DragonRadio

Sam Gallagher

The DragonRadio project contains:

- A C++ library containing glue-code, data structures, and network layer implementations
- A Python interface which can access the C++ library

This is fairly standard for modern projects, where Python is the preferred language but all the heavy lifting is done in C/C++. The C++ library is composed of:

- Data structures
  - RadioConfig (global radio configuration)
  - Header (PHY header specific to DragonRadio)
  - IQBuf (buffer for I/Q data
  - Packet
    - NetPacket
    - RadioPacket
  - ModPacket (modulated data packet, contains a NetPacket)
- Network layer-specific classes and methods
  - PHY (phy)
  - MAC (mac)
  - Network (net)
- Utilities
  - Real-time clock
  - Logger
  - Signal-processing (dsp, liquid)
  - Snapshot
  - Python bindings (python)
  - Statistical estimators (stats)

The interface for the user is shown as a block diagram, in Figure 1.

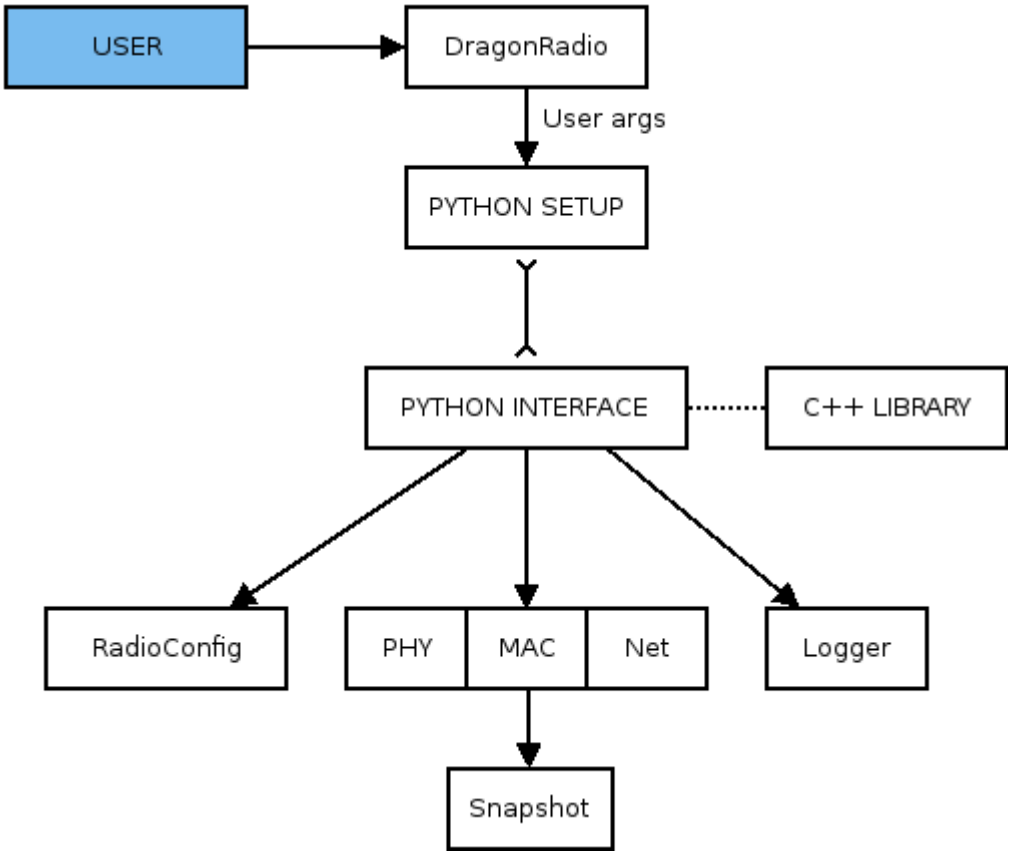

Figure 1. DragonRadio user interface block diagram

The classes which the Python interface exposes are shown in Figure 2.

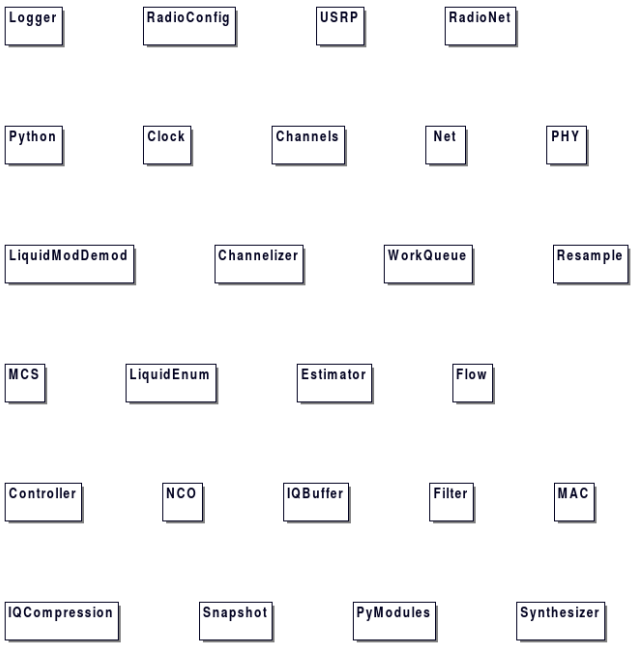

Figure 2. DragonRadio Python interface classes

To begin a radio session, the user runs the DragonRadio binary, passing a Python setup script as the first argument, and radio configuration arguments can follow. All of these configuration arguments are sent to the Python script to be processed. In the Python script, the radio is configured according to the arguments and the fixed parameters in the file.

The following libraries are seen in these Python scripts:

- asyncio – Support for concurrent code with async/await syntax, used for structured network code
- IPython – Interactive Python, now Jupyter, provides a standard interface for working with Python scripts
- signal – Set handlers for asynchronous events
- dragonradio – Python bindings for Dragonradio library
- dragon.radio – Located in dragon/radio.py, DragonRadio python interface

The python interface is stored in dragonradio/python/dragon, and contains: channels.py, collab.py, controller.py, gpsd.py, internal_pb2.py, internal.py, protobuf.py, radio.py, remote_pb2.py, remote.py, schedule.py, scoring.py, and signal.py. The dragon/ directory is a library.

Outside of the library we also have radio-client.py, which contains a main() method definition, and standalone-radio.py and sc2-radio.py, which run the actual radio as used in general command-line interfacing (standalone-radio) and competition (sc2-radio). ecet680-radio.py also defines a main() method.

I don't know what 'dragonradio' is, although *import dragonradio* is seen in most scripts. It must be part of a nonlocal library, because none of the files seemed to contain the string *def dragonradio* and no files are called dragonradio.

The file ecet680-radio.py can serve as a template for other configurations. To run DragonRadio, you can either pass arguments to configure the radio, or set defaults within the python script. It defines a few methods for managing MAC schedules (when devices transmit, for a fixed round-robin style protocol) and calculating the SNR. Then we get to main() and create a radio Config object which stores the tx_gain, rx_gain, auto_soft_tx_gain, bandwidth, and aloha_prob.

Then the parser handles the input arguments, some of which go to the Config object for basic handling, and the rest of which are handled by the setup script (MAC layer, number of nodes, interactivity). Then logging is set up, followed by the creation of the dragon.radio.Radio() object. At this point, the MAC layer is configured, and the IPython shell is started if the user requested interactivity.

Running the Radio with a Python Setup Script

The python script uses the asyncio module. It also uses the radio module from dragon (dragon.radio) and uses the dragonradio python bindings in dragonradio. Other things (os, sys, random, logging, IPython, signal) are used as utilities to interface with the system. Thus the imports at the start of any setup script should be:

```
import argparse
import asyncio
from concurrent.futures import CancelledError
import IPython
from itertools import chain, cycle, starmap
import logging
import os
import random
import signal
import sys

import dragonradio
import dragon.radio
```

So we need to know how to use some of these, namely asyncio (which handles the event loop), and possibly itertools. Then we'll need to have a reference for dragonradio and dragon.radio as well as other utilities in the dragon/ library.

Some preliminaries will make everything easier.

Event-Driven Programming

A quick note about event-driven programming is in order. When we are first introduced to programming, it is typically of a step-by-step program variety. The program starts up, it runs code in order, moves around in functions and loops and if/else statements, and eventually the program comes to an end and it exits. You can trace the program step-by-step, and by reading the source you can understand how the program works.

This is limited, in that the program needs all information required before it runs, and user input is only accepted at certain allotted times. We quickly realize as young programmers the magic of the infinite loop, the calculator which provides a set of options, like "Enter (1) for addition, (2) for subtraction, ..., and (-1) to exit."

In event-driven programming, the infinite loop concept is used. Every user action is received by the main event loop (sent by the operating system) and one of the program's responsibilities is performing required actions whenever a certain event has occurred. An "event" consists of e.g. a mouse button press with location and possibly other information about the environment. It can also be a program-specific event, like a startup event, or an exit event.

When an event occurs, the event loop emits a "signal", which is to say it calls all functions that have "connected" to the event of interest. A function, which is connected to a signal, will be called when that signal is "emitted". Such a function (whose purpose is to handle an event) is called variously a -callback-, or -slot-. The term -event handler- refers to the part of the event loop that uses incoming events from the OS, and decides which functions to call, and in what order.

Typically, callbacks are called in order of connection, or the order is determined by the place of the callback in an object hierarchy. The priority of a callback can also be overridden, so that the particularly important callbacks are always called first. This can be used e.g. to capture a mousepress on the X (exit) button in the window before the program closes.

Some distinctions:

- An **event** comes from the OS and represents the raw user input data. This is dealt with by the event-driven framework you're using, although you can capture these events on your own (though I don't know how exactly...). A **signal** represents a notification that a particular event has occurred, which is sent to the callback functions. We can make our own signals, make new signals, most classes in an event-driven framework will have a load of default signals for capturing button-presses etc. But we cannot "create" an event.
- The **event loop** is an infinite loop which "listens" for events from the operating system; it passes the events to the -event handler- for processing.

One drawback of event-driven programming is it makes debugging and reading the source much more difficult. Because we can connect signals anywhere in the program, once the program calls the event_loop's run() function, we don't have anything to follow. Normally, function calls are written explicitly, so the program might look like this:

```
main:
    do_a();
    do_b();
    var x;
    run_z();
    return 0;
```

Meanwhile, an event-driven program looks like this:

```
main:
    EventLoop main_event_loop;
    MyObj obj;
    main_event_loop.run();
    return 0;
MyObj.constructor:
    signal_mouse_press.connect(on_mouse_press)
MyObj.on_mouse_press:
    do_a();
```

You can imagine then that with a program spanning dozens or hundreds of files, finding out what happens e.g. on program startup is a matter of finding any connect() calls for the signal_startup signal anywhere in the program. This also assumes you know that there's a signal called signal_startup which is emitted on program startup.

Some useful events most frameworks have:

- Button press event
- Key press event
- Program startup event
- Timeout event (emits every x seconds according to a timer started by the user)
- Draw event (called whenever a window is redrawn, e.g. when it is moved, scaled, minimized and restored, etc)

Compiling C/C++

When we build C/C++ code like the DragonRadio library, the following happens as a bare minimum.

Assume we have 16 source (.cpp) files, and 15 header files which define prototypes for all the classes and functions in the source files. In the header files we can include other header files, and in source files we can include other header files. We cannot include source files in other source files.

1. All .cpp files are **compiled** into **object files**
    a. No header files are required, just source files
    b. The object files don't depend on each other, each can be compiled separately
2. The object files are **linked** together and a **binary** is generated
    a. All object files will now depend on their associated header files

b. Object files are filled with references to other functions etc. The object files don't know how to "follow" these references, but they assume the references exist (all functions are defined, all variables are declared, etc) somewhere

c. We need to link object files against **headers**, which tell the object files where to find all their references. If no header files were used, then linking will assume everything is available in the source files.

Libraries are bunches of object files (compiled source files) put into one master object file (the library). Libraries can thus be compiled, but they don't need to be linked into a binary, there's even no main() function. When we want to use a library, we only need to see the header files, which are not stored in the library file. They're stored as .h files e.g. in a lib/ directory on your computer. You include these headers in source files, then compile, and when it's time to link, you also throw in the headers and the library file with the source.

The headers look like this:

* example.h *

```
int myfunc(char x, float y);
class MyClass
{
public:
    MyClass();
    ~MyClass();
    double classfunc();
};
```

Note how none of the functions are defined, they're only prototyped. This is all we need to use the library in our program, because we assume the functionsdo what they say, and we read comments. We don't need the implementation to be shown. The implementation is stowed away in the library file.

When compiling, the object files will refer to myfunc() and MyClass, but they only assume that (a) they accept the parameters shown, and (b) they return the expected datatype. They have no way of knowing if the definitionof these functions even exists anywhere in the library. Only upon linking do we need to track down all the definitions.

Dragonradio is a C/C++ library, so it only defines a bunch of header files and source files which define classes, functions, etc. But there's nothing to actually "run" in the library, it doesn't have a main() function. We must use the library in a program which actually has a main() function.

Note: The main() function is part of every C/C++ program, and it is called the program entry point. Note that a program is very distinct from a library Dragonradio is a library. But there is a dragonradio binary generated which simply uses the library to run the desired python script in a proper environment.

Python Bindings

Python scripts cannot access C/C++ binaries or libraries directly, because of course C/C++ is not written in python. So how do we use C/C++ libraries in python?

The answer is -language bindings-, which allow us to bridge the gap between two separate languages. How does this work? One of the most common requirements is to use C/C++ in python and vice versa, and for this we can use pybind [https://github.com/pybind/pybind11].

We start in C/C++, include the pybind library, and we tell pybind which functions, classes, etc we are going to -export- for use in Python. Then when we compile and run, the bindings are generated in a module with the appropriate names. These bindings take the form of a binary which python recognizes as a module, so we can import the name of the binary, and we're done.

Here's a detailed example (from the docs):

```
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

    m.def("add", &add, "A function which adds two numbers");
}
```

When we build this and run it, a module named "example" is created, and it has defined a function called "add" with a docstring "A function which…" that can be used in python. When we're in the same directory as the binary, we can now run python and enter:

```
>>> import example
>>> example.add(26,47)
[1] 73
```

And that's it! For more basics of pybind11 see here.