UNIVERSITÁ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Computer Engineering

Implementation of a suite of components for Software Defined Radio using an SCA-compliant framework

Tesi di Laurea

Candidato:
Fabio Del Vigna

Relatori:
Prof. Marco Luise

Prof. Mario Giovanni C. A. Cimino

Ing. Carmine Vitiello

Anno Accademico 2013/2014

"Experience is a hard teacher because she gives the test first, the lesson afterwards."
–Vernon Law

Abstract

The aim of this work is to introduce Software Defined Radio (SDR) technology, present an open source SCA-compliant framework whose name is Redhawk, which derives from the OSSIE project and describe an implementation example of some processing instances.
Since in SDR applications it is necessary to run the same software on different hardware, portability becomes the main important aspect in the development of software radio applications. The use of a SCA-compliant framework solves this issue making hardware transparent to the programmer and reducing time and costs of code development.
This aspect can be exploited for prototyping applications quickly without the need of a specific hardware or testing new standards and protocols.
We will introduce some basic concepts of SDR, of the SCA architecture, based on CORBA, and Redhawk. We will then talk about of the implementation of a suite of components, written by using Redhawk IDE and C++ programming language. These will be tied together to form an application called waveform.
We will also present the results obtained by enforcing a certain level of parallelism in our algorithm to speed up computation in Redhawk components and boost performances against a more simpler non concurrent implementation of the same algorithms.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is SDR

Nowadays communication protocols appear, change and disappears at an impressive rate, making the realization of radio devices compliant with all the commercial and military standards quite difficult and expensive. It's pretty common to find inside a smartphone some chips appointed to signal processing for common protocols like **GSM**, **UMTS**, **HSDPA**, **Wi-Fi**, **Bluetooth**, **NFC** and other.
Clearly with the rapid introduction of new standards, todays chips will be not adequate in a short time, and are not flexible to be updated since are developed using ASIC technology.
A SDR device is capable of working on a wide band of the spectrum and to handle different kinds of standards by substituting or updating internal firmware but without the necessity to change the hardware [1].
This is achievable thanks to the growing computational power of General Purpose Processor units of consumer electronics and PCs and to the wide diffusion of DSPs and FPGAs. This solution is progressively shadowing the success of ASICs that are connected to very high development costs and risks, requiring huge volumes of production to be economically convenient.
Software Defined Radio is a technology that allows developers to implement radio applications that are capable of processing baseband signals through blocks defined using software, while data is acquired through a programmable device that implements a RF *front-end*.
The aim is to exploit programmable hardware to execute the computations and managing RF signal processing required by a radio application (waveform). A waveform is responsible for signal processing and it is composed by a series of components, variable in number, which performs all the needed computations and communicate to each other by using a middleware

or a transport layer protocol such as TCP/IP. More practically a waveform can be intended as the series of digital operations employed in transmission and reception of the signal.
A typical SDR *front-end* device is a chain of a filter, a down converter and a ADC/DAC, plus some elaboration units like a DSP, FPGA and GPP. SDR is suitable for situations in which it is necessary to prototype new standards and are subjected to frequent modifications, when a high portability is required for the application or in some fields like satellite communications or mobile since it is convenient to be able to change the application without the necessity to renew completely the hardware, a solution not always trivial, especially for satellites. On the other hand this technology is still less performant than ASIC solutions and it is not always well power optimized.
In this scenario it is requested to perform real-time analysis and processing of the signals; often to accomplish this it is required to develop code with very high performances and optimizations. In this work we will implement on a SCA complaint framework such as Redhawk a suite of components that have been optimized for this purpose and we will show how different implementations will affect performances.

Chapter 2

Overview on SDR

2.1 History

Software Defined Radio was born in 70's in the defense sector as a possible evolution of the ordinary concept of radio, thanks to the evolution of the Information Technology. The first time the Software Radio was used was in 1984 with E-Systems which first realized the filtering operation digitally. The term "Software Defined Radio" was coined in 1991 by Joseph Mitola, who published the first paper on the topic in 1992 [2]. During 90's several project were started as the SPEAKEasy (phase I and II) and the DARPA, more flexible and adaptive. The primary goal of the SpeakEasy project was to use programmable processing to emulate more than 10 existing military radios, operating in frequency bands between 2 and 2000 MHz. SpeakEasy design included the fastest DSP available at the time, the Texas Instruments TMS320C40 processor, which ran at 40 MHz, progressively improved in order to support four C40s. Programmable hardware made possible the separation of the sampling of the signal from the filtering, demodulation, and more in general processing of signals. Nowadays many applications for radio amateurs are available and the hardware as become cheaper and affordable to many. The wide diffusion of mobile phones based on cellular communications gave an important impulse to the SDR technology.
Market is looking at SDR evolution that is becoming smarter and smarter and is moving toward the field of the cognitive radios that are able to "sense" frequencies, noise, geographical location.

2.2 SDR paradigm

Differently from classical radios, SDR radios are programmable devices able to change its features through software.
Radios can be classified accordingly to their capabilities:

Hardware Radio (HR): : The system cannot be changed via software;

Software-Controlled Radio (SCR): : Control functionality implemented in software, but change of attributes such as modulation and frequency band cannot be done without changing hardware;

Software-Defined Radio (SDR): : Capable of covering wide frequency range and of executing software to provide variety of modulation techniques, wide-band or narrow-band operation, communications security functions and meet waveform performance requirements of relevant legacy systems. System software should be capable of applying new or replacement modules for added functionality or bug fixes without reloading entire set of software. Separate antenna system followed by some wideband filtering, amplification, and down conversion prior to receive A/D-conversion. The transmission chain provides reverse function of D/A-conversion, analog up-conversion, filtering and amplification;

Ideal Software Radio (ISR): : All of capabilities of software defined radio, but eliminates analog amplification and heterodyne mixing prior to A/D-conversion and after D/A conversion;

Ultimate Software Radio (USR): : Ideal software radio in a chip, requires no external antenna and has no restrictions on operating frequency.

Software Defined Radio is further classified into three categories:

- Reconfigurable Radio, based on FPGA technology only.
- Programmable Radio, based on a hybrid architecture DSP/FPGA
- Fully Software Radio, based on General Purpose Processor

In our work we adopt fully software-radio approach employing Ettus USRP platform and RTL2832U devices.
Following tho approach a radio for SDR is associated to a PC of which it exploits the computational power, while it acts as a *front-end*. The *front-end* is responsible for sampling the

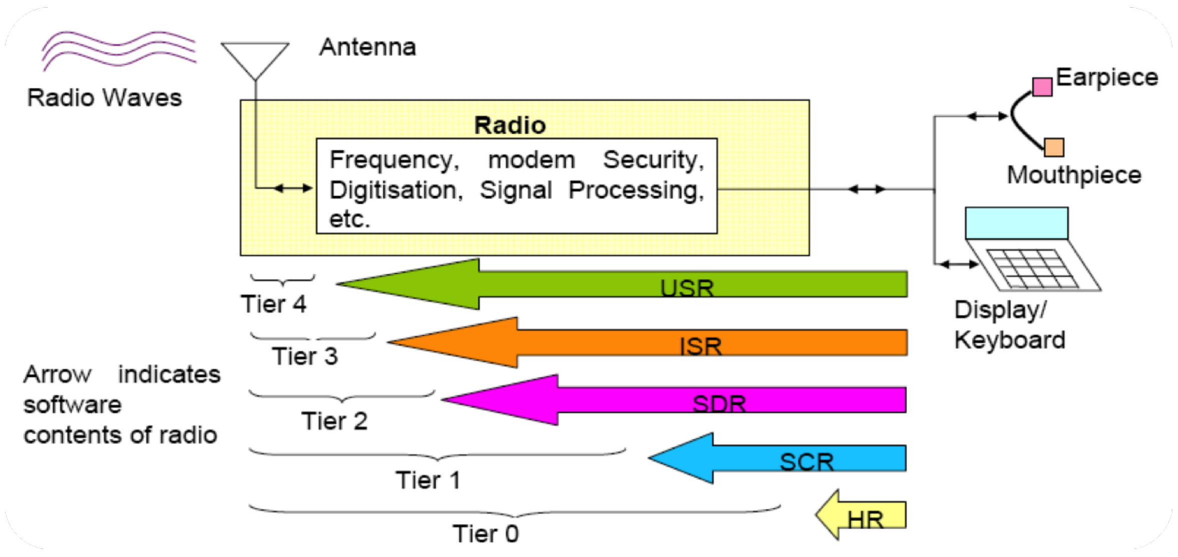

Radio Waves
Antenna
Radio
Frequency, modem Security, Digitisation, Signal Processing, etc.
Earpiece
Mouthpiece
Display/ Keyboard
Tier 4
Tier 3
Tier 2
Tier 1
Tier 0
Arrow indicates software contents of radio
USR
ISR
SDR
SCR
HR

Figure 2.1: *Classification of software radio devices*

signal, executing a down-convertion from the RF to an Intermediate Frequency (IF) and converting the signal into digital information using an ADC. Clearly it is also possible to execute transmissions providing I-Q samples to the *front-end* that it will convert into a signal using a DAC and then will transmit at the RF.
SDR devices are programmable in the sense that are able to receive commands from a PC such as working parameters like the central frequency to tune, the sampling frequency etc. thanks to the FPGA technology and DSPs that substitute ASICs.
Since this kind of units are programmable, it is pretty easy to configure or update them without the need to manage hardware modules.

2.3 SCA standard

Since the panorama of SDR solution is quite wide [3], USA decided to start a project called Join Tactical Radio System (JTRD) to standardize the implementation of SDR solutions. The aim of the project was to standardize the distributed system architecture in order to facilitate portability of components and waveforms from a physical device to another regardless of the diversity of the hardware platforms used by the different members of the group JTRD.
The result of this operation is the Software Communications Architecture (SCA) [4] which nowadays is a standard *de facto* for military applications. SCA is a distribute system ar-

chitecture that permits to deploy different components on different processing elements and to make them communicate through a middleware that, according to the specification, is CORBA. For specialized processors such as DSPs and FPGAs there exist several adapters that allow the communications with CORBA components.
The SCA objective is to ”establish an implementation-independent framework with baseline requirements for the development of software for Software Defined Radio”. It provides some constraints regarding the interfaces and structures of the software to promote reusability of code. The Core Framework is composed by four set of interfaces (Base Application Interfaces, Base Device Interfaces, Framework Control Interfaces and Framework Services Interfaces) and distinguishes between software components that require to access hardware resources (SCA devices) and software that does not require hardware resources (software only) referred as services.
Non-CORBA capable elements must use adapters that provide a translation between non-CORBA-capable components or devices and CORBA-capable Resources.

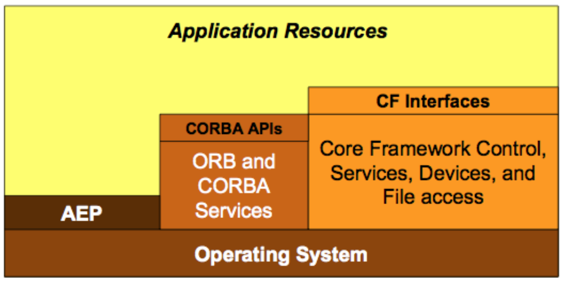


Figure 2.2: *Operating Environment and relation with the Application Environment Profile*

Application Components may need to access operating system resources but with some restrictions specified by the SCA Application Environment Profile. This environment is based on the POSIX specification which is a standard *de facto* and with some extensions is able to manage also real time applications.
The SCA specifies also the interface for some kind of components that are essentially a software abstraction of a physical hardware (Devices) and allow some particular kind of operations like capacity operations. A logical device implements one of the Base Device Interfaces. The most diffused version of the standard is actually the 2.2.2 although current version released by JTRD is 4.0.

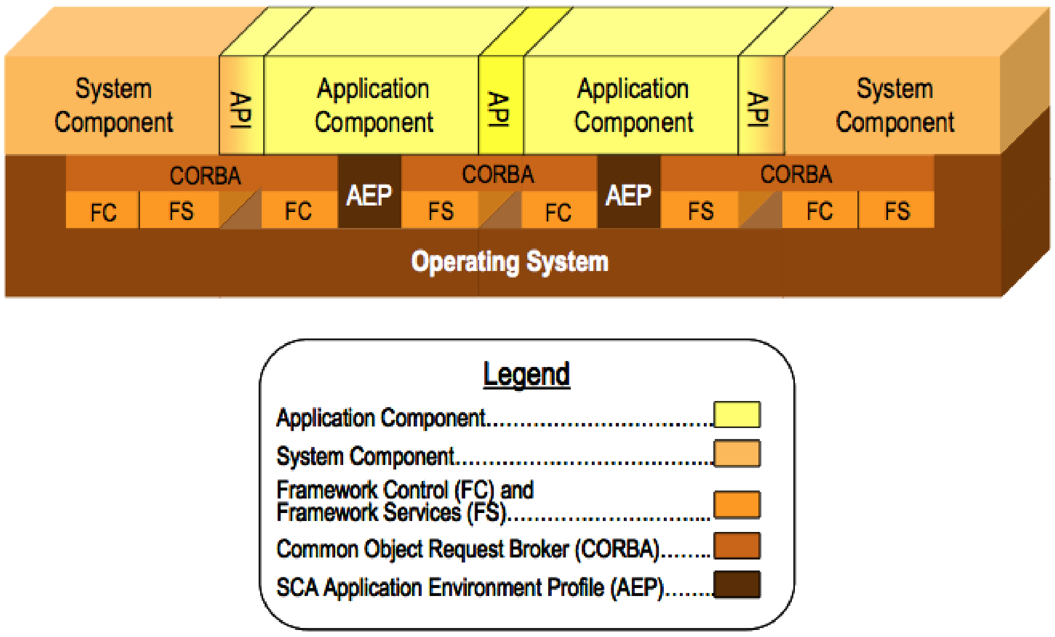

System Component
API
Application Component
API
Application Component
API
System Component
CORBA
AEP
CORBA
AEP
CORBA
FC
FS
FC
FS
FC
FS
FC
FS
Operating System
Legend
Application Component................................
System Component....................................
Framework Control (FC) and Framework Services (FS)...............................
Common Object Request Broker (CORBA)........
SCA Application Environment Profile (AEP).......

Figure 2.3: *SCA Architecture Layer*

2.3.1 CORBA

The Common Object Request Broker Architecture is a middleware for communication between remote objects by using the RPC mechanism. By introducing CORBA in SCA it is possible to realize different components handled by different processes, possibly residing on different machines with different architectures, communicate each other by exchanging messages without carying of the types implementation, the endian format, the address (by implementing the location transparency), and many other aspects of a connection that makes usually non trivial the exchanging of information in real time with different systems [5].
CORBA acts as a middleware because favor the inter-process communication without the programmer has to care about how the communication happens and the protocols involved. It is advised by SCA specification although it introduces an important overhead in the communication between components. This overhead is mainly due to the fact that CORBA exploits the TCP/IP protocol for communicating and has the necessity to make several type conversion, but, on the other hand, it has a large diffusion and maturity on almost all platforms with GPP and there exist several adapters for non fully supported platforms.
The overhead introduced by CORBA is not linear but depend strongly from the packet length [6]. By increasing the length of the packet, the effect of the middleware overhead is mitigated and performances approach to those of the pure TCP/IP [7].

Since SCA uses CORBA, the learning of this architecture could be not easy and would require several months. To obviate to this inconvenient, some tools have been developed to facilitate the learning and utilization of SCA by letting the tool handling the CORBA connection and generating the XML files, while the programmer can focus on higher level task such as the development of the algorithm inside components. CORBA allows clients to send requests to objects implementations [8] by using the Dynamic Invocation interface or an IDL Stub which are a Object Reference.

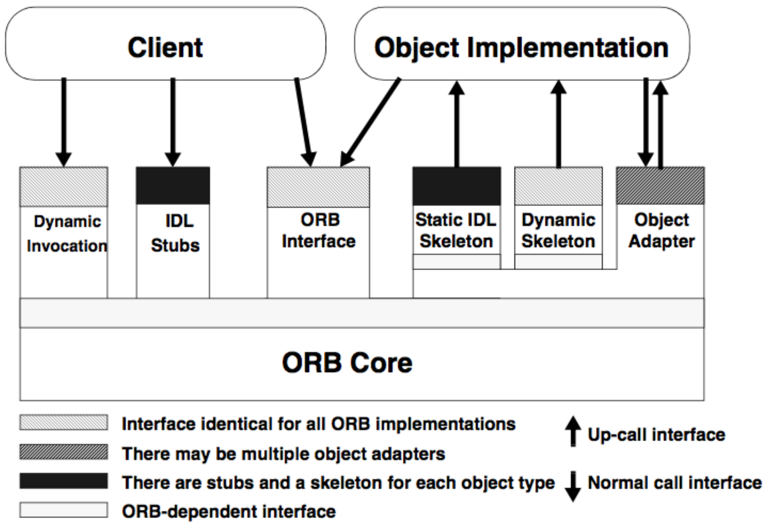


Figure 2.4: *CORBA client-object interaction*

The ORB locates the right implementation of the interface which contains the method invoked by the client. The implementations are kept int the Implementation Repository.
The client must know the type of the object on which is performing operations and the method to call.

The Object Implementation may choose which Object Adapter to use according to the service required. An Object Implementation provides the code for the objects methods, but clients have no knowledge of the implementation of the object.

In SCA CORBA provides also the Naming Service, the Log Service and the Event Service. Event channels allow the asynchronous communication between many producers and many consumers. Since this is an important source of delay, implementations of CORBA on a Network On Chip are under investigation [9].

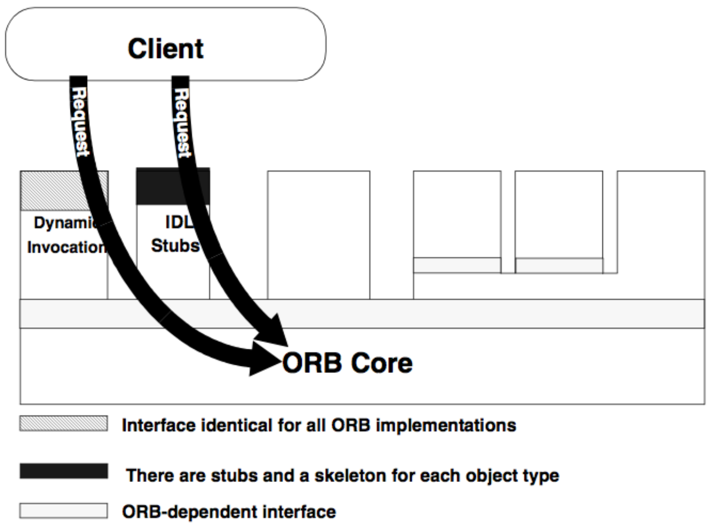

Client
Request
Request
Dynamic Invocation
IDL Stubs
ORB Core
Interface identical for all ORB implementations
There are stubs and a skeleton for each object type
ORB-dependent interface

Figure 2.5: *Client performs a method invocation on a remote object through a remote reference*

2.3.2 OSSIE

OSSIE, which has reached the release 0.8.2, is a SCA-compliant framework that stands at the basis of Redhawk. It is a project voted to research and education and includes an Integrated Development Environment based on Eclipse IDE, a waveform builder called WaveDash, a Core Framework based on SCA architecture and a collection of pre-built components.
OSSIE project has now been discontinued in favor of Redhawk which utilizes many of its libraries and tools and presents a similar Graphic User Interface. OSSIE provides a graphical debugger for testing waveforms called ALF that helps the programmer to develop its own programs by means to some tools such as FFT, I-Q diagram, etc or to connect several blocks together.
Within OSSIE IDE it is possible to create new components or utilize those pre-built in the Core Framework, and deploy them into a node, that is a set of devices capable of managing streams of data like a General Purpose Processor or physical devices. Pre-defined node allocates a General Purpose Processor which is able to run the majority of waveforms.
The IDE allows the programmer to add ports and properties to components and to automatically generate the template of code in which it is possible to insert user code and contains the signature of the most used functions.
As Redhawk, OSSIE supports several programming languages such as C++ and Python. Since it has been discontinued, bugs will no more be fixed and developers strongly recommend to adopt the successor Redhawk.

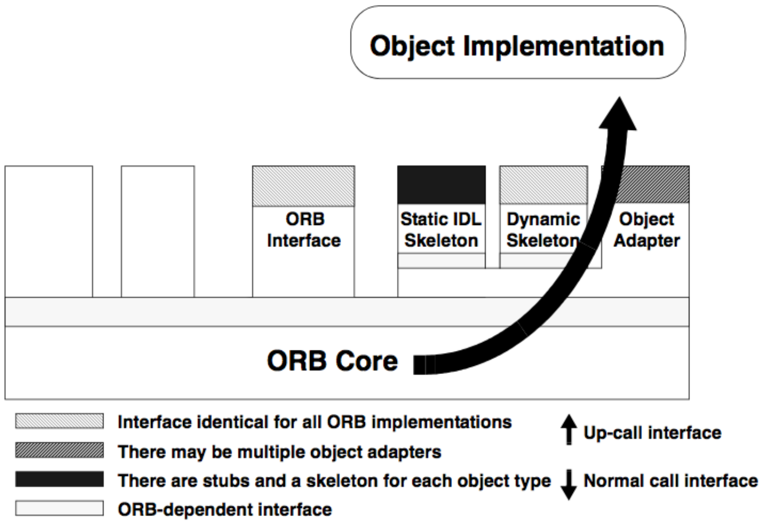


Figure 2.6: *The object receives a method invocation*

However it remains the basic development tool in SCA world, often used in less recent releases.

Differences with Redhawk

Redhaw, as OSSIE evolution, presents several innovations and improvements.
First of all, the project management has become easier thanks to the tool-generated stereotype of the code that does not overwrite previously user code. The Redhawk IDE allows the user to add ports, properties and implementations at any time of the development phase of the component, without any loss of the code.
Redhawk enhances also the waveform management. Now the programmer can use the mouse to link components, set or read their properties, move them along the waveform diagram, start and stop them singularly and connect them to a device in a way that is much more user friendly.
The developer has now the possibility to monitor in real-time many parameters of the stream of data that is flowing across component links through Stream Related Information, a structure containing information related to the data stream like the sampling period or the type of data. Furthermore a SRI listener has been added to monitor data streams in real-time and allowing to adjust component properties. SRI are also used to communicate to other components the nature of the data stream, namely if it contains scalar or complex values, removing the need of a couple of ports to handle complex streams of data. Now complex streams contains I and Q samples interleaved and can be received using just one port. The programmer must pay attention to the nature of the stream if it is required by the applica-

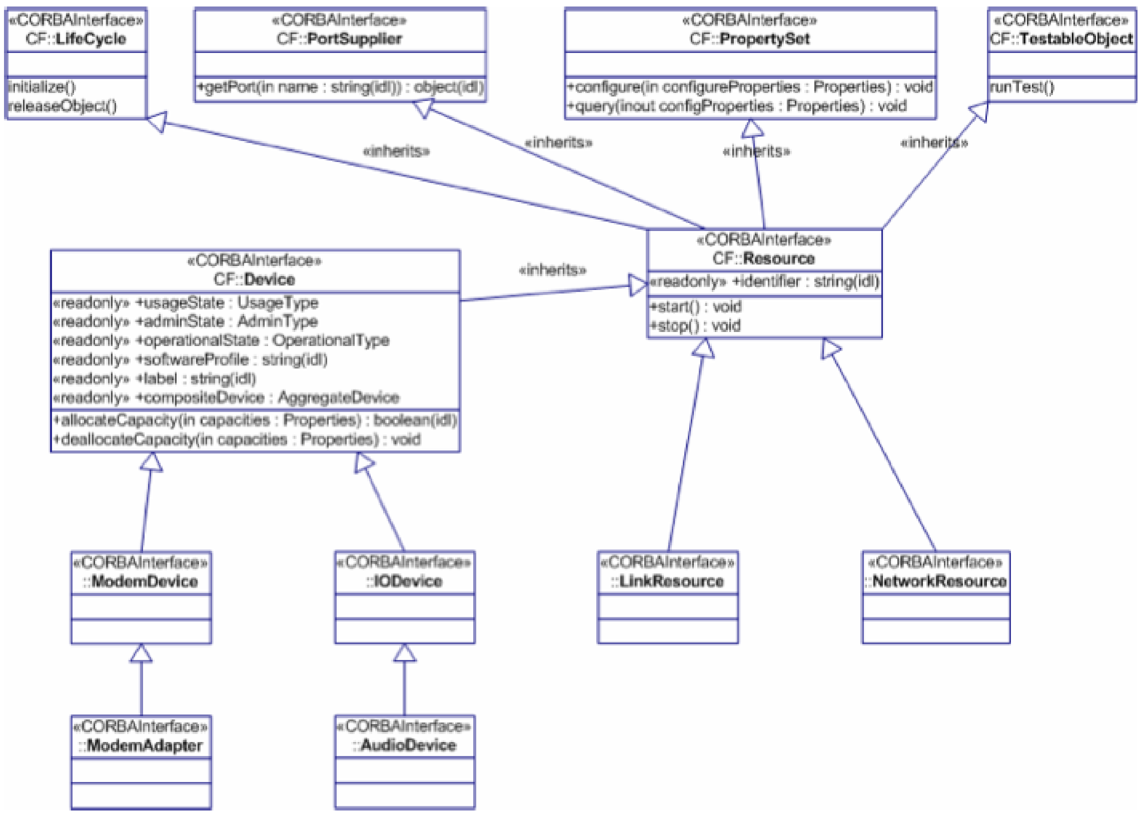

«CORBAInterface» CF::LifeCycle
initialize()
releaseObject()
«CORBAInterface» CF::PortSupplier
+getPort(in name : string(idl)) : object(idl)
«CORBAInterface» CF::PropertySet
+configure(in configureProperties : Properties) : void
+query(inout configProperties : Properties) : void
«CORBAInterface» CF::TestableObject
runTest()
«inherits»
«inherits»
«inherits»
«inherits»
«CORBAInterface» CF::Device
«readonly» +usageState : UsageType
«readonly» +adminState : AdminType
«readonly» +operationalState : OperationalType
«readonly» +softwareProfile : string(idl)
«readonly» +label : string(idl)
«readonly» +compositeDevice : AggregateDevice
+allocateCapacity(in capacities : Properties) : boolean(idl)
+deallocateCapacity(in capacities : Properties) : void
«inherits»
«CORBAInterface» CF::Resource
«readonly» +identifier : string(idl)
+start() : void
+stop() : void
«CORBAInterface» ::ModemDevice
«CORBAInterface» ::IODevice
«CORBAInterface» ::LinkResource
«CORBAInterface» ::NetworkResource
«CORBAInterface» ::ModemAdapter
«CORBAInterface» ::AudioDevice

Figure 2.7: *CORBA middleware in SCA*

tion.
It is now possible to plot graphics more easily in real-time just by clicking on ports, either for real or complex streams of data. New kinds of plots are now available like the waterfall and the oscilloscope and it is possible to monitor port statistics.

2.3.3 GNU Radio

SCA standard is not the only solution for SDR applications. Another solution coming from Open Source Community is GNU Radio, which is a toolkit that provides a development environment to develop waveforms and its components. It is based on C++ components connected through a Python interface. The performance of this framework is about 25 times faster in latency and 5 times lighter in computational load [10] wrt OSSIE, one of the most popular SCA implementations. This values are affected by the absence of a middleware and may vary according to the number of components that form the waveform.
This choice emphasizes the performance against the portability of the code that is one of the main issues in military applications since it is required to be able to move waveforms from

a device to another or to be compatible with newer devices. This could be an important limitation in case a developer wants to realize a waveform with a parallel architecture.
GNU Radio suite of components provides to the developer a wide range of built-in components such as filters, channel codes, synchronization elements, equalizer, demodulators, vocoders, decoders, and others that allow to run several waveforms immediately after the installation [11]. It presents a GUI similar to Simulink and the possibility to use built in components or to create new ones.
GNU Radio is Open Source and has a large community of active developers that realized versions of this program for all major operating systems, including MS Windows and Mac OS X.
It provides a huge suite of signal processing libraries and supports a wide range of SDR devices, especially NI Ettus platforms. Probably it is the most used development tool by researchers and radio amateurs.
GNU Radio offers also the possibility to customize existing components or to create new ones and to share code with the community.

Chapter 3

Redhawk

3.1 Redhawk Core Framework

Redhawk, as described here [12], is a Software Defined Radio framework which provides several tools for the development of software radio applications. Redhawk provides an IDE (Integrated Development Environment) which is useful for the realization of components that, combined together, can generate a waveform.
Redhawk derives from OSSIE project and is SCA compatible. Every component realized by using Redhawk should be portable to any other SCA compliant tool. It is very flexible since comes with a powerful code generator that masks the underlying middleware CORBA to the programmer and is suitable for didactic purposes due to its simplicity and the possibility to learn practically the signal processing. Some powerful tools are included within the IDE, such as an oscilloscope and a spectrum analyzer through which it is possible to monitor in real-time waveform conditions and adjust properties consequently.

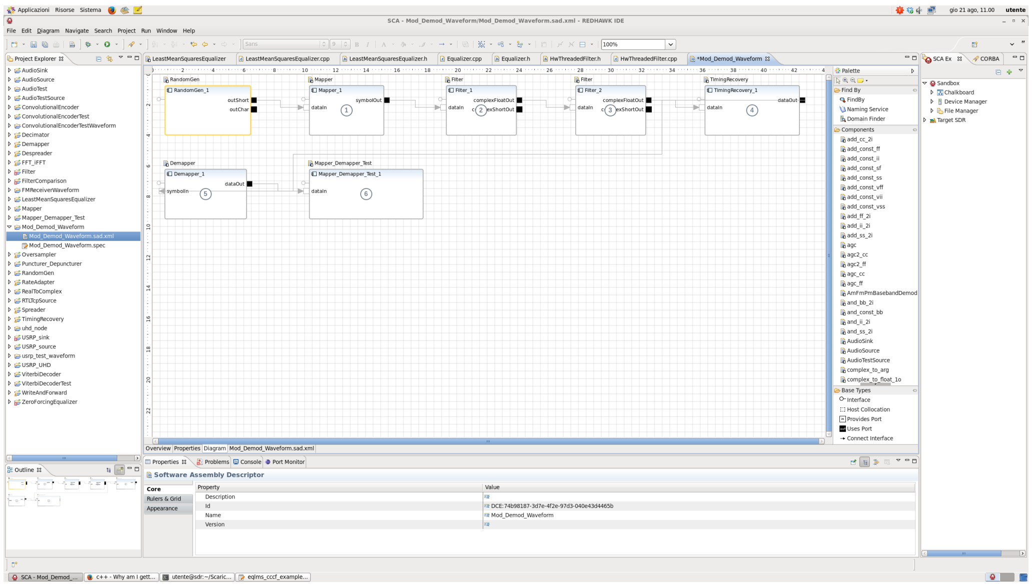
3.2 IDE

Redhawk IDE is derived directly from Eclipse [13] and it's easy to use for who is accustomed to program in C++ and Java since it presents and interface where the programmer can browse all her projects, open several file containing code, writing code with real-time syntax checking and compile. It also provides a Sandbox, a tool which offers the possibility to try components in customized waveforms very quickly, speeding up the production. The connection between components can be managed using the XML code manipulated through the text editor or through the graphical user interface of the program, by a simple drag and drop of the connection between ports. The IDE offers also the possibility to debug the code,

to start and stop nodes, to manage waveforms and a library of components.
The default layout of the IDE presents a left side panel listing all the project belonging to the user workspace. By clicking on them it is possible to inspect their content, open code files and add or remove contents. By right clicking on a project a menu appears: the user is allowed to compile code, delete all implementations and perform some other operations like cut, copy and paste.
On the bottom of the screen, a resizable panel occupy a third of the screen and its aim is to show the console where all compiling operation results are shown and where executables can print their log to communicate messages to the user. Also the Domain Manager and the Device Manager exploit this tool to show messages to the user. This panel has also some tabs that are used to show component's properties, errors and the spectrum analyzer or the oscilloscope of Redhawk.
On top of the window there's a menu bar that includes all the functionalities of the program, while on the right there is a list of all components installed in the computer and the controller of the Sandbox and the Domain Manager.
In the center there's a space for the text editor for handling open files or to host the chalkboard or the editor of waveforms.

3.3 Sandbox

Redhawk Sandbox is a powerful tool of the IDE that allows the programmer to develop and test components very rapidly without the necessity to deploy a waveform to test components.
To use the Sandbox, expand the Sandbox menu on the right panel and double click on the Chalkboard application.
The Sandbox presents a whiteboard in the central panel of the scene and a list of all available components that can be deployed. The user drags and drops components from the list to the board and they will be expanded to form a box. Each box contains inside the name of the component and has on the edge some squares that represent ports. Ports on the left are input ports and are of white color, while those on the right are output ports and are black.
ports can be connected together through a wire. In order to connect two ports together, they the must be of the same type. Simply draw a wire with drag and drop from the square that represent an output port to the square that stands for an input port.
Components can be configured with some parameters. To configure a component simply click on it and a menu will appear on the bottom panel of the application. You can set up all the properties allowed for the component that are listed in a table.



Property	Value
Description	
Id	DCE:74b98187-3d7e-4f2e-97d3-040e43d4465b
Name	Mod_Demod_Waveform
Version	

Figure 3.1: *Redhawk IDE and the Sandbox*

To start a component right click on it and then Start or, alternatively, from the chalkboard menu right click on the component icon and then Start. To start all components right click on the Chalkboard and start all components. The same procedure applies when you want to stop a component. You can selectively stop a single component or the entire waveform. When you're done you can click on the Chalkboard and then release all components.
The Sandbox tool is comfortable for fast prototyping waveforms or testing single components, but presents a general slow down of the performances. If a single component crashes all the others can continue working and it is possible to correct bugs and deploy it again.
The Sandbox can also be used and configured from terminal, and requires the knowledge of Python language.
The Sandbox offers to the user the possibility to save prototyped waveforms into the files system for further retrieve although it is not possible yet to load stored waveform in the Sandbox.
When components are running there is the possibility to monitor the current status of each one simply by clicking on it and inspecting its properties. If you right-click on a port you can select to show port statistics from the menu. Port statistics are useful to monitor the stream of data flowing through the component and its amount. For output ports it is also possible to plot the oscilloscope and the spectrum analyzer.

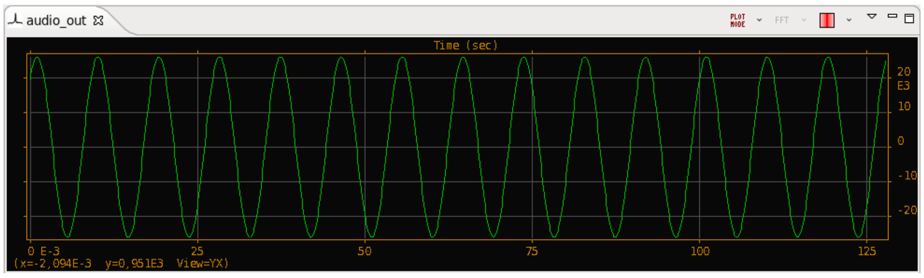


Figure 3.2: *Oscilloscope tab*

3.4 Components

Redhawk components present a similar schema which we're going to introduce.
Typically they present two kinds of ports: input ports and output ports. There exist also input/output ports but we will not talk about them. Input ports of a component are connected to output ports of another component, to form a chain.
Components can be developed using different programming languages. Redhawk supports C++, Java and Python as developing languages. Since we require a high level of performances, we chose to utilize C++, that is a compiled language and doesn't run in a virtual machine. By using C++ we have also a finer control over data, at the cost of the need to manually manage memory allocation and deallocation.
Components can be placed into the Sandbox and connected together to form a waveform, then can be configured by modifying by hand certain parameters called properties. Once all the settings have been performed, the waveform can be started. It is possible to monitor the output port of each component and plot the data in the time and frequency domain. It is also possible to adjust parameters and connections in real time, without restarting the Sandbox. Therefore this solution of using components is the easiest for developing and testing. Usually, after design, they're placed into a waveform stored in a SAD file from which they can always be retrieved.
Component life cycle is expressed explicitly in the SCA standard, and here we will summarize the basic usage and development. To create a new components a user has to click on File → new → SCA Component Project. The system will prompt several windows to the user that will have to specify at least the name of the component and its implementing language. The system will generate automatically a skeleton of the project, including SPD file. Using the graphical editor, the programmer can add ports and properties. Once the file has been completed, Eclipse generates automatically all the implementation of the component.
In the implementation file (i.e. *FilterComparison.cpp*) the user will find a brief manual of

basic programming rules:

```
/**

    Basic functionality:

        The service function is called by the serviceThread object (of type
            ProcessThread).
        This call happens immediately after the previous call if the return
            value for
        the previous call was NORMAL.
        If the return value for the previous call was NOOP, then the
            serviceThread waits
        an amount of time defined in the serviceThread's constructor.

    SRI:
        To create a StreamSRI object, use the following code:
                std::string stream_id = "testStream";
                BULKIO::StreamSRI sri = bulkio::sri::create(stream_id);

        Time:
            To create a PrecisionUTCTime object, use the following code:
                BULKIO::PrecisionUTCTime tstamp = bulkio::time::utils::now()
                    ;


    Ports:

        Data is passed to the serviceFunction through the getPacket call (
            BULKIO only).
        The dataTransfer class is a port-specific class, so each port
            implementing the
        BULKIO interface will have its own type-specific dataTransfer.

        The argument to the getPacket function is a floating point number
            that specifies
        the time to wait in seconds. A zero value is non-blocking. A
            negative value
        is blocking.  Constants have been defined for these values, bulkio::
            Const::BLOCKING and
        bulkio::Const::NON_BLOCKING.

        Each received dataTransfer is owned by serviceFunction and *MUST* be
        explicitly deallocated.
```

```
To send data using a BULKIO interface, a convenience interface has
    been added
that takes a std::vector as the data input

NOTE: If you have a BULKIO dataSDDS port, you must manually call
       "port->updateStats()" to update the port statistics when
           appropriate.

Example:
    // this example assumes that the component has two ports:
    //  A provides (input) port of type bulkio::InShortPort called
        short_in
    //  A uses (output) port of type bulkio::OutFloatPort called
        float_out
    // The mapping between the port and the class is found
    // in the component base class header file

    bulkio::InShortPort::dataTransfer *tmp = short_in->getPacket(
        bulkio::Const::BLOCKING);
    if (not tmp) { // No data is available
        return NOOP;
    }

    std::vector<float> outputData;
    outputData.resize(tmp->dataBuffer.size());
    for (unsigned int i=0; i<tmp->dataBuffer.size(); i++) {
        outputData[i] = (float)tmp->dataBuffer[i];
    }

    // NOTE: You must make at least one valid pushSRI call
    if (tmp->sriChanged) {
        float_out->pushSRI(tmp->SRI);
    }
    float_out->pushPacket(outputData, tmp->T, tmp->EOS, tmp->
        streamID);

    delete tmp; // IMPORTANT: MUST RELEASE THE RECEIVED DATA BLOCK
    return NORMAL;

If working with complex data (i.e., the "mode" on the SRI is set to
true), the std::vector passed from/to BulkIO can be typecast to/from
std::vector< std::complex<dataType> >.  For example, for short data:
```

```
            bulkio::InShortPort::dataTransfer *tmp = myInput->getPacket(
                bulkio::Const::BLOCKING);
            std::vector<std::complex<short> >* intermediate = (std::vector<
                std::complex<short> >*) &(tmp->dataBuffer);
            // do work here
            std::vector<short>* output = (std::vector<short>*) intermediate;
            myOutput->pushPacket(*output, tmp->T, tmp->EOS, tmp->streamID);

        Interactions with non-BULKIO ports are left up to the component
            developer's discretion

    Properties:

        Properties are accessed directly as member variables. For example,
            if the
        property name is "baudRate", it may be accessed within member
            functions as
        "baudRate". Unnamed properties are given a generated name of the
            form
        "prop_n", where "n" is the ordinal number of the property in the PRF
             file.
        Property types are mapped to the nearest C++ type, (e.g. "string"
            becomes
        "std::string"). All generated properties are declared in the base
            class
        (FilterComparison_base).

        Simple sequence properties are mapped to "std::vector" of the simple
             type.
        Struct properties, if used, are mapped to C++ structs defined in the
        generated file "struct_props.h". Field names are taken from the name
             in
        the properties file; if no name is given, a generated name of the
            form
        "field_n" is used, where "n" is the ordinal number of the field.

        Example:
            // This example makes use of the following Properties:
            //  - A float value called scaleValue
            //  - A boolean called scaleInput

            if (scaleInput) {
                dataOut[i] = dataIn[i] * scaleValue;
            } else {
                dataOut[i] = dataIn[i];
            }

        A callback method can be associated with a property so that the
            method is
        called each time the property value changes.  This is done by
            calling
        setPropertyChangeListener(<property name>, this, &FilterComparison_i
            ::<callback method>)
        in the constructor.

        Example:
            // This example makes use of the following Properties:
            //  - A float value called scaleValue

        //Add to FilterComparison.cpp
        FilterComparison_i::FilterComparison_i(const char *uuid, const char
            *label) :
            FilterComparison_base(uuid, label)
        {
            setPropertyChangeListener("scaleValue", this, &
                FilterComparison_i::scaleChanged);
        }

        void FilterComparison_i::scaleChanged(const std::string& id){
            std::cout << "scaleChanged scaleValue " << scaleValue << std::
                endl;
        }

        //Add to FilterComparison.h
        void scaleChanged(const std::string&);


*/
```

In these suggestions there are some important aspects of Redhwak framework. First of all Stream Related Information. These informations are collected in a structure that is passed along the waveform between component's ports. SRI have several fields, some of which are very important for signal processing; one of the most important is for sure the field `mode`, which expresses the nature of the stream of data. A value of this field equal to 0 indicates that the

stream contains real values, 1 that the stream is complex. As stated in these suggestions, complex streams are sent as an interleaved stream of real and imaginary parts, compatible with the complex type of C/C++ languages.
SRI contains also time information like the sampling period called `xdelta`, useful when we need to sample or filter data. Last useful field is the streamID, that contains a string that identifies the stream, in case we're working with multiple streams.
Another important aspect suggested in this brief guide is the typical behavior of the component. Usually it starts with a call to a port to get a packet. A packet is a structure containing a burst of data, a timestamp, a SRI structure and a flag. The data type contained in the packet must be one of the supported CORBA data types and is stored as a `std::vector` of the same type of the port. It's forbidden to connect two ports that are not of the same type.
Once data has been received, it can be processed using whatever algorithm the programmer retains necessary. After processing, the Stream Related Information must be forwarded to the adjacent component. If some elaborations have changed the values of the structure, the `sriChanged` flag must be set. Then processed data can be sent down the stream for further elaboration.
The usage of `std::vector` structure is recommended since it easiers life to the programmer while classical C/C++ pointers are more error prone. Anyway there's fully compatibility between the two types since it is possible to convert a `std::vector` into an array with a little trick:

```
std::vector<float> myVector;

// do some work with myVector

float * myArray = &myVector[0];
```

This technique is useful since all data information comes from ports that treat data as `std::vector` and are sent still as `std::vector`, but for elaboration sometimes C arrays can be more comfortable.
Auto generated code derives from the base implementation of the component which is responsible of implementing basic functionality of the life cycle, ports and properties as variables of the class. In order to access or modify a property it's sufficient to refer it through its name as a common variable. It is also possible to register the component to listen for property modifications, in case it is necessary to perform special operations when a change occurs.
Once the component has been developed, drag and drop it on to the library icon in the right panel called Target SDR to install it into the `$SDRROOT` folder.

3.5 Devices and Device Manager

According to SCA, Redhawk permits to develop a special kind of components: the devices. Devices have the same structure of common components but also have special methods to allocate and deallocate capacity. Devices may, in fact, require resources that the hosting operating system must provide through system resources or physical devices. The allocation of a device fails if it is not possible to satisfy its requirements. Redhawk basic installation comes with a predefined device, the GPP.
Devices can be grouped to form nodes. These may contain different hardware devices and a GPP.
Devices are handled by a special module of Redhawk called Device Manager. The aim of the Device Manager is to parse Node's Device Configuration Descriptor

3.6 Domain Manager

The Domain Manager is a program whose interface allows the control of the entire system Domain.
The Domain Manager keeps track of a File Manager, a set of Device Managers and Application Factories. It also manage user's waveforms. The Domain Manager is also responsible of facilitating the interaction of the user with the system parts like Devices, Services and Application capabilities.
Applications (better known as waveforms) are descripted in a SAD file which is a XML file containing a list of all components used and how they're are linked together. It may also specify how Devices are connected to components and their allocation properties.
In SAD files it is also expressed the start order of each component of the waveform.
The Domain Manager is responsible of the life cycles of the nodes. Nodes can be created by the user to host devices. The system has a default node which includes a GPP, but it is possible to create a customized one.
To create a new node:

- From the file menu click on new and then SCA Node Project. The editor will prompt you a window where it will be possible to customize the node.
- To add devices to the node simply drag and drop them from the list on the right.
- It is possible to change some properties of the devices by clicking on the property tab.

- Once all configuration have been completed, save the file and drag and drop the Node project in the Target SDR library.
- In order to launch a node, right click on the Target SDR library and then launch. A popup window will ask you which node to launch.
- Select the node you want to launch and then ok.

If everything will work correctly, the node will be listed on top right and showed as connected.
It is possible to inspected a running node to monitor some parameters like ports and SRI, plot some data with the spectrum analyzer or the oscilloscope and allocate or deallocate a front-end tuner.
To allocate a front-end tuner you need to right-click on the device of the node you want to customize and then allocate. You can specify the central frequency, the bandwidth, the sampling rate and tolerances. If parameters are corrected, the tuner will be allocated. The user can connect an allocated front-end tuner to a waveform or some Sandbox components.

3.7 Installation

Redhawk installation is not as easy as one could expect. Several problems often arises during the installation process that requires some IT skills in order to be solved. On official Redhawk website a list of the steps necessary to accomplish this task can be found. Anyway we decided to add some additional information and suggestions that could help in this preliminary phase.
It is strongly recommended to install Redhawk framework on a CentOS 6.5 machine since older versions can have some compatibility issues and other distributions of Linux, like Ubuntu or Fedora, may present some incompatibilities with some libraries that are not easy to solve for a non very skilled programmer and require time.
Other operating systems different from Linux are discouraged although many libraries are available also for Mac OS X thanks to programs like Mac Ports and Homebrew. Anyway the framework is not mature enough to allow a easy porting on other Operating Systems.
Before installing the framework, it is required to install a series of packages. These can be easily managed through the packet manager Yum. Many packages are available only after adding the Extra packages for Enterprise Linux repository to Yum. We discourage to install packages through source code because this will make the removal or upgrade harder.
Once all packages have been installed, it is time to install the framework. It comes in a *tar* archive that contains all the packages required. After the installation process, the user must

add herself to the Redhawk group for permissions.
In order to install the Eclipse IDE, Java must be installed and configured on the machine. At the end of the installation process, the user must configure CORBA omniORB. During the framework usage, CORBA often does not release the name of the Domain Manager and in order to start it again it is required to restart omniNames and omniEvents. We provide a simple script that should be launched before initiating a session and when the Domain Manager stops responding and has to be terminated:

```
#!/bin/sh

/sbin/service omniEvents stop

/sbin/service omniNames stop

rm -f /var/log/omniORB/* /var/lib/omniEvents/*

/sbin/service omniNames start

/sbin/service omniEvents start
```

3.7.1 UHD driver

In order to be able to use the UHD USRP device produced by Ettus research [14] it is necessary to install appropriate driver for the device.
Drivers can be downloaded from the official website but require some steps to accomplish all the task required. In particular there are not *rpm* packges available for CentOS distribution so it is necessary to install them using the source code. Before this operation, anyway, some dependencies must be satisfied. In particular:

1. Download the driver archive from here https://github.com/EttusResearch/UHD/tags
2. Follow these instruction to install drivers http://files.ettus.com/manual/page_build_guide.html paying particular attention to dependencies
3. Execute the command *ldconfig* at the end of the process installation.
4. Setup Udev for USB as specified here http://files.ettus.com/manual/page_transport.html#transport_usb
5. Check the correctness of the installation by connecting the USRP to the USB port and executing *uhd_find_devices* command in the terminal

6. If the command is not found, consider exporting the path of the folder in which the program is installed.

7. If required, edit the file `/etc/ld.so.conf` and add the path of the library to the file

3.7.2 RTL

RTL is a USB wideband receiver for DVB-T and FM and is a useful tool for learning SDR through direct experience. In order to use it with Redhawk it is necessary to install appropriate drivers from Osmocom website [15] and follow the installation procedure.

Figure 3.3: *RTL USB DVT-B receiver*

Once the driver has been installed, it is necessary to download also the RTLTcpSource component for Redhawk. To use the USB dongle with Redhawk

1. Plug the RTL dongle into a USB port

2. Open the terminal of your operating system and execute *rtl_tcp* program

3. Open Redhawk and launch the Sandbox

4. Place the RTLTcpSource in the Chalkboard and the required components to form a waveform.

5. Configure properties of the components adequately.

6. Right click on the Chalkboard icon and then start. The RTLTcpSource component will connect to the *rtl_tcp* program and will start to receive data.

Chapter 4

Components

In this chapter many components realized for this work will be introduced and explained in their main functions and aim. The strategy adopted on many of them is to include classes into Redhawk components in order to be able to export the code easily instead of embedding the code together with the management of the data. To facilitate this aspect and to avoid loss of performance we decided in many circumstances to instantiate/garbage objects when components are started and stopped. Many objects have a sort of status or memory, so it would be a bad choice to allow the user to change certain properties dynamically.

4.1 Convolutional encoder and Viterbi decoder

Convolutional Encoder is a component whose aim is to encode data accordingly to a schema provided by a generator. The generator is a set of masks that is used in encoding and it's provided by the user through the GUI of Redhawk as a property of the component.
As dual component, the Viterbi Decoder is able to decode a string of bits encoded by the Convolutional Encoder, provided the same parameters of the generator. The Decoder is able to execute also the depuncturing operation on its own according to the mask set by the user or work in conjunction with the Depuncturer component.

To work correctly, the strings of bits must have a minimum length that depends on the parameters used to configure the component.
Convolutional codes are used in telecommunications as error correction codes to make digital communications more reliable and have been employed in many fields, from Voyager program to many satellites. Viterbi algorithm [16] tracks the state of a stochastic process with a recursive method that is optimum [17]. The decoder is based on shift registers and a trellis structure divided in segments that form different branches along which are disposed received

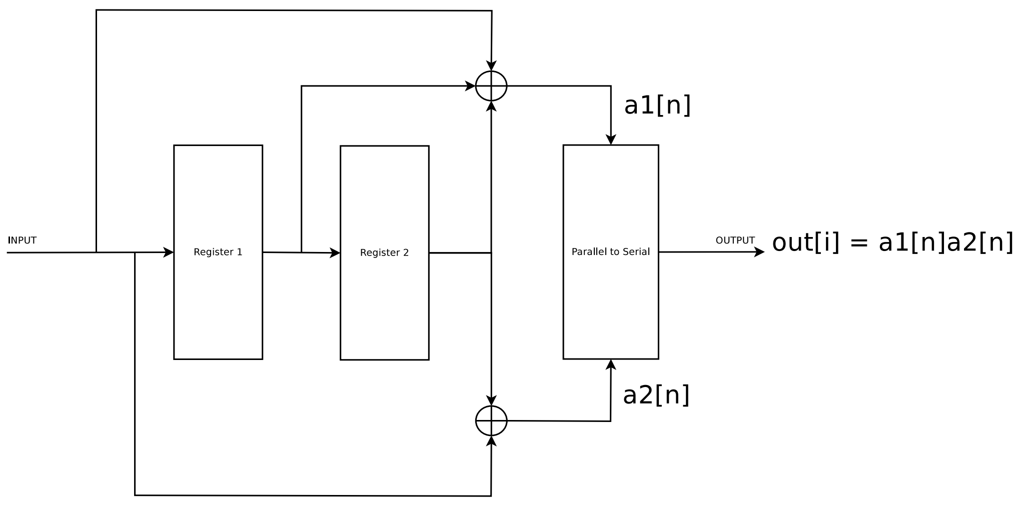


Figure 4.1: *Convolutional Encoder with generator (7, 5)*

data. A reading of the constructed path from the end to the beginning gives us the original sequence of bits that were encoded with a convolutional code.

4.2 Mapper and Demapper for QAM and PSK

One of the most common operation that must be performed by SDR applications is the mapping of strings of bits into symbols and viceversa. We realized a couple of components that are able to execute PSK and QAM mapping and demapping.
Both components include a class for handling PSK and QAM mapping and demapping. The components can be configured through some properties to execute a mapping/demapping mode or another and some parameters related to the choice such as constellation size. The component itself instantiates the class required and, upon the reception of new data, calls the correct method of the class. In case of mapping the Mapper transforms a sequence of bits in symbols. A symbol is a complex number that represents one or more bits according to the size of the constellation and mode. The symbols are then forwarded towards other components and then transmitted by the radio *front-end*.
Demapping operation, on contrary, takes a sequence of one or more symbols as input and transforms it into a sequence of bits.
The components are configured at the moment of the start and can be reconfigured only if the user stops them.
The QAM mapper is configured providing it the size of the constellation (the number of symbols); it will build a matrix, stored in memory, that will contain the IQ samples to

send. Symbols will be then associated to strings of bits. Adjacent elements of the matrix maps strings that differ for only one bit accordingly to the Grey code in order to minimize errors. The association of strings of bits to the symbols is performed by realizing first a table containing all the Grey coded values of the numbers from 0 to the size of the matrix - 1.

$$N_{16} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 7 & 6 & 5 & 4 \\ 8 & 9 & 10 & 11 \\ 15 & 14 & 13 & 12 \end{pmatrix} \tag{4.1}$$

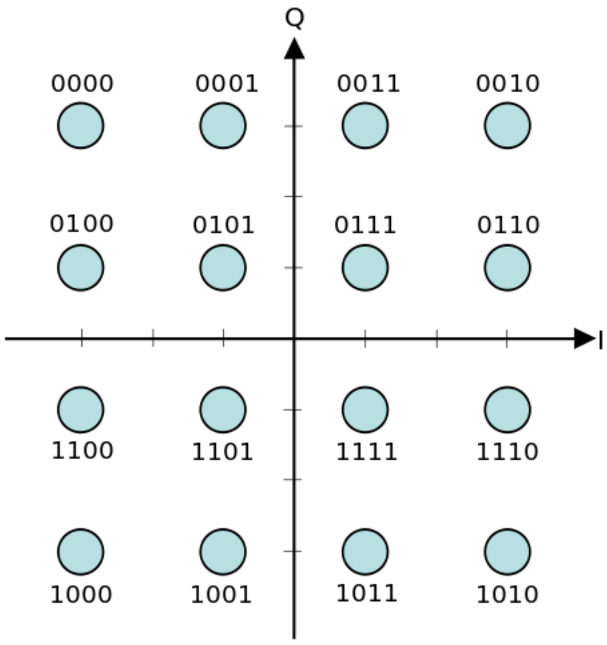


Figure 4.2: *16 QAM map*

Then each input sequence to map is compared with every element in the table.

Number	Grey Coded Value
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

When the match is found, say the i-th entry, the (complex) value that corresponds to the i-th element in the symbol table is returned. Entries are not numbered sequentially from left to right, from top to down, but rather as a snake from top left to down right, to put adjacent codes close to each other.

$$Q_{16} = \begin{pmatrix} 0000 & 0001 & 0011 & 0010 \\ 0100 & 0101 & 0111 & 0110 \\ 1100 & 1101 & 1111 & 1110 \\ 1000 & 1001 & 1011 & 1010 \end{pmatrix} \tag{4.2}$$

PSK mapping and demapping is obtained by creating a table containing all the symbols with their real and complex parts. Parallel to this table, another table is used for Grey encoding of numbers. Both tables have the same size. When you need to encode a bit string, first look for a match in the table of codes and take the index of the match. Then use the symbol at the same index in the symbol table.
Grey code is generated as follows:

```
/**
 * @brief Codify a symbol by using the Grey code
 *
 * Given a bit string, the function looks for the index
 * of the symbol that maps the sequence by using the Grey code.
 * The index is evaluated as seq ^ ( seq >> 1 )
 *
 * @param seq sequence of bit to map
 *
 * @return the mapped value in Grey code
 */
int qam::greyCode(unsigned char seq){
        return (seq ^ (seq>>1));
}

/**
 * @brief Decode a symbol by using the Grey code
 *
 * Given the index of a symbol, the function looks for the string
 * of bits that is mapped by the symbol by using the Grey code.
 * The decoding operation is performed as follow:
 * for each bit of the code d1... dn count the number of bits
 * that are set on the left of the i-th bit (so d1... dk with
 * k < i). If the count is odd, flip the bit. Repeat until all the
 * bits are analyzed.
 *
 * @param code the sequence of bit to demap
 *
 * @return the sequence of bit demapped
 */
unsigned char qam::greyDecode(unsigned char code){
        unsigned char result = 0x00;
        for (unsigned int i = 0; i < N_bit; i++){
                unsigned char tmp = code >> (i+1);
                bitset<8> set(tmp);
                bool bit = (code >> i) & 0x01;
                if (set.count() % 2 != 0)
                        result |= !bit << i;
                else
                        result |= bit << i;
        }
        return result;
}
```

4.3 Spreader and Despreader

Spread spectrum communications like UMTS have nowadays a large use in commercial applications. This system is based on the composition of two different signals, one that carries the information and one at high frequency that adds some redundancy to the signal and is orthogonal to the codes used for other signals.

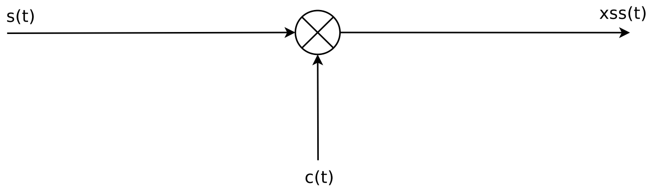


Figure 4.3: *Spreading of a signal s(t) using the signal c(t)*

This makes it possible to transmit different signal $s(t)$ (represented as a NRZ function, unitary amplitude and duration T) in the same channel without interference.

$$s(t) = \sum_i a_i \, p(t - i\,T) \tag{4.3}$$

Each spreading signal is composed by a sequence of chips at a high rate wrt the data signal.

$$c(t) = \sum_l c_l \, q(t - l\,T_c) \tag{4.4}$$

The resulting signal is the multiplication of the two signals. The ratio between the rate of the two signals is the spreading factor M.

$$M = \frac{R_b}{R_c} \tag{4.5}$$

$$x_{ss}(t) = s(t) \cdot c(t) \tag{4.6}$$

By defining $\gamma_l = a_{\lfloor \frac{l}{M} \rfloor} \cdot c_l$

$$x_{ss}(t) = \sum_i a_i \left[\sum_l c_l \, q(t - l\,T_c) \right] p(t - i\,T) = \sum_l \gamma_l \, q(t - l\,T_c) \tag{4.7}$$

Since we're going to treat bits that can be assimilated to a signal modulated as OOK, the operation of spreading is performed by xoring data bits with the Hadamard codes $a_i \oplus c_l$ The Hadamard codes are taken from a Hadamard table that is built in memory by the component. The size of the Hadamard table depends on the configuration of the respective property of the component and is connected to the *spreading factor* that the user want to realize.

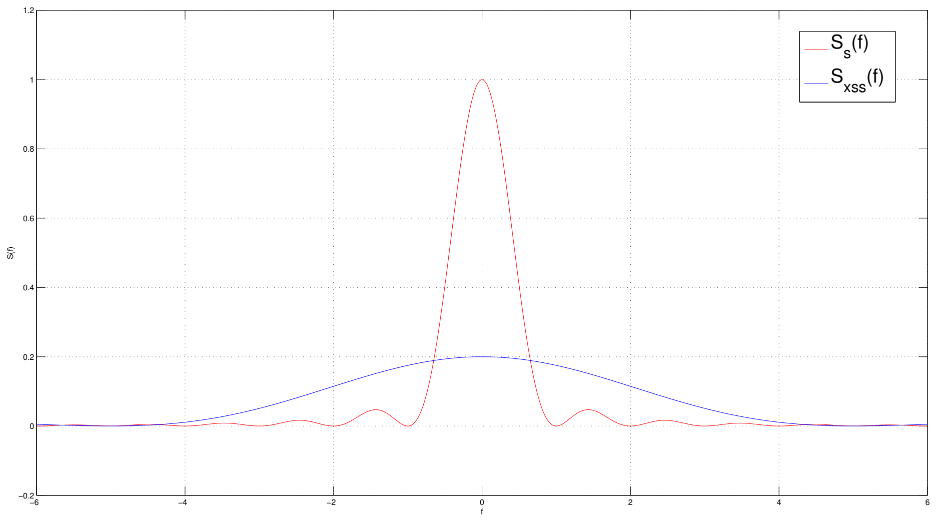


Figure 4.4: *Spectrum of the signal (red) and the spreaded signal (blue) with spreading factor equal to 5*

Hadamard Codes Hadamard Codes, also known as Walsh Codes or Walsh-Hadamard Codes, are used to distinguish communications channels. Codewords are used to identify users inside cells in CDMA communications and have the property to be orthogonal. Unless two users adopt the same codeword, the transmission of a user will appear like noise to the other users in the cell. Codewords are composed by a binary alphabet. The code matrix can be built using a generator matrix that is replicated according to a schema. To generate the Hadamard table, starting from the smallest table of 1x1 elements, it's sufficient to append copies of the matrix on the right and on the bottom while the copy on the bottom right must be changed in sign.

$$H_1 = (1) \tag{4.8}$$

$$H_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} H_1 & H_1 \\ H_1 & -H_1 \end{pmatrix} \tag{4.9}$$

$$H_3 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} = \begin{pmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{pmatrix} \tag{4.10}$$

In general:

$$H_i = \begin{pmatrix} H_{i-1} & H_{i-1} \\ H_{i-1} & -H_{i-1} \end{pmatrix} \tag{4.11}$$

4.4 Puncturer and Depuncturer

Puncturer_Depuncturer is a component that is used in conjunction with encoding and decoding operations for convolutional codes to add and remove bits from a string according to a mask provided by the user. The Puncturer and the Depuncturer utilize the mask to remove the specified bits from a sequence or to add zeroes in certain positions to enlarge a sequence.

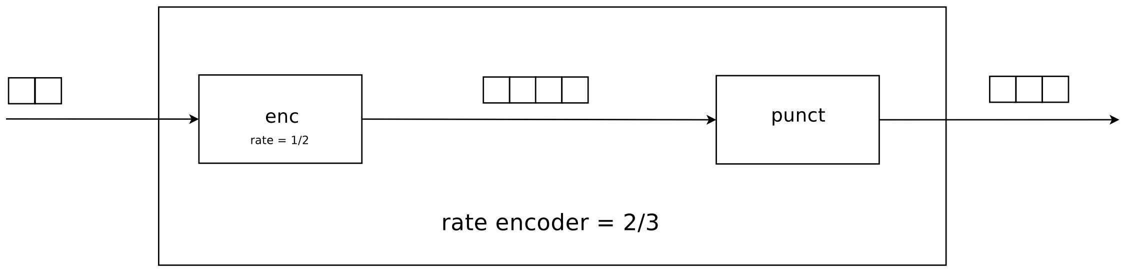


Figure 4.5: *Encoder with rate 2/3*

This can be used to get fractional rates starting from a rate of 1/2 and to reduce the number of bits that is necessary to send over the channel, with a consequent increment of the error probability. This components can be built as an embedded function of the encoder or decoder for convolutional codes but we decided to make also a specific component edition to be more flexible when building waveforms.

4.5 FFT/iFFT

Transformations from time domain to frequency domain and viceversa are common operations in SDR waveforms. It can be necessary for filtering or some analysis and requires a lot of computational power. Luckily this issue has been faced several times in the past and an existing good solution can be represented by FFTW library, which is a performant library that is able to perform FFT and inverse FFT. This library allows the programmer to execute the FFT with an arbitrary resolution and to exploit the parallelism to increase the performances of the system. Although it's possible to manage all the operations manually by writing a component that performs all the transform computations, we chose to integrate the FFTW solution as is in this component without any modification because we repute it very optimized.

To check the goodness of the component, we can send a signal to the FFT and then apply the inverse transform to check if the result is the same as the input. Obviously it is also possible to check the result of the FFT block by looking at the spectrum analyzer offered by Redhawk IDE or comparing the signal observed through the oscilloscope before and after FFT/iFFT.
Since the FFTW library is robust and well made, specifying the compiler to use multiple threads to compute the transform will help to make computations faster.

4.6 Converters

This class of component is similar to that provided by the Core Framework of Redhawk but is able to handle only short and float values. It can convert input data from real to complex and can transform float data into short and viceversa.
There exists some components of this family such as ShortToFloat, Float_to_Short and Real_to_Complex, which perform the cast specified by the name. When a cast from float to short is required, the conversion takes into account the representation interval available for the short type. In fact float input data (real or complex) are first scaled and then multiplied by the maximum integer representable in a short value ($2^{15} - 1 = 32767$).
Scaling factor is evaluated as the maximum value that the float input can assume. User may specify in a property the maximum value to use for scaling in order to maintain a dynamic of values that occupies all the possible values offered by the short type.

4.7 Rate Adapter

Rate Adapter is a component that acts as a buffer to adapt the rate of symbols from a component to another that have different speeds. In particular the adapter varies the length of the streams of symbols that are sent from the output port. The size can be enlarged or reduced according to the necessity. It has an internal memory buffer to store temporary samples before packetizing them. The buffer is implemented as a queue due to its frequent size change and implements the FIFO policy. If the buffer contains enough elements, a block of size specified by the user is output from the port, otherwise data is retained until it reaches the volume required. The front of the queue is continually shifted, while new samples are enqueued at the end. This component may affect a lot the performance of the waveform since it can increase or decrease rates and anyway introduces a delay due to the processing of enqueueing/dequeueing.

Its application could be necessary when some components require a input rate that is constant or has a specified number of symbols, while the source of data is not constant and thus it's necessary to apply a buffer that maintain the rate constant at the desired value. This version of the buffer works on symbols which carry a different number of bits according to the mapping used, but its bit version implementation is straight forward.

4.8 Random Generator

The Random Generator is a component whose aim is to produce a pseudo random sequence of bits of arbitrary length. In order to generate the sequence of bits, the generator requires to know the seed to adopt in the generation process and the size of the strings of bits generated. To not overwhelm following components the Random Generator waits a certain amount of time (actually 1 s) before sending another string.
If the rate of generation of bits exceed the waiting time, the rate obtained will be less than expected. The generated bits are represented as short values since CORBA does not provide a boolean type to map C++ `bool` type.

4.9 Write and Forward

This component has been realized in order to monitor and record the traffic that flows through a branch of the waveform. In fact it is able to read and store data that pass through it in a location specified by the user and then push data towards the following component. It is completely transparent to data but since it requires to perform some disk IO operations it could reduce performance of high data rate. Anyway with modern SSD technology this issue should be alleviated and shouldn't be noticed by the end user. In case performances were considered essential and there were no margin for delays, we recommend to not insert this component in the middle of critical paths of data since may slow down the effective data rate.

4.10 Scrambler

Scrambler component, also called *randomizer*, is used to invert the normal ordering of bits in order to exploit some code property and for cryptography. The variation may be used also when a sequence of bits is repeated frequently. The scrambler is structured as a series of registers that are used to memorize a pattern, initialized with a seed. At every iteration bits are shifted between register components. According to a mask, bits are also summed each

other. The user may choose to use all the registers or only a part of them, starting from the lowest ones (least significant bits). If used, a scrambler component must be placed either in the transmitter and in the receiver, both initialized with the same seed and with the same mask.
Altering the normal sequence of bit by scrambling helps to reduce errors on streams of data and makes the initial sequence of bit appearing as random noise to an observer along the channel.

4.10.1 Scrambler for symbols

We realized a version of the scrambler oriented to PSK symbols. This version is a slight modification of the scrambler for bits, since the internal structure remains the same. This component is able to map the internal status of the scrambler into a symbol, according to the size of the PSK constellation specified as a property.
The obtained symbol, which is simply the mapping of the first bits of the scrambler in the constellation, is then multiplied with the input symbol of the component. The waveform programmer must be careful when using the component not to use symbols that map strings of bits longer than the internal register size.

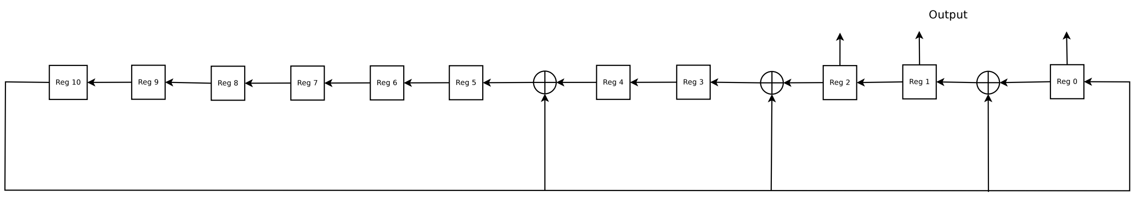


Figure 4.6: *Scrambler with 12 registers and generator 1000001010011.*

Since register elements depend on each other, this component does not beneficiate significantly from a parallel architecture.

4.11 Block Interleaver

According to some military standards, we realized a component that is able to execute the interleaving of the input in blocks.
Interleaving operation is performed on streams of data to change the natural ordering of bits. This helps to reduce the impact of noise on transmissions. Data is usually transferred in bursts and it may happen that one or more of them are corrupted due to noise in the channel. Some error-correcting algorithms are usually applied but they have some limits on

the maximum number of flipped bits they're able to detect and correct. Repositioning of bits creates a more uniform distribution of errors and therefore it's easier to recover a damaged burst of data.
Input bit strings are picked up in sequence and are put in a matrix according to a schema provided by the user which spreads the input in all the rows of the matrix. Bits are then mixed again inside the matrix, typically copying the loaded matrix into another, inverting the previous ordering. Once data is received by the receiver, bits are treated in inverse order and repositioned correctly in the matrix. Then bits are extracted from the matrix following the same schema used to insert them.
The matrix in which bits are inserted can be represented as an array made by a sequence of rows. Input bits are inserted every k position (load factor), filling remaining gaps with zeroes. Then the matrix is copied into the second matrix. This time bits are fetched every n (fetch factor) positions from the input array and are inserted sequentially into the output matrix.

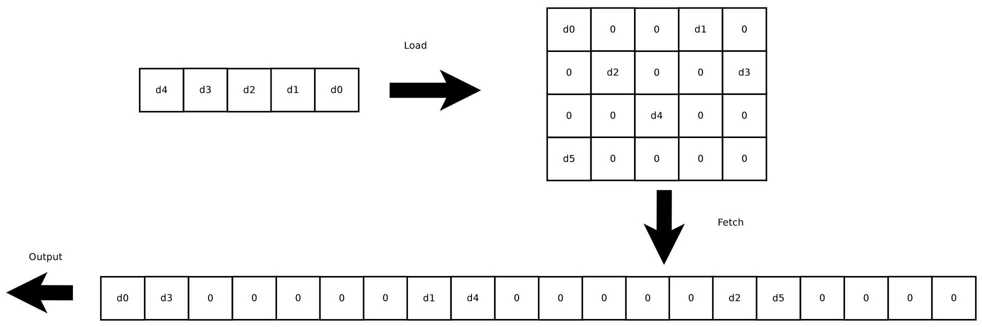


Figure 4.7: *Example of block interleaver with load factor of 3 and fetch of 9.*

In order to deinterleave the sequence, the same procedure is applied to the sequence, but with $k' = n$ and $n' = k$.

4.12 Convolutional Interleaver

The Convolutional Interleaver is a kind of interleaver that makes use of a series of shift registers of different length to memorize bits coming from the input data stream. The register number is chosen by the user and must be the same for the Interleaver and the Deinterleaver. Their length is different and starts from the unitary registers and increase according to a step provided by the user as a property in the Interleaver, while in the Deinterleaver, on the contrary, the register length starts from longest down to the unitary register with decrements equal to the the increment used in the Interleaver. Bits are inserted in the first element of each shift register of the Interleaver. The schema used to insert bits in the registers may vary according to user specifications, but the default is sequentially. Then registers are shifted of a position and last bits of each register are output. Then a new series of bits can be inserted in the Interleaver. The process continues until all the input bits haven't been inserted and all the bits in the longest register have been output. Zeroes are inserted when no more input bits are available and we need to shift the registers. Registers are initialized with zeroes.

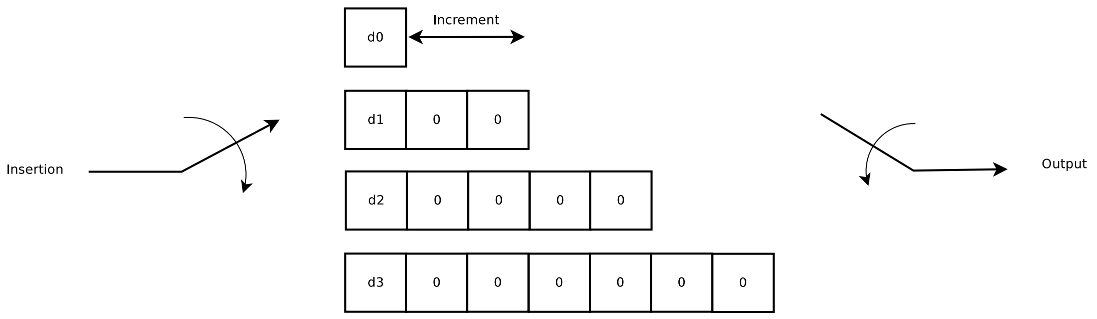


Figure 4.8: *Convolutional interleaver with 4 shift registers and increments of 2.*

The Deinterleaver acts in a dual way. Received bits are inserted in the registers in their arrival order and then elements are shifted of a position. Output is picked up according to the same schema used for inserting bits in the Interleaver (default sequentially), starting from the longest register. The process terminates once all bits have been inserted into registers, shifted and extracted. The output will have a set of zeroes preceding the sent data. Since the schema used is known a priori, the number of zeroes added is known and it is possible to extract just the desired data from the output.

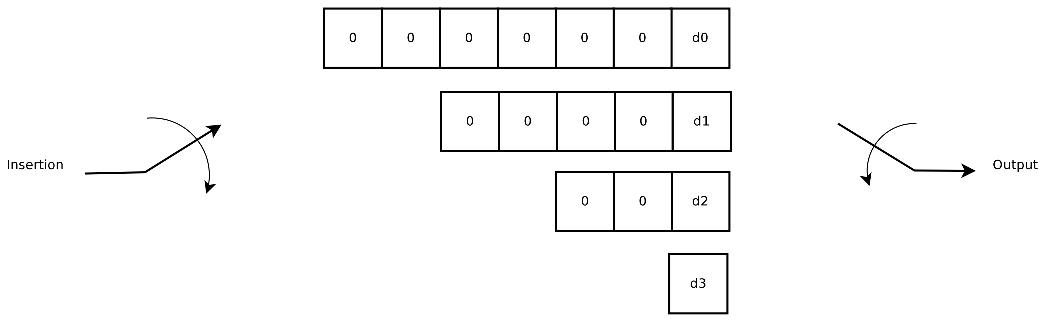


Figure 4.9: *Convolutional deinterleaver with 4 shift registers and decrements of 2.*

The number of zeroes is given by the length of the longest register times the number of the registers, minus the first column that is equal to the number of registers.

4.13 Delay Estimator

The Delay Estimator is a component that is able, given a burst of symbols, to look for a special sequence of symbols that precedes each packet, called preamble, and, once detected, to remove it and return the following packet of data.
It is used for synchronization purposes and offers a more evolved algorithm than the Timing Recovery to detect the presence of a packet in a burst of data.
The recognition of the preamble happens when the correlation between received samples s and the preamble samples p (known a priori) is over a user specified threshold. The threshold default value is the autocorrelation of preamble samples but can be increased or decreased according to the user choice.

$$R_x = \sum_i^n p_i p_i^* \tag{4.12}$$

This value is just theoretical and does not consider the influence of the additive noise introduced by the physical channel. Computed value may differ from this value, so it is necessary to introduce a tolerance.

$$corr = \sum_i^n s_i p_i^* \tag{4.13}$$

The threshold can be raised or lowered by the user, using a multiplicative real coefficient (in the interval [0, 1]) in a component property.

$$threshold = coeff \cdot R_x \tag{4.14}$$

The Delay Estimator is also able to estimate the phase delay of the signal and correct it. That is possible by exploiting the correlation previously evaluated in delay estimation.

$$\frac{\sum_i^n s_i p_i^*}{n} = e^{j\hat{\phi}} \tag{4.15}$$

The burst of data is distorted by the channel, which introduces a phase delay and an attenuation. In order to decode the correct symbol of the constellation, the phase delay must be estimated and canceled. Supposing that the channel doesn't change quickly during the transmission, the same phase delay is applied to all data contained in a burst. On receiver size, once the packet has been detected using the preamble, it is necessary to apply a correction to the phase of symbols. In order to realize how much the the constellation must be rotated,

we make an estimation of the angle, using the known symbols contained in the preamble. In particular it's possible to compute the correlation between pairs of symbols of the received preamble and known ones, and estimate the phase delay ϕ. To cancel the noise random effect, instead of using a single couple of symbols, we average the product along all the preamble size. At the end, we need to rotate all the burst of an angle equal to the opposite of the delay angle. The resulting angle is just the conjugate of the delay phase. The rotation is obtained by multiplying each sample times a complex number with unitary module and phase equal to the desired rotation angle.
In order to match the correct preamble sequence of symbols, since phase is a identically uniform distributed random variable, we use a pre-estimation of the phase. In fact a preamble may be present in a burst but not recognized if the rotation angle is wide (the correlation between known preamble and received data is too low). To obviate to this inconvenient, we divided a round angle into N parts of size small enough that does not influence the computation of the correlation. Then we multiply each angle times the input sequence and we look for the presence of a preamble. If there's no match, we multiply by the angle plus an increment until all the possible angles have been tried and no match higher than the threshold has been found with the preamble sequence.

4.13.1 Timing Recovery

Timing Recovery component can be used together with RX Filter in order to obtain symbol synchronization and cut exceeding symbols introduced by filtering operation. When the signal arrives at the receiver it is not obvious which is the best instant to sample the signal. A bad sampling strategy leads to incorrect decoding of symbols. The aim of this component is to minimize the error (or maximize power) by finding a temporal offset τ that allows the user to choose the correct instant for sampling the signal. The Timing Recovery component in particular aims to cut away from the burst of data the tails introduced by filtering operation through convolution between the FIR and the symbols. This component is a simplified version of the DelayEstimator that just removes queues introduced from filtering operations.

4.14 Packetizer

The packetizer is a component whose aim is to add a user specified preamble to a packet of data for synchronization purposes. The preamble is used by the receiver to obtain the best synchronization between symbols. The user may provide a file containing the preamble to use to the component in a property, specifying the filename and path.

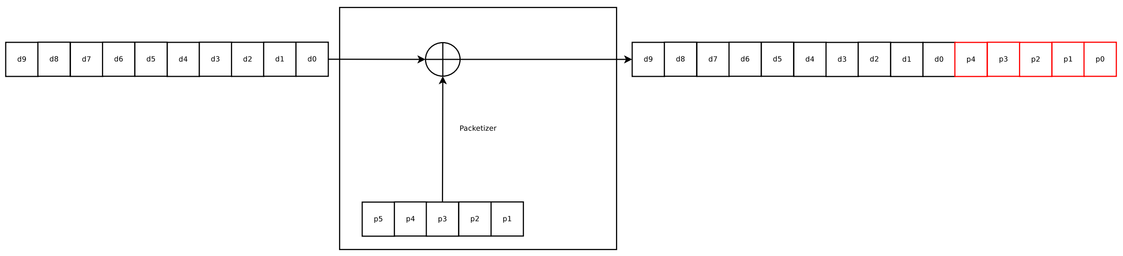


Figure 4.10: *The packetizer adds a preamble to a data packet*

4.15 Noisy Channel

Communication doesn't occur through ideal channels but instead these can present several forms of disturb such as noise, distortion, multi paths...
This component simulates a channel with an additive gaussian noise. The output of the component is the sum of the input symbols plus noise generated according to user specified parameters (μ for mean value and σ for standard deviation).

$$x'[n] = x[n] + w[n] \tag{4.16}$$

where

$$w[n] = \mu + \sigma \cdot rndn() \tag{4.17}$$

and *rndn()* is a function that returns a real number between 0 and 1 following normal distribution.

4.16 Delay Generator

The aim of this component is to simulate a delay in time and phase of a signal. The delay in time is realized by simply adding some zeroes before the burst containing data. The number of zeroes can be specified by the user or random. On receiver side is then necessary to synchronize the received data using a Delay Estimator.
The phase shift is realized by multiplying the data with a complex number whose module is unitary and whose phase is specified by the user. The user must be aware that is necessary to compensate the phase shift on receiver side. The phase must be expressed in degrees.

4.17 ConstMultiplier

The multiplier is a very simple component whose aim is to multiply whatever input times a constant value provided by the user. The component accepts real and complex sequences and multiplies each value times a constant. If the constant zero is provided, the output will be a sequence of zeroes. The maximum value for the constant is limited by the float type range. There could be a loss of precision for high values of the constant.

4.18 Wide Band FM Demodulator

The wide band FM demodulator is responsible for FM demodulation of the received signal. FM demodulation is accomplished by computing the derivative of the phase of the samples. The modulated signal is:

$$s(t) = Acos\Big(2\pi f_0 t + 2\pi K_F \int_{-\infty}^{t} a(\tau)\,\mathrm{d}\tau + \phi\Big) \tag{4.18}$$

where K_F is the maximum frequency deviation that for FM broadcast radio is 75 Khz. To be more performant, we avoid the use of operations and functions offered by the complex type built-in the C++ libraries. Instead we manipulated directly the real and imaginary parts exploiting the fact that complex numbers are represented in memory as couple of adjacent values (i.e. two `float` values). Usage of complex type's functions is strongly discouraged since slows down the execution and may affect seriously audio performances as happened during this work.
In order to compute the derivative of the argument of the signal, we consider couple of symbols $s1$ and $s2$ from the input stream and multiply each couple between them, conjugating the oldest of the two samples to compute the difference of angles.

$$d_{2,1} = (e^{j\theta_1})^* \cdot e^{j\theta_2} = e^{j(\theta_2-\theta_1)} \tag{4.19}$$

The angle is then extracted from the complex number using the `atan2` function for a fast calculation. It would be also possible to create a table in memory with pre-computed values of angles and tangents to improve further the performance.

$$\theta_{2,1} = \theta_2 - \theta_1 = arg(d_{2,1}) = arg(e^{j(\theta_2-\theta_1)}) \tag{4.20}$$

4.19 De-emphasis filter

De-emphasis is a necessary operation to perform after the reception of a channel whose data have been compressed using a pre-emphasis filter. Pre-emphasis is usually applied to FM transmissions to reduce the effect of the noise at high frequencies by increasing the magnitude of the signal at the higher frequencies.
De-emphasis filtering is performed by using a IIR filter. The filter has been designed using Liquid DSP library [18] and embedded in the component for performance matters.

$$H(s) = \frac{b_0 + b_1 s}{1 + a_1 s} \tag{4.21}$$

4.20 Hamming filter

The Hamming filter is a low pass filter used to limit the signal bandwidth based on a sliding window applied to a *sinc* function.
The taps of the Hamming filter are designed taking into account the input and output rate, while the filter is implemented using the Liquid DSP library [18].
In a waveform voted to receive commercial FM transmissions it can substitute the TuneFilterDecimate component (belonging to the Core Framework) if it is used in conjunction with a Decimator.

$$h(n) = h_d(n) w(n) \tag{4.22}$$

where $h_d(n)$ is given by the *sinc* function and is always the same while the window is

$$w[n] = \alpha - \beta \, cos\Big(\frac{2\pi i}{n-1}\Big) \tag{4.23}$$

α and β are respectively 0.54 and 0.46

$$w[n] = 0.54 - 0.46 \, cos\Big(\frac{2\pi i}{n-1}\Big) \tag{4.24}$$

and i ranges from $-\frac{n}{2}$ to $+\frac{n}{2}$ and n is the window size.

$$h_d[n] = \sum_{-\frac{N}{2}}^{\frac{N}{2}} \frac{sin(\pi n)}{\pi n} \tag{4.25}$$

$$h[n] = \sum_{-\frac{N}{2}}^{\frac{N}{2}} \frac{sin(\pi n)}{\pi n} \cdot w[n] \tag{4.26}$$

Chapter 5

Parallelism

5.1 What is parallelism

According to the Moore law the performances and the number of transistors in a chip double every 18 months. So, in SDR terms this means that it would be possible to execute more complex algorithms and waveforms of double computational requirements every 18 months. This is not truly true since performance increment can be obtained not only by increasing clock frequency of processors, but also exploiting other capabilities of modern CPUs. Recently Intel has moved towards parallel solutions to increase the throughput of its processors due to the fact that processors with very high clock frequencies have a high density power with consequently problems in energy consumption and heat dissipation. An example of this necessity was the failure of the Tejas project [19].
Parallel solutions can be used to increase the throughput of the processor and speed up the computation, but require more attention in developing applications that can fully exploit this characteristics of the chip. There are several architectures available on the market nowadays: multicore processors seem to be the most common and cheap solutions although maybe the level of parallelism can't go further than 4-8 processes at a time. Optionally some Intel processors present the Hyper Threading technology [20] which seems to be a very promising innovation since it allows the execution of multiple hardware thread simultaneously. A third solution would be represented by vectorial machines (SIMD machines) that allow the parallel elaboration of massive streams of data, like it happens in modern GPUs. The most common technologies on the market that use GPUs as GPPs are CUDA proposed by Nvidia and OpenCL [21] introduced by Apple (the latter is also able to manage parallelism on General Purpose Processors and not only GPUs). In this document we will focus on MIMD solutions that can be managed and implemented easily by modern Operating Systems.

So with parallel architectures it is possible to execute different instructions on different data at the same time.

5.2 How to reach parallelism

The parallelism management is quite OS dependent so, to handle this issue, we will make use of the POSIX threads that are managed by many Operating Systems and are very easy to learn and use. Now it is important to make a distinction because we can identify two different kinds of parallelism: data parallelism and instruction parallelism. To ease the understanding of this difference, we can imagine that each component of a waveform is handled by a process (they could be many, but for simplicity let's say one) and all components are running together. We can imagine that each component is performing its task on its data, and that data residing in one component is different from data being processed by another component at the same time. So this could be a good example of instruction parallelism since every process elaborates different data with different algorithms and different speeds. As can be noticed, waveforms are intrinsically parallel and can beneficiate of multicore processes also without any optimization. Now let's suppose that a component must perform the same instruction on every stream of data it receives. Since often data is moved in blocks, components process blocks of data. Usually this kind of elaboration is obtained in code by making use of loop instructions. So, repeatedly the same operation is applied to every element of the block. It is possible to gain some time if the operation is of a kind that doesn't require the result of the previous elaboration nor it is required by the following one (this can be thought as data independence) and thus it is not necessary to enforce any synchronization (we can start data elaboration on a block before all previous data have been processed). In a multicore environment, when we're in presence of data independence, it is possible to process data in parallel by splitting the stream of data in chunks and assigning different chunks to different elaboration units. This is known as data parallelism.

5.3 Filter

Filtering is one of the most common and important operations in a waveform and can be performed either in the time domain or in the frequency domain. While in the frequency domain the filtering operation configure itself as a mere product between the filter characteristic *H(f)* and the signal spectrum *X(f)*, in the time domain it is a convolution operation. The convolution is a long chain of products and sums and requires a lot of computational

capacity. On the other hand, in order to perform a product in the domain of the frequency, we need to apply the FFT transform to the input data and then apply the inverse transform on the data after filtering.
As can be imagined the convolution operation can beneficiate of the parallelism of data since the filter is substantially memoryless.

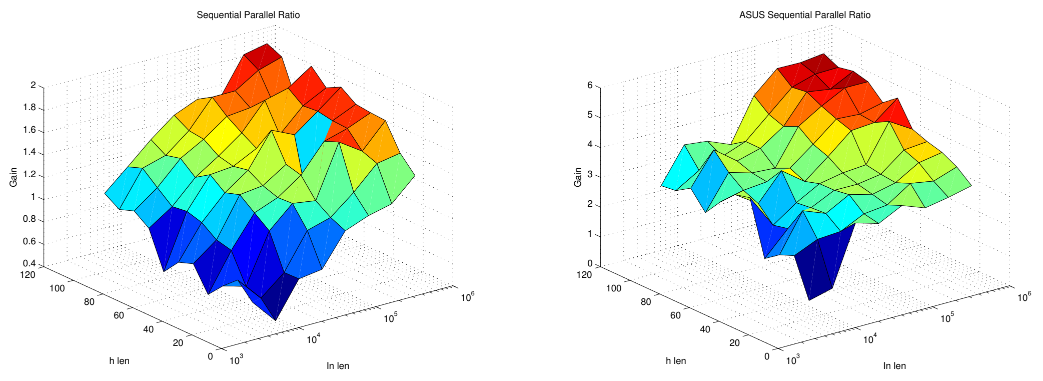
$$y(t) = x(t) * h(t) = \int_{-\infty}^{+\infty} x(\tau) \cdot h(t-\tau)\,\mathrm{d}\tau \tag{5.1}$$

$$Y(f) = X(f) \cdot H(f) \tag{5.2}$$

Obviously in every implemented filter you will work by using samples, since it is not possible to represent continuous variables such as time, and thus the filtering operation will be a finite sum of products between samples and the Finite Impulse Response FIR of the filter.

$$y[n] = x[n] * h[n] = \sum_k x[k] \cdot h[n-k] \tag{5.3}$$

This simplifies a lot the computation and makes it realizable on the GPP. The first thing to do is to realize how many threads would be efficient to use. We performed several tests by fixing the input length and the FIR length and varying the number of threads used in the filtering operation. We ended up that the best number of threads to use in the filtering operation is equal to the number of hardware thread that the CPU is able to handle. This means that for a *Intel I7* Quad Core with 8 hardware threads, the best number of threads to use is 8.



(a) *Intel Core 2 Duo* **(b)** *Intel i7*

Figure 5.1: *Performance of multithreaded filtering on two different Intel processors*

As can be seen in the figure 5.1 we have large benefits from a multithreaded structure of the filter when the the number of threads approaches to the maxim number of threads physically manageable by the CPU and the size of the data block and the FIR length are sufficiently high. For a low number of threads the gain is almost linear.

$$gain = t_{seq}/t_{par} \tag{5.4}$$

We recall that the Finite Impulse Response length is $2 * k * m + 1$ values, where k is the number of samples per symbol and m is the filter delay in symbols. The output length after the convolution will be (assuming the input length l_{in}) $l_{in} + l_{FIR} - 1 = l_{in} + 2 * k * m$ symbols.

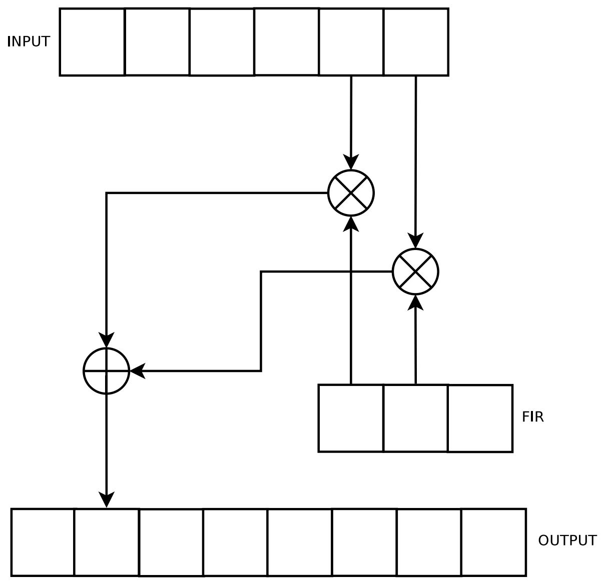


Figure 5.2: *Calculation of convolution. The convolution is evaluated as a sum of products. Since output elements are independent from each other, they can be evaluated in parallel*

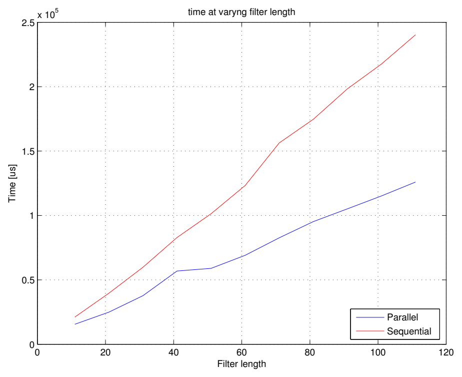
The filtering operation is structured as follows: first the component receives some data from to process, usually in block, then allocates enough space to contain the output of the filtering operation. After this, data is passed to the filter object which unlock its thread for calculating the convolution. Since the operations that involve the creation and managing of the POSIX threads are expensive, we chose to start the threads once for all and to eventually block them on semaphores waiting for new inputs. As soon as new data is available for processing, the output is divided into chunks and each chunk is assigned to a thread according to an id (which is a progressive number to identify the thread inside the class). When each thread has been assigned its work, every thread is unlocked by simply executing a V operation on the semaphore on which each thread is blocked. Threads will automatically block again on

the semaphore when the work will be completed.
Every thread computes a part of the output by using the same input data of the other threads. Since every thread writes in a distinct part of the memory not shared with the others, there's no need of protecting the memory from concurrent accesses. Differently, all the threads share the same input data and FIR, but they just need to read them, and again there is no need of enforcing a synchronization. Once all threads have completed their elaboration, data can be forwarded to the following component.

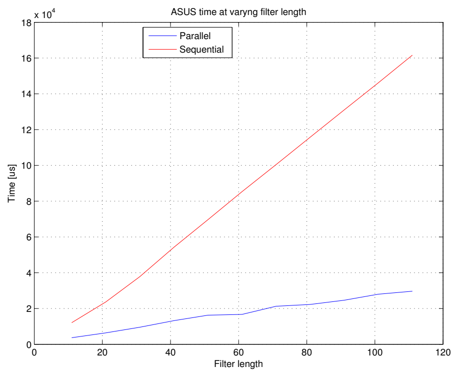
5.4 Performance comparison in filtering

In order to show the gain that is possible to reach using multithreading in the filtering operation, we made a series of tests on a Intel Core 2 Duo 2.4 GHz and on a Intel i7-2670QM 2.20 GHz.
The tests were focused on showing the response time of the filter by varying the FIR length of the filter component and the length of the input provided to the filter. Results proved that we can end up in important improvements if parallelism is considered for heavy computations tasks. For example fixing the length of the input and varying the FIR length. The gain obtained is quite impressive wrt serial elaboration of data. By fixing input and FIR size and varying the number of threads used in the computation we discover that we have a gain until the number of threads is less than or equal to the number of hardware threads the processor is able to handle. When the number of threads used to calculate the convolution is larger than the number of manageable hardware threads, threads are obviously enqueued and the scheduler of the operating system will assign them to a processing unit when available. In this way we do not take advantage from a parallel solution since threads do not work together and the handling of threads reduces the time available for computation. Exceeding in the number of threads will lead to a degradation in performance, that can become worse than a sequential solution.

Although the performance increment is strongly dependent from the number of cores of the processor, the gain is not linear as one could expect. Some hypothesis can be formulated regarding this aspect: first of all Redhawk environment is not running on a dedicate hardware but instead on a general purpose operating system which has its own processes that must be scheduled and compete with the filter for the possess of the processors; the policy adopted by the OS could influence this aspect since could affect the way that cores are assigned to each process. Caches, page fault etc. can in some way influence the activity of the filter. In case of swapping performance drops down dramatically.

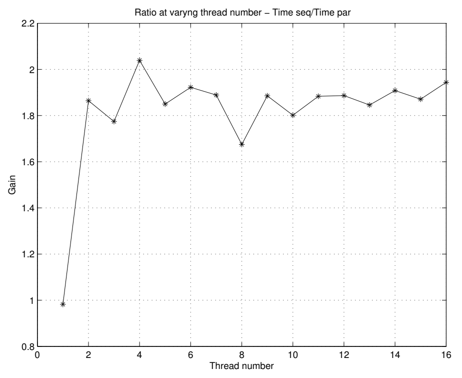


(a) *Intel Core 2 Duo*

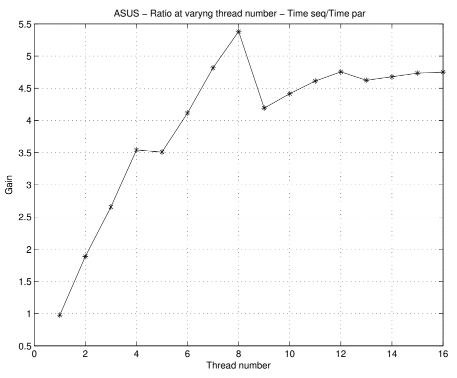


(b) *Intel i7*

Figure 5.3: *Time of elaboration at varying of the FIR length*



(a) *Intel Core 2 Duo*

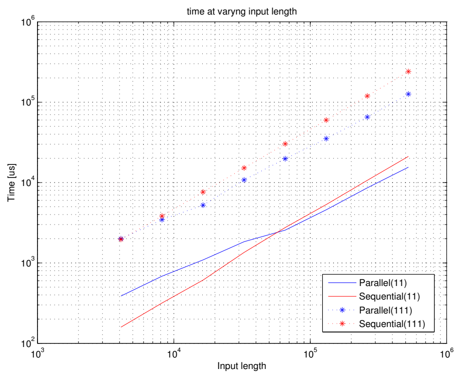


(b) *Intel i7*

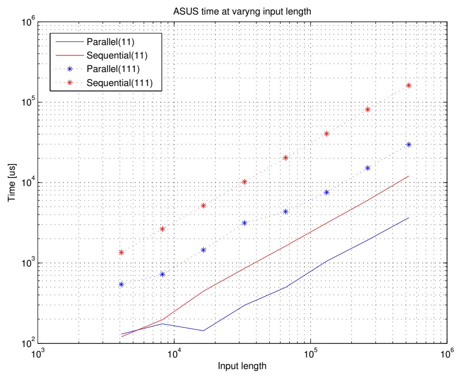
Figure 5.4: *Ratio between time of elaboration of a sequential filter and a parallel one*

Another intuition regards the fact that rarely all available cores are assigned the filter threads together. So usually the computation is longer than expected because other processes are executing at the same time.

As can be observed, parallelism has also a cost, that in case of short input loads can make the classical serial elaboration more convenient. The cost of this must be searched in thread creation, semaphores, context switches and mutexes. In particular POSIX threads have an important impact on the performance when the workload is not high. This aspect must be accompanied with a consideration: is it worthwhile to implement a parallel solution? Are usually loads high enough to justify the usage of several threads? Since we're in presence of an important speed up for the most cases, and since usually block sizes are quite large due to the high rate of the waveform, yes.



(a) *Intel Core 2 Duo*



(b) *Intel i7*

Figure 5.5: *Comparison between filter varying the input length*

5.5 Oversampler and Decimator

The Oversampler and the Decimator are two components that present very simple algorithms that can be parallelized in order to speed up the computation. As in the filter implementation, the Decimator and the Oversampler are managed by a dedicated class that instantiates some threads for the elaboration. All the threads are created once for all and are then stopped on a semaphore. In both cases, once the component receives a packet containing data, it allocates space for the output. Then all the data is passed to the class that must perform the operation. In the case of decimation, threads simply run along the input data and take 1 per k samples. It's a task of the programmer to check that the input length is multiple of the decimation factor, otherwise some samples are lost. To adapt the input size to the decimation factor it is possible to use the RateAdapter component. Input data is split in chunks and each of them is assigned to a thread which will discard exceeding samples. Remaining samples will be written in the output and then will be sent away from the component.

Oversampler is a bit more complicated since it offers two different kind of oversampling: zero padding and sample and hold. With zero padding configuration enabled the component will add zeros between symbols. In particular, assuming k as the oversampling factor, it will add $k - 1$ zeroes after each symbol. Regarding the sample and hold option, the oversampler will replicate each symbol of the input sequence k times. In both cases the output length will be $l_{in} * k$ symbols.

5.6 Delay Estimator with parallel architecture

The purpose of this component has already been descripted in 4.13. Since it is a computational intensive component, it can beneficiate from a parallel architecture when looking for the maximum correlation.

In fact the parallel architecture of the component is based on threads and computes in parallel several values of the autocorrelation. There are some synchronization points since the return value must be unique. In order to avoid race conditions we made use of mutexes.

The implementation of parallelism is similar to that of the filter (see par. 5.3) with threads that are never destroyed to increase performance. In fact, to reduce the latency, threads are simply stopped on a private semaphore. When their use is required, they can be unlocked by executing a V operation on the semaphore on which they're blocked.

The preamble identified by the component must be unique, so threads communicate each other using a shared variable. To avoid race condition such variable must be accessed in mutual exclusion using mutexes.

Chapter 6

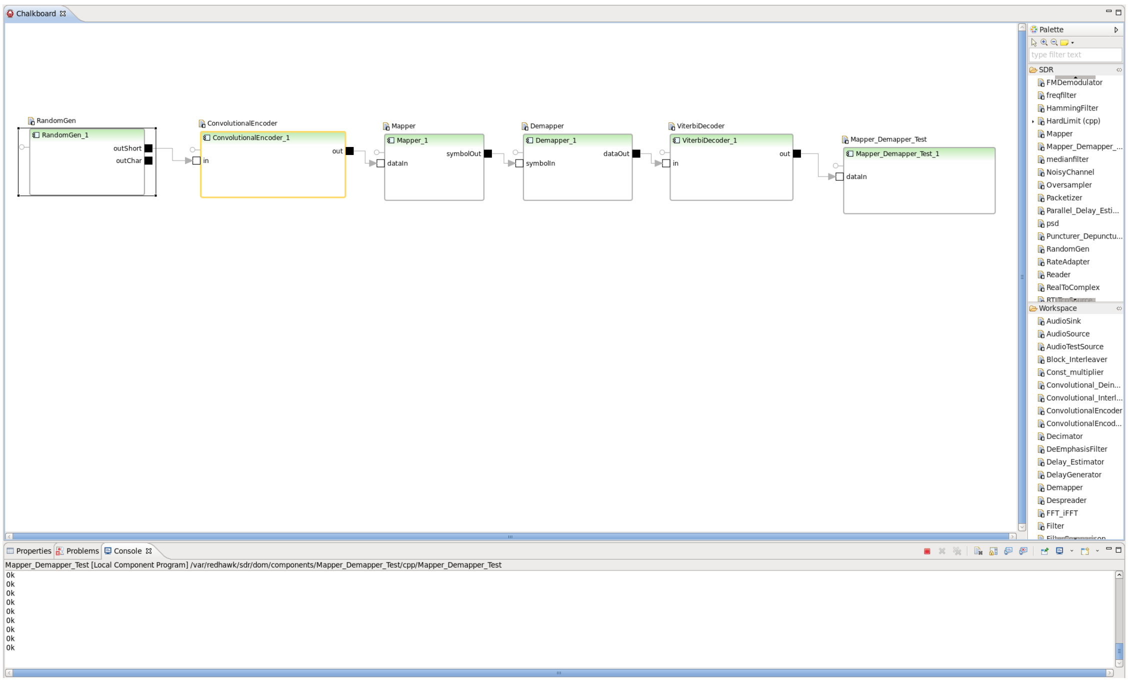
Waveforms

Waveforms are a sort of application specifically written for software radio purposes. In Redawk these are constituted by a serie of components represented as boxes tied together trough wires and ports.
A programmer may create its own collection of waveforms and switch between them quickly without the necessity to recreate them from the ground every time she needs of them.
Time required to switch between waveforms can be critical and depends on the application. Redhawk offers poor performances in terms of time required for switching between waveforms because often it is necessary to stop and start again the Domain Manager and if it is necessary to switch in a short slice of time we strongly suggest to put all your components in a unique waveform and add some logic to handle different standards at the same time.
It is possible to create prototypes of waveforms by exploiting the Sandbox environment, or create the own SAD files using the editor offered by Redhawk and based on Eclipse. Although the Sandbox is a comfortable programming environment, due to its simplicity, it sometimes lacks of performance, especially if the hosting OS is running in a virtual machine, so we recommend to use SAD files once the structure of the application has been decided. It is possible also to prototype a waveform in the Sandbox and then store it into the disk in a SAD file. Storing the waveforms offers also the possibility to save settings for each component for a faster recovery when needed.
Anyway it is not possible to load a SAD file in the Sandobox so, once a prototype of a waveform is stored, it can be retrieved from the disk only as a complete waveform application.
In this chapter we will present some waveforms we adopted for testing developing.

6.1 How to build a waveform

There are two ways to crate a waveform using the Eclipse IDE. The first one, as introduced in previous paragraphs, consists in prototyping it using the Chalkboard of the Sandbox and then to store it in a SAD file once ready.
The second way to build a waveform is by using the editor. Let's create a demo waveform using the editor:

1. Click on the File menu and then new SCA Waveform Project
2. The editor will open in the center of the screen
3. Drag the RandomGen component in the Diagram panel.
4. Drag the ConvolutionalEncoder in the Diagram panel
5. Connect the two components by clicking on the output port of type short of the RandomGen and dragging a line up to the input port of the other component.
6. Add also a Mapper, a Demapper and a ViterbiDecoder.
7. Place a Mapper_Demapper_Test at the end of the waveform.
8. Connect all the components together as showed in figure 6.1
9. Set the rate property of the RandomGen to 64 samples/s and do the same for the test at the end of the waveform
10. Save the waveform and install it by dragging the project on the Target SDR library
11. Start the Domain Manager clicking on the Target SDR icon with the right button of the mouse and then launch. Start the default node with a GPP
12. Right click on the running Domain and select the option "Launch Waveform..."
13. From the list, select the created waveform and then "Finish"
14. Your demo waveform will be loaded and if you click on the start button you'll see that a log will be printed in the console. If parameters are correct, the test will show the correctness of the coding/decoding operations.



```
Mapper_Demapper_Test [Local Component Program] /var/redhawk/sdr/dom/components/Mapper_Demapper_Test/cpp/Mapper_Demapper_Test
Ok
Ok
Ok
Ok
Ok
Ok
Ok
Ok
Ok
```

Figure 6.1: *Example of waveform realized using the Sandbox*

6.2 A real example

The RTL USB dongle for DVB-T is a broadcast receiver for digital TV transmissions and it is based on the Realtek RTL2832U chip. It is possible to use it for SDR purposes [15] and a Redhawk component has been realized to receive data from the USB dongle.

This component can be exploited to build a waveform to receive broadcast FM transmissions and RDS data stream. Here we will show briefly how to listen FM transmissions and how to monitor the spectrum to detect useful signals. Lets start by creating a new waveform as specified above, and in the Diagram pane let's add the RTLTcpSource component. Connect it to a TuneFilterDecimate component provided by the Core Framework or use a filter connected to a decimator. The RTLTcpSource component must be tuned on a FM radio channel by specifying the central frequency in its property and the sampling frequency (a sampling frequency of 1024-1200 ksample/s would be ok). The filter can decimate by a factor of 2-4 and must have a bandwidth of 200 kHz. Then a FM demodulator must be connected to the decimator. We have developed our own FMDemodulator component but for this example we will use the pre-build AmFmPmBasebandDemodulator which acts in the same way. So put this component in the Diagram by dragging it from the palette on the right and set the frequency deviation to 75 kHz. After the demodulation we can add a de-emphasis filter

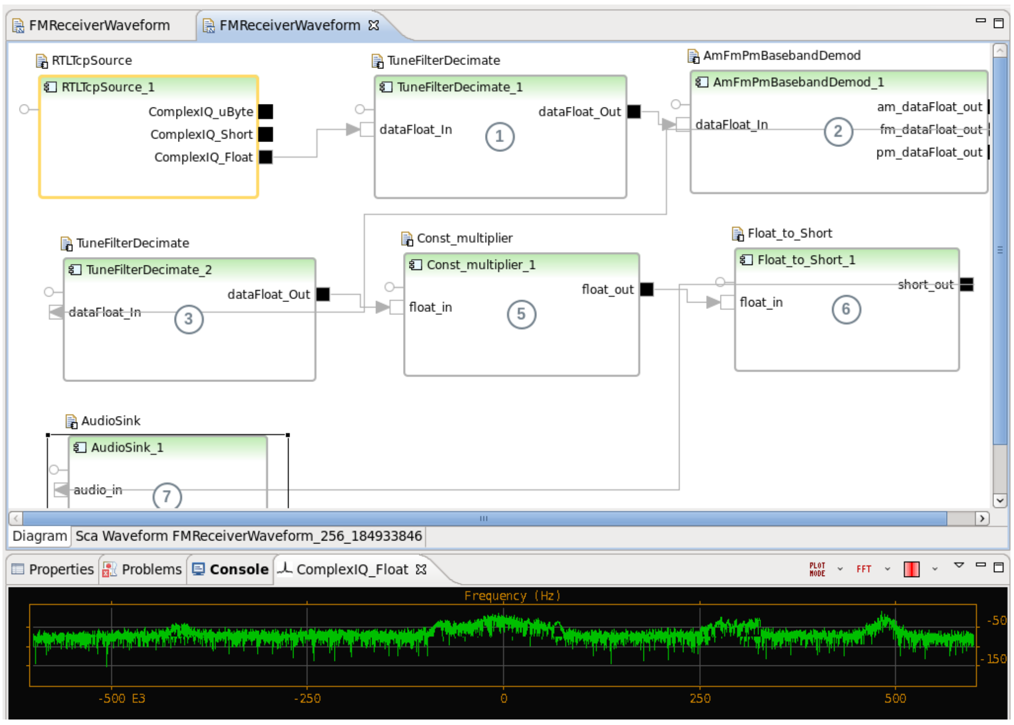


Figure 6.2: *RTL based FM receiver with spectrum analyzer*

and a decimator or another TuneFilterDecimate and set the output rate to 48 kHz and the bandwidth to 15 kHz to get the mono channel of the FM transmission. Then add a multiplier to increment the magnitude of the output and connect then the AudioSink component. It is possible to download the component from the Axios Engineering repository [22].
It is possible to exploit the tools offered by Redhawk to explore the FM signal and compare it to the expected spectrum.

As can be seen the signal of the FM station contains several audio channels plus a channel for RDS data.

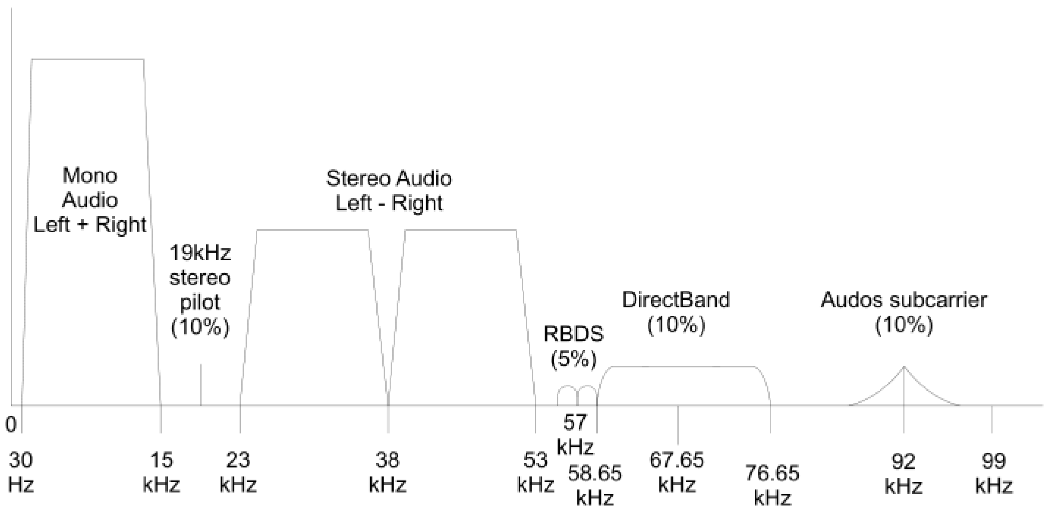


Figure 6.3: *Spectrum of the FM baseband signal*

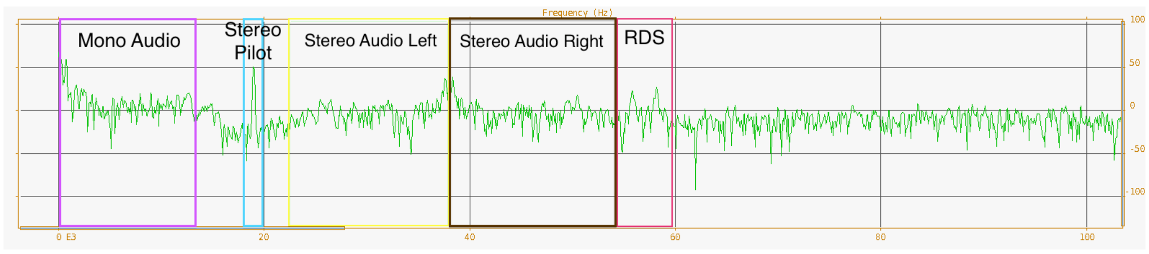


Figure 6.4: *Spectrum seen by the Redhawk tool*

Chapter 7

Conclusions

Approaching the end of this work we summarize briefly the results: we realized a series of components for a SCA compliant framework (Redhawk) that can be used for SDR and signal processing. It has been shown that, where possible, a waveform can benefit from the usage of parallel solution in the implementation of the code with a speed up from 1.9 to 5.5 times wrt a sequential implementation, according to the number of available hardware threads it is possible to use. We remark that this kind of result is strongly dependent from the OE and the nature of the algorithm. Algorithms with strong independency of output data can be easily parallelized, such as happen in the filter case.
For real-time processing of massive streams of data it would be desirable to adopt arrays of processors or SIMD machines. This requires powerful and expensive hardware although recently GPUs are emerged as good candidates for processing large amount of data with specific characteristics. In particular GPUs are cheap compared to specific ad hoc solutions and are installed on every personal computer. They're able to work on large amount of data on which it's necessary to perform heavy computations especially with floating point values. Introducing these massive parallel elaboration with the right algorithms in SDR could push the diffusion of this technology towards the overcoming of ASIC solutions in radio applications. It has been noticed also that the IDE environment reduce the learning curve since it is not required a deep knowledge of CORBA middleware. The adherence to the SCA standard makes it easier to port waveforms from a device to another without the need of any adaptation and this enlarges device life and reduces costs.
We also remark that the imposition to use CORBA as a middleware slows down computations significantly in favor of portability. CORBA is an essential component of the SCA but often it has been criticized due to its lack of performance. Nowadays new communication solution

between processors on which are executing threads are under investigation like Network on Chip (NoC) technology [9].

7.1 Future perspective

SDR will probably see a very significant use in communications due to its high flexibility although different scenarios will profile: in military and civil application SCA will dominate the scene due to its large development and standardization but in commercial devices proprietary solutions could compete with open source software. GNU Radio is a valid alternative for those that are looking for homemade solutions and want to learn programming with the radio but does not have important characteristics that appear to be crucial like multithreading native support, since hardware and in particular processors are moving towards parallel architectures.
Probably we will observe the introduction of SDR technology also in smartphones since actual signal processors can support very few standards and often it's necessary to purchase a different version the device for the world areas that adopt different standards.
Other important application fields in which SDR will show its strength are massive MIMO applications, which will make use of arrays of antennas to sense the spectrum at high frequencies on multiple channel and to handle multiple transmissions at the same time, and cognitive radio, since we expect that devices will be smarter and able to recognize those channels that are occupied and those that are free and exploit the band with less interference automatically and have a finer power control to be able to transmit with as less power as possible but with good SNR. Cognitive radio will also allow the sharing of the spectrum with other users and a good spectrum management to reach the desired QoS.
Other important aspects regard performances. Up to now we need performant algorithms and powerful GPPs to process huge amount of data in real-time. The introduction of parallel architectures, as it has been shown with this work, will improve the computation time required by components. It could be interesting to apply SIMD machines to the elaboration of data with the GPGPU technique, by using technologies like CUDA by Nvidia [23] or OpenCL [21].
We aim also to consolidate the components database by improving algorithms and adding new ones to our library to share with the community.

7.2 Acknowledgements

I wish to thank my family for their support and my darling Serena for the comprehension and help.
A special thanks is reserved to Carmine for his support in laboratory and patience.

Appendix A

Filter code

Listing A.1: *code/HwThreadedFilter.h*

```
/*
 * HwThreadedFilter.h
 *
 *  Created on: 23/lug/2014
 *      Author: Fabio Del Vigna
 */

#ifndef HWTHREADEDFILTER_H_
#define HWTHREADEDFILTER_H_
#include <cstdlib>
#include <cmath>
#include <complex>
#include <liquid/liquid.h>
#include <pthread.h>
#include <iostream>
#include <semaphore.h>
#include <fstream>
#include "Resampler.h"

using namespace std;

class HwThreadedFilter;

struct t_params{
        unsigned long id;
        HwThreadedFilter * filter;
```

```
};

class HwThreadedFilter {
        Resampler * resampler;
        unsigned k;
        float * h;
        unsigned long h_len;
        complex<float> * in;
        unsigned long in_len;
        complex<float> * out;

        //Thread structures
        unsigned long num_threads;
        pthread_t * threads;
        pthread_attr_t * attrs;
        sem_t * semaphores;

        //Stop condition + mutexes
        bool * go;
        pthread_mutex_t * protections;
        pthread_mutexattr_t * protAttrs;

        //Condition + mutexes
        pthread_cond_t queue;
        pthread_condattr_t queueAttr;
        pthread_mutex_t mutex;
        pthread_mutexattr_t mAttr;

        unsigned completed;
        t_params * parameters;
        bool opt;
        string mode;

public:

        HwThreadedFilter(liquid_rnyquist_type filterType, unsigned k,
            unsigned m, float beta, float dt, unsigned nThreads, string mode
            );
        HwThreadedFilter(float *, unsigned long, unsigned);
        virtual ~HwThreadedFilter();
        void execute(complex<float> * in, unsigned long l_in, complex<float>
             * out);
        friend void * threadBody(void * ptr);
        void setOptimization(bool);
};


#endif /* HWTHREADEDFILTER_H_ */
```

Listing A.2: *code/HwThreadedFilter.cpp*

```
/*
 * HwThreadedFilter.cpp
 *
 *  Created on: 23/lug/2014
 *      Author: Fabio Del Vigna
 */

#include "HwThreadedFilter.h"

/**
 * Body of the thread
 *
 * @param ptr pointer to the parameters of type t_params
 *
 */
void * threadBody(void * ptr){
        //Retreive parameters
        unsigned long id = ((t_params *) ptr) -> id;
        HwThreadedFilter * myFilter = ((t_params *) ptr) -> filter;
        while(true){
                //Block the thread until new data is available
                sem_wait(&(myFilter->semaphores[id]));
                pthread_mutex_lock(&(myFilter->protections[id]));
                if (!myFilter->go[id]){
                        pthread_mutex_unlock(&(myFilter->protections[id]));
                        break;
                }
                pthread_mutex_unlock(&(myFilter->protections[id]));
                //Calculate how many elems each thread must compute in the
                    output vector
                unsigned long blockSize = ((myFilter->in_len + myFilter->
                    h_len - 1) % myFilter->num_threads == 0)?(myFilter->
                    in_len + myFilter->h_len - 1) / myFilter->num_threads :
                    (myFilter->in_len + myFilter->h_len - 1) / myFilter->
                    num_threads + 1;
                //Calculate convolutional product of a block of elements
                unsigned long start = id*blockSize;
                unsigned long stop = (id < (myFilter->num_threads - 1)) ? (
                    id+1)*blockSize : (myFilter->in_len + myFilter->h_len -
                    1);

                for (unsigned long b = start; b < stop; b++){
                        if (!myFilter->opt || myFilter->resampler == NULL){
                                //Initialize output
                                myFilter -> out[b] = 0;

                                //Initial case
                                if (b < myFilter->h_len - 1){
                                        for (unsigned long i = 0; i <= b; i
                                            ++){
                                                myFilter -> out[b] +=
                                                    myFilter -> h[myFilter->
                                                    h_len - 1 - b + i] *
                                                    myFilter -> in[i];
                                        }
                                }
                                //Final case
                                else if (b > myFilter->in_len - 1){
                                        unsigned long j = 0;
                                        for (unsigned long i = b - myFilter
                                            ->h_len + 1; i < myFilter->
                                            in_len; i++){
                                                myFilter -> out[b] +=
                                                    myFilter -> h[j] *
                                                    myFilter -> in[i];
                                                j++;
                                        }
                                }
                                //All the others
                                else {
                                        for (unsigned long i = 0; i <
                                            myFilter->h_len; i++){
                                                myFilter -> out[b] +=
                                                    myFilter -> h[i] *
                                                    myFilter -> in[i + b -
                                                    myFilter->h_len + 1];
                                        }
                                }
                        }
                        else {
                                //Initialize output
                                myFilter -> out[b] = 0;

                                //Initial case
                                if (b < myFilter->h_len - 1){
                                        for (unsigned long i = 0; i <= b; i
                                            +=myFilter->k){
                                                myFilter -> out[b] +=
                                                    myFilter -> h[myFilter->
                                                    h_len - 1 - b + i] *
                                                    myFilter -> in[i];
                                        }
                                }
                                //Final case
                                else if (b > myFilter->in_len - 1){
                                        unsigned long j = ((b - myFilter->
                                            h_len + 1) % myFilter->k == 0) ?
                                             0 : (myFilter->k - (b -
                                            myFilter->h_len + 1)%myFilter->k
                                            );
                                        for (unsigned long i =  (((b -
                                            myFilter->h_len + 1)%myFilter->k
                                            ==0)?(b - myFilter->h_len + 1):(
                                            b - myFilter->h_len + 1)+(
                                            myFilter->k - (b - myFilter->
                                            h_len + 1)%myFilter->k)); i <
                                            myFilter->in_len; i+=myFilter->k
                                            ){
                                                myFilter -> out[b] +=
                                                    myFilter -> h[j] *
                                                    myFilter -> in[i];
                                                j+=myFilter->k;
                                        }
                                }
                                //All the others
                                else {
                                        for (unsigned long i = (((b -
                                            myFilter->h_len + 1)%myFilter->k
                                             == 0) ? 0 : myFilter-> k - (b -
                                             myFilter->h_len + 1) % myFilter
```

```
                                                    ->k); i < myFilter->h_len; i+=
                                                    myFilter->k){
                                                         myFilter -> out[b] +=
                                                             myFilter -> h[i] *
                                                             myFilter -> in[i + b -
                                                             myFilter->h_len + 1];
                                                }
                                        }
                                }
                        }
                        pthread_mutex_lock(&(myFilter->mutex));
                        myFilter->completed++;
                        pthread_cond_signal(&(myFilter->queue));
                        pthread_mutex_unlock(&(myFilter->mutex));
        }
        pthread_exit(NULL);
}

/**
 * Constructor
 *
 * Build the filter using the parameters passed
 * by the user
 *
 * @param filterType kind of filter as described in Liquid libraries
 * @param k number of samples per symbol
 * @param m filter delay in symbols
 * @param beta rolloff factor
 * @param dt fractional sample delay
 * @param nThreads number of threads to use
 * @param mode specify if the filter is in rx or tx mode. If it is in rx
     mode the input
 * is not oversampled
 */
HwThreadedFilter::HwThreadedFilter(liquid_rnyquist_type filterType, unsigned
     k, unsigned m, float beta, float dt, unsigned nThreads, string mode) {
        h_len = 2 * k * m + 1;
        h = new float[h_len];
        this -> k = k;
        this -> mode = mode;
        completed = 0;
        this -> in = NULL;
        this -> in_len = 0;
        this -> out = NULL;
        liquid_firdes_rnyquist(filterType, k, m, beta, dt, h);
        num_threads = nThreads;

        //Create threads and semaphores
        threads = new pthread_t[num_threads];
        semaphores = new sem_t[num_threads];
        attrs = new pthread_attr_t[num_threads];
        parameters = new t_params[num_threads];

        //Create queue + mutexes
        pthread_condattr_init(&queueAttr);
        pthread_cond_init(&queue, &queueAttr);
        pthread_mutexattr_init(&mAttr);
        pthread_mutex_init(&mutex, &mAttr);

        //Create control variables for stopping threads + mutexes
        go = new bool[num_threads];
        protections = new pthread_mutex_t[num_threads];
        protAttrs = new pthread_mutexattr_t[num_threads];

        for (unsigned i = 0; i < num_threads; i++){
                parameters[i].id = i;
                parameters[i].filter = this;
                sem_init(&semaphores[i], 0, 0);
                go[i] = true;
                pthread_mutexattr_init(&protAttrs[i]);
                pthread_mutex_init(&protections[i], &protAttrs[i]);
                pthread_attr_init(&attrs[i]);
                if(pthread_create(&threads[i], &attrs[i], threadBody, (void
                    *) &parameters[i]))
                        cerr << "An error has occurred while creating
                            threads for filtering" << endl;
        }

        if (k > 1 && !this->mode.compare("tx"))
                resampler = new Resampler(k, num_threads, true);
        else
                resampler = NULL;

        //FIR normalization
        for (unsigned int i = 0; i < h_len; i++){
                h[i] = h[i]/sqrt(k);
        }

        opt = true;
}

/**
 * Constructor
 *
 * Build the filter given the fir. It doesn't perform oversampling.
 *
 * @param h filter characteritics
 * @param h_len length of the array h
 * @param nThreads number of threads to use
 *
 */
HwThreadedFilter::HwThreadedFilter(float * h, unsigned long h_len, unsigned
    nThreads) {
        this -> h = new float[h_len];
        for (unsigned int i = 0; i < h_len; i++)
                this -> h[i] = h[i];
        this -> h_len = h_len;
        this -> k = 1;
        completed = 0;
        this -> in = NULL;
        this -> in_len = 0;
        this -> out = NULL;
        num_threads = nThreads;

        //Create threads and semaphores
        threads = new pthread_t[num_threads];
        semaphores = new sem_t[num_threads];
        attrs = new pthread_attr_t[num_threads];
        parameters = new t_params[num_threads];

        //Create queue + mutexes
        pthread_condattr_init(&queueAttr);
        pthread_cond_init(&queue, &queueAttr);
        pthread_mutexattr_init(&mAttr);
        pthread_mutex_init(&mutex, &mAttr);

        //Create control variables for stopping threads + mutexes
        go = new bool[num_threads];
        protections = new pthread_mutex_t[num_threads];
        protAttrs = new pthread_mutexattr_t[num_threads];

        for (unsigned i = 0; i < num_threads; i++){
                parameters[i].id = i;
                parameters[i].filter = this;
                sem_init(&semaphores[i], 0, 0);
                go[i] = true;
                pthread_mutexattr_init(&protAttrs[i]);
                pthread_mutex_init(&protections[i], &protAttrs[i]);
                pthread_attr_init(&attrs[i]);
                if(pthread_create(&threads[i], &attrs[i], threadBody, (void
                    *) &parameters[i]))
                        cerr << "An error has occurred while creating
                            threads for filtering" << endl;
        }

        resampler = NULL;
        opt = false;
}

HwThreadedFilter::~HwThreadedFilter() {
        if (resampler != NULL)
                delete resampler;
        for (unsigned i = 0; i < num_threads; i++){
                pthread_mutex_lock(&protections[i]);
                go[i] = false;
                pthread_mutex_unlock(&protections[i]);
                sem_post(&semaphores[i]);
                //Wait for thread termination
                pthread_join(threads[i], NULL);
                pthread_mutex_destroy(&protections[i]);
                pthread_mutexattr_destroy(&protAttrs[i]);
                pthread_cancel(threads[i]);
                pthread_attr_destroy(&attrs[i]);
                sem_destroy(&semaphores[i]);
        }
        pthread_mutex_destroy(&mutex);
        pthread_mutexattr_destroy(&mAttr);
        pthread_cond_destroy(&queue);
        pthread_condattr_destroy(&queueAttr);

        delete [] h;
        delete [] threads;
        delete [] semaphores;
        delete [] attrs;
        delete [] parameters;
        delete [] go;
        delete [] protections;
        delete [] protAttrs;

}

void HwThreadedFilter::setOptimization(bool optimized){
        opt = optimized;
}



/**
 * @brief Filter data
 *
 * Execute convolution of the fir of the filter with the input
 *
 * @param in data to filter. May be oversampled by the filter according to
     its
 * specifications.
 * @param l_in length of the in array
 * @param out output of the filter
 *
 */
void HwThreadedFilter::execute(complex<float> * in, unsigned long l_in,
    complex<float> * out){
        complex<float> * resampled = NULL;
        if (resampler != NULL){
                resampled = new complex<float>[l_in * this -> k];
                resampler -> resample(in, l_in, resampled);
                this -> in = resampled;
                this -> in_len = l_in * this -> k;
        }
        else {
                this->in = in;
                this->in_len = l_in;
        }

        completed = 0;
        this->out = out;
        //Start threads
        for (unsigned i = 0; i < num_threads; i++){
                //Unlock thread i
                sem_post(&semaphores[i]);
        }
        //Wait for the termination of all threads.
        pthread_mutex_lock(&mutex);
        while(completed < num_threads)
                pthread_cond_wait(&queue, &mutex);
        pthread_mutex_unlock(&mutex);

        if (resampler != NULL)
                delete [] resampled;
}
```

Appendix B

Resampler code

Listing B.1: *code/Resampler.h*

```
/*
 * Resampler.h
 *
 *  Created on: 24/lug/2014
 *      Author: Fabio Del Vigna
 */

#ifndef RESAMPLER_H_
#define RESAMPLER_H_

#define ZERO_PADDING 0
#define SAMPLE_AND_HOLD 1

#include <pthread.h>
#include <semaphore.h>
#include <cmath>
#include <complex>
#include <cstdlib>
#include <iostream>

using namespace std;
class Resampler;

struct resampler_params{
        int id;
        Resampler * filter;
};
class Resampler {
        unsigned k;
        bool oversampling;
        //Thread structures
        unsigned num_threads;
        pthread_t * threads;
        pthread_attr_t * attrs;
        sem_t * semaphores;
        resampler_params * parameters;

        complex<float> * in;
        unsigned long in_len;
        complex<float> * out;
        //Stop condition + mutexes
        bool * go;
        pthread_mutex_t * protections;
        pthread_mutexattr_t * protAttrs;

        //Condition + mutexes
        pthread_cond_t queue;
        pthread_condattr_t queueAttr;
        pthread_mutex_t mutex;
        pthread_mutexattr_t mAttr;

        unsigned completed;
        unsigned oversamplingMode;

public:
        Resampler(unsigned, unsigned long, bool);
        virtual ~Resampler();
        void resample(complex<float>*, unsigned long l_in, complex<float> *
            out);
        friend void * resamplerBody(void * ptr);
        void setOversamplingMode(int mode);
};

#endif /* RESAMPLER_H_ */
```

Listing B.2: *code/Resampler.cpp*

```
/*
 * Resampler.cpp
 *
 *  Created on: 24/lug/2014
 *      Author: Fabio Del Vigna
 */

#include "Resampler.h"

/**
 * Body of the thread
 *
 * @param ptr pointer to the parameters of type resampler_params
 *
 */
void * resamplerBody(void * ptr){
        //Retreive parameters
        int id = ((resampler_params *) ptr) -> id;
        Resampler * resampler = ((resampler_params *) ptr) -> filter;
        while(true){
                //Block the thread until new data is available
                sem_wait(&(resampler->semaphores[id]));
                pthread_mutex_lock(&(resampler->protections[id]));
                if (!resampler->go[id]){
                        pthread_mutex_unlock(&(resampler->protections[id]));
                        break;
                }
                pthread_mutex_unlock(&(resampler->protections[id]));
                //Calculate how many elements each thread must compute in
                    the output vector
                unsigned long blockSize = ((resampler->in_len) % resampler->
                    num_threads == 0) ? resampler->in_len / resampler->
                    num_threads : (resampler->in_len / resampler->
                    num_threads + 1);

                //Calculate resampled output
                unsigned long start = id*blockSize;
                unsigned long stop = (id < (resampler->num_threads - 1)) ? (
                    id+1)*blockSize : resampler->in_len;
                if (resampler->oversampling){
                        unsigned long j = id*blockSize*resampler->k;
                        for (unsigned long b = start; b < stop; b++){
                                for (unsigned int r = 0; r < resampler -> k;
                                     r++){
                                        switch(resampler->oversamplingMode){
```

```
                                        case ZERO_PADDING:
                                                //Oversampling. Put zero in
                                                    new samples.
                                                if (r == 0)
                                                        resampler->out[j++]
                                                            = resampler->in[
                                                            b];
                                                else
                                                        resampler->out[j++]
                                                            = 0;
                                                break;
                                        case SAMPLE_AND_HOLD:
                                                resampler->out[j++] =
                                                    resampler->in[b];
                                                break;
                                        }
                                }
                        }
                }
                else {
                        for (unsigned long b = start; b < stop; b++){
                                if ((b % resampler -> k) == 0){
                                        resampler->out[b/resampler->k] =
                                            resampler->in[b];
                                }
                        }
                }

                pthread_mutex_lock(&(resampler->mutex));
                resampler->completed++;
                pthread_cond_signal(&(resampler->queue));
                pthread_mutex_unlock(&(resampler->mutex));
        }
        pthread_exit(NULL);
}

/**
 * Constructor
 *
 * @param k resampling factor
 * @param nThread number of thread to use
 * @param mode Specify if the class must act as a decimator or oversampler
 *
 */
Resampler::Resampler(unsigned k, unsigned long nThread, bool mode) {
        this->k = k;
        this->num_threads = nThread;
        oversampling = mode;
        oversamplingMode = ZERO_PADDING;
        //Create threads and semaphores
        threads = new pthread_t[num_threads];
        semaphores = new sem_t[num_threads];
        attrs = new pthread_attr_t[num_threads];
        parameters = new resampler_params[num_threads];

        //Create queue + mutexes
        pthread_condattr_init(&queueAttr);
        pthread_cond_init(&queue, &queueAttr);
        pthread_mutexattr_init(&mAttr);
        pthread_mutex_init(&mutex, &mAttr);

        //Create control variables for stopping threads + mutexes
        go = new bool[num_threads];
        protections = new pthread_mutex_t[num_threads];
        protAttrs = new pthread_mutexattr_t[num_threads];

        for (unsigned i = 0; i < num_threads; i++){
                parameters[i].id = i;
                parameters[i].filter = this;
                sem_init(&semaphores[i], 0, 0);
                go[i] = true;
                pthread_mutexattr_init(&protAttrs[i]);
                pthread_mutex_init(&protections[i], &protAttrs[i]);
                pthread_attr_init(&attrs[i]);
                if(pthread_create(&threads[i], &attrs[i], resamplerBody, (
                    void *) &parameters[i]))
                        cerr << "An error has occurred while creating
                           threads for filtering" << endl;
        }
}

Resampler::~Resampler() {
        for (unsigned i = 0; i < num_threads; i++){
                pthread_mutex_lock(&protections[i]);
                go[i] = false;
                pthread_mutex_unlock(&protections[i]);
                sem_post(&semaphores[i]);
                //Wait fo thread termination
                pthread_join(threads[i], NULL);
                pthread_mutex_destroy(&protections[i]);
                pthread_mutexattr_destroy(&protAttrs[i]);
                pthread_cancel(threads[i]);
                pthread_attr_destroy(&attrs[i]);
                sem_destroy(&semaphores[i]);
        }

        pthread_mutex_destroy(&mutex);
        pthread_mutexattr_destroy(&mAttr);
        pthread_cond_destroy(&queue);
        pthread_condattr_destroy(&queueAttr);

        delete [] threads;
        delete [] semaphores;
        delete [] attrs;
        delete [] parameters;
        delete [] go;
        delete [] protections;
        delete [] protAttrs;

}

/**
 * Resample the input data by decimating or oversampling
 *
 * @param in input data (symbols)
 * @param l_in length of the input
 * @param out resampled input by the function. Must be allocated by the
     caller.
 * It is k times longer or shorter than the input.
 */
void Resampler::resample(complex<float>* in, unsigned long l_in, complex<
    float> * out){
        completed = 0;
        this->in = in;
        this->in_len = l_in;
        this->out = out;
        //Start threads
        for (unsigned i = 0; i < num_threads; i++){
                sem_post(&semaphores[i]);
        }
        //Wait for the termination of all threads.
        pthread_mutex_lock(&mutex);
        while(completed < num_threads)
                pthread_cond_wait(&queue, &mutex);
        pthread_mutex_unlock(&mutex);
}

/**
 * Set the oversampling mode
 *
 * @param mode Specify how to oversample data
 */
void Resampler::setOversamplingMode(int mode){
        if (mode == ZERO_PADDING)
                oversamplingMode = ZERO_PADDING;
        if (mode == SAMPLE_AND_HOLD)
                oversamplingMode = SAMPLE_AND_HOLD;
}
```

Acronyms

AEP: Application Environment Profile. 10

ASIC: Application Specific Integrated Circuit. 7, 8, 60

CORBA: Common Object Request Broker Architecture. 4, 9, 11, 12, 14–16, 24, 27, 38, 60

CPU: Central Processing Unit. 47

DCD: Device Configuration Descriptor. 25

DSP: Digital Signal Processor. 7–9

FFT: Fast Fourier Transform. 36, 49

FIR: Finite Impulse Response. 43, 49–52

FM: Frequency Modulation. 45, 46

FPGA: Field Programmable Gate Array. 7–9

GPP: General Purpose Processor. 7, 8, 11, 12, 25, 47, 49, 56, 61

GPU: Graphics Processing Unit. 47

JTRD: Join Tactical Radio System. 9

NoC: Network on Chip. 60, 61

OE: Operating Environment. 60

OOK: On-Off Keying. 34

PSK: Phase-Shift Keying. 30, 32, 39

QAM: Quadrature Amplitude Modulation. 30

RPC: Remote Procedure Call. 11

SAD: Software Assembly Descriptor. 19, 25, 55, 56

SCA: Software Communications Architecture. 4, 9–12, 14–16, 19, 25, 56, 60, 61

SDR: Software Defined Radio. 4, 7–9, 14–16, 28, 30, 36, 47, 57, 60, 61

SPD: Software Package Descriptor. 19

SRI: Stream Related Information. 13, 23, 24, 26

Glossary

Component: A component is a software module that can be combine d with other components to form a waveform. Components may provide or require ports. 4, 7, 9–14, 16–19, 23–26, 28–30, 34, 36–39, 41–44, 46, 48, 50, 51, 53–58, 60, 61

Core Framework: It is the essential set of open application-layer CORBA interfaces and services which provide an abstraction of the underlying system software and hardware. 9, 12, 37, 46, 57

Device: It's an abstraction of a hardware device that defines the capabilities, attributes, and interfaces for that device. In the context of REDHAWK, a Device is is a software module that implements the Device interface class. 7, 9, 10, 13–15, 25, 26, 60, 61

Node: A set of devices tied together which can run components. 12, 17, 25, 26

Port: A CORBA object that produces or consumes data and/or commands. 12, 13, 16–19, 23, 24, 26, 37

Property: An SCA Property is a variable that contains a value of a specific type. Configuration Properties are parameters to configure and query operations of the PropertySet interface. Allocation Properties define the Resource capabilities required from a Device. 12, 13, 16–19, 24, 25, 28–30, 34, 37, 39, 40, 42, 43, 56, 57

Waveform: A program written specifically for SDR purposes such as encoding, decoding, filtering... a signal. 4, 7, 12–14, 16–19, 23, 25, 26, 28, 36–39, 46–48, 52, 55–57, 60

Bibliography

[1] A. Morelli, "Realizzazione di una waveform per comunicazioni in banda vhf su piattaforma software-defined radio di tipo sca (software communications architecture)," Master thesis, University of Pisa, 2012.

[2] J. M. III., "Software radios-survey, critical evaluation and future directions," *Telesystems Conference*, pp. 13–15, 13–23, 1992.

[3] T. Ulversøy, "Software Defined Radio: Challenges and Opportunities," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 4, pp. 531–550, 2010.

[4] J. P. E. O. (JPEO) and J. T. R. S. (JTRS), *SOFTWARE COMMUNICATIONS ARCHITECTURE SPECIFICATION*, Space and Naval Warfare Systems Center San Diego, 53560 Hull Street, San Diego CA 92152-5001, May 2006, v 2.2.2.

[5] J. Zou, D. Levy, and A. Liu, "Evaluating Overhead and Predictability of a Real-time CORBA System," in *Proceedings of the 37th Hawaii International Conference on System Sciences*, 2004.

[6] T. Ulversøy and J. O. Neset, "On Workload in an SCA-Based System, with Varying Component and Data Packet Sizes," NATO, Tech. Rep., 2008.

[7] F. Casalino, G. Middioni, and D. Paniscotti, "Experience report on the use of CORBA as the sole middleware solution in SCA-based SDR environments," in *Proceeding of the SDR 08 Technical Conference and Product Exposition*, 2008, p. 7.

[8] *Common Object Request Broker Architecture (CORBA) Specification, Version 3.3*, OMG, November 2012.

[9] E. Fernandez-Alonso, D. Castells-Rufas, J. Joven, and J. Carrabina, "Survey of NoC and Programming Models Proposals for MPSoC," *IJCSI International Journal of Computer Science Issues*, vol. 9, no. 3, March 2012.

[10] G. Abgrall, F. L. Roy, J.-P. Diguet, and G. Gogniat, "A Comparative Study of Two Software Defined Radio Environments," in *Proceeding of the SDR 08 Technical Conference and Product Exposition*, 2008, p. 6.

[11] G. R. Wiki. Gnu radio. [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki

[12] V. Tech. Redhawk SDR Framework. [Online]. Available: http://redhawksdr.github.io/Documentation/index.html

[13] E. Foudation. Eclipse. [Online]. Available: http://www.eclipse.org

[14] E. Research. Ettus wiki. [Online]. Available: http://code.ettus.com/redmine/ettus/projects/uhd/wiki

[15] OsmocomSDR. RTL-SDR. [Online]. Available: http://sdr.osmocom.org/trac/wiki/rtl-sdr

[16] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260 – 269, 1967.

[17] G. D. F. Jr., "The viterbi algorithm," *Proceedings of the IEEE*, vol. 13, no. 2, pp. 260 – 269, 1967.

[18] J. Gaeddert. liquid sdr. [Online]. Available: http://liquidsdr.org

[19] N. Y. Times. Intel halts development of 2 new microprocessors. [Online]. Available: http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007

[20] Intel. Intel hyper threading technology. [Online]. Available: http://www.intel.it/content/www/it/it/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[21] K. Group. OpenCL. [Online]. Available: http://www.khronos.org/opencl/

[22] A. Engineering. GitHub repository. [Online]. Available: https://github.com/Axios-Engineering

[23] NVIDIA. CUDA. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[24] M. Luise and G. M. Vitetta, *Teoria dei segnali*, 3rd ed. McGraw-Hill, 2009.

[25] C. R. J. Jr and W. A. Sethares, *Telecommunication Breakdown.* Pearson Prentice Hall, February 2003.