Plug-ins for GNU Radio Companion

Shravan Sriram,
Gunturi Srivatsa
Student, Department of ECE
Amrita Vishwa Vidyapeetham,
Coimbatore.

Gandhiraj R
Assistant Professor,
Communication Engineering
Research Group,
ECE Department, Amrita
Vishwa Vidyapeetham

Soman K P
Head, Computational
Engineering and Networking,
Amrita Vishwa Vidyapeetham

ABSTRACT
This paper gives an insight on how to develop plug-ins (signal processing blocks) for GNU Radio Companion. GRC is on the monitoring computer and does bulk of the signal processing before transmission and after reception. The coding done in order to develop any block is discussed. A block that performs Huffman coding has been built. Huffman coding is a coding technique that gives a prefix code. A block that performs convolution coding at any desired rate using any generator polynomial has also been built. Both Huffman and Convolution coding are done on data stored in file sources by these blocks. This paper thus describes the ease of signal processing that can be attained by developing blocks in demand by changing the C++ and PYTHON codes of the HOWTO package. Being an open source it is available to all, is highly cost effective and is a field with great potential.

Keywords
Flowgraphs, FPGA, USRP, Blocks, plug-ins, GRC, convolution coder, Huffman coder, C++, Python, Software Defined Radio (SDR)

1. INTRODUCTION
The need to update radio transceivers through software updates lead to a concept known as Software Defined Radio. When the modification of the software is propagated to processes before transmission and after reception it leads to the development of new Blocks used in GRC. Signal processing blocks are of great help in SDR when the signal received is raw and processing is to be done in order to obtain information from a signal or to pack information into a signal. Hence blocks in GRC play a strong role in the software domain before transmission and after reception. These blocks perform a wide variety of tasks such as transforms, compression, coding, decoding, error correction, filter, modulation etc…More blocks can be incorporated to improve the signal processing ability in the software domain. Being an open source it is available to all, is highly cost effective and is a field with great potential.

2. SOFTWARE DEFINED RADIO (SDR)
Joe Mitola named a class of radios that is a single hardware that can perform different functions at different times as Software Defined Radio [2]. This radio is said to be able to handle a wide spectrum, traffic, air interfaces and applications. Thus SDR is a radio platform that performs signal processing on the digitized version of a complex input (input that is varied in traffic, frequency, application and experiences different air interfaces).

The USRP (Universal Software Radio Peripheral) turns general purpose computers into flexible SDR platforms. The core of any USRP will contain a motherboard with four high-speed ADCs and DACs and an FPGA [2]. The main principle behind the USRP is that the digital radio tasks are divided between the internal FPGA and the external host CPU. The high speed general purpose processing, like down and up conversion, decimation, and interpolation are performed in the FPGA, while waveform-specific processing, such as modulation, demodulation, coding and decoding are performed at the host CPU.

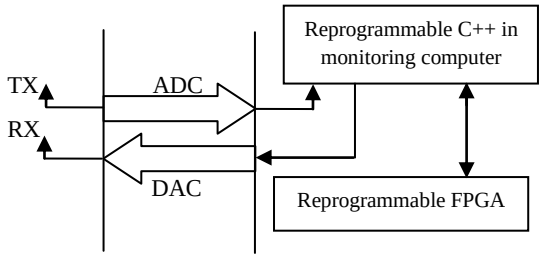

Figure 1: block diagram of a software defined radio.

3. GNU RADIO
3.1 What is GNU Radio?
GNU Radio project was started by Eric Blossom. It is free software that can be used in Linux based OS (Ubuntu) for simulating communication systems and for designing Software Defined Radios. It is hardware independent.

3.2 Organization
GNU radio is associated with two languages, C++ for creating signal processing block, python for connections and generating a signal flow graph. A signal processing block is typically a C++ function which will perform any processing on the input signal. A Signal Flow Graph portrays how various signal processing blocks are inter-connected. The C++ codes of the blocks in GNU radio are accessed by python codes by importing them using SWIG (Simplified Wrapper and Interface Generator) [3]. In the flow graph vertices represent signal processing blocks and edges represent the data flow between them. Flow graphs in CPU can be used along with an USRP to perform radio applications.

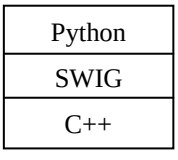

Figure 2: layers of a GNU radio block.

3.3 GNU Radio Companion (GRC)

GNU Radio Companion is a Graphical User Interface (GUI) tool for creating Signal Flow Graph(s). It is currently under development by Josh Blum.

Users can drag and drop GNU radio blocks, give inter-connections, can edit various block parameters and thus create a Signal Flow Graph (see Figure 4). The GNU radio companion is also a useful tool for simulation as GRC contains various WX widgets (block) which can be connected to verify the output. Knowledge of Python is not required for using GRC. GRC executes the flow graph by generating a Python code.

4. Creation of a Block in GRC

The GNU Radio companion contains signal processing blocks that implement a C++ code to perform signal processing. A number of signal processing blocks are given by the GNU radio software as a package. Since the software is open source, new user defined blocks can be created. This is an advantage as MATLAB Simulink type of blocks can be created and used for various applications.

The *how to create* package (a tar file) is available for download in the GNU radio website, the tar file is extracted to get the C++, Python, SWIG and XML files that define the pair of sample blocks, *square and square2* (plug-ins for squaring a number).

To create the *square* block in GRC, the following commands are to be given in the terminal of a Linux based OS (Ubuntu), [4] after entering the directory that has the extracted files.

```
>>./bootstrap
>>./configure
>>cd swig
>>make generate makefile swig
>>cd
```

Re-enter the directory containing the extracted files.

```
>>sudo make install
>> sudo ldconfig
```

After giving the above commands, a new tab called HOWTO is created in GRC that contains the square and square2 blocks that performs the squaring of a number.

To obtain a signal processing block one can edit the C++ and XML codes available for square block. This requires understanding of C++ code for the square block. XML should be edited to alter the way block should appear in the GUI.

Any signal processing block will have 3 files associated with it

- The .h and .cc files that defines the new block class.
- The .i file describes how the block is imported to python using SWIG.

The C++ concepts used in the code of the square block are explained in detail.

Howto_square_ff.cc and *howto_square_ff.h* are the C++ file and header files respectively for the square block, these files are present in the extracted content of the HOWTO package.

In *howto_square_ff.h*, gr_block.h header is included. howto_square_ff.h header is included in *howto_square_ff.cc*, gr_io_signature.h is included in *howto_square_ff.cc*.

gr_block.h is one among the extracted files. The base class for all signal processing blocks is gr_block. A new block class when created in GRC derives its class from the gr_block class or one of its sub-classes. Code snippet for gr_block class

```
private:
std::string d_name;
gr_io_signature_sptr      d_input_signature;
gr_io_signature_sptr      d_output_signature;
int                       d_output_multiple;
double                    d_relative_rate;
gr_block_detail_sptr      d_detail;
long                      d_unique_id;
```

d_name has the blocks name, d_unique_id has the ID of the block [3]. Another file that is present is *gr_io_signature.h*. This header file contains the class gr_io_signature, describes how input or output flows. It also gives information on the size of the input and output streams.

```
class gr_io_signature {
public:
~gr_io_signature ();
Int min_streams ()    const  {return d_min_streams; }
int max_streams()     const  {return d_max_streams; }
size_t    sizeof_stream_item (int index) const { return
          d_sizeof_stream_item; }
private:
int       d_min_streams;
int       d_max_streams;
size_t    d_sizeof_stream_item;
gr_io_signature (int min_streams, int max_streams,
size_t sizeof_stream_item);
friend    gr_io_signature_sptr gr_make_io_signature
          ( int          min_streams,
            int          max_streams,
            size_t       sizeof_stream_item);
};
```

The class gr_io_signature defines the upper and lower bounds of the input and output streams as d_min_streams and d_max_streams respectively. And the size of an item in the stream is given by sizeof_stream_item. When a block is created, there is need to indicate two signatures for both input and output flows. d_input_signature and d_output_signature are two smart pointers referring to gr_io_signature objects for the input and output flows, which is the basic information of a block.

The method general_work is the brain of the box that does all the signal processing.

```
virtual int general_work
( int noutput_items,
gr_vector_int &ninput_items,
gr_vector_const_void_star &input_items,
gr_vector_void_star &output_items)
```

noutput_items and ninput_items are the number of output items to be written on each output stream and the number of input items available on each input stream respectively. We may have a vector of integers, each element of which gives the input on the input stream. But noutput_items is just an integer. Thus input rate can be different but the output rate is constant.

```
typedef std::vector<void *>
gr_vector_void_star;
typedef std::vector<const void*>
gr_vector_const_void_star;
```

This the way the input and out data types need to be initialised. To know more about the special data types in GNU see gr_types.h [3].

GNU Radio makes use of a special class of C++ pointers called Boost smart pointers [3]. Boost is a collection of C++ libraries that provides powerful extensions to C++ from many aspects, such as algorithm implementation, math/numeric's, input/output, iterations, etc. Their behavior is very much similar to that of C++ pointers except that they ensure proper destruction of dynamically allocated objects. This is a pre-required package d the installation of GNU radio.

After adding the boost header file, the boost shared pointer can be used by defining it as

```
Boost::shared_ptr  <T> pointer_name
```

Where, T is the type of object pointed to by the smart pointer. Thus,

```
boost::shared_ptr<gr_io_signature>
```

declares a smart pointer pointing to an object with class type gr_io_signature. Since the constructors used in the .h files are private they cannot be called outside the class. And in order to avoid the private constructor being pointed by a raw C++ pointer the boost shared pointers are made use of. First a friend function is declared so that we can access all the private data. Then in the function, the private constructor is called to create an instance.

```
howto_square_ff::general_work (int noutput_items,
gr_vector_int &ninput_items,
gr_vector_const_void_star &input_items
gr_vector_void_star &output_items)
```

The above snippet is the definition of the function that performs bulk of the signal processing. The C++ code defining any system is defined here, that is, the main C++ program of the block is written under this function.

If the block contains multiple inputs, Specific constraints on the number of input and number of output streams must be specified.

This information is used to construct the input and output signatures (2nd & 3rd arguments to gr_block's constructor). The input and output arguments are used by the system to check that a valid number and type of inputs and outputs are connected to this block during execution in GRC. In the case of a simple square block, one input is accepted and one output is given out [8]. A new input vector must be instantiated corresponding to a second input when needed. In the convolution encoder it is done in the following way

```
const float *in0 = (const float *) input_items[0];
const float *in1 = (const float *) input_items[1];
```

In order to obtain certain parameters from the user one has to define the required parameters in the function definition, and also pass these parameters to each of the function calls and every use of the shared pointer.
The XML file is responsible for the outlook of the block. Any changes made in the code that involves change in the number of inputs, outputs and also obtaining parameters from the user must reflect in the XML file.

For the case of 2 inputs, the following changes in *howto_square_ff.xml* file give the 2 ports required by the block

```
<sink>
  <name>N</name>
  <type>float</type>
 </sink>
<sink>
  <name>Ngb</name>
  <type>float</type>
 </sink>
```

And the following snippet gives the ports to provide the block with the required parameters Eg.vlen, N.

```
 <make>howto.square_ff($N)</make>
<callback>set_N($N)</callback>
        <param>
                <name>N</name>
                <key>N</key>
                <type>int</type>
        </param>
```

Any information about the block for the user to see can be given as documentation.

```
<doc>
details...about block
</doc>
```

There is a makefile in the extracted folder which has to be compiled so that all the changes made are checked for errors and compiled and the final block with changes can be seen in GRC.

For compiling, the following command is given in the terminal.

>>sudo make install

5. HUFFMAN CODING

Consider a discrete memory less source of characters {s0, s1,….., sk-1} with probabilities {p0, p1, …,pk-1}. For such a source , the code has to be uniquely decodable. This restriction ensures that for each finite sequence of symbols emitted by the source, the corresponding sequence of code words is different from the sequence of code words corresponding to any other source sequence. A prefix code is defined as a code in which no codeword is a prefix of any other codeword. Huffman coding is one such algorithm that produces prefix codes [5].

In Huffman coding the probabilities of the characters are first calculated. Then a node is created where the bottom and top elements are the lowest and second lowest probabilities in the sorted list. Elements of this node are added. Then the process of sorting, creation and adding of nodes is continued to end up with 1 remaining node. Thus a tree is formed and a 0 is placed on the node on the bottom and a 1 on the node on the top. From the tree the code for each character is obtained. To represent the message, the character is replaced by the corresponding codes [7].

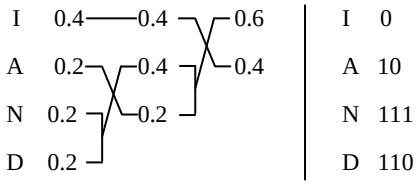

Figure 3: Huffman Coding performed for INDIA

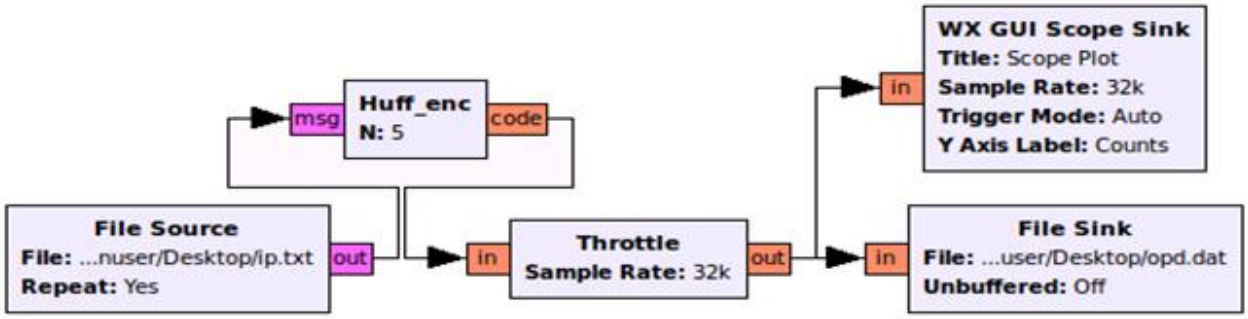

Figure 4: A GRC implementation of Huffman Coding.

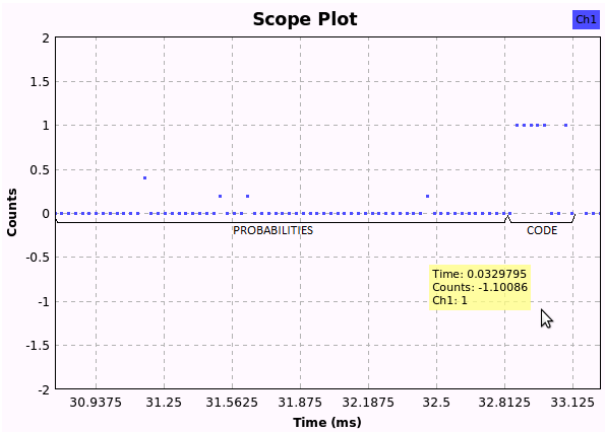

Figure 5: a dot diagram showing Huffman Coder output for INDIA as input.

6. CONVOLUTION CODING

To perform convolutional encoding of data, start with k memory registers, each holding 1 input bit. Unless otherwise specified, all memory registers start with a value of 0. The encoder has n modulo-2 adders (a modulo 2 adder can be implemented with a single Boolean XOR gate, where the logic is: 0+0 = 0, 0+1 = 1, 1+0 = 1, 1+1 = 0, and n generator polynomials — one for each adder (see figure below). An input bit m_1 is fed into the leftmost register. Using the generator polynomials and the existing values in the remaining registers, the encoder outputs n bits. Now bit shift all register values to the right (m1 moves to m2, m2 moves to m3) and wait for the next input bit. If there are no remaining input bits, the encoder continues to give on output until all registers have returned to the zero state [6].

The figure below is a rate 1/2 (m/n) encoder with constrain length (k) of 3. Generator polynomials are G1= (1,1,1) and G2 = (0,1,1) Therefore, output bits are calculated (modulo 2) as follows:

f1 = m1 + m2 + m3; f2 = m2 + m3

For data of a known length convolution encoding is performed by the multiplication of the message bit with each of the generator polynomials and alternating the bits from these products at the output so that we get k output bits for every input bit. For a message sequence of 10011 and Generator polynomials (1,0,1), (1,1,1).

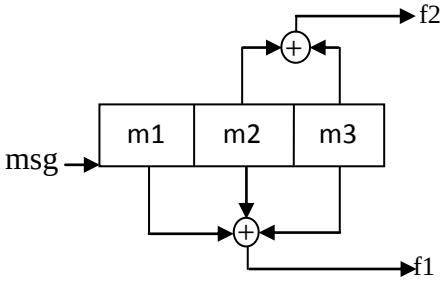

Figure 6: block diagram of a 1/2 convolution encoder.

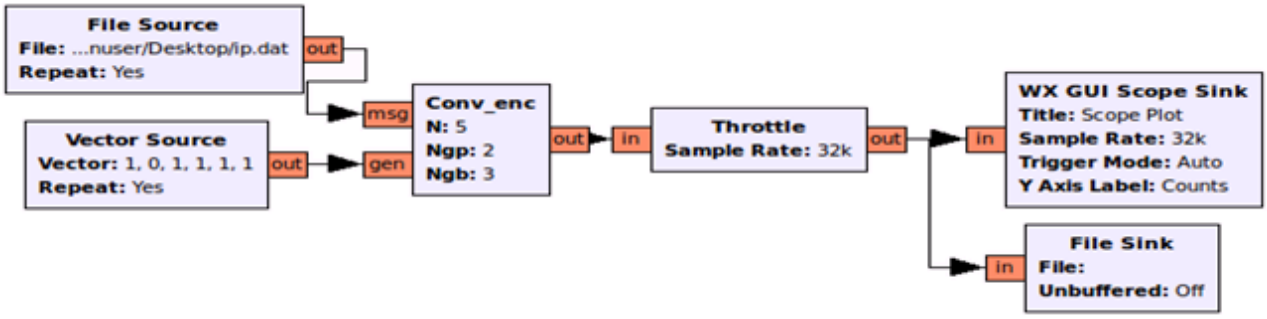

Figure 8: snapshot of convolution encoder in GRC.

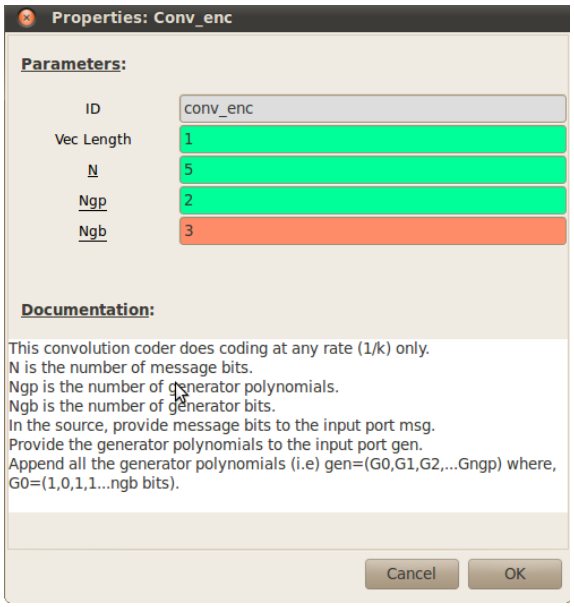

Figure 7: snapshot of the properties of conv_coder block where we can set different parameters.

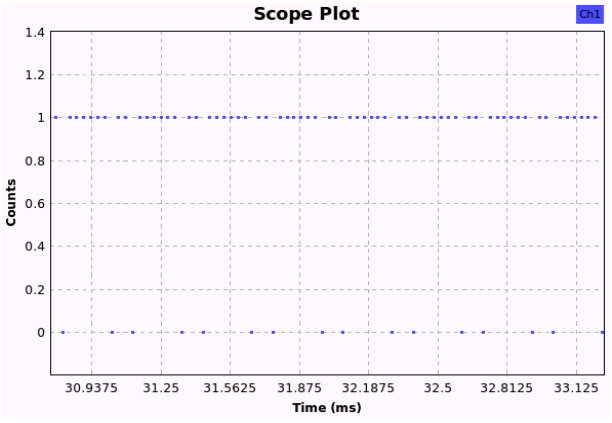

Figure 9: o/p of the conv_coder, i/p – 10011.

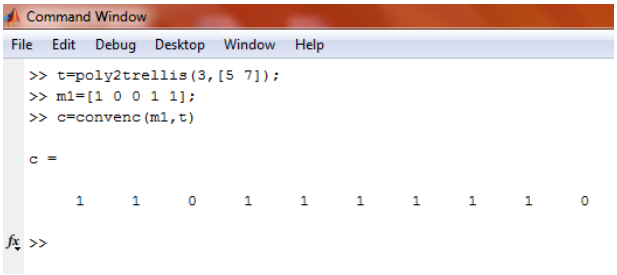

Figure 10: snapshot of MATLAB command window, where convolution encoder can be performed with single function convenc.

7. CONCLUSION

Block creation in GRC is possible if we have knowledge in C++ and XML. It is not difficult in GRC as a basic howto_square block is available at (www.gnuradio.org) and if we can write an algorithm for the operation required from the block, we can create MATLAB Simulink type of blocks by ourselves. These blocks can be used to process our input signals. Improving the scope of these blocks greatly reduces the burden on the hardware engineers, as almost all work done by the hardware can be accomplished in the software domain. Being an open source it is available to all, is highly cost effective and is a field with great potential.

8. ACKNOWLEDGMENTS

The success of any project depends largely on the encouragement and guidance of many others. We would like to take this opportunity to express our gratitude to Mr. Premanand, Research Associate, Department of Computation and Excellence in Networking for his valuable help.

9. REFERENCES

[1] Byron S Gottfried, 1996. Programming with C, The McGraw-Hill Company.

[2] Danilo Valerio, 2008 Open Source Software-Defined Radio: A Survey on GNU Radio and its Applications.

[3] Dawei Shen, May-June,2005, Tutorials on Software Defined Radio.

[4] GNU Radio website http://www.gnuradio.org.

[5] Ranjan Bose, 2008 Information Theory, Coding and Cryptography, 2nd Editon, Tata McGraw-Hill Publishing Company Limited, New Delhi.

[6] Simon Haykin, 1995 Communication Systems, 3rd Edition, John Wiley & sons (Asia) Pvt. Ltd. , New Delhi.

[7] Willi-Hans Steeb, 2005 Mathematical Tools in Signal Processing with C++ and Java Simulations, International School for Scientific Computing.

[8] Yashwanth P.Kanetkar, 1997. Understanding Pointers in C, 1st edition, BPB Publications.