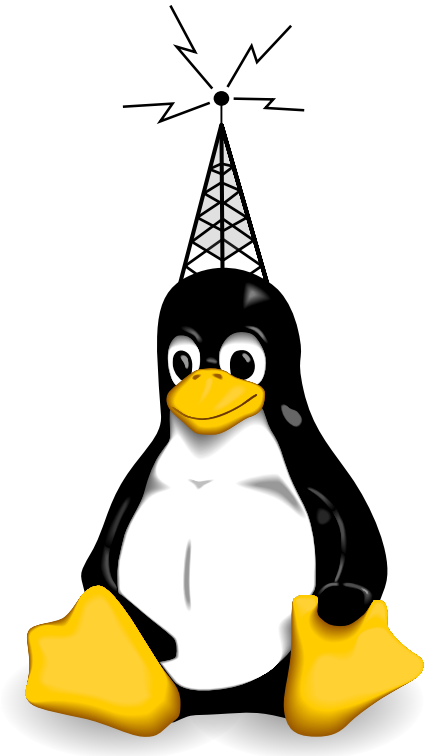
ODR-mmbTools
Open-Source Software-Defined DAB$^+$ Tools

Project Documentation

Opendigitalradio
http://opendigitalradio.org
2014–2017



Contents

Acronyms

1PPS One pulse per second

CIF Common Interleaved Frame

CRC Communications Research Centre Canada

DAB Digital Audio Broadcasting

DMB Digital Multimedia Broadcasting

ETI Ensemble Transport Interface

ETSI European Telecommunications Standards Institute

FIC Fast Information Channel

HE-AAC High Efficiency Advanced Audio Codec

mmbTools Mobile Multimedia Broadcasting Tools

MNSC Multiplex Network Signalling Channel

NTP Network Time Protocol

OCXO Oven-Controlled Crystal Oscillator

OFDM Orthogonal Frequency-Division Multiplexing

PRBS Pseudo-Random Bit Sequence

SFN Single-Frequency Network

TCXO Temperature-Compensated Crystal Oscillator

TIST Timestamp field in the ETI frame

TM Transmission Mode

UHD USRP Hardware Driver

USRP Universal Software-Radio Peripheral

1 Introduction

This is the official documentation for the ODR-mmbTools. These tools can be used to experiment with DAB modulation, learn the techniques behind it and setup a DAB or DAB$^+$ transmitter.

This documentation assumes that you are already familiar with base concepts of the DAB system. To get started with the ODR-mmbTools, understanding how the DAB transmission chain is structured is a prerequisite. The "DAB Bible" by Hoeg and Lauterbach [4] and the "Guide to DAB standards" from the ETSI [1] can be used as a starting point.

In this document, the terms "DAB" and "DAB$^+$" are used somewhat interchangeably, since many parts of the transmission chain are identical between the two variants. In most cases, "DAB" will be used, and "DAB$^+$" when talking about specific details about the newer version of the standard.

2 Purpose

The different programs that are part of the ODR-mmbTools each have their own documentation regarding command-line options and configuration settings, and the opendigitalradio.org wiki[1] contains many explanations and pointers, but there is no single source of documentation available for the whole tool-set.

This document aims to solve this, by first outlining general concepts, presenting different usage scenarios and detailing a complete transmission setup.

With this document in hand, you should be able to understand all elements composing a ODR-mmbTools transmission chain, and how to set one up.

3 Presentation of the Tools

3.1 Origins

Before we begin with technical details, first a word about the history of the mmbTools. In 2002, Communications Research Centre Canada[2] started developing a DAB multiplexer. This effort evolved through the years, and was published in September 2009 as CRC-DabMux under the GPL open-source licence.

CRC also developed a DAB modulator, called CRC-DABMOD, which could create baseband I/Q samples from an ETI file. This I/Q data could then be set to a hardware device using another tool. For the Ettus USRPs, a "wave player" script was necessary to interface to GNURadio. Only DAB Transmission Mode 2 was supported. CRC-DABMOD was also released under the GPL in early 2010.

As encoders, toolame could be used for DAB, and CRC developed a closed-source CRC-DABPLUS DAB$^+$ encoder.

These three CRC- tools, and some additional services available on the now unreachable website[3] `http://mmbtools.crc.ca` were part of the CRC-mmbTools. These tools made it possible to set up the first DAB transmission experiments.

[1] `http://opendigitalradio.org`
[2] `http://crc.ca`
[3] There are some snapshots of the website available on `http://archive.org`.

In 2012, these tools received experimental support for single-frequency networks, a functionality that has been developed by Matthias P. Brändli during his Master's thesis[4]. Because SFNs are mainly used in TM 1, CRC subsequently released a patch to CRC-DABMOD that enabled all four transmission modes.

At that point, involvement from CRC started to decline. The SFN patch was finally never included in the CRC-mmbTools, and as time passed by, the de-facto fork on `http://mpb.li` was receiving more and more features. Having two different programs with the same name made things complicated, and the tools were officially forked with the approval of CRC in February 2014, and given the new name ODR-mmbTools. They are now developed by the Opendigitalradio association.

In April 2014, the official CRC-mmbTools website went offline, and it has become very difficult, if not impossible to acquire licences for the CRC-DABPLUS encoder. Luckily there is an open-source replacement available, which was part of Google's Android sources. This encoder has been extended with the necessary DAB$^+$-specific requirements (960-transform, error correction, framing, etc.), and now exists under the name fdk-aac. The encoder ODR-AudioEnc can use this library to encoder DAB$^+$.

3.2 Included Tools

The ODR-mmbTools are composed of several software projects: ODR-DabMux, ODR-DabMod, ODR-AudioEnc, ODR-PadEnc, and other scripts, bits and pieces that are useful for the setup of a transmission chain.

3.2.1 ODR-DabMux

ODR-DabMux implements a DAB multiplexer that combines all audio and data inputs into an ETI output. It can be used off-line (i.e. not real-time) to generate ETI data for later processing, or in a real-time streaming scenario (e.g. in a transmitter).

It can read input audio or data from files (".mp2" for DAB, ".dabp" for DAB$^+$), FIFOs (also called "named pipes") or a network connection. The network connection can use UDP or ZeroMQ. The CURVE authentication mechanism from ZeroMQ can also be used to authenticate the encoder, in order to avoid that a third-party can disrupt or hijack a programme.

The ensemble configuration can be specified on the command line using the options described in the manpage, or using a configuration file. The command line options are kept to be compatible with CRC-DABMUX, but using the configuration file is preferred, because it supports more options.

3.2.2 ODR-DabMod

ODR-DabMod is a software-defined DAB modulator that receives or reads ETI, and generates modulated I/Q data usable for transmission.

This I/Q data which is encoded as complex floats (32bits per complex sample) can be written to a file or pipe, or sent to a USRP device using the integrated

[4] The corresponding report is available at `http://mpb.li/report.pdf`

UHD output. Other SDR platforms can be used if they are able to accept the I/Q data. The output of the modulator can also be used in GNURadio if format conversion or graphical analysis (spectrum) is to be done.

3.2.3 ODR-AudioEnc

The ODR-AudioEnc encoder can be used to encode for DAB and DAB$^+$. It includes a toolame-based MPEG encoder, and uses the fdk-aac library as an external dependency to encode DAB$^+$.

The integrated TooLAME library is a MPEG-1 Layer II audio encoder that is used to encode audio for the DAB standard. The original project has been unmaintained since 2003, but the twolame fork that pursues the development removed the DAB framing. Because of this, twolame is not suitable for DAB.

The necessary framing and error-correction that DAB$^+$ mandates, the PAD insertion, the ZeroMQ output and the ALSA input were then added by different parties.

3.2.4 ODR-PadEnc

This encoder is able to generate programme associated data that can be injected into ODR-AudioEnc. It supports DLS, reading from a file, and MOT Slideshow, where the slides are read from a folder.

4 Interfacing the Tools

4.1 Files

The first versions of these tools used files and pipes to exchange data. For offline generation of a multiplex or a modulated I/Q, it is possible to generate all files separately, one after the other.

Here is an example to generate a two-minute ETI file for a multiplex containing two programmes:

- one DAB programme at 128kbps
- one DAB$^+$ programme at 88kbps

We assume that the audio data for the two programmes is located in uncompressed 48kHz WAV in the files `prog1.wav` and `prog2.wav`. The first step is to encode the audio. The DAB programme is encoded to `prog1.mp2` using:

```
odr-audioenc --dab -b 128 -i prog1.wav -o prog1.mp2
```

The DAB+ programme is encoded to `prog2.dabp`. The extension `.dabp` is arbitrary, but since the framing is not the same as for other AAC encoded audio, it makes sense to use a special extension. The command is:

```
odr-audioenc -i prog2.wav -b 88 -o prog2.dabp
```

These resulting files can then be used with ODR-DabMux to create an ETI file. ODR-DabMux supports many options, which makes it much more practical to set the configuration using a file than using very long command lines. Here is a short file that can be used for the example, which will be saved as `2programmes.mux`:

```
general {
    dabmode 1
    nbframes 5000
}
remotecontrol { telnetport 0 }
ensemble {
    id 0x4fff
    ecc 0xec ; Extended Country Code

    local-time-offset auto
    international-table 1
    label "mmbtools"
    shortlabel "mmbtools"
}
services {
    srv-p1 { label "Prog1" }
    srv-p2 { label "Prog2" }
}
subchannels {
    sub-p1 {
        ; MPEG
```

```
22          type audio
23          inputfile "prog1.mp2"
24          bitrate 128
25          id 10
26          protection 5
27      }
28      sub-p2 {
29          type dabplus
30          inputfile "prog2.dabp"
31          bitrate 88
32          id 1
33          protection 1
34      }
35  }
36  components {
37      comp-p1 {
38          label Prog1
39          service srv-p1
40          subchannel sub-p1
41      }
42      comp-p2 {
43          label Prog2
44          service srv-p2
45          subchannel sub-p2
46      }
47  }
48  outputs { output1 "file://myfirst.eti?type=raw" }
```

This file defines two components, that each link one service and one subchannel. The IDs and different protection settings are also defined. The bitrate defined in each subchannel must correspond to the bitrate set at the encoder.

The duration of the ETI file is limited by the `nbframes 5000` setting. Each frame corresponds to 24 ms, and therefore $120/0.024 = 5000$ frames are needed for 120 seconds.

The output is written to the file `myfirst.eti` in the ETI(NI) format. Please see Appendix A for more options.

To run the multiplexer with this configuration, run:

```
1 odr-dabmux 2programmes.mux
```

This will generate the file `myfirst.eti`, which will be $5000 * 6144 \approx 30$MB in size.

Congratulations! You have just created your first DAB multiplex! With the configuration file, adding more programmes is easy. More information is available in the `doc/example.mux`

4.2 Over the Network

In a real-time scenario, where the audio sources produce data continuously and the tools have to run at the native rate, it is not possible to use files anymore

to interconnect the tools. For this usage, a network interconnection is available between the tools.

This network connection is based on ZeroMQ, a library that permits the creation of a socket connection with automatic connection management (connection, disconnection, error handling). ZeroMQ uses a TCP/IP connection, and can therefore be used over any kind of IP networks.

This connection makes it possible to put the different tools on different computers, but it is not necessary. It is also possible, and even encouraged to use this interconnection locally on the same machine.

4.2.1 Between Encoder and Multiplexer

Between ODR-AudioEnc and ODR-DabMux, the ZeroMQ connection transmits AAC superframes, with additional metadata that contains the audio level indication for monitoring purposes. The multiplexer cannot easily derive the audio level from the AAC bitstream without decoding it, so it makes more sense to calculate this in the encoder.

On the multiplexer, the subchannel must be configured for ZeroMQ as follows:

```
sub-fb {
    type dabplus
    bitrate 80
    id 24
    protection 3

    inputfile "tcp://*:9001"
    zmq-buffer 40
    zmq-prebuffering 20
}
```

The ZeroMQ input supports several options in addition to the ones of a subchannel that uses a file input. The options are:

- `inputfile`: This defines the interface and port on which to listen for incoming data. It must be of the form `tcp://*:<port>`. Support for the `pgm://` protocol is experimental, please see the `zmq_bind` manpage for more information about the protocols.
- `zmq-buffer`: The ZeroMQ input handles an internal buffer for incoming data. The maximum buffer size is given by this option, the units are AAC frames (24 ms). Therefore, with a value of 40, you will have a buffer of $40 * 24 = 960$ ms. The multiplexer will never buffer more than this value, and will discard data one AAC superframe (5 frames = 100 ms) when the buffer is full.
- `zmq-prebuffering`: When the buffer is empty, the multiplexer waits until this amount of AAC frames are available in the buffer before it starts to consume data.

The goal of having a buffer in the input of the multiplexer is to be able to absorb network latency jitter: Because IP does not guarantee anything about the

latency, some packets will reach the encoder faster than others. The buffer can then be used to avoid disruptions in these cases, and its size should be adapted to the network connection. This has to be done in an empirical way, and is a trade-off between absolute delay and robustness.

If the encoder is running remotely on a machine, encoding from a sound card, it will encode at the rate defined by the sound card clock. This clock will, if no special precautions are taken, be slightly off frequency. The multiplexer however runs on a machine where the system time is synchronised over NTP, and will not show any drift or offset. Two situations can occur:

Either the sound card clock is a bit slow, in which case the ZeroMQ buffer in the multiplexer will fill up to the amount given by `zmq-prebuffering`, and then start streaming data. Because the multiplexer will be a bit faster than the encoder, the amount of buffered data will slowly decrease, until the buffer is empty. Then the multiplexer will enter prebuffering, and wait again until the buffer is full enough. This will create an audible interruption, whose length corresponds to the prebuffering.

Or the sound card clock is a bit fast, and the buffer will be filled up faster than data is consumed by the multiplexer. At some point, the buffer will hit the maximum size, and one superframe will be discarded. This also creates an audible glitch.

Consumer grade sound cards have clocks of varying quality. While these glitches would only occur sporadically for some, bad sound cards can provoke such behaviour in intervals that are not acceptable, e.g. more than once per hour.

Both situations are suboptimal, because they lead to audio glitches, and also degrade the ability to compensate for network latency changes. It is preferable to use the drift compensation feature available in ODR-AudioEnc, which insures that the encoder outputs the AAC bitstream at the nominal rate, aligned to the NTP-synchronised system time, and not to the sound card clock. The sound card clock error is compensated for inside the encoder.

Complete examples of such a setup are given in the scenarios.

4.2.2 Authentication Support

In order to be able to use the Internet as contribution network, some form of protection has to be put in place to make sure the audio data cannot be altered by third parties. Usually, some form of VPN is set up for this case.

Alternatively, the encryption mechanism ZeroMQ offers can also be used. To do this, it is necessary to set up keys and to distribute them to the encoder and the multiplexer.

```
    encryption 1
    secret-key "keys/mux.sec"
    public-key "keys/mux.pub"
    encoder-key "keys/encoder1.pub"
```

Add configuration example

4.2.3 Between Multiplexer and Modulator

The ZeroMQ connection can also be used to connect ODR-DabMux to one or more instances of ODR-DabMod. One ZeroMQ frame contains four ETI frames, which guarantees that the modulator always assembles the transmission frame in a correct way, even in Transmission Mode I, where four ETI frames are used together.

4.3 Pipes

Pipes are an older real-time method to connect several encoders to one multiplexer on the same machine. It uses the same configuration as the file input but instead of using files, FIFOs, also called "named pipes" are created first using `mkfifo`.

This setup is deprecated in favour of the ZeroMQ interface.

5 Usage Scenarios

5.1 Experimentation

5.1.1 Creation of Non-Realtime Multiplex

The creation of a ETI file containing two programmes, one DAB and one DAB$^+$ is covered in section 4.1.

5.1.2 Modulation of ETI for Offline Processing

The ETI file generated before can then be used with ODR-DabMod to generate a file containing I/Q samples. Here, we must chose between using the command line or the configuration file. For a very simple example, using the command line makes sense, but for more advanced features it is preferable to use a configuration file. For illustration, we will present both.

To modulate the file `myfirst.eti` into `myfirst.iq`, with the default options, the command is simply

```
odr-dabmod myfirst.eti -f myfirst.iq -n 1
```

This will create a file containing 32-bit interleaved I/Q at 2048000 samples per second. Where the maximal amplitude is bounded by 1.The transmission mode is defined by the ETI file.

The equivalent configuration file would be

```
[input]
transport=file
source=myfirst.eti

[output]
output=file

[fileoutput]
format=complexf
normalize=1
filename=myfirst.iq
```

This is a very minimal file that defines only the necessary settings equivalent to the above command line options. The configuration file however supports more options that the command line, and becomes easier to manager once the set becomes more complex. It is best to use the example configuration availble in the `doc/` folder.

5.2 Interfacing Hardware Devices

5.2.1 Ettus USRP

ODR-DabMod integrates support for the UHD library that can interface with all USRP devices from Ettus. The following configuration file `mod.ini` illustrates how to send the `myfirst.eti` over a USRP B200 on channel 13C:

```
[remotecontrol]
telnet=1
telnetport=2121

[input]
transport=file
source=myfirst.eti
loop=1

[modulator]
gainmode=2
digital_gain=0.8

[firfilter]
enabled=0

[output]
output=uhd

[uhdoutput]
master_clock_rate=32768000
type=b200
txgain=40
channel=13C
```

This example also shows more options that the example for the file output:

- `remotecontrol telnet=1` enables the Telnet server that can be used to set parameters while the modulator is running.
- `loop=1` rewinds the input file when the end is reached. The same ETI file will be transmitted over and over.
- `gainmode=2` sets the GainMode to VAR, which reduces overshoots in the output.
- `digital_gain=0.8` reduces the output sample deviation, to reduce compression in the USRP.
- `firfilter enabled=0` can be set to 1 to enable an additional FIR filter to improve the spectrum mask. This filter needs a file containing the filter taps, which can be generated using `ODR-DabMod/doc/fir-filter/generate-filter.py`. An example taps file is also available in this folder.
- `master_clock_rate=32768000` sets the USRP internal clock to a multiple of 2048000, which is required if we want to use the native DAB sample rate.
- `txgain=40` Sets the analog transmit gain of the USRP to 40dB, which is specific to the B200.

Some of these options are not necessary for the system to work, but they improve the performance.

Remarks concerning the USRP B200 The USRP B200 depicted in figure 1 is the device we are using most. It's performance is proven in a production environment, it supports the transmit synchronisation necessary for SFN and is robust enough for 24/7 operation.

However, care has to be taken about the host system, especially about the USB controller. Using USB 2.0 is not a problem for a DAB transmission, both USB 2.0 and USB 3.0 host controllers can therefore be used. Since USB 2.0 has been around for longer and is more mature, it is sometimes preferable because it causes less USB errors. This heavily depends on the exact model of the USB controller inside the host PC, and has to be tested for each system.

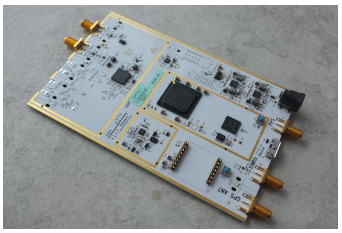


Figure 1: Ettus USRP B200

The txgain on the B200 varies between 0dB and about 90dB. Experience shows that compression effects begin to appear at values around 85dB. This might be different from device to device and needs to measured.

Similarly, the digital gain has to optimised for a given setting. It is important that there is no digital clipping in the chain, because that leads to problematic spurious spectrum components, that can disturb or even damage a power amplifier.

There are some performance measurements available on the Opendigitalradio wiki.[5]

Remarks concerning other USRP models We have used the USRP1, the USRP2 and the USRP B100 with the tools. The WBX is the most appropriate daughterboard for these models.

The txgain setting has another range, it is best to start at 0dB, and increase it in steps of 3dB or smaller while measuring the output signal, until the correct power is reached.

5.2.2 Other Hardware

ODR-DabMod supports other radio interfaces using SoapySDR or through standard output, or via a fifo – the latter must be created prior to runtime with the `mkfifo` command. Due to limitations in the UHD driver library, `/dev/stdout` will only function correctly if ODR-DabMod is configured at compilation time with the following argument:

```
1 --disable-output-uhd
```

SoapySDR[6] is a vendor neutral and platform independent library to drive SDR devices. It can be used to drive the LimeSDR board and the HackRF from Great Scott Gadgets. Installation dependencies are shown in the `INSTALL` file, and an example configuration is in `doc/example.mux`. The TX Gain setting should be chosen inside the valid range for the device being used. This range can be shown by calling `SoapySDRUtil -probe`.

[5] `http://wiki.opendigitalradio.org/index.php/USRP_B200_Measurements`
[6] `https://github.com/pothosware/SoapySDR/wiki`

ODR-DabMod has been tested working with HackRF on i386 and x86 architectures.[7]

The unit is an entry level yet versatile SDR which provides coverage between ≈ 10MHz to 6GHz, and DAB signals been successfully generated with it in VHF Band III (174–240MHz), L-Band (1462–1467.5MHz) and even the worldwide ISM Band (2400–2500MHz). The latter (subject to local regulations) is a licence exempt band which may be useful for performing freely radiating tests at low power. Cheap MMDS converters are currently available which helpfully provide a Band III IF output providing a direct feed to the aerial input of a receiver. Before choosing a converter it is important to pay close attention to the specifications. The local oscillator phase noise performance, and the dynamic range (due to the heavy use of the band) are both particularly important.

To use the HackRF through stdout (i.e. without SoapySDR), the output of ODR-DabMod must be set (in the configuration file) to produce 8-bit signed integers, rather than the default complex floats. HackRF has selectable baseband filters, however the lowest filter setting (1.75MHz) does not provide adequate image rejection at the native sampling rate of 2048k samples per second. An appropriate rate to start with is 4096k, and for some purposes this may well be adequate as this moves the image signals generated within the radio far enough into the stop-band of filter to attenuate them significantly. The digital gain in the ODR-DabMod configuration file should be set to a maximum of 2.4 at this rate to avoid digital clipping on modulation peaks.

Example of the settings in the `mod.ini` file suitable for use with HackRF:

```
[remotecontrol]
telnet=1
telnetport=2121

[input]
transport=file
source=myfirst.eti
loop=1

[modulator]
gainmode=2
digital_gain=2.4
rate=4096000

[firfilter]
enabled=1

[output]
output=file

[fileoutput]
```

[7] HackRF has not been tested to any degree of success with ARM-based computers at this time as they are not (yet) capable of resampling to the required higher rates as the process is highly CPU intensive.

```
format=s8
filename=/tmp/ofdm.fifo
```

The output fifo has to be created beforehand, and the `hackrf_transfer` utility is then used to transmit the signal to the device.

Depending on the capabilities of the host computer, using higher sampling rates (6144k, and even 8192k) may be possible. This oversampling is desirable as it helps to produce a cleaner spectral output. At higher rates one needs to ensure that samples are not being dropped on the USB and that CPU resources are not being contended. It is also important to note that the digital gain value must also be scaled accordingly as the sampling rate is increased. Two sets of values are provided which reflect the theoretical values, and the second set given in parentheses are empirical maximum values determined while monitoring shoulder performance (measured at 970kHz offset from the centre frequency) using a spectrum analyser in ≈ 3 kHz resolution bandwidth. The digital gain figures for the tested sampling rates are shown below:

Rate	Dgain	Max Dgain (Empirical)
4096ksps	2.0	2.25
6144ksps	3.0	3.37
8192ksps	4.0	4.50

The shoulder performance has been measured with shoulder performance at a little better than 35dB, which is roughly equivalent to that obtained from first generation commercial modulator equipment. This can be increased to a relatively respectable $\approx$ 40dB by enabling the FIR baseband filter in ODR-DabMod, and supplying it with an appropriate coefficient (tap) file. The maximum output power available to mmet these performance figures is approximately -10dBm RMS.

Example of using ODR-DabMod with the `hackrf_transfer` utility:

```
mkfifo /tmp/ofdm.fifo
odr-dabmod mod.ini &
hackrf_transfer -t /tmp/ofdm.fifo -f 216928000 -x 47 \
    -a 1 -s 4096000 -b 1750000
```

5.3 Audio Sources

Preparing a DAB multiplex with different programmes requires that we are able to read and encode several audio sources. We have seen in section 4.2.1 how the encoders can be interfaced to the modulator. In this section we'll go through the different ways to carry the audio data to the encoder.

5.3.1 Local Audio Card

It is possible to use an audio card connected to the computer as source. For very simple scenarios, the ALSA input for ODR-AudioEnc is easiest to set up. This however limits the usage of a single encoder per sound-card, and will not scale well if more than one programme has to be encoded on the machine. It is however ideal

for dedicated encoding machines that can contribute the encoded audio over an IP network.

An alternative to using ALSA is JACK[8] that can be used with a multi-channel sound card. JACK will expose every audio input channel, and several encoders can be launched that also connect to JACK. The input channels can be freely connected to the encoders thanks to the virtual JACK patch panel.

It might be possible to use the libVLC input too, to be defined.

5.3.2 Using Existing Web-Streams

One common scenario is to transmit radio stations that already are available as web-radio streams. For simplicity, it makes sense to get these web streams, which are most often encoded in mp3 and available through HTTP, decode them, and use them as audio source for the DAB or DAB$^+$ encoder.

The advantage of this approach is that the radio itself does not need to setup a new infrastructure if the stream is of good quality. The main disadvantage is that the audio is encoded twice, and this coding cascading degrades the audio quality.

Often, web-streams are encoded in mp3 at 44100Hz sample-rate, whereas DAB is most often 48000Hz or sometimes 32000Hz. A sample-rate conversion is necessary in the stream decoder.

There are many different stream decoders, and gstreamer, mpg123 and mplayer have been tested. By far the easiest way is to use the libVLC binding that can be compiled for ODR-AudioEnc. This library has the same features as the VLC audio player, but the audio data is directly passed to the encoding routines. This allows the encoder to receive all network sources VLC supports, not only HTTP web-streams but also less common setups e.g. encoded audio inside multicast UDP MPEG-TS. This is illustrated in "Studio A" in figure 2.

We have also achieved good results with mplayer, and the dab-scripts repository[9] contains the script `encode-jack.sh` that uses mplayer, and illustrates how it is possible to encode a web-stream to DAB$^+$. JACK is used to interconnect the stream decoder to the DAB$^+$ encoder. This is illustrated in "Studio B".

The scripts are designed for production use, and also contain automatic restart logic in case of a failure. They send an email and write a message into the system log.

5.3.3 Encoders at Programme Originator Studios

In order to avoid the unavoidable encoder cascading when using mp3 web-streams, the DAB or DAB$^+$ encoder has to be moved to the programme originator's premises, and should directly encode the audio signal coming from the studios. This is illustrated in "Studio C" in figure 2.

If "Studio C" is able to prepare slides for MOT Slideshow and text to be sent as DLS, ODR-PadEnc can be used to prepare the PAD data for ODR-AudioEnc.

[8] The JACK Audio Connection Kit is a virtual audio patch, `http://www.jack-audio.org`

[9] `http://github.com/Opendigitalradio/dab-scripts`

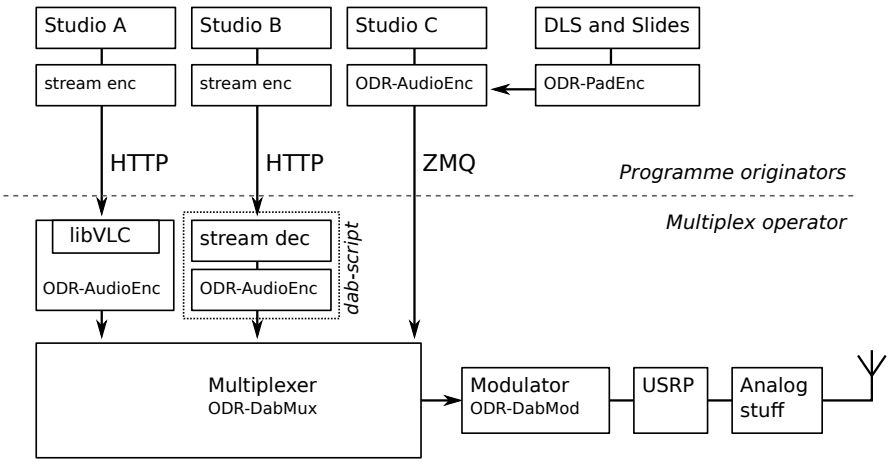


Figure 2: Three common ways to encode a remote audio sources.

6 Data Features

6.1 Announcements

The ODR-DabMux multiplexer supports the insertion of FIG 0/18 and FIG 0/19 that are used to define and trigger announcements according to ETSI TR 101 496-2 Clause 3.6.8 [2]. An example configuration is available in the ODR-DabMux repository, in `doc/advanced.mux`.

The best known application for announcements is traffic information, but other kinds of announcements can also be signalled. ODR-DabMux allows triggering the announcements through the telnet and ZMQ remote control interfaces.

6.2 Service Linking

ODR-DabMux also supports the ability to inform receivers about other ways to receive a given service, through the FIGs 0/6, 0/21 and 0/24. FIG 0/6 communicates the identifiers of services linked together, 0/21 informs the receiver about other frequencies, and 0/24 includes information about other DAB ensembles carrying the linked service. Their interaction is outlined in ETSI TS 103 176 [3].

You will find an example configuration in the ODR-DabMux repository, in `doc/servicelinking.mux`.

7 System Environment

In this section, we describe the system configuration requirements for the continuous operation of the tools. The production environment differs in some respects to those used for experimentation and in laboratory testing. Monitoring, automatic recovery (in case of errors) and resilience are crucial in 24/7 operations. The term *production environment* will be used here to refer to such use.

7.1 Launching the tools

Services running in a production environment are usually administered remotely, and must be able to run without user intervention, or connection. Traditionally, such services are implemented (in UNIX terminology) as 'daemons'. These are started and stopped using the init system contained within the distribution. The ODR-mmbTools cannot daemonise themselves, so this requires another approach:

Screen multiplexer A simple approach is to use a screen multiplexer such as *GNU Screen* or *tmux* - either of these can be used to launch a session from which the user can detach, and later re-attach at will - leaving the tools running within it. Please see the relevant manpages for more information.

Although a screen multiplexer alone permits the tools to run without a user being connected, it alone cannot automatically restart failed processes, and it is unable to provide warnings in the case of a problem.

The dab-scripts, already mentioned in 5.3.2, can be employed to monitor the processes and (if necessary) restart them, and send an alert via email.

supervisord As an alternative to using the scripts, the execution of the tools can also be carried out by a dedicated tool. `supervisor`[10] is (as the name implies) such a tool.

Once installed, supervisor reads its configuration file in `/etc/supervisor.conf` and launches the processes that are to be monitored. Each process is described by a file. The following example assumes the tools are run as user `odr`, and that the multiplex configuration is in `/home/odr/config.mux`, and that ODR-DabMux is to be launched.The logs of ODR-DabMux is written to the specified log files.

```
[program:ODR-DabMux]
command=odr-dabmux config.mux
directory=/home/odr
user=odr
autostart=true
autorestart=true
stdout_logfile=/home/odr/logs/mux.out.log
stderr_logfile=/home/odr/logs/mux.err.log
```

Once this configuration has been added to the supervisor configuration, the settings have to be re-read using:

```
supervisorctl reread
```

[10] `http://supervisord.org`

In order for supervisor to start managing and running this process, it needs to be added:

```
supervisorctl add ODR-DabMux
```

Setting up more processes (such as any of the other tools) can be easily achieved by customising the configuration template above. Examples are provided in the `mmbtools-aux` repository, under the `supervisor` folder - these need to be changed to reflect the paths that are in use on your system.

supervisor also includes a small web-server that can display the state of the managed processes. It is enabled with the `[inet_http_server]` setting in the configuration file.

7.2 Logging

Collecting information about events is essential within a production environment. This information is essential forensic analysis, and tracing sources of trouble. This is achieved through the logging of important messages that can be sent by the tools.

ODR-DabMux and ODR-DabMod both support logging to standard error, to a file and to the system logger `syslog`. Logging to syslog is the most flexible approach; log information can be forwarded over the network to a centralised logging server - where logs can then be filtered according to the priority of each message. Both tools log to the LOCAL0 facility which in turn can be redirected into an ODR-mmbtools specific log file.

Describe rsyslog configuration

In order to avoid the log files from becomming undesirably large, `logrotate` should be set to rotate the files automatically.

Describe logrotate configuration

7.3 Timing

The ODR-mmbTools require the system time to be accurate in order for them to function correctly - this is especially important when running a SFN, but is also important for standalone transmitters when in a production environment. It is also important to remember that most receivers have a clock that is synchronised to the clock time which is being transmitted by the multiplex to which it has been tuned.

The system needs to run a NTP client that synchronises the system time over the network. Correct synchronisation can be checked using the `lpeers` command of the `ntpq` tool. The magnitude of the offset should be below 10 ms.

The performance of the NTP synchronisation should also be monitored permanently during operation.

7.4 Monitoring using munin

The Munin[11] monitoring tool can create graphs for essential system health parameters. It can also send emails if values transgress the defined bounds - this

[11] `http://munin-monitoring.org/`

assists the operator in the assessment of system status, as well as the health of the services.

In addition to basic system measurements like CPU, RAM and disk usage, NTP synchronisation, disk and network performance (and much more besides), there are also custom data sources for ODR-DabMux;

These data sources include ZMQ input buffer monitoring (buffer level, underruns and overruns) and the peak audio input levels (mono, or stereo). It can be installed by copying `doc/stats_dabmux_multi.py` to `/etc/munin/plugins.d`. They require that the ODR-DabMux management server is enabled in the configuration, and will automatically generate the graphs for the subchannels used in the configuration.

7.5 Real-time Scheduling

As a general principle, it is prudent not to run tools (that do not need superuser privileges) as the `root` user. The same principle also applies to the ODR-mmbTools, but care has to be taken that the tools can still request real-time scheduling when it is needed.

This is achieved by adding the following to `/etc/security/limits.conf`, assuming the tools are run under the user `odr`.

```
odr             -       rtprio          65
odr             -       nice           -10
```

If you have installed JACK with real-time privileges, you may find this has already been configured for the 'audio' group, written as `@audio`, which should suffice providing your desired user is a member of the 'audio' group.

7.6 Accessing the USRP as Non-root

Superuser privileges are not required to access USB-connected USRP devices, but sometimes the system lacks the configuration to enable normal users to communicate with the device. In that case, it is necessary to add a rule file for `udev`. This file is included in the UHD sources, but might not have been automatically installed. The file is called `10-uhd-usrp.rules`, should be placed into `/etc/udev/rules.d/` and should contain

```
#USRP1
SUBSYSTEMS=="usb", ATTRS{idVendor}=="fffe", ATTRS{idProduct}=="0002", MODE:="0666"
#B100
SUBSYSTEMS=="usb", ATTRS{idVendor}=="2500", ATTRS{idProduct}=="0002", MODE:="0666"
#B200
SUBSYSTEMS=="usb", ATTRS{idVendor}=="2500", ATTRS{idProduct}=="0020", MODE:="0666"
```

8 Single-Frequency Networks

8.1 Requirements

The DAB standard has been designed to enable the creation of transmission networks where several transmitters share the same frequency, and send the same signal synchronously. Such networks are called "Single-Frequency Networks". Each transmitter needs to be fed the same multiplex stream, which must include timing information required for synchronisation. This timing information implies that a time reference must be installed at each transmitter.

The requirements for a SFN can therefore be summarised in three points:

- The signal must be *identical* for each transmitter. This requires a common multiplexers, and a distribution network that carries the ETI to all modulators.
- All transmitters must transmit on the *same frequency*. The modulators require a frequency reference.
- The signal must be transmitted at the *same time*, which requires a time reference at each site. It also implies that the ETI stream must contain timestamps.

The figure 3 shows a SFN setup with two transmitters.

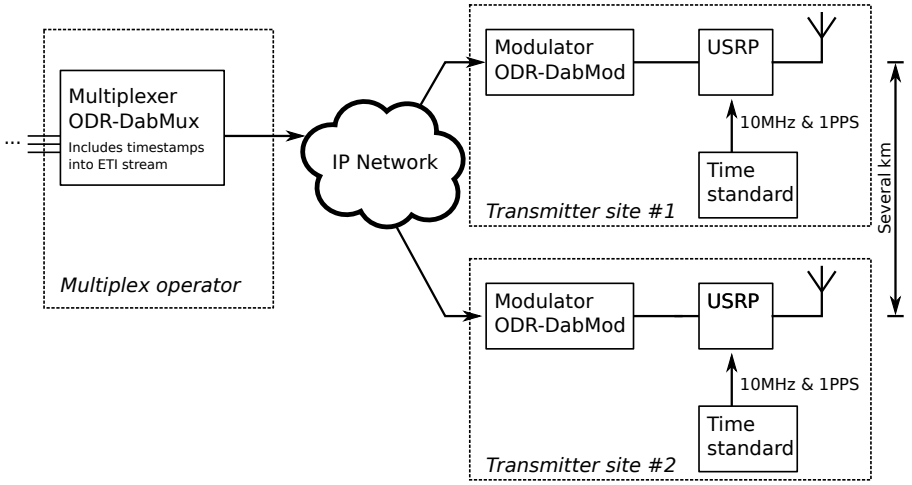


Figure 3: This outline for a SFN shows two transmission sites.

Explain requirements on system time, NTP

8.2 Multiplexer Configuration

On the ODR-DabMux configuration, there are not many options that are specific to an SFN setup. Most importantly, the timestamp feature must be enabled using the "tist" option in the "general" section.

Furthermore, it is recommended to use the ZeroMQ transport between the multiplexer and the modulators, which can be enabled in the "outputs" section.

Care has to be taken to have an output that slows ODR-DabMux down to nominal rate. The ZeroMQ output alone does not enforce this. The following listing shows the relevant options we just covered.

```
general {
    tist true
    ...
}

...

outputs {
    ; Accept connections on all interfaces, on port
       9100
    zmq  "zmq+tcp://*:9100"
    ; This throttles muxing down to nominal rate
    throttle "simul://"
}
```

8.3 Modulator Configuration

Since the modulator has to ensure that the three SFN requirements are satisfied, its configuration is more complex.

We will assume, in this explanation, that one of the following USRP devices is used: USRP2, USRP B100, USRP B200. Other devices also support the necessary time and frequency synchronisation, but they have not been well tested. These USRP devices can accept different sources for the reference clock:

- The default "internal" source uses the non-disciplined clock generator inside the USRP. It is not suitable for SFN.
- The "external" source corresponds to the SMA connector on the USRP. A 10MHz signal from an external source must be connected to it.
- The optional GPSDO that can be mounted inside the USRP, and is selected as source with the "gpsdo" setting.

For the time reference, the "pps_source" option is used. Possible values are "none", "external" and "gpsdo", with analogous meaning as for the reference clock.

In case the USRP is connected to external references, the relevant configuration would be as follows:

```
[uhdoutput]
refclk_source=external
pps_source=external
```

These settings alone do not tell the modulator to enable synchronisation of the transmission, they only select how the USRP is configured. To enable timestamp decoding and the frame synchronisation logic in ODR-DabMod, the following settings must also be set:

```
[delaymanagement]
synchronous=1

; The constant offset to be added to the TIST, in
   seconds
offset=2.0
```

The "offset" setting deserves some further explanations. The ETI data stream contains TIST information, from which a time-stamp for each ETI frame can be derived. Each ETI frame (24 msinterval) is therefore associated with a precise point in time that defines the time of transmission of the corresponding transmission frame.[12] The TIST information is set to current time at ETI frame generation, and does not take in account the propagation delay across the distribution network. Therefore, we need to add an offset, called δ, to the TIST to define transmission time.

$$t_{transmission} = t_{TIST} + \delta$$

If this offset is set to a higher value, there will be a bigger delay (measured in absolute time) between the point in time a frame is multiplexed and the point in time the frame is transmitted. More frames therefore will be buffered in the ODR-DabMod ZeroMQ input, increasing robustness against network latency fluctuations.

The offset already has two functions: it compensates for network delay and allows a trade-off between delay and robustness. But it also serves a third purpose: When doing coverage planning for an SFN, it is necessary to be able to control the relative delay between transmitters in the order of milliseconds. This tuning of relative delay is included in the "offset" setting. We can therefore rewrite the above equation as:

$$t_{transmission} = t_{TIST} + \delta_{network} + \delta_{planning}$$

$$\delta = \delta_{network} + \delta_{planning}$$

Explain relationship with ZeroMQ max buffer size

8.4 Using ODR LEA-M8F GPSDO board

The ODR GPSDO board integrating a u-blox LEA-M8F module can be used as time and frequency reference for the USRP B200. The board design is available on the Opendigitalradio website, with a bill of materials describing how to source the components. The PCB itself can be manufactured in any PCB fab.

TODO: Add Picture

The module includes the correct pin header so that it can be mounted directly onto the USRP B200, but also includes footprints for SMA connectors for other usages. Communication between the PC and the GPS is possible through USB or over UART through the B200.

The u-blox LEA-M8F module is a GPS disciplined TCXO module, with a one-pulse per second and a reference clock output at a frequency of 30.72 MHz. This is different than what the normal USRP firmware expects.

[12] It is slightly more complex, because one transmission frame is composed of several ETI frames in some transmission modes, but the principle stays the same. It suffices for this explanation that we can derive the transmission time from the TIST information.

Because the UART communication protocol and the reference clock frequency are different than for the GPSDO units Ettus supports, a modified version of UHD is necessary. This version includes new UHD sensors, used by ODR-DabMod to verify that the GPSDO is locked properly, and different configuration settings for the clock management PLL inside the USRP, making the USRP compatible to the 30.72MHz reference clock frequency.

The modified UHD version is available at `http://www.github.com/Opendigitalradio/uhd.git` and replaces Ettus' UHD.

ODR-DabMod can be configured as follows:

```
[uhdoutput]
refclk_source=gpsdo
pps_source=gpsdo
behaviour_refclk_lock_lost=crash
max_gps_holdover_time=600
```

8.5 Using Ettus GPSDO

Give example

A ODR-DabMux ETI file formats

ODR-DabMux supports three output formats for the ETI stream, that have been described on the mmbTools forum website.[13]

The three formats are called *framed*, *streamed* and *raw*.

The *framed* format is used for saving a finite ETI stream into a file. Each frame does not contain any padding, and the format can be described as follows:

```
uint32_t nbFrames
// for each frame
  uint16_t frameSize
  uint8_t data[frameSize]
```

When streaming data, in which case the number of frames is not known in advance, the *streamed* format can be used. This format is identical to the first one except for the missing `nbFrames`.

```
// for each frame
  uint16_t frameSize
  uint8_t data[frameSize]
```

The *raw* format corresponds to ETI(NI), where each frame has a constant size of 6144 Bytes. The padding in this case is necessary.

```
// for each frame
  uint8_t data[6144]
```

In order to select the format, the following syntax for the `-O` option is used: `-O file://filename?type=format`, where `format` is one of `framed`, `streamed` or `raw`.

B Bibliography

References

[1] ETSI. *TR 101 495, Digital Audio Broadcasting (DAB); Guide to DAB standards; Guidelines and Bibliography*, November 2000. V1.1.1. All DAB standards are available at `http://www.etsi.org/WebSite/Technologies/DAB.aspx`.

[2] ETSI. *TR 101 496-2, Guidelines and rules for implementation and operation; Part 2: System features*, November 2000. V1.1.1.

[3] ETSI. *TS 103 176, Digital Audio Broadcasting (DAB); Rules of implementation; Service information features*, July 2013. V1.1.2.

[4] Hoeg, W., and Lauterbach, T. *Digital Audio Broadcasting; Principles and Applications of DAB, DAB+ and DMB*. John Wiley & Sons Ltd., 2009.

[13] `http://mmbtools.crc.ca/component/option,com_fireboard/Itemid,55/func,view/id,4/catid,13/#28`