Software Development on the E310 and E312

Contents

- 1 Application Note Number
- 2 Revision History
- 3 Abstract
- 4 Overview
- 5 QUICKSTART
- 6 Preparing the device
    - 6.1 Network setup
- 7 Preparing the development machine
    - 7.1 Preparation using PyBOMBS
    - 7.2 Preparation from Source
        - 7.2.1 Installing the SDK
        - 7.2.2 Setting up the environment
- 8 Compiling and installing UHD
- 9 Running the new UHD via sshfs
- 10 Compiling and installing GNU Radio
    - 10.1 Using GNU Radio Companion
    - 10.2 Compiling and installing out-of-tree modules (e.g. gr-ettus)

AN-311

Date	Author	Details
2016-05-24	Martin Braun Nicolas Cuervo	Initial creation

This application note covers the software development process on the USRP E310 and E312.

Note: Linux only.

When developing software for the embedded series, the recommended workflow is to set up a cross-compiling environment. Software is *not* compiled on the device, but on a separate machine (a more powerful machine that can compile much faster).

E310/E312 manual page with supplemental info:

[1] https://files.ettus.com/manual/page_usrp_e3xx.html

Also useful:

[2] https://wiki.gnuradio.org/index.php/Embedded_Development_with_GNU_Radio

PyBOMBS documentation:

[3] https://wiki.gnuradio.org/index.php/OE_PyBOMBS

[4] https://github.com/gnuradio/pybombs

If you are in a rush to get stuff done, below you can see a sequence of commands that you can follow to have your device ready to go. For a detailed device preparation, please refer to the following sections.

- This Quickstart makes use of PyBOMBS and assumes a fresh device as well as a fresh installation of a linux distribution, and makes a lot of assumptions which are described as comments.

In your development machine (i.e. host - computer) run:

```
# Necessary software. Assumes: installation directory at ~/prefix
$ sudo apt-get install openssh-server
$ sudo restart ssh
$ sudo apt-get install screen
$ sudo apt-get install python-setuptools python-dev build-essential
$ sudo easy_install pip
$ sudo pip install git+https://github.com/gnuradio/pybombs.git
$ pybombs recipes add gr-recipes git+https://github.com/gnuradio/gr-recipes.git
$ pybombs recipes add ettus https://github.com/EttusResearch/ettus-pybombs.git
$ sudo dpkg-reconfigure dash # select NO
$ pybombs prefix init ~/prefix -R e3xx-rfnoc -a e300

# Set up your e300 device connection. Connect the e300 using USB to the computer. Also connect the e300 to the computer using the ethernet
$ sudo screen /dev/ttyUSB0 115200
# Login is 'root'. Default password is empty.
$ ifconfig eth0 192.168.10.42 netmask 255.255.255.0 up  #This given IP is arbitrary and used along this tutorial.
# Open a new shell and run
$ ssh root@192.168.10.42  #This should have worked. If not, consult [1]
$ ssh dev@desktop         #This should have worked too. This proves the ssh connection needed for sshfs.
$ exit
$ pwd                     #check where we are. Should be /home/root
$ mkdir usr
$ sshfs dev@desktop:prefix/ usr/
$ which uhd_find_devices  # This should point to the OLD UHD, which we dont want: /usr/bin/uhd_find_devices
$ source usr/setup_env.sh
$ which uhd_find_devices  # This should prove that you are ready to go! : /home/root/usr/bin/uhd_find_devices
```

Now you should be able to run examples, or start developing.

It's best to start from scratch for any kind of embedded development, so the first step is to download the images corresponding to the latest release for the device. For the E310/E312, this means downloading the SD card images from the website: http://files.ettus.com/e3xx_images/ and installing it to the SD card according to the manual ([1] - Upgrading / Writing image to sd card).

Once the latest image is flashed, turn on the device.

For the following tutorial, it is necessary that you can SSH into the embedded device from your development machine and vice versa. For information on how to set up networking on the embedded device, consult the manual ([1] - First boot).

In the following, we will assume that the hostname of the development machine is `desktop`, the hostname of the embedded device is `e3xx` and the username on the development machine is `dev`.

On the development machine you should be able to run the following command:

```
$ ssh root@e3xx
```

And on the embedded device you should be able to run the following command:

```
$ ssh dev@desktop
```

This procedure can be done by using PyBOMBS or manually installing all the elements from source.

PyBOMBS (Python Build Overlay Managed Bundle System) has currently a recipe that automates the installation of all the elements necessary to cross-compile for the E3XX. In order to use this method, first download and install PyBOMBS:

```
$ sudo pip install git+https://github.com/gnuradio/pybombs.git
```

NOTE: If you have already PyBOMBS installed, it's better to be sure that you have the latest version. You can update it by running the following command [4]:

```
$ [sudo] pip install [--upgrade] git+https://github.com/gnuradio/pybombs.git
```

After you have PyBOMBS installed, download and install the necessary recipes to set up the machine:

```
$ pybombs recipes add gr-recipes git+https://github.com/gnuradio/gr-recipes.git
$ pybombs recipes add gr-etcetera git+https://github.com/gnuradio/gr-etcetera.git
$ pybombs recipes add ettus https://github.com/EttusResearch/ettus-pybombs.git
```

The "ettus" recipe provides the necessary elements for the E3XX, the gr-recipes repository provides all the other packages (GNU Radio, out of tree modules, etc.). More information about the recipes and PyBOMBS in general can be found at the PyBOMBS manual [4].

On some distributions (e.g. Ubuntu), PyBOMBS can run into shell issues. To avoid those problems, it is recommended to run the following command before continuing, and choose "**no**" in the prompted dialogue:

```
$ sudo dpkg-reconfigure dash
```

Now you can proceed to install the SDK, UHD, GNU Radio and gr-ettus with this simple command:

```
$ pybombs prefix init /path/to/prefix -R e3xx-rfnoc [-a alias]
```

Where "/path/to/prefix" any directory where you want to download and install the whole set of files for the e300 development (for example ~/prefix/ or ~/prefix/e300). Keep in mind that this command will take a while to finish, due to the size and amount of files that need to be downloaded, cloned and installed.

After the installation is finished, proceed to the Running the new UHD via sshfs section.

On your development machine, you will need to create a directory in which the development will take place. This directory is called the **prefix**, and in the following, we will assume it is in `~/prefix`. If you have not created it before, run the following command:

```
$ mkdir ~/prefix
```

Since you're cross-compiling (i.e., compiling for a different architecture than the development machine's) you require an SDK to do the development. Go the page where you downloaded the device image, and make sure you get the corresponding SDK.

SDKs have filenames such as `oecore-x86_64-armv7ahf-vfp-neon-toolchain-nodistro.0.sh`. The files are quite big, so it can take a while to download them.

The SDKs contain the compiler toolchain, but also contain libraries etc. that already exist on the embedded device. This way, when compiling and linking, you know that what you compile on your development machine will also work on the device.

To install the SDK, execute the downloaded file:

```
$ sh ./oecore-x86_64-armv7ahf-vfp-neon-toolchain-nodistro.0.sh
```

Depending on your setup, you may have to use the following command:

```
$ bash ./oecore-x86_64-armv7ahf-vfp-neon-toolchain-nodistro.0.sh
```

(If that's the case, you may want to run the sudo dpkg-configure dash command, see the PyBOMBS section). It will ask you where to install it to. Choose the prefix path (`~/prefix`). Unzipping and installing takes a while. After it's done, you should see something like this:

```
$ ls ~/prefix
environment-setup-armv7ahf-vfp-neon-oe-linux-gnueabi
site-config-armv7ahf-vfp-neon-oe-linux-gnueabi
sysroots/
usr/
version-armv7ahf-vfp-neon-oe-linux-gnueabi
```

(The exact contents of the directory may differ).

Before running any of the following commands, you need to set up the environment as outlined on the device manual. You need to source the environment file:

```
$ cd ~/prefix
$ source ./environment-setup-armv7ahf-vfp-neon-oe-linux-gnueabi
$ echo $CC # This is just to confirm it worked.
arm-oe-linux-gnueabi-gcc  -march=armv7-a -mfloat-abi=hard -mfpu=neon --sysroot=~/prefix/e300/sysroots/armv7ahf-vfp-neon-oe-linux-gnueabi
```

As you can see, it will change all kinds of paths to use different compilers and libraries. Note that during this shell session, you won't be able to run a lot of other things on your development machines, so you should dedicate a separate shell window for this.

Your development machine is now ready to cross-compile.

To cross-compile software, the SDK and the environment settings are usually not sufficient. Your software requires a toolchain setup at configuration time to build correctly. UHD brings support for this out of the box, so let's use UHD as an example on how to build custom software. First, let's make sure you have the source code for UHD in your prefix (it can be elsewhere, but for this example we'll keep *everything* inside the prefix directory).

```
$ cd ~/prefix
$ mkdir src/ # Let's create a src/ subdirectory which we'll use throughout the tutorial
$ cd src/
$ git clone https://github.com/EttusResearch/uhd.git
$ cd uhd/host
$ mkdir build
$ cd build
```

Now, before we do the next step, confirm that you have the correct environment loaded (see the step before). We now need to configure the build setup to cross-compile correctly.

```
$ cmake -DCMAKE_TOOLCHAIN_FILE=../host/cmake/Toolchains/oe-sdk_cross.cmake -DCMAKE_INSTALL_PREFIX=/usr -DENABLE_E300=ON ..
```

The exact command may differ (please consult [1]). However, two things are notable:

- We specify a toolchain file. This is how CMake knows how to set up a cross-compile, if the environment variables are set up correctly.
- Second, we specify the install prefix as `/usr`. This might seem odd, but it's actually the prefix *on the device*, not on our development machine.

Now that we have it all set up, we can compile:

```
$ make -j4
```

This step might take longer than you're used to -- that's because cross-compiling is slightly slower than native compiles. However, it's much, much faster than trying to compile on the device.

Installing is also slightly different than usual:

```
$ make install DESTDIR=~/prefix
```

The `DESTDIR` part will make sure it gets installed into the prefix, and doesn't try to actually install to `/usr/`.

The cleanest way to test your newly built UHD is to mount the prefix onto your embedded device. From your development machine, log onto the embedded device and mount the prefix:

```
$ ssh root@e3xx # After this, we're logged onto the embedded device
$ pwd # Check where we are
/home/root
$ mkdir usr # Create a directory to mount our prefix, this only needs doing once
$ sshfs dev@desktop:prefix/ usr/ # This will mount the prefix on the development machine to ~/usr
$ ls usr/ # The actual output of this may differ
environment-setup-armv7ahf-vfp-neon-oe-linux-gnueabi
site-config-armv7ahf-vfp-neon-oe-linux-gnueabi
sysroots/
usr/
version-armv7ahf-vfp-neon-oe-linux-gnueabi
```

The output of the final `ls` command should be the same as before when doing `ls ~/prefix` on the development machine. Nearly there! Now, we need to update the environment variables and paths in our ssh session on the embedded device. We will add a new file called `set_env` into the prefix with the following contents (if you prepared your development machine with PyBOMBS, this file should be already included):

```
LOCALPREFIX=~/usr
export PATH=$LOCALPREFIX/bin:$PATH
export LD_LOAD_LIBRARY=$LOCALPREFIX/lib:$LD_LOAD_LIBRARY
export LD_LIBRARY_PATH=$LOCALPREFIX/lib:$LD_LIBRARY_PATH
export PYTHONPATH=$LOCALPREFIX/lib/python2.7/site-packages:$PYTHONPATH
export PKG_CONFIG_PATH=$LOCALPREFIX/lib/pkgconfig:$PKG_CONFIG_PATH
export GRC_BLOCKS_PATH=$LOCALPREFIX/share/gnuradio/grc/blocks:$GRC_BLOCKS_PATH
export UHD_RFNOC_DIR=$LOCALPREFIX/share/uhd/rfnoc/
export UHD_IMAGES_DIR=$LOCALPREFIX/share/uhd/images
```

You can see the contents is very simple -- all it does is point the most important paths into our prefix. Going back to the SSH session on the embedded device, you should be able to source that file now:

```
$ which uhd_find_devices # This will point to the old UHD, which we do not want
/usr/bin/uhd_find_devices
$ source ./set_env
$ which uhd_find_devices # Now it should point to the new version
/home/root/usr/bin/uhd_find_devices
```

> NOTE: If you still see:
>
> ```
> $ which uhd_find_devices # This will point to the old UHD, which we do not want
> /usr/bin/uhd_find_devices
> ```
>
> be sure that you are pointing to the directory that contains folders such as bin/ and lib/. Under some configurations, you may want to modify the `set_env` from `LOCALPREFIX=~/usr` to `LOCALPREFIX=~/usr/usr`

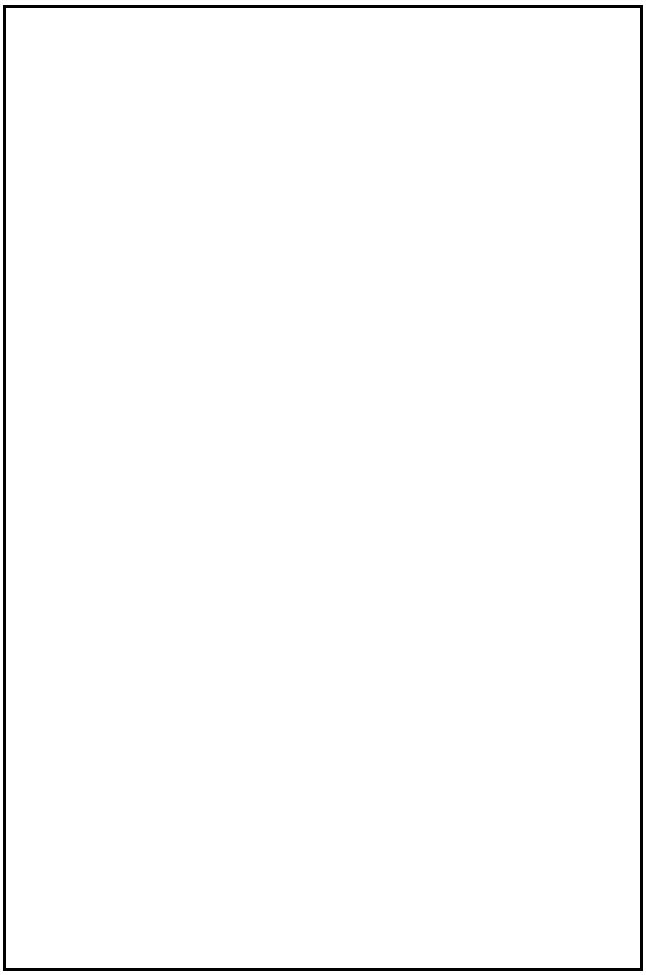
The last command is to make sure you're actually using the newly compiled UHD, because all of the SD card images already ship a version of UHD. In the example above, it worked, and we're ready to go!

The examples get installed to `lib/`, so if you want to try your newly compiled UHD on the device, you can do so by running, e.g., the following command:

```
$ ~/usr/lib/uhd/examples/benchmark_rate --tx_rate 1e6
```

Or anything else in the `examples` subdirectory.

At this point, it should be clear why the prefix/sshfs approach is so useful. Everything that is related to the install is stored in the same directory, you can zip it up and move it another computer if you like, and the embedded device doesn't actually store any of the development files. As soon as you unmount the sshfs mount, and reset the environment, it will be back in its default state.



Output example of CMakeGUI for the build of GNU Radio.
In principle, the work flow for GNU Radio is just the same as for UHD. However, GNU Radio depends on UHD, and you want it to use the new UHD instead of the one that ships with the device (and is in your SDK). There are two ways to go ahead: You could either also install UHD into your SDK, but the downside is that you then taint your SDK and it's a non-reversible operation.

A much easier way is to simply point your GNU Radio configuration to the new UHD libs instead of the old ones. Again, clone the repo into `~/prefix/src/gnuradio`, and then run the following command (again, you need to make sure the environment is set up correctly):

```
$ cd ~/prefix
$ source ./environment-setup-armv7ahf-vfp-neon-oe-linux-gnueabi # To be sure that our environment is correctly set up
$ cd src/
$ git clone --recursive https://github.com/gnuradio/gnuradio.git
$ cd gnuradio
$ mkdir build-arm
$ cd build-arm
$ cmake -Wno-dev -DCMAKE_TOOLCHAIN_FILE=../cmake/Toolchains/oe-sdk_cross.cmake -DCMAKE_INSTALL_PREFIX=/usr -DENABLE_GR_VOCODER=OFF -DENABL
$ make -j4 # This may take a while
$ make -j4 install DESTDIR=~/prefix
```

To confirm that this worked, it can help to open the CMake configuration in a GUI and check that the UHD-related variables point to the correct libraries. To see more detailed information about the compile, click the "Advanced" checkbox. By doing so, a several number of configuration options appear, which can be not only read but also modified. Notice that the path for UHD_includes and UHD_libraries should point to the UHD source folder installed in the step before. If they are still pointing to the UHD that comes with the device (which is [$PREFIX$}/sysroots/armv7ahf-vfp-neon-oe-linux-gnueabi/usr/lib/cmake/uhd) they should be modified by changing the path directly on the CMake GUI or by recompiling since cmake from terminal by adding the directives accordingly, as follows:

```
$ cmake -Wno-dev -DCMAKE_TOOLCHAIN_FILE=../cmake/Toolchains/oe-sdk_cross.cmake \-DCMAKE_INSTALL_PREFIX=/usr -DENABLE_GR_VOCODER=OFF -DENAB
```

Since the embedded devices usually don't have a screen, GNU Radio Companion (GRC) is not a useful tool on the device. However, the development machine can be used to generate flow graphs and compile to Python. Some best practices:

- Save both the .grc file and the generated .py file into the prefix
- Set the 'No GUI' option for Python files that are to be executed on the embedded device
- Run the Python scripts generated from GRC on the command line on the embedded device

- If you need visualization, run a separate flow graph on the development machine with the visualization GUI widgets. Use UDP sinks/sources or ZeroMQ blocks to pass data from the embedded device to the development machine.

Again, the process is very similar to building GNU Radio. You must make sure that all the correct libraries are linked to, and that it doesn't accidentally pick up libraries from the SDK (e.g. GNU Radio libraries).

```
# TODO add screenshot cmake-gui on how this should look like
```