Using an oscilloscope as a GNURadio source through a VXI11 communication link

J.-M Friedt, February 2, 2020

GNURadio being opensource software, the user is encouraged to add missing functionality. One attractive aspect of `gnuradio` is its ability to perform signal processing *in real time*, as opposed to post-processing in Matlab. My development technique is to first record some signals in a file and process them with Matlab to make sure I understand the theory, then convert the signal processing steps from Matlab to C(++) and process the same file from the `gnuradio` environment to make sure results are the same, and finally modifiy the `gnuradio` block input and output for real time processing.

In this example I wish to show how the output of an oscilloscope can be streamed into GNURadio for real time analysis of the resulting signal (e.g. cross-correlating the two channels of the oscilloscope).

GNURadio processing blocks are written in Python or C++. Despite some communication toolbox available for Python, I am not good at Python programming and am familiar with controlling instruments through the Ethernet link using a protocol commonly known as VXI11 or LXI. GNURadio provides a skeleton generator named `gr_modtool` which creates all the files and core function (constructor, destructor and work function called by the scheduler), as well as the XML description of the block when displayed in `gnuradio-companion`.

We first create a new class which we will call `gr-oscilloscope` with

```
gr_modtool create oscilloscope
```

Inside this processing class, we describe dedicated blocks: at the moment we shall have a single processing block, which we will also call `oscilloscope`:

```
cd gr-oscilloscope  # enter the project directory
gr_modtool add oscilloscope
```

in which we inform `gr_modtool` that the block is a *source* written in C++ with no argument and no Quality Assurance code is provided.

We have now created the skeleton of the project. The directory structure is

1. `lib` contains the C++ source files, and will be the directory in which we spend most time,
2. `grc` contains the XML file describing the block structure and parameters (if any),
3. `apps`, `include` and `swig` are ready for use and will not be modified,
4. `example` should be filled with some example flowchart for the user to get started.

In addition to these default directories, `cmake` requires that an independent directory, commonly called `build`, be created as a subdirectory of the project for compiling the program. Hence, after creating `build` next to these directories, we enter the new directory (`cd build`) and ask cmake to configure the compilation environment: `cmake ../`. If all dependencies have been met, this step will be completed with the creation of a `Makefile`, which is executed with `make -j8`. Upon completion of the compilation *the software must be installed* to be visible from gnuradio-companion: this is done with `sudo make install`. Launching `gnuradio-companion` after the new processing block will display a new entry in the list of blocks called `oscilloscope`, in which our processing block will be available. However before compiling, we need to fill a few parameters that `gr_modtool` was unaware of when creating the skeleton, namely the type and number of input(s) and output(s).

We wish to feed GNURadio with the two channel outputs of an oscilloscope: both outputs are real values (as opposed to complex), so we can start by filling `grc/oscilloscope_oscilloscope.xml` with the appropriate parameters:

```
<?xml version="1.0"?>
<block>
  <name>oscilloscope</name>
  <key>oscilloscope_oscilloscope</key>
  <category>[oscilloscope]</category>
  <import>import oscilloscope</import>
  <make>oscilloscope.oscilloscope()</make>

  <source>
    <name>out1</name>
    <type>float</type>
  </source>
  <source>
    <name>out2</name>
    <type>float</type>
  </source>
</block
```

states that the block has no input (no sink field since we are not connected to any other block generating data before us), and we are the source of two streams called `out1` and `out2`, both of which are 32-bit floating point types. All the other entries can be deleted in this simple examples.

We shall from now on focus on the `lib` directory, and namely `oscilloscope_impl.cc`. The file skeleton comes with some fields to be filled before it can first be compiled:

- `gr::io_signature::make(<+MIN_OUT+>, <+MAX_OUT+>, sizeof(<+OTYPE+>))` in the *constructor* must be replaced with the type of data we generate, namely floating point values: `gr::io_signature::make(2,2,sizeof(float))` stating that two streams of floats are generated.
- `<+OTYPE+> *out = (<+OTYPE+> *) output_items[0];` in the *work* function must be adapted as well not only with the stream type, replacing `OTYPE` with `float`, but also with the fact that we have *two* output streams:

```
    oscilloscope_impl::work(int noutput_items,
      gr_vector_const_void_star &input_items,
      gr_vector_void_star &output_items)
    {float *out0 = (float *) output_items[0]; // 2 outputs
     float *out1 = (float *) output_items[1];
...
```

We are now ready to fill the processing blocks: the header file in the `lib/` directory allows defining private and public variables.

1 TCP/server feeding gnuradio with data

Controlling the instrument requires the ability to communicate through VXI11 – also known as LXI[1], a protocol running over TCP/IP (make sure that the computer IP address is on the same subnet than the instrument subnet). In order to avoid tackling two issues at the same time, we start with a dummy TCP/IP server which will later be replaced by a VXI11 client connecting to the instrument.

A brief description of the main features of the `lib/oscilloscope_impl.cc` program is as follows:

```
namespace gr {
  namespace oscilloscope {

    oscilloscope::sptr
    oscilloscope::make()
    {return gnuradio::get_initial_sptr
        (new oscilloscope_impl());
    }

    oscilloscope_impl::oscilloscope_impl() // The private constructor
      : gr::sync_block("oscilloscope",
              gr::io_signature::make(0, 0, 0),
              gr::io_signature::make(2, 2, sizeof(float)))
    {int length;
     struct sockaddr_in adresse;
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     adresse.sin_family=AF_INET;
     adresse.sin_addr.s_addr = inet_addr("127.0.0.1"); // localhost
     adresse.sin_port =htons(9999);
     bzero(&(adresse.sin_zero),8);
     length = sizeof(adresse);
     result=connect(sockfd, (struct sockaddr *)&adresse, length);
    }
```

The private constructor is in charge of initializing the block: in our case, it opens the socket with the localhost on port 9999, and connects to the server through a TCP/IPv4 connection (`AF_INET, SOCK_STREAM`). This function is only called once at the initialization of the block.

```
    oscilloscope_impl::~oscilloscope_impl() // Our virtual destructor.
    {int val;
     val=htonl(-1);
     write(sockfd,&val,sizeof(int));
```

[1] `http://www.lxistandard.org/`

```
     close(sockfd);
    }
```

The destructor is called when leaving GNURadio, and is in charge of freeing allcated resources. In our case we only close the socket when the network link is no longer needed.

The following work function is the core of the processing part: it is an infinite loop called by the GNURadio scheduler. The scheduler requests `noutput_items` from the source, to be filled in the two arrays `float *out0` and `float *out1` since we have declared two output streams.

```
    int oscilloscope_impl::work(int noutput_items,
      gr_vector_const_void_star &input_items,
      gr_vector_void_star &output_items)
    {float *out0 = (float *) output_items[0]; // 2 outputs
     float *out1 = (float *) output_items[1];
     float tab1[2*noutput_items];
     float tab2[2*noutput_items];
     long int k,val;
     val=htonl(noutput_items*2);
     write(sockfd,&val,sizeof(long int));
     read(sockfd, tab1, sizeof(float)*noutput_items*2);
     for (k=0;k<noutput_items;k++) {tab2[k]=tab1[2*k];tab1[k]=tab1[2*k+1];} // deinterleave
     for (k=0;k<noutput_items;k++)
          {out0[k]=tab1[k];
           out1[k]=tab2[k];
          }
      return noutput_items; // Tell runtime system how many output items we produced.
    }
  } /* namespace oscilloscope */
} /* namespace gr */
```

This function hence requests `noutput_items*2` samples from the server (*2 for the two channels) by sending the amount of data expected to the server, and then fetches the resulting sample. We will see in this example that the synthetic data are just sine waves generated on the fly: they will later become samples collected from the oscilloscope.

A private variable `sockfd` describing the access point is defined in `lib/oscilloscope_impl.h` and accessible (and shared) by all the functions of the class:

```
namespace gr {
  namespace oscilloscope {
    class oscilloscope_impl : public oscilloscope
    {private:
       int sockfd;
     public:
      oscilloscope_impl();
...
```

This very simple client connects to port 9999 of the localhost (127.0.0.1) and fetches as many data as requested by the GNURadio scheduler, as mentioned in the `noutput_items`. We fill the two arrays `out1` and `out2` with the values sent by the server in `tab1`. The server is told the communication is completed when the client send the value -1, otherwise the client requests `noutput_items` values from the server every time the scheduler needs some new value to process.

The server is similarly very simple and kept to a bare minimum

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <arpa/inet.h>
#include <math.h>
#include <unistd.h>

#define MY_PORT         9999

int main()
{int sockfd;
 struct sockaddr_in self;
 float *buffer;
 struct sockaddr_in client_addr;
 int clientfd,addrlen=sizeof(client_addr);
 int size,k;
```

```
17
18  sockfd = socket(AF_INET, SOCK_STREAM, 0);  // socket type: TCP/IP
19  bzero(&self, sizeof(self));
20  self.sin_family = AF_INET;
21  self.sin_port = htons(MY_PORT);
22  self.sin_addr.s_addr = INADDR_ANY;
23
24  bind(sockfd, (struct sockaddr*)&self, sizeof(self));
25  listen(sockfd, 20);
```

At this point the socket has been initialized with a TCP/IPv4 protocol (again `AF_INET, SOCK_STREAM`) and the server is listening for any incoming connection from clients. The infinite loop now expects connections from clients, and sends as many data as requested by the client (`size` variable – notice the network to host (`ntohl`) conversion to avoid any issue of endianness between communicating processor architectures). The data sent are cosine and sine values, sent until the client sends a request for a negative number of samples, which indicates the client wishes to stop communicating.

```
1  while (1) {
2   clientfd = accept(sockfd, (struct sockaddr*)&client_addr, &addrlen);
3   printf("%s:%d␣connected\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
4   while (size!=-1)
5     {recv(clientfd, &size, sizeof(long), 0);
6      size=ntohl(size);
7      printf("%d\n",size);
8      if (size>0)
9         {buffer=(float*)malloc(sizeof(float)*size);
10         for (k=0;k<size/2;k++)
11             {buffer[2*k]=sin(2*M_PI*(float)k/(float)size*10);
12              buffer[2*k+1]=cos(2*M_PI*(float)k/(float)size*10);
13             }
14         send(clientfd, buffer, size*sizeof(float), 0);
15        }
16     }
17   close(clientfd);
18  }
19  close(sockfd);return(0);  // Clean up (should never get here)
20 }
```

The server waits for a connection on port 9999 of the localhost. Upon connection, we accept the client and look at the amount of data it requests. Knowing the size of the vector to be sent back, we fill a vector `buffer` with sine waves (floating point values) which are sent back to the client. We then wait for a new request (blocking `recv`), unless the size request we have received is negative, indicating the end of the transactions. The result of this graph is shown in the Fig. 1.

2 Communicating with the hardware

Like most modern instruments, the Agilent Infiniium 54855A DSO is fitted with an Ethernet port allowing for a VXI11 connection. This protocol has been implemented under GNU/Linux at the University of Nottingham, with the latest release available at `github.com/applied-optics/vxi11`. This library is extremely simple to use since once the IP address of the instrument is known and the computer is configured on the same subnet (check by `ping` from the computer to the instrument IP address), the link is opened with `vxi11_open_device()`. Data exchanges are then performed using `vxi11_receive()` or `vxi11_send()`, as is well illustrated in the application examples at `https://github.com/applied-optics/vxi11/tree/master/utils`. In the case of GNURadio, we shall replace the client initialization in the constructor with the VXI11 initialization, and the work function is filled with sending GPIB commands and fetching the samples from the waveforms. The only trick then lies in compiling the software using `cmake`: assuming we have put the github repository under the `lib` directory – putting all the source codes needed for compilation in `lib/vxi11/library` – we add to the `CMakeLists.txt` in the `lib/` directory the following source files:

```
1 list(APPEND oscilloscope_sources
2     oscilloscope_impl.cc
3     vxi11/library/vxi11_clnt.c
4     vxi11/library/vxi11_user.c
5     vxi11/library/vxi11_xdr.c
6    )
```

(notice that the first two lines already exist, we only need to add the source files implementing the VXI11 protocol). The program now looks like

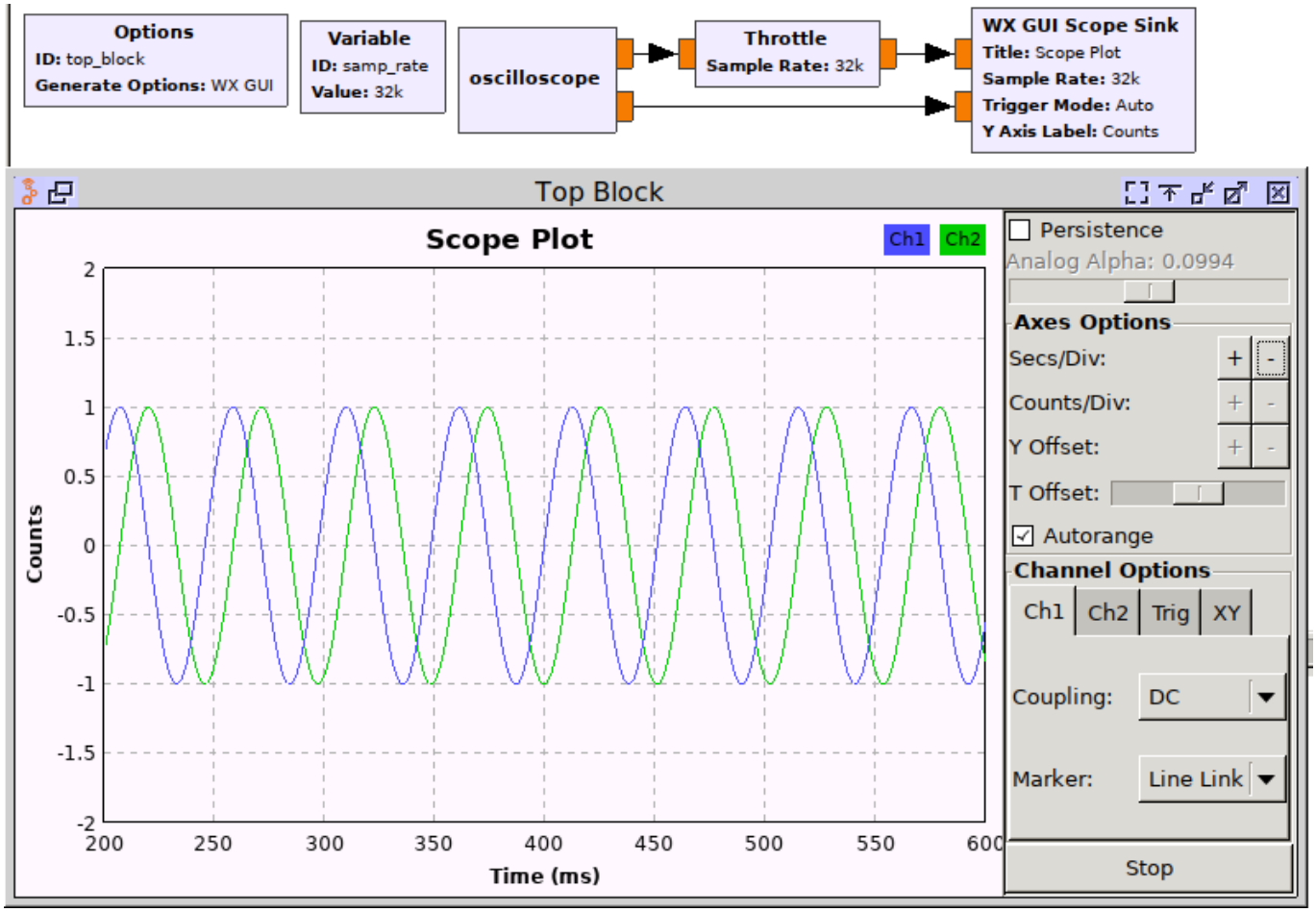


Figure 1: Gnuradio displays the datastream generated by a TCP/IP server feeding the source block.

```
#define SIZE 80000

namespace gr {
  namespace oscilloscope {

    oscilloscope::sptr
    oscilloscope::make()
    {return gnuradio::get_initial_sptr
        (new oscilloscope_impl());
    }

    oscilloscope_impl::oscilloscope_impl() // The private constructor
      : gr::sync_block("oscilloscope",
              gr::io_signature::make(0, 0, 0),
              gr::io_signature::make(2, 2, sizeof(float)))
    {char  device_ip[25];
     char *device_name=NULL;
     sprintf(device_ip,"169.254.202.240"); // Agilent 54855DSO
     if (vxi11_open_device(&dev,device_ip,device_name)!=0) printf("erreur ouverture\n");
        else printf("connect OK\n");
     char buffer[256];
     int buffer_length=256;
     float sampleDuration=5e-7,samplingRate=1e10; // 1 GHz
     int sampleSize;
     sampleSize = (int)(sampleDuration * samplingRate);
     sprintf(buffer,"*IDN?");
     envoi(dev,buffer);
     relit(dev,buffer,buffer_length);
     printf("%s\n",buffer);
```

```
30
31      sprintf(buffer,"*CLS"); envoi(dev,buffer);
32      sprintf(buffer,"*RST"); envoi(dev,buffer);
33      sprintf(buffer,":SYSTEM:HEADER␣OFF"); envoi(dev,buffer);
34 //   sprintf(buffer,":AUTOSCALE");envoi(dev,buffer); // + (sampleDuration));
35      sprintf(buffer,":TRIGGER:EDGE:SOUCE␣CHANNEL1;SLOPE␣POSITIVE");envoi(dev,buffer);
36      sprintf(buffer,":TRIGGER:EDGE:LEVEL␣CHANNEL1,0.0");envoi(dev,buffer);
37      sprintf(buffer,":TIMEBASE:REFERENCE␣LEFT;POSITION␣0;RANGE␣%e",sampleDuration);envoi(dev,buffer);
38      sprintf(buffer,":CHANNEL1:RANGE␣.5;OFFSET␣0.0");envoi(dev,buffer);
39      sprintf(buffer,":CHANNEL2:RANGE␣.5;OFFSET␣0.0");envoi(dev,buffer);
40      sprintf(buffer,":TRIGGER:SWEEP␣SINGLE"); envoi(dev,buffer);
41      sprintf(buffer,":ACQUIRE:MODE␣RTIME;AVERAGE␣OFF;SRATE␣%e;POINTS␣%d",samplingRate,sampleSize);envoi(dev,buffer)
42 // Right Click on sine wave on top of display, Setup Acquisition and see SamplingRate/MemDepth
43     }
```

The constructor looks quite similar as before, except that the socket connection is replaced with a VXI11 connexion, and the oscilloscope initialization functions are executed once upon block initialization.

```
1     oscilloscope_impl::~oscilloscope_impl() // Our virtual destructor.
2     {char buffer[256];
3      printf("Bye\n");
4      sprintf(buffer,":CHANNEL1:DISPLAY␣ON"); envoi(dev,buffer);
5      sprintf(buffer,":CHANNEL2:DISPLAY␣ON"); envoi(dev,buffer);
6      sprintf(buffer,":TRIGGER:SWEEP␣AUTO"); envoi(dev,buffer);
7     }
8
9     int oscilloscope_impl::work(int noutput_items,
10      gr_vector_const_void_star &input_items,
11      gr_vector_void_star &output_items)
12    {float *out0 = (float *) output_items[0]; // 2 outputs
13     float *out1 = (float *) output_items[1];
14     float tab1[2*noutput_items];
15     float tab2[2*noutput_items];
16     long int k,val,offset;
17     char buffer[SIZE];
18     int buffer_length=SIZE;
19     int res;
20     sprintf(buffer,":DIGITIZE␣CHANNEL1,CHANNEL2\n");envoi(dev,buffer);
21     sprintf(buffer,":WAVEFORM:SOURCE␣CHANNEL1");envoi(dev,buffer);
22     sprintf(buffer,":WAVEFORM:FORMAT␣WORD;BYTEORDER␣LSBFIRST\n");envoi(dev,buffer);
23     sprintf(buffer,":WAVEFORM:DATA?");envoi(dev,buffer);
24     res=relit(dev,buffer,buffer_length);
25     if (buffer[0]!='#') printf("error␣in␣trace␣header\n"); // #
26     else
27       {offset=buffer[1]-'0'; // header size
28        for (k=0;((k<((res-offset-1)/2)) && (k<noutput_items));k++)  // rm # and header
29           tab1[k]=(float)(*(short*)(&buffer[2*k+offset+2]))/65536.; // valid only on Intel/LE
30        sprintf(buffer,":WAVEFORM:SOURCE␣CHANNEL2");envoi(dev,buffer);
31        sprintf(buffer,":WAVEFORM:VIEW␣MAIN");envoi(dev,buffer);
32        sprintf(buffer,":WAVEFORM:FORMAT␣WORD;BYTEORDER␣LSBFIRST\n");envoi(dev,buffer);
33        sprintf(buffer,":WAVEFORM:DATA?");envoi(dev,buffer);
34        res=relit(dev,buffer,buffer_length);
35        if (buffer[0]!='#') printf("error␣in␣trace␣header\n"); // #
36        else
37          {offset=buffer[1]-'0';
38           for (k=0;((k<((res-offset-1)/2)) && (k<noutput_items));k++)  // rm # and header
39              tab2[k]=(float)(*(short*)(&buffer[2*k+offset+2]))/65536.; // valid only on Intel/LE
40          }
41       }
42     for (k=0;k<noutput_items;k++)
43          {out0[k]=tab1[k];
44           out1[k]=tab2[k];
45          }
46      return noutput_items; // Tell runtime system how many output items we produced.
47     }
48   } /* namespace oscilloscope */
```

```
49 } /* namespace gr */
```

The work function loops by continuously requesting waveforms from the oscilloscope. At the moment we have not taken care of the consistency between the amount of data requested by the scheduler (`noutput_items`) and the number of samples provided by the oscilloscope (`res/2`, which actually has hardly any relation with the requested number of samples defined by the GPIB function `ACQUIRE:POINTS`). Two wrapper functions have been written for easily switching from VXI11 to GPIB peripherals: sending (`envoi()`) et receiving (`relit()`) is achieved with

```
1 int relit(VXI11_CLINK *clink,char *buffer,int buffer_length)
2 {int ret;
3  ret=vxi11_receive(clink,buffer,buffer_length);
4  buffer[ret-1]=0;
5  return(ret);
6 }
7
8 void envoi(VXI11_CLINK *clink,char *buffer)
9 {vxi11_send(clink, buffer,strlen(buffer));}
```

Here again, a private variable describing the access point is defined in `lib/oscilloscope_impl.h`:

```
1     class oscilloscope_impl : public oscilloscope
2     {private:
3        VXI11_CLINK   *dev;
4      public:
5       oscilloscope_impl();
6 ...
```

The resulting flowgraph allows for plotting both oscilloscope channels in real time, and hence feed the GNURadio flowgraph with more fancy processing steps (Fig. 2).

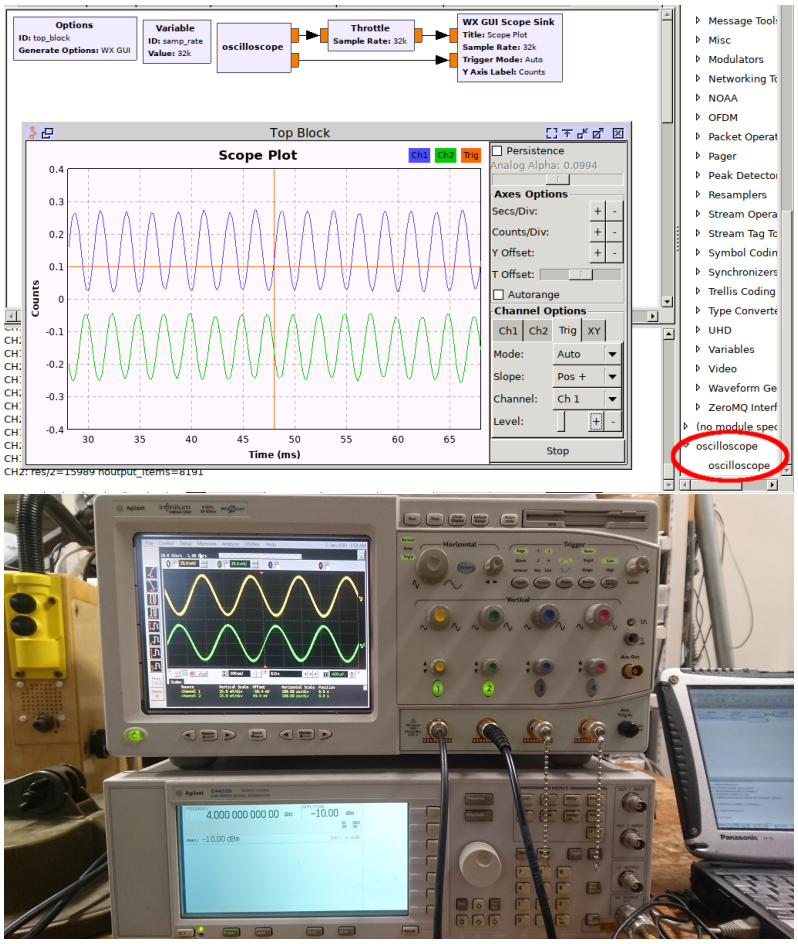
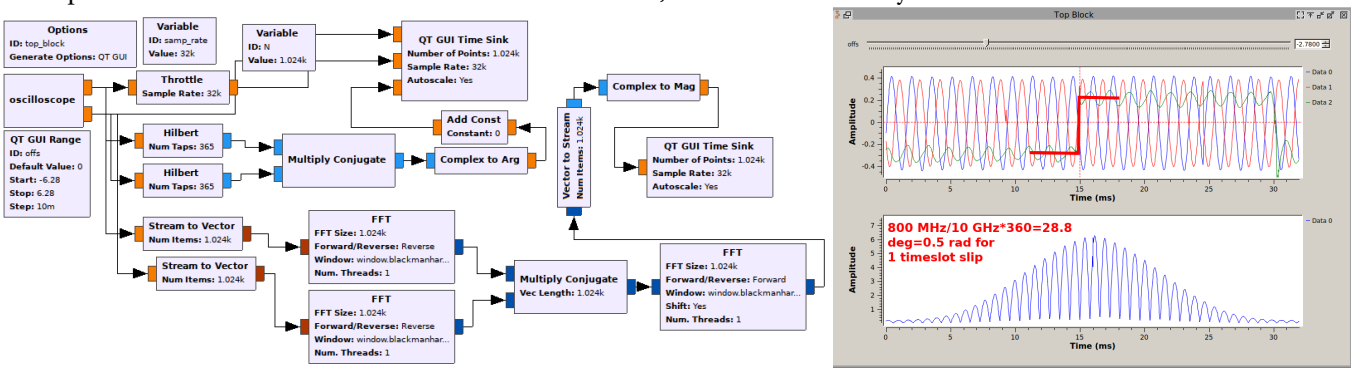


Figure 2: Top: result from GNURadio companion acquiring data from the oscilloscope. Bottom: experimental setup.

Real time acquisition and processing of the collected data allows for real time analysis of the cross correlation (computed as the product of the Fourier transorm of the two time series) and hence assess any drift between the two channels.



3 Callback functions

So far the acqusition parameters are set upon compiling the oscilloscope block, and changing measurement duration, sampling rate or voltage range requires compiling again the block. Being able to dynamically change these measurement parameters to adapt to signal quality would be welcome: such a functionality is implemented through callback functions. The `gnuradio-companion` interface is informed of the variables in the XML description of the block, and the C++ implementation accepts as input parameters these parameters. Changing these functions as the flowgraph is running is possible through callback functions implemented again in C++. Hence, adding these parameter tuning callback functions requires updating four files:

1. `gr-oscilloscope/grc/oscilloscope_oscilloscope.xml` defines the parmeters including their name and the type of value expected (integer, floating point number, complex ...),
2. `gr-oscilloscope/include/oscilloscope/oscilloscope.h` defines the constructor prototype, i.e. the types of the arguments the user will provide when starting the flowgraph,
3. `gr-oscilloscope/lib/oscilloscope_impl.h` defines the private variables holding the parameters, the callback function prototypes and constructor prototype,
4. `gr-oscilloscope/lib/oscilloscope_impl.cc` defines the C++ implementation of the callback functions and uses the incoming parameters to configure the oscilloscope during the initialization step.

We have implemented callback functions for the sampling rate, measurement duration and IP address definition of the VXI11 peripheral. The first two parameters can be changed as the flowgraph is running, while modifying the former only makes sense before the flowgraph execution is started. We only develop the case of the sampling rate, the other callback functions are similar.

3.1 XML file

The XML file defining the block properties is updated with the appropriate fields, the callback function is defined, and the parameters provided to the main function defined:

```
<make>oscilloscope.oscilloscope($ip,$range,$rate,$duration)</make>
<callback>set_rate($rate)</callback>

<param>
  <name>Rate</name>
  <key>rate</key>
  <type>int</type>
</param>
```

Here the parameter is of type integer: it could have been float, complex, string ... depending on the type of the argument. The types of the parameters must be consistent with the prototype of the constructor and the callback functions.

3.2 Header files

Two header files must be filled with the types of the arguments provided: `gr-oscilloscope/include/oscilloscope/oscillosco` declares the `make` method which must accept the right list of arguments :

```
static sptr make(char*,float,int,float);
```

Additionnally, `gr-oscilloscope/lib/oscilloscope_impl.h` declares the private variables, callback function prototypes and constructor prototype

```
    class oscilloscope_impl : public oscilloscope
    {private:
       int _rate;

     public:
      void set_rate(int);
      oscilloscope_impl(char*,float,int,float);
```

Now that all the prototypes are defined, the implementation must be addressed.

3.3 C++ implementation file

The C++ implementation file in `gr-oscilloscope/lib` is updated accordingly

```
namespace gr {
  namespace oscilloscope {
    oscilloscope::sptr
    oscilloscope::make(char *ip,float range,int rate,float duration)
    {
      return gnuradio::get_initial_sptr
        (new oscilloscope_impl(ip,range,rate,duration));
    }

    oscilloscope_impl::oscilloscope_impl(char *ip,float range,int rate,float duration)
      : gr::sync_block("oscilloscope",
             gr::io_signature::make(0, 0, 0),
             gr::io_signature::make(2, 2, sizeof(float))),
         _range(range),_rate(rate),_duration(duration)
    {
... oscilloscope VXI11 & GPIB initialization commands ...
     set_range(rate);
}

void oscilloscope_impl::set_rate(int rate)
{char buffer[256];
 int samplesize = (int)(_duration * (float)rate);
 printf("new␣rate:␣%f\n",rate);fflush(stdout);
 sprintf(buffer,":ACQUIRE:MODE␣RTIME;AVERAGE␣OFF;SRATE␣%e;POINTS␣%d",rate,samplesize);
 envoi(dev,buffer);
 _rate=rate;
}
  } /* namespace oscilloscope */
} /* namespace gr */
```