Qt-DAB:6: Building an executable (V0.7)

Jan van Katwijk, Lazy Chair Computing
The Netherlands

December 6, 2023

Contents

1 Introduction

Precompiled versions for Linux, and installers for Windows for the Qt-DAB program are available. Nevertheless, it might be desirable to build one's own version of an executable. This guide will give directions for doing so.

The precompiled versions, i.e. the AppImages for Linux x64 and the installers for Windows are created using Qt, gcc and mingw64. It is most likely possible to use other toolchains, i.e. MVC for Windows, but I do not have experience with it.

Generating a Make file for compiling and building is done by either cmake - a CMakeLists.txt file exists, or with QMake, for all three versions such configuration files exist.

It is strongly advised to use the Qmake/make route, The configuration files for qmake, i.e. the ".pro" files are the primary ones used in the development and contain by far the most configuration options.

The recommende path for creating an executable is:

- Install the required common libraries;
- Have the Qt-DAB sources downloaded. Note that there are three versions, the 4, 5 and 6 version, each with its own directory, and within that directory a ".pro" file, specific for that version.
- Set in the onfiguration file(s) the required options "on" or "off" (i.e. it is possible to include a number of device interfaces, there is of course no need to have device interfaces installed for unavailable devices).
- Install - if needed - libraries required by the settings you made.
- run qmake in the directory for the version you have chosen (Note that qmake may appear with variations on the name in different Linux versions, i.e. qmake-qt5);
- run "make", the resulting executable will be put into a "linux-bin" directory within the directory the "pro" file is in

2 Installing the common libraries

Prior to compiling, some libraries have to be available. For Debian based systems (e.g. Ubuntu for PC and Buster for the RPI) one can load all required libraries with the script given below.

```
sudo apt-get update
sudo apt-get install git cmake
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
sudo apt-get install qtbase5-dev libqt5svg5-dev
```

Note that on different versions of Ubuntu, this list might be different, especially w.r.t. Qt5 libraries. On Ubuntu 16 it is required to install a package "qt5-default", on later versions this is not required anymore, even worse, the "qt5-default" package does not exist anymore.

Note further that in previous versions, the package "rtl-sdr" was included in the list above. This package, when installed, might give problems. It is advised to generate one's one library for the rtl-sdr device, if neede dof course.

3 Setting configuration parameters

3.1 Console or not (qt-dab.pro only)

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

While for tracing and debugging purposes it might be handy to see all the (text) output generated during execution, for normal use it is not. Including or excluding *console* in the configuration determines whether or not a console is present when executing.

3.2 Single or double precision

In qt-dab-6.pro one may choose between single and double precisions for the signal processing.

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

The precompiled versions are compiled with double precision "on".

3.3 Configuring for parallelism

Current CPU's usually contain more than one core. Qt-DAB is designed such that selected operations can be done in parallel (For the RPI 2 this was essential, running Qt-DAB on a single core overloaded that core). In qt-dab-6.pro :

```
#
#       For more parallel processing, uncomment the following
#       defines
#DEFINES        += __MSC_THREAD__
DEFINES        +=  __THREADED_BACKEND
```

In case cmake is used, edit the file CMakeLists.txt and comment out or uncomment the line

```
#add_definitions (-D__MSC_THREAD__)  # uncomment for use for an RPI
#add_definitions (-D__THREADED_BACKEND)  # uncomment for use for an RPI
```

The first option ensures that the largest part of the FFT processing is done in a function running in a separate thread. The second option ensures that each backend - recall that more than a single backend may be active - runs in its own thread.]par The first option was included to have the software run on an RPI 2, for systems with faster running CPU's it is advised to not set that option.

3.4 Configuring the deconvolution

Deconvolution is a rather time consuming process, and is a critical step in transforming the decoded DAB data to the actual bits. An optimized version of the deconvolution uses the SSE instructions as available on a current PC. For RPI type systems, use NO_SSE.

The options in the qt-dab-6.pro file are:

```
CONFIG          += PC
#CONFIG         += NO_SSE

PC {
DEFINES += SSE_AVAILABLE
HEADERS += ../src/support/viterbi-spiral/spiral-sse.h
SOURCES += ../src/support/viterbi-spiral/spiral-sse.c
}

NO_SSE {
HEADERS += ../src/support/viterbi-spiral/spiral-no-sse.h
SOURCES += ../src/support/viterbi-spiral/spiral-no-sse.c
}
```

Using CMake, the option SSE_AVAILABLE is set for Linux PC. For RPI type systems, the CMakeLists.txt file contains an attempt, but one should carefully look into it.

3.5 Configuring an AAC decoder

The source tree contains - in the directory *specials*, the sources for the libfaad-2.8 version. It is quite simple to create and install an appropriate library if the Linux version supports a faad library that is somehow incompatible.

An *alternative* is to use the *fdk-aac* library to decode AAC (contrary to the libfaad the fdk-aac library is able to handle newer versions of the AAC format, these newer versions are not - yet - used in DAB (DAB+)).

Selecting the library for the configuration is by commenting out or uncommenting the appropriate line in the file *qt-dab.pro* (of course, precisely one of the two should be uncommented).

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

faad {
DEFINES += __WITH_FAAD__
HEADERS += ../includes/backend/audio/faad-decoder.h
SOURCES += ../src/backend/audio/faad-decoder.cpp
LIBS += -lfaad
}

fdk-aac {
DEFINES += __WITH_FDK_AAC__
INCLUDEPATH += ../specials/fdk-aac
HEADERS += ../includes/backend/audio/fdk-aac.h
SOURCES += ../src/backend/audio/fdk-aac.cpp
PKGCONFIG += fdk-aac
}
```

By default - i.e. in the precompiled v ersions - version 6 is compiled with the fdk-aac library.

3.6 Configuring the TII library

```
#CONFIG          += preCompiled
CONFIG         += tiiLib

tiiLib  {
        INCLUDEPATH     += ../src/support/tii-library
        HEADERS         += ../src/support/tii-library/tii-codes.h
        SOURCES         += ../src/support/tii-library/tii-codes.cpp
}
```

As mentioned elsewhere, the code to upload the TII library from the host is **not** part of the open source tree. The code is included in the precompiled versions though. Since I am using the preCompiled option, it is part of the configuration, but please uncomment the option *tiiLib* only.

Note that CMakeLists.txt does not provide an option here.

Installing the database The sourcetree for Qt-DAB contains a mysterious file named *tiiFile.zip* that can be unzipped to show a - recently updated - database with TII data. Installing that as *.txdata.tii* in the user's hoe directory, and having the *tiiLib* installed, gives access to TII data to be shown on a map.

3.7 Configuring a data streamer

The TDC data is not interpreted by Qt-DAB. To allow external interpretation that TDC data can be sent through an IP data stream. An experimental, simple server for connecting to a tdc handler can be configured. In qt-dab-6. pro we see

```
CONFIG          += datastreamer

datastreamer    {
        DEFINES         += DATA_STREAMER
        DEFINES         += CLOCK_STREAMER
        INCLUDEPATH     += ../server-thread
        HEADERS         += ../server-thread/tcp-server.h
        SOURCES         += ../server-thread/tcp-server.cpp
}
```

In cmake the parameter "-DDATA_STREAMER=ON" can be passed to include handling TPEG as described in Qt-DAB.

The source tree contains a simple client.

3.8 Configuring for embedded IP data

Embedded IP data van be send to an IP port - datagrams - by uncommenting (qt-dab-6.pro only)

```
#CONFIG         += send_datagram

send_datagram {
        DEFINES         += _SEND_DATAGRAM_
        QT              += network
}
```

3.9 Configuring the channel estimator

In order to inspect the effect of the (transmission)channel on the reception of the DAB data, a channel estimator is added to the configuration options in qt-dab-6.pro.

```
CONFIG          += estimator
stimator {
        DEFINES         += __ESTIMATOR_
        INCLUDEPATH     += /usr/include/eigen3
        HEADERS         += ../includes/ofdm/estimator.h
        SOURCES         += ../src/ofdm/estimator.cpp
}
```

Note that selecting the option requires installing the "eigen" library.

3.10 Configuring common devices

Configuring devices is simple, for devices as mentioned above as well as for *rtl_tcp* the *qt-dab.pro* file and the *CMakeLists.txt* contain a description. File input (all versions, i.e. raw files, sdr files and xml files) is by default configured in Qt-DAB executables, changing this is possible, but implies changes to the sources.

Using the qt-dab.pro file In order to have devices configured for which no driver support is available on the target machine, the Qt-DAB software does not need the driver support libraries for devices *as long as these devices are not selected*. On selecting a device, the interface software will (try to) link to the required functions in the device support library.

For configuring devices in the *qt-dab.pro* file, comment out or uncomment the line with the device-name.

```
CONFIG += airspy
CONFIG += dabstick-linux
CONFIG += sdrplay-v2
CONFIG += sdrplay-v3
CONFIG += lime
```

```
CONFIG += hackrf
CONFIG += pluto
CONFIG += soapy
CONFIG += rtl_tcp
```

Note that for soapy additional libraries have to be installed first. Note further that for Windows, soapy is not supported, that "dabstick-linux" should is replaced by "dabstick-windows", and a support library should be available.

Using the CMakeLists.txt file The CMakeLists.txt file contains support for AIRspy, SDRplay, SDRplay_V3, RTLSDR-LINUX, RTLSDR-WIN, Hackrf, pluto and LimeSDR.

Since some time the *interface* for RTLSDR type devices is different for Windows. The reason being that the Windows version of the RTLSDR library, i.e. *rtlsdr.dll* cannot handle sequential closing and reopening the device without crashing. Closing and reopening the device is the mechanism of choice on changing a channel and implemented in all versions of Qt-DAB and derived programs. The Windows version of the interface library was adapted and renamed *RTLSDR-WIN*, while - to avoid confusion - the Linux version - unchanged - was renamed *RTLSDR-LINUX*.

Including a device in the configuration is by adding "-DXXX=ON" to the command line, where XXX stands for the device name.

3.11 Configuring audio

- When running the Qt-DAB program remotely, e.g. on an RPI near a decent antenna, one might want to have the audio output sent through an IP port (a simple listener is available).
- Maybe one wants to use the audio handler from Qt (last years not tested);
- The default setting is to use *portaudio* to send the PCM samples to a selected channel of the soundcard.

The *Linux* configuration for the Qt-DAB program offers in the qt-dab.pro file the possibility of configuring the audio output:

```
#if you want to listen remote, uncomment
#CONFIG         += tcp-streamer         # use for remote listening
#otherwise, if you want to use the default qt way of sound out
#CONFIG         += qt-audio
#comment both out if you just want to use the "normal" way
```

If cmake is used, pass "-DTCP_STREAMER=ON" as parameter for configuring the software for remote listening, use "-DQT_AUDIO=ON" for qt audio, or *do not specify anything* for using portaudio in the configuration.

Note that the configuration for Windows is only for "portaudio".

4 Preparing the build: installing device libraries

4.1 Downloading of the sourcetree

The sourcetree for Qt-DAB can be downloaded from the repository by

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

The command will create a directory *qt-dab* with subdirectories for qt-dab-s4, qt-dab-s5 and qt-dab-s6.

4.2 Installing support for the SDRplay devices

For support of SDRplay devices. one needs to download the device drivers from the website of SDRplay (www.sdrplay.com). Note that Qt-DAB is able to use the 2.13 as well as the 3.xx libraries, however, under Windows, using the 2.13 library together with the 3.xx library will not work.

The 2.13 library does support RSP models, including the SDRplay RSP I, i.e. the first one, it does not support the RDSDx however. On the other hand, the 3.xx library does not support the original SDRplay RSP I.

4.3 Installing support for the Adalm Pluto

The Pluto device uses the *iio* protocol. Support for *Pluto* is by including

```
sudo apt-get install libiio-dev
```

and - to allow access for orinary users over the USB - ensure that the user name is member of the pugdev group, and create a file "53-adi-plutosdr-usb.rules" is in the "/etc/udev/rules" directory.

```
#allow "plugdev" group read/write access to ADI PlutoSDR devices
# DFU Device
SUBSYSTEM=="usb", ATTRS{idVendor}=="0456", ATTRS{idProduct}=="b674",
MODE="0664", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2fa2", ATTRS{idProduct}=="5a32",
MODE="0664", GROUP="plugdev"
# SDR Device
SUBSYSTEM=="usb", ATTRS{idVendor}=="0456", ATTRS{idProduct}=="b673",
MODE="0664", GROUP="plugdev"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2fa2", ATTRS{idProduct}=="5a02",
MODE="0664", GROUP="plugdev"
# tell the ModemManager (part of the NetworkManager suite) that
# the device is not a modem,
# and don't send AT commands to it
SUBSYSTEM=="usb", ATTRS{idVendor}=="0456", ATTRS{idProduct}=="b673",
ENV{ID_MM_DEVICE_IGNORE}="1"
SUBSYSTEM=="usb", ATTRS{idVendor}=="2fa2", ATTRS{idProduct}=="5a02",
ENV{ID_MM_DEVICE_IGNORE}="1"
```

4.4 Installing support for the RTLSDR stick

It is advised - when using an RT2832 based "dab" stick - to create the library for supporting the device. The library, as provided by the various distributions are compiled without the parameter "DETACH_KERNEL_DRIVER" set. In that case you have to blacklist the kernel module. Generating and installing a library is by executing the following lines.

```
git clone git://git.osmocom.org/rtl-sdr.git
cd rtl-sdr/
mkdir build
cd build
cmake ../ -DINSTALL_UDEV_RULES=ON -DDETACH_KERNEL_DRIVER=ON
make
sudo make install
sudo ldconfig
cd ..
rm -rf build
cd ..
```

Note that it might be required to tell the system that the path "/usr/local/lib" is to be added to the searchpath of the loader. I usually add a small file "local.conf", with a single line "/usr/local/lib", to the directory "/etc/ld.conf.d/"

4.5 Installing support for the AIRspy

If one wants to use an AIRspy, a library can be created and installed by

```
wget https://github.com/airspy/host/archive/master.zip
unzip master.zip
cd airspyone_host-master
mkdir build
cd build
cmake ../ -DINSTALL_UDEV_RULES=ON
make
sudo make install
sudo ldconfig
cd ..
rm -rf build
cd ..
```

4.6 Installing support for SDRplay RSP

If one wants to use an RSP from SDRplay, one has to load and install the library from "www.SDRplay.com".

4.7 Making the installed libraries visible

The installation of these device handlers will install libraries in the

```
/usr/local/lib
```

directory. Note that the path to this directory is NOT standard included in the search paths for the Linux loader. To add this path to the searchpaths for the Linux loader, create a file

```
/etc/ld.so.conf.d/local.conf
```

with as content

```
/usr/local/lib
```

The change will be effective after executing a "sudo ldconfig" command.

The installation of these device handlers will furthermore install some files in the

```
/etc/udev/rules.d
```

directory. These files will ensure that a non-root user has access to the connected device(s).

Note that in order for the change to be effective, the *udev* subsystem has to be restarted. The easiest way is just to reboot the system.

5 Finally: building an executable

5.1 Using cmake to build the executable

After installing the required libraries, and after editing the configuration (if required), compiling the sources and generating an executable is simple.

Using cmake, creating an executable with as devices the SDRplay, the AIRspy, and the RTLSDR based dabsticks, the following script can be used:

```
cd qt-dab/qt-dab-s6
mkdir build
cd build
cmake .. -DSDRPLAY=ON -DPLUTO=ON -DAIRSPY=ON -DRTLSDR=ON ... -DRTL_TCP=ON
make
```

The CMakeLists.txt file contains instructions to install the executable in "/usr/local/bin".

```
sudo make install
cd ..
cd ..
```

5.2 Using qmake to build the executable

Assuming the file qt-dab.pro is edited, the same result can be obtained by

```
cd qt-dab/qt-dab-s6
qmake
make
```

In some Linux distributions replace qmake by qmake-qt5!
The *qt-dab-6.pro* file contains in both the section for unix as for windows a line telling where to put the executable

```
DESTDIR         = ./linux-bin
```

By default in Linux the executable is placed in the *./linux-bin* director in the *qt-dab* directory.

5.3 Generating an AppImage

Each of the three versions has - in the local subdireftory - a file *qt-dab-X-vuild.txt*. Making that file executable and running on a not-to-modern distribution can create an AppImage file.

6 The Qt-DAB device interface: adding support for a device

The Qt-DAB software provides a simple, well-defined interface to ease interfacing a different device. The interface is defined as a class, where the actual device handler inherits from. Note that while it might be tempting to implement the devicehandler as an abstract interface, there are several arguments to implement it as a class, providing a number of default functions

- the device interfaces all have their "own" widget, the operations for visibility of the widget are shared;
- take the raw file input as an example, it does not make much sense to implement setting the frequency there, and the device interface should not be forced to implement a useless function;
- a typical default value is to set the "isFile" property to false, of course the burden is for the file readers that will reimplement it by setting it to true (that function is used to inhibit scanning the band while reading a file).

it provides some functions used in all device

```
class deviceHandler {
public:
         deviceHandler  ();
virtual ~deviceHandler ();
virtual bool restartReader (int32_t);
virtual void stopReader ();
virtual int32_t getSamples (std::complex<float> *, int32_t);
virtual int32_t Samples ();
virtual void resetBuffer ();
virtual int16_t bitDepth ();
virtual QString deviceName ();
void show ();
void hide ();
bool isHidden ();
private:
int32_t lastFrequency;
QFrame myFrame;
};
```

As can be seen, the functions "show", "hide" and "isHidden" are not virtual, they operate on "myFrame" and are common to all device interfaces.]par A device handler for a - yet unknown - device should implement this interface[1]. The semantics of most functions are obvious, a few are more specific

- if opening the device fails, the constructor should give up by throwing an exception. Device specific exceptions are defined in the file "device-exceptions.h";
- *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. This function is used in the definition of a proposed filename for the various *dump*s.
- The GUI contains a button to hide (or show) the control widget for the device. The implementation of the control for the device will implement - provided the control has a widget - functions to *show* and to *hide* the widget, and *isHidden*, to tell the status (visible or not). The related functions are NOT virtual, they assume the widget for the new device uses the *myFrame* frame.

What is needed for another device Having an implementation for controlling the new device, the Qt-DAB software has to know about the device handler. This requires adapting the configuration file (here we take qt-dab.pro) and the file "device-chooser.cpp", the file with which devices are selected.

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

A name of the new device e.g. *newDevice* is to be added to the list of devices, i.e.

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

Next, somewhere in the qt-dab.pro file a section describing XXX should be added, with as label 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
}
```

Modifications to device-chooser.cpp The file "device-chooser.cpp" needs to be adapted in three places

- In the list of includes add

```
#ifdef HAVE_NEWDEVICE
#include "new-device.h"
#define NEW_DEVICE 301
#endif
```

The value of the number is not really important, as long as it is unique for devices.

[1]For each function not implemented in the derived class a *no-op* function exists in the base class

- The constructor of the class should be informed that a new device is available

```
#ifdef HAVE_SDRPLAY
deviceList. push_back (deviceItem ("new-device", NEW_DEVICE)'
#endif
```

  This will ensure that both the name and the assocated number ar eknown to the deviceChooser.

- The function *createDevice* will be called to create an instance of the device. It will map the presented name to the associated number, and if found will use that number to select in a case statement the code to acually create an instance of the class implementing the device interface code.

```
   case NEW_DEVICE:
              try {
                 inputDevice_p  = new newDevice (dabSettings,  ....);
              }
              catch (const std::exception &e) {
                 QMessageBox::warning (nullptr, "Warning", e. what ());
                 return nullptr;
              }
              break;
#endif
```

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 on instantiation of the class. This allows execution of Qt-DAB even on systems where some device libraries are not installed.