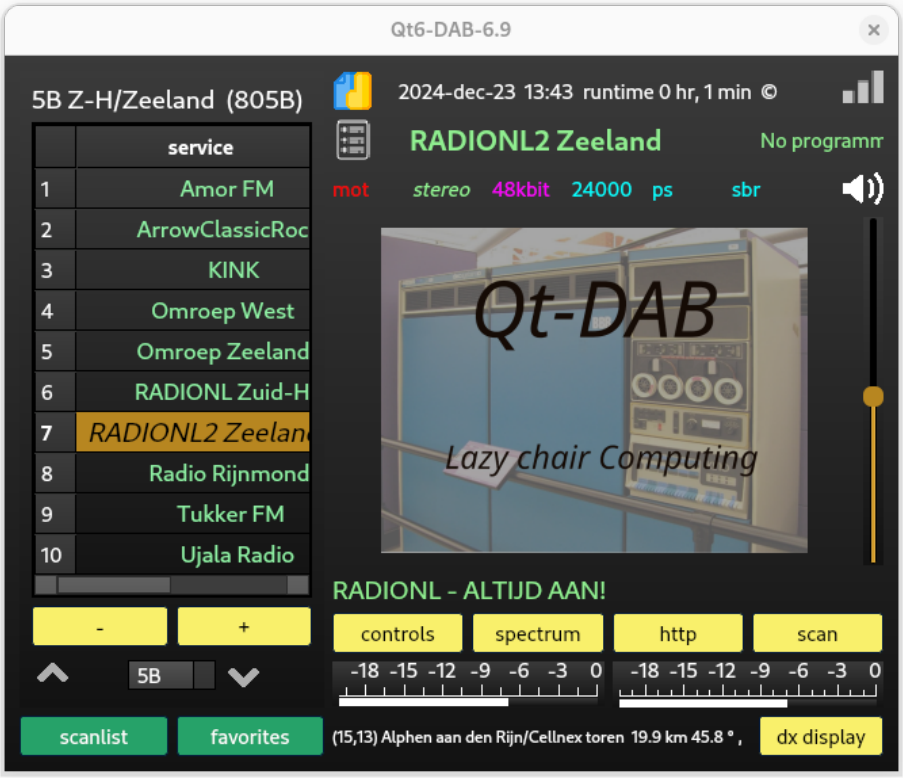
Building Qt-DAB 6.9*

Jan van Katwijk, Lazy Chair Computing
The Netherlands

April 9, 2025



Contents

1 A Note on building an executable

1.1 Introduction

While for both Windows (32 and 64 bits) and x64 Linux precompiled versions are available, there might be situations were once wants (or needs) to build and executable from the sources. All developent is done using a gcc based toolchain, for the Windows executables the *mingw64-xx* chain is used. In *theory* it would be possible to build a Windows version with the microsoft C and C++ toolchain, due to incompatibilities between the different toolchains this requires source modifications.

The description here is - therefore - based on the use of a gcc based toolchain for a unix/linux type system.

The process itself consists of the following steps:

- Download the sources and required libraries;
- Select the AAC decoder;
- Configure deviices and install support libraries;
- Handle further configuration issues;
- Selecting a Viterbi decoder;
- Compiling and installing.

For building, one uses the *qmake* or the *cmake* path. Personally I prefer to use *qmake*, however, a CMakeLists file is available and building an executable with cmake is cerainly possible. The CMakeLists.txt contains - in its current form - less possibilities for setting the configuration than the qt-dab-69.pro file, used for qmake.

Anyway, assuming the configuration settings are done, use

```
mkdir build
cd build
cmake .. -DXXX=ON
make
```

(obviously, replace XXX by the setting you want enabled) Note that the CMakeLists.txt file is set for use with Qt6.

```
qmake (or qmake-qt5, qmake6 ...)
make
```

1.2 Download the source, the required libraries

For downloading the Qt-DAB sources, one needs "git" to be present, on a Debian based system, e.g. Raspbian on an RPI3 or 4

```
sudo apt-get update
sudo apt-get install git
```

Sources for Qt-DAB can be downloaded

```
git clone https://github.com.JvanKatwijk/qt-dab
```

Sources, specific to Qt-DAB-6.9, can be found in the subdirectory "qt-dab-6.9". The so-called ".pro" file, i.e. the one processed by qmake, is named "qt-dab-6.9.pro".

Continuing on the Debian based system, one should load the required libraries (and toolchain)

```
sudo apt-get install qt5-qmake build-essential g++
sudo apt-get install pkg-config
sudo apt-get install libsndfile1-dev
sudo apt-get install libfftw3-dev portaudio19-dev
sudo apt-get install zlib1g-dev
sudo apt-get install libusb-1.0-0-dev mesa-common-dev
sudo apt-get install libgl1-mesa-dev libqt5opengl5-dev
sudo apt-get install libsamplerate0-dev libqwt-qt5-dev qtmultimedia5-dev
sudo apt-get install qtbase5-dev libqt5svg5-dev
```

Note that building an using Qt-6 will bring some changes, especially in the Qt-Audio handling. The sources are parameterized on the Qt version, on my development laptop I can build versions using Qt5 or Qt6 without any need to change the sources. The ".pro" file is parameterized on the Qt version as well. The location of the Qwt include files and library files may differ though.

The CMakeLists.txt file is using the Qt6 and accompanying Qwt libraries,

1.3 Select the AAC decoder

While both the *libfaad* and *libfdk-aac* library can be used, the latter is preferred. Unfortunately, not all Linux distribitions provde a correct version. Depending on the choice in the configuration file (the file qt-dab-6.9.pro)

```
CONFIG         += faad
#CONFIG          += fdk-aac
```

the support library for *libfaad* or for *libfdk-aac* needs to be installed. For libfaad one may try

```
sudo apt install libfaad
```

For libfdk-aac one may try

```
sudo apt install libfdk-aac-dev
```

It turns out that in some cases the *libfdk-aac* as provided by the Linux distribution does not work properly. One can easily build the library from the sources as is done for the AppImage built on Ubuntu 20.

```
git clone https://github.com/mstorsjo/fdk-aac
cd fdk-aac
mkdir build
cd build
cmake ..
make
sudo make install
```

1.4 Configure devices and install support libraries

Of course the support library for the device used should be installed as well, the followin devices can be included in the configuration (note that support for file input is always included in the configuration):

```
CONFIG += sdrplay-v2
CONFIG += sdrplay-v3
CONFIG += dabstick-linux
CONFIG += airspy-2
CONFIG += hackrf
CONFIG += lime
CONFIG += soapy
CONFIG += pluto
CONFIG += rtl_tcp
CONFIG += spyServer-16
CONFIG += spyServer-8
```

It is advised to comment out all devices from the configuration that are not used.

- Support libraries for SDRplay devices can be downloaded from "SDRplay.com",
- support for the SDRplay-v2 library (2.11) can be built in, however, for Windows the v2 library will not work anymore;
- Most Linux distributions provide support libraries for the Airspy, for Lime and for hackrf devices.
- Support libraries for the Adalm Pluto can be obtained from Analog Devices.
- While most Linux distributions provide a support library for the RT2832 based dabsticks, they need "blackboxing " some kernel modes. It is often easier to build a linrary yourself, see "https://osmocom.org/projects/rtl-sdr/wiki/Rtl-sdr" for details on building a shared library.
- For *Soapy*, look at "https://github.com/pothosware/SoapySDR"
- for rtl_tcp, and the 8 and 16 bit spy server interface, one needs to install the server, which is beyonf the scope of this guide.

1.5 Handle further configuration issues

The *qt-dab-6.9.pro* file contains a few possible configuration settings, most of them have reasonable defaults.

```
#CONFIG         += console
CONFIG          -= console
```

sets whether or not output is to be written to the terminal.

```
#DEFINES        += __MSC_THREAD__
DEFINES         += __THREADED_BACKEND__
```

The first option states whether or not a part of the (rather heavy) FFT operation in the front end of the processing are to be done in a separate thread or not. For RPI 3 and up there is no need to have that enabled

The second option, when enabled, instructs the software to run each backend on its own, separate, thread. A "backend" is the set of modules that interprets a selected (sub)service. Of course, when running several backends simultaneously, it is beneficial to have this option enabled.

```
#CONFIG         += double
CONFIG          += single
```

The setting determines whether all computations on the incoming signal (the "front end") are to be done in single or double precision

1.6 Selecting a viterbi decoder

The viterbi decoder, used to transform a sequence of "soft" bits as being generated by the front end of Qt-DAB into "hard" bits, is implemented with different flavours of support by specialized CPU instructions. The ".pro" files offers

```
#CONFIG         += viterbi-scalar
#CONFIG         += viterbi-sse
#CONFIG         += viterbi-avx2
CONFIG          += spiral-sse
#CONFIG         += spiral-no-sse
```

As the names suggest, the first and the last do not use specific CPU support, the others SSE resp. AVX2 support. If unsure: use the first or the last configuration option. For support from specific (neon) instructions on an RPI, see the ".pro" file.

1.7 Compiling, installing and running

Once all required libraries are installed, and the configuration is as it should be, run

```
qmake
```

Depending on the Linux distribution, *qmake* or *qmake-qt5* is the correct name. There is always a chance that running qmake fails, because some library cannot be found. Usually this is an issue with the location of the qwt library. Add the correct path to the qwt include files to the INCLUDES section of the configuration file.

```
make -j X
```

The *X* in the second line tells how many parallel threads should be used. For an RPI, use 4.

The resulting executable is installed in the subdirectory *linux-bin*.

2 Adding support for a device

2.1 The Qt-DAB device interface

The Qt-DAB device interface is defined as a class, where the actual device handler inherits from.

```
class deviceHandler: public QThread {
Q_OBJECT
public:
deviceHandler ();
  virtual ~deviceHandler ();
  virtual bool restartReader (int32_t freq);
  virtual void stopReader ();
  virtual int32_t getSamples (std::complex<float> *, int32_t);
  virtual int32_t Samples ();
  virtual void resetBuffer ();
  virtual int16_t bitDepth () { return 10;}
  virtual QString deviceName ();
  virtual bool isFileInput ();
  virtual int32_t getVFOFrequency ();
//
// all derived classes are subject to visibility settings
// performed by these functions
bool getVisibility ();
void setVisibility (bool);
//
protected:
superFrame myFrame;
int32_t lastFrequency;
        int theGain;
signals:
void frameClosed  ();
};
```

While the class is merely an interface class, visibility of the driver's widget is common to all inheritors and therefore implemented in the body of this class. The functions to

handle visibility are therefore not *virtual*, they operate op *myFrame*. The class *super-Frame* merely adds controlled termination to a widget.

A device handler for a - yet unknown - device should implement this interface. While not stated explicitly, it is assumed (and essential) that the samplerate for the delivered samples is 2048000 Samples/second.

A description of the interface elements follows

- *stopReader* and *restartReader* are called on switching from one channel to another, and their function is what the name suggests, stopping the data stream to Qt-DAB and restarting the data stream on the given frequency. Note that for most devices the device-IO is actually stopped and restarted, there is no need to implement it that way though. Calling *restartReader* when already running or the *stopReader* when already stopped should have no effect.

- *getVFOFrequency* returns the current oscillator frequency in Hz;

- *getSamples* is the interface to the samples. It asks for reading the amount (the number passed as parameter) samples, the return value is the number of samples actually read into the buffer;

- *Samples* tells the amount of samples available for reading. If the Qt-DAB software needs samples, the function *Samples* is continuously called (with the delay between the calls) until the required amount is available, after which *getSamples* is called.

- *resetBuffer* will clear all buffers. The function is called on a change of channel.

- *bitDepth* tells the number of bits of the samples. The value is used to scale the Y axis in the various scopes and to scale the input values when dumping the input.

- *deviceName* returns a name for the device. Some (dumping) operations use the devicename in the created filename.

- *isFileInput* tells - as the name suggests - whether or not the input is from a file or a device. When file input is "on", operations involving e.g. changing the channel (scanning) are not very useful and will be ignored.

2.2 What is needed to install another device

Having an implementation of the above mentioned functions for the new device, Qt-DAB has to know about the device handler. This requires adapting the configuration file (here we look at qt-dab.pro) and the *device selector*.

Modification to the qt-dab.pro file Driver software for a new device, here called *newDevice*, should implement a class *newDevice*, derived from the class *deviceHandler*.

It is assumed that the header is in a file *new-device.h*, the implementation in a file *new-device.cpp*, both stored in a directory *new-device*.

The name of a new paragraph, e.g. *newDevice*, will be added to the list of devices, i.e.

```
   CONFIG += AIRSPY
...
   CONFIG += newDevice
```

Next, somewhere in the qt-dab.pro file a paragraph describing the files for the new device should be added, with as name the same name as used in the added line with CONFIG.

```
newDevice {
   DEFINES         += HAVE_NEWDEVICE
   INCLUDEPATH     += ./qt-devices/new-device
   HEADERS         += ./qt-devices/new-device/new-device.h \
                        .. add further includes to development files, if any
   SOURCES         += ./qt-devices/new-device/new-device.cpp \
                        .. add further implementation files, if any
   FORMS           += ./qt-devices/new-device/newdevice-widget.ui
   LIBS            += .. add here libraries to be included
}
```

2.3 Modifications to the device selector

The class *deviceChooser* implements device selection, and is implemented in the file *device-chooser.cpp*

In this file, first the include file need to be added, and a constant needs to be chosen for identifications.

In the list of includes add

```
#ifdef HAVE_NEWDEVICE
#include new-device.h
#define NEW_DEVICE XXX
#endif
```

where XXX is a number unique in the device chooser.

The constructor of the class *deviceChooser* builds op a list (vector) with assocations between the name of the new device as string, and the constant identifier, defined above. In the neighbourhood of e.g.

```
#ifdef  HAVE_HACKRF
        deviceList. push_back (deviceItem ("hackrf", HACKRF_DEVICE));
#endif
```

the text

```
#ifdef HAVE_NEWDEVICE
deviceList. push_back (deviceItem ("newDevice", NEW_DEVICE));
#endif
```

is added.

In the function *_createDevice* code is added to actually create an instance of the driver for the new device Again, in the environment of

```
#ifdef  HAVE_HACKRF
case HACKRF_DEVICE:
   return new hackrfHandler (dabSettings, version);
#endif
```

the code for allocating a device handler is added

```
#ifdef  HAVE_NEWDEVICE__
case NEW_DEVICE:
   return new newDevice (...);
#endif
```

with parameters as needed.

2.4 Linking or loading of device libraries

The approach taken in the implementation of the different device handlers is to *load* the required functions for the device library dynamically, i.e. on instantiation of the class. This allows execution of Qt-DAB even on systems where some device libraries are not installed.

The different existing drivers can be used as example if there is a need to implement the dynamic loading feature. Obviously, if an executable is generated for a target system that does have the library for the device installed, there is no need to dynamically load the functions of that library.