GNU Radio Blocks and Flowgraphs

Reference

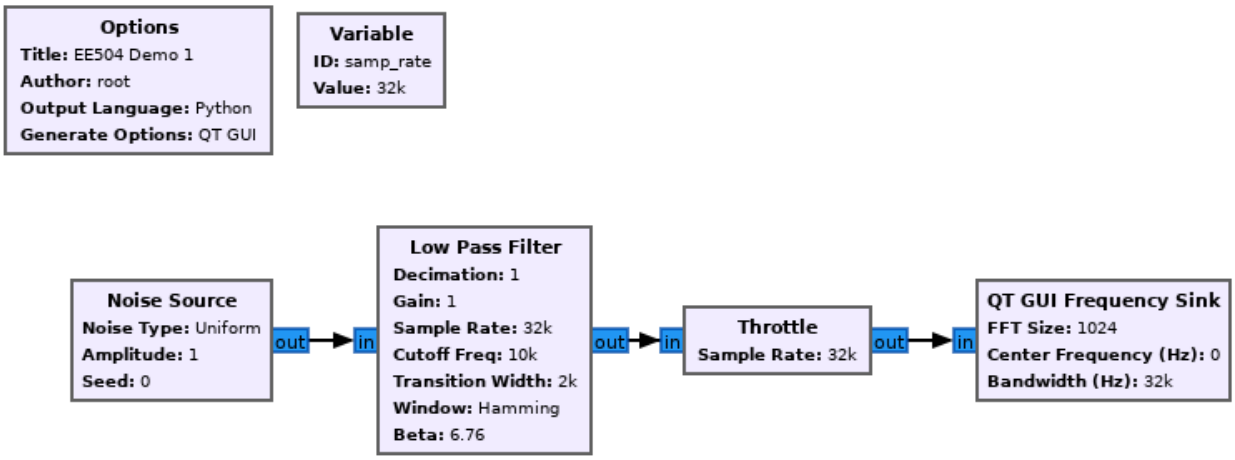
- GNU Radio Block Wiki Search
- QPSK Modulation Example
-

GNU Radio Block Basics

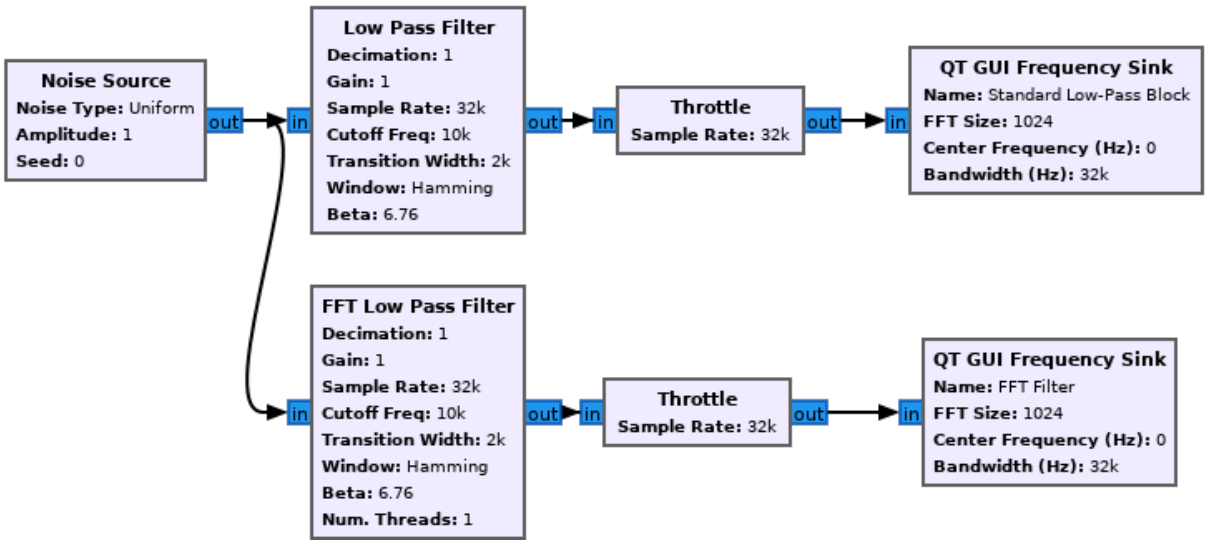
Last week, you were introduced to the basic GNU Radio environment, the most basic blocks, and built a simple system with a GUI. This lab, we're going to build more complex blocks, including our own custom Python blocks.

The first new blocks we'll look at are filters. GNU Radio provides options for how you choose to filter your signal. We will focus first on the simple filters:

Generate the following flowgraph using the low-pass filter:

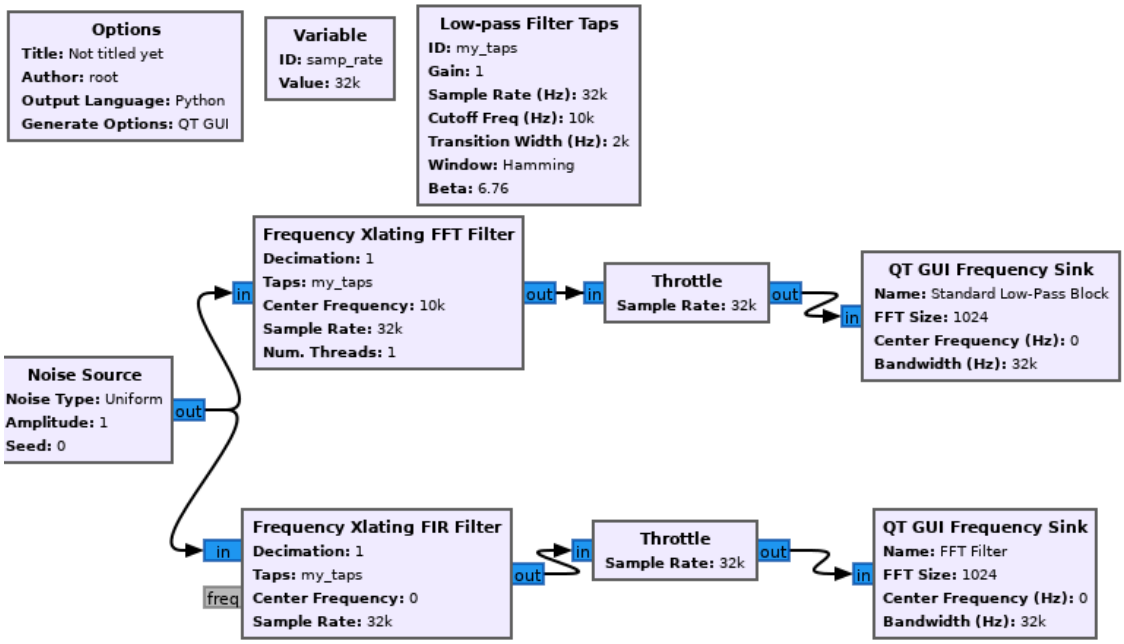


It's pretty easy to see how the standard low-pass filter works. Our uniform noise source gives us an easy way to check the relative "shape" of our filter's frequency response. Next, we're going to add the FFT Low Pass Filter block by adding a second branch and Frequency GUI:

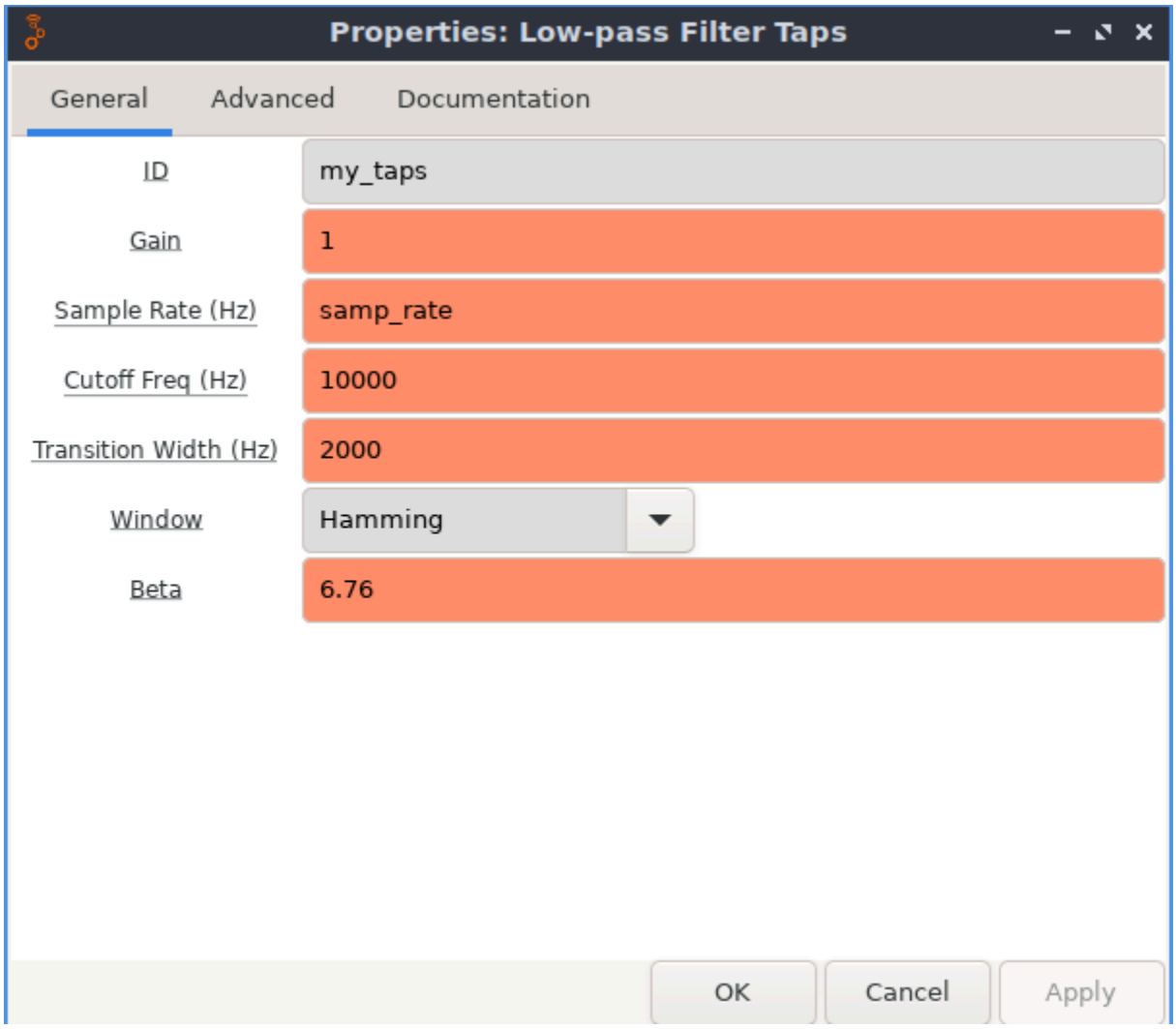


When you run this flowgraph, you should see these will output graphs with a similar response. They essentially function the same, even though the Low Pass Filter is based on an FIR filter instead. If you were to increase the number of threads in the FFT block, it would theoretically perform better, however the point of sampling rates is so that our hardware resources are not completely taken over by the program. The “Low-Pass” FFT Filter is just a wrapper of the standard FFT Filter block we will be looking at next.

Replace the Low Pass Filter and FFT Low Pass Filter blocks with the Xlating FFT Filter and Xlating FIR Filter. Search for the “Low Pass Filter Taps” block and add it to the flowgraph. It will look something like this, without all of the data filled in:



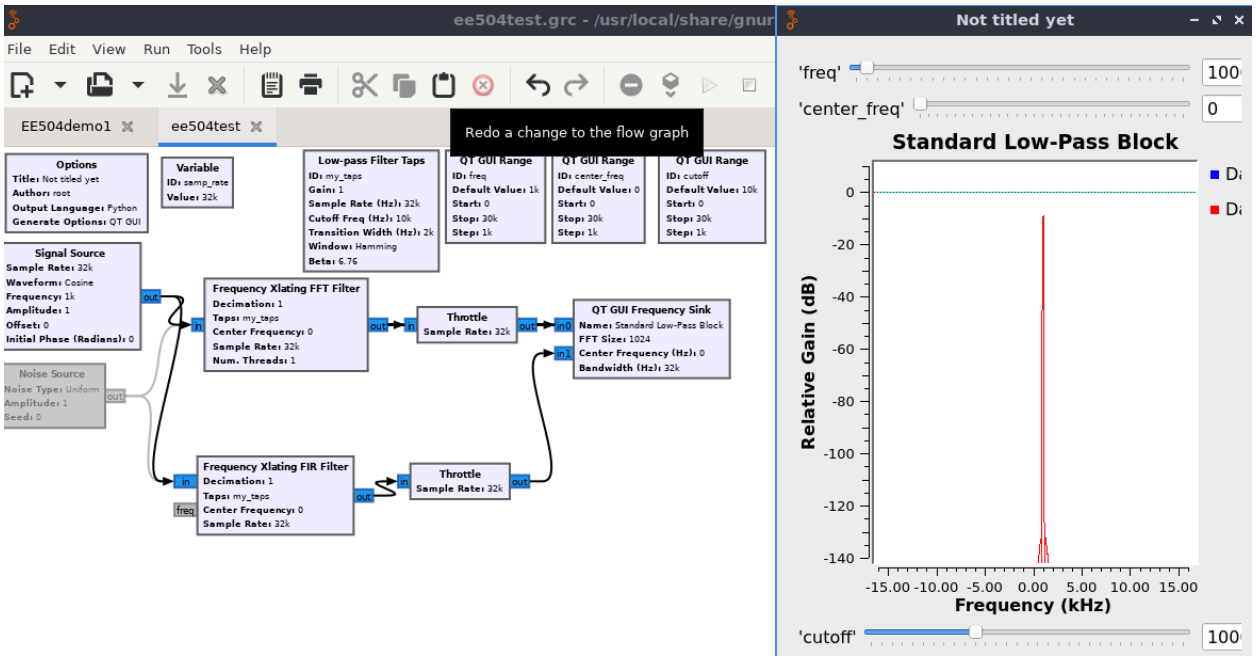
Double click on the Filter Taps block and change the parameters to match the same low pass filter blocks we just used. Name your taps variable something so that you may reference it in the Filter blocks:



T

And finally add your taps' variable name to the Filter blocks. Run the simulation and again, we should see the graphs look similar. Let's directly compare their responses to a given symbol now. Remove the bottom GUI Sink and double click the remaining one. Go to "Number of Inputs" and change it to 2. Connect the FIR and FFT filters to the same GUI Sink. Click on the Noise source and disable it by pressing "D". Add a new signal source block to the flowgraph and connect it to the inputs of the filters.

Next we're going to make multiple sliders for different parameters we want to change. Create new variables with the IDs "freq", "center_freq" and "cutoff" that all start at 0Hz, end at 30kHz, and have their default values to reasonable numbers within their range. Your flowgraph should look something like this:



Use the sliders to confirm the two filters have an essentially identical response to a given input frequency, cutoff and center frequency. Take a screenshot of your flowgraph and GUI.

Questions to consider:

1. If the FIR and FFT filter function almost identically, what is the point of having both? (Think about hardware/computing cost?)
2. Is it easier to use the general block and modify the taps parameters or use the wrapper blocks (the first filters we used)?

Writing Your Own Block

Embedded Python blocks can be extremely helpful for specific data processing or control applications that the standard GNU Radio blocks just can't achieve. For a block to be worth writing ourselves, we first need a use case. There is a "Stream Mux" block that exists in GNU Radio, however it only combines the input streams into a vector stream. We will design our own mux that selects one of the inputs and allows us to switch it.

Create a new flowgraph and create 3 signal sources representing Cosine, Square and Sawtooth waves. Make them all the same frequency and all output types float instead of

complex. Add the “Python Block” block to the flowgraph and double click on it, select open in editor, and open the python file in whatever text editor.

```
"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Embedded Python Block',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.example_param = example_param

    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0] * self.example_param
        return len(output_items[0])
```

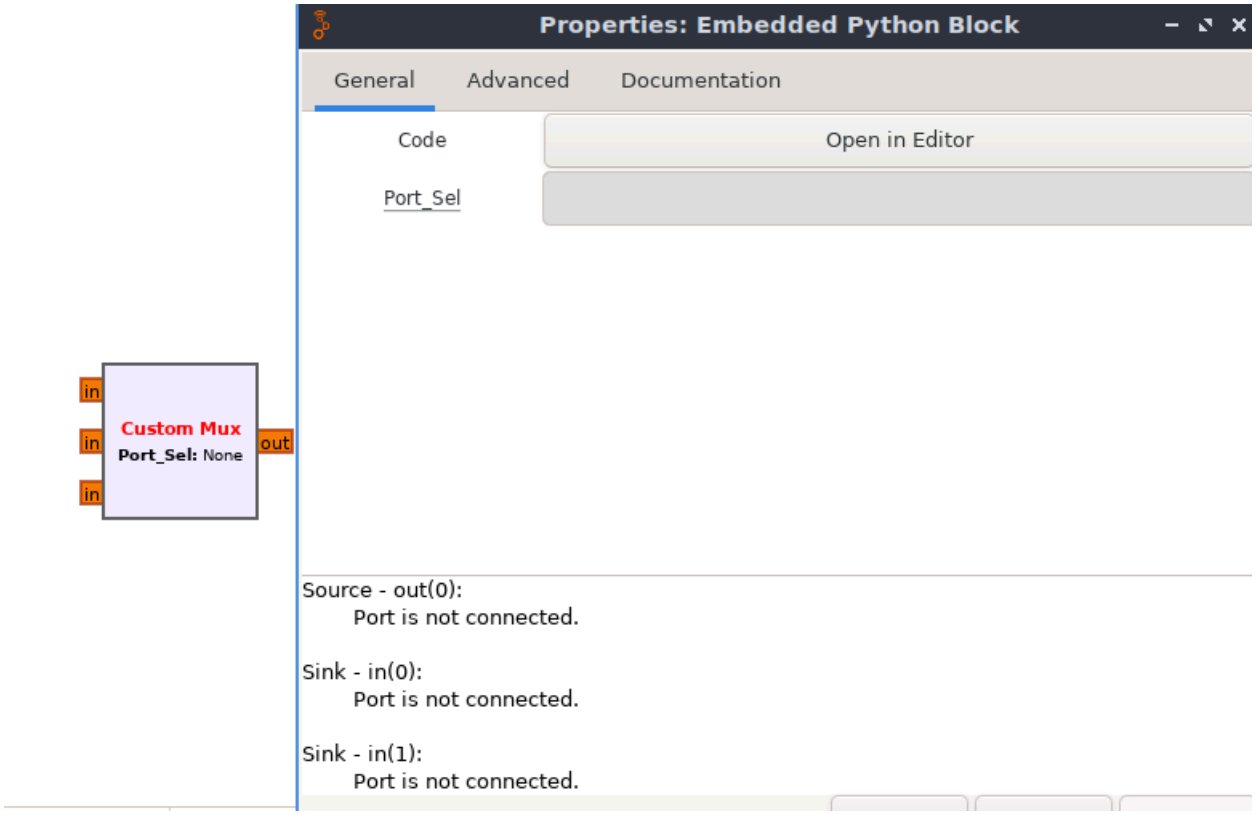
The init function defines how the block is initialized as well as seen in the flowgraph. Parameters in the function definition will be shown as parameters in GRC that you can enter values into (i.e. example_param). Modify the init definition so that there are 3 np.float32 inputs, one np.float32 output by changing the data types and adding 2 more inputs into the in_sig list. Change the name of “example_param” to change the display name in the GNU Radio block GUI, we will use this parameter to select between inputs 0-2. Changing the “name” variable in initialization will change the name displayed on the block in GRC, so name it something like “Custom Mux”.

The work function defines the work the block will do continuously while running. For our multiplexor, we want to use our port selection parameter to choose which of the input

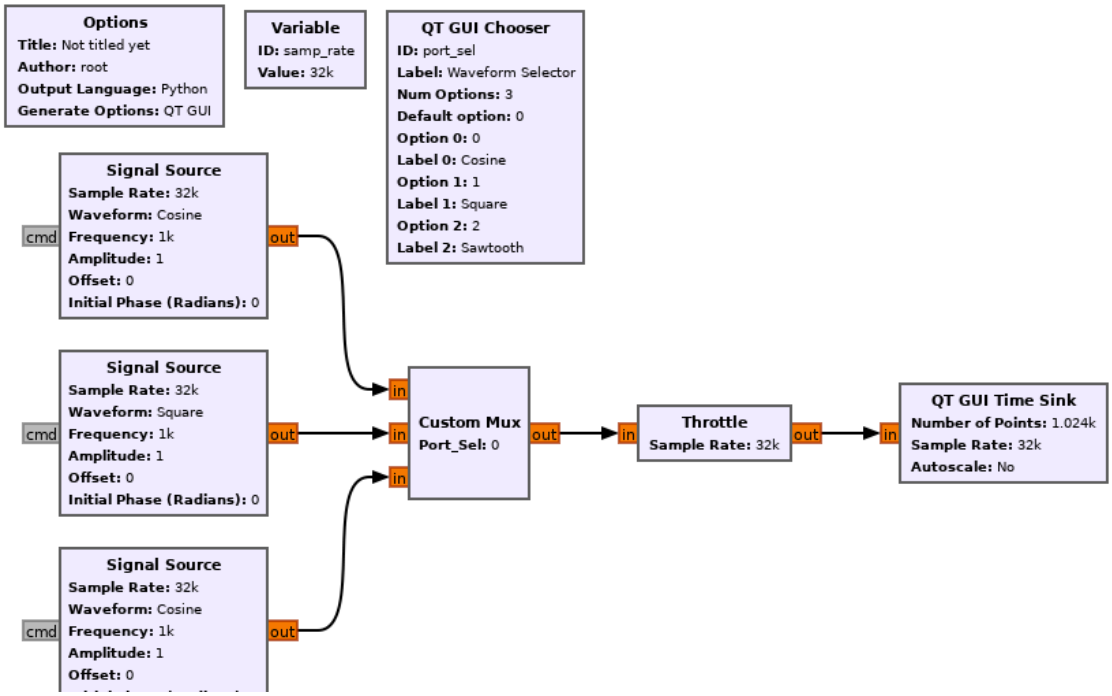
ports is outputting to the output port. Here is the code to map input port 0 to a output port 0 to get you started. To select other ports, change the index of input_items:

```
output_items[0][:]  = input_items[0]
```

One more tip, leave the return statement alone. All work done and outputs will be accomplished by mapping the input items to the output items function parameters, nothing “returning” to another function. When you think you’ve successfully written code for this, save and close the editor. If done correctly, your block will look something like this, and all ports in the settings will have no errors like so:



If errors exist in your code, they will be shown under the source and sink sections in the GUI, or GNU Radio will write out the crash that occurs to the terminal when you try to run it. Finish configuring your flowgraph with a throttle and Time Sink GUI. Add a GUI Selector block with 3 options for 0, 1, 2 representing the different waveforms. The result will have these blocks connected:



Run the flowgraph and you should be able to select between 3 different inputs using a GUI Chooser.

Sometimes we want to have procedural behavior controlling our switching instead of user input. To do this, we will use the asynchronous “messages” in GNU Radio and an additional Embedded Python block to house the control logic. Start by making a new embedded python block like you did before, right after the first, and opening the editor.

We will want np.float32 input and output again, and name it something along the lines of “Mux Control”. Add init parameters for sample rate and switch time. Use the switch time and sample rate to calculate the target number of cycles before switching. We will define this as self.numSamples, meaning we will not store the sample rate, frequency, or cycles. Make a self.port_sel variable that represents which port will currently be selected, initialize it to 0. Define the self.portName parameter and set it to a string representing whatever you want to name this message port. In your file add this import to the top:

```
import pmt
```

And this line the the initialization function:

```
self.message_port_register_out(pmt.intern(self.portName))
```

This creates the output message port with the specified port name. A PMT, or Polymorphic Type, is just a safe datatype GNU Radio uses to pass any kind of data through messages safely. There is a lot to learn about GNU Radio's PMTs, but in essence they all consist of a "key" data type in the form of a directory, and the "value" data type holding the actual value mapping to that key. This lets us safely pass data while still having the necessary information to convert this data back to its original data type. What we do here is simply initialize a PMT data type with our specific port name.

Now we just need to finish the actual control logic. Your implementation of this logic may vary, however the idea is to increment a counter by the number of items being processed by the work function, until it hits our calculated numSamples, where it then switches to the next input. Creating counter and state variables will help keep track of this and easily write it out. Here's one hint for actually sending out the PMT message which you might not yet know how to do:

```
msg = pmt.from_long(self.selected_port)
self.message_port_pub(pmt.intern(self.portName), msg)
```

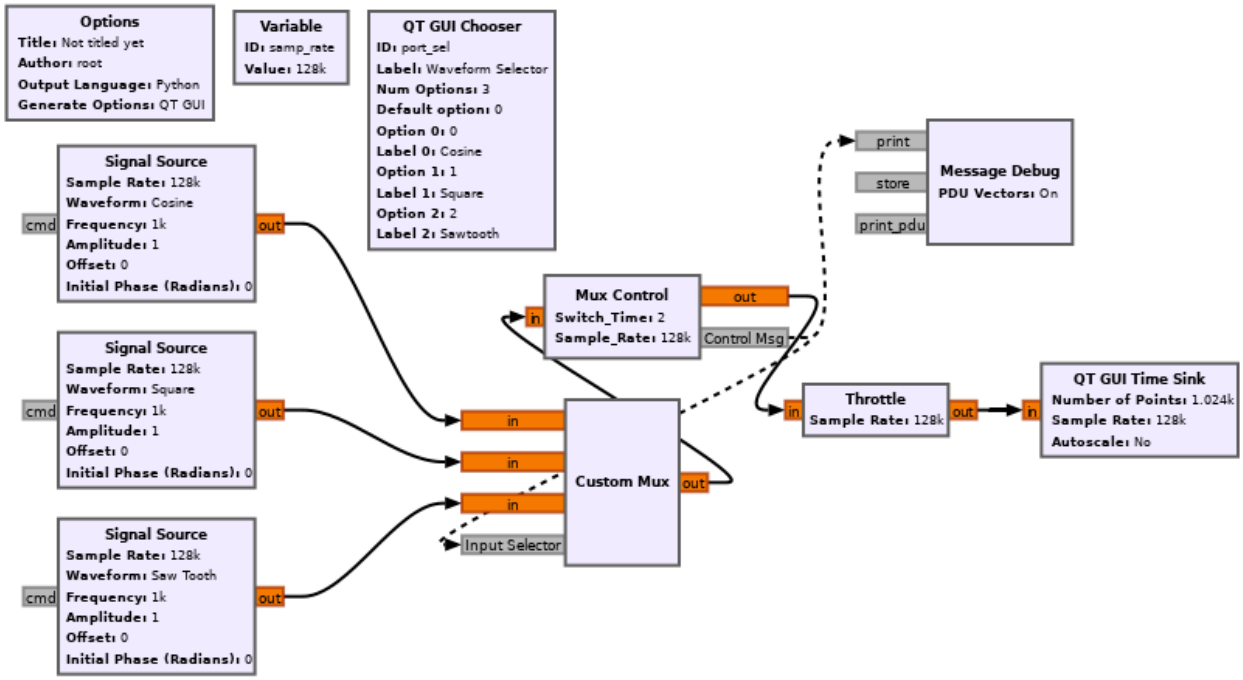
The .from_long method is called to safely convert the datatype of our selected port variable to the PMT, and .message_port_pub sends out the PMT on the specified port with our new PMT called "msg".

Once you have completed the logic to send out the selected_port message, we need to modify the mux to accept and react to this control. Double click our mux block to edit the code, and remove the selected port parameter from the function header, but leave it as a variable since we will still use it. Create a message port register in the same way you did with the control block, but switch "out" with "in", and name it something. We will add one more line to initialization:

```
self.set_msg_handler(pmt.intern(self.selPortName), self.handle_msg)
```

This configures our block to run the "handle_msg" method when a message is input into the block. You will need to define this method in the code and handle converting the data from PMT back to a standard datatype python can use. (Remember "from_long" to change to PMT, well there's also "to_long" to change back).

Once this is done save your block and connect the control block properly in the flowgraph, including routing the messages between the two gray pads.



A Message Debug block was also added to read the data from the messages. This is also an effective way of debugging blocks or receiving status updates through the terminal. When you run the flowgraph, set the switch time anywhere from 0.5-2.5 seconds and take a video of the waveforms switching on the graph.

Questions to Consider:

1. What are the benefits and disadvantages of asynchronous information in flowgraphs?
2. What other advantages might the message debugger have if you spend the time to write debug messages?

Streams and Tagging Streams

We have one last GNU Radio “feature” to go over before we’ve covered the basics and are ready to build systems. We already know that streams are what GNU Radio calls the path that data flows from one block to another (and through a block). GNU Radio has tags that can be “attached” to streams of data representing your signal. Tags are really an additional stream of data that is synchronous with the underlying datastream. Streams have a “key” and “value” represented by PMTs where the key describes its respective data stored in “value”. Here’s what the GRC Wiki says:

> “Tags are a way to convey information alongside digitized RF samples in a time-synchronous fashion. Tags are particularly useful when downstream blocks need to know upon which sample the receiver was tuned to a new frequency, or for including timestamps with specific samples.” -GNU Radio Wiki

There are already many tagging blocks already built into GNU Radio. Tags can be used in many ways, but become incredibly important when dealing with packetized data. Tags can be used to describe the length of packets, store information about the packets, then be read further down the line to create headers, preambles, or other packet-specific data.

For this example, we’re going to stick to very simple tags, just to get experience adding tags to a stream and reading tags from a stream, including in our own python blocks. We’ll use the same example we were just working on. Add a “Tag Strobe” and “Add” blocks to your flowgraph. The number of samples in the Tag Strobe block tells you how many samples before the next tag is added. Set this to your sample rate divided by around 100. Play around with this number to find a good rate where there are always tags on the graph but they do not clutter it when the flowgraph is run. Change the names of both the key and value of the tag, and then use the add block to add the tag to the stream after the control block. Run the flowgraph and take a picture of your graph with at least one tag on it.

Next, we’re going to add tags in our Mux block so that we can also practice reading tags from a string in our control block. Again, these will be fairly meaningless tags. Open the code for the mux and add two variables for a “timer” and a “ready to tag” flag. Next, make a for loop that goes through all of the items in the current selected input and checks if any of them are between -0.1 and 0.1. Create two PMT’s holding the key “Near Zero” and the value (voltage) of the sample, and then use this function to create a new tag:

```
self.add_item_tag(0, #output port 0
    self.nitems_written(0) + idx, #absolute index of tag
    #HINT: idx is the idx used to loop through all items in
    input_items[0]
    key, #the key PMT you made
    value #the value PMT you made
    )
```

Use the timer and tag_ready variables to make sure you aren't spamming tags for every sample near 0. Take a screenshot of the graph with the near 0 tags. Once this is working, open the control block so we can add some simple tag reading code. Start by adding a float32 output to the block, and a tag_counter variable. Every time "Near Zero" tags are read, add the number of tags to the counter. Have this counter reset at some point so the value does not increase indefinitely. Here is a line to get all of the tags in the window:

```
tagTuple = self.get_tags_in_window(0, 0, len(input_items[0]))
```

Connect the second output to its own Time Sink GUI, and take a screenshot of the two graphs together showing the output incrementing every time the other graph nears 0. When you run this, you will notice the tags have also propagated to the second output. By default, all custom GRC Blocks will map all the input tags to all the output ports as well. This can be changed to map one port to one port, or to stop all tag propagation together. Instead we will connect the second output to the "tag gate" block before routing it to the GUI, so that only the waveforms have the tags. Take a screenshot of this.

You now know the basics of using, adding, and grabbing tags. Everything above can get way more in-depth than what we have covered, and I encourage you to read the GNU Radio wiki pages if there is any confusion, as they helped shape this tutorial significantly.

Questions to Consider:

1. What is the main difference between tags and messages? What are the use cases for both?
2. Why are tags particularly useful for formatting PDUs/packets in flowgraphs?

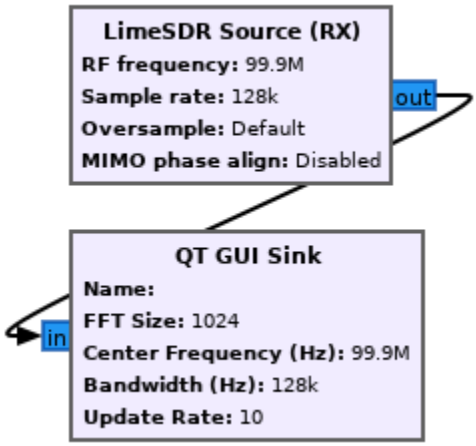
Hardware and Real Systems

Finally, we're ready to build real systems and actually use the LimeSDR. Save your last flowgraph and open a new one. Add a LimeSDR Source block and QT GUI Sink block to your flowgraph to start, and connect them. Add QT GUI Range blocks for variables for the center frequency, sampling rate, and gain. Put the variables in the correct place in the GUI Sink and LimeSDR Blocks, documentation about the LimeSDR block's parameters can be read in the "documentation" page of the block.

Once this is completed, you've made a simple spectrum analyzer in GNU Radio using your LimeSDR. Run the program to ensure you are seeing noise on the graph, and use the sliders to adjust the parameters.

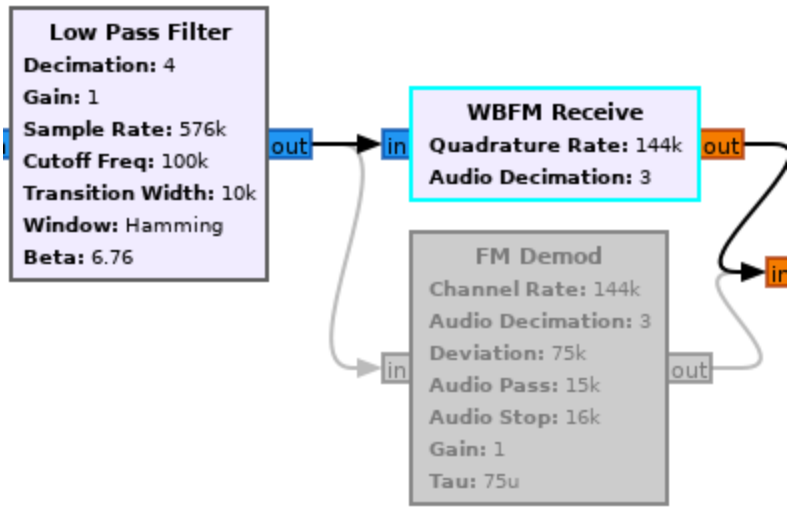
Note: You can find the LimeSDR Mini's serial by running LimeUtil –find in the terminal, or the block will automatically connect to the first LimeSDR it sees when the flowgraph is run.

Question: What does the center frequency of the LimeSDR block do?



Now that you've verified the LimeSDR works in GNU Radio, let's make something slightly more interesting and applicable. You can leave the GUI Sink, but now add your own Low Pass Filter block. The specifications are a cutoff frequency of 100kHz and a window of 5kHz, and we want to use decimation to reduce the resources needed to run out flowgraph, especially since we will not need incredible amounts of bandwidth to demodulate this signal. The decimation factor you choose will depend on the sample rate you choose and the overall output audio rate you desire, so this can be adjusted later, however the typical audio rate is 48kHz.

Next, add the "WBFM Receive" and "FM Demod" blocks to the flowgraph after the filter, all their own path. We will disable one while we use the other, having both in the flowgraph simply helps demonstrate that there are multiple ways to modulate and demodulate the same signal in GNU Radio.



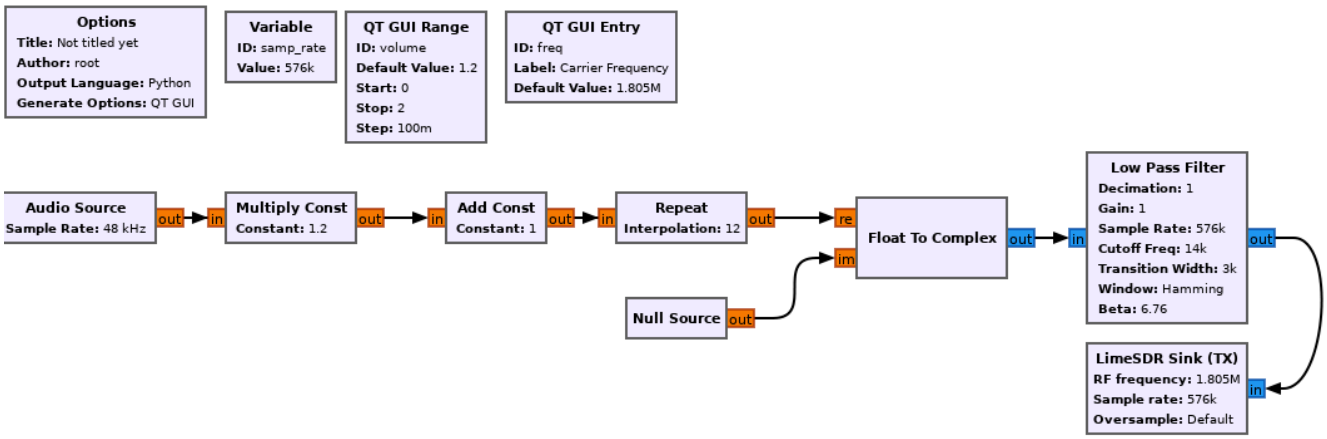
The quadrature/channel rate of the block will be your original sample rate divided by your decimation factor. Calculate an audio decimation factor to result in an output of 48kHz from out of this block. You may need to adjust the previous decimation factor or your sample rate. You can also achieve this by using the “rational resampler” block that allows for both interpolation and decimation factors, just make sure your math is right!

Connect the outputs of these blocks to a Multiply Constant and then connect an Audio Sink to finish the data path. Add a QT GUI Range going from 0-1 named “volume” and put this variable in the multiply constant. Run the flowgraph with only 1 demodulator active at a time and find a radio station using the original frequency range slider. When the flowgraph is run, the Audio Sink will output the audio to your default device if the device field is left blank. Record a video of reception of an FM radio signal, or ask for help if you cannot find or hear one. Switch which demodulator you are using to verify they both work, showing that there are multiple blocks/ways to do the same thing.

Now It’s Your Turn

The last deliverable is to design a small receiver yourself based off of the simple AM transmitter below. You can use the AM Demod block, but the same effect can be achieved with the below blocks as well (parameter values should be changed)

The receiver really won't take many blocks, and the procedure for building the last receiver can more or less be followed, but for an AM signal instead. Here is the Transmitter, reference this image to build it and use the parameters to design your receiver. If your device does not have a microphone, you can download a .wav file and use the WAV Source block instead of Audio Source.



Additional New/WIP GNU Radio tutorials:

https://wiki.gnuradio.org/index.php?title=Tutorials_2024