ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

A Graphical User Interface for the LCM Software Radio

José Ferreiro and Jérôme Braure
5th year Communications Systems Departement students
Mobile Communications Laboratory
Supervisor: Linus Gasser **Teacher**: Bixio Rimoldi

6th February 2003

This report is dedicated to my beautiful girlfriend, Diana, for all her love, patience and support, for the joy that she has brought into my life, and support of me in all my endeavors.

José Ferreiro

To my cat.

Jérôme Braure

Contents

List of Figures

List of Tables

Thank you!

We would like to sincerely thank Mr. Linus Gasser who helped us in different ways to realize this project with his precious information, material and reading.

◇ The current version of this document is available in *.pdf, & .ps* format from:
`http://diwww.epfl.ch/~jferreir/softwareRadio/`

Abstract

This report contains a description about the project task which is to develop a user friendly interface to display the LCM software radio data, the elaboration of the adopted solution and the final result.

1 Introduction

Let's say that we have at our disposal several black boxes with their own features inside. We do not care about what happens inside. The black boxes can receive inputs, produce outputs, and can be connected together.

In the real world it is easy to connect these black boxes manually. We can assume that the outputs and inputs are linked with wires.

The challenge of this project was to find a way to show the interconnections in a user friendly way. Therefore we developed an algorithm to give them the most suitable disposition on a graphical display. The display is dynamically updated during runtime with fresh data!

1.1 LCM software radio

The *"Laboratoire de Communications Mobiles (LCM) Software Radio"* Test-bed is a project with the objective of designing and implementing a real-time software platform to validate advanced mobile communication signal processing algorithms. A *UMTS*-like system has already been implemented.

1.1.1 Signal Processing classes (SPC)

The LCM Software Radio does the following things:

1. load the software radio **core components**. The information related to these elements is stored in the folder `cdb/` (`class database`) of the `sradio/` root directory.

2. load as many *instantiated* classes (**subsystem**) from the `cdb` as we want into the radio system. The subsystem is composed by a list of **S**ignal **P**rocessing **C**lasses (**SPC**). We name the SPC elements **modules** in this report. The information related to the subsystem is stored in the folder `sdb/` (`subsystem database`).

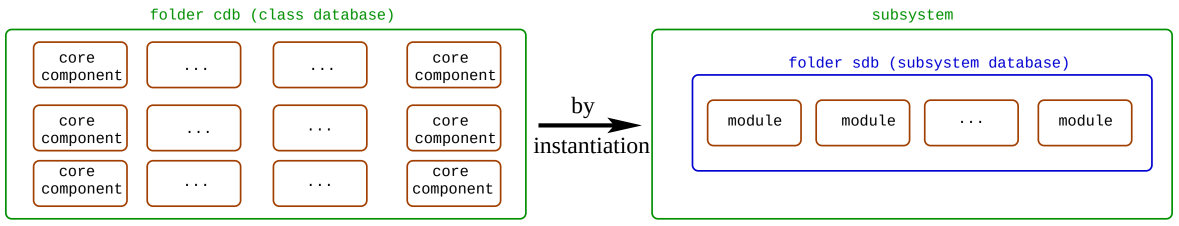

Figure 1: LCM software structure

LCM radio core components are out of the scope of this project. We are only interested in the data of the `sdb` folder. You will find detailed information in the next section.

1.2 Job description

The *LCM Software Radio* generates a lot of data which is stored in a way that will be introduced in the next section. A **G**raphical **U**ser **I**nterface (**GUI**) is required to display the current state of the *modules* and their different interconnections during the runtime.

Let's enumerate the main tasks to be accomplished :

- To **extract** and **parse** the data generated from the *LCM Software Radio*
- To build a nice and user friendly **GUI tool**
- To show the different **interconnections** between the modules
- To display the modules' **information** (e.g. statistics)

We chose the *Qt free c++ toolkit* to develop the application, which provides GUI development functionalities and above all is *multi-platform*, according to our supervisor.

2 Underlying Data structure

The *LCM Software Radio* generates the data and stores it in a hierarchical order into different folders. The root folder is named `sradio`. The data structure is the core of our GUI because the latter needs the data all the time to display some state of the Software Radio. Without data, the GUI would have nothing to show!

Figure 2 shows the storage data structure. As we introduced before, `sradio/` is the *root* folder which contains two sub folders: `cdb/` and `sdb/`. We are only interested in the `sdb` folder which contains all the data related to the *LCM's Software Radio* modules that are running at a given time.

Each instanciated folder is named in hexadecimal mode (e.g. `001F/`) and contains a different number of files associated to a given module. We are going to classified the files in two classes:

1. **static**: these files are always present in the module folder
   - `config`: data about the configuration
   - `inputs`: data about the input ports
   - `name`: module name
   - `outputs`: data about the output ports
   - `stats`: data for statistics to plot and display
2. **dynamic**: those ones are optional
   - <`block`*XX*>: this file derives from the `stats` file where *XX* corresponds to an integer between `01` to `99` (e.g. `block03`).
   - <`port_out`*YY*>: this file contains the data that corresponds to a given module port output where `00` $\geq$ *YY* < `#output ports` (e.g. `port_out01`).

The reader will find detailed information for all *static* files except `config` (out of the scope) in the next section. The dynamic files store data blocks. We use them to build the associated plots to be displayed for each module on the GUI if the user requests it.

The purpose of all these modules is to build a *UMTS* frame. Therefore, there exists a special module named **STFA** (**S**lot **T**o **F**rame **A**llocation). The STFA has fifteen inputs, each input corresponds to

```
sradio/                        root folder
 |__cdb/                       class database  (out of the project scope)
 |__sdb/                       subsystem database
     |__<number>/              number given to a module (SPC)
     |    |__<file>            name of a file
     |__<...>/
     |    |__<...>
     |__001F/                  module number
     |    |__config            file out of the project scope
     |    |__inputs            input port connexions
     |    |__name              module name
     |    |__outputs           output port connexions
     |    |__stats             statistics
     |    |__blockXX           statistics data block
     |    |__port_outYY        output port data block
     |__<N-1>/
     |    |__<...>
     |__<folder N>/
          |__<...>
```

Figure 2: LCM Software Radio data structure tree

a *slot*. Each of these slots has different functions and some may change their functions during the uptime of a radio. The *UMTS standard* has fifteen slots; this radio-structure parameter `N` can be configurated at *LCM Software radio* (currently `N = 15`). A slot is built by a chain of modules (SPCs). The input of the STFA is sent through the air and the output of the STFA is what has been received.

We are now able with this information about all the modules to display their state at a given time on the GUI.

2.1 Module data structure

We are going to give a detailed table for each *static* file previously defined. These files are stored into the `sdb` folder.

The character `#` means "number" in the following tables.

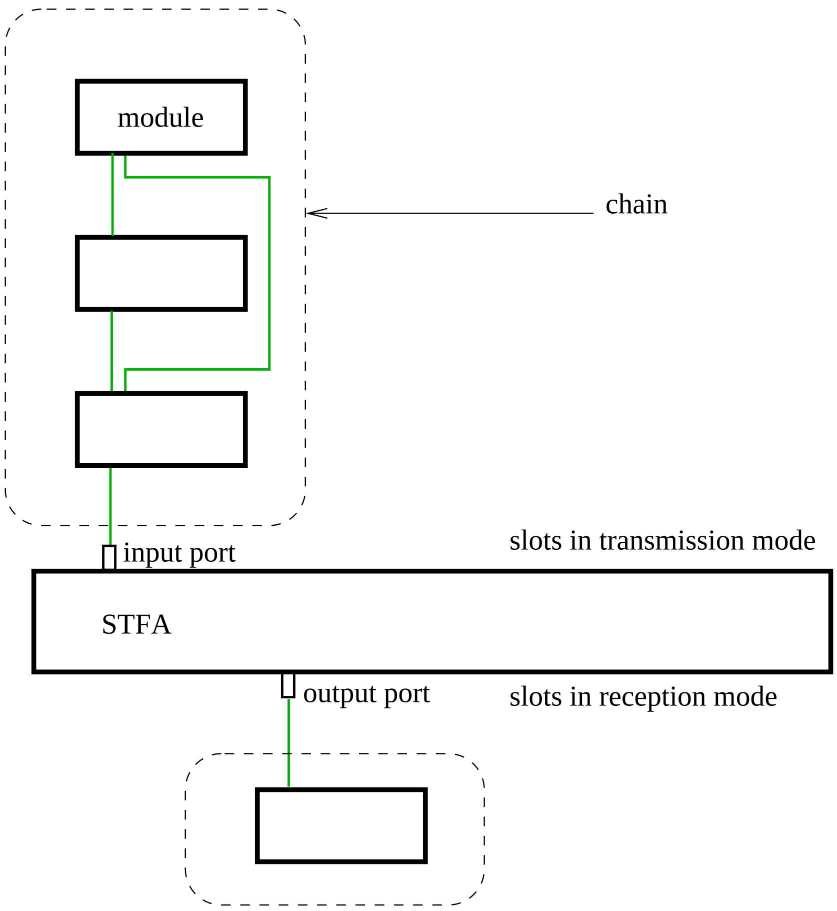

Figure 3: STFA with one chain to build a slot

2.1.1 `name` file

Table 1 shows the unique attribute in the `name` file.

Name
`<module name>`

Table 1: `name` file format

2.1.2 `inputs` file

Table 2 shows some of the features of `inputs` file.

Inputs				
`<#input port>`	`<type>`	`<length>`	`<#previous module>`	`<#port previous module>`

Table 2: `inputs` file format

2.1.3 `outputs` file

Table 3 shows some of the features of `outputs` file.

Outputs				
`<#output port>`	`<type>`	`<length>`	`<#next module>`	`<#port next module>`

Table 3: `outputs` file format

2.1.4 `stats` file

Tables 4 and 5 show the features of `stats` file.

Stats			
`<variable name>`	`<index>`	`<type> = (1 | 2 | 3)`	`<decimal value>`

Table 4: `stats` file format when the type $\leq$ `3`

Stats				
`<variable name>`	`<index>`	`<type>`	`<type_p>`	`<type_p length>`

Table 5: `stats` file format when the type is `4`

The `<type>` variable from Tables 4 and 5 are shown in Table 6. Moreover, Table 7 derives from Table 5 which describes the different variable data types that `<type_p>` respectively can take.

<type>	
Value	**Data type**
0	*none*
1	`int`
2	`double`
3	`String[128]`
4	`pointer`

Table 6: variable `<type>`

<type_p>	
Value	**Data type**
0	*none*
1	u8, (unsigned char)
2	s16, (short int)
3	(real_s16, imag_s16), (short int, short int)
4	real_s16, imag_s16, 0, 0
5	s12 (signed 12-bit)
6	s32 (signed 32-bit)

Table 7: variable `<type_p>`

Sample file with comments:

```
debug@radio1:~/testdir/sradio/sdb/0004/stats> more stats
4                         // stats file  contains 4 variables
offset, 0, 1, 00000       // <variable name>, <index>, <type>, <decimal value>
energy, 1, 1, 00000
MaFi_re, 2, 4, 2, 2048  // <variable name>, <index>, <type>, <type_p>, <type_p length>
MaFi_im, 3, 4, 2, 2048
```

We can notice the following things:

- `stats` file belongs to the module numbered `0004` in the `sdb/` folder. This is known by looking at the path directory (first line).
- there are two variables of `type` $\leq$ `3` (`offset & energy`), and two others where it is equal to `4` (`MaFi_re & MaFi_im`). As the two last variables have the same `<type_p>`, that is, the number `2`, we can conclude by looking at Table 7 that their `Data type` is `s16 (short int)`.

2.1.5 `block`*XX* and `port_out`*YY* files

Both contain data blocks that can be plotted. If we look back at the `stats` file sample given in the previous subsection, we can extract the following line:

```
MaFi_re, 2, 4, 2, 2048  // <variable name>, <index>, <type>, <type_p>, <type_p length>
```

The <`block`*XX*> file is associated with the variable name `MaFi_re` where the `type` is equal to `4`. Let's analyze this line deeply. `MaFi_re` has the associated file: `block02` where as we notice *XX* is equal to `02`, and `02` is the variable `<index>`. `block02` contains the data block associated to the variable name `MaFi_re` that can be displayed on the GUI if the user makes the corresponding request (this subject will be explained in the next section). Similarly, for the other file (`port_out`*YY*) we will get this time a data block related to a given output port.

2.1.6 Remarks

We parse all the data according to the previously mentioned file formats table. If some files should be modified to another format, it will likely follow that the GUI will not run correctly.

3 C++ classes description

This section provides a short description of each class. Figures 4 and 5 depict simplified UML class diagrams. They only mention attributes and functions that help to get a better understanding of the general behaviour.

3.1 Module

A `Module` is an instance of an `SPC`. It contains all information related to one particular instance like, the number of inputs or outputs, two lists of I/O links, a list of statistical data concerning the current module, a.s.o.
It also holds information about the graphical features (see Figure 6) like the module size and position, the name and value of the variable shown on it. There are even variables used to keep track of which are the direct neighbours, and a boolean telling whether the module is currently visible.

3.2 IOLink

This class represents input/output links that connect modules together. It is very simple, as it only consists of four integer variables: the source and destination module and port. There is only a header, no `.cpp` file.

3.3 Block

This class[1] is not directly instantiated (virtual); it has two subclasses: `Port` and `Stats` and inherits from class `QFile`. It represents a block of data, that could for example be plotted. Its function named `Data( double *x, double *y, int samples )` loads the data from the file to two vectors in memory.

3.4 Port

Subclass of `Block`, it initialises a port, that is, checks for it's validity and puts the right filename. Each `Port` object refers to one of the `port_out`*XX* file.

3.5 Stats

Second subclass of `Block`, it implements statistics data. It inherits from `QFile` but depending on the type of statistic, may not refer to any file when the concerned data is only an integer.

3.6 ModuleGenerator

The module generator is the object that browses through the file system to create the modules. In the software-radio simulation mode, a file may be unavailable for a short period of time. In order to handle this problem efficiently, the module generator re-attempts to access the file in case of an unsuccessful read.
When a module has been built, it is appended to the mapper's module list.

[1]Classes `Block`, `Port`, `Stats`, and `Show` were written by supervisor Linus Gasser

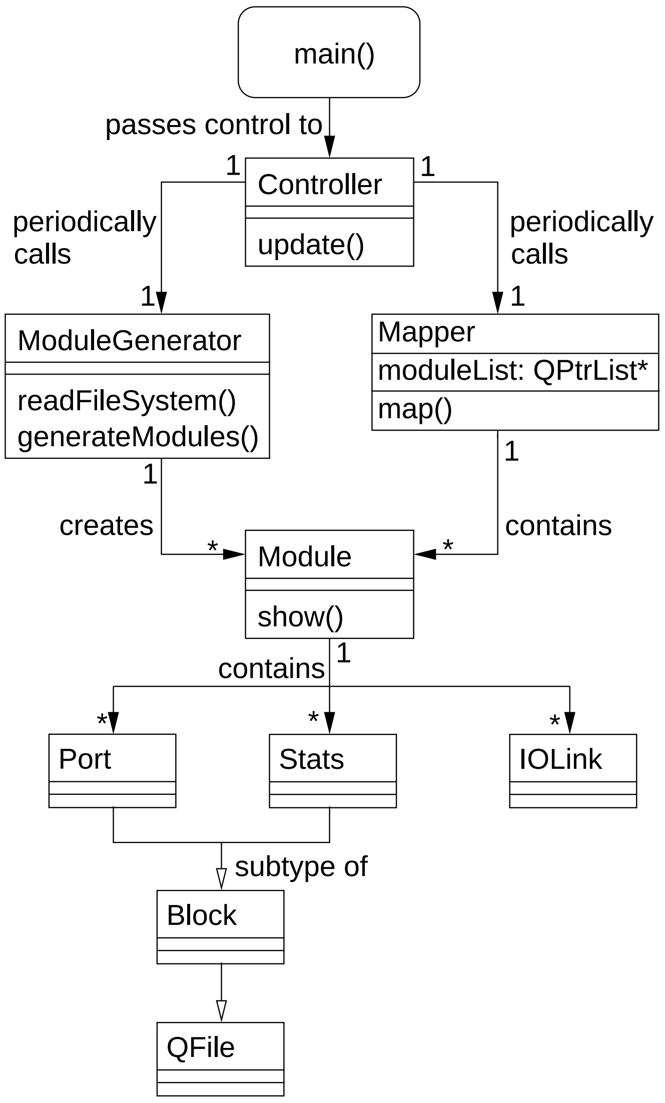

Figure 4: Simplified UML class diagram showing the classes related to the software radio data.

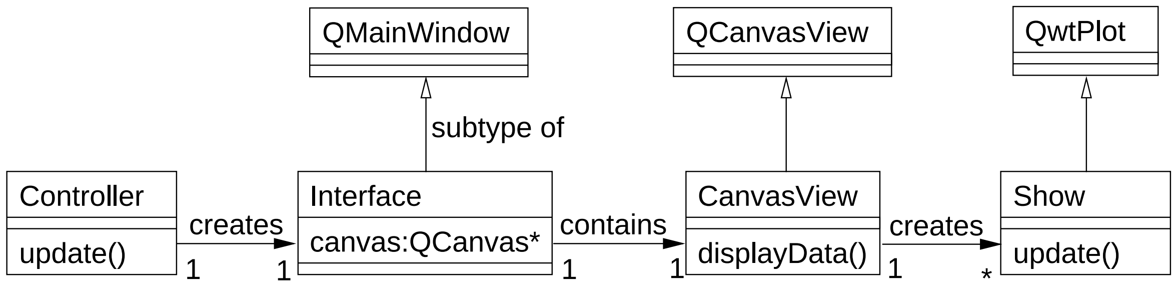

Figure 5: Simplified UML class diagram showing the classes related to the graphical part.

3.7 Mapper

The mapper's task is to take care of displaying the modules on the GUI. It first has to decide in which order they will be aligned (see next section) and then has the responsibility to draw them with their links. The mapper is the object through which the modules can be accessed, via one of the two `getModule()` functions. A module can be searched either given its number or its location (actually the returned module is the one that contains a given point).
The mapper holds a couple of other variables needed to keep track of the position of the already-drawn modules (e.g. `inputMaxXTab`) and a list of small data blocks needed to remember which variables were displayed before the last update.

3.8 Controller

The controller is the entity that regularly asks the module generator and the mapper to perform their tasks in order to refresh the displayed data.

3.9 DisplayedData

This short class, consisting only of a header file, two integers, a string and a "set" function is used to remember which variables the user has decided to monitor. Without this class, the displayed data would be reset to the default value at each update.

3.10 Interface

This is the graphical interface's main part. It is a subclass of `QMainWindow` and its central widget is the canvas on which the modules are drawn.

3.11 CanvasView

Sub-typing `QCanvasView`, this class is needed to handle the mouse events which are: a) move the canvas b) open a contextual menu.

3.12 Show

This is a subclass of `QwtPlot` and can be used to display either statistical data or the output of a module. The format can be chosen between *complex, real, imaginary*, and *absolute*.

3.13 `defines.h`

The file `defines.h` is not a class; it just gathers the `#define`s that can be modified to tune the GUI differently. For example, the colors, sizes, timer periods, a.s.o, can be easily changed and the software recompiled.

4 Drawing and connecting modules

The main challenge of this project was to think of an efficient way to arrange the modules and provide a so-called *mapping* algorithm to implement it. In order to improve readability it was decided to first draw the modules that are directly connected together. Thus we start by drawing a module, say `m1`, that is linked with the STFA, then we draw a module that is directly connected to `m1` and so on until the current module has no more input (or output, depending on which side of the STFA is processed). The result will be a chain of modules whose root is the STFA.

That is a good start but there is more to do: a module can have several inputs (or outputs). So, at each iteration of the algorithm only one link is drawn while the others, the *remaining* links, (see Figure 7) have to be put into a vector so that they can be handled later. After a complete chain of module has been built, the *remaining* links can be drawn. But among those exist some links of a third kind: the *interzone* links which stretch from one zone to another. These are then put in yet another list, to be processed only when all modules of a given zone have been displayed. A graphical representation of these events is shown on Figure 8.

5 Viewing information and plotting data with the GUI

We are now able to display a GUI. The final snapshot of the *LCM Software Radio* GUI (see Figure 9) shows the different slot chains with their associated modules.

Each module is represented by a rectangle which contains some information when the GUI is started. It contains three lines that are described in the following enumeration:

1. module name
2. a `stats` module variable and its value
3. another `stats` module variable and its value

In Figure 10 one can see a subpart of the GUI. Four modules are shown, twice a `matched_filter` and twice a `demodulator_qpsk`. The variables in Figure 10 are `energy` and `offset` belonging to `matched_filter` modules.

We decided to show only two module variables at the same time in the module rectangle. This choice was made to avoid large module rectangles and to display efficiently different information.

We added a *menu bar* at the top. For the moment it doesn't offer any special application to be launched. We introduced it with the idea that additional items could be added in the future. In addition, the user can interact with the mouse in the GUI, get information about the module statistics, and plot data by pressing the right mouse button. In following subsections, we explain the different mouse button functionalities that can be performed.

5.1 Left mouse button

The number of modules displayed on the interface can be arbitrarily large. Thus, they may not all be visible at the same time, extending beyond the window borders. Scrollbars were added to solve that

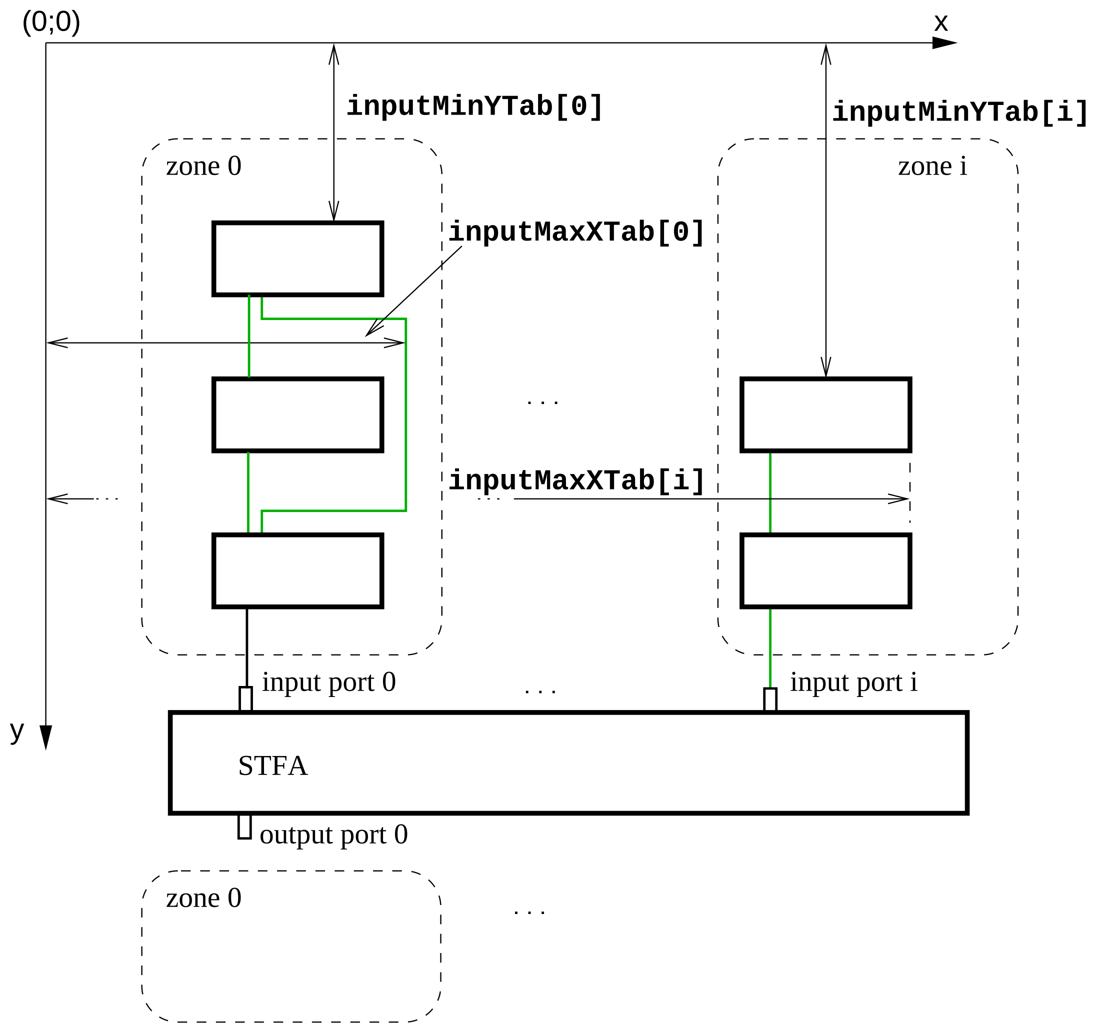

Figure 6: Each module (except the STFA) belongs to a zone. There are arrays (e.g. `inputMaxXTab[]`) that hold global variables needed to keep track of the different zones' position.

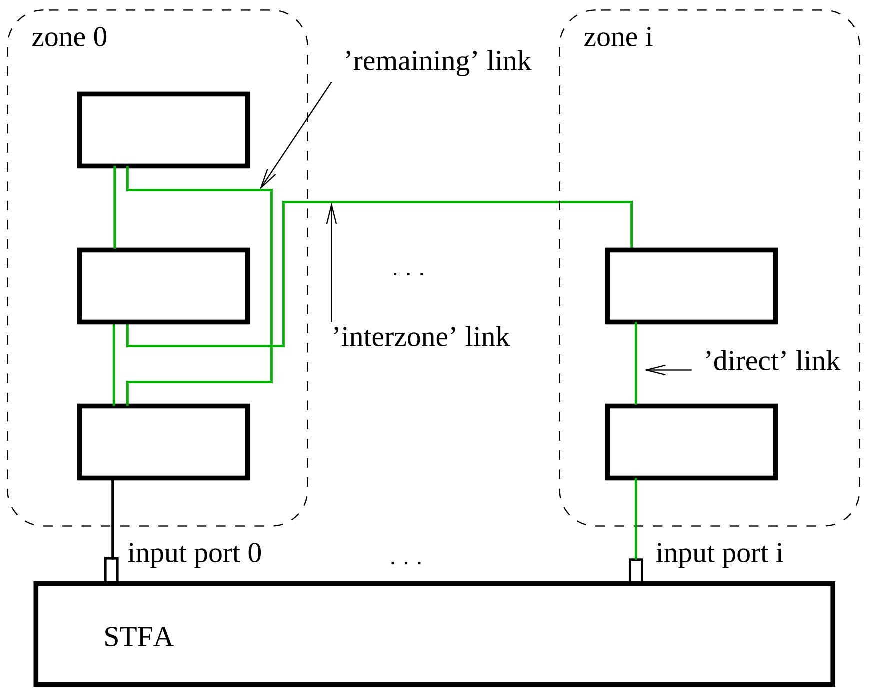

Figure 7: There are three types of links: *direct* links, which connect neighbouring modules, *remaining* links who connect not-neighbouring modules but still within the same zone, and finally *interzone* links.

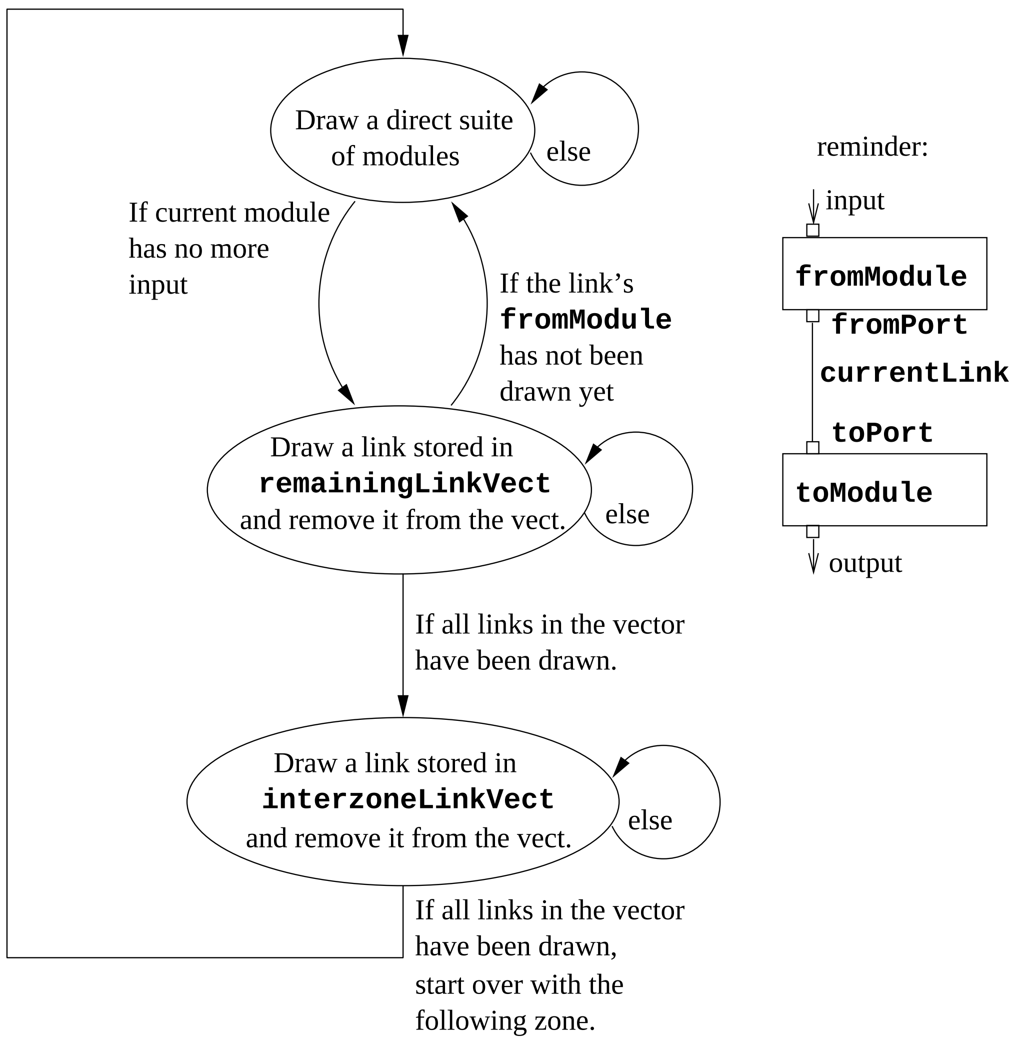
Draw a direct suite of modules
else
If current module has no more input
If the link's fromModule has not been drawn yet
Draw a link stored in remainingLinkVect and remove it from the vect.
else
If all links in the vector have been drawn.
Draw a link stored in interzoneLinkVect and remove it from the vect.
else
If all links in the vector have been drawn, start over with the following zone.
reminder:
input
fromModule
fromPort
currentLink
toPort
toModule
output

Figure 8: The core of the mapping algorithm works as a state machine. The one depicted here is the one used for drawing the modules that are above the STFA. A similar mechanism is used for the other modules (i.e. below the STFA).

problem and to provide more efficiency and comfort, we implemented a canvas-dragging feature. To see hidden parts of the canvas, the user just has to drag the display area in the appropriate direction.

5.2 Right mouse button

The user gets a pop up menu when he right clicks on a module.

The pop up menu is composed by three items as shown in the Figures 11 and 12. The first item is the module name. A pull-down menu is shown when the user moves the mouse over one of the next two items:

1. `display data`
2. `display outputs`

The user can access to the module's information through the items in the pull-down menu. Technically, each pull-down menu consists of a list of variables for the `display data` item. The different variables and their values can be displayed in the corresponding module. Not all items are variables

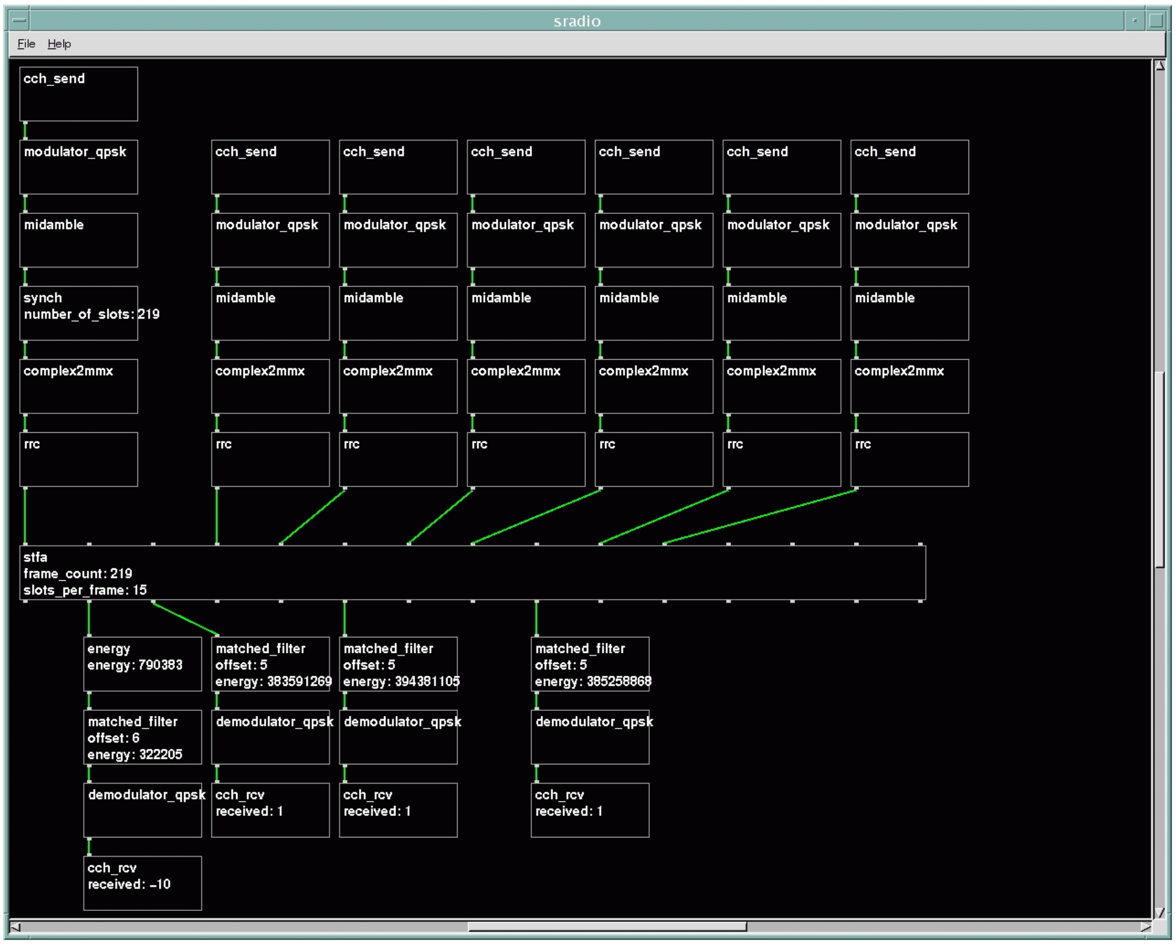

Figure 9: LCM Software Radio **G**raphical **U**ser **I**nterface

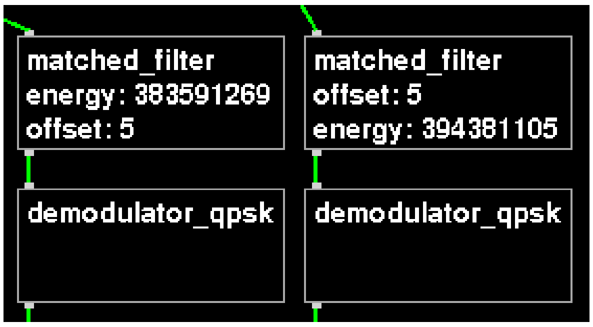

Figure 10: GUI module samples

that can be shown in the module, some generate a plot, that will be displayed in a individual window. In the second pull-down menu, we have a list of output ports for the `display port` item that can be plotted individually, too. Plots have some peculiarities: when the user clicks on the combo box (Figures 13 and 17), a list pops up, from which he can choose the display style, which is one of the following: *complex, real, imaginary,* and *absolute*. These different styles are shown in the Figures 13, 14, 15 and 16. If there are no variables to be shown the item `<no data>` will be displayed.

5.3 Conclusion

We achieved a very satisfactory result. The GUI fulfills to the LCM Software Radio GUI objectives.

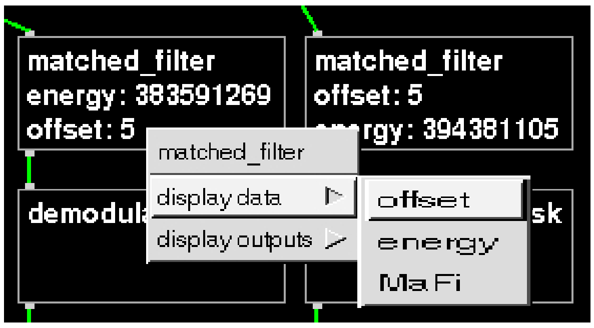

Figure 11: `display data` pull-down menu from `matched_filter`

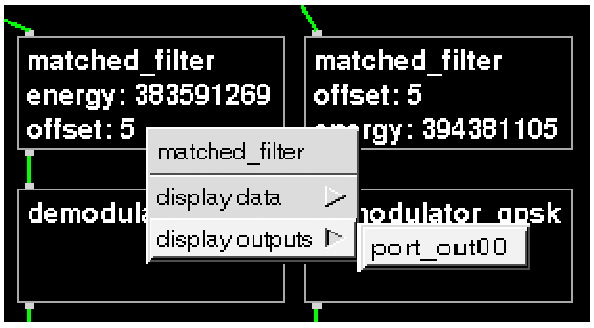

Figure 12: `display outputs` pull-down menu from `matched_filter`

6 Conclusion

We started the GUI from scratch, thus accomplishing all the project tasks described at the beginning. A lot of functionalities could be added, for example, *LCM Software Radio* could be started from the GUI, and some other special functions for each module should be added depending on the user's requirements. Therefore, we suggest that other students should extend the *LCM Graphical Interface*.

Ecublens, 6th February 2003

José Ferreiro Jérôme Braure

A Source Code

Enumeration of the source files:

1. `main.cpp`
2. `block.h` & `block.cpp`
3. `canvasview.h` & `canvasview.cpp`
4. `controller.h` & `controller.cpp`
5. `defines.h`
6. `displayeddata.h`
7. `interface.h` & `interface.cpp`
8. `iolink.h`
9. `mapper.h` & `mapper.cpp`
10. `module.h` & `module.cpp`
11. `modulegenerator.h` & `modulegenerator.cpp`
12. `port.h` & `port.cpp`
13. `show.h` & `show.cpp`
14. `stats.h` & `stats.cpp`

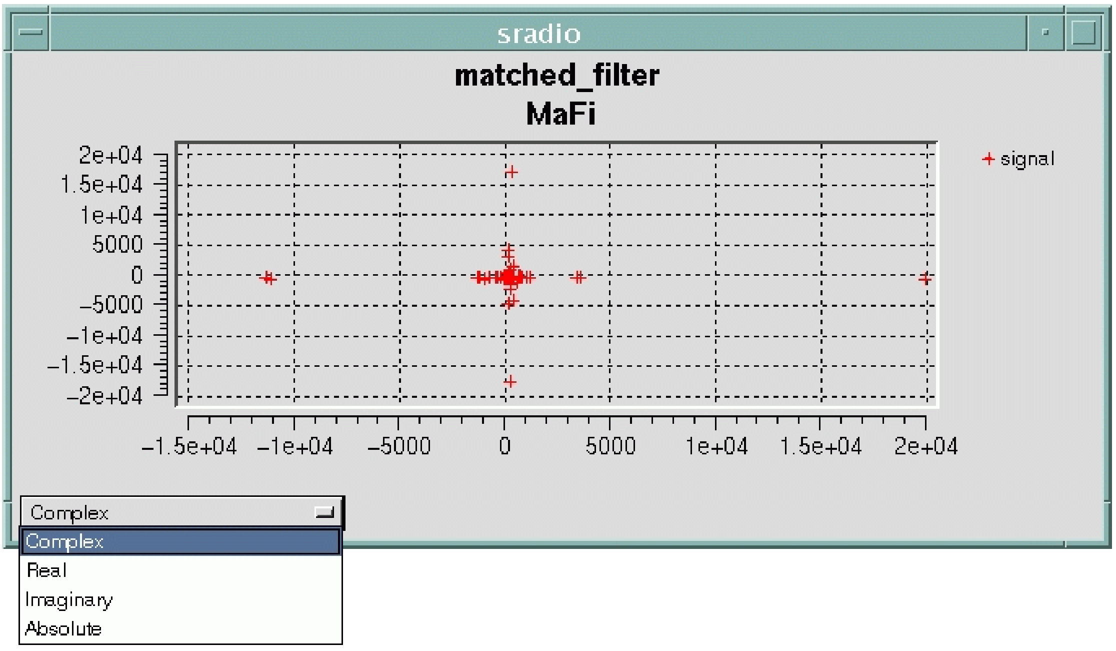

Figure 13: variable `MaFi` from `matched_filter` (complex display)

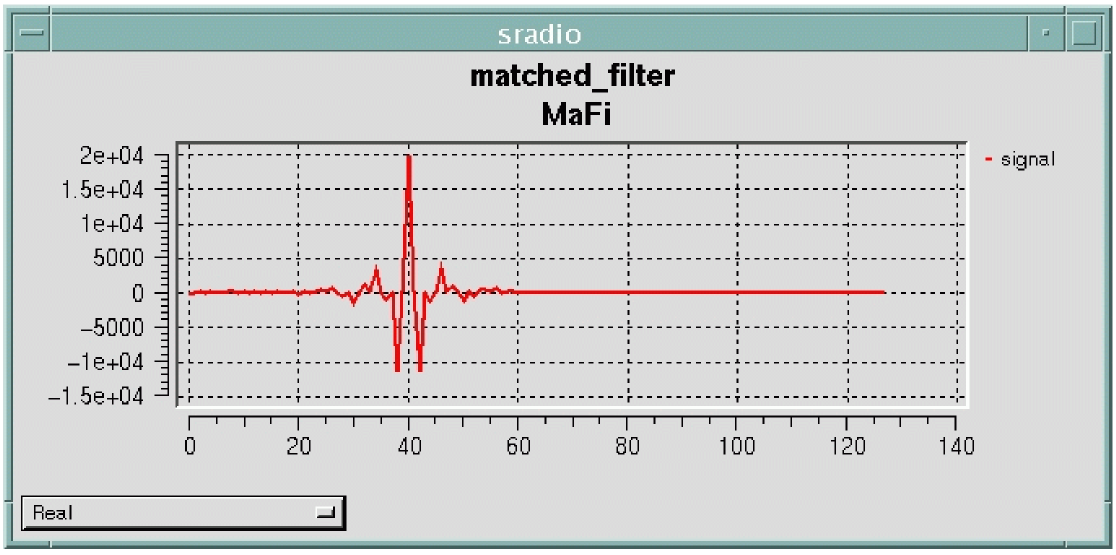

Figure 14: variable `MaFi` from `matched_filter` module (real part)

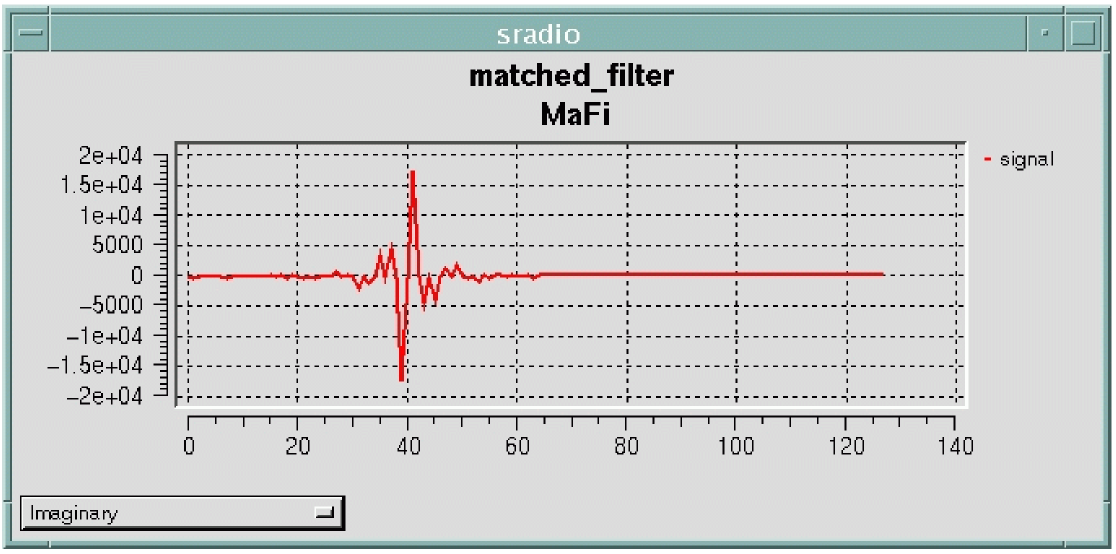

Figure 15: variable `MaFi` from `matched_filter` module (imaginary part)

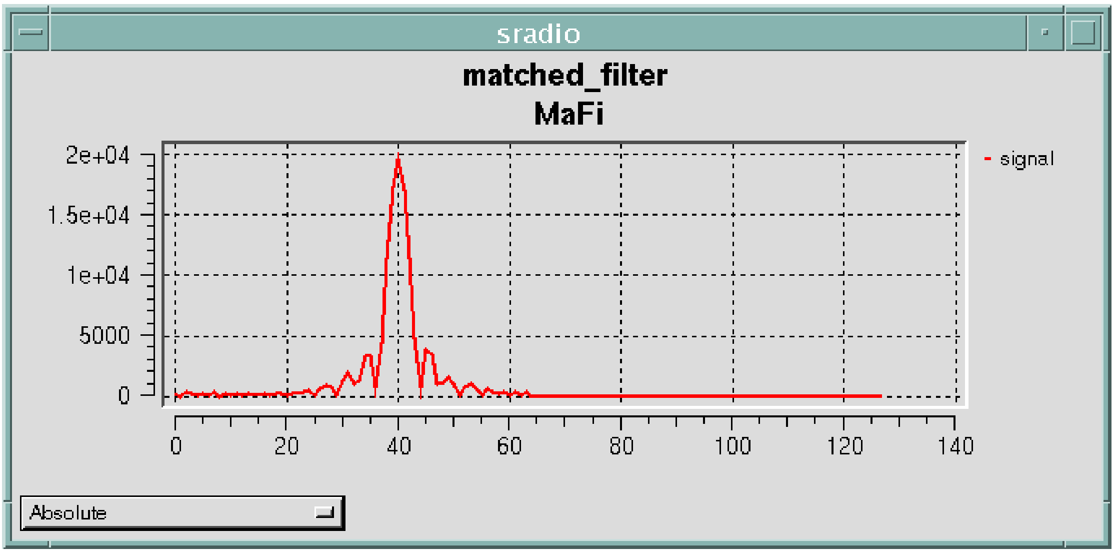

Figure 16: variable `MaFi` from `matched_filter` module (absolute value)

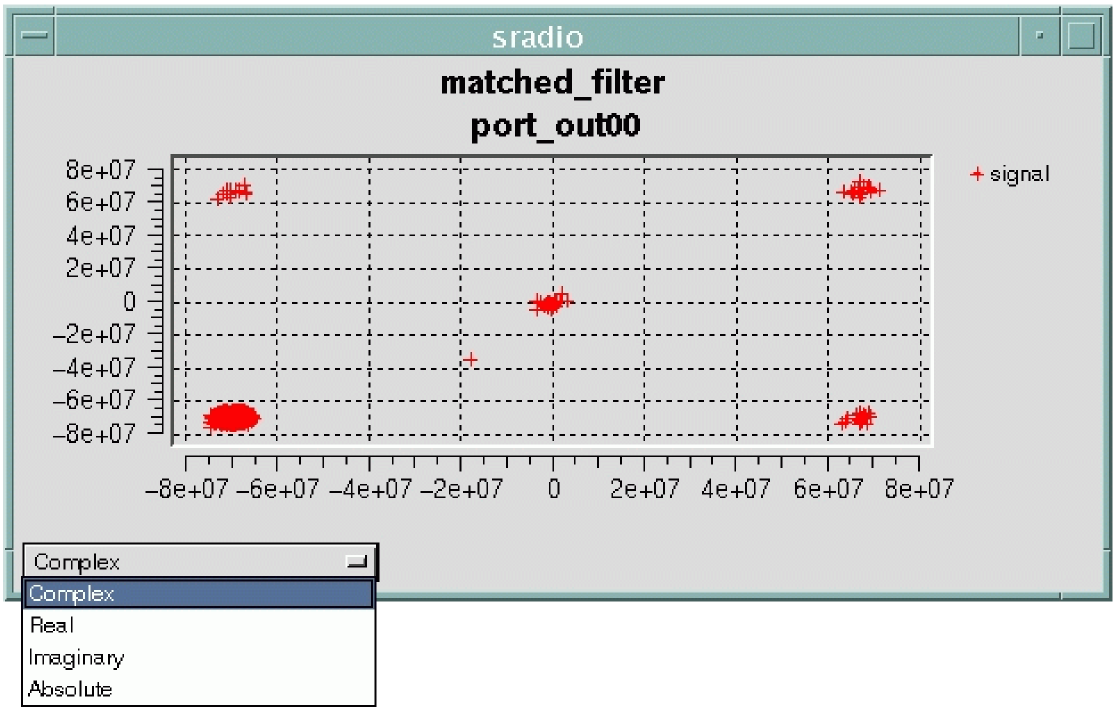

Figure 17: `port_out00` from `matched_filter` module (complex axis)

References

[1] Matthias Kalle Dalheimer. *Programming with Qt, Second Edition.* O'Reilly Verlag Gmbh & Co. KG. (Editor) 2002.
`ISBN: 0-596-00064-2`

[2] Tony Gadis. *Standard Version of Starting Out with C++, Third Edition.* Scott/Jones, Inc. (Editor) 2001. `ISBN: 1-57676-063-3`

[3] On-line Qt reference documentation in HTML format. `http://doc.trolltech.com/3.1/`

[4] On-line Qwt user's guide in HTML format. `http://qwt.sourceforge.net/`