Cardiff University Computer Science
CM3203 Final Report

Implementing a GNU Radio driver for the FLEX-6400 SDR Transceiver

by

Jonny Slim - C1634544

Supervisor: **David Humphries**
Client: **Derek Kozel**

May 2021

Contents

Introduction

Brief Summary

This project is entitled "Implementing a GNU Radio driver for the FLEX-6400 SDR Transceiver". The first step of this report breaks down this title into it's component parts and elucidates them. Firstly, GNU Radio is an open-source development toolkit[13], capable of developing and simulating real-world radio systems. It is a modular flow-based framework, in which predefined functions and operations are pieced together to develop specific communication systems. Its infrastructure is written in C++, but the user-created systems and tools can be written in Python, which is ideally suited for quick prototyping and adaptable approaches; given the time constraints of the project both these factors proved beneficial. The project would have required building with C++ if the data rates involved were high, but streaming from the radio is less than 1 Mb/s (48 kSample/s * 4 bytes/sample * 2 channels): easily processed by Python in real-time.

Figure 1 shows a simple example of how a GNU Radio program is built, where different

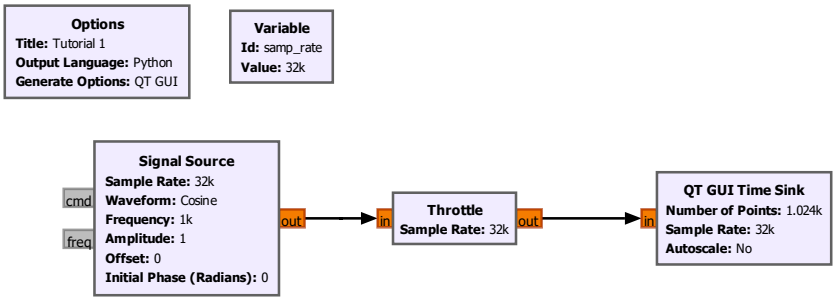


Figure 1: A simple GNU Radio program generating a cosine wave.

types of blocks are connected to form the creation and flow of data. The blocks used in this example are readily available, but GNU Radio also allows the user to create their own signal processing blocks - known as 'Out Of Tree Modules'. These could be in the form of new functionality or interfaces to external systems.
The idea of this project is twofold: 1 - to create a standalone component library allowing access to the FLEX-6400 radio from any Python-based application, and 2 - to create a FLEX block in GNU Radio which uses this library to provide an interface to the FLEX radio. This means that, instead of using a Signal Source created by the local machine, a block module provides data directly from the FLEX radio. It will also be possible to use the library to directly adjust the radio's parameters such as tuning frequency and demodulation type (eg. FM, AM, etc).

A Software Defined Radio is one in which components and calibrations usually defined in a radio's hardware are instead implemented with software. Most of the RF signal processing is performed using an API/Driver, allowing for greater flexibility in what channels and frequencies the radio is listening to, as well as easier modulation of the signal it's transmitting. SDRs can be reconfigured "on the fly", essentially completely transforming its purpose with plug-and-play programs able to run on demand.
Currently, the FLEX-6400 radio comes with an API (Fig. 3), but it is restrictive due to the fact it is built with the .NET framework. Building a new API with GNU Radio would enable cross-platform uses of the SDR – such as on embedded devices or Linux systems – opening many more opportunities to use its potential. Figure 2 shows FLEX's own User Interface, which is the standard way for a user to interact with the radio. This was used intermittently in the project to validate the data being received, but it would be beneficial to implement some of the visual aspects of it so the Python library has them available "out the box".
When creating an initial plan[27] for this project, clear goals and objectives were laid out that are expected to be met in order to call this project a success. They are as follows:

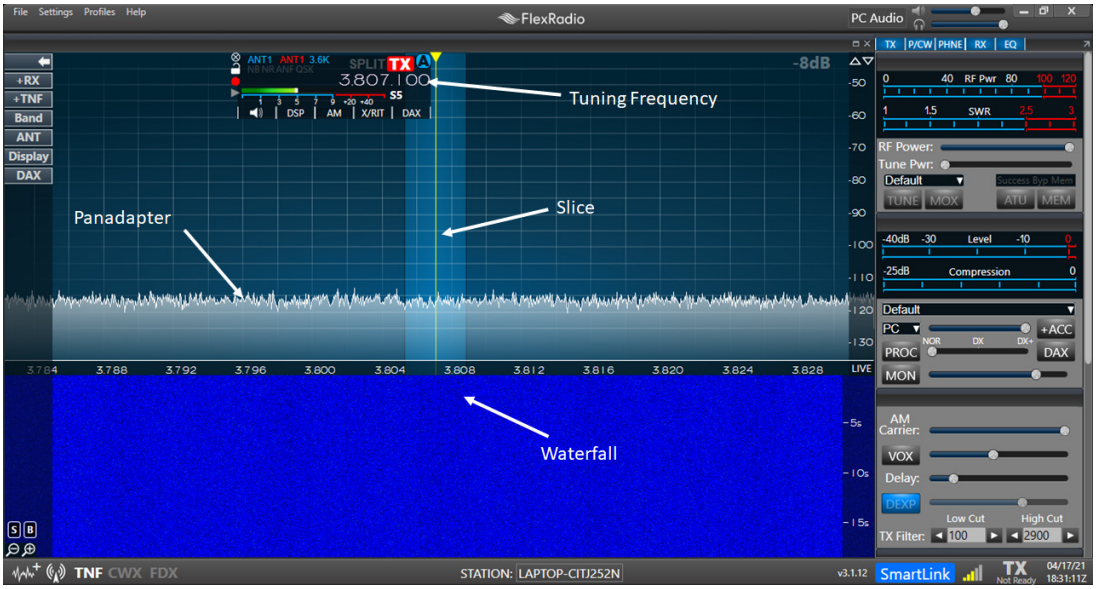


Figure 2: FlexRadio's SmartSDR .NET UI

- **Discover and Configure FLEX-6400** - Establish a connection with the radio and modify its settings. This is a fundamental part of the project as all interaction with a remote FLEX radio is done using network-based links; without completing this goal it will not be possible to complete the subsequent ones.

- **Receive Audio Data** - The next stage is requesting Audio Data from the SDR. To do so requires creating an audio stream on the radio and then receiving the VITA 49 formatted packets over a UDP connection, which is the protocol carrying the audio payload.

- **Receive IQ Data** - This is the logical next step after receiving audio data. Receiving IQ data involves more steps and the translation is more complex but completing this goal would make the project very useful to others in research or academia. IQ data represents an accurate capture of the raw radio waves, enabling demodulation with any mode (permitted by the width of the Slice).

- **Transmit Audio and IQ Data** - These two goals are extensions to the project that would be nice to achieve if the time permits. As transmitting on Amateur Radio frequencies requires a license (by passing an OFCOM exam[20]), this could

also affect how likely it is I complete these goals.

Motivation

Radio Communications and Amataur Radio have had a very large impact on development in the modern world. The ability to communicate rapidly across large distances furthers the speed with which important information can be spread, and in using this technology, operators discover improved ways of doing this communication.

In the pursuit of knowledge and discovery, amateur radio users have founded new industries[6] and contributed to the evolution and empowerment of communication in the developing world[5]. Amateur Radio has time and again come to the aid of first responders during times of emergency or disaster, like in South Asia[2] or Puerto Rico[11].

The use of Amateur Radio Frequencies is still alive and well, with many applications in research and academia. Clearly, the addition of available, versatile radio hardware is a benefit to such projects. The motivation behind this project is to allow non-Windows users to use a FLEX radio. It would also allow all users to interact with the radio using a well known Software Radio Modelling Toolkit: GNU Radio is very popular and well maintained, and the vast number of blocks available mean complex signal processing tasks can be completed with ease. It's a significant project because it has an aspect of novelty to it; this would be the only way for a FLEX radio to be controlled directly from within the GNU Radio user interface.

Making this project open-source also allows future developers the chance to extend it beyond what I achieve in these 13 weeks. Just as I have relied on the code and information supplied by others before me, it continues the possibility of development on this project or ones like it. From a personal perspective, SDRs and radio communication was something I had never encountered before and was quite alien to me, but attempting to do something new with little experience is a profitable skill to develop and one that would be relied on in the corporate world.

Background

The Wider Context

The idea of the project deliverable is to act as a translator and communicator between the two separate entities - GNU Radio and the FLEX-6400. This means that anyone wanting to use GNU Radio with high performance SDR hardware or who wants to use the FLEX-6400 on a system other than Windows would find great utility from my project. Most importantly, it is to provide the fundamental plumbing that would allow others to extend the project for their own objectives. If the basics of discovery, configuration and communication are available, a user will be able to do much of the standard radio interaction without any significant work. In addition, due to the class and data structures I employed, building anything more complex would be much quicker than having to start from scratch. As discussed later, there were many steps in this project where progress was hindered in trying to fix issues with which there was very little public information and I had to rely on the contacts of my supervisors to make progress.

Due to the sparse nature of FLEX API documentation, this report could be extremely informative to anyone attempting to do something similar. I had to comb through a lot of community posts and organise meetings with FLEX developers throughout my project to truly understand the FLEX radio and how it communicates. Having accumulated a lot of that information here, it will not only help someone using my library but also advise a developer who wanted to create an API in another language or for another Radio Toolkit.

There are numerous projects currently proposed or in the design phase that could heavily benefit from this one. The Ham Radio Science Citizen Investigation are creating a "Personal Space Weather Station"; a multi-instrument system capable of making ground-based measurements of the space environment. Referring to the architecture for the PSWS[10], it looks very similar to my project, with a SDR communicating with a Computer. The radio is obtaining, among others, IQ data which would then be displayed and sent to a central server. This is very similar to what my library will be doing, with the API requesting IQ data which can then be presented by GNU Radio. As the author mentions in the presentation, the PSWS is receive only. Theoretically, the library produced by this project could therefore be implemented directly into the PSWS.

Another such example is the Weak Signal Propagation Reporter or WSPR network. Radio Waves in the range of 3-30 MHz propagate around the Earth by bouncing from the atmosphere and the Earth's surface[18]. Due to the variance in atmospheric conditions, the ability of radio waves to reach destinations can vary wildly. WSPR allows people to see a real-time network of available connections between 2 points on Earth. This project could be used as another node in the network, receiving and transmitting to other nodes to establish possible links. Through the use of this project's API this process could be automated, therefore constantly updating the WSPR network.

Additionally, an experiment detecting how radio waves are altered when passing through Earth's ionosphere during a solar eclipse - conducted by William Lloyd[16] - could also be another example. Here the author streams IQ data[4] from a different make of SDR, but it would benefit from the greater transmit ability of the FLEX-6400. It also provides a nice example of how the FLEX can be plugged straight into an experiment provided it has the drivers I plan to produce in this project.

Finally, FreeDV[30] is a Digital Voice demodulation service which can run on any HF radio. It has already been implemented as a GNU Radio block[12] so could be connected to the FlexSource block produced from this project to demodulate Digital Voice data received on the FLEX-6400. Digital Voice modes encode speech into a data stream be-

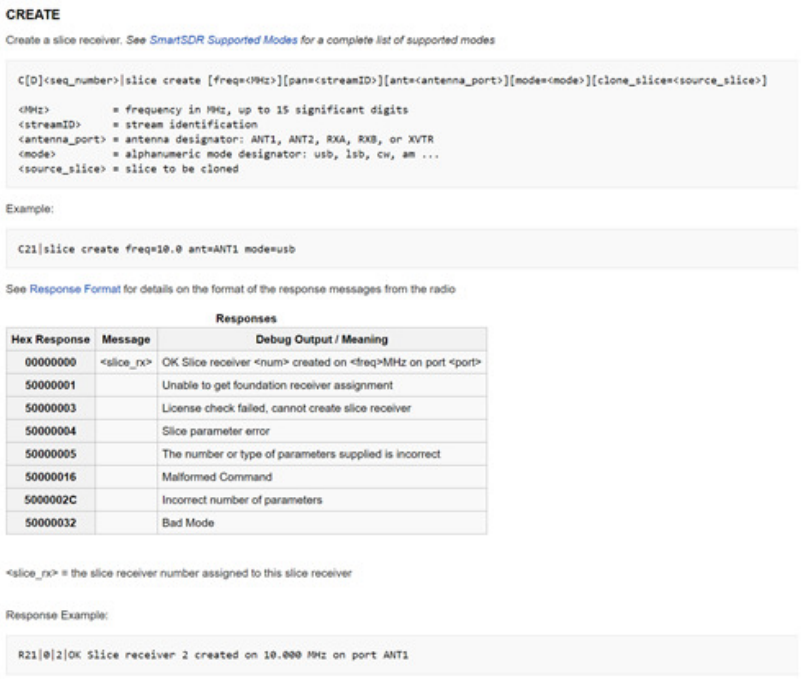
fore transmitting it. They are able to provide much better spectral and power efficiency levels than Analog modes and can often be received and understood (or 'copied', in radio communications terminology) at much lower signal levels.

Sources of Relevant Information

A key aspect of the project's development was to research existing implementations/applications which made use of the FLEX API. I came across a few instances of people who had created partial or otherwise restrictive APIs. This GitHub Repo[9] is an implementation of how to communicate with a FLEX radio using Windows Powershell commands. I found this very useful in the initial stages, as actually creating the connection with the radio was much more complex than I had first appreciated. While I couldn't use the exact commands the author used, I could see the general process that needed to be emulated. As you can see from my Initial Plan[27], I expected the authentication and connection to the radio to be fairly trivial but it proved otherwise. In particular, file `Connect-FlexSmartLink.ps1` was really informative in helping me build the required flow (Fig. 8).

It revealed the exact URI I needed to visit to authenticate myself, including parameters required like 'client_id' and 'state'. It also told me that I needed a JSON Web Token to register my application with the SmartLink Server (this will be discussed in greater detail in the Implementation section). This Powershell implementation while informative was far from complete, and not usable by anyone not wanting to use an API employing Powershell commands.

Another valuable piece of information I drew from was the SmartSDR wiki[7] created by the FLEX team. While this source isn't particularly well maintained and may contain some inaccuracies (the API has been updated since it was created), for the majority it was a very useful compass when finding what commands I needed to send to achieve an action and what response I should expect from the radio. For example Figure 3 shows



CREATE

Create a slice receiver. See SmartSDR Supported Modes for a complete list of supported modes

```
C[D]<seq_number>|slice create [freq=<MHz>][pan=<streamID>][ant=<antenna_port>][mode=<mode>][clone_slice=<source_slice>]

<MHz>          = frequency in MHz, up to 15 significant digits
<streamID>     = stream identification
<antenna_port> = antenna designator: ANT1, ANT2, RXA, RXB, or XVTR
<mode>         = alphanumeric mode designator: usb, lsb, cw, am ...
<source_slice> = slice to be cloned
```

Example:

```
C21|slice create freq=10.0 ant=ANT1 mode=usb
```

See Response Format for details on the format of the response messages from the radio

Responses

Hex Response	Message	Debug Output / Meaning
00000000	<slice_rx>	OK Slice receiver <num> created on <freq>MHz on port <port>
50000001		Unable to get foundation receiver assignment
50000003		License check failed, cannot create slice receiver
50000004		Slice parameter error
50000005		The number or type of parameters supplied is incorrect
50000016		Malformed Command
5000002C		Incorrect number of parameters
50000032		Bad Mode

<slice_rx> = the slice receiver number assigned to this slice receiver

Response Example:

```
R21|0|2|OK Slice receiver 2 created on 10.000 MHz on port ANT1
```

Figure 3: FLEX API wiki on Slice creation

how to request a new Slice (see next section) from the radio.

On the FLEX Radio Community Forum, a really useful thread was made by John Linford discussing how to make your own API for a FLEX radio. From this he authored a document called "A FLEX 6000 API Primer"[15] in which he lays out some high level information on the steps to take in order to build an API to a FLEX radio. Unsurprisingly this was also very helpful, in particular when it came to translating the UDP data packets received from the radio. While I only found this document after I'd completed much of the Discovery and Configuration work, it also helped me validate the steps I'd taken. As stated previously, reliable and correct information was hard to come by. However, it should be noted that this API Primer is limited in its scope, and as the name suggests it only scratches the surface of the capabilities that a potential API might have. For example, on Page 27, the author writes: *"Here be dragons territory" for me includes the audio and panadapter streaming data. I've not even looked at these data streams yet, so I can offer no advice at all.* He continues by saying that his main goal for his API was to allow finer control of the FLEX SDR using custom hardware, such as a tuner control

knob. Audio and IQ data streams are the main goals of this project, so unfortunately this document could only take me so far.
In summation, no reliable sources which fully describe interfacing with the FLEX radio exist, but that is another advantage of completing this project. Having combined all the research necessary to create my library, I was able to make an accurate and trust-worthy instrument to allow developers and radio-enthusiasts to understand the FLEX radio and API. In turn this will benefit a wide variety of applications and research projects.

Helpful Background on Amateur Radio and SDRs

When I started this project, I had zero experience in Amateur Radios and the only Digital Signal Processing I'd learnt was in the second year Scientific Computing module. While a well-versed software engineer could pick up the nomenclature of Ham and DSP fairly easily, I thought I should add a brief section on some terms that I will use frequently throughout this report. Refer to Figure 2 for a visual representation of some of these terms.

- **Amateur Radio** - A non-commerical radio service as set by a recognised Government Agency (Ofcom in the UK)
- **Carrier Wave** - A pure continuous radio wave at a fixed frequency, which is then imposed on with the input signal
- **Center Frequency** - The frequency a radio is tuned to in order to receive or transmit a specific signal. The "Bandwidth" of this signal is upper and lower cutoff frequency points
- **Modulation** - The way the shape of the carrier wave is changed to encode the data being transmitted e.g FM being Frequency Modulation [17]
- **Panadapter** - A 2D graphical representation of the signal amplitude against the frequency, which plots in real-time to display the received energy in any given period

- **Propagation** - The means or path by which a radio signal travels from a transmitting station to a receiving station
- **RX** - Shorthand for Receive or Receiver
- **Slice** - A portion of the radio spectrum to receive, centered at a specific frequency
- **Transceiver** - A radio that both transmits and receives
- **TX** - Shorthand for Transmit or Transmitter
- **Waterfall** - The Fourier Transform of a signal, plotting the time a signal is active against the frequency

Approach

This project has a strong likelihood of being used - and extended - by a great many users including researchers, ham radio operators and the project's client. Given this, I wanted to ensure that the system was as robust as possible. This is because building functionality on top of a strong foundation of code is much easier than having to rewrite many sections in order to integrate changes. The creation of a standard framework, with my carefully-chosen structure means a future developer will be able to extend it by simply adding methods. Writing a program in this way is a skill that translates directly into a professional, industrial environment so will also be beneficial to me personally.

Architecture

With all this in consideration, the architecture of the project is shown in Figure 4, with captions in red for areas I was responsible for. The FlexModule API communicates with

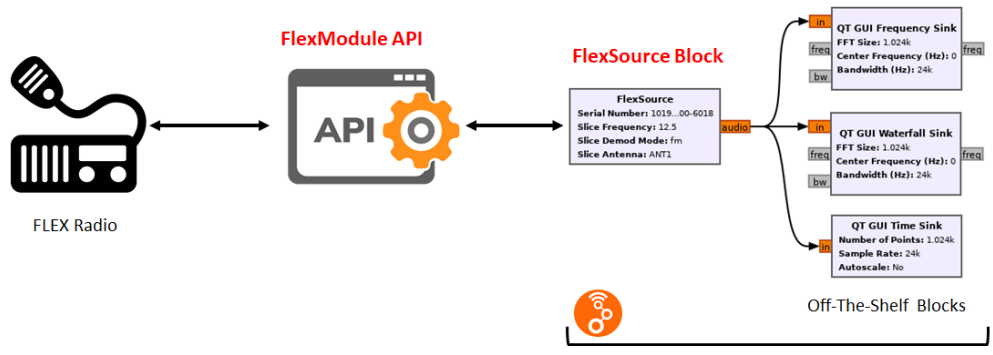


Figure 4: High-level Diagram of Program Architecture

the radio: querying status info and storing appropriate updates, sending commands and receiving responses, and creating audio streams/slices/pan adapters. This "administrative" communication is sent over a TCP/IP connection. A UDP connection between the two is also present, where raw data like the audio itself is transferred.
In effect, the FlexModule API I have developed provides a translation service between the FLEX-6400 and GNU Radio, handling all the data the radio outputs and only presenting the user with what is requested. The `FlexSource` Block is how the user interacts with the API. It enables requests for the creation or alteration of radio objects (Slices, Panadapters etc.), and can then allow for manipulation of the data that is returned from said objects. In the aforementioned architecture diagram, audio data is passed into three types of visualisation "sinks" or endpoints (however band-pass filtering or other signal processing could take place here).

The API uses classes as its underlying structuring, with each class encapsulating and abstracting all the detail that isn't necessary on the GNU Radio side. Figure 5 shows a slightly cut-down Class Diagram of the API, demonstrating my object-based approach.

Lets take `FlexModule\Radio.py` as an example:

```
6  class Radio(object):
7    cmdCnt = 0
8    """ Class to create connection with FLEX radio and establish communication channel """
9    def __init__(self, radioData, smartlink):
10     self.ResponseList = {}
11     self.StatusList = []
12     self.AntList = []
13     self.SliceList = [Slice(self, 0, "ANT1", "fm")] # FLEX has a default slice on start
       up
14     self.Panafall = Panafall(self, "0x40000000", "0x42000000", 0, 50, 20) # FLEX also has
        a default Panafall
15     self.RxAudioStreamer = None
16
```

Listing 1: Radio.py Class Example

Listing 1 introduces part of Radio.py's initialisation code; a Radio() object must be created each time a user wishes to connect to a radio. As this initialisation is defined as part of a class, we can create multiple Radio() objects and establish connections to multiple radios simultaneously, with a user sending commands to each separately. This multiple

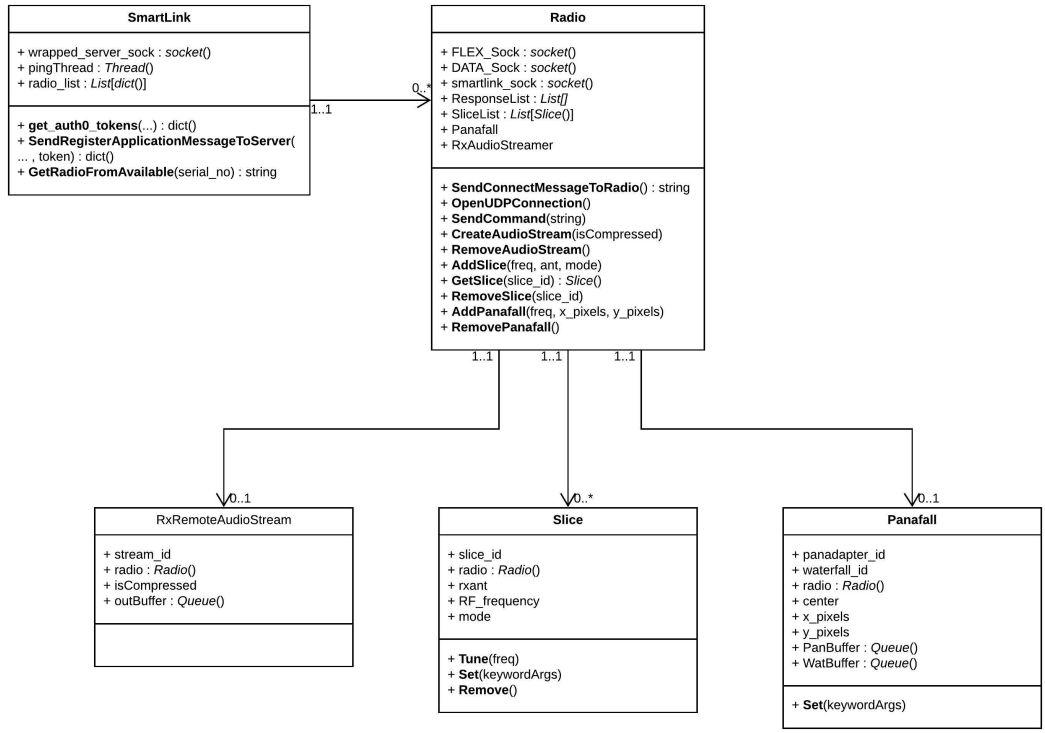


Figure 5: Pruned Class Diagram of my FlexModule API

instantiation is very advantageous for other classes as well. Looking at line 13 in Listing 1, we can see that the radio is initialised with a SliceList, already containing a Slice() object. However, we can create additional slices should we desire, which would be stored in this list. All this creation can be done separately from the Radio() class, with the exception that the command needs to be sent to the physical FLEX radio. Therefore we can keep the Radio class and Radio Object classes detached, so the radio only needs to store the references to the slices it owns, not all the detailed information about the slice itself.

Listing 2 shows the continuation of this concept. All the relevant variables and methods for the Slice are stored in this class, and only a reference to the Radio object is needed for consistency when the Slice is created.

```
4  class Slice(object):
5    """ A Class to create, remove and alter radio frequency slices """
6    Id_iter = itertools.count() # slice_id needs to be a unique attribute for each slice
7
8    def __init__(self, radio, freq, ant, mode):
9      self.slice_id = next(self.Id_iter)
10     self.radio = radio
11
```

```
12      if freq < 0.03:
13        self.RF_frequency = 0.03
14        # log attempt to set below min
15      elif freq > 54.0:
16        self.RF_frequency = 54.0
17        # log attempt to set above max
18      else:
19        self.RF_frequency = freq
20
21      self.rxant = ant
22
23      self.mode = mode.upper()
24
25
26    def Remove(self):
27      command = "slice r " + str(self.slice_id)
28      self.radio.SendCommand(command)
29
30
31    def Tune(self, freq):
32      if freq < 0.03:
33        self.RF_frequency = 0.03
34        # log attempt to set below min
35      elif freq > 54.0:
36        self.RF_frequency = 54.0
37        # log attempt to set above max
38      else:
39        self.RF_frequency = freq
40
41      command = "slice t " + str(self.slice_id) + " " + str(self.RF_frequency)
42      self.radio.SendCommand(command)
43
```

Listing 2: Slice.py Class example

The structure of a GNU Radio block has to conform to the standard framework, with slight variance depending on its type. This GNU Radio document[14] describes the different types of blocks available, and the corresponding code snippets explain how they must be built in order to work.

The biggest takeaway is the following, which describes the ratio of number of inputs per port to number of outputs per port:

- **Synchronous Blocks** (1:1)
- **Decimation Blocks** (N:1)
- **Interpolation Blocks** (1:M)
- **Basic (a.k.a. General) Blocks** (N:M)

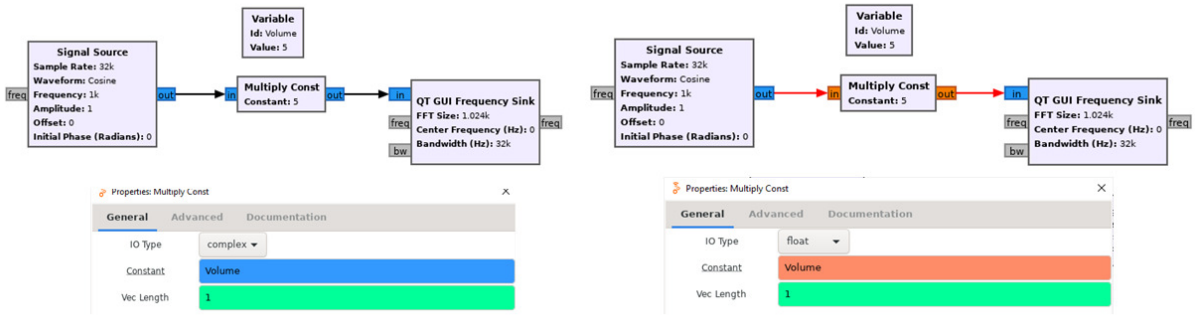


Figure 6: GNU Radio Block Architecture

This means that, for example, a Decimation Block takes more items in than it outputs, by a ratio of N:1. If you wanted to down-sample a signal, then you would use a Decimation Block.

Every block inherits from one of these four types, but the signal processing performed inside it is pretty limitless. Almost anything achieved in a regular Python script can be run in a Out-Of-Tree GNU Radio Block. Every GNU Radio block must also have a 'work()' function, which is where the actual signal processing takes place. One of the most simple blocks is the 'Multiply Const' block, in which the work function would simply multiply each input value by a constant number. This would be an example of a synchronous block as the number of inputs = the number of outputs. Figure 6 shows a 'Multiply Const' block in action, where I'm using a variable called "Volume" as the multiplier. This figure also shows another important aspect of GNU Radio: data type consistency. In the right-hand portion of the diagram, the red arrows are indicating an error in the flow. This is occurring because the Signal Source block is outputting complex data, but the Multiply Const block is expecting data with type float.

Up to this point I have mentioned two types of blocks that aren't explicitly described in the Types of Block web-page - Source and Sink - but note this line: "When a sync block has zero inputs, its called a source. When a sync block has zero outputs, its called a sink"[14].

In summation, My `FlexSource` block will inherit from a Sync block, and import my

FlexModule Python library. It will then leverage this API to communicate with the FLEX radio and output the audio/IQ data received, which can then be manipulated in any way GNU Radio allows. Not only this, but the project will be made into a package, capable of being installed using Python's Package Installer 'pip'.

Benefits & Drawbacks

Using an object-based approach to the project affords me a few advantages when it comes to the implementation and usability of the software. A non-object orientated approach would mean that the API would be a long list of commands, functions and subroutines. In order for there to be interaction between functionality and data, it's common for these commands and variables to be accessible from any part of the program. However as the library grows, allowing any function to modify any piece of data would make the program very unstable and bugs to be very pervasive.

Instead, using classes means I have control over what data is accessible and modifiable by the rest of the API. Consequently, the code is more maintainable as it's easier to identify the source of errors and bugs tend to be self-contained. Classes also mean an object can be used in any context, making it a perfect fit for plugging into another program such as GNU Radio.

From a potential developer's point of view, this approach also improves the scalability of the API. A user wanting to extend the API can add classes independently from the rest of the program without worrying about affecting it. And from a broader point of view, building my library using Python makes the API more portable as it can be used on a variety of platforms and systems. Having it as an installable package also means that it is very easy to access in any Python environment on a local machine.

One notable drawback of my approach is that I'm scripting with Python. Although it allows faster prototyping and testing (advantageous for a short term project) the high-level nature of the language means it is inherently slower than others. A library written

in C/C++ would be able to process incoming data much faster which is valuable for a real-time application such as this.

Implementation

In this section of my report, I would like to expand upon four key areas of my project that deserve specific attention. Each area had notable time devoted to it, and either is fundamental in its contribution to the project or presented challenging roadblocks. Sometimes both.

Authorisation & Authentication

The first key area is the steps required to Authenticate a user and connect to the FLEX radio. This was the entry point to the project, and in my Initial Plan I expected it to take three weeks to complete the authentication and setting of radio parameters. I was partly correct in this estimate; while adjusting radio parameters turned out to be fairly straight forward, the authentication was much more complicated. This was in part due to the lack of reliable documentation outlining the steps, so myself and my supervisors had to pull information from multiple sources and patch together a working flow (Fig. 8).

FLEX Radios use a well known Authentication Service called Auth0. This wasn't immediately obvious, but looking at the SmartLink Quick Start Guide[29], on page 7 we can see a SmartLink Login prompt. Underneath this prompt is a tag saying "Protected with Auth0", so this was the place to build from. Figure 7 is taken from Auth0's website[3], and describes the process to obtain an Auth0 access token (a type of JSON Web Token or jwt). This flow is exactly what I needed to authenticate myself with FLEX's SmartLink server, which holds the records of which radios each user has access to. The only difference

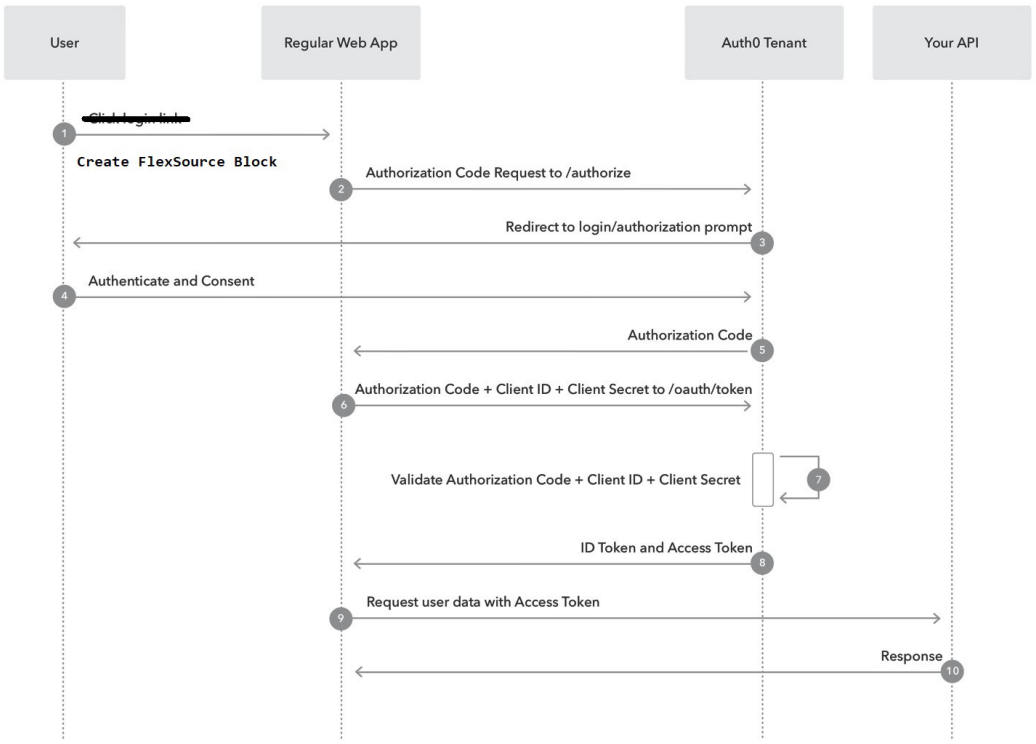


Figure 7: Required flow to Receive an Access Token

is instead of clicking a Login button, the authentication process starts from my FlexSource GNU Radio block.

We begin our authentication with

```
token_data = self.get_auth0_tokens( self.HOST_Auth, self.CLIENT_ID,
self.REDIRECT_URI, self.SCOPE_LIST, self.BROWSER )
```

a function authored by my supervisor David Humphreys, which is emulating the Auth0 Authorization Flow (Listing 3). We start by creating a HTTPS connection to Auth0's server, asking for an Authorization Code. The server redirects us to a login page where we enter the account details associated with our FLEX-6400. To be able to enter the details, we must open up a browser and type them in manually (lines 81-98 Listing 3). My supervisors and I tried to get information from the FLEX team about whether there was a non-browser alternative, but we didn't receive anything concrete and I didn't want to get bogged down at such an early stage. This did cause quite a few problems later in

the project but I'll discuss these in a following section.

Once user details have been entered correctly we extract the Authorization Code from the URI we are redirected to and then, along with our client id and some other information, this code is used to perform an HTTP request for an Access Token. Once this token has been received, the Auth0 flow has been completed and we are ready to discover our radio.

```
52 # Takes a hostname as input, and attempts auth0 authentication using a web browser.
53 #  The browser is set to Firefox() currently, but can be any which the Selenium module
       supports (e.g. Chrome()).
54 #  The output is None for an unsuccessful login, or the response dictionary for a
       successful one.
55 #  The token required by smartlink.flexradio.com is stored under the key "id_token".
56 def get_auth0_tokens(self, host, client_id, redirect_uri, scope_list, browser):
57   """ Author - David Humphreys """
58
59   """ to hide non-harmful error """
60   options = webdriver.ChromeOptions()
61   options.add_experimental_option('excludeSwitches', ['enable-logging'])
62   """ """
63   browsers = { 'chrome' : webdriver.Chrome(options=options, executable_path=r"C:\Program
       Files\chromedriver_win32\chromedriver.exe")} #,'firefox' : webdriver.Firefox(
       executable_path=GeckoDriverManager().install()) }
64   scope = "%20".join( scope_list )
65   state_len = 16
66   state = "".join( choices( ascii_letters + digits, k = state_len ) )       # was "
       ypfolheqwpezrxdb" when testing
67
68   conn = http.client.HTTPSConnection( host )
69   # print(conn)
70   # Step 1: request an auth0 code
71   #   (this seems to return a redirect to a login URL)
72   url1 = "/authorize"
73   payload1 = "response_type=code&client_id=" + client_id + "&redirect_uri=" +
       redirect_uri + "&scope=" + scope + "&state=" + state
74   # print(url1 + "?" + payload1)
75   conn.request( "GET", url1 + "?" + payload1 )
76   response = self.get_response( conn )
77   #print( response )
78
79
80   # Step 2: open a browser, and display the login URL
81   rstr = "Found. Redirecting to "
82   if response.find( rstr ) != -1:
83     url2 = response.split( rstr )[ 1 ]
84     driver = browsers[ browser ]
85     url2 = "https://" + host + url2
86     driver.get( url2 )
87   else:
88     print( "ERROR: request for authorisation did not return a valid login URL" )
89     return
90
91
92   # Step 3: wait for the URL in the browser to change (i.e. the user has entered their
       login information, hopefully correctly!),
```

```
93  #   and then close the browser
94  response = driver.current_url
95  while( response == url2 ):
96    sleep( 1 )
97    response = driver.current_url
98  driver.close()
99
100
101 # Step 4: attempt to extract the auth0 code from the URL the browser was directed to,
102 #   and use if to request (finally!) the id_token needed to register with smartlink.
      flexlib.com
103 rstr = "code="
104 if response.find( rstr ) != -1:
105   code = response.split( rstr )[ 1 ]
106   url3 = "/frtest.auth0.com/oauth/token"
107   headers3 = { 'content-type': "application/x-www-form-urlencoded" }
108   payload3 = "response_type=token&client_id=" + client_id + "&redirect_uri=" +
      redirect_uri + "&scope=" + scope + "&state=" + state + "&grant_type=
      authorization_code&code=" + code
109   conn.request( "POST", url3, payload3, headers3 )
110   response = self.get_response( conn )
111   #print( response )
112 else:
113   print( "ERROR: code was not returned during the login attempt; was your login
      incorrect?" )
114   return
115
116
117   # Step 5: attempt to extract the token data (in particular, id_token) from the auth0
      server's response
118 rstr = '"id_token":"'
119 if response.find( rstr ) != -1:
120   response = loads( response )
121   # print( "id_token is:", response[ "id_token" ] )
122   return response
123 else:
124   print( "ERROR: id_token was not returned by the auth0 server" )
125   return
126
```

Listing 3: get_auth0_tokens()

The next step is to register our application with SmartLink. This allows us to see all our available radios, and enables direct connection to them. The process is started with

```
self.radio_list = self.SendRegisterApplicationMessageToServer(
"FlexModule", self.OS, token_data['id_token'])
```

where we are using the token we received from Auth0. In this method, we check a TLS socket to the SmartLink server created in the class initialisation is open, and then send the "application register" command. We then start receiving data about our account, our local IP details and importantly, a radio list of all our available radios (Listing 4). This radio data is stored in a class variable and when we create a Radio() object, we pass the SmartLink() class as a parameter so that this data is available to us.

```
134 def SendRegisterApplicationMessageToServer(self, appName, platform, token):
135   command = "application register name=" + appName + " platform=" + platform + " token="
        + token + '\n'
136   radioData = []
137   if self.wrapped_server_sock.version() != None:
138     # print(self.wrapped_server_sock.version())
139     print(command)
140     self.wrapped_server_sock.send(command.encode("cp1252"))
141     """ Communicate with SmartLink Server """
142     inputs = [self.wrapped_server_sock]
143     while inputs:
144       readable, writable, exceptional = select.select(inputs, [], [], 2)
145       # pdb.set_trace()
146
147       for s in readable:
148         data = s.recv(1024).decode("utf-8")
149         print(data)
150         if "radio_name" in data:
151           radioData.append(self.ParseRadios(data))
152         # else:
153         #   """ Never gets here as no longer any sockets in readable """
154         #   inputs.remove(s)
155       if len(readable) < 1:
156         """ no sockets are readable so must escape loop """
157         inputs.clear()
158
159   else:
160     print("Socket connection not established....")
161
162   return radioData
163
```

Listing 4: SendRegisterApplicationMessageToServer()

The final step is connecting to our desired radio. As I previously stated, we do so

by creating a Radio() object with the info of a specific radio. During the initialisation of this object the function `SendConnectMessageToRadio()` is called, where we tell the SmartLink server we intend to connect to radio R on port P (Listing 5). This port will become our Peer-To-Peer TCP/IP port once the connection has been established.

```
36 def SendConnectMessageToRadio(self):
37   try:
38     command = "application connect serial=" + self.radioData['serial'] + "
       hole_punch_port=" + str(self.radioData['public_upnp_tls_port']) + "\n"
39   except TypeError:
40     print("Radio Serial not returned - is radio On?")
41     return
42   print("\nSending connect message: " + command)
43   self.smartlink_sock.send(command.encode("cp1252"))
44   handle_data = self.smartlink_sock.recv(128).decode("cp1252")
45   # print(handle_data)
46   try:
47     handle = handle_data.split('handle=')[1].strip()
48     return handle
49   except IndexError:
50     print("Server Handle not received")
51     return ""
52
```

Listing 5: SendConnectMessageToRadio()

If all is well, the SmartLink Server replies with a unique session handle. Once this is received, we create a TCP/IP connection to the radio directly; the IP address and Port number to connect on are included in the radio data we obtained in the SendRegisterApplicationMessageToServer() step. Finally, we inform the radio that we are communicating over a Wide Area Network and use the session handle to confirm our identity (Listing 6).

```
54 def WanValidate(self):
55   command = "wan validate handle=" + self.serverHandle + "\n"
56   print("\nSending Wan Validate command: " + command + "\n")
57   self.SendCommand(command)
58
```

Listing 6: WanValidate()

At the conclusion of these steps, we have established a Peer-To-Peer connection with the radio and can begin to send commands and receive responses. This process directly recreates the functionality described in the SmartLink Quick Start Guide[29] and is the absolute foundation for my project. With the Authentication and Authorization done, I could then start building my API and create the functionality required to achieve the rest

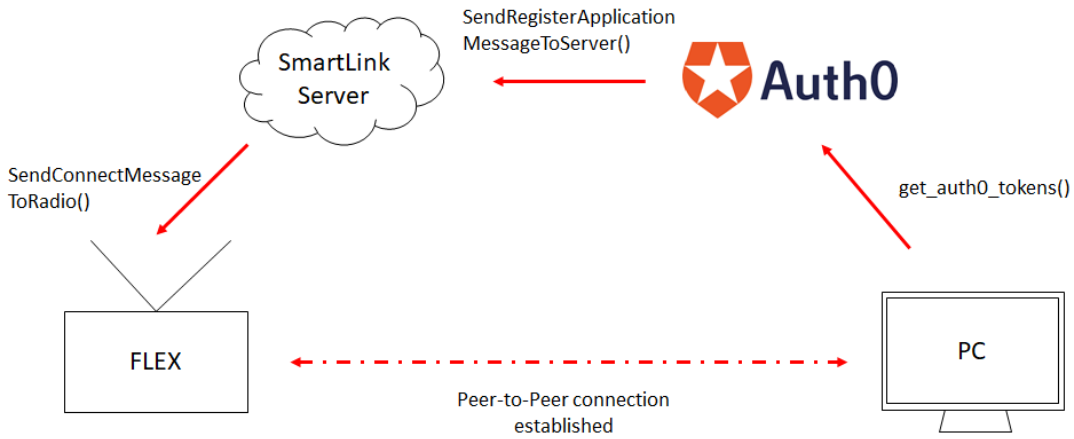


Figure 8: Required flow to Connect to a Radio

of my goals.

Data Handler

Because of the middle-man characteristic of this library - being the link between the FLEX-6400 and GNU Radio - having a robust Data Handler was key to the success of the project. As a consequence, this was another area I spent an extended period of time on. To reiterate what I have previously mentioned, data from the radio comes in two forms: TCP/IP data and UDP data. The former contains all information about command-response, subscription and update messages to and from the FLEX-6400. The latter is how all RF data is transferred, like audio, IQ, panadapter and waterfall data.

`FlexModule\DataHandler.py` is how I implemented this middle-man. Listing 7 shows how data is actually received from the radio:

```
8  class ReceiveData(threading.Thread):
9    """ Thread to contiually receive tcp data in BG """
10   def __init__(self, radio):
11     threading.Thread.__init__(self, daemon=True)
12     self.radio = radio
13     self.running = True
14     # self.read_socks = []
15
16   def run(self):
17     read_socks = [self.radio.FLEX_Sock, self.radio.DATA_Sock]
18     tcpResponse = ""
19     udpResponse = ""
20     while read_socks:
21       readable, w, e = select.select(read_socks,[],[],0)
22       for s in readable:
23         if s.type == 1: # "SOCK_STREAM"
24           data = s.recv(512).decode("cp1252")
25           tcpResponse += data
26           if data.endswith("\n"):
27             ParseRead(self.radio, tcpResponse.rstrip())
28             tcpResponse = ""
29         elif s.type == 2: # "SOCK_DGRAM"
30           if self.radio.UdpListening:
31             udpResponse, addr = s.recvfrom(8192)
32             # print(udpResponse)
33             ParseVitaPacket(self.radio, VitaPacket(udpResponse))
34
35         # if not data:
36         #   read_socks.remove(s)
37       if not self.running:
38         read_socks.clear()
```

Listing 7: Receive Thread

As the API will be called from GNU Radio, we can't be waiting for data to be received

and parsed, as this would cause the UI to freeze. Therefore the Data Handler must be threaded and running in the background. We initialise with the Radio() as a parameter to allow us to update local information about it, as well as ensuring we know which sockets we will be receiving data from.

Once the thread is running, I use a Python module called 'select'[23], which performs non-blocking polls of all the sockets in the 3 lists provided (Line 21). As this is a receive-only thread, I have no need providing write or exception sockets. Returned is a list of all sockets with data available to be read.

If the data received is from a Stream Socket (TCP/IP), I concatenate the data until I receive a newline character. This is as a consequence of some of the subscription updates being very long, and we want to keep them as one message. Once that check is passed, we then send the full message along with the radio it came from to a parsing function and reset the message buffer.

If we have just received data from a Datagram Socket (UDP), we receive all the data at once, but also hand it to a parsing function with it's corresponding radio. This is acceptable because I know the max sizes of the UDP packets I'll be receiving and can guarantee they will be less than 8192 bytes. Each packet received on the Datagram Socket is using the VITA 49 packet format, so accordingly I load it into a VitaPacket() class before handing it to the parser (I will elaborate on the VITA 49 format shortly).

Continuing with TCP/IP messages, an if/else statement switches on the received message depending on its type (Listing 8). As discussed on the Flex SmartSDR Wiki[7], there are five types of message received from the radio. "R" response messages are received from the radio in response to a command sent by the user. All commands will have a response message generated in return. These response messages are very important to confirm action on the radio intended by a client does in fact take place, and that no errors occur. In order to know which response is for which command, it's up to the user (my API) to increment a command sequence number every time one is sent, as this is then mirrored

by the radio.

The next type is a “S” Status Message, an unsolicited alert informing the client every time a radio setting is changed. There are a few types of radio objects that produce these messages, and the client must subscribe to each one in order to receive them. A FLEX radio can be connected to and controlled by multiple users, so these status messages allow people to know when the radio has been modified, and update their local settings accordingly. However, I found these status messages very useful also, as they could be used as another confirmation that changes I made were indeed completed by the radio.

Thirdly, “M” Messages are emitted by the radio when notable administrative actions occur on the radio. These can be used for logging as they contain severity warnings, however the only one I’ve received thus far is to confirm that my IP has connected to the radio.

“V” Version and “H” Handle messages describe the protocol version in use and the client connection handle respectively. Even though these are detailed individually by the wiki[7], I’ve only ever seen these two message types sent by the radio in the same message (i.e. without a ‘\n’ separating them).

```
42 def ParseRead(radio, string):
43   # print(string)
44   read_type = string[0]
45   if read_type == "R":
46     print(string)
47     ParseResponse(radio, string)
48   elif read_type == "S":
49     ParseStatus(radio, string)
50   elif read_type == "M":
51     ParseMessage(radio, string)
52   elif read_type == "H":
53     ParseHandle(radio, string)
54   elif read_type == "V":
55     ParseVersion(radio, string)
56   else:
57     print("Unknown response from radio: " + radio.radioData["serial"])
58
```

Listing 8: Switching on Message Type

In order to better understand my Data Handler, lets take a deeper look at how it addresses response and status messages, as these are the two most valuable. As eluded to before, “R” messages come in the form:

`R<seq_number>|<hex_response>|<message>[|Debug output]`

where

- <**seq_number**> = consecutive count echoed from client
- <**hex_response**> = 32-bit Hexadecimal number detailing success or failure of the command
- <**message**> = response value(s) for parsing
- <Debug output> = optional debug text output

The first steps of parsing a radio response then, are splitting the message into these partitions. Every command I send from my API is stored in a dictionary, so I can take the sequence number in the response to find the exact command sent (Listing 9 line 71).

```
60 def ParseResponse(radio, string):
61   try:
62     (response_code, hex_code, rec_msg) = string.split('|')
63   except ValueError:
64     print("Error - Incomplete reply")
65     return
66
67   response_code = int(response_code[1:])
68   hex_code = int(hex_code, 16)
69   # rec_msg = rec_msg.strip()
70   try:
71     sent_msg = radio.ResponseList[response_code]
72   except ValueError:
73     print('Unexpected reply')
74   ...
```

Listing 9: Parse Response segment

We can then use the corresponding command sent to decide exactly what to do with the response. For example, if the Response Code is 0 and the command was "stream remove" then we know we can safely delete the stream stored locally. As of the time of submission, I had only handled responses for commands that were necessary to progressing the project; a small fraction of those listed in the SmartSDR wiki. Due to the modular, class-based design I chose for the library, other commands could be easily handled by the addition of other handlers (classes), and of calls to their methods within ParseResponse().

"S" Status Messages are emitted from the radio when settings for radio objects are altered. In order to receive these status messages, the user has to subscribe to them by sending a command "sub ..." and the desired object[8]. Some examples of possible subscribe-able objects are Slices, Panadapters, Audio Streams, etc. They come in the form:

```
S<handle>|<message>
```

where

- <**handle**> = the handle of the client that triggered the update
- <**message**> = status message for parsing

```
171 def ParseStatus(radio, string):
172   try:
173     (radio_handle, rec_msg) = string.split('|')
174   except ValueError:
175     print("Error - Invalid status message")
176     return
177 
178   if rec_msg.startswith("slice"):
179     if "removed" in rec_msg:
180       return
181     split_msg = rec_msg.split(sep=' ', maxsplit=2)
182     s_id = int(split_msg[1])
183     slice_info = dict(param.split("=") for param in split_msg[2].split(" "))
184 
185     """ handler errors here if radio creates slice without us knowing about it e.g. when
    Panadapter is created """
186     try:
187       radio.GetSlice(s_id)
188     except IndexError:
189       return
190 
191     for key, value in slice_info.items():
192       # check to see if class attribute exists
193       try:
194         val_type = type(getattr(radio.GetSlice(s_id), key))
195       except AttributeError:
196         # I haven't implemented this variable, i didn't require it at this point and
    wanted to keep class uncluttered
197         continue
198 
199       # if class attribute is numerical, we want to keep it numerical
200       if val_type is float or val_type is int:
201         setattr(radio.GetSlice(s_id), key, float(value))
202       else:
203         setattr(radio.GetSlice(s_id), key, value)
204   ...
```

Listing 10: Parse Status segment

```
SENDING : C3|ant list
New Client Handle: 4E472B89
SENDING : C4|sub slice all
SENDING : C5|slice list
SENDING : C6|sub pan all
[RADIO] : R1|0|
[RADIO] : R2|0|598F6E3D-B5C8-43B2-9C9D-9F05BC57950B
[RADIO] : R3|0|ANT1,ANT2,RX_A,XVTA
[RADIO] : R4|0|
[RADIO] : R5|0|0
[RADIO] : R6|0|
SENDING : C7|slice create freq=14.222 ant=ANT2 mode=FM
[RADIO] : slice 1 in_use=1 RF_frequency=14.222000 client_handle=0x4E472B89 index_letter=B rit_on=0
        rit_freq=0 xit_on=0 xit_freq=0 rxant=ANT2 mode=FM wide=0 filter_lo=-8000 filter_hi=8000 step=100
        step_list=50,250,500,2500,3000,5000,10000,12500 agc_mode=off agc_threshold=60 agc_off_level=10
        pan=0x40000001 txant=ANT1 loopa=0 loopb=0 qsk=0 dax=0 dax_clients=0 lock=0 tx=0 active=1 audio_level=50
        audio_pan=50 audio_mute=0 record=0 play=disabled record_time=0.0 anf=0 anf_level=0 nr=0 nr_level=0 nb=0
        nb_level=50 wnb=0 wnb_level=0 apf=0 apf_level=0 squelch=1 squelch_level=20 diversity=0 diversity_parent=0
        diversity_child=0 diversity_index=1342177293 ant_list=ANT1,ANT2,RX_A,XVTA mode_list=LSB,USB,AM,CW,DIGL,
        DIGU,SAM,FM,NFM,DFM,RTTY fm_tone_mode=OFF fm_tone_value=67.0 fm_repeater_offset_freq=0.000000
        tx_offset_freq=0.000000 repeater_offset_dir=SIMPLEX fm_tone_burst=0 fm_deviation=5000 dfm_pre_de_emphasis=0
        post_demod_low=300 post_demod_high=3300 rtty_mark=2125 rtty_shift=170 digl_offset=2210 digu_offset=1500
        post_demod_bypass=0 rfgain=0 tx_ant_list=ANT1,ANT2,XVTA
R7|0|
New Slice Added: <FlexModule.Slice.Slice object at 0x000001B1F282A508> slice_id=1
SENDING : C8|slice s 1 mode=USB
[RADIO] : slice 1 mode=USB filter_lo=100 filter_hi=2800 agc_mode=med agc_threshold=50 agc_off_level=50
        qsk=0 step=100 step_list=1,10,50,100,500,1000,2000,3000 anf=0 anf_level=0 nr=0 nr_level=0 nb=0 nb_level=50
        wnb=0 wnb_level=0 apf=0 apf_level=0 squelch=1 squelch_level=20
[RADIO] : slice 1 filter_lo=100 filter_hi=2800 post_demod_low=300 post_demod_high=3300
[RADIO] : slice 1 fm_tone_mode=OFF fm_tone_value=67.0 fm_repeater_offset_freq=0.000000 tx_offset_freq=0.000000
        repeater_offset_dir=SIMPLEX fm_tone_burst=0 fm_deviation=5000 dfm_pre_de_emphasis=0 post_demod_low=300
        post_demod_high=3300 post_demod_bypass=0
[RADIO] : R8|0|
SENDING : C9|slice list
[RADIO] : R9|0|0 1
```

Figure 9: Slice Creation and Modification

Again, we start by splitting the status message into these two parts. As "S" messages are sent regardless of interaction by the client, we don't have a matching list to pull from like "R" messages. However each status update starts with the radio object it is describing so we can branch the message according to this. Theoretically, the radio only wants to update us on the settings that have changed, but it groups multiple settings together, meaning if one setting is altered the whole group is included in the status message.

Figure 9 shows an example status message (with the handle already removed), which is essentially a long string of key, value pairs. For each radio object I handle the update slightly differently, but one commonality is that from this string I create a dictionary containing all the pairs received (Listing 10 line 183). In this dictionary will be far more information than I need or will need, and to keep my Classes uncluttered I didn't create variables for all keys in the status message. However, I wanted a succinct way of updating the class variables I do have, and I think I came up with an elegant solution.

Listing 10 lines 191-203 show that for each key in the dictionary, I try and find the data type for its corresponding value with 'getattr()'. If the Class has a variable matching

that key name, it returns the data type and I can continue. If not, an 'AttributeError' is caught and I go onto the next key, knowing that I haven't implemented it in my class. Thereafter I use 'setattr()' to update the class variable, ensuring to keep it numerical if it was before.

This solution was highly successful, as it meant my Data Handler could pass a status update no matter which groups were included in the message or what class variables I had implemented. I had previously used multiple try, except clauses to check if a particular key was in the message and then see if it could be updated, but I knew this would get very ungainly as the program grows. Instead, in just a few lines I created a very robust system that can expand or contract at the wishes of the designer.

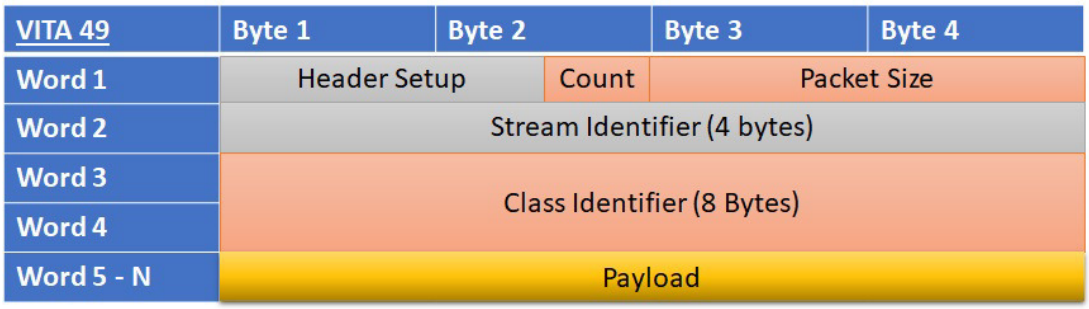


Figure 10: VITA 49 Packet Format [15]

Returning back to data received on the Datagram Socket, let's begin by looking at the format it comes in. The UDP data is packaged into VITA 49 packets, a common standard for SDR communication. The API Primer[15] was a great resource to understand the packet format, especially pages 24 and 25 (the author like me found that information online about the VITA format is "sparse"). The packet is in two sections: Header and Payload. The payload is variable size depending on what is being sent and contains the raw data. The packet size is stored in the header, along with a packet count and class identifier. There are other fields in the header too (Fig. 10), but we only require the fields in orange (and the payload obviously).

Once the UDP packet has been loaded in a VitaPacket() class, we can parse it.

```
265 def ParseVitaPacket(radio, packet):
266   Id = int.from_bytes(packet.class_id, byteorder='big') & int('FFFF',16)  # all but the
        last 2 bytes are the same
267   ValidatePacketCount(Id, packet.pkt_count)
268   if Id == int('FFFF',16):
269     # DISCOVERY Packet
270     pass
271   elif Id == int('8003',16):
272     # FFT Packet
273     if radio.Panafall:
274       pan_data = ParsePanadapterPacket(packet, radio.Panafall.x_pixels, radio.Panafall.
        y_pixels)
275       radio.Panafall.PanBuffer.put_nowait(pan_data)
276   elif Id == int('8004',16):
277     # WATERFALL Packet:
278     if radio.Panafall:
279         ParseWaterfallPacket(packet)
280   elif Id == int('8005',16):
281     # OPUS AUDIO Packet
282     if radio.RxAudioStreamer:
283       opusData = ParseOpusPacket(packet)
284       radio.RxAudioStreamer.outBuffer.put_nowait(opusData)
285   elif Id == int('3E3',16):
```

```
286     # IF NARROW Packet
287     if radio.RxAudioStreamer:
288       ParseIfNarrowPacket(packet, radio.RxAudioStreamer.outBuffer)
289
```

Listing 11: Parse VITA Packets

Listing 11 shows how this parsing is achieved, switching on the class id which represents the type of data received. So far I've only addressed FFT, Waterfall, OPUS encoded audio and raw audio, but this structure allows new VITA packets to be added with ease. The parsing of the specific packets is all done in a similar way, so let us consider one in more detail. The Pan Adapter plot as seen in the official SmartSDR UI (fig. 2) can be plotted by a client too, as the FLEX-6400 sends the required data to do so. As a FLEX community post[21] describes, the data is essentially just an array of Amplitude values the width of your Panafall() object, so it's fairly trivial to get this data and plot it.

```
366 def ParsePanadapterPacket(packet, x, y):
367   pan_data = []
368   index = 0
369
370   StartBin_index, NumBins, BinSize, TotalBinsInFrame, FrameIndex, PacketCount = unpack(">
        HHHHLL", packet.payload[index:index+16])
371   # Packet_Count could be used to error check, but we already have an error check for all
        VITA packet types
372   index += 16
373
374   for i in iter_unpack(">H", packet.payload[index:index+TotalBinsInFrame*2]):
375     pan_data.append(y - i[0]) # invert y axis as FLEX graphs differently
376
377   return pan_data
378
```

Listing 12: Parsing a PanAdapter Packet

Data received over the UDP socket is in the form of a bytestring, and so I used a very efficacious Python module 'struct' which manipulates such strings. Line 370 in Listing 12 shows how I unpack the start of an FFT packet, which holds information about the PanAdapter being displayed. The argument ">HHHHLL" says that we're expecting four unsigned shorts and two unsigned longs all big-endian in the first 16 bytes of data (see python struct documentation[25] for more detail). Then lines 374-375 show how I iteratively unpack the rest of the packet to obtain the actual Pan Adapter data. Struct methods 'unpack()' and 'iter_unpack()' both always return tuples, so I must index the

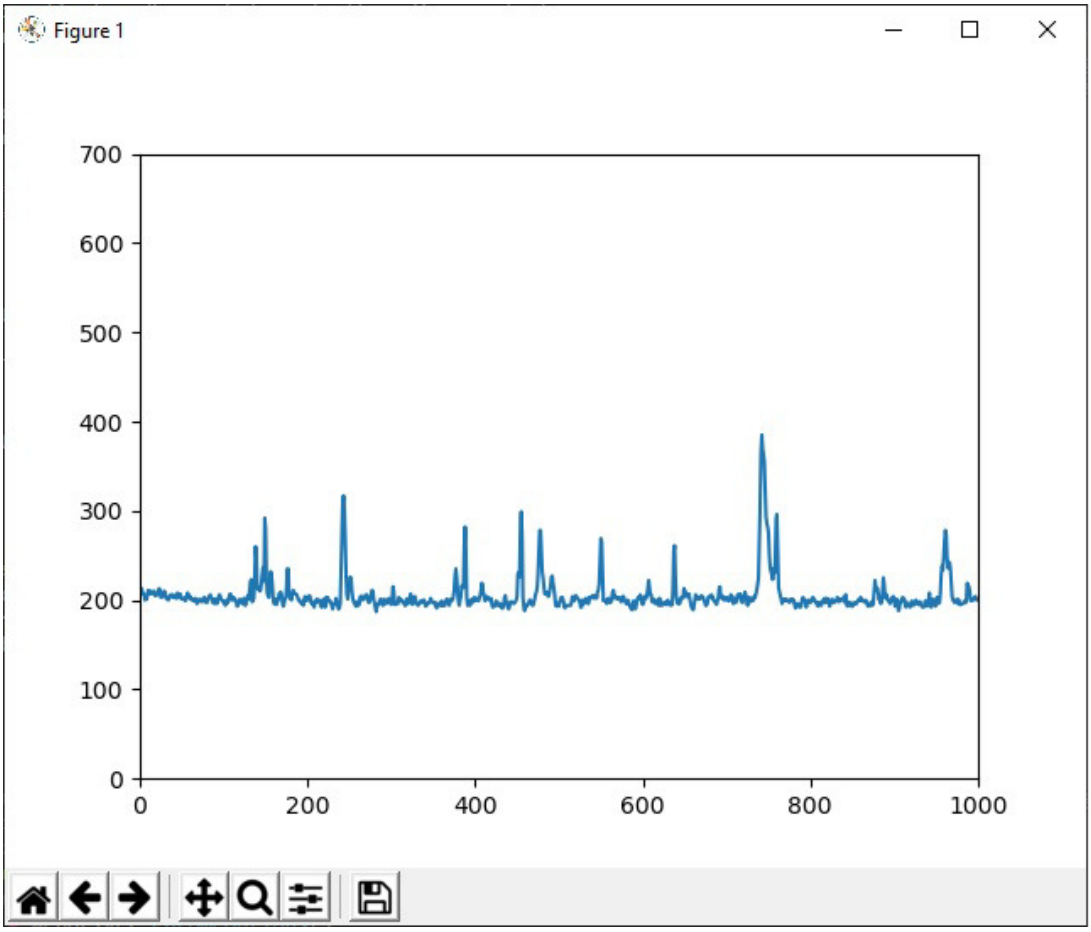


Figure 11: Panadapter Plot from FlexModule library

result to obtain just the integer value. This data then needs to be inverted as I believe FLEX do their FFT plot from the top right not bottom left. This unpacking or iterative unpacking is the basis of how I handle the UDP data, and I think it's a neat and concise way of doing it. Figure 11 shows an example plot of the Panadapter data being received from the FLEX-6400. This data is actually plotted live, updating in real-time. Each peak represents a received signal in the spectrum, with the largest peak being an Upper Side-Band transmission. The axis labels are currently indicating the number of pixels of the display, but it wouldn't be too difficult to change this to the frequency and decibel values, as these are currently stored as variables in the Panadapter class. It would require scaling the array of values received however, and I didn't have time to implement this before the submission date.

I also handle the waterfall data sent from the FLEX radio, but at the time of submission it wasn't working as well as I would like. This will be amended in time for the VIVA but I couldn't include a screenshot in this report.

My ultimate goal for this data handler was to allow GNU Radio to very accurately describe the radio's system, without discrepancies between the two. I believe I successfully achieved that, due in part to the way new objects on the local side are created. For example, when my API sends a "slice create..." command, it could create a Slice() object locally at the same time. However, if the message was dropped or the radio failed in creating it, there would be a divergence between the number of slices on the radio and the number of slices stored locally. To avoid this, the Data Handler only creates a Slice() when it receives a "good" slice-create response back from the radio (Listing 13). Looking at Figure 9, we can see Command 7 requesting the creation of a Slice, and our API only doing so locally once we receive Response 7 with no errors. This still brings up the issue of what happens if this response message is dropped, but I believe it's better to not know that a slice is available than not know a slice is unavailable.

```
75 if "slice c" in sent_msg:
76   if hex_code != 0:
77     # log error
78     pass
79   else:
80     slice_data = dict(param.split("=") for param in sent_msg.split(" ")[2:])
81     newSlice = Slice(radio, float(slice_data["freq"]), slice_data["ant"], slice_data["
       mode"])
82     radio.SliceList.append(newSlice)
83
```

Listing 13: Consistent Slice Creation

Creating an Audio Stream

Receiving Audio Data was a big milestone for my project, and while not as time-consuming as the previous two key areas it is worth discussing in this section. In order to have audio streaming from the FLEX radio, there are a few prerequisites that must be done. Firstly, the UDP channel must be connected and open. Secondly a Slice must be present and subscribed to. With these conditions met, a user can send the command to request a Remote Audio Stream:

```
82 def CreateAudioStream(self, isCompressed):
83     command = "stream create type=remote_audio_rx compression="
84     if isCompressed:
85       command += "opus"
86     else:
87       command += "none"
88     self.SendCommand(command)
89
```

Listing 14: Requesting a Remote Audio Stream

Here we have the option to either receive raw uncompressed data or opus encoded data. Opus is an audio coding format that is quite common in real-time applications such as radio due to its low latency[1]. As the data is received over the UDP port, both types are packaged into a VITA 49 packet. I did not get around to decoding opus audio as I didn't have time to decode it myself. The library could however easily be extended to include opus decoding by using a Python library already available[28].

The next step is to unpack the received packets. This is performed by the Data Handler as discussed in the previous section, but just for further clarification Listing 15 shows how its done.

```
298 def ParseIfNarrowPacket(packet, buffer):
299   switch = True
300   for flt in iter_unpack("!f", packet.payload): # take every 4 bytes and cast to float
301     # FLEX sends 2 channels of same audio stream - I'm only saving 1 channel
302     if switch:
303       buffer.put_nowait(flt[0]) # iter_unpack returns tuple of 1 item
304     # else:
305     #   do something here if 2nd channel required
306     switch = not switch
307
```

Listing 15: Parsing an Audio Packet

Again, I use the 'struct.iter_unpack()' method to iteratively take a four byte float in the bytestring. This is then stored in the RxRemoteAudioStream's buffer. The radio actually sends each float value twice. While I haven't confirmed the reasoning behind this - again information online is fairly limited - I believe this is because it's sending two channels of data at a time. The radio allows for two Remote Audio streams to be active simultaneously, so if I had another one running this second value would be for the second stream. I have only implemented the option for one stream in my API at this time, but the switch condition in `ParseIfNarrowPacket()` allows for simple modification to save the second stream.

Looking at the RxRemoteAudioStream() class (Listing 16), you can see I implemented the buffer data structure as a Queue(). This structure is thread-safe and adding a member to it is an O(1) operation meaning it won't slow the Data Handler.

```
5  class RxRemoteAudioStream(object):
6    """class for a RX Remote Audio Stream """
7
8    def __init__(self, radio, stream_id, isCompressed):
9      # super(RXRemoteAudioStream, self).__init__()
10     self.radio = radio
11     self.stream_id = stream_id
12     self.isCompressed = isCompressed
13     self.outBuffer = Queue()
14
```

Listing 16: RxRemoteAudioStream Class

GNU Radio

As discussed in the Approach chapter, an Out-Of-Tree GNU Radio block must conform to the standard block structure in order to run. Apart from that, we have a lot of flexibility in the code we can use inside it. From Listing 17 you can see that I'm importing my FlexModule API and instantiating SmartLink() and Radio() classes, which is connecting to the FLEX-6400.

```
23 import numpy
24 from gnuradio import gr
25 from FlexModule.SmartLink import SmartLink
26 from FlexModule.Radio import Radio
27 from time import sleep
28 import FlexModule.DataHandler
29
30 class FlexSource(gr.sync_block):
31     """
32     Source block for FLEX radio connection, streaming audio data
33     """
34     def __init__(self, serial):
35         gr.sync_block.__init__(self,
36             name="FlexSource",
37             in_sig=None,
38             out_sig=[numpy.float32, ])
39         self.serial = serial
40
41         self.smartLink = SmartLink()
42         if len(self.smartLink.radio_list) < 1:
43             return
44         self.radioInfo = self.smartLink.GetRadioFromAvailable(self.serial)
45         self.flexRadio = Radio(self.radioInfo, self.smartLink)
46
47         if self.flexRadio.serverHandle:
48             receiveThread = FlexModule.DataHandler.ReceiveData(self.flexRadio)
49             receiveThread.start()
50
51             self.flexRadio.UpdateAntList()
52             self.flexRadio.SendCommand('sub slice all')
53             self.flexRadio.SendCommand("sub pan all")
54
55             self.flexRadio.CreateAudioStream(False)
56
57             """ should find a nicer way of doing this """
58             for _ in range(5):
59                 if not self.flexRadio.RxAudioStreamer:
60                     sleep(1)
61             self.flexRadio.OpenUDPConnection()
62         else:
63             # raise exception in GR interface
64             return
```

Listing 17: Defintion of FlexSource Block

In lines 35-39, the block itself is initialised, inheriting from the standard *Sync* type. However, I set in_sig to None, making it fundamentally of type *Source*.

I check to see if a connection to the radio is actually established by verifying it has obtained the session handle from the SmartLink server. If that is the case, I can create a ReceiveData() thread and do the preliminary updating to make sure the local Radio() object is mirroring the state of the actual FLEX-6400. Finally, we can create an audio stream, the parameters of which are defined in the block itself (Listing 20).

Listing 18 lays out the 'work()' function. For my FlexSource block, the work function pulls the data from the RxRemoteAudioStream queue and appends it to a list. Once it has either pulled all the data or filled its own buffer, it then converts the list to a numpy array, as this is what GNU Radio expects. The reason I append to a temporary list instead of straight into a numpy array is because an appended value simply extends a list. Numpy arrays have fixed length, so every append would create a new array; this would be very inefficient on memory.

```
67 def work(self, input_items, output_items):
68     out = output_items[0]
69     # if self.flexRadio.RxAudioStreamer.isCompressed:
70         # do Opus decompression
71 
72     """ Queue() implementation"""
73     out_len = min(len(output_items[0]), self.flexRadio.RxAudioStreamer.outBuffer.qsize())
74     # print(out_len, end=" ")
75     if out_len == 0:
76         return 0
77 
78     temp = []
79     for i in range(out_len):
80         temp.append(self.flexRadio.RxAudioStreamer.outBuffer.get_nowait())
81     out[:out_len] = numpy.array(temp)
82 
83     return out_len
```

Listing 18: FlexSource Work Function

Below I have some more API calls in local methods, to tune the Slice object from where the Audio Stream is capturing data. A user could also change the demodulation mode or the antenna.

```
86 def setFreq(self, newFreq):
87     self.flexRadio.GetSlice(0).Tune(newFreq)
88
89 def setMode(self, newMode):
90     self.flexRadio.GetSlice(0).Set(mode=newMode)
91
92 def setAnt(self, newAnt):
93     self.flexRadio.GetSlice(0).Set(ant=newAnt)
```

Listing 19: Example FlexModule API calls

To actually generate the block on the UI, GNU Radio uses .yml files to describe the number of I/Os, the I/O data types and their parameters. Listing 20 shows how Figure 12 is represented in yml. Note that the callback list is how these parameters are updated by the user, which then subsequently makes the API calls to update the remote SDR.

```
5  templates:
6    imports: import gnu_flex
7    make: gnu_flex.FlexSource(${serial})
8    callbacks:
9    - setFreq(${freq})
10   - setMode(${mode})
11   - setAntenna(${ant})
12
13 #  Make one 'parameters' list entry for every parameter you want settable from the GUI.
14 #     Keys include:
15 #     * id (makes the value accessible as \$keyname, e.g. in the make entry)
16 #     * label (label shown in the GUI)
17 #     * dtype (e.g. int, float, complex, byte, short, xxx_vector, ...)
18 parameters:
19 - id: serial
20   label: Serial Number
21   dtype: string
22 - id: freq
23   label: Slice Frequency
24   dtype: float
25 - id: mode
26   label: Slice Demod Mode
27   dtype: string
28 - id: ant
29   label: Slice Antenna
30   dtype: string
```

Listing 20: gnu_flex_()FlexSource.block.yml

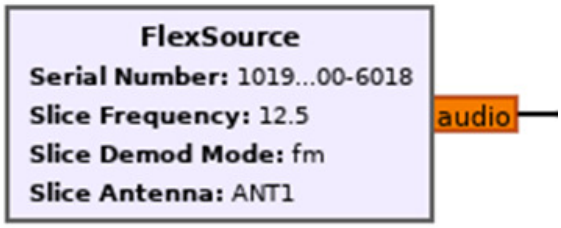


Figure 12: FlexSource Block in GNU Radio

Finally, Listing 21 shows how the input/output boxes are defined. Again, referring to Figure 12 we can see that it only has an output, of type float (which is what the colour orange represents). This is reflected in the aforementioned listing, as it has no inputs defined - the section has been commented out - and the outputs exactly represent the block pictured.

```
32 #  Make one 'inputs' list entry per input and one 'outputs' list entry per output.
33 #  Keys include:
34 #      * label (an identifier for the GUI)
35 #      * domain (optional - stream or message. Default is stream)
36 #      * dtype (e.g. int, float, complex, byte, short, xxx_vector, ...)
37 #      * vlen (optional - data stream vector length. Default is 1)
38 #      * optional (optional - set to 1 for optional inputs. Default is 0)
39 # inputs:
40 # - label: ...
41 #   domain: ...
42 #   dtype: ...
43 #   vlen: ...
44 #   optional: ...
45
46 outputs:
47 - label: audio
48   domain: stream
49   dtype: float
50   vlen: 1
51   optional: 0
```

Listing 21: FlexSource I/O's

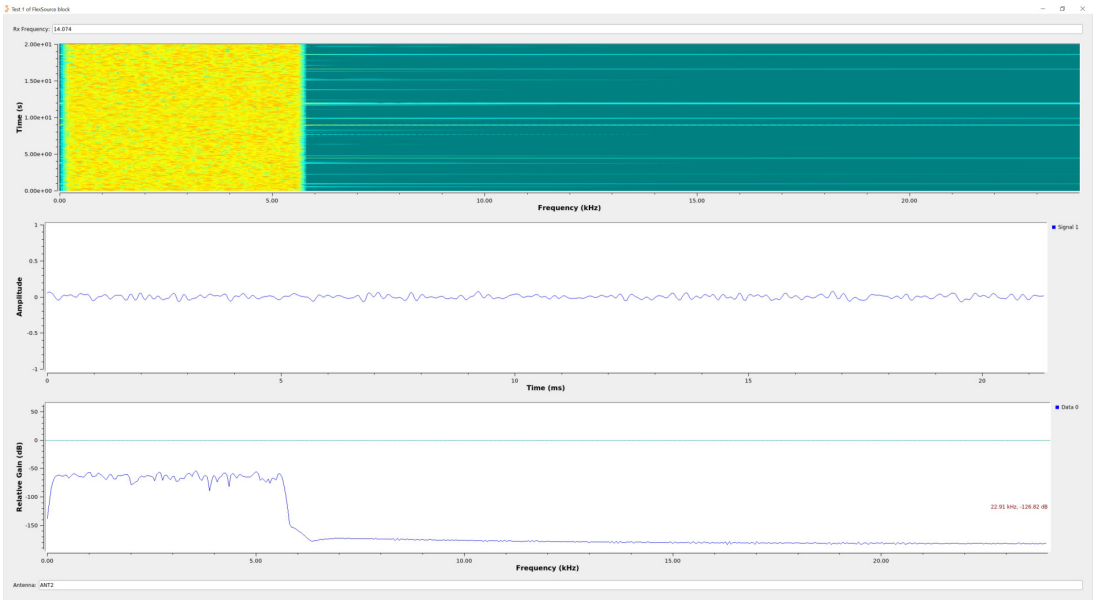


Figure 13: Audio Data Visualised on GNU Radio

Figure 13 shows the combined results of all these key areas. Audio data is obtained from the FLEX-6400 from my FlexSource GNU Radio block. This data is then passed into 3 visualisation sinks (the three shown in Fig. 4).

Future Work

While I believe I made some good ground in the project and delivering a working prototype is satisfying, I am slightly disappointed in what is left to do. Given a few more weeks I know I could have enabled effective IQ data streaming from the FLEX-6400 and into GNU Radio. I also would have liked to at least attempted transmitting audio, even though I stated in my initial plan this would be bonus work.
I know the process required to request an IQ stream from the radio. A Panadapter is needed - which I have implemented successfully - and then these three commands get IQ data streaming over the UDP port:

- `sub daxiq all`
- `stream create daxiq=1`
- `dax iq set 1 pan=<pan_id>`

Then I just need to parse the data being received over the Datagram Socket, but with the structure of my Data Handler and Vita() class this would be trivial.

Transmitting audio would have been slightly more complicated, but if I upload an audio file I can iteratively pack it into a bytestring. Then I package it into a VITA 49 packet by making a simple method in the Vita() class and it would be ready to be transmitted. The only thing left to do would be to make a TX equivalent of my RxRemoteAudioStream in the Radio class and instruct the FLEX radio I will be transmitting it.

So while I was unable to complete these goals, the functionality is either there or only a

few extra steps away from being possible. However, these goals are not the only things I would add to my library if given the time. While I think the API is stable and robust, I still think there are areas which could be improved to make it more extendable.

Firstly, I would like to remove some of the strain from the Data Handler. In its current state it handles all traffic from the radio: TCP/IP and UDP data. It performs well with the subset of radio objects I've implemented so far, but as more are added I know packets will start to be dropped while it parses the old ones. The more packets accepted mean the more packets that need to be parsed and hence a backlog is more likely to occur. To alleviate this issue I would do two things: 1 - split the handler in two so each socket has a dedicated parser, and 2 - introduce asynchronous tasks[22] so that the 'Receive-Data()' thread (Listing 7) can be solely committed to receiving data from the sockets. These asynchronous tasks would then do all the actual parsing of the packets in a separate thread which ends once the function calls finish. I believe these two updates to the Data Handler would make it much more reliable and elastic; it could handle implementations with few packets types enabled or many.

Next, I would like to find a way of allowing user authentication without using a browser. I will highlight the issues I had using browser authentication in the Evaluation section, but finding an alternative would be much more flexible and avoid many of the problems I had moving my library from Windows to Linux. I wanted to make the API as cross-platform as possible, and this was the biggest hindrance. Along with this I'd like to try and implement another type of Auth0 access called a refresh token[4], which means a user can stay authenticated longer than just the run-time of the program. Consequently, they don't have to login every time and the library thereby becomes much more user friendly.

Finally, I want to implement a functional and effective logging system, to give real-time feedback on a user's interaction with the API. As you can see in the 'ParseResponse()' function I've left spaces to report errors that are returned from incorrect commands.

Unfortunately I didn't have time to properly address this area while I was trying to meet the main goals of the project, but I know this should be met if I want to ensure the maintainability and usability of the library. I would also like to make this logging coalesce with GNU Radio, so that alert boxes are generated on the UI. This prompts the user to address their error and helps them understand what is wrong, which is better than making them look in a text file. This extension could get complicated but I think it would be worth attempting.

Evaluation, Results & Testing

Testing

Throughout my project, I included tests and validation checks to increase the stability and fault tolerance of the API. These tests could be to recognise if an error had occurred, recover from an error or gracefully exit from an error if the API cannot progress further. These checks were also very useful during development to identify where and why the program was failing, a fundamental way to increase the speed of the project's progress.

I will first discuss some tests embedded in the code itself, which provide feedback on the state of the API or the communication it is parsing. Beginning with `FlexModule\SmartLink.py`, the function *get_auth0_tokens()* has a number of verification tests in it to ensure the process is completed correctly. Referring to Listing 3, the first test is on line 82. Here we make sure that the URL returned from connecting to Auth0 contains the string "Found. Redirecting to ". If this is not the case, the necessary redirection shown as Step 3 in Figure7 has not been achieved. Then on line 104, we verify we have indeed received an authorization code from Auth0 by checking "code=" is present in the URL. As the Auth0 code is required to be exchanged for the JWT, if this steps fails we cannot proceed. Finally, we test to see we do indeed get a token in return on line 119. Without the token we cannot authenticate ourselves with the SmartLink server and therefore we will not be able to see our available radios, so this step being successful is imperative to the program.

Continuing in `SmartLink.py`, a test is also performed in *SendRegisterApplicationMessageToServer()*. Line 137 in Listing 4 shows how I confirm the TLS connection to the SmartLink is still present. As I establish the connection in the initialisation of the class, there could be some time before the token data is actually sent to the server. This is due to the user having to enter their details in the browser. To keep the connection alive, I created a thread to periodically ping the Server. If this pinging fails or the connection breaks, the socket will not be live and therefore not have a version type to return. The benefit of these tests in `SmartLink.py` is I can see exactly at what stage the authentication has failed. This allows me diagnose very quickly what may be the problem and focus my efforts.

The next stage of tests take place in `FlexModule\Radio.py`. During the process of telling the SmartLink server we intend to connect to our chosen radio (*SendConnectMessageToRadio()*, lines 37-41 of Listing 5 verify that we did indeed receive the corresponding data from the server. This data confirms that the radio is available and active; if not it's probably switched off or we have entered a serial number not matching one of our radios. If this step fails, we would be telling SmartLink to try and communicate with a radio that we don't own or isn't on. We wouldn't receive an error message from SmartLink if we attempted this, so this test is very important to get verification that we can indeed connect to the specified radio.

Another test is performed in the same function, shown in lines 46 to 51. Here the program verifies a handle is returned from SmartLink by trying to split the message to obtain it. If this is not possible, it would mean that the connect message we sent failed and we can not establish a Peer-To-Peer link to the radio. If that's the case, the method returns an empty string - essentially a boolean False in Python. This prevents the API from trying to create the physical link to the radio but allows a developer to handle an empty string as they'd like. Once again, these two stages of tests in `Radio.py` allow me to quickly determine the success or failure in connecting directly to the radio and why it

may or may not have occurred.

The last group of tests present in the API are found in `FlexModule\DataHandler.py`. These tests are validating the data received from the radio is correct and doesn't contain errors. As before I'll go through the functions individually, discussing the tests involved are what the advantages are of having them implemented. First looking at *ParseRead()*, a simple test is performed to ensure the data TCP/IP received conforms to the format specified in the SmartSDR wiki[7]. This allows me to confirm the data received from the radio isn't garbage and it's being decoding correctly.

Further validation is performed in *ParseResponse()* and *ParseStatus()*. In both cases, the received message is verified that it further matches the expected format again taken from their respective wiki pages. These necessary tests as it's important all responses and status updates are handled by the API. If they are being received incompletely or the radio isn't sending them correctly, effective control of the radio and proper reflection of its status in the API can't be achieved. I previously mentioned it in the Future Work section, but another test performed in *ParseResponse()* is on the hex code contained in the message. This code is how the API can identify if a command was accepted by the radio or not. While I haven't implemented individual actions for each response, the test will allow me to add logging or error handling in the future, which is an essential part of any good program.

The final test performed by the API I will discuss is how it verifies the packets being received over UDP. As I detailed in the Implementation chapter, the VITA packet format contains a 4-bit packet count (Fig 10). This count increments from 0 to 15 and overflows to 0 again. Each packet type has it's own packet count, meaning we can keep track of the counts for only the VITA packets we have implemented.

```
380 def ValidatePacketCount(pkt_id, pkt_cnt):
381   Error = False
382   if pkt_cnt == 0:
383     prev_cnt = 15
384   else: prev_cnt = pkt_cnt - 1
385
```

```
386   if pkt_id == int('8003',16):
387     try:
388       if ValidatePacketCount.fftCount != prev_cnt:
389         Error = True
390     except AttributeError:
391       # not been intialised yet, will be below
392       pass
393     ValidatePacketCount.fftCount = pkt_cnt
394   elif pkt_id == int('8004',16):
395     try:
396       if ValidatePacketCount.wtrflCount != prev_cnt:
397         Error = True
398     except AttributeError:
399       pass
400     ValidatePacketCount.wtrflCount = pkt_cnt
401   elif pkt_id == int('8005',16):
402     try:
403       if ValidatePacketCount.opusCount != prev_cnt:
404         Error = True
405     except AttributeError:
406       pass
407     ValidatePacketCount.opusCount = pkt_cnt
408   elif pkt_id == int('3E3',16):
409     try:
410       if ValidatePacketCount.ifNCount != prev_cnt:
411         Error = True
412     except AttributeError:
413       pass
414     ValidatePacketCount.ifNCount = pkt_cnt
415   # Add more packet types when required
416
417   if Error:
418     print("Packet Dropped:" + str(pkt_id))
```

Listing 22: VITA 49 test function ValidatePacketCount()

Listing 22 is how the API does this, with static variables for each packet type. The count for the most recently received packet is passed in as a parameter, and then the previous count is calculated. If the stored packet count is not the same as this calculated number, we must have dropped a packet and an Error is thrown.

This method allowed me to see how efficient my Data Handler was and whether it could contend with all the data the radio was transmitting. I found that even my slow laptop could have almost all the data without dropping packets and it proved to me that my approach was more than satisfactory.

I also performed regular testing on my API throughout development. This involved

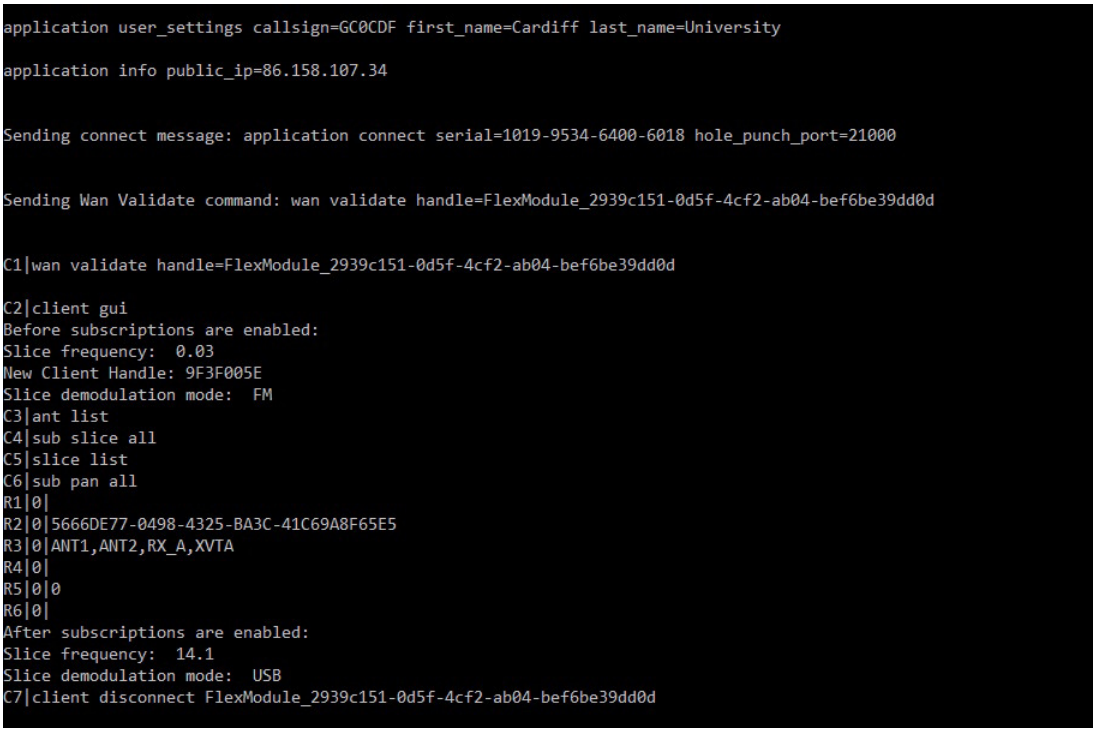


Figure 14: Testing the Subscription Handling in the terminal

testing the commands sent were performing the correct actions, subscriptions and status updates were updating the API settings and that the UDP data was valid. One such test is shown here.

The above Figure displays the subscription handling in effect. As Listing 1 line 13 shows, a Radio() object is intialised with a Slice already present. This Slice is given default values: a frequency of 0 MHz (which is saved as 0.03 MHz) and a demodulation mode of FM. As Figure 14 indicates, once the slice radio object is subscribed to, the slice stored locally on the API is updated to reflect the slice on the radio: the frequency becomes 14.1 MHz and the mode becomes USB; all done without user input. This an example of the testing I carried out during development. I performed lots of these tests to verify my data handling was correct and functioning.

Evaluation

When looking at the goals I laid out at the start of my project, it is unfortunately true that I didn't manage to achieve them all in the allotted time. I was disappointed that this was the case, but I don't feel it was through a lack of effort or capability. Rather, I believe I hit too many roadblocks which took too long to unravel than what a 13 week project allows for. I will discuss these roadblocks shortly, but first we should take an empirical evaluation of the results of my project.

The first goal I wanted to accomplish was to Discover and Configure the FLEX-6400 radio. I achieved this goal, but the project started on the wrong foot as it took me a lot longer than I expected to establish the connection to the radio. Nevertheless, my API allows a user to authenticate themselves with the SmartLink server and then connect to their desired radio. Once connected, they can then send general administrative commands to configure the SDR's settings remotely.

The second goal was to be able to receive Audio data from the FLEX-6400. This was also achieved: a user can create an Audio Stream on the radio which then sends data to my API over UDP. This was a big milestone for the project, as it requires a lot of functionality in the API to be present. I was very happy with the architecture of the library and the quality of it meant when it came to creating the audio stream it was straight forward. So even though again I took longer on this stage than i would have liked, I completed this goal.

The third and last goal that I expected myself to complete in the 13 weeks was receiving IQ data from the radio. This was where time ran out on me. As I mentioned in the Future Work chapter, I was unable to accomplish this goal but I know given a week or two more I could have done. The functionality is present in my API to request and accept an IQ stream, I just wasn't able to put it together and test it. It's disappointing that I couldn't meet this goal.

I also discussed additional goals in my Initial Report should I finish the expected ones. As I ran out of time, I also couldn't attempt transmitting Audio or IQ data. While I can't consider not achieving these goals a failure, I think it would have been really beneficial to the library and very interesting to discuss in the report if I had the time to attempt transmission of data.

However there were other achievements to come out of this project that should be mentioned. The creation of my library in general is a great accomplishment as it is something that had not been done before but will be useful immediately. This report itself is also a nice result of the project, as it's an amalgamation of a lot of research from different sources. It's a detailed and accurate manual in how remote communication with a FLEX must be established and performed, which will be of great utility to many developers and operators, be they professionals or enthusiast.

This project could be characterised by Hemingway's Law of Motion, where he describes bankruptcy as something that happens slowly, and then all at once. I felt this occurred quite cyclically during the project development, where I would have a week or 2 of no progress at all and then a few days of sudden rapid expansion.

I will next discuss aspects of the project that caused these stalls in progress. I have already mentioned the delay at the start of the project due to difficulties in completing the authentication and discovery of the radio. There were multiple issues in this task, but they all stemmed from a lack of clear documentation in how to authorize and authenticate oneself with the SmartLink server. Although the auth0 flow[3] informed me on the steps required, it couldn't help me in knowing what site I needed to visit and what to do once I'd got the token. The latter was the most troublesome because documentation became even sparser, and when the right steps were eventually realised, it came about through days of trial-and-error. For example, when creating the TLS socket directly to the radio, the SSL context must be customized so that it does not require certificate verification[19]. If this

was not done, the socket connection would be established and the command `WanValidate` would be delivered, but the Peer-To-Peer connection would not be accepted. As far as I've seen, nowhere details that this SSL context modification must be done in order for the connection to be completed successfully.

Not only that, but this connection and WAN validation must be performed immediately after the SmartLink server sends you the session handle. I lost a few days trying to figure out why I wasn't successfully connecting to the radio with what I believed to be a valid handle. Again, nowhere in any FlexRadio documentation says how long the handle is valid for until it must be sent to the radio. The reason for my issue was I was creating the TLS socket after receiving this handle, by which time it had expired before i could send the actual "wan validate handle=<handle>" message.

```
19 context = ssl.create_default_context()
20 context.check_hostname = False
21 context.verify_mode = ssl.CERT_NONE
22 self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23 self.FLEX_Sock = ssl.wrap_socket(self.sock) # socket to comms with the FLEX radio
24 self.DATA_Sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
25 self.UdpListening = False
26
27 self.clientHandle = ""
28 self.serverHandle = self.SendConnectMessageToRadio()
29 if self.serverHandle:
30   self.FLEX_Sock.connect((self.radioData['public_ip'], int(self.radioData['
       public_upnp_tls_port'])))
31   # print(self.FLEX_Sock.getpeername())
32   self.WanValidate()
33   self.SendCommand("client gui")
```

Listing 23: Excerpt from Radio.py initialisation code

Note lines 19-23 of Listing 23 which show how I create all the socket instances first before I connect to them. Then, once I receive the session handle from the server (line 28), I immediately connect and call the `WanValidate()` function. Receiving the handle and then creating the socket instance after seemed to be just time-consuming enough to allow the handle to expire.

These issues with Authentication and Discovery really impeded my progress for the first weeks of the project, and solutions online were few and far between. It took a lot of work and communication between myself and my supervisor to remedy the problems and

eventually complete the authentication, but with only 13 weeks assigned setbacks such as this right at the start of the project have real impact.

Another notable problem I had was hanging or disconnecting sockets during run-time. Often, when I came to receive data from the TCP/IP (TLS) socket, I could take all the data available but then the socket would hang, completely locking the API. I was using the standard "socket.recv(bytes)" method to accept messages, but for some reason the socket would sit trying to read more data even when none was available. I tried to make the socket disconnect once it had received the data but obviously this would mean I'd have to reconnect every time I wanted to receive data and would lead to lost packets. I also tried putting the TLS socket into blocking mode, so it would timeout after so many seconds of not receiving data. This did fix the problem, but I wasn't satisfied with the solution because being in blocking mode would freeze the whole API while it tried to receive data. As the API expands, this would obviously become more and more of an issue. After a week or so of troubles with this hanging socket problem, the client found the 'select' Python module, which can poll the sockets and try to see if data is available to be received without actually accepting it (Listing 7 line 21).

In the Implementation chapter, I talked about Response messages, and how the first part of them contains the sequence number. This sequence number is echoed from the command sent from the user, relying on you incrementing it for every new command you send. However, at first I misunderstood the SmartSDR wiki documentation and thought the sequence number was unique for each individual command: i.e. a "slice create" command had a sequence number of 21, a "slice remove" command had a sequence number of 22 and so on. This misinterpretation led to some strange behaviour from the radio, as it would accept a command if the sequence number had jumped ahead, but if I sent a repeated command I would get unexpected error messages in return. It wasn't until I went through the code with my supervisor in one of our weekly meetings that we realised my

mistake. This not only slowed down the progress during the Discovery and Configuration phase, but also meant I had to significantly alter my data handler. Beforehand, I believed I could take the sequence number in the Response message and determine directly what message it was responding to, from a simple message-type dictionary lookup. However, in reality the same command would have a different sequence number each time it was sent, and therefore I had to keep track of exactly what messages I was sending in order to know what the response was pertaining to.

One of the most significant issues I had during this project was an adverse result of using browser authentication in the Discovery and Configuration phase. The machine I used to build this project is a laptop with a Windows OS. However, building custom GNU Radio blocks requires Linux, as it uses 'cmake' commands (although I should mention GNU Radio have a Windows implementation in the pipelines). Due to COVID-19, I couldn't use the Linux Lab machines available in the ComSci building, and so had to use a Windows Subsystem for Linux (or WSL) installed from the Microsoft Store. This presented many problems, the first of which being WSL doesn't natively provide the ability to allow the use of UI's, like a browser or the GNU Radio interface. This problem could be solved by installing a program called XServer, but my WSL still wouldn't allow me to pull up a browser. This meant I had no way to enter my login details and the API couldn't run. This caused a substantial delay in progress, while I spent weeks trying to configure my WSL and it's chromedriver settings. This was really frustrating for me as it meant I couldn't focus on Software Development and building GNU Radio blocks, instead spending a lot of time playing around with the kernel settings on my WSL which was never the aim of this project. Eventually, I had to admit defeat and instead build a separate script which simply obtains the Auth0 token and saves it to a text file. I could run this script on Windows and then in the WSL start my API, which picks up the token in order to gain authorisation to the SmartLink server.

The problems with the WSL didn't end there though. Once I was finally getting the API running in the Linux environment, I went on to start building my FlexSource block in GNU Radio. However, when I started using the API calls, I was getting an error that I'd never seen up to that point. Specifically, on the first use of a TLS socket (Listing 5 line 43), I would get a "SSLEOFERROR". According to Python's ssl documentation[24], an end of file error is raised when the connection is terminated abruptly. Not much to go on, and any other information online was even more obscure or unhelpful. Even more frustrating was the fact that the client - who has a normal Linux OS - could run the code without issue. The SSLEOFERROR seems to occur randomly, and I haven't been able to fix it at this time. However, I'm confident it is due to compatibility issues with WSL and GNU Radio and the actual code is correct.

Personal Reflection

When reflecting on what I've achieved in these 13 weeks, I am happy with what was completed. Although I'd like to have gotten further - I met two of the three expected goals - I also created an API that's ready to be implemented in other projects and a way to use it in GNU Radio. I believe I managed the project well and was under control throughout. To keep track of the project, I created a weekly blog on WordPress[26], which allowed me to see how I was progressing and what stages were taking too long to complete. It was also invaluable when it came to writing this report, as I could see the clear path the project took; it's easy to forget what was done in the first few weeks of the project.

To ensure the integrity of the code I was writing, I used version control continually through the project. Admittedly, this was something I had not had much experience in prior to this module, but in doing so I have become much more confident in it. For example, during some of our review meetings, myself my supervisor and the client would look through some of my code to see areas that could be improved or might cause issues

further down the line. The client then updated the software with his suggestions and pushed to the git repository. However there would be occasions where I had made more changes to my own local branch before the meeting. Pushing these changes, wouldn't be immediately possible as the remote and local branches had become out of sync and the remote was ahead. To fix this problem, I had to *stash* my own commit, and interactively *rebase* my branch to be the same as the remote master. I could then retrieve my stashed commit and push it as normal. This would not be an uncommon problem in industry, as many more people would be working on an application and each would be using their own branch. It is therefore easy to see how branches can become out of sync and would need commands such as I described to realign them. Version control is fundamental in professional practices so I will be much better prepared when I enter the corporate world.

I also believe the communication and collaboration with my supervisor and the client was strong throughout the project. We held a meeting every week to discuss the progress being made and to try and resolve issues I may be having. We also had more formal "Review" meetings according to the dates laid out in my Initial Plan [27], where we took a broader assessment of the project and re-evaluated the goals. These review meetings were very beneficial as they reaffirmed to me that I was making suitable progress and the setbacks were indeed valid and unavoidable. It was great to have Derek Kozel involved as he provided real clarity when it came to working with GNU Radio. They always tried to be available as much as possible and were very responsive to all the questions I had; I don't think the project would have been nearly as successful if I had had busier supervisors. I'd also like to thank David Humphreys for his help in writing the function to access the Auth0 token, because as I've stated throughout this report this was a very challenging step to begin the project with.

I believe I would have had a stronger start if I could have begun from a local link to the radio. Instead, at the time of submission I still haven't seen the FLEX-6400 in person. Due to COVID-19 restrictions, I had to do all work on it remotely. While I know every project this year will have been affected by COVID, I feel projects involving hard-

ware such as mine are more deeply so because you don't get that physical interaction you otherwise would have. I also feel I was slightly hampered by the quality of the resources I had at my disposal: my laptop is over 5 years old and very slow. I would have wanted to work more in the university and make use of the better machines they had available. Cardiff did allow remote access to them via virtual machines, but installing the software I required was very difficult and I didn't have the flexibility in time to properly do this.

From a personal perspective, I gained so much from doing this project. Having come in with no knowledge of Amateur Radio and RF Communication, I have learnt not only the background of how this communication works but also how to create a way to interface with a Software Defined Radio. These are two things that will only be beneficial to me in the future. Given the extent of the problems I had and the sparsity of information I had to assist me, I believe the end product surpassed what could be expected. There will always being issues when creating software but how you handle them is what can make or break a project and I felt I did so more than adequately. In addition to developing my skills in software development by actually writing the library, I also gained invaluable skills in having to pick something up from scratch and learn about the system. It's very common in a professional environment like consulting to be given a new system and have to improve or develop it, with only some documentation and technical specifications to help you. I have no doubt that the experiences I've had doing this project will translate directly into my future work practices.

List of Figures

Listings

Abbreviations

API	Application Programming Interface
JWT	JSON Web Token
SDR	Software Defined Radio
SSL	Secure Sockets Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

Bibliography

[1] Opus (audio format). URL https://en.wikipedia.org/wiki/Opus_(audio_format).

[2] ARRL.org. Amateur radio saved lives in south asia. 2005. URL https://web.archive.org/web/20050101021716/http://www.arrl.org/news/stories/2004/12/29/100/?nc=1.

[3] Auth0. *Authorization Code Flow* documentation, . URL https://auth0.com/docs/flows/authorization-code-flow.

[4] Auth0. *Refresh Tokens* documentation, . URL https://auth0.com/docs/tokens/refresh-tokens.

[5] A. Bazlur-Rahman. Role of amateur radio in development communication of bangladesh. 2012.

[6] P. R. Brown. The influence of amateur radio on the development of the commercial market for quartz piezoelectric resonators in the united states. URL http://www.bliley.net/XTAL/Industry-Hams.html.

[7] FlexRadio. Smartsdr tcp/ip api, . URL http://wiki.flexradio.com/index.php?title=SmartSDR_TCP/IP_API.

[8] FlexRadio. Smartsdr tcp/ip subscription docs, . URL http://wiki.flexradio.com/index.php?title=TCP/IP_sub.

[9] R. Foust. Powershell module for flexradio. URL https://github.com/rfoust/FlexModule.

[10] N. Frissel et al. Hamsci distributed array of small instruments personal space weather station: Architecure and current status. Technical report, 2020. URL https://hamsci.org/basic-project/personal-space-weather-station.

[11] S. Gamboa. Puerto rico amateur radio operators are playing key role in puerto rico. 2017. URL https://www.nbcnews.com/news/latino/puerto-rico-amateur-radio-operators-are-playing-key-role-puerto-n805426.

[12] GNU Radio. Freedv demodulator block, . URL https://wiki.gnuradio.org/index.php/FreeDV_demodulator.

[13] GNU Radio. Gnu radio wiki, . URL https://wiki.gnuradio.org/index.php/Main_Page.

[14] GNU Radio. *Types Of Blocks*, . URL https://wiki.gnuradio.org/index.php/Types_of_Blocks.

[15] J. Linford. *FlexRadio 6000 SmartSDR - The Application Programming Interface, A Primer*, 1.002 edition.

[16] W. C. Lloyd. Ionospheric sounding during a total solar eclipse. Master's thesis, 2019. URL https://vtechworks.lib.vt.edu/handle/10919/89951.

[17] J. Noordhof. How does modulation work? *Tait Academy*, . URL https://www.taitradioacademy.com/topic/how-does-modulation-work-1-1/.

[18] J. Noordhof. What is propagation? *Tait Academy*, . URL https://www.taitradioacademy.com/topic/what-is-propagation-1/.

[19] Norton Security. What is an ssl certificate. URL https://uk.norton.com/

internetsecurity-how-to-ssl-certificates-what-consumers-need-to-know.html.

[20] Ofcom. *Amateur Radio Guidance: How to become an radio amateur on the UK*, 2019. URL https://www.ofcom.org.uk/__data/assets/pdf_file/0026/109547/guidance-become-radio-amateur.pdf.

[21] S. Phillips. Generating a pan adapter and waterfall display. URL https://community.flexradio.com/discussion/6683599/generating-a-pan-adaptor-and-waterfall-display.

[22] Python Software Foundation. *Coroutines and Tasks* documentation, . URL https://docs.python.org/3/library/asyncio-task.html.

[23] Python Software Foundation. *select* documentation, . URL https://docs.python.org/3/library/select.html.

[24] Python Software Foundation. *ssl* documentation, . URL https://docs.python.org/3/library/ssl.html.

[25] Python Software Foundation. *struct* documentation, . URL https://docs.python.org/3/library/struct.html.

[26] J. Slim. Implementing a gnu radio drive for the flex-6400. 2021. URL https://flexandgnuradio.wordpress.com/2021/02/22/implementing-a-gnu-radio-driver-for-the-flex-6400/.

[27] J. Slim. Initial plan: Implement a gnu radio driver for the flex-6400 sdr transceiver. 2021.

[28] Svartalf. opuslib gitrepo. URL https://github.com/orion-labs/opuslib/blob/master/tests/decoder.py.

[29] E. Wachsmann and E. Gonzalez. *SmartLink Quick Start Guide for SmartSDR for Windows*, 1.4 edition. URL `https://www.flexradio.com/documentation/smartlink-quick-start-guide-for-smartsdr-pdf/`.

[30] D. Whitten and D. Rowe. Freedv: Open source amateur digital voice. URL `https://freedv.org/`.