GNU Radio-Companion Cook Book
Tips, Tricks and Design Patterns

One of the major design goals of GNU Radio-Companion is modularity and one of the ways that this is accomplished is by ensuring that the official in-tree blocks are atomic. In other words, each one only does a single simple thing. Of course there are exceptions, such as the Frequency Xlating FIR Filter block, which can decimate, shift frequency and apply a filter. In general though, they are each very simple. By stringing together many of these simple blocks you can make complex applications with a high degree of granularity.

The downside to this approach is that it dramatically steepens the learning curve. In order to do fairly simple things it is necessary to have a working knowledge of the platform itself as well as a detailed understanding of many different blocks and their various quirks. Not to mention all the DSP stuff that a newcomer is probably unfamiliar with. Complicating this situation even further is the… eclectic documentation for the blocks.

In my personal experience, this has lead to reinventing several different wheels, sometimes more than once. It finally struck me that when you find yourself struggling to do some relatively simple task in a conventional programming environment you can usually fall back onto some well established design pattern.

With that in mind, I decided to document some “design patterns” for GRC. What developed was a collection of solutions to common problems, some of which could be seen as design patterns if you squint just right. These aren’t complete documentation for blocks, nor are they tutorials for building parts of a radio or DSP system. Rather, this is a collection of quick solutions for ordinary problems. It is important to note that the problems given could have several other solutions besides the ones shown here. These are just the solutions that have worked for me.

Please note: I purposely omit throttle blocks from my examples in order to make the pictures more concise. If your graph needs a throttle block, use one!

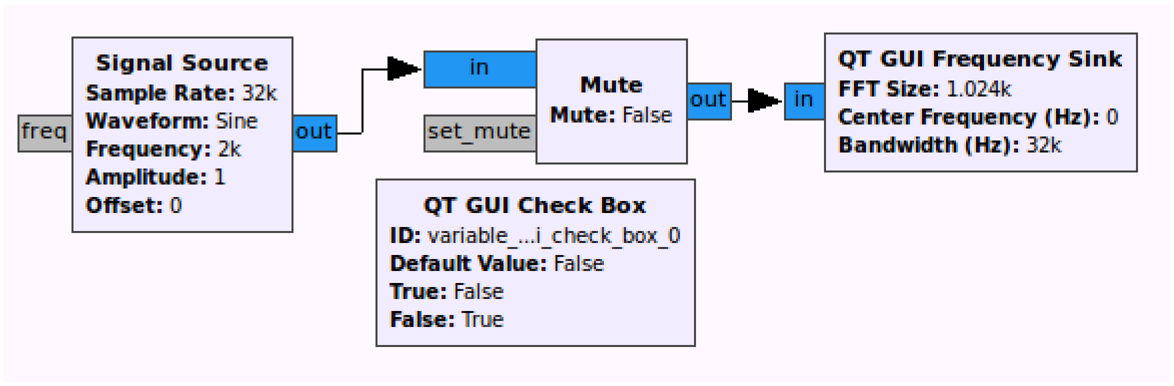
The solutions documented are (in no particular order):

Problem 1: Turn a stream on or off from the GUI

Solution: Use the Mute block in combination with a QT GUI Check Box block.

Set the QT GUI Check Box type field to “Any” or “Boolean” (it works either way) and use the values True and False (no quotes, case sensitive) for the values. False means not muted, true means muted. Then copy the widget ID into the Mute block’s “Mute” field. When the stream is muted it will continue to pass samples but they will all have a value of 0.

Any QT GUI widget can be used instead of a check box as long as you can set the type to “Any”, “Raw” or “Boolean”. You can also use a Variable block for non-GUI use cases.


Signal Source
Sample Rate: 32k
Waveform: Sine
Frequency: 2k
Amplitude: 1
Offset: 0

Mute
Mute: False

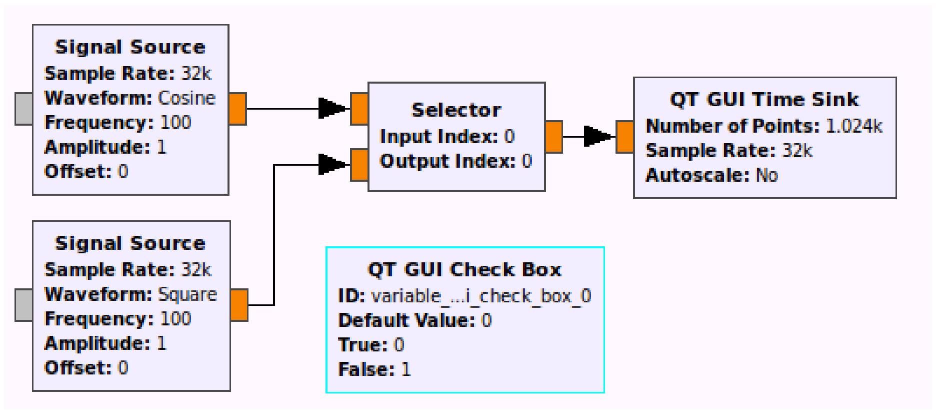
QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

QT GUI Check Box
ID: variable_...i_check_box_0
Default Value: False
True: False
False: True

Problem 2: Multiplex or demultiplex signals

Solution: Use the Selector block in combination with a QT GUI Check Box or QT GUI Chooser block. Please note that this does not do the same thing as the Stream MUX block which merges multiple streams into a single stream. This chooses between multiple streams.

To multiplex two signals, set the Selector "Num Inputs" to 2 and "Num Outputs" to 1. Make the "Input Index" the ID of the QT GUI Check Box and the "Output Index" 0. Set the GUI widget type to "Int". Setting the check box value to 0 selects the first (top) signal, setting it to 1 selects the second. You can have many more than two inputs in which case you would need to use a QT GUI Chooser block so that you can make more than two possible choices.



Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 100
Amplitude: 1
Offset: 0

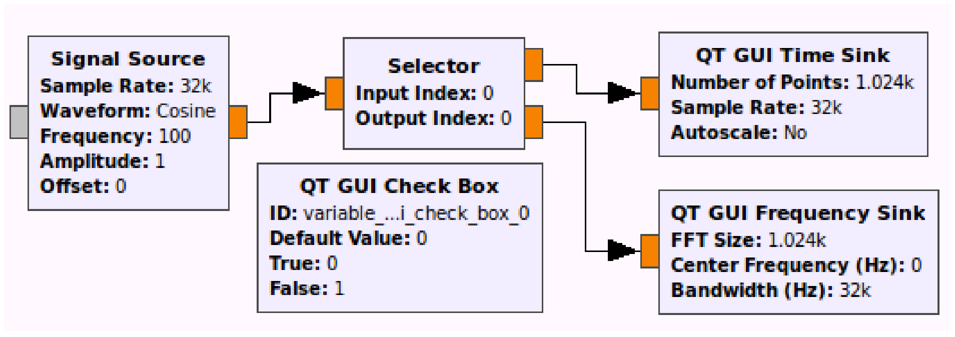
Signal Source
Sample Rate: 32k
Waveform: Square
Frequency: 100
Amplitude: 1
Offset: 0

Selector
Input Index: 0
Output Index: 0

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

QT GUI Check Box
ID: variable_...i_check_box_0
Default Value: 0
True: 0
False: 1

To demultiplex two signals, reverse the above values in the Selector blocks "Num Inputs" and "Num Outputs" fields. Then reverse the values in the "Input Index" and "Output Index" fields. Adjust the QT GUI Check Box settings appropriately.



Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 100
Amplitude: 1
Offset: 0

Selector
Input Index: 0
Output Index: 0

QT GUI Check Box
ID: variable_...i_check_box_0
Default Value: 0
True: 0
False: 1

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

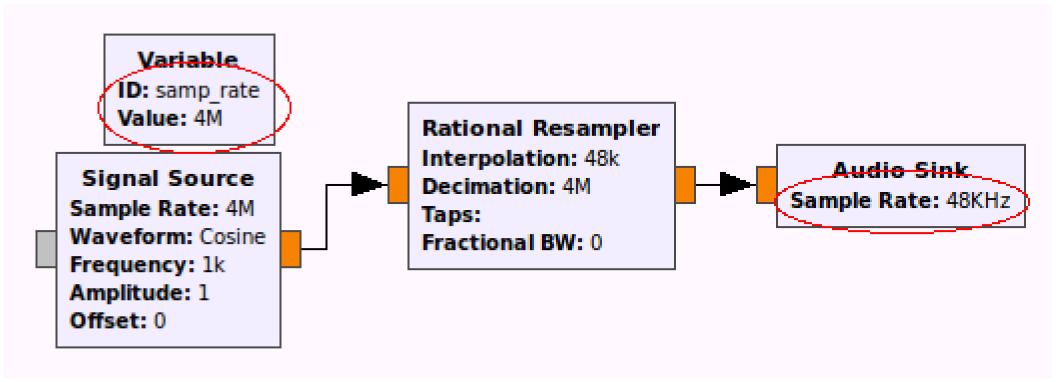
QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

More elaborate uses of the Selector block can be made involving many inputs and many outputs. In such a case you can use two QT GUI Chooser blocks; one to select the input and one to select the output. However, the block never maps more than one input or more than one output at any given time. You can't have one input go to multiple outputs or vice versa.

Problem 3: Match dissimilar sample rates

Solution: Use a Rational Resampler block to change the upstream sample rate to the downstream sample rate without altering the frequency range of the stream.

Set the Rational Resampler block's Interpolation field to the downstream sample rate and the Decimation field to the upstream sample rate. It is important to note that these fields are expecting integer values so it is not possible to enter a floating point value such as "4e6". In such cases you will need to cast the value to an integer by entering int(4e6). You can cast to an integer even if the value is a variable such as int(samp_rate).



Variable
ID: samp_rate
Value: 4M

Signal Source
Sample Rate: 4M
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0

Rational Resampler
Interpolation: 48k
Decimation: 4M
Taps:
Fractional BW: 0

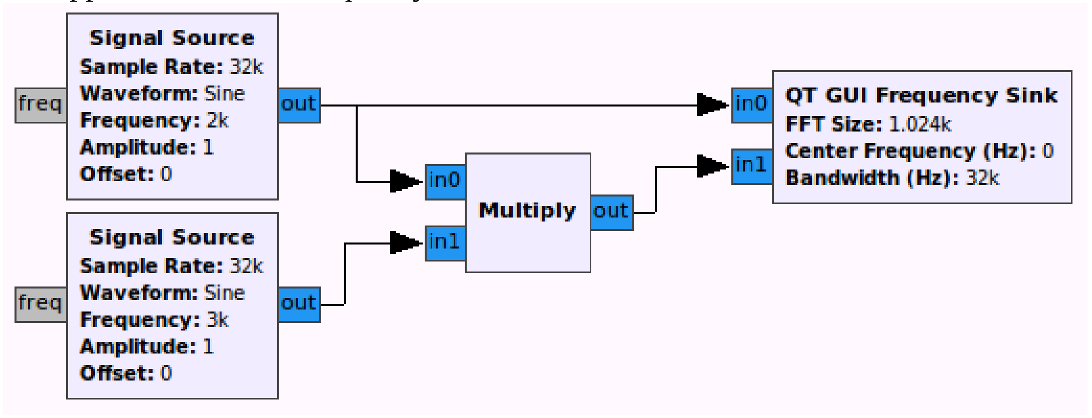
Audio Sink
Sample Rate: 48KHz

It should be noted that decimating without filtering will cause aliasing. If the decimation value is anything other than 1, a low-pass or band-pass filter should be applied.

Problem 4: Shift the frequency of a signal without re-sampling or filtering

Solution: Use a second Signal Source block and a Multiply block to change the existing frequency to a different one.

Set the Signal Source block’s waveform field to sine wave and the amplitude to the same as that of the original signals (assuming you don’t want to amplify or attenuate the shifted signal). Set the frequency field to the the number that, if *added* to the original frequency, will result in the desired target frequency. For example; you have a signal at 2Mhz and want it at 5Mhz so you set the second Signal Source to 3Mhz because 2 + 3 = 5. This can be a negative amount for down-shifting. Connect both Signal Source blocks to the Multiply block. The resulting signal will be identical to the original, but now appears at the new frequency.

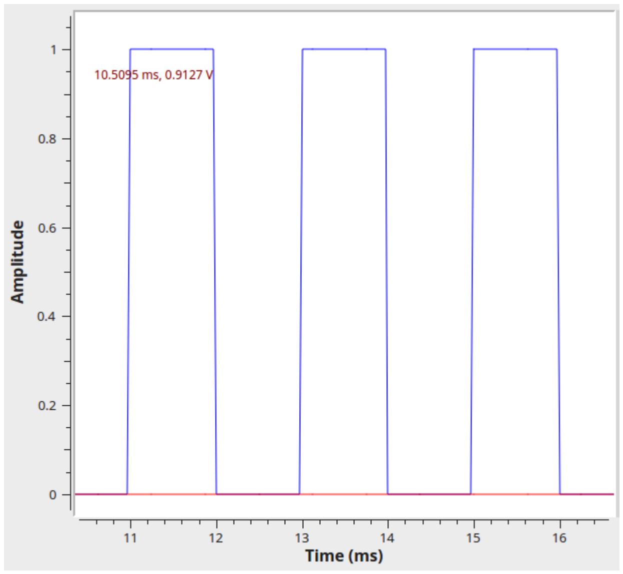


This can also be accomplished with a Frequency Xlating FIR Filter block with decimation set to 1 by setting the Center Frequency field to the amount you want to shift the frequency, negative or positive. However, this block also forces you to put in a filter which introduces additional overhead. If the signal needs to be filtered anyway, this method is preferable.

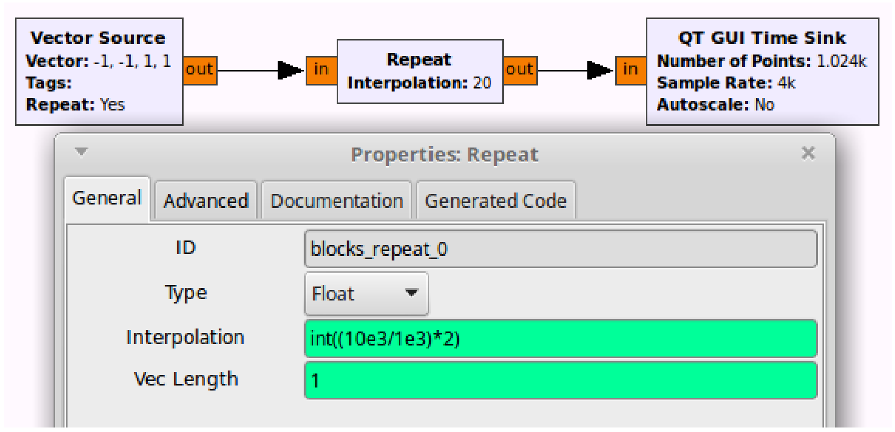
Problem 5: Match the period of a digital symbol to a desired time period

Solution: Use a Repeat block to extend the period of a digital symbol to a particular length of time. Use a Repeat block and a Keep 1 in N block to reduce the period of a digital symbol to a particular length of time.

First use a QT GUI Time Sink block to measure the length of the existing digital symbol. The below example shows a peak-to-peak signal that is 2ms long with 1ms long symbols.



If you wish to increase the period of the symbol, use a Repeat block and place the following formula into the Interpolation field: int((desired_length/existing_length)*2). Replace desired_length and existing_length with appropriate values. For example; if you want to convert the 1 ms symbol length to a 10ms long symbol, then the final formula would read: int((10e3/1e3)*2). The multiplication by 2 at the end is to account for the difference between measuring a digital symbol and a peak-to-peak wavelength.



Vector Source
Vector: -1, -1, 1, 1
Tags:
Repeat: Yes

Repeat
Interpolation: 20

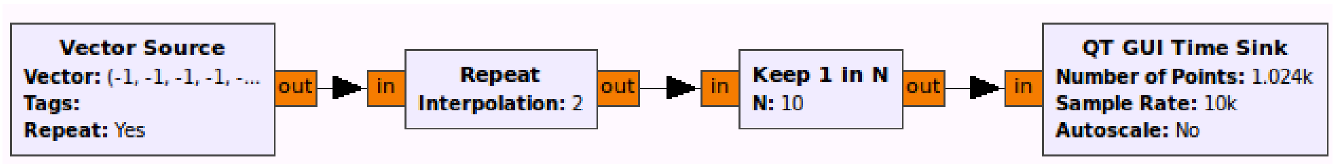
QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 4k
Autoscale: No

Properties: Repeat

General | Advanced | Documentation | Generated Code

ID	blocks_repeat_0
Type	Float
Interpolation	int((10e3/1e3)*2)
Vec Length	1

If you wish to decrease the period of the symbol, use the Repeat block followed by the Keep 1 in N block. Set the Repeat block's Interpolation field to 2 at first. Now set the Keep 1 in N blocks N field to the following formula: int((existing_length/desired_length)*1). For example, if you wanted to change the above 1ms symbol length to a 0.1ms symbol length, the formula would read: int((1e3/1e2)*1). The Repeat block doubles the number of points to account for the difference between symbol length and peak-to-peak wavelength. The Keep 1 in N block then reduces the period. The difficulty is that you might reduce the number of points down to only 1 point per period or even miss symbols altogether. In such cases you will get a QT GUI Time Sink output that looks like a triangle wave. If this happens, increase the Repeat blocks Interpolation field by an order of magnitude (20) and the 1 in the above formula by an order of magnitude (10). Check the time sink output again. Continue increasing these two fields by 1 order of magnitude until you have a square wave again.



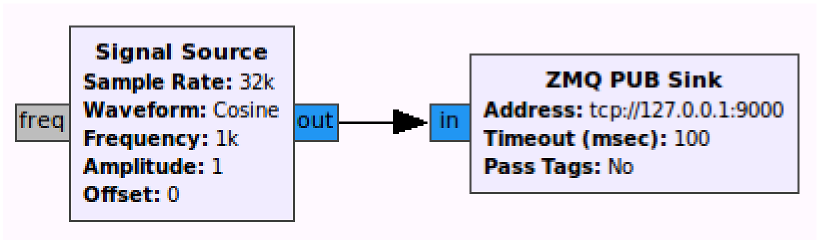
In all cases, a change to the overall sample rate will also change the symbol rate.

Problem 6: Stream data over a network

Solution: Use a ZMQ PUB Sink to publish the stream to a network socket and a ZMQ SUB Source to subscribe to that network socket.

ZMQ is short for ZeroMQ (the MQ stands for message queue) which is a library for message passing. The ZMQ sinks and sources work better than the TCP sinks and sources in my experience. There are various other sinks and sources available in the ZeroMQ Interfaces section of the Block Tree Panel that pair together to pass messages in various ways. These are PUB/SUB (Publish/Subscribe), PUSH/PULL, and REP/REQ (Reply/Request). Each pair type has advantages and disadvantages so you can research and choose the best type for your situation. This demonstration uses the PUB/SUB blocks.

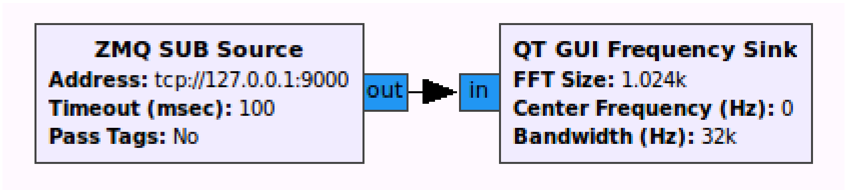
On the computer serving the data, use a ZMQ PUB Sink and put tcp://ipaddress:port in the address field. Use the serving computers IP address and an unused port. For testing purposes with the PUB and SUB blocks on the same computer, use the localhost address: 127.0.0.1.



Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0

ZMQ PUB Sink
Address: tcp://127.0.0.1:9000
Timeout (msec): 100
Pass Tags: No

On the computer subscribing to the published stream, use a ZMQ SUB Source. Put the same data as above into the address field. Both computers need to be on the same local network. You can increase the timeout length if the network is particularly slow, but the source and sink timeout fields should match.

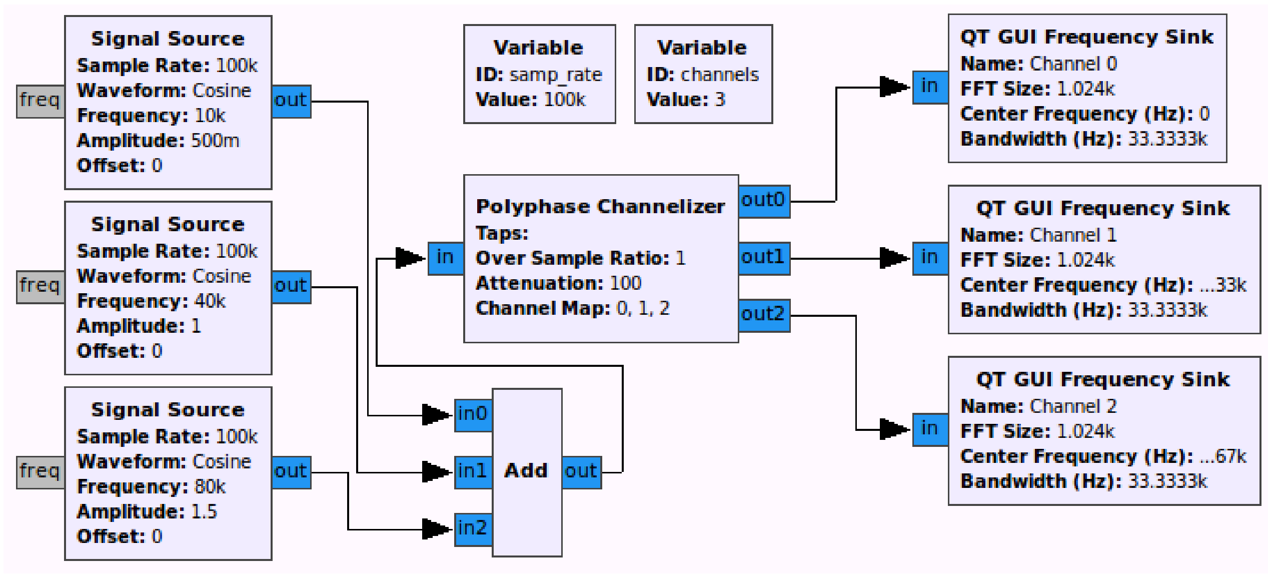


ZMQ SUB Source
Address: tcp://127.0.0.1:9000
Timeout (msec): 100
Pass Tags: No

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

Problem 7: Divide a section of bandwidth into multiple channels

Solution: Use a Polyphase Channelizer block to split a section of sampled spectrum into multiple channels.



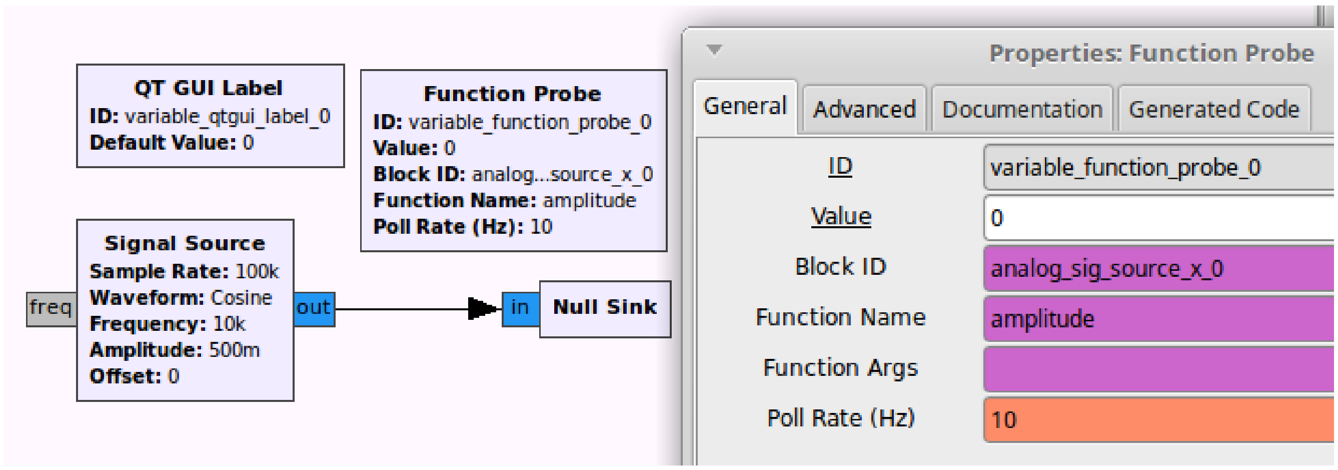
Set the Channels field to the number of channels you want to break the spectrum up into. The block will create that many evenly sized outputs. By default the lowest frequency channel will be output 0, the next will be output 1, and so on. You can change the Channel Map field to an array of integers to change this arrangement of channels to outputs. For example; setting the Channel Map field to [2,1,0] would reverse the order. There is also a Taps field where you have the option of setting up a filter which will be applied to each channel independently. This is a convenient place to apply something like a low-pass filter because you can avoid putting separate filter blocks on each stream.

Because the channels are each the same size, you can take this into account when visualizing or otherwise manipulating the channels. In the above example, the number of channels is set to be 3 by a variable called "channels" and the original size of the spectrum being channelized is set to 100k by the "samp_rate" variable. If you were using QT GUI Sinks or other similar sink blocks, you would set the Bandwidth field in each GUI sink to samp_rate/channels. The Center Frequency field of the lowest channel would be set to 0. The next block up would have it's Center Frequency field set to samp_rate/channels. The next one up would have it's Center Frequency field set to (samp_rate/channels)*2 and so on for as many channels as you have. This will display each channel properly centered with the correct frequency markings.

Problem 8: Call functions in a block

Solution: Use the Function Probe block to call functions in other blocks in order to extract data from them or set values in them.

Set the Block ID field to the ID of the block you want to work with. Set the Function Name field to the function you want to use in the block and the Function Args field to the argument or arguments you want to pass to the function (if applicable). Once done, the Function Probe ID can now be used as a variable name just like a variable block. In the below example the Function Probe block calls the amplitude() function from the Signal Source block. The QT GUI Label is then set by using the Function Probe blocks ID in its Default Value field.


QT GUI Label
ID: variable_qtgui_label_0
Default Value: 0

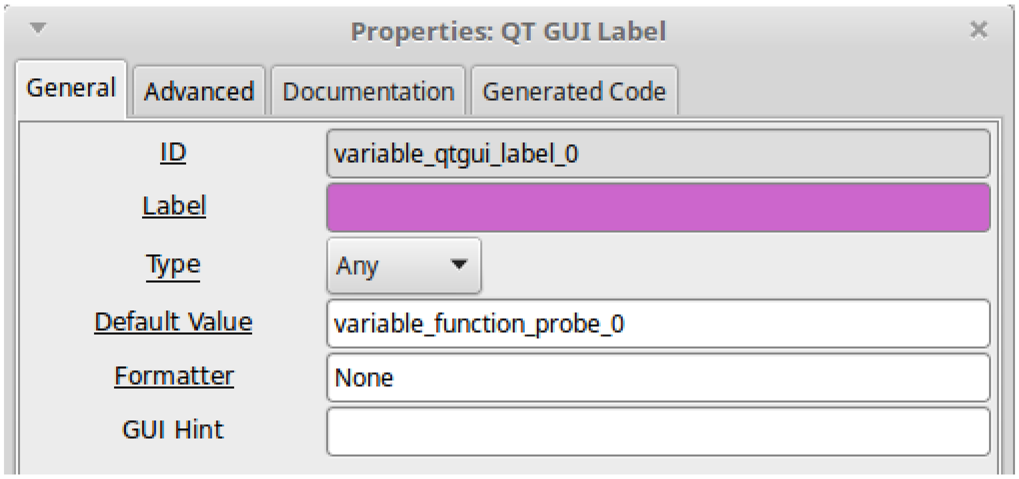
Function Probe
ID: variable_function_probe_0
Value: 0
Block ID: analog...source_x_0
Function Name: amplitude
Poll Rate (Hz): 10

Signal Source
Sample Rate: 100k
Waveform: Cosine
Frequency: 10k
Amplitude: 500m
Offset: 0

Null Sink

Properties: Function Probe

ID	variable_function_probe_0
Value	0
Block ID	analog_sig_source_x_0
Function Name	amplitude
Function Args	
Poll Rate (Hz)	10

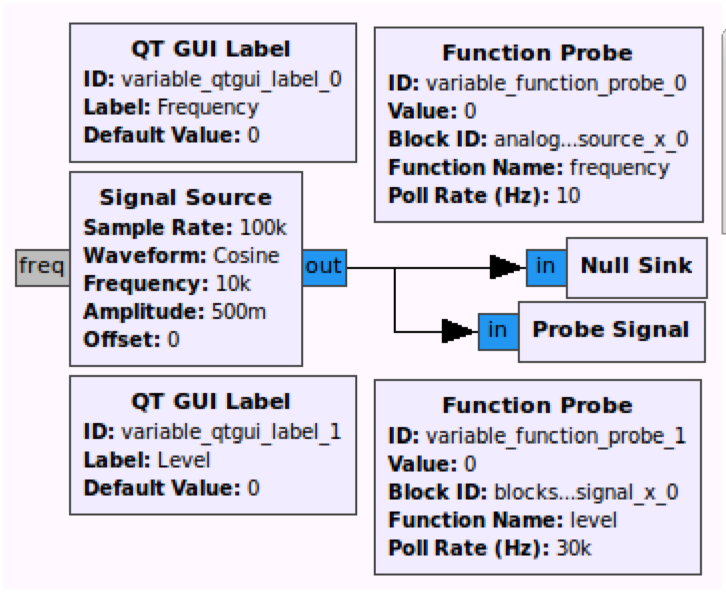

Properties: QT GUI Label

ID	variable_qtgui_label_0
Label	
Type	Any
Default Value	variable_function_probe_0
Formatter	None
GUI Hint	

The functions available in each block mostly match the parameter fields in their properties boxes but the exact wording doesn't always match. In order to see the exact names of the functions that are available and what arguments they take, it is necessary to go to: https://www.gnuradio.org/doc/doxygen/group__block.html.

Pick the block you want to learn about and scroll down the the section titled “Public Member Functions”. This gives you the function names, any arguments they take, and what data type they return. However, you will often times also see more functions named than are present in the Properties box. These additional functions usually don’t work, but it may be worth your while to try them.

Finally, there is a special set of blocks designed for use with the Function Probe block. They are in the Measurement Tools section of the block tree panel and all of them have names that start with “Probe”. The most commonly useful one is the Probe Signal block. This block has a function called “level” that returns the level of a signal when called. If it is a complex signal, then the function returns the real and imaginary parts as a Python list containing two items with the imaginary part appended with “j”. As you can see below, probing the frequency of the Signal Source block just gives the frequency value it is generating but the Probe Signal block’s level function returns the value of the stream at that moment.



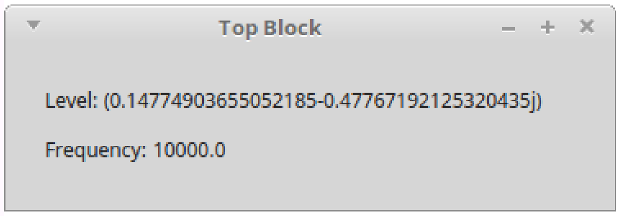
QT GUI Label
ID: variable_qtgui_label_0
Label: Frequency
Default Value: 0

Function Probe
ID: variable_function_probe_0
Value: 0
Block ID: analog...source_x_0
Function Name: frequency
Poll Rate (Hz): 10

Signal Source
Sample Rate: 100k
Waveform: Cosine
Frequency: 10k
Amplitude: 500m
Offset: 0

QT GUI Label
ID: variable_qtgui_label_1
Label: Level
Default Value: 0

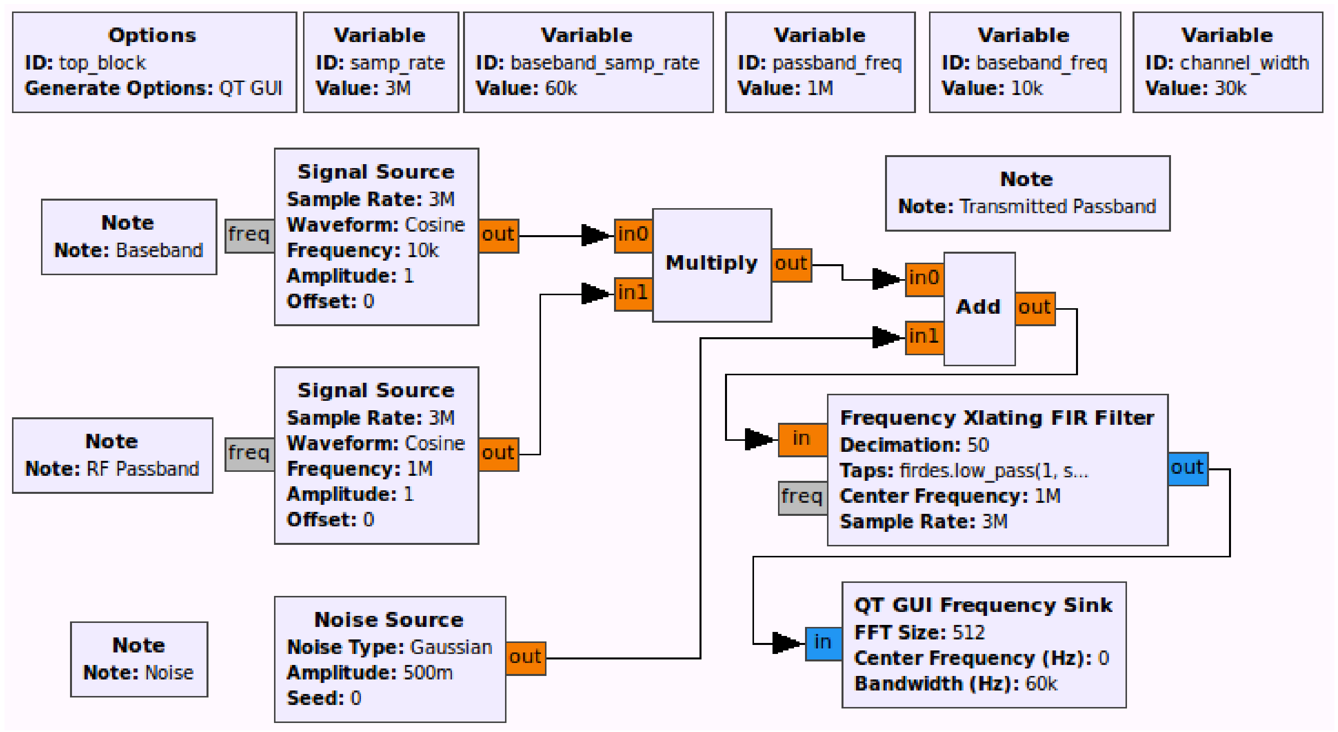
Function Probe
ID: variable_function_probe_1
Value: 0
Block ID: blocks...signal_x_0
Function Name: level
Poll Rate (Hz): 30k



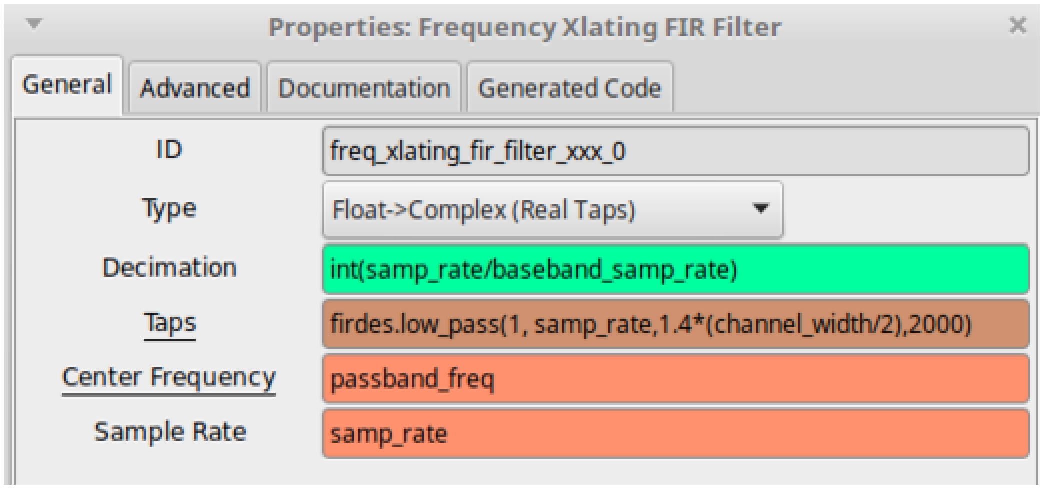
```
Level: (0.14774903655052185-0.47767192125320435j)

Frequency: 10000.0
```

Problem 9: Retrieve the baseband signal from an RF signal

Solution: Use a Frequency Xlating FIR Filter block to 1) Shift the frequency to bring the desired signal down to 0 Hz (or any other value you're targeting), 2) apply a filter (usually a low-pass filter) to the down-shifted spectrum to remove everything you aren't interested in, and 3) reduce the sample rate if desirable (likely). The block performs these steps in the order given above.



Above, you can see a baseband signal in the top left, the RF passband below it and a noise source added to the final result to emulate a channel. The transmitted passband is then received by the Frequency Xlating FIR Filter which applies the frequency shift, filter and resampling.



ID	freq_xlating_fir_filter_xxx_0
Type	Float->Complex (Real Taps)
Decimation	int(samp_rate/baseband_samp_rate)
Taps	firdes.low_pass(1, samp_rate,1.4*(channel_width/2),2000)
Center Frequency	passband_freq
Sample Rate	samp_rate

To shift the frequency down, first put into the Decimation field: int(samp_rate/baseband_samp_rate), where samp_rate is the sample rate used by the RF hardware and baseband_samp_rate is the sample rate used prior to up-converting the original data. Then put into the Center Frequency field: passband_freq.

To apply the filter, call the firdes function to calculate the filter taps that the Taps field requires. Set the Type field to a type of taps that matches the firdes function you are intending to call. For example; if you want to use firdes.complex_band_pass() then the Type field should have a selection with complex taps. If you want to use firdes.low_pass() then the Type field should have a selection with real taps.

In the below example, the firdes.low_pass function is called and is sent parameters of:

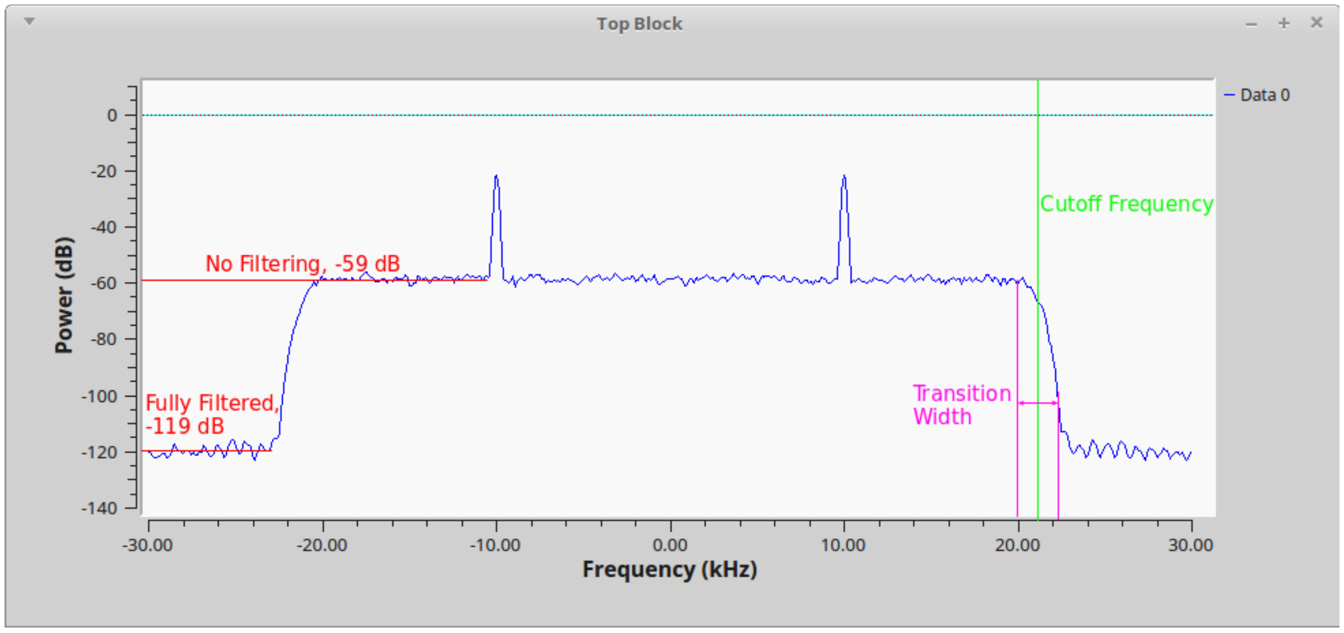
1: This is the gain applied.

samp_rate: This is the RF sample rate, not the baseband sample rate.

1.4*(channel_width/2): This sets the cutoff frequency measured in Hz away from 0. Cutoff is the mid-point of the transition area (the size of the transition area is the next parameter). Using this formula sets the cutoff to be slightly outside the bounds of your channel width for a little margin of error. You can increase the margin of error or decrease it by increasing or decreasing the 1.4 value.

2000: This sets the transition width. Transition is the measurement of how much space in Hz there is between where attenuation begins (no filtering) and where it is fully attenuated. The smaller the size of the transition, the more perfectly straight the cutoff it, but the harder the processor has to work.

Finally, there is the window type. The firdes.low_pass function defaults to a Hamming window which happens to applys 60 dB of attenuation. That means the level of the signal outside the channel will be approximately 60 dB lower than the signal inside the channel.



Problem 10: Convert a stream of bytes into a stream of bytes with values 0 or 1

Solution: Use a Packed to Unpacked block to convert all eight bits of each incoming byte into a separate byte with a value of either 0 or 1.

Set the Bits per Chunk field to 1. This makes each bit into a separate byte. Set the Endianness field to MSB (Most Significant Bit) to output the bits from each incoming byte leftmost first. For example, byte [1,0,1,1,0,0,1,0] would turn into 8 bytes in the following sequence:

[0,0,0,0,0,0,0,1]

[0,0,0,0,0,0,0,0]

[0,0,0,0,0,0,0,1]

[0,0,0,0,0,0,0,1]

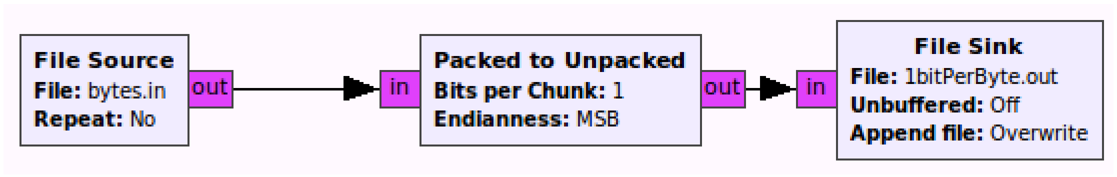
[0,0,0,0,0,0,0,0]

[0,0,0,0,0,0,0,0]

[0,0,0,0,0,0,0,1]

[0,0,0,0,0,0,0,0]

Setting the Endianness field to LSB (Least Significant Bit) would reverse the sequence of the output bytes.

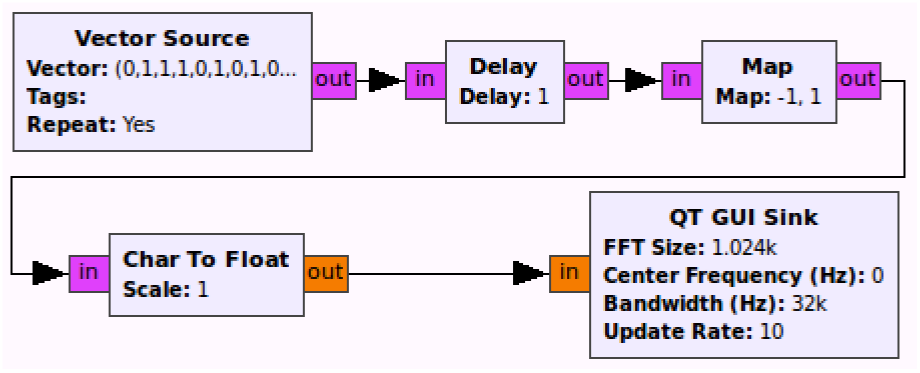

File Source
File: bytes.in
Repeat: No

Packed to Unpacked
Bits per Chunk: 1
Endianness: MSB

File Sink
File: 1bitPerByte.out
Unbuffered: Off
Append file: Overwrite

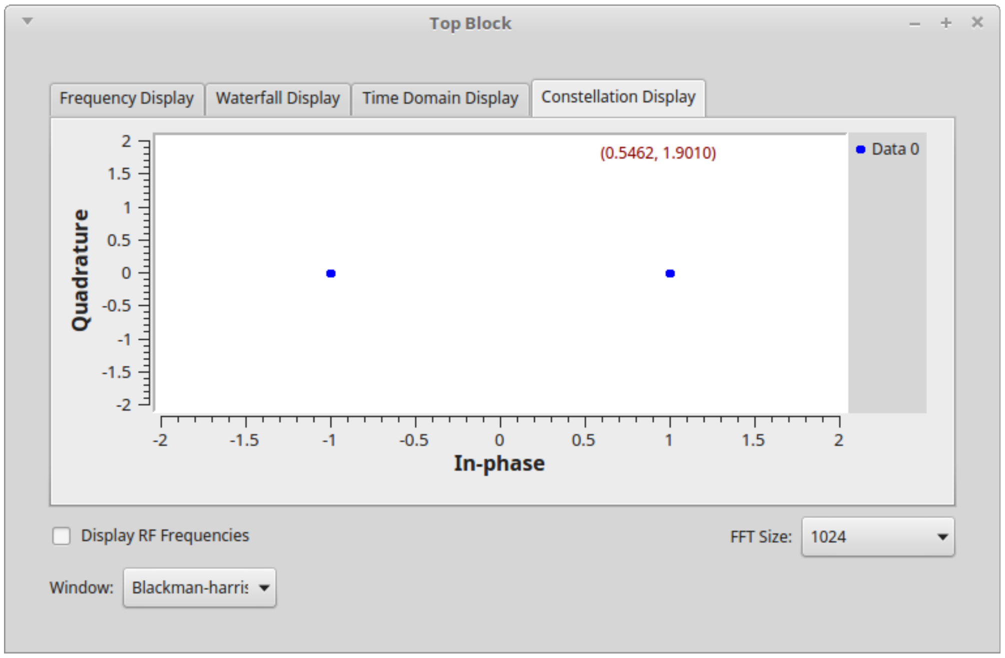
Problem 11: Prepare a stream of bytes for the Frequency Mod block

Solution: Use a Delay block to insert a single 0 into the stream before anything else passes through (more on this later). Then use a Map block to convert the 0's into -1's while leaving the 1's as 1's. This assumes that each byte has a value of either 0 or 1. Finally use a Char to Float block to convert these integer -1's and 1's into floating point -1's and 1's. This takes arbitrary digital data and prepares it for the Frequency Mod block which expects real numbers centered around 0.



Set the Delay blocks Delay field to 1. This delays the incoming stream by one sample. The one sample delay is replaced by a single 0 value. This ensures that the next block, the Map block receives a 0 first, no matter what the original data was. Set the Map blocks Map field to [-1,1]. This will cause the Map block to map the first byte it receives into a -1 and the next into a 1. It will continue to map all subsequent bytes of the value into the same results.

The constellation output shows that the real values are now centered around 0 as the Frequency Mod block requires.


Top Block
Frequency Display
Waterfall Display
Time Domain Display
Constellation Display
(0.5462, 1.9010)
Data 0
Quadrature
In-phase
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
Display RF Frequencies
FFT Size:
1024
Window:
Blackman-harris