GNU Radio Internals

Tanguy Risset

Citi Laboratory, INSA de Lyon

April 3, 2019

Table of contents

Source material and Warning

- These slides were built from many sources among which:
  - Gnuradio wiki tutorial (https://wiki.gnuradio.org/index.php/Tutorials)
  - Gnuradio API doc (https://gnuradio.org/doc/doxygen/), various GNU Radio version available
  - Tom Rondeau slides (http://www.trondeau.com/)
  - "Developing Signal Processing Blocks for Software Defined Radio", Gunjan Verma and Paul Yu, Army Research Laboratory, ARL-TR-5897, 2012.
- Gnuradio is evolving quickly, some of the details mentioned here can become optional or are not yet deployed if you use older version

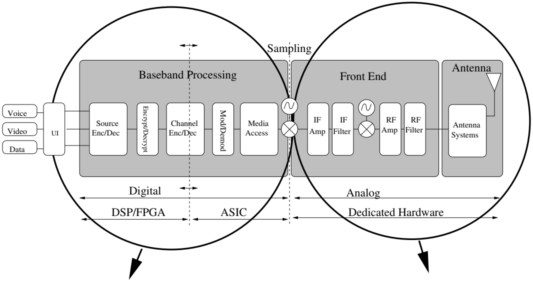
Table of Contents

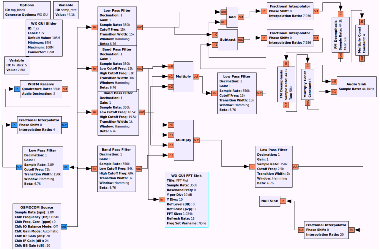
What is GNU Radio?

An open source framework for building software radio transceivers

- An open source software toolkit
  - Creating signal processing applications
  - Defining waveforms in software
  - Processing waveforms in software
- An important development community
  - Active developer community producing examples
  - GNU radio conference (2011-2014)
- A set of hardware platforms
  - USRP1 & USRP2, Universal Software Radio Peripheral,
  - RTL2832 TV tuners
- an *easy-to-use* approach (Simulink-like)

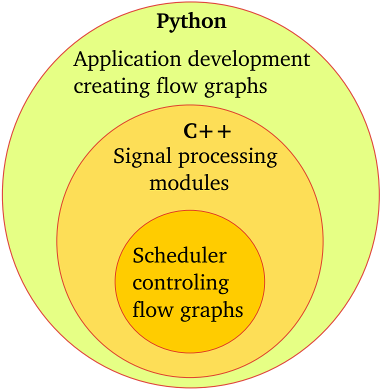
The big picture





A 3 tier architecture

- Python scripting language used for creating "‘signal flow graphs"’
- C++ used for creating signal processing blocks
  - An already existing library of signalling blocks
  - Tools for enabling the addition of new blocks
- The scheduler is using Python’s built-in module threading, to control the ‘starting’, ‘stopping’ or ‘waiting’ operations of the signal flow graph.

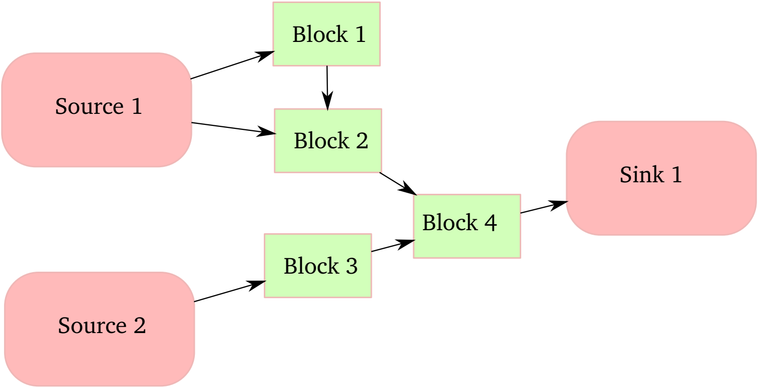


GNU Radio ‘Hello World’ application

```
#!/usr/bin/env python

from gnuradio import analog
from gnuradio import blocks
from gnuradio import audio
from gnuradio import gr

class top_block(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self, "Hello Word")

        samp_rate = 32000
        freq1=440
        ampl = 0.4

        self.audio_sink = audio.sink(32000, "", True)
        self.analog_sig_source_1 = analog.sig_source_f(samp_rate,
                    analog.GR_COS_WAVE, 350, ampl, 0)
        self.analog_sig_source_0 = analog.sig_source_f(samp_rate,
                    analog.GR_COS_WAVE, 440, ampl, 0)

        self.connect((self.analog_sig_source_0, 0), (self.audio_sink, 1))
        self.connect((self.analog_sig_source_1, 0), (self.audio_sink, 0))

if __name__ == '__main__':
    tb = top_block()
    tb.start()
    raw_input ('Press Enter to quit: ')
    tb.stop ()
```

Data-flow programming

- Sources, Sinks, Computational Blocks and Data Flows



GNU Radio Library

Fundamentals

- gr-analog
- gr-audio
- gr-blocks
- gr-channels
- gr-digital
- gr-fec
- gr-fft
- gr-filter
- gr-trellis
- gr-vocoder
- gr-wavelet

Graphical Interfaces

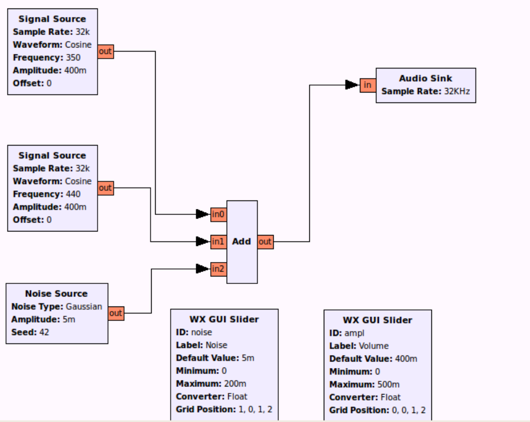
- gr-qtgui
- gr-wxgui

Hardware Interfaces

- gr-audio
- gr-comedi
- gr-shd
- gr-uhd

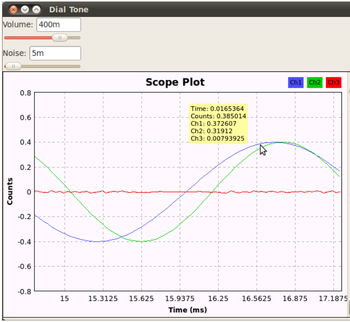
A simple example with GNU Radio companion (GRC)

- Dial tone `GNURADIO/gr-audio/example/grc/dial-tone.grc`

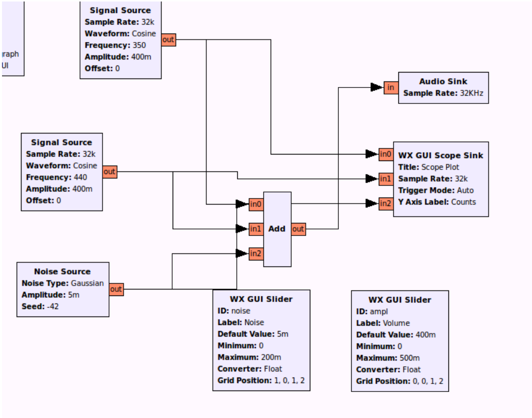


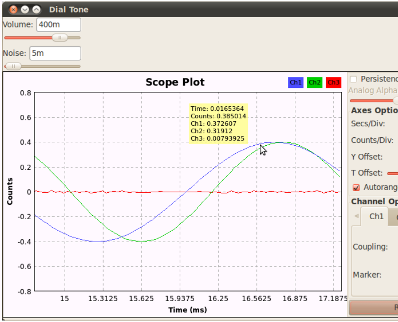
Run-Time Execution

- The `dial-tone.grc` in an XML interface instantiating python and C++ code.
- It can be:
  - Compiled (it generates a python file: `dial-tone.py`)
  - Executed (i.e. executes the generated python file)
  - Debugged (with spectrum analyzer for instance)



Debugging dial tone





Dial Tone: GRC XML code

```
<?xml version='1.0' encoding='ASCII'?>
<flow_graph>
  <timestamp>Tue May  6 17:48:23 2014</timestamp>
  <block>
    <key>options</key>
    <param>
      <key>id</key>
      <value>dial_tone</value>
    </param>
    <param>
      <key>_enabled</key>
      <value>True</value>
    </param>
    <param>
      <key>title</key>
      <value>Dial Tone</value>
    </param>
    <param>
      <key>author</key>
      <value>Example</value>
[...]
```

```
 <block>
   <key>analog_sig_source_x</key>
   <param>
     <key>id</key>
     <value>analog_sig_source_x_0</value>
   </param>
   <param>
     <key>_enabled</key>
     <value>True</value>
   </param>
   <param>
     <key>type</key>
     <value>float</value>
   </param>
   <param>
     <key>samp_rate</key>
     <value>samp_rate</value>
   </param>
   [...]
 </block>
[...]
<connection>
   <source_block_id>blocks_add_xx</source_block_i
   <sink_block_id>audio_sink</sink_block_id>
   <source_key>0</source_key>
   <sink_key>0</sink_key>
 </connection>
 <connection>
   <source_block_id>analog_sig_source_x_0</source
   <sink_block_id>blocks_add_xx</sink_block_id>
   <source_key>0</source_key>
```

Dial Tone: Python code (manual)

```
from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_option import eng_option
from optparse import OptionParser
from gnuradio import analog

class my_top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self)

        [....]
        sample_rate = int(options.sample_rate)
        ampl = 0.1

        src0 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 350, ampl)
        src1 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink(sample_rate, options.audio_output)
        self.connect(src0, (dst, 0))
        self.connect(src1, (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
    except KeyboardInterrupt:
        pass
```

Dial Tone: Python code (generated from .grc)

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Dial Tone
# Author: Example
# Description: example flow graph
# Generated: Tue May  6 17:48:25 2014
##################################################

from gnuradio import analog
from gnuradio import audio
from gnuradio import blocks
[...]
class dial_tone(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Dial Tone")
        _icon_path = "/usr/share/icons/hicolor/32x32/apps/gnuradio-grc.png"
        self.SetIcon(wx.Icon(_icon_path, wx.BITMAP_TYPE_ANY))

        self.samp_rate = samp_rate = 32000
        self.noise = noise = .005
        self.ampl = ampl = .4

        _noise_sizer = wx.BoxSizer(wx.VERTICAL)
        self._noise_text_box = forms.text_box(
         parent=self.GetWin(),
         sizer=_noise_sizer,
         value=self.noise,
         callback=self.set_noise,
         label="Noise",
```

Dial Tone: C++ code (manual)

```
/*
 * GNU Radio C++ example creating dial tone
 * ("the simplest thing that could possibly work")
 *
 * Send a tone each to the left and right channels of stereo audio
 * output and let the user's brain sum them.
 */

#include <gnuradio/top_block.h>
#include <gnuradio/analog/sig_source_f.h>
#include <gnuradio/audio/sink.h>

using namespace gr;

int main(int argc, char **argv)
{
  int rate = 48000; // Audio card sample rate
  float ampl = 0.1; // Don't exceed 0.5 or clipping will occur

  // Construct a top block that will contain flowgraph blocks.  Alternatively,
  // one may create a derived class from top_block and hold instantiated blocks
  // as member data for later manipulation.
  top_block_sptr tb = make_top_block("dial_tone");

  // Construct a real-valued signal source for each tone, at given sample rate
  analog::sig_source_f::sptr src0 = analog::sig_source_f::make(rate, analog::GR_SIN_WAVE, 350, ampl);
  analog::sig_source_f::sptr src1 = analog::sig_source_f::make(rate, analog::GR_SIN_WAVE, 440, ampl);

  // Construct an audio sink to accept audio tones
  audio::sink::sptr sink = audio::sink::mak
```

GNU Radio software layers

- GRC: Graphical design tool
  - `GNURADIO/gr-audio/example/grc/dial-tone.grc`
  - ...
- python : Mostly Composite Block application
  - `GNURADIO/gr-audio/examples/python/dial_tone.py`
  - `GNURADIO/gr-digital/python/digital/ofdm.py`
  - ...
- C++ : Mostly Low level functions
  - `GNURADIO/gr-audio/examples/c++/dial_tone.cc`
  - `GNURADIO/gr-digital/lib/ofdm_cyclic_prefixer_impl.c`
  - ...

Table of Contents

GNU radio C++ naming convention

- Words in identifiers are separated by underscores (e.g. `gr_vector_int`)
- All types begin by `gr` (e.g. `gr_float`)
- All class variable begin by `d_` (e.g. `d_min_stream`)
- Each C++ class is implemented in a separated file (e.g. class `gr_magic` implemented in file `gr_magic.cc` with header file `gr_magic.h`)
- All signal processing blocs contain their input and output types in their suffixes. e.g.:

dc_blocker_ff_impl.cc

```
[..]
dc_blocker_ff_impl::
  dc_blocker_ff_impl(int D, bool long_form)
  : sync_block("dc_blocker_ff",
      io_signature::make (1, 1, sizeof(float)),
      io_signature::make (1, 1, sizeof(float))),
  d_length(D), d_long_form(long_form)
```

dc_blocker_cc_impl.cc

```
[..]
dc_blocker_cc_impl::dc_blocker_cc_impl(int D, bool long
   : sync_block("dc_blocker_cc",
       io_signature::make (1, 1, sizeof(gr_complex)),
       io_signature::make (1, 1, sizeof(gr_complex))),
    d_length(D), d_long_form(long_form)
```

Block signature

- A bloc signature is a specification of the data types that enter or exit the bloc.
- There are always two bloc signatures, one for inputs, the other for outputs.
- Each bloc signature specifies the number and types of ports.
- excerpt from `gr_io_signature.h`:

```
class GR_RUNTIME_API io_signature
{
  int                    d_min_streams;
  int                    d_max_streams;
  std::vector<int>       d_sizeof_stream_item;

  io_signature(int min_streams, int max_streams,
               const std::vector<int> &sizeof_stream_items);

public:
  typedef boost::shared_ptr<io_signature> sptr;

  ~io_signature();

  static sptr make(int min_streams, int max_streams,
                   int sizeof_stream_item);
  /*!
   * \brief Create an i/o signature
   *
```

Boost Pointer

- Gnu radio uses `Boost` smart pointers.
- `Boost` is a software library that provides a *smart* implementation of C++ pointers that offers garbage collection (i.e. delete object not used anymore).
- Gnu radio uses only the `shared_ptr` type of `Boost`
- Instead of declaring a pointer to a type `X`:

  `X* myPointer;`

  you can declare:

  `boost::shared_ptr<X> myBoostPointer`
- example in `gr_io_signature`

```
typedef boost::shared_ptr<io_signature> sptr;
static sptr make(int min_streams, int max_streams,
                 int sizeof_stream_item);
```

Volk library

- Gnu radio uses VOLK (which stands for Vector-Optimized Library of Kernels)
- `volk` provides a number of optimized function for vector processing using SIMD instructions.
- Developing with `volk` might be tricky because it is sensible to alignment of vector in memory.
- Understanding code using `volk` simply requires to understand `volk` naming convention:
  - The basic naming scheme will look something like this:
    `volk_(inputs params)_[name]_(output params)_[alignment]`
  - example:

```
volk_32f_invsqrt_32f
```

Other Volk example

- General naming convention when there are several inputs or outputs:

```
volk_(input_type_0)_x(input_num_0)_(input_type_1)_x(input_num_1)_...
    _[name]_(output_type_0)_x(output_num_0)_(output_type_1)_x(output_num_1)_..._[alignment]
```

- Examples:
  - Multiply two complex float vectors together (aligned and unaligned versions) and the dispatcher:

    ```
    volk_32fc_x2_multiply_32fc_a
    volk_32fc_x2_multiply_32fc_u
    volk_32fc_x2_multiply_32fc
    ```

  - Add four unsigned short vectors together:

    ```
    volk_16u_x4_add_16u
    ```

  - Multiply a complex float vector by a short integer:

    ```
    volk_32fc_s16i_multiply_32fc
    ```

SWIG

- SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages.
- SWIG is used in GNU Radio to link Python and C++ code

SWIG example (1)

- write a C file `example.c` code that defines the `int fact(int n)` function.
- write an *interface* file for SWIG:

```
%module example
%{
  int fact(int n);
%}
int fact(int n);
```

- execute the `swig` command:
  `swig -python example.i`

⇒ it generates a file `example_wrap.c`

- Compile the .c files with -fPIC option (and path to Python.h):
  `gcc -fPIC -I/usr/include/python2.7 -c example.c example_wrap.c`

SWIG example (1)

- link it to **_example.so**:
  `ld -shared example.o example_wrap.o -o _example.so`
- import **example** in python and use the `func` function:
  python [...] `>>> import example`
  ```
  >>> example.fact(5)
  120
  ```

Creating GNU radio modules

- A gnu radio module `newModule` corresponds to a directory `newModule` should contain the following directories:
  `CMakeLists.txt docs grc include lib python swig`
- the `gr_modtool` tool helps you create the various directory
- Hence the flow for creating a block in a module
  - Create the module file hierarchy with `gr_modtool`
  - Create a block in the module with `gr_modtool`
  - Edit the C++ file to code the module functionalities
  - Test, debug and validate the functionalities

Creating a trivial module: module creation

- creating the module directory structure:
  `gr_modtool newmod arith`
- Go into the new directory:

```
$ cd gr-arith/
$ ls
apps   CMakeLists.txt  examples  include  MANIFEST.md  swig
cmake  docs            grc       lib      python
```

Creating a trivial module: adding a bloc `times2`

- We will create a bloc of type general block without parameter.
- Go into the module directory (`$ cd gr-arith/`) and type the following command:
  `gr_modtool add -t general times2`
  Answer `cpp` to the first question and **nothing** to the other questions.
- create a python test method: edit `python/qa_times2.py`
- update `python/CMakeLists.txt` (nothing to do here)
- create the build directory `cd ..;mkdir build;`
- build the project: `cd build; cmake ../`

Creating a trivial module: directory hierarchy

```
gr-arith
   [...]
|-- CMakeLists.txt
|-- docs
|    |-- CMakeLists.txt
|    |-- doxygen
|    |    |-- CMakeLists.txt
     [....]
|-- grc
|    |-- arith_times2.xml
|    `-- CMakeLists.txt
|-- include
|    `-- arith
|         |-- api.h
|         |-- CMakeLists.txt
|         `-- times2.h
|-- lib
|    |-- CMakeLists.txt
|    |-- qa_arith.cc
|    |-- qa_arith.h
|    |-- qa_times2.cc
|    |-- qa_times2.h
|    |-- test_arith.cc
|    |-- times2_impl.cc
|    |-- times2_impl.cc~
|    `-- times2_impl.h
|-- python
|    |-- CMakeLists.txt
|    |-- __init__.py
|    |-- qa_times2.py
```

Creating a trivial module: The C++ part

- Header files are in `include/arith` directory
- Code is in `lib/times2_impl.cc`, edit it and replace the $<++>$ by values.
  - in `times2_impl()` (constructor
  - in `forecast` (indicate scheduler how many input are requires for how many output items)
  - in `general_work` core of the treatment.
- `make` it (in the `build` directory), and `make test`

Creating a trivial module: initial debugging

- use `printf` (`#include <sdtio.h>`)
- use `make; make test` (from python testbench)
- log output in `Testing/Temporary/LastTest.log`
- Or equivalently: `ctest -V`

Creating a trivial module: gnuradio-companion block

- Now that you have written a valid block, you can create a valid `grc` block
- go up to `gr-arith` directory:
  `gr_modtool makexml times2`
- install it:
  `cd build; sudo make install`
- You can also install the new blocks in a local directory and set environnement variables to indicate where gnuradio-companion should find them (see labs)
- create a simple grc application (use throttle, remove printf)
- run it (warning: no print!)

Table of Contents

Block important function

- Each Gnu radio bloc inherits from the `gr_block` class.
- The `gr_block` class contains the following important function (file `$GNURADIO/include/gnuradio`:

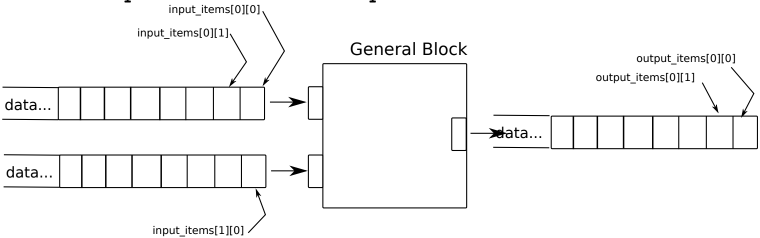
```
void  set_history(unsigned history);
virtual void forecast(int noutput_tems,
            gr_vector_int &ninput_items_required);
 virtual int general_work(int noutput_items,
            gr_vector_int &ninput_items,
            gr_vector_const_void_star &input_items,
            gr_vector_void_star &output_items);
 void consume(int which_input, int how_many_items);
```

function general_work

- The `general_work()` function computes output streams from input streams
- It has 4 arguments
  - `int noutput_items` Number of output items to write on each output stream (all output streams **must** produce the same number of output).
  - `int ninput_items[]` Number of input items to read in each input stream
  - `void* intput_items[]` Vectors of pointers to elements of the input stream(s), i.e., element *i* of this vector points to the i^{th} input stream.
  - `void* output_items[]` Vectors of pointers to elements of the output stream(s), i.e., element *i* of this vector points to the i^{th} output stream.

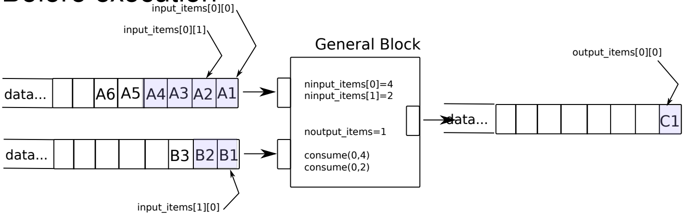
function general_work

- The `general_work` function implement the signal processing algorithm.
- It is called by the scheduler (implicitly, i.e. you do not have to invoke this function explicitly)
- The `consume` function indicates to the scheduler how many data have been consumed once the `general_work` has been executed
- Use of `input_items` and `output_items` vectors:

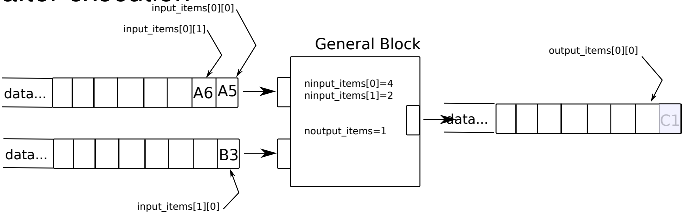


One execution of the block

- Before execution



- after execution



What the code of `work` function could be

- Example: sum the 6 samples in input on the output

```
for(unsigned int j = 0; j < 4; j++) {
 output_items[0][0] += input_items[0][j];
}
for(unsigned int j = 0; j < 2; j++) {
 output_items[0][0] += input_items[1][j];
}
```

- But it is not that simple...
- Gnu radio scheduler invokes the work function for computing a chunks of output (i.e. not one output by one output, in order to avoid too many context switches)
- `noutput_item` stays symbolic, it will be set dynamically during the execution by the scheduler for performance optimization (usually between 4000 and 10000).

What the code of `work` function should be

- add one loop over all `noutput_items` output samples:

```
for (i = 0; i < noutput_items; i++) {
  for(j=0 ; j < 4; j++) {
    output_items[0][i] += input_items[0][4*i+j];
  }
  for(unsigned int j = 0; j < 2; j++) {
    output_items[0][i] += input_items[1][2*i+j];
  }
}
```

- Remember to avoid as much as possible samples copy.

What the code of `work` function really is

- Usual Gnu radio way of writing:

```
const gr_complex *in1 = (const gr_complex*)input_items[0];
const gr_complex *in2 = (const gr_complex*)input_items[1];
gr_complex *out = (gr_complex*)output_items[0];

for (i = 0; i < noutput_items; i++) {
  for(j=0 ; j < 4; j++) {
    *out += *in1++;
  }
  for(unsigned int j = 0; j < 2; j++) {
    *out += *in2++;
  }
  *out++;
}
```

forecast function

- `forecast()` is a function which tells the scheduler how many input items are required to produce `noutput_items` output items.
- In most of the case, they are the same:

```
void
    my_general_block::forecast (int noutput_items,
                                gr_vector_int &ninput_items_required
    {
      ninput_items_required[0] = noutput_items;
    }
```

- It is used as an information by the scheduler to schedule the executions of the different blocs so as to prevent starvation or buffer overflows.

consume function

- The `consume (int which_input, int how_many_items)` function tells the scheduler that `how_many_items` of input stream `which_input` were consumed.
- This function should be called at the end of `general_work()`, after all processing is finished
- `consume_each (int how_many_items)` can be used it the number of items to consume is the same on each input streams

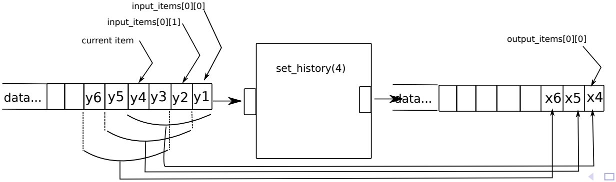
summary: code for my_general_block

```
 my_general_block::general_work (int noutput_items,
             gr_vector_int &ninput_items,
             gr_vector_const_void_star &input_items,
             gr_vector_void_star &output_items)
{
  const gr_complex *in1 = (const gr_complex*)input_items[0];
  const gr_complex *in2 = (const gr_complex*)input_items[1];
  gr_complex *out = (gr_complex*)output_items[0];

  for (i = 0; i < noutput_items; i++) {
    for(j=0 ; j < 4; j++) {
      *out += *in1++;
    }
    for(unsigned int j = 0; j < 2; j++) {
      *out += *in2++;
    }
    *out++;
  }
  consume(0,4*noutput_items);
  consume(1,4*noutput_items);
}
void
    my_general_block::forecast (int noutput_items,
                       gr_vector_int &ninput_items_required)
    {
      ninput_items_required[0] = 4*noutput_items;
      ninput_items_required[1] = 2*noutput_items;
    }
```

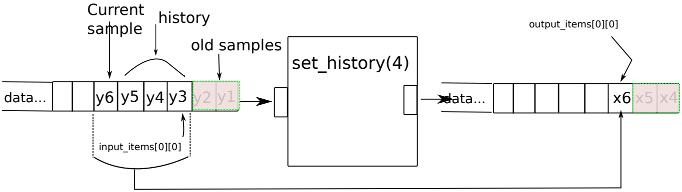
History or pipelined blocs

- Previous example was referred as a block without history in Gnu Radio: *every input is read only once to produce a single output*.
- Or equivalently: each data read is immediately consumed
- Many processing blocs act in a pipeline fashion:
  - produce one output data per input data
  - but... use more than one input data to produce an output data.
- Example of a 4 taps FIR filter:

$$x(i) = \sum_{k=0}^{4} y(i-k)w(k)$$



use of history in blocs



- the `set_history()` function is used by the scheduler to keep some old sample *alive* (or available) to current sample computation.
- `set_history(hist)` means that we are using `hist` sample (including current) to produce current output.
- `input_item[0][0]` points to the oldest sample.
- Usually we shift the input stream: `*in = *(in+hist-1)` such that `*in` point to the current sample.

Other types of blocs

- `gr::sync_block` is derived from `gr::block` and implements a 1:1 block:
  - It has a `work()` function rather than `general_work()` function
  - it omits the unnecessary `ninput_items` parameter, and do not need the `consume_each()` to be called
- `gr::gr_sync_decimator` is used when the number of input items is a fixed multiple of the number of output items.
  - The `gr_sync_decimator` constructor takes a 4th parameter, the decimation factor
  - The user should assume that the number of `ninput_items = noutput_items*decimation`
- `gr::gr_sync_interpolator` is used when the number of output items is a fixed multiple of the number of input items.
  - The `gr_sync_interpolator` constructor takes a 4th parameter, the interpolation factor
  - The user should assume that the number of `ninput_items = noutput_items/interpolation`

GNU Radio scheduler

- Dataflow programming model
- Each block needs a given number of data before running once (i.e. running the `general_work` method)
- the `forecast` method of a bloc indicate this information to the scheduler.
- The scheduler decides to group several execution of each bloc and provides a trade-off between performance efficiency and buffer size between blocs.

Table of Contents

Message passing protocols

- GNU Radio was originally a (infinite) streaming system with no other mechanism to pass data between blocks.
- Not adapted to control data, metadata, and, packet processing
- For solving this problem, gnuradio introduced
  - *Metada files*
  - *Stream tags*
  - *Message passing*
  - All that heavily relying on the polymorphic types

Polymorphic Types: PMT

- Polymorphic Types are opaque data types that are designed as generic containers of data.
- mostly contained in file `pmt.h`

In Python

```
>>> import pmt
>>> P = pmt.from_long(23)
>>> type(P)
<class 'pmt.pmt_swig.swig_int_ptr'>
>>> print P
23
>>> P2 = pmt.from_complex(1j)
>>> type(P2)
<class 'pmt.pmt_swig.swig_int_ptr'>
>>> print P2
0+1i
>>> pmt.is_complex(P2)
True
```

In C++

```
#include <pmt/pmt.h>
// [...]
pmt::pmt_t P = pmt::from_long(23);
std::cout << P << std::endl;
pmt::pmt_t P2 = pmt::from_complex(gr_complex(0, 1));
            // Alternatively: pmt::from_complex(0, 1)
std::cout << P2 << std::endl;
std::cout << pmt::is_complex(P2) << std::endl;
```

PMT function

- Creating, extracting; `pmt::from_<type>`, `pmt::to_<type>`.
- Test, comparison `pmt::is_<type>`
- PMT dictionaries : lists of `key:value` pairs, function for various dictionary operation:

```
pmt_t pmt::dict_add(const pmt_t &dict, const pmt_t &key,
                    const pmt_t &value)#
```

- PMT vectors come in two forms: vectors of PMTs and vectors of uniform data., example of operation:

```
void pmt::vector_set(pmt_t vector, size_t k, pmt_t obj)
```

- The PMT library has methods to serialize data into a string buffer or a string, example:

```
bool pmt::serialize(pmt_t obj, std::streambuf &sink)
```

Metadata files

- Metadata files is a tool to handle metadata on streams (i.e. additional information on samples: rate, types etc.).
- Metadata are present in sample file header.
- There are two kind of Metadata files:
  - inline: headers are inline with the data in the same file.
  - detached: headers are in a separate header file from the data.

Metadata files

- We write metadata files using `gr::blocks::file_meta_sink` and read metadata files using `gr::blocks::file_meta_source`.
- The information that can be contained in a header:
  - `version`: (char) version number (usually set to `METADATA_VERSION`)
  - `rx_rate`: (double) Stream's sample rate
  - `rx_time`: (`pmt::pmt_t pair - (uint64_t, double)`) Time stamp
  - `size`: (int) item size in bytes - reflects vector length if any.
  - `type`: (int) data type
  - `cplx`: (bool) true if data is complex
  - `strt`: (`uint64_t`) start of data relative to current header
  - `bytes`: (`uint64_t`) size of following data segment in bytes

Metadata files: example

- The file metadata header is created with a PMT dictionary of `key:value` pairs,
- then the dictionary is serialized into a string to be written to file.
- Simplest example (`mp`, *make PMT* it a shortcut to the correct `from_<type>` function):

```
const char METADATA_VERSION = 0x0;
pmt::pmt_t header;
header = pmt::make_dict();
header = pmt::dict_add(header, pmt::mp("version"),
                       pmt::mp(METADATA_VERSION));
header = pmt::dict_add(header, pmt::mp("rx_rate"),
                       pmt::mp(samp_rate));
std::string hdr_str = pmt::serialize_str(header);
```

Stream Tags

- Stream tags are an isosynchronous data stream that runs parallel to the main data stream.
- A stream tag:
  - is generated by a block's work function
  - from there on flows downstream with a particular sample
  - until it reaches a sink or is forced to stop propagating by another block.
- Stream tags allows other blocks to identify that an event or action has occurred or should occur on a particular item.

Stream Tags

- An extension to the API of `gr::block` is provided to keep track of absolute item numbers:
  - Each input stream is associated with a concept of the 'number of items read' and
  - each output stream has a 'number of items written.'
- the `gr::tag_t` data type is added to define tags which is composed of the following attributes:
  - `offset`: The offset, in absolute item time, of the tag in the data stream
  - `key`: the PMT symbol identifying the type of tag
  - `value`: the PMT holding the data of the tag.
  - `srcid`: (optional) the PMT symbol identifying the block which created the tag
- Example of stream tag API function:

```
void add_item_tag(unsigned int which_output, const tag_t &tag);
```

Message passing

- Stream tags are useful to pass information with samples but it only goes in one direction
- We need message passing
  - to allow blocks downstream to communicate back to blocks upstream.
  - to communicate back and forth between external applications and GNU Radio(e.g. MAC layer)
- The message passing interface API has been added to the `gr::basic_block` module.
- Message passing between block is identified by dashed lines in `gnuradio-companion` (- - - - - - -)

Message passing

- A block has to declare its input and output message ports in its constructor:

```
void message_port_register_in(pmt::pmt_t port_id)
void message_port_register_out(pmt::pmt_t port_id)
```

- The ports are now identifiable by that a global port name.
- Other blocks may want to post a messages,
- They must subscribe to the port and the publish message on it

```
void message_port_pub(pmt::pmt_t port_id,
pmt::pmt_t msg);
void message_port_sub(pmt::pmt_t port_id,
pmt::pmt_t target);
```