, which obviously shows whenever a list of services in the transmission is detected.

5.2.2. Determining fine frequency offset

The fine frequency offset, i.e. for DAB between -500 and 500 Hz, may be the result of channel conditions and may very over time. The offset can be computed and - together with the value for the coarse frequency offset, used in correcting the tuned frequency.

Recall that a *guard* was introduced in the time domain samples. The guard, with a length Tg, is a replica of the last Tg samples of the time domain symbol. As well known (see e.g. [11]), a frequency offset causes the phases of the elements in the original time domain symbol to differ from the corresponding elements in the guard. The difference can be used to estimate the frequency offset. So, if we assume that the original time domain symbol has a length Tu, we can compute for datablock j

$$X_j = \sum_{i=0}^{Tg} element_{j,i} * conj(element_{j,Tu+i}) \tag{2}$$

and estimate the offset by computing

$$offset = arg(X_j)/(2*\pi)*samplerate/Tu \tag{3}$$

where $samplerate/Tu$ indicates the frequency difference between successive caarriers.

Of course, it is better to just average over more than one time domain data block:

$$offset = arg(\sum_{j=0}^{K}\sum_{i=0}^{Tg} element_{j,i} * conj(element_{j,Tu+i})) * samplerate/Tu \tag{4}$$

5.2.3. Applying correction

Having computed a frequency error offset, a correction can be made. While it seems obvious to signal the radio device to alter the frequency, in Qt-DAB another approach is taken.

Qt-DAB abstracts from the attached device by providing an interface, and pulling samples from the interface when needed. It is therefore - for Qt-DAB - not clear how many samples there are already in the buffer whenever a signal for a frequency update arrives.

The approach chosen for Qt-DAB is to apply frequency corrections per DAB frame, using a software local oscillator, as soon as a sample is entering the OFDM handling, it is shifted with the frequency offset computed. One advantage is of course that samples being read from a file also can be corrected in frequency.

6. In lieu of conclusion

Qt-DAB is an open source project, one of the advantages is that there are absolutely no deadlines, and parts can be - and are - rewritten many times, applying newer ideas and better algorithms or just for experimenting. Most of the experiments had to do with the synchronization and efficiency. As an example, an alternative way of computing the fine frequency offset is by looking at phase differences in the FFT output [11] rather than computing the phase differences between samples in the time domain, just to verify that the two approaches gave (roughly) the same results.

Other experiments dealt with looking for a reasonable partitioning of functionality over objects. Running these objects in their own thread ensures that - even with almost continuously expanding functionality - the software runs smoothly on a simple device such as an Raspberry Pi 2.

One of the recent experiments that ended as a configuration option is with the Adalm Pluto device. The Pluto has - next to a receiver - also transmit capabilities. Qt-DAB was extended such that the audio output of a selected audio service, augmented with the text shown with the audio (the dynamic label), is transformed into an FM stereo signal with RDS and transmitted on a user selected frequency.

Having made the software publicly available resulted in quite some feedback. Some feedback was as can be expected: ”it does not work” or ”it does not compile”. There was, however, also a lot of really helpful technical feedback, instructions for simple things like making widgets resizeable, interpreting MP2 data, proper handling of AAC segments (the encoding of the DAB+ audio), introduction of presets, even recently the suggestion the use of an address sanitizer, all these things led to improving the Qt-DAB software.

The resulting program has a reasonable size, app 40000 lines of code. Of course, within these 40000 lines, *functional blocks* can be identified as was mentioned in section 4. The OFDM handling deals with reading samples, synchronzing, building op DAB frames and generating (soft) bits. The FIC handler then is responsible for setting up and maintaining a structure that contains the descriptions of the services in the current transmission but for technical reasons should be combined with OFDM handling in a single object. The backend dispatcher collects the MSC data from the OFDM handler and passes the appropriate segments on to the various selected backends. Backends themselves are backends for audio decoding (MP2 and AAC), and backends for data decoding, e.g. MOT, EPG, IP, TPG handling etc. The approximate sizes are given in Table 1.

The large size of *device support* follows from supporting about 8 different device types, and three different programs for (different types of) file input.

Of course, Qt libraries, libraries for FFT handling, AAC decoding, support libraries for devices, and libsndfile and libsamplerate are not included in these

Table 1. Size of components

component	size
device support	13500
ofdm and FIC handling	5500
backend dispatching	1000
audio decoding	2500
data decoding	9000
control and display	6000
various support	2500

figures since libraries are used in their binary form and their sizes not includes in the figures given here.

References

[1] *Radio Broadcasting Systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers*, ETSI EN 300 401
[2] *Digital Radio Mondiale (DRM); System Specification*, ETSI ES 201 980
[3] Katwijk van J *The Qt-DAB decoder*, URL: `https://github.com/JvanKatwijk/qt-dab`
[4] *Orthogonal frequency-division multiplexing*, URL: `https://en.wikipedia.org/wiki/Orthogonal_frequency-division_multiplexing`
[5] *Single Frequency Network*, URL: `https://en.wikipedia.org/wiki/Single-frequency_network`
[6] Fiedler J M, KJMP2 – a minimal MPEG-1/2 Audio Layer II decoder library, Copyright 2006 - 2013
[7] *Freeware Advanced Audio (AAC) Decoder, FAAD2*, URL: `https://ecsoft.org/faad2`
[8] *Viterbi algorithm*, URL: `https://en.wikipedia.org/wiki/Viterbi_algorithm`
[9] Clarke P K C July 2002 *Reed-Solomon error correction*, BBC R & D White Paper, WHP 031
[10] *Qt Software Development Platform*, URL: `https://www.qt.io/`
[11] Tzi-Dar Chiueh and Pei-Yun Tsai 2007 *OFDM baseband Receiver Design for Wireless Communications*, John Wiley & Sons (Asia) Pte Ltd
[12] *Open source Spiral System*, URL: `www.spiral.net/codegenerator.html`
[13] *Linux apps that run everywhere*, URL: `https://appimage.org`

Jan van Katwijk is a retired professor in Software Engineering at Delft University of Technology. He received his MSc in Mathematics in 1971 and a PhD in Computer Science in 1987 from Delft University of Technology. He worked at Delft University from 1971 till 2009 as assistent, associate resp. full professor. From 1998 till 2007 he acted as dean of the faculty of Electrical Engineering, Mathematics and Computer Science at the Delft University. His research interests shifted from compiler design and construction to software specification, construction, engineering and software quality. He was a member of IFIP WG 2.4 and served on ISO/TC97/SC22. After his retirement he got interested in software aspects and software engineering issues for Software Defined Radio and he developed software for a variety of SDR programs.