Open-Source SCA Implementation-Embedded and Software Communication Architecture

OSSIE and SCA Waveform Development

EDOARDO PAONE

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:27

Open-Source SCA Implementation-Embedded and Software Communication Architecture

OSSIE and SCA Waveform Development

Edoardo Paone

15 February 2010

School of Information and Communication Technology
Royal Institute of Technology (KTH)
Stockholm, Sweden

Supervisor and examiner at KTH:
Prof. Gerald Q. Maguire Jr

Industrial supervisor at Saab:
Dr. Anders Edin

Abstract

Software Defined Radios (SDRs) are redefining the current landscape of wireless communications in both military and commercial sectors. The rapidly evolving capabilities of digital electronics are making it possible to execute significant amounts of signal processing on general purpose processors rather than using special-purpose hardware.

As a consequence of the availability of SDR, applications can be used to implement flexible communication systems in an operating prototype within a very short time. However, the initial lack of standards and design rules leads to incompatibility problems when using products from different manufacturers. This problem is critical for the military and public safety sectors, for this reason the US Army was interested in SDR and carried out research into the specification of a common software infrastructure for SDR. This initiative started in the mid-1990s and evolved into the Software Communications Architecture (SCA).

SCA is a non-proprietary, open architecture framework that allows a designer to design interoperable and platform independent SDR applications. At the same time the SCA framework, by abstracting the radio communication system, speeds up waveform development because developers no longer have to worry about hardware details.

This thesis project uses OSSIE, an open source SCA implementation, to illustrate the process of developing a waveform. Today companies are exploiting open source solutions and investing money to evaluate and improve available technologies rather than developing their own solutions: OSSIE provides a working SCA framework without any license cost. OSSIE also provides some tools to develop SCA waveforms. Of course open source software comes with some limitations that a designer must take into account. Some of these limitations will be described for OSSIE (specifically the limited documentation and lack of libraries), along with some suggestions for how to reduce their impact.

This thesis project shows in detail the development process for SCA waveforms in OSSIE. These details are examined in the course of successfully implementing a target waveform to enable the reader to understand the advantagies and disadvantages of this technology and to facilitate more people using OSSIE to develop waveforms. Although a waveform was successfully implemented there were unexpected issues with regard to the actual behavior of the waveform when implemented on the hardware used for testing.

Sammanfattning

Software Defined Radio (SDR) håller på att förändra hur trådlös kommunikation utvecklas inom både den militära och den kommersiella sektorn. Den snabba utvecklingen av digital elektronik gör det möjligt att utföra digital signalbehandling på generella processorer istället för att använda särskild hårdvara för signalbehandling.

Tack vare tillgången till SDR, kan mjukvara användas för att utveckla flexibla kommunikationssystem på väldigt kort tid. Bristen på utvecklingsregler kan leda till problem med kompatibilitet när produkter från olika tillverkare används tillsammans.

Den amerikanska armén har därför utvecklat en arkitektur för SDR. Detta initiativ startade i mitten av 1990-talet och har lett fram till Software Communications Architecture (SCA).

SCA är en generisk, öppen arkitektur som möjliggör att utveckla interoperabla och plattformsoberoende SDR-applikationer. Samtidigt kan SCA, genom att abstrahera radiokommunikationssystemet, påskynda vågformsutveckling eftersom utvecklarna inte längre behöver ta lika stor hänsyn till radiohårdvaran.

Detta examensarbete använder OSSIE, en implementation av SCA som öppen källkod, för att illustrera utvecklingsprocessen för en vågform. Företag utnyttjar idag öppen-källkodslösningar och investerar pengar för att utvärdera och förbättra tillgänglig teknik istället för att utveckla sina egna lösningar. OSSIE ger tillgång till en SCA platform utan någon licenskostnad. OSSIE erbjuder också enkla verktyg för att utveckla SCA-vågformer. Naturligtvis leder öppen källkod tilll vissa problem som en utvecklare måste ta hänsyn till. Några av dessa begränsningar kommer att beskrivas för OSSIE (särskilt brist på dokumentation), tillsammans med några förslag för att lösa problemen.

Målet med detta examensarbete är att förstå, samt visa fördelar och nackdelar med vågformsutveckling med OSSIE.

Acknowledgements

This thesis work took place during the fall of 2009 at Saab Systems in Järfälla, Sweden. There are a number of people whom I would like to thank for being helping during my thesis work.

First and foremost I would like to thank my supervisor at Saab AB, Anders Edin, for helping me and guiding me through the thesis work. Also my supervisor at KTH, Pr. Gerald Q. Maguire Jr, has given me great support: this report would not have shaped up so nice without his help and advice.

Also, I would like to thank my director at Saab AB, Stefan Hagdahl, and all my collegues in the Communication Systems section. The work environment has been very nice and I had a very good time. I really appreciate the open atmoshere where I have been able to get support when needed. The friday afternoons have been really nice, I have had the opportunity to enjoy the swedish *fika*. I am greatful to you all, thanks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of this master's thesis is to illustrate waveform development using OSSIE (Open Source SCA Implementation - Embedded). Software Communications Architecture (SCA) is a standard architecture that is widely used for specifying software defined radios (SDRs). The waveform that has been selected for implementation is part of the protocol described by IEEE 802.15.4 [24], a standard protocol for wireless personal area networks (WPANs). The two main properties of SCA, **portability** and **modularization**, will be described referring to development process of the target waveform.

This thesis project also utilizes other open-source initianitives for SDR, specifically GNU Radio (section 2.3) and the Universal Software Radio Peripheral (USRP, section 2.2). These two efforts (the first software and the second hardware) allow developers to implement software radios using readily-available, low-cost external RF hardware and commodity processors.

This thesis project took place during the fall of 2009 at Saab Systems in Järfälla, Sweden.

1.1 Motivations

Before starting this project, I did not know anything about SDR: I had studied computer engineering and my background was limited to system on a chip (SoC) design and low-level programming. After reading some documentation about OSSIE and SCA, in two months I implemented a radio communication protocol and could transmit data between two USRP stations. SDR enables rapid waveform prototyping, even for people who are not experienced in radio applications. SDR removes a significant barrier to the development of new radio communication systems and communication protocols, as SDR shifts the focus from RF circuit design to programming.

Rapid prototyping is not the only advantage of SDR, another important feature comes with using SCA: waveform applications can be both **interoperable** and **platform independent**, so that the waveform developer does not need to understand the radio engineering aspects of the design. Additionally, since SCA abstracts the details of the underlying hardware platform, it is easier to interface correctly with the underlying software core framework. Hence, the report will focus on the characteristic of SCA, **abstraction**, **modularization**, and **portability**; these characteristics, in fact, represent the difference and – at the same time – the advantage of SCA compared to other SDR frameworks, such as GNU Radio.

Since SDR waveforms are implemented as software, it is possible to run more than one waveform on the same platform; in the same way as different programs can be executed in parallel on the same PC. However, just as in the case of other software the bound is the computational resources available on the platform, such as available memory at each layer of the memory hierarchy, the performance of each type of memory, CPU speed and number of CPU cores, capabilities of the analog to digital and digital to analog converters (ADC and DAC), and attached RF hardware. The resources that are generally available today enable the integration of multiple different waveforms on the same platform, with a decrease in product size and potentially a greater than linear decrease in cost. One result is that implementing five different waveforms is not five times as expensive, but perhaps only a little more expensive than a single waveform. This is achieved because the margin cost of replicating software is nearly zero – this is the true contribution of SDR. Additionally, SDR is able to exploit the Moore's law developments of digital electronics – providing cost versus performance improvements at a far greater rate than traditional radio communication technologies.

The flexibility of using software for waveform development facilitates the development process, but it also means that radios can be modified over their life-cycle, for example to keep up with new radio protocols, via a software upgrade. In this way radio manufacturers can rapidly add functionality to their radios – while simultaneously a better product quality can be achieved. The ability to program (and reprogram) the radio is why reconfigurability is a key feature of SDR.

Last, but not least, the economic aspect has to be considered. In the past radios were typically designed to implement a specific functionality and this was often done using hardware components such as filters and amplifiers. In comparison, using SDR an FM receiver can be implemented by writing a few lines of code and running the waveform on a normal PC connected to a low-cost hardware device with a generic RF front-end (for example a USRP, described in section 2.2), creating a very expensive and power consuming FM radio. However, this same platform - a normal PC and an USRP - can be used to implement a TV receiver, a WLAN interface, or an *ad hoc* wireless network *without* requiring any additional hardware.

SDR allows functionality to evolve or change, for the manufacturer, this means that there is a late binding of what the functionality is – this enables (1) hardware and software to be developed in parallel, reducing time to market and (2) multiple products to share the same hardware platform, but be differentiated by the software that is installed on this platform, increasing the product manufacturing volume. The increased product volume is important because this means that SoC techniques can be used – because there will be sufficient return on investment to justify the development cost – this reduces the marginal cost of the hardware. SDR has a disruptive effect due to both hardware and software marginal costs being reduced – turning radios into commodities.

The advantages and disadvantagies of the waveform development with SCA will be discussed in details in the following chapters.

1.2 Problem description

This thesis project concerns the development of a waveform using OSSIE, an open-source software package which includes an SDR core framework based on the Software Communications Architecture (SCA). More information about OSSIE is given in section 2.4. This waveform allows two nodes to communicate with each other by implementing (part of) the IEEE 802.15.4 standard. The nodes are each a PC running Linux with a USB attached USRP (described in section 2.2) and an RF daughter board card installed in the USRP; the only additional hardware required is an antenna.

This waveform had already been implemented using GNU Radio, another SDR framework (described in section 2.3). What is unique to this thesis is the waveform development process, because OSSIE waveforms – according to the SCA specification – use CORBA for communications between components.

Hence as part of this thesis project I will analyze the benefits of the SCA and compare the SCA implementation of this waveform with the GNU Radio implementation, in order to understand why SCA-compliant software defined radios are becoming a standard. Moreover, rather than simply implementing a single specific waveform I want to understand and evaluate the SCA development process, including the different stages of waveform development, the architectural choices, and the problems that were encountered during this process. The end goal is both to learn about this important new area technology and to provide input to my employer about OSSIE and SCA.

1.3 Thesis organization

Chapter 2 introduces SDR and the SCA and illustrates their characteristics, together with some example applications. To understand SCA requires some knowledge of both CORBA and XML, hence these will be presented when needed in the chapter. The protocol to be implemented, IEEE 802.15.4, is also introduced in this chapter; but the details of this standard will be explained in chapter 3 during the discussion of the development process of the selected waveform.

In chapter 4 the different stages of the design are described along with a proposed solution. This chapter will also present the problems that occurred during the thesis project. The chapter ends with a section dedicated to the testing of the selected waveform application. This will be followed by a discussion of possible extensions of the waveform and an analysis of the advantages and disadvantages of the SCA development process. Finally the thesis will conclude with some conclusions and suggestions for future work.

Chapter 2

Background

The possibility to change the waveform implemented by a radio by using software and the ability to use the same software on different platforms have lately become more and more interesting capabilities for both users and manufacturers. This flexibility is enabled by a technique called Software Defined Radio (SDR). In response to this interest a standardized architecture for creating software waveforms has been developed in the United States by the Department of Defense. This architecture is known as the Software Communications Architecture (SCA) and is described in section 2.4.

There are other methods of developing software waveform applications, one of them is GNU Radio. This method is presented in section 2.3. Section 2.2 describes the hardware platform that has been used in this thesis. This same hardware platform was used for the GNU Radio effort described in chapter 4.

2.1 Software Defined Radio

This section is an introduction to Software Defined Radio (SDR) in general. In sections 2.3 and 2.4 two different ways to implement SDR are introduced.

2.1.1 Introduction

Tranter, et al., describe software radio as: “Software radio is a radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software” [16]. Since radio was invented, waveforms and radio functions were implemented by using signal generation, modulation/demodulation, filter functions, and up/down-conversion of frequencies. Because of cost and manufacturing volumes, each radio was designed for a specific function, for example a consumer FM-radio implements exactly one waveform, the receiver converts an FM modulated radio signal into sound.

The master's thesis by Sundquist states: "Now imagine a radi technology that can turn your kitchen radio into a GSM telephone, or a GPS receiver, or maybe a satellite communications terminal. Or why not a garage door opener? That is exactly the opportunities that emerge with software radios!" [25]

2.1.2 History

The term "Software Defined Radio" was coined in 1991 by Joseph Mitola, who published the first paper on the topic in 1992 [14]. Though the concept was first proposed in 1991, software-defined radios have their origins in the defense sector in the late 1970s in both the U.S. and Europe. One of the first public software radio initiatives was a U.S. military project named SpeakEasy, from 1992 to 1995. The primary goal of the SpeakEasy project was to use programmable processing to emulate more than ten existing military radios, operating in frequency bands between 2 and 2000 MHz. Further, another design goal was to be able to easily incorporate new coding and modulation standards in the future, so that military communications could keep pace with advances in coding and modulation techniques.

2.1.3 Waveforms

There are a lot of references to the word "waveform" in this thesis, so it is important to explain what this term means when talking about radio communications. A waveform can be described as a radio's function, everything that is used to generate and to decode the radio signal making communication possible is included in the concept of a waveform. Amplification, modulation, filtering, . . . , are components used to implement a radio communication channel. Together they are used to realize a waveform.

For instance, Wideband Code-Division Multiple-Access (WCDMA) is one of the main technologies for the implementation of third-generation (3G) cellular systems. This is an example of a waveform. While GSM and FM are other examples of waveforms.

Hence when talking about a waveform, we are referring to more than the actual electromagnetic waves, as one might think when first encountering the word.

2.1.4 SDR applications

A Software Defined Radio (SDR) is one way to realize a radio communication system. In this approach, many of the components that have typically been implemented in hardware (e.g. mixers, filters, amplifiers, modulators/demodulators, detectors. etc.) are instead implemented using software. Since the radio communication system is now implemented primarily as software, it can be run on a personal computer (PC) or other embedded computing device with sufficient input, output, and computational capabilities. Therefore, to realize the selected waveform I connected a laptop computer to a USRP (see 2.2), thus the SDR application, running on the processor of my PC, performs some of the computations and passes samples to and from the USRP, which in turn transmits (or receives) an RF signal through an antenna (or antennas).

While the concept of SDR is not new, the application of this technology was limited in the past by the limited execution performance of low cost general purpose processors. Today the rapidly evolving capabilities of digital electronics are making practical many processes that were once only theoretically possible. In fact, significant amounts of signal processing are performed today on a general purpose processor, rather than using special-purpose hardware. For example, when using the USRP, the processor installed in a modern PC is fast enough to execute several different kinds of waveform. In fact, the bottleneck is in many practical cases the limited throuput of the USB connection between the PC and the USRP. This is why the USRP2, the successor version of the USRP, utilizes a Gigabit Ethernet rather than a USB interface.

The principle of performing this signal processing using the main processor of a PC is the basis for softmodem technology: a softmodem, or software modem, is a modem with minimal hardware, designed to use a host computer's resources (mostly CPU power and RAM, but sometimes even using the built-in audio hardware) to perform most of the tasks previously performed by dedicated hardware in a traditional modem. Similarly many WLAN interfaces are implemented today by executing much of the processing on the PC's CPU rather than having a separate digital signal processing chip associated with the WLAN interface. Having most of the modulation functions implemented in software provides the advantage of easier upgrades and allows easily implementing new modem standards. A more practical advantage of softmodems is given by the reductions in production costs, component size, and weight compared to a hardware modem. Today most modems that are integrated in portable computer systems (including laptops and PDAs) are softmodems.[1]

Since waveforms are implemented in software, SDR makes it possible to execute different waveforms in parallel on the same processor in the same way as different programs run at the same time in a PC[2]. This represents a challenge for SDR developers: never before have there been so many consumer and military devices that use at least one form of wireless technology, with many implementing two or more wireless protocols in a single device. One way to do this is to add transistors

[1]Christian Olrog, Direct Radio Link Protocol Interface, Masters Thesis, KTH, Stockholm, Sweden, April 1999 - implemented a softmodem for GSM using Linux.

[2]Of course doing this comes with the same problems of running two different programs, along with the additional limitations that these are generally near-real-time programs – hence scheduling and interleaving of the execution of these parallel executions becomes an issue. Fortunately, multi-core machines make this easier – as one core can be assigned to each waveform.

and electrical components, so that each waveform is implemented by a specific signal processor. SDR instead allows one platform to support multiple wireless protocols, hence reusing the same hardware for different waveforms.[3]

Radio communications are particularly important in the militar sector, this explains why SDR projects often involve the military. Different kinds of communications are used by the military and the radio equipment is traditionally single-purpose, meaning that each specific waveform requires dedicated hardware. A driving force was the incompatibility of the different devices with each other – especially bad when the different services could not even talk to each other (for example, the Navy shelling a beach, on which there are Marine and Army units – but none of these different units can talk to each other because their radios are incompatible). SDR technology allows different protocols to be integrated in the same platform, thus compact radios can replace multiple traditional radios needed for the same communication capabilities. Additionally, the flexibility of the software makes it easy to upgrade the equipment in order to keep pace with new communication protocols. It also allows units to change waveforms when in battle – to use different waveforms than during peace-time exercises. This flexibility is also necessary because increasingly *ad hoc* coalitions are formed for specific actions and they need to communicate, but at other times they might be opponents; hence there is a need to have shared waveforms and a need for different waveforms.

2.1.5 Operation scheme of an ideal SDR

The ideal receiver scheme would be to attach an analog-to-digital converter to an antenna. The operational scheme is illustrated in figure 2.1. The digital signal would be received and processed by a general purpose processor (GPP), according to the software that implements a specific waveform. An ideal transmitter would be similar: a GPP would generate a stream of samples as the output of the SDR application. These samples would be sent to a digital-to-analog converter connected to a radio antenna.

Such a design produces a radio that can receive and transmit different waveforms by running different software. However, this is the ideal scheme, in reality a low-noise amplifier must precede the analog to digital conversion step; unfortunatly, non-linearities and noise in the amplifier may introduce distortion in the desired signals. The standard solution is to insert band-pass filters between the antenna and the amplifier, but unfortunatly these reduce the radio's flexibility. Today microelectricalmechanical systems are being used to physically switch in and out filters (and to implement filters), thus restoring some of the flexibility.

Antennas are designed for a specific range of frequencies, so the same antenna would usually not be used for 2 MHz and 3 GHz.[4] Another aspect to take into

[3]See note above with respect to multicore processors and scheduling.

[4]Note that there are log periodic antennas that cover the range from 200 MHz to 2GHz (see for

account when deploying a SDR is the capability of the ADC and DAC because they have a limited sampling rate.[5] As a result of power and cost considerations, SDR applications that work in different frequency bands utilize different RF front-ends. The USRP, as will be described in section 2.2, has been designed to be flexible and is open source hardware so that developers can make their own daughterboards for specific needs.

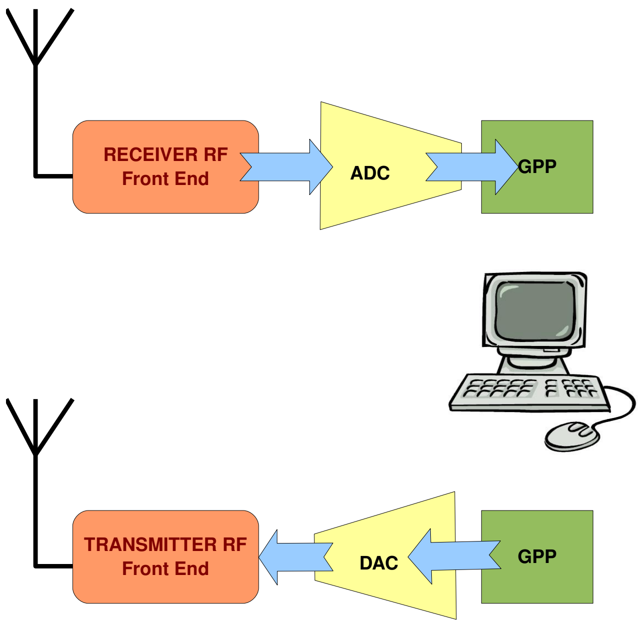


Figure 2.1. Ideal transmitter and receiver in SDR

example the Agilent 11956A antenna), there are horn antennas that cover 200 MHz to 2GHz (see the ATH200M2G) or the Agilent 11966P 26 MHz to 2GHz BiConiLog antenna, 26 MHz to 2GHz with the ETS Lindgren 314B (Biconic/log) antenna, etc. There is a AOR SA7000 that is designed for 30kHz to 2GHz (receiving antenna) – this antenna is 1.8m high and consists of two antennas and built-in duplexer.

[5]Commercial ADCs are available today that sample at upto several Gsps (Tektronix has a number of oscilloscopes that sample at 50 Gsps); while DACs such as Analog Devices AD9122 can process 16 bit samples at 1.2 Gsps or the AD9739 with 14 bit samples at 2.4Gsps, TEK Microsystems dual 14 bit DACs at 3Gsps, Tektronix has a number of arbitrary waveform generators that operate at upto 24 Gsps.

2.2 The Universal Software Radio Peripheral

The Ettus Research LLC Universal Software Radio Peripheral (USRP) is a peripheral for implementing software radios. The USRP was developed by a team led by Matt Ettus [10].

2.2.1 Characteristics

The USRP is a low-cost hardware device designed for rapid prototyping and research in SDR applications. Its first realization allows a developer to create a software radio using any computer with a USB 2.0 port. The USRP has an open design, with freely available schematics and drivers, and free software to integrate with GNU Radio. It was also designed to be flexible. In addition to the open source hardware approach that has been used for the USRP and the daughterboards sold by Ettus Research, developers can make their own daughterboards for specific needs with regard to connectors, different frequency bands, etc.

As showed in table 2.1, the USRP2 offers higher performance and increased flexibility in comparison to the original USRP. It also utilizes a much larger FPGA which can even be used to operate the device in a standalone mode, i.e., without a host computer.

Table 2.1. USRP and USRP2 specifications

subsystem	**USRP**	**USRP2**
ADC	64 Msps, 12bit (AD9862)	100 Msps, 14 bit (LTC2284)
DAC	128 Msps, 14 bit (AD9862)	400 Msps, 16 bit (AD9777)
FPGA	Altera Cyclone EP1C12Q240C8	Xilinx Spartan 3-2000
interface	High-speed USB 2.0	Gigabit Ethernet
expansion sockets	2 TX, 2 RX	2 Gbps serial interface

2.3 GNU Radio

GNU Radio[6] is an open-source SDR platform. A large worldwide community of developers and users have contributed to it and to provide many practical applications for the hardware and software[7]. Using GNU Radio it is very easy to create your own SDR applications because it provides libraries to support all common software radio needs, including various modulations, error-correcting codes, signal processing constructs, and scheduling. It is a very flexible system and it allows

[6] http://www.gnu.org/projects/gnuradio/

[7] Refer to the Comprehensive GNU Radio Archive Network (CGRAN), a free open source repository for 3rd party GNU Radio applications

applications to be developed using C++ and Python. The signal processing code (blocks) in the libraries are written in C++, while Python is used to interconnect the blocks and to provide various user interface functions.

2.3.1 GNU Radio architecture

Figure 2.2 illustrates the GNU Radio architecture. The top layer provides the user interface to do waveform development. The waveform can be represented as a flow graph, where the nodes correspond to the signal processing blocks and the arcs represent interconnections between these processing blocks (see the flow graph for a simple dialtone – shown in figure 2.3). The interconnection of the blocks is done by writing Python code. Details of the waveform development process will be described later in section 2.3.2.

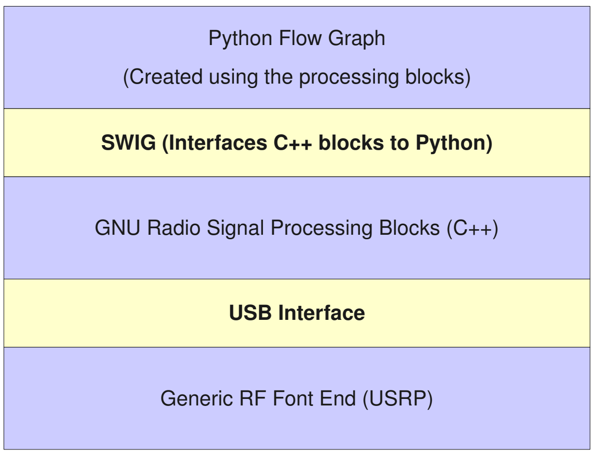

Python Flow Graph
(Created using the processing blocks)
SWIG (Interfaces C++ blocks to Python)
GNU Radio Signal Processing Blocks (C++)
USB Interface
Generic RF Font End (USRP)

Figure 2.2. Block diagram of the GNU Radio architecture

The signal processing blocks have attributes, such as the number of inputs and outputs, type of data that each stream consists of, etc. The most common types of data are shorts, floats, and complex numbers. Some building blocks have only output ports or only input ports, these serve as data sources and sinks (respectively). Examples of sources include blocks for reading from a file or a block which generates a specific signal (such as a sine wave). A sink might be a sound card, a graphical display, or a USRP transmitter.

If you need a building block that is not (yet) available in the GNU Radio library, you can create it yourself. C++ is the language used to create signal processing blocks in GNU Radio. From a high-level point of view, unbounded streams of data flow through the ports; at the C++ level, these streams correspond to arrays of the particular data type to be used. There is a lot of documentation and very good tutorials that explain in detail how to write your own signal processing block.

One of GNU Radio's strengths is that it uses Python, a powerful scripting language, to construct the flowgraph for the waveform at a high level, but uses C++ functions at a lower level of abstraction to implement the signal processing blocks. In order for programs written in Python to communicate (which in GNU Radio means to transfer streams of data) with the building blocks written in C++ a library, called SWIG (Simplified Wrapper and Interface Generator), is used. The advantage of SWIG is that it makes the details of passing the data between the blocks transparent to the waveform developer.

As we saw in figure 2.2, the signal processing blocks access the USRP RF resources through the USB interface. The blocks needed to talk to the USRP already exist in the GNU Radio library, thus the waveform developer does not have to worry about low-level programming in order to communicate with the external device. However, sometimes the waveform developers do have to concern themselves with these details in order to get the performance that they want. As with many systems, getting a functional implementation is not too difficult, but tuning the performance in order to get high performance (or in some cases to actually meet the protocol's specification) may require very deep knowledge across the entire protocol stack from the hardware to the waveform.

2.3.2 An example: Generating a Dial Tone

A North America dial tone consists of a combination of a 350 Hz sine wave and a 440 Hz sine wave at -13dBm each [11]. The flow graph in figure 2.3 represents the waveform we want to implement in GNU Radio, i.e., a simple signal generator that reproduces the North American dial tone.

```
+------------------------+
| Sine generator (350Hz) +----+
+------------------------+    |   +------------+
                              +-->+            |
                                  | Audio sink |
                              +-->+            |
+------------------------+    |   +------------+
| Sine generator (440Hz) +----+
+------------------------+
```

Figure 2.3. Flow graph of the GNU Radio dial tone example

The source code shown in listing 2.1 comes with the GNU Radio package. If this application is executed on a PC, the user will hear on the PC's audio (output) device a dial tone. However, the actual dial tone that is heard is generated by the appropriate two sine waves, with one heard on the left channel and the other on the right channel of the audio device.[8] A complete tutorial and all the details about this source code can be found on the GNU Radio website[9], here only the main properties will be described.

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

from gnuradio import gr
from gnuradio import audio

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

        sample_rate = 32000
        ampl = 0.1

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


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

Listing 2.1. Dial Tone application for GNU Radio

The class `my_top_block` is derived from another class, `gr.top_block`. This class is basically a container for the flow graph. Hence by deriving the `my_top_block` from `gr.top_block` it becomes possible to add your own block and connect it to other blocks.

The class has only a constructor method where the sub-components are instanciated and properly connected. Two signal sources are generated (called `src0` and `src1`). These sources continuously create sine waves at the specified frequencies (350 and 440Hz) and a given sampling rate (here 32 thousand samples per second). The amplitude is controlled by the `ampl` variable and set (arbitrarily) to 0.1.

The call to `audio.sink()` returns a block which provides soundcard control and plays any samples input to it (note that the sampling rate needs to be set explicitly as the stream does not provide such meta-information about itself).

[8] Note that the tones are not actually combined in a single channel – as would be required in a telephony system – however, another block could be added to the flow graph to sum these two inputs and output a single output to one (or both) channels of the audio device. Additionally, there is no guarantee that the amplitude of the audio output is correct, to correct this the developer would need to calibrate the audio device and adjust the value of the “ampl” variable of the program to produce a 10 dBm signal at the audio output (Note that a 10 dBm signal is the sum of two 13 dBm signals).

[9] http://gnuradio.org/trac/wiki/Tutorials/WritePythonApplications

The general syntax for connecting blocks is `self.connect(block1, block2, block3...)` which would connect the output of `block1` with the input of `block2`, the output of `block2` with the input of `block3`, and so on. The audio sink, which is your PC's audio device, has two channels specified by the tuples `(dst, 0)` and `(dst, 1)`, in this way one sine wave is played on the left channel and the other on the right channel.

The waveform that has been created can be executed by running the main method (for example, if you are in the source directory you can simply say: `./dial_tone.py`).

2.3.3 The development process

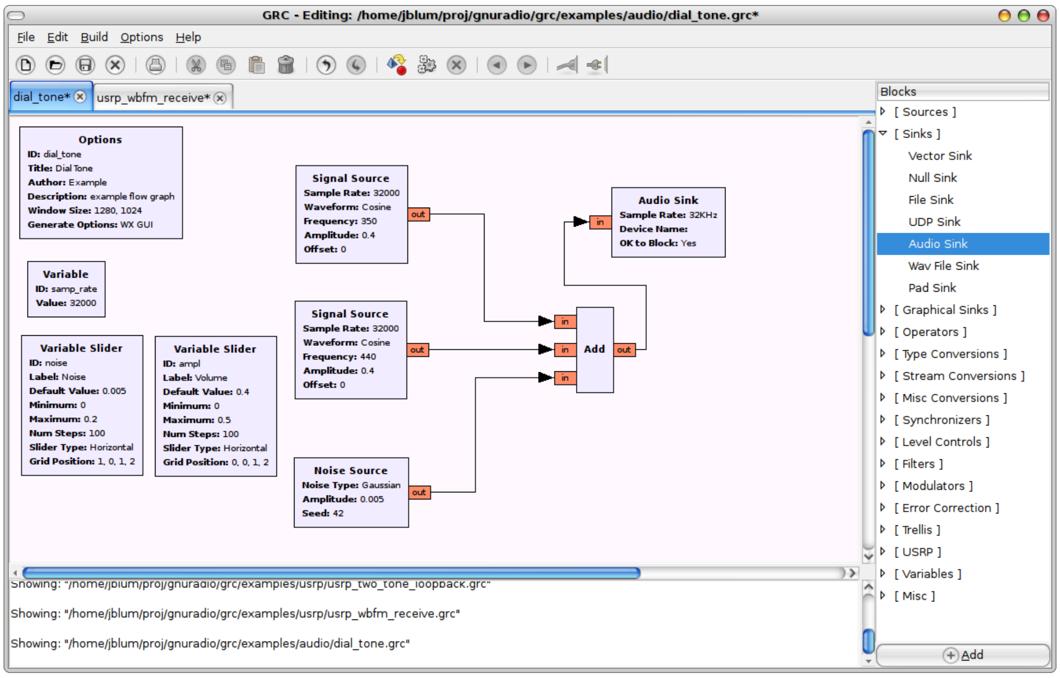
The above example illustrates some properties of developing a waveform using GNU Radio:

- The GNU Radio library includes many useful blocks, thus often a waveform can be created by simply connecting library components. This allows rapid prototyping and facilitates development of new waveforms.
- It is easy to implement a waveform, as the flow graph representing the connections between components can be easily translated into calls to the function `connect()`.
- In the example, the audo sink has two ports; these can be addressed using tuples of form: `(block_name, port_name)`.
- GNU Radio applications are primarily written using the Python programming language, which provides the glue logic for interconnecting components (see figure 2.2).
- Performance-critical signal processing blocks can be implemented in C++ using floating point processing extensions (where available).
- There is a graphical tool, called GNU Radio Companion (GRC), for creating signal flow graphs and generating flow-graph source code (see figure 2.4).

2.3.4 Conclusions

GNU Radio comes with a large amount of user written documentation. In addition, very good tutorials are available on the web. The libraries and examples provide many building blocks, often allowing waveform development to be easy and rapid.

In addition to writing Python code, graphical tools enable a waveform developer to draw a flow graph and have the glue logic automatically generated. The


GRC - Editing: /home/jblum/proj/gnuradio/grc/examples/audio/dial_tone.grc*

Options
ID: dial_tone
Title: Dial Tone
Author: Example
Description: example flow graph
Window Size: 1280, 1024
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 32000

Variable Slider
ID: noise
Label: Noise
Default Value: 0.005
Minimum: 0
Maximum: 0.2
Num Steps: 100
Slider Type: Horizontal
Grid Position: 1, 0, 1, 2

Variable Slider
ID: ampl
Label: Volume
Default Value: 0.4
Minimum: 0
Maximum: 0.5
Num Steps: 100
Slider Type: Horizontal
Grid Position: 0, 0, 1, 2

Signal Source
Sample Rate: 32000
Waveform: Cosine
Frequency: 350
Amplitude: 0.4
Offset: 0

Signal Source
Sample Rate: 32000
Waveform: Cosine
Frequency: 440
Amplitude: 0.4
Offset: 0

Noise Source
Noise Type: Gaussian
Amplitude: 0.005
Seed: 42

Audio Sink
Sample Rate: 32KHz
Device Name:
OK to Block: Yes

```
Showing: "/home/jblum/proj/gnuradio/grc/examples/usrp/usrp_two_tone_loopback.grc"

Showing: "/home/jblum/proj/gnuradio/grc/examples/usrp/usrp_wbfm_receive.grc"

Showing: "/home/jblum/proj/gnuradio/grc/examples/audio/dial_tone.grc"
```

Figure 2.4. GNU Radio Companion (GRC), a tool for creating signal flow graphs and generating flow-graph source code

communication between Python programs and C++ signal processing blocks is transparent to the (casual) developer, so the developer does not have to worry about these details.

The building blocks for accessing the RF resources on the USRP are included in the GNU Radio library, thus it is easy to create your own waveforms – as a waveform developer can concentrate on using the higher layer abstraction.

All these aspects, and the fact that it is free, are advantages of the GNU Radio platform. Together these make GNU Radio a nearly-perfect toolkit for learning about, building, and deploying a software radio. Several projects have been developed using GNU Radio, including a complete HDTV receiver & transmitter and a multichannel FM receiver capable of listening to several channels at the same time.

It should be noted that GNU Radio is primarily intended to enable research and education concerning SDR and wireless communications (see for example the recent thesis project using GNU Radio and the USRP for laboratory execises to teach undergraduates about the physical and media access and control (MAC) layer of protocols [15]). "While the GNU Radio project is educational and potentially beneficial in a research environment, the architecture definition is not yet extensive enough to compete for serious commercial or military applications" [7]. This is why a lot of research is done using the SCA, as described in the next chapter.

2.4 Software Communications Architecture

SDR is a technology without any prior constraints or design specification. SDRs are characterized by a significant software component and provide flexibility on the physical layer. The utilization of this technology by both the military and commercial firms is only practical if a common SDR architecture is defined and a design model is standardized. In the 2006 master's thesis by Jacob A. DePriest, he says "The JTRSs Software Communications Architecture (SCA) is currently the most complete and well-dened architecture available for SDRs"[7]. This appears to still be true today.

2.4.1 Introduction

Following Moore's law, the computational capabilities of general purpose processors (GPPs), digital signal processors (DSPs) and field programmable gate arrays (FPGAs) are growing very rapidly and as a consequence more of the radio signal processing can be now be done in software, thus has lead to practical SDRs. However, each manufacturer has developed solutions with different architectures and often multiple implementations. So even if each product could support SDR, it is not always possible to communicate using radio products from different manufacturers.

This issue is critical in the military and public safety sectors. "When multiple radio sets from different manufacturers came together in response to some coordinated exercise, military operation, or disaster, the radios did not operate and could not be easily reconfigured to do so" [4]. This is the reason why the US military sponsored research into the specification of a common software infrastructure for SDR. This initiative started in the mid-1990 and evolved into the Software Communications Architecture (SCA).

2.4.2 What is SCA?

The Software Communications Architecture (SCA) is a non-proprietary, open architecture framework that tells designers how elements of hardware and software have to operate in harmony within a software defined radio. The SCA is not a system specification, but was intended to be implementation **independent**. Thus rather than being a specification, it is a set of design constraints. If a developer designs a system according to these design rules, the system will be **portable** to other SCA implementations regardless of what operating system or hardware that implementation utilizes.

The aim of SCA is to define an Operating Environment (OE), often referred to as the Core Framework (CF). This CF implements the management, deployment, configuration, and control of the radio system and the applications that run on top of it. The following excerpt from the introduction to the SCA specification, explains what the SCA is.

> "The Software Communication Architecture (SCA) specification is published by the Joint Tactical Radio System (JTRS) Joint Program Office (JPO). This program office was established to pursue the development of future communication systems, capturing the benefits of the technology advances of recent years, which are expected to greatly enhance interoperability of communication systems and reduce development and deployment costs. The goals set for the JTRS program are:
>
> - greatly increased operational flexibility and interoperability of globally deployed systems;
> - reduced supportability costs;
> - upgradeability in terms of easy technology insertion and capability upgrades; and
> - reduced system acquisition and operation cost.
>
> In order to achieve these goals, the SCA has been structured to
>
> - provide for portability of applications software between different SCA implementations;
> - leverage commercial standards to reduce development cost;
> - reduce development time of new waveforms through the ability to reuse design modules; and
> - build on evolving commercial frameworks and architectures."

SCA V2.2, November 17, 2001, p. vii

There are two key objectives of the SCA: flexibility and interoperability. They are important because SCA is intended to provide for **portability of applications** and support the **reuse of waveform design modules**.

2.4.3 CORBA

SCA is based on several underlying technologies: Object-Orientation (OO) design, the Common Object Request Broker Architecture (CORBA), and CORBA Component Model (CCM). CORBA is an industry standard for describing interfaces using Interface Description Language (IDL). IDL allows designers to specify the interfaces between two components, then an IDL compiler generates the code necessary to support communication between the components – using remote procedure calls. The remote procedure calls can be between processes on the same computer or between computers, i.e. in a distrubuted environment.

CORBA is one of the most specific characteristics of the SCA. The transfer of data between two components in the waveform is implemented as CORBA remote procedure calls. Figure 2.5 shows an example of a client making a request to a server using a remote procedure call.

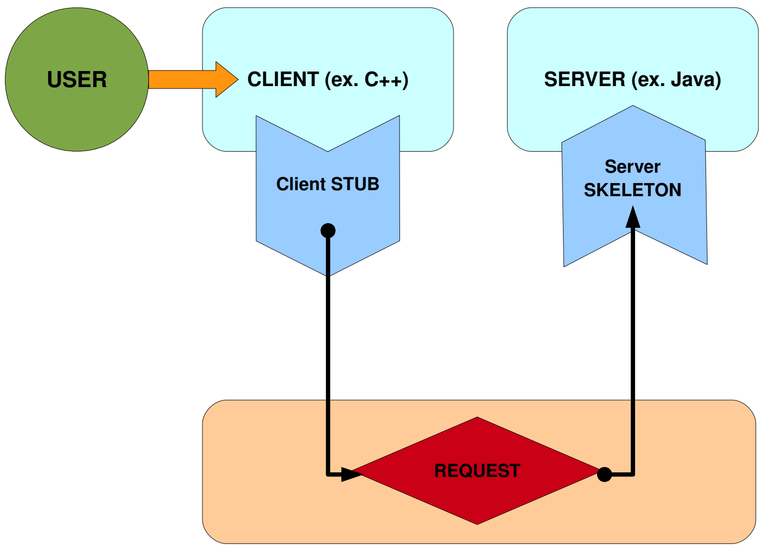


Figure 2.5. Client-server model in CORBA using stubs and skeletons

2.4.3.1 IDL

The component's interface must be described using IDL. An example of IDL is shown in listing 2.2. The listing shows two interfaces, corresponding to two classes. The IDL compiler generates the necessary application code by producing skeletons for the server and stubs for the client, making these interfaces and methods available to callers. These stubs and skeletons run on top of an Object Request Broker (ORB) acting as proxies for servers and clients, respectively.

```
module bankidl {

    interface Account {
                readonly attribute float balance;
                exception rejected { string reason; };
                void deposit(in float value) raises( rejected);
                void withdraw(in float value) raises( rejected);
    };

    interface Bank {
                exception rejected { string reason; };
                Account newAccount( in string name) raises( rejected);
                Account getAccount ( in string name);
                boolean deleteAccount( in string acc );
    };

};
```

Listing 2.2. IDL interface for a banking application

Using IDL compilers, it is possible to generate stubs and skeletons in different languages (such as C, Java, Python, . . .). As a result each client and server can be written in a different language, but they can communicate with each other because the remote procedure calls are done through the ORB.

2.4.3.2 ORB

Figure 2.5 illustrates how remote methods are invoked in CORBA. Every CORBA server object has a unique object reference. In order to invoke the methods implemented by this server, a client needs to know this reference. There are several ways the client can retrieve the reference, one of these is to use a Naming Service (the Naming Service is part of the Domain Manager – see figure 2.7). Once the reference is obtained, the client can invoke the operations implemented by the server object. The client stub uses the ORB to forward a request to the server object, through the server skeleton.

The ORB manages all the communication necessary to forward the request from the client's stub to the server's skeleton. The client and server do not need to be connected to the same ORB so the General Inter-ORB Protocol (GIOP) is used to forward the request, from one domain to another. The communication within and between ORB domains is transparent both to the client and the server, in fact the client just needs to know the server's reference and the ORB handles all the communication details (as shown in figure 2.6).

While it might seem that the overhead introduced by CORBA for remote calls causes a large delay and would make it impossible to use for SDR, this turns out not to be true. First there are optimizations for local invocation: in SCA, components often run on the same platform so that it is possible for CORBA ORBs to exchange requests via shared memory rather than via interprocess communication. As Schmidt, et al., note: "In such cases, CORBA ORBs can transparently employ co-location optimizations to ensure there is no unnecessary overhead of (de)marshaling data or transmitting requests/replies through a "loopback" communication device" [22].

2.4.3.3 Naming Service

The Naming Service allows a potential client to retrieve the reference to a server object. The basic function of the naming service is to provide a mapping between names and object references. In order to be invokable, each server object creates an association between a name and its object reference and registers this information with the Naming Service. Consequently, a client that knows the name of an object can retrieve the server's object reference by querying the Naming Service.

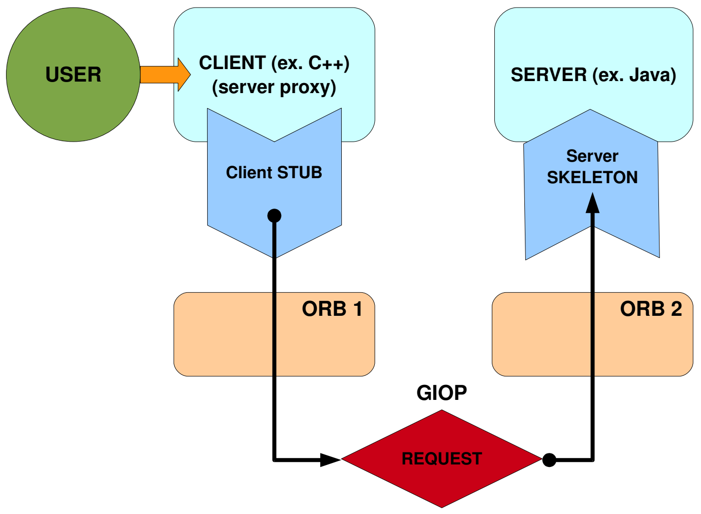


Figure 2.6. The inter-ORB communication using GIOP

2.4.4 Component ports in SCA

Each component has a well-defined set of ports. When instanciating a port it is necessary to specify two properties:

1. **A port can be either an input or output port.**
   Input ports can receive remote procedure calls, while output ports can make remote procedure calls to the components they are connected to. A connection can be done only between an input port and an output port of the same type (it is not possible to connect two input ports or two output ports nor to connect two ports with different types). The type is defined based upon the type of the object that is to be passed across this connection (see the next property). Input ports represent the server side, hence they wait (passively) for remote calls; while output ports are used by clients to send requests to the server. Thus in the context of waveform output ports send data; while input ports receive data. An SCA component can be both a client, a server, or both depending on wheter it is sending data to another component, it is waiting for data on its input ports, or is doing both.

2. **Each port has a well-defined type.**
   To assign a type to a port is equivalent to defining its interface. It is possible to use standard types (such as integer, floating point, etc.) or to create your own custom interface. When creating a custom interface, you have to give a name to the port type and define the available methods. This is done using the Interface Description Language (IDL) (see chapter 4).

Bard and Koverik have stated that IDL and CORBA represent "an important step forward in the ability to develop modular software while encapsulating the internal logic and requiring that each of the developers agree on a set of IDL" [4]. The main advantages of using a set of standard interfaces are:

- that it is possible to **reuse** the same component on different SCA-based platforms and
- interface and implementation become two distinct concepts as two components can communicate using CORBA calls, but can be implemented using two different languages, for example one can be written in Python and the other one in C++. CORBA plays a comparable role to SWIG that was described earlier in section 2.3, but offering the additional features of **language independence** and **distributed computation**.

The expressions "*uses port*" and "*provides port*" are commonly used in SCA development. The statement "*uses port*" requests data or a service from another component, while the statement "*provides port*" indicates that a component returns the requested data or has performed the requested service. Under this model, a SCA component assumes the role of a CORBA client when it is calling through a *uses port* and the role of a CORBA server when it is answering at a *provides port*. Therefore, in the waveform development context *provide port* is a synonym for *input port* and *use port* is a synonym of *output port*.

2.4.5 XML

The eXtensible Markup Language (XML) is used in the SCA to describe component properties and configurations as well as to describe the interconnections between components in a waveform. In the same way as graphical tools are available in GNU Radio to generate the Python code which represents the waveform, in SCA waveforms can be generated with graphical tools by interconnecting components. To do this, the tool needs to know the component's properties (the set of available ports), to configure the component (for example, by setting the gain for an amplifier), and to produce the flowgraph corresponding to the waveform, in a standard format that the SCA infrastructure can understand: all this is done in XML.

XML is a text-based language that utilizes tags to define items, their attributes and values. XML can be used to describe data components, records, and other data structures. XML is a markup language much like HTML. It is extensible because it is not a fixed format like HTML (which is a single, predefined markup language). Instead, XML is actually a metalanguage, i.e. a language for describing other languages. A specific language is called a profile. The SCA Domain Profile is based on XML and will be described in the next sub-section.

2.4.6 The SCA Operating Environment (OE)

The SCA OE consists of the Core Framework (CF), the CORBA ORB, and the underlying operating system. The aim of this thesis is not to describe how to create a SCA-compliant radio system, but to explain how to devolop waveforms on an SCA-based platform, thus only the properties of the SCA OE that are needed to understand waveform development will be presented. Readers interested in details of the SCA OE are referred to [4].

As shown in figure 2.7, a SCA-based radio has conceptually three segments: (i) Waveform Deployment, (ii) Core Framework, and (iii) Domain Profile.

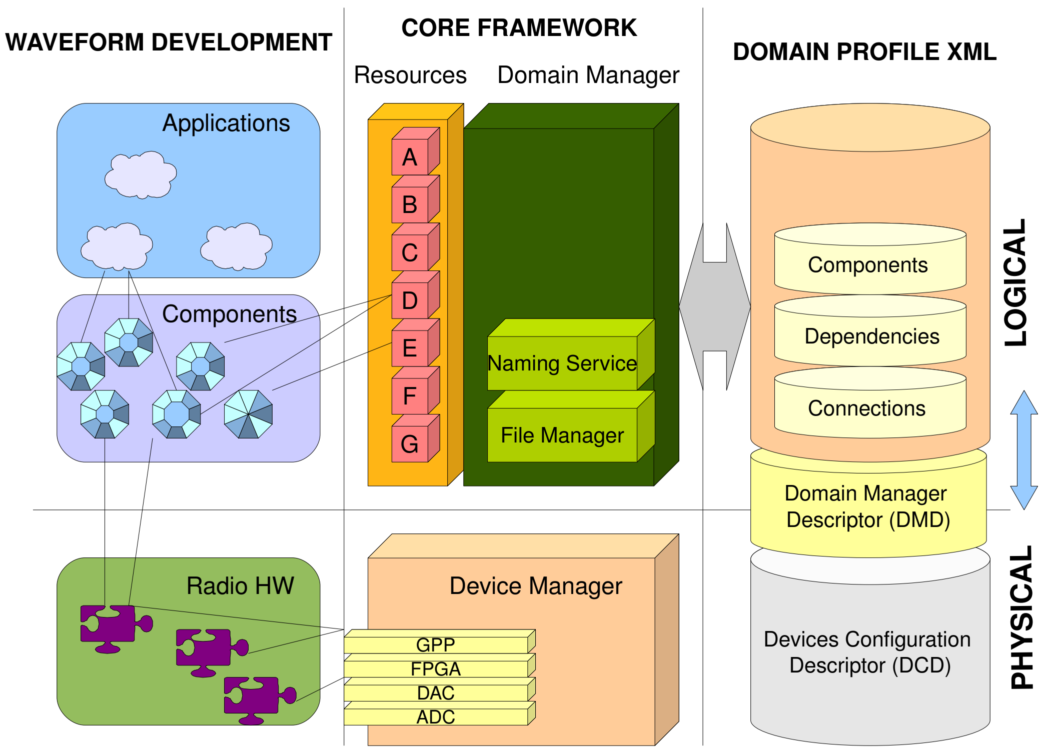


Figure 2.7. Abstraction layers in an SCA system

2.4.6.1 Waveform Deployment

From a physical point of view, a waveform uses some hardware resources (for example the RF front-end available on a USRP daughterboard or the GPP of the computer). When developing a waveform, only the **logical view** is used; the components are software entities that are platform independent, thus they can be reused on different radio systems. It is important to note with the abstraction layer offered by the SCA: waveform development in theory does **not** have to care about the technical details of the underlying hardware platform. However, *in practice* sometimes it is necessary to understand the underlying hardware in detail.

2.4.6.2 Core Framework

The Core Framework segment of the OE includes all the software required to manage the radio system and to deploy applications. The physical view provides high-level management of the physical devices in the radio system. The waveform sees only the logical view, that is a set of resources and services. The resources may include the available components, these components can be addressed through the Naming Service.

2.4.6.3 Domain Profile

The Domain Profile segment consists of a set of XML files that describe the hardware resources within the radio system, the waveform application structure, and the properties of the available components. Details of the XML files are described in [4].

2.4.7 Base Application Interfaces

SCA specifies a set of specific operational interfaces for components: *PortSupplier*, *LifeCycle*, *Port*, *PropertySet*, *TestableObject*, *Resource*, and *ResourceFactory*. Each interface defines the methods used by the Domain profile to install and control the components. The relationship between many of these interfaces are shown in Figure 2.8. Note that a *Resource* is created by a *ResourceFactory*.

2.4.7.1 PortSupplier interface

Provide ports (in the waveform context “provide” means that this is an input port to a component) inherit from the *PortSupplier* interface which defines the `getPort()` operation. This is used (for example by `connectPort()`) to obtain a specific port from a component, as described below.

2.4.7.2 Port interface

An SCA system is composed of components, these components communicate with each other through ports. These ports are inherited from the class *Port*, whose interface provides two operations to set up communications, `connectPort()` and `disconnectPort()`. When a waveform is installed, all the connections between components must be set up. Suppose for example that we want to establish a connection between a use port on component A and a provide port on component B, then during the set up the method `connectPort()` on A's use port is called and a connection with B's provide port is established. This is done by contacting the naming service provided by the Domain Manager to retrieve B's reference, i.e. get the requested port. All this is done automatically by the SCA framework.

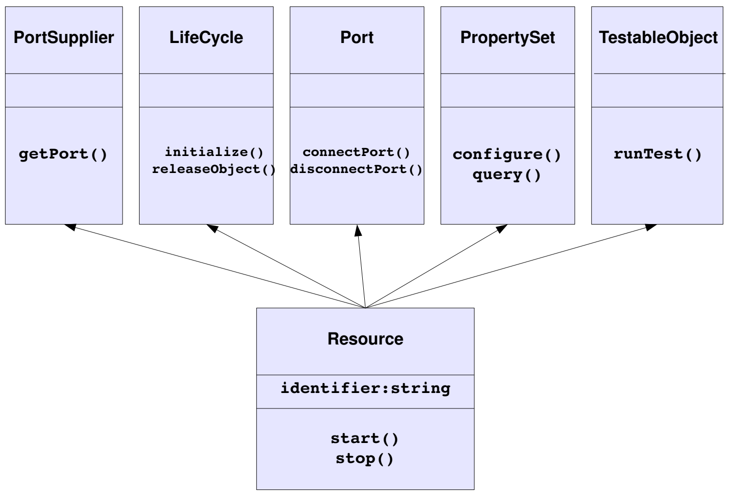


Figure 2.8. Resource Interface UML Diagram

2.4.7.3 LifeCycle interface

The *LifeCycle* interface defines two operations, `initialize()` and `releaseObject()`. The operation `initialize()` is used to set a component to a known initial state, while `releaseObject()` tears a component down when it is no longer needed.

2.4.7.4 PropertySet interface

The *PropertySet* interface is used to access component properties/attributes. It defines two operations: `configure()`, which makes runtime configuration possible, and `query()`, to read the values of the properties of a component.

2.4.7.5 TestableObject interface

TestableObject is inherited by a component to run built-in tests. The system designer can use the `runTest()` operation to test the component, for example to search for errors within the component.

2.4.7.6 Resource

Every software component in an SCA waveform inherits the *Resource* interface; this in turn inherits from *LifeCycle*, *TestableObject*, *PortSupplier*, and *PropertySet* (as shown in figure 2.8). Two operations are provided, `start()` and `stop()`, to be able to start and stop the component (these methods are useful to start and stop the generation of a signal).

2.4.7.7 ResourceFactory

A *Resource* can be created by a *ResourceFactory*. The same *ResourceFactory* should be used to tear down the *Resource*.

2.4.8 OSSIE

Open Source SCA Implementation - Embedded (OSSIE) is being developed by the Mobile and Portable Radio Research Group at Virginia Polytechnic Institute and State University. It is primarily intended for research and education in SDR and wireless communications. The software package includes an SDR core framework based on the Software Communications Architecture (SCA), tools for rapid development of SDR components and waveforms applications, and an evolving library of pre-built components and waveform applications.

Pre-built VMWare images are available on the website and can be used on any operating system that supports the VMWare Player[10]. Otherwise OSSIE can be installed on Linux systems. The user guide [6] describes the installation procedures for Fedora and Ubuntu. A plugin allows developers to develop waveforms and components in the Eclipse IDE[11].

2.4.8.1 The OSSIE framework

The OSSIE framework is an implementation that follows the SCA 2.2 specifications [19]. OSSIE uses omniORB to make CORBA calls. OmniORB is a compliant Object Request Broker (ORB) implementation of the 2.6 version of the Common Object Request Broker Architecture (CORBA) specification.

Several measurements of the impact of the use of CORBA for inter-component communication in SCA have been done. In [2] P. Balister, et al., conclude: "Results show that while CORBA's impact on the performance of the system is measurable, they were significanly lower than the signal processing time required for this waveform. [...] Furthermore, when weigthed against the benefits associated

[10]For information about VMWare see the company's website http:www.vmware.com.

[11]For details of the Eclipse Integrated Development Environment (IDE) see the website http:www.eclipse.org.

with the use of CORBA, namely distributed system support, strong typing, and memory management, it is clear that the use of CORBA is compatible with the overall performance needs of an SDR, even one in a constrained environment".

The configuration file for omniORB specifies the transport rules for CORBA calls, such as Unix domain sockets or TCP/IP. In [26] Thomas Tsou, et al., compared the use of Unix domain sockets as transport layer for omniORB to Internet Inter-Orb Protocol (IIOP) using TCP. They measured the mean latency for the transmission of a sequence of 512 short integers, which equates to 2048 bytes of sample data per call. They conclude that "the use of Unix domain sockets provided a significant performance improvement with a mean latency of 62μs compared 107μs for TCP" [26].

Most of the tutorials available on the OSSIE website concern the development of distributed waveforms, which are deployed on one or more computers and use TCP for the transport layer. However, OSSIE can also be installed on embedded devices (see the master's thesis of Philip J. Balister [3]). Today the goal is to exploit modern multicore architectures, deploying waveform components on different processors in the same computer, hence shared memory and UNIX domain sockets can be used.

2.4.8.2 The OSSIE plugin for Eclipse

OSSIE waveforms can be developed in Eclipse. The plugin available on the OSSIE webside allows the waveform developer to create a project in Eclipse, both for components and waveforms. Eclipse was used for this thesis project; refer to chapter 3 for further information about the OSSIE waveform development using Eclipse.

2.5 The IEEE 802.15.4 protocol

The IEEE 802.15.4 protocol was designed to be used in applications that require simple wireless communications over short-ranges, with limited power, at the cost of low throughput. This standard has been implemented by many manufacturers and it is widely used for wireless communication with sensors and actuators.

2.5.1 Personal Area Network

Many modern devices contain embedded control and monitoring. However, these devices were not necessarily designed to facilitate connectivity, hence applications that wish to utilize many of these devices face an integration bottleneck. These communication links are typically wired. On one hand wires allow reliable transmission of signals from a controller to its peripherals; on the other hand the required wiring raises issues such as cost of installation, safety, and integration difficulty. Wireless technology overcomes some of these obstacles, although it introduces new challenges: propagation, interference, security, radio spectrum regulations, power source & power consumption, etc.

Applications that do not require a wireless communication system with high performance, such as provided by IEEE 802.11 WLAN, can often utilize a low-cost wireless technology. Unfortunately, there are many different low-cost wireless solutions, many of them proprietary, hence a standard solution would increase interoperability among different manufacturers and lead to cost, functionality, and integration benefits to the end consumer.

Gutiérrez, et al. have stated: "A low-rate personal area network (LR-WPAN) is a network designed for low-cost and very low-power short-range wireless communications. This definition is at odds with the current trend in wireless technologies whose focus has been on communications with higher data throughput and enhanced quality of service (QoS)" [12]. Wireless personal area networks (WPANs) are designed to operate in the Personal Operating Space (POS), covering the volume of space around a person of up to 10 m in all directions. Unlike WLANs, connections in WPANs utilize limited (or no) infrastructure.

The IEEE 802.15 (WPAN) Working Group has defined three classes of WPANs that are differenciated by data rate, battery drain, and QoS:

- *IEEE Std 802.15.3*: a high-data WPAN suitable for multimedia applications that require high QoS;
- *IEEE Std 802.15.1/Bluetooth™*: a medium-rate WPAN designed to replace cables for consumer electronic devics (mobile phones and PDAs);
- *IEEE 802.15.4-2006*: a LR-WPAN for aplications with low power and low rate requirements (for example, for Wireless Sensor Networks).

Figure 2.9 illustrates the operating space of the IEEE 802 WLAN and WPAN standards. Table 2.5.1 compares the principal characteristics of standards IEEE 802.11b, IEEE 802.15.1 (Bluetooth™) and IEEE 802.15.4.

2.5.2 LR-WPAN applications

IEEE 802.15.4 was designed to be used in applications requiring simple wireless communication links over short-ranges with limited power and low throuput. These applications can be placed in four classes:

- *Stick-On Sensor*: Wireless sensor networks often belong to this classification; sensors perform monitoring and remote diagnostics and often use battery-powered transceivers.

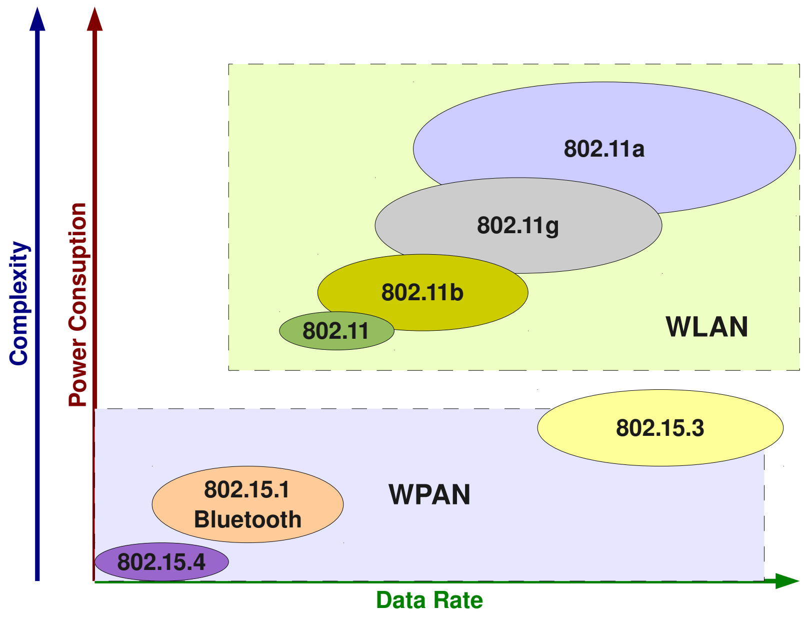


Figure 2.9. Operating spaces of WLAN and WPAN standards

Table 2.2. Comparison of LR-WPAN and other wireless technologies

	802.11 WLAN	**Bluetooth™WPAN**	**Low rate WPAN**
Range	100 m	10-100 m	10 m
Data Rate	54 Mb/s[12]	1 Mb/s	0.25 Mb/s
Power Consuption	Medium	Low-Medium[13]	Ultra Low

- *Virtual Wire*: Monitoring and control applications can only be enabled through wireless connectivity, in places where a wired communication link cannot be implemented, e.g. tire pressure monitoring.

- *Wireless Hub*: a wireless hub acts as a gateway between a wired network and a wireless LR-WPAN network.

- *Cable Replacement*: it is possible to add value to some consumer electronic portable devices by removing wires (for example, Bluetooth™applications for mobile phones).

[12]Today this value ranges up to 54 Mb/s for a single stream and 600 Mb/s with 4 spatial streams (using 40 MHz channels), see IEEE 802.11n-2009.

[13]The power consumption of Bluetooth is not always lower and when a 100m range is desired the power consumption per bit is higher than for WLAN. Refer to [8] for some measurements of Bluetooth vs. WLAN for distributing information.

Gutiérrez, et al. have stated: "IEEE Std 802.11, IEEE Std 802.15.1, and IEEE Std 802.15.3 were created to target specific applications. In contrast, IEEE Std 802.15.4 was designed to address a wide range of applications in different market segments" [12]. In fact IEEE 802.15.4 is primarily an application enabler, since the value to the user will be in the application – rather than in the device's wireless capability. A view of the mobile phone's market can clarify this concept. The design of the first GSM cell-phones focused on the communication capabilities, since the primarily function was to make calls. With new communication devices such as the Apple iPhone, Huawei's 3G USB modules, etc., the value has shifted away from the voice as the main application.

2.6 Python

Python was the programming language used to write component implementations in OSSIE for this thesis project. It is also used in GNU Radio to write the glue logic for interconnecting components, as has been described in section 2.3. This section presents some of its features.

2.6.1 Dynamic programming languages

There is a category of programming languages which share the properties of being high-level, dynamically typed, and open source. These languages have been referred to in the past by some as "scripting languages" and by others as "general-purpose programming languages". David Ascher proposes "the term dynamic languages as a compact term which evokes both the technical strengths of the languages and the social strengths of their communities of contributors and users"[1]. However, this still leaves open the question: what is a dynamic language? For the purposes of this thesis a dynamic language has the following properties: it is high-level, open-source, and dynamically typed.

High-level Modern programming languages inherit most of their properties from several earlier languages. These properties make the programming easier and hide the details of the underlying operative system and hardware. Here we consider the following specific properties:

1. the availability of more abstract built-in data types[14];
2. particular syntactic choices emphasizing readability, conciseness, and other "soft" aspects of language design;

[14]Algol-68 had this in the 1960s.

3. more flexible typed variables, variously referred to as “loosely typed”, “dynamically typed”, or “weakly typed”, in clear opposition to “static typing”;
4. automation of routine tasks such as memory management and exception handling[15];
5. a tendency to use interactive interpreter-based systems over machine-code-generating compiler models[16].

The main reason for these trends is that, as computers become commodities and humans have more to do in the same amount of time, it is imperative that programming languages should support the human programmers and users while exploiting the decreased computational constraints (i.e., using the increased resources to save human resources). Thus, high-level languages enable the human programmer to more rapidly develop applications, by using the resources of the computer to support the user in this task. This leads, generally, to languages that are easier to use, but may be slower to execute. However, compiler technology often allows the performance to be comparable (and in some cases much higher than human generated code).

Open source Open source can refer to different aspects of software design:

- the legal usage of the term “open-source” refers to open source software licenses which encourage code sharing; and
- the methodological usage of the term “open-source” refers to a development model characterized by networks of volunteer developers and by close relationships between users and developers. For this sense of open-source, the reader is referred to [18].

Dynamically typed *Dynamically typed* languages are those languages where the programmer does not need to declare a variable before using it, in fact a variable is created at runtime when a value is assigned to it; unlike *statically typed* languages such as C, where a variable must first be declared together with its type. *Dynamically typed* languages are often also *weakly typed*, such as Python. In contrast, the Java language is *strongly typed* and does not allow implicit type conversion. In Python variables are untyped so that it is not possible to perform type-checking before the program is executed. In addition to dynamic typing, dynamic languages often include other dynamic behaviors, such as loading arbitrary code at runtime and runtime code evaluation.

[15]Lisp had this since the 1950 and numerous languages have had this since at least Ada in the late 1970s.

[16]Lots of systems since the LISP systems of the 1970s support intermixing of interpretive and compiled code.

2.6.2 Rapid prototyping

SCA specifies the IDL interfaces for all components of a waveform *independently* of the language used for the implementation of the component. The OSSIE tool provides support for component implementations in C++ and Python. Python makes it easy to create components with user-defined custom interfaces.

In this thesis project some custom interfaces were defined to enable interaction between OSI layers and the IEEE 802.15.4 standard. The main reason for implementing waveform components in Python was that by default IDL custom interfaces are compiled in Python by OSSIE. Additionally, lots of the desired data structures were already available in Python libraries. It should be noted that the GNU Radio libraries are written in Python, thus they can be reused inside OSSIE components, for example for signal modulation. This enables an OSSIE waveform developer to directly leverage all of the work done by GNU Radio developers at the cost of encapsulating the GNU Radio code into a OSSIE component and writing the requisite IDL to make it available.

Since the aim of this thesis project was to develop a prototype of the IEEE 802.15.4 waveform using OSSIE and there were no specific performance requirements, I could focus on understanding SCA and the design of the waveform. Prior to this project, I had not worked with either SDR or SCA; thus it was initially difficult to have a clear concept of what should be done. For this reason I started "sketching" with Python, the result was that after only two months I had an OSSIE based waveform up and running. This idea of sketching is from the following excerpt from Eric S. Raymond's book "The Cathedral and the Bazaar" [18] (used in the course "Dynamic Programming Languages" at KTH):

> I was taught in college that one ought to figure out a program completely on paper before even going near a computer. I found that I did not program this way. I found that I liked to program sitting in front of a computer, not a piece of paper. Worse still, instead of patiently writing out a complete program and assuring myself it was correct, I tended to just spew out code that was hopelessly broken, and gradually beat it into shape. Debugging, I was taught, was a kind of final pass where you caught typos and oversights. The way I worked, it seemed like programming consisted of debugging.
>
> For a long time I felt bad about this, just as I once felt bad that I didn't hold my pencil the way they taught me to in elementary school. If I had only looked over at the other makers, the painters or the architects, I would have realized that there was a name for what I was doing: sketching. As far as I can tell, the way they taught me to program in college was all wrong. You should figure out programs as you're writing them, just as writers and painters and architects do.
>
> Realizing this has real implications for software design. It means that a programming language should, above all, be malleable. A programming language is for thinking of programs, not for expressing programs you've already thought of. It should be a pencil, not a pen. Static typing would be a fine idea if people actually did write programs the way they taught me to in college. But that's not how any of the hackers I know write programs. We

need a language that lets us scribble and smudge and smear, not a language where you have to sit with a teacup of types balanced on your knee and make polite conversation with a strict old aunt of a compiler.

The Cathedral and the Bazaar, Eric S. Raymond [18]

In my case not only was the project evolving, but the OSSIE tool itself was under continuous development. OSSIE is a very young open-source project and it is significantly updated from one release to the next one, thus changes in the tool also affected the design. SCA itself is more stable, but OSSIE is an incomplete implementation of SCA.

Chapter 3

Method

In this chapter the design properties of OSSIE based waveform development are illustrated in detail, then analyzed. This chapter starts with some considerations about the project organization and the method of working. As explained in the previous chapter, the implementation of the IEEE 802.15.4 standard and realization of a working prototype were important goals, but they are not the main aim of this thesis project. Rather the goal was to analyze the usefulness of the OSSIE tool as a base for future work on SCA waveform development in an open-source environment.

3.1 The waveform structure

The purpose of this section is to introduce the waveform design flow and the main building blocks, i.e. the SCA components.

3.1.1 Introduction

The IEEE 802.15.4 specification describes the physical layer and the MAC sublayer. The set of standards developed by the IEEE 802 working groups differ at the physical layer and MAC sublayer, but share a common interface at the data link layer. This is achieved through the standardized IEEE 802.2 Logical Link Control (LLC) sublayer.

This standard LLC interface was defined before WLANs and WPANs were introduced, so it was basically designed for wired local area networks. Therefore, IEEE 802.15.4 allows the definition of other LLC functions more appropriate for wireless media. Gutiérrez, et al. remark that "The IEEE 802.15.4 MAC sublayer definition contains enhanced functionality normally located in the LLC, making it suitable to be interfaced with the network layer directly, allowing the simple implementation of wireless devices"[12].

Since WPANs require very simple protocols, the complete design of an enhanced

LLC sublayer is out of the scope of this project. However, as suggested by Gutiérrez, et al., the typical use of the IEEE 802.15.4 does not require the LLC sublayer; since the MAC layer can in most cases interface directly with the network layer (note that this is also typically the case for Ethernet – where the only real use of the IEEE 802.2 LLC is to add a few extra (fixed) bytes to the frame header to enable other protocols to utilize their own link layer protocols). For this reason our design will focus on the physical layer and MAC sublayer.

3.1.2 The physical layer and the MAC sublayer

SCA allows a waveform developer to create a waveform by interconnecting components, using a **modular design**. Therefore, the first step is to identify those functions that are logically related in order to group them into the same component. This was easy in our case because the physical layer and the MAC sublayer are well defined in the ISO/OSI model, thus they can be considered as two distinct components.

The physical layer requires some basic methods to interface with the MAC layer. The physical layer component performs the encoding of the frames received from the MAC layer and the decoding of the signal received from the USRP; in addition to this, it is in charge of accessing the USRP and controlling the RF front-end under the control of the MAC layer protocol (for example configuring it in receiving or transmitting mode). So at this very early design step, the physical layer can be split into two sub-components, one in charge of interfacing with the MAC layer and the other hiding all the details about modulation and signal transmission. In future steps each of the components can be split into further subcomponents:

- This **modularization** makes possible the **reuse of library components**. When the functionality of a component is well defined, it becomes easier to find this functionality in the libraries. For this reason it is better to split the waveform into small components with well-defined functionalities, then interconnect components to create more complex functionalities.

- Separating the modulation from the rest of the protocol makes it easy to use different modulation schemes. In the case of many radio communication protocols, the details of the modulation scheme chosen affect only the lowest level of the protocol, specifically the modulation and the demodulation. If this functionality is separated from the rest of the physical layer, then it is possible to change modulation scheme simply by using a different component to realize the modulation/demodulation functionalities. The only requirement is that the components have to have the same interface. Fortunatly, this is generally the case since typically the modulation component receives a stream of values as input and returns a stream of complex values (for example, I-Q might be produced as a stream of complex short integers).

- It is better to split large blocks into smaller components, so that the design is clearer and the functionality can be more easily recognized from the waveform interconnections. In the limit as the data passes through different components, it is possible to understand the signal processing performed to realize the waveform by just looking at the operations done at each stage.

- However, modularization can become a performance problem if the modules are too small, because the number of interconnections within a waveform are proportional to the number of components. As will be discussed in section 3.3.2, complex waveforms may thus require a lot of CORBA calls, slowing down signaling. Standard compiler techniques can be applied to produce a smaller number of components that realize the same functionality, but eliminate many of the inter-component communication steps.

3.1.3 The upper layers

What about the upper layers? The waveform uses TUN/TAP, a virtual kernel network driver available on UNIX systems, to allow the waveform to be seen as a virtual interface (also called a pseudo interface) by the operating system and hence by the network and higher layer protocol stack. TAP emulates an Ethernet device and it operates with layer 2 Ethernet frames. TUN (a contraction of network TUNnel) emulates a network layer device, hence it receives and produces layer 3 packets, such as IP packets. Packets sent by an operating system via a TUN/TAP device are delivered to a user-space program that opens the device. A user-space program may also pass packets into a TUN/TAP device. In both cases, the behavior is the same as sending and receiving packets via any network interface. This means that you can easily set up the routing table to route network traffic to this interface.

The TUN/TAP functionality makes it possible to send and receive IP traffic through the USRP. To realize this functionality it is necessary to add to the waveform a component, whose function is to open the TUN/TAP of the underlying operating system in order to instantiate the waveform as an active interface.

Since IP packets are larger than IEEE 802.15.4 frames, another component is required to fragment IP packets into smaller packets. This component has been associated with the link layer, but it simply splits the network data packet into multiple link layer frames[1]. This is consistent with the earlier statement that the IEEE 802.15.4 standard allows the definition of a LLC sublayer appropriate to the specific waveform.

[1]In this solution the IP stack takes care of the message transfer units (MTUs); however, there are some issues about the time to transmit fragments and the IEEE 802.15.4 superframe, see optimization such as proposed in [28].

3.1.4 Conclusions

After this first step, the waveform structure is as illustrated in figure 3.1; a more accurate description of the implementation is given in section 3.2.

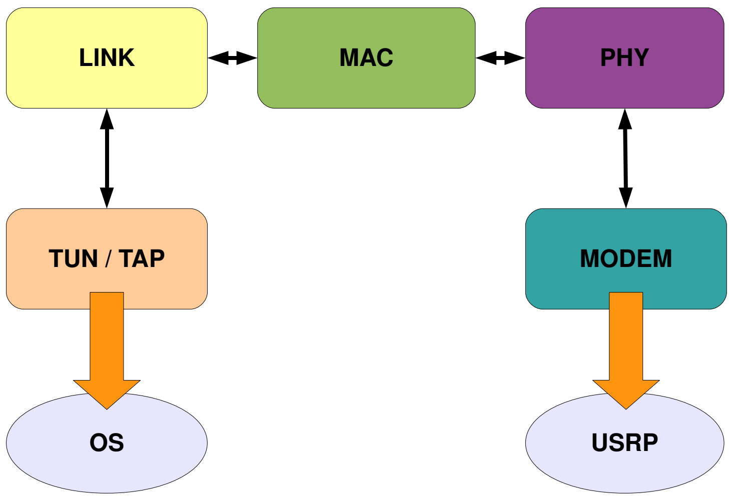


Figure 3.1. The waveform structure

3.2 The UCLA Zigbee library

The introduction in chapter 1 mentioned that there is an existing implementation of the IEEE 802.15.4 protocol. This implementation is called the “UCLA ZigBee PHY” and it was developed by Thomas Schmid at the University of California at Los Angeles (UCLA) [20].

3.2.1 Description

The UCLA Zigbee PHY project realized a GNU Radio implementation of the IEEE 802.15.4 physical layer. The GNU Radio library developed by Thomas Schmid is publicy available and it is compatible with GNU Radio version 3.2.

Once his library (*ieee802_15_4*) is compiled and installed, it provides two GNU Radio components: *ieee802_15_4_mod* and *ieee802_15_4_demod*, which perform modulation and demodulation, respectively. This implementation is described in his technical report “GNU Radio 802.15.4 En- and Decoding” [21]. This implementation is illustrated in figures 3.2 and 3.3.

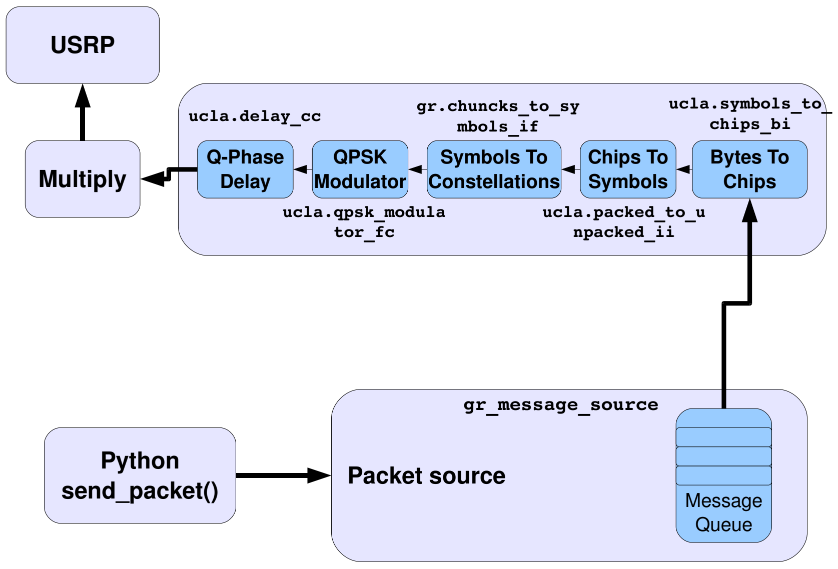


Figure 3.2. Block schema of the modulator implemented in GNU Radio (adopted from [21])

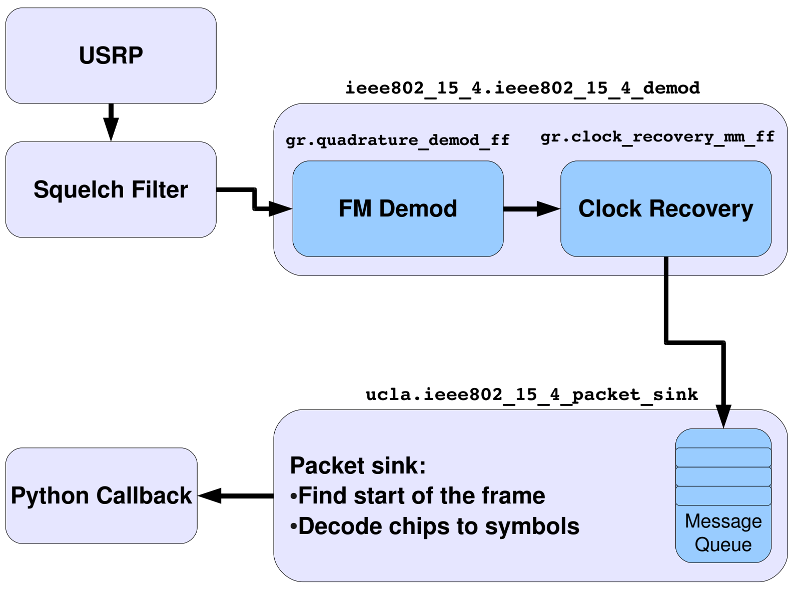


Figure 3.3. Block schema of the demodulator implemented in GNU Radio (adopted from [21])

3.2.2 OSSIE and GNU Radio

OSSIE allows the developer to implement components in Python. As noted earlier, this makes it possible to reuse GNU Radio libraries inside OSSIE components. For this reason I realized the modulation and demodulation in a single OSSIE component: MODEM_USRP (see section 4.3). This component is separated from the physical layer interface. Inside this modem component the existing UCLA Zigbee library is used to modulate and demodulate the signal.

The main problem was that the OSSIE version available when this project began (OSSIE 0.7.4) used GNU Radio 3.1 to communicate with the USRP while UCLA Zigbee PHY requires GNU Radio 3.2. For this reason it was not initially possible to use the UCLA Zigbee library and the OSSIE USRP device (the OSSIE interface to talk to the physical USRP platform) on the same platform. This problem could be solved in four different ways:

1. Use the OSSIE USRP commander to talk to the USRP and rewrite the UCLA Zigbee PHY library so that modulation and demodulation could be performed *without GNU Radio*;

2. Use two computers to run the waveform:

   - one computer, configured with GNU Radio 3.2, could run the modem component; thus the modem component would use the UCLA Zigbee library to modulate and demodulate the signal
   - the rest of the waveform could run on the other computer, configured with GNU Radio 3.1 and the full-featured OSSIE 0.7.4 distribution: thus the USRP commander would run on this computer and control the USRP.

   SCA allows a waveform to be distributed, as the waveform can use the General Inter-ORB Protocol (GIOP) to perform CORBA calls between different CORBA domains. This is rather easy to realize if OSSIE is running in a VMWare virtual machine – hence both implementations can be running on the same physical processor and the GIP traffic will not actually be sent via a network interface, but will simply pass through the loop back interface on the same physical computer.

3. Edit the OSSIE source files and update OSSIE so that it could use GNU Radio version 3.2. Since OSSIE uses just a few GNU Radio libraries, specifically those required to access the USRP – only small changes would be required.

4. Use only one computer and configure the system with GNU Radio 3.2 and OSSIE 0.7.4, but not compile the USRP node for OSSIE. In this case the modem component, beside doing the modulation and demodulation with UCLA Zigbee PHY libraries, would also access and control the USRP using the GNU Radio code.

In the first version of the target waveform the forth strategy was used. This was the most direct method and it allowed the waveform to easily be modified when the new OSSIE version had been released. Although the third solution was interesting, it was estimated that the time to update OSSIE would have reduced the time available for the real focus of the thesis project itself, while clearly this functionality would eventually become available in the next OSSIE release. For this reason I decided to first implement the handshaking protocol in the MAC sublayer and in the physical layer, using the available GNU Radio libraries to send and receive frames through the USRP. Later, after the new OSSIE version was released, I worked on the physical layer, to implement the modulation and demodulation of the signal in a dedicated component and used another component, the USRP_commander available in the OSSIE library, to control the RF front-end of the USRP.

Figure 3.4 represents my first implementation of the IEEE 802.15.4 waveform. The waveform layer shows the SCA components that have been described in section 3.1 and the interconnections between them. On this layer the arrows in the diagram correspond to interconnections between input and output ports, implemented through CORBA calls.

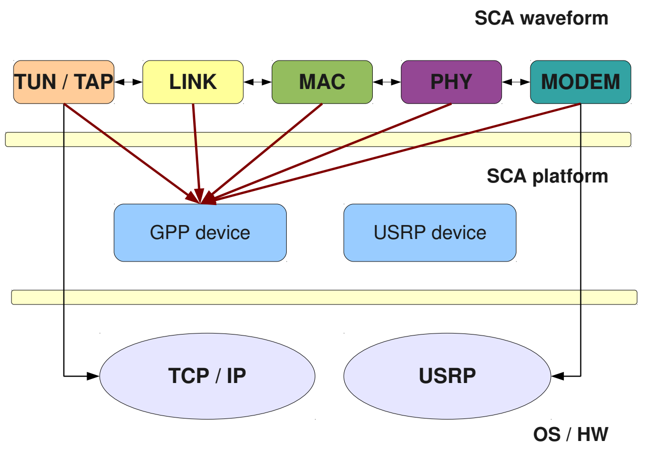


Figure 3.4. Waveform implementation without the SCA platform abstraction layer

All the components are deployed on the GPP node since (as discussed before) the USRP node was not initially available because of configuration problems. The work around was for the modem to directly access the USRP device using the GNU Radio libraries. In the same way, in this first implementation, the TCP/IP connection is not implemented as a node in the SCA platform, but was accessed directly from the TUN/TAP component.

This, of course, limits the abstraction of the underlying operating system and hardware resources. However, the aim of the SCA framework (section 2.4) was to abstract the hardware and software resources available on a platform by implementing some nodes with well defined functionalities. The SCA waveform should run over an SCA platform and the SCA components should not see the hardware resources and the underlying operating system, but rather should interface with the functionality provided by the SCA framework.

In order to hide the underlying OS and hardware, the next step in the design was to create a OSSIE resource which offers the TUN/TAP functionality. In this way it would be possible to hide the implementation details, including details of the underlying operating system, thus improving the portability of the waveform. In fact it would be possible to run the waveform on any platform which provides the TUN/TAP functionality, no matter how it is implemented on the underlying operating system. The resulting waveform has the form shown in figure 3.5.

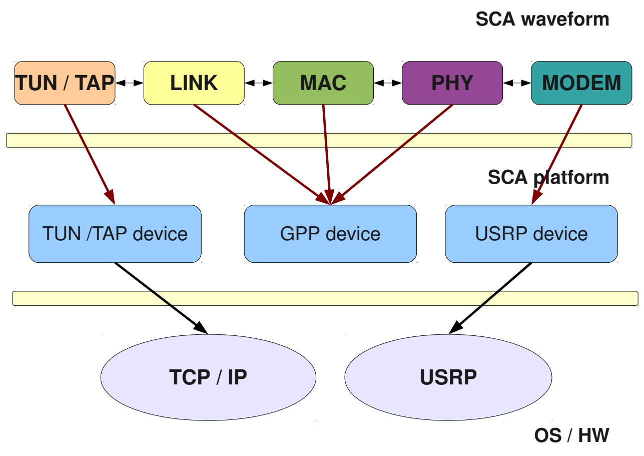


Figure 3.5. Waveform implementation using the SCA platform abstraction layer

3.3 The work method

This section discusses how the project was carried out. Starting with a description of the interface, followed by a presentation of the design method.

3.3.1 Working on the interfaces

One of the most specific aspects of SCA is the interconnection of components, which is done through CORBA. This affects the development process from three aspects:

1. A lot of effort is required to analyze the interfaces, which **precedes** the first design step.

2. Once the interfaces have been decided, then components can be developed in parallel by different teams.

3. A waveform is made up of several components, but they can run on different platforms, in this case the communication is performed through the General Inter-Orb Protocol (GIOP).

Since I carried out the project alone and I ran the whole waveform on my PC, the second and third considerations were not relevant in my case. What is specific for SCA waveform development is that the design process starts from the interface, in fact when you create components using the OSSIE plugin for Eclipse, the first step is to define the **component ports**.

This approach has some similarities with the black/white box model: at the beginning the components can be interpreted as black boxes as only the interface is available outside. The methods available on the interfaces must be carefully selected so that the waveform can be implemented in a second stage by properly interconnecting the components.

This first step was not easy, as the MAC layer requires several methods in order to control all the internal functions; then some problems emerged with the definition of a custom interface in OSSIE. Both of these impediments are discussed later.

As a design choice, I decided to implement a first version with only the data service primitives. This made it easier to develop the component interfaces, but allowed me to understand how SCA works and how OSSIE must be used. Thus the first waveform I implemented was a very simple protocol, with only a few functions within the MAC and the physical layers. However, this design choice allowed me to quickly transmit and receive radio signals through the USRP.

3.3.2 The design method

The approach described above must not be confused with a top-down approach. It is not possible to develop waveforms in OSSIE in a top-down fashion because all the components are on the same level and OSSIE does not support a hierarchy. As shown earlier in figure 3.1, the signal modulation is performed by a specific component on the same level as the physical layer. Since normally modulation is done by the physical layer, it would have made sense to implement it in a subcomponent, but this is not possible within OSSIE. Thus although SCA supports **functional composition**, it does not support functional de-composition – as would be required for top-down design.

SCA waveforms have a flat structure (figure 3.6) and inter-block communication is done through CORBA calls, i.e., handled by the SCA framework. The splitting of SDR applications into components enables scalable systems in that the various components may be deployed on additional processors as needed. Generally, an application composition with many small components increases the probability of being able to reuse the components in other waveform applications. However, the component approach adds CORBA overhead through the processing of the CORBA invocations and format conversions in the system. Also, the approach increases the number of separate processes and threads, which increases context switching in the cases where several components are deployed on the same processor.

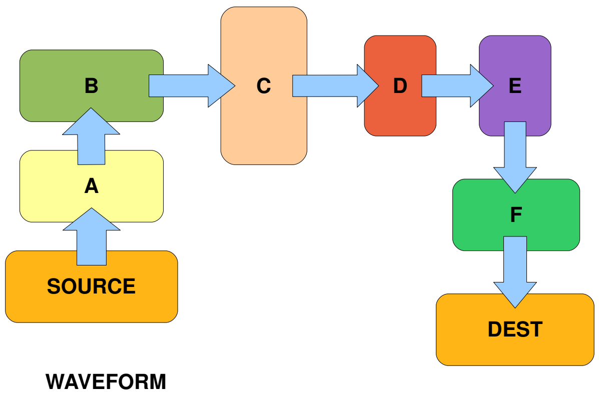


Figure 3.6. No blocks hierarchy in OSSIE waveforms

In [27] T. Ulversøy and J. Olavsson Neset have examined this by using a scenario with one CORBA capable GPP which runs the same functional application, but implemented as a variable number of components. As part of their experiment, other parameters were controlled, such as the workload of components, the size of the data packets transferred between the components and the data rate in the system. The empirical analysis was performed using the OSSIE Core Framework (CF) which uses omniORB. The result was that the processor workload increased as the number of components increased, while the total functional processing work decreased for increasing data packet sizes. Hence, T. Ulversøy and J. Olavsson Neset concluded that “the scalability and reusability benefits that result from implementing the SDR application with a high number of components, must be balanced against the processing efficiency loss that occurs when having to run several components on the same processor”.

Blocks could actually be emulated by the development tool, thus making it possible to design hierarchical SCA waveforms, but unfortunatly this functionality is not yet available in OSSIE. The OSSIE user guide mentions the *Automation Tool* [6], a feature which allows the developer to aggregate waveforms to be built. However, this functionality is still at the experimental stage and only allows aggregating only waveforms, not components.

In contrast, GNU Radio allows the waveform developer to easily connect together smaller blocks in order to obtain more complex functionality. The combined structure can be put into an higher abstraction layer block, that acts as a container and hides the internal implementation and the lower abstraction layer components.

Figure 3.7 is the direct representation of a waveform flowgraph. In this flowgraph there are no limits on the structure hierarchy, hence it is possible to design very complex blocks by interconnecting smaller components. As noted in section 2.3 there is a graphical tool, called GNU Radio Companion (GRC), for creating signal flow graphs and generating flow-graph source code.

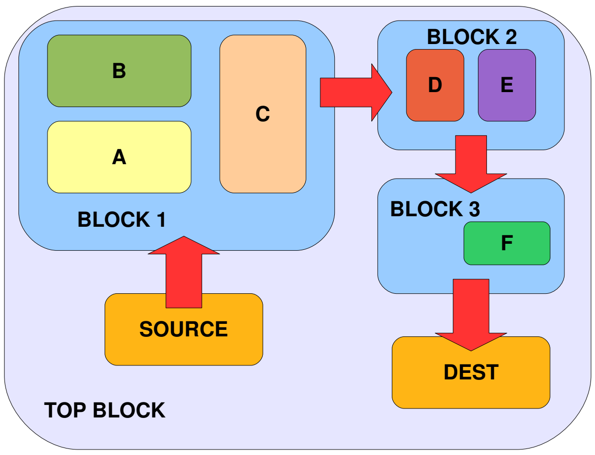


Figure 3.7. Blocks hierarchy in GNU Radio

3.3.3 Conclusions

In conclusion, the interface between components is the most important aspect to take into account when designing SCA waveforms, because the final functionality will be a result of the interconnection between components. Once the interface is agreed upon, then the component can be implemented in different ways to meet different constraints, regarding performance, memory size, and development time (these issues are discussed for each component in the next sections).

SCA allows components to be reused on different platforms, because the SCA framework guarantees a standard set of interfaces that abstract the underlying hardware and operating system. In contrast, GNU Radio has internal dependencies that also have to be satisfied; this may be more difficult to mantain, because all the necessary libraries need to be installed. However, GNU Radio makes it possible to develop components with a hierarchical structure, so that waveform development leads to more flexible and modular results.

3.4 OSSIE components

This section illustrates how waveform components can be created in OSSIE.

3.4.1 The OSSIE plugin for Eclipse

To develop OSSIE components I used the OSSIE plugin for Eclipse. To begin it is necessary to create a new project and select the option "OSSIE component". After giving a name to the project, the OSSIE perspective is opened (figure 3.8).

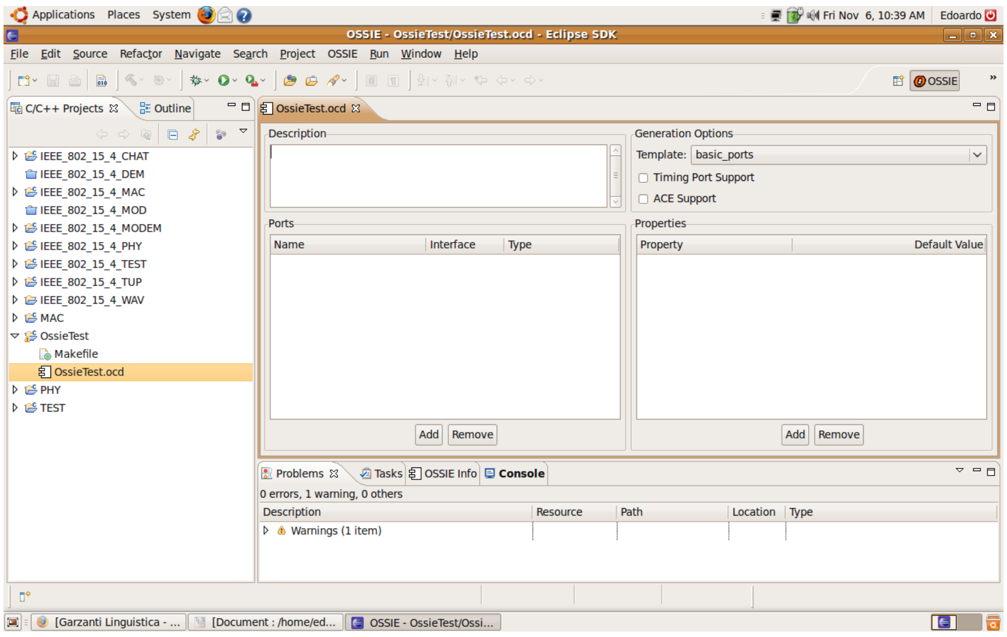


Figure 3.8. OSSIE perspective for creating a new component in OSSIE

We can see four panels that offer different functionalities:

- **Description** - The developer can enter a basic description of the component here (as text).
- **Ports** - Allows for the addition and removal of ports for the component. When you add a port, you have to give a name, set the input or output direction, and specify the type. Each type has a specific interface and provides the methods specified in the corresponding IDL file.
- **Generation Options** - Defines the ports template to be used and indicates if Timing Port Support and Adaptive Communication Environment (ACE) Support are enabled. It is possible to choose among three port templates:

1. *basic_ports*, use standard types and the implementation is written in C++;
2. *custom_ports*, use custom types and the implementation is written in C++;
3. *py_comp*, ports are implemented in Python, using both standard types and custom types.

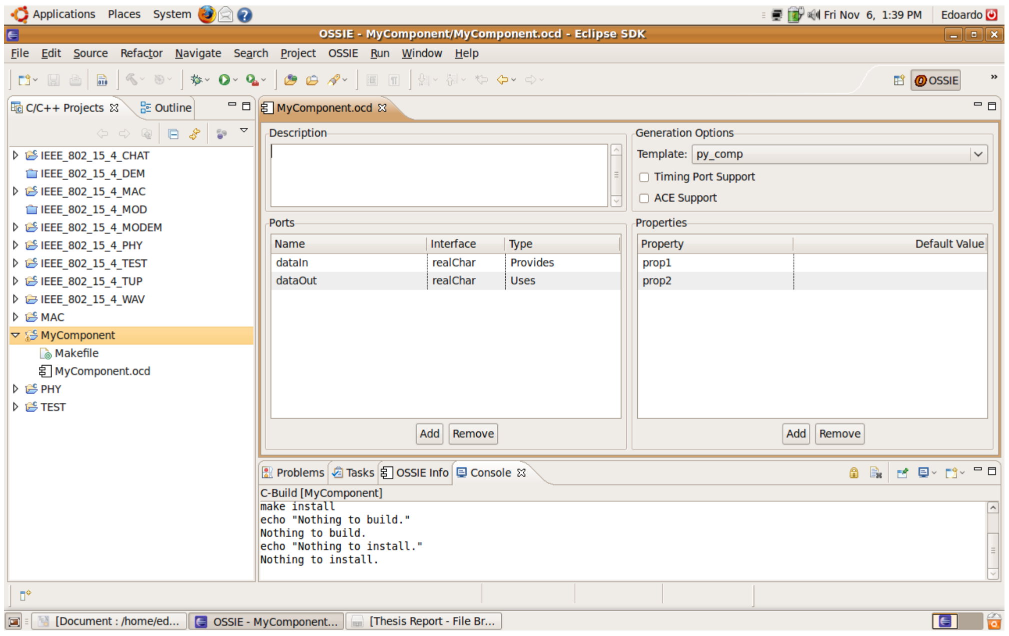
- **Properties** - Allows for the addition and removal of editable properties for the component. When you add a property you have to set some values:

  1. *property name*
  2. *property description*
  3. *value type*
  4. *default value*

All this information is stored into XML files, providing a description of the component for the Domain Profile.

3.4.2 An example of creating a component

To illustrate some issues about OSSIE we will create a new component called *MyComponent*, with the following configuration (see figure 3.9):



Template: py_comp
☐ Timing Port Support
☐ ACE Support

Ports

Name	Interface	Type
dataIn	realChar	Provides
dataOut	realChar	Uses

Properties

Property	Default Value
prop1	
prop2	

```
C-Build [MyComponent]
make install
echo "Nothing to build."
Nothing to build.
echo "Nothing to install."
Nothing to install.
```

Figure 3.9. OSSIE perspective for creating MyComponent

- an input port named "dataIn" of type *realChar*;
- an output port named "dataOut" of type *realChar*;
- a property *prop1* of type *short*, default value 0; the type and the default value can be selected in a window, which will pop up when the button "Add" is clicked[2];
- a property *prop2* of type *float*, default value 2.0;
- the *py_comp* template option has been selected, since all components in the target waveform are to be written in Python.

After saving the current configuration, the component can be generated by selecting "Generate Component" from the OSSIE menu: we can see that several files are generated and stored in the project folder (the contents of these files are in appendix B):

- *MyComponent.prf.xml*, component property file: this XML file contains a list of the component properties, each one associated with a unique ID.
- *MyComponent.scd.xml*, software component descriptor: this XML file describes the component ports and the set of interfaces used.
- *MyComponent.spd.xml*, software package descriptor: this XML file provides a basic description of the component together with some information about the executable file.
- *MyComponent.py*, Python file with port implementations.
- *WorkModule.py*, Python file where processing is done.
- *setup.py*, install script used to copy Python and XML files into the appropriate subdirectories under `/sdr` once the component is edited to provide functionality. This is executed by typing `python setup.py install`.

OSSIE is a young project and even these basic operations have some bugs:

- You need to edit *MyComponent.spd.xml* before the Python component will work properly. Find the XML tag below the tag `<code type="Executable">`. By default the next tag is:

  ```
  <localfile name ="bin/MyComponent"/>
  ```

  This needs to be changed to:

[2]Note that the default value is not shown in the Property window, because of a bug.

```
<localfile name ="bin/MyComponent/MyComponent.py"/>
```

- The class for the input port is called *dataIn_complexShort_i* but it should be *dataIn_realChar_i*
- Some spelling errors exist in the automatically generated comments.
- The methods for receiving and processing data are defined for ports of type *complexShort* (for this reason the `AddData` method receives two signals, I and Q). As the comment suggests, I and Q may have to be changed depending on what data you are receiving (e.g. to bytesIn for *realChar*).
- It is not possible to change the default value for properties by using the Eclipse plugin, rather it is necessary to open the property file (*MyComponent.prf.xml*) and manually change it in the XML code (tag name <value>).

3.4.3 Conclusions

Some properties of the code structure reveal what the OSSIE developers aim to do:

- A class is created for every input port type, these classes inherit from the corresponding IDLs (in this example, the `dataIn` port is derived from *standardInterfaces___POA.realChar*); for this reason the methods defined for the interface must be implemented (initially the *realChar* interface has only the method `pushPacket`).
- A class is created for every output port, all these classes inherit from *CF___POA.Port* which provides the methods `connectPort`, `disconnectPort`, `releasePort`, and `send_data`. In order to establish a connection with another component, the connect method needs to know the IDL:

  ```
  port = connection._narrow(standardInterfaces__POA.realChar)
  ```

- The `start` and `stop` methods are empty by default: the reason is that components become active after they are initialized and configured. In fact, the processing thread for the output ports are created when the ports are instantiated; while the processing thread in the work module, which processes the data received on the input ports, is started when the work module is created, i.e., after the first call to the `configure` method. So after an OSSIE component is created and configured, the component is active and starts waiting for input data to process. It will remain in this state until it is explicitly released, that is until the waveform is terminated and the Application Factory releases all the components. This point is discussed further in the section dedicated to the Assembly Controller, see section 3.6.

- The implementation of the standard Interfaces separates the CORBA handling from the signal processing code. This is performed by buffering data in the `WorkModule` class and subsequently returning the CORBA call without entering the signal processing code. The goal of this structure is to decouple the CORBA call from signal processing [26].

- The OSSIE tool is still young and does not provide much support for various types of data. When creating a port, it is necessary to specify the port type and it is only possible to choose among a limited set of types. The current limitation is that the code generated by the tool does not actually consider the port type, thus the component implementation has to be manually modified or written.

3.5 Component structure

Section 3.4 showed how to create OSSIE components in Eclipse. The code generated by the tool is included in appendix B. This code works well for components with only one input port and one processing thread. Unfortunatly, the design of the physical layer and MAC layer in the target waveform required more complex blocks, with several input and output interfaces and different processing threads. Therefore, I decided to create an internal component structure that would make it easy to customize the functionality and add/remove ports on the interface.

The resulting structure is shown in figure 3.10 and reproduced in appendix C.

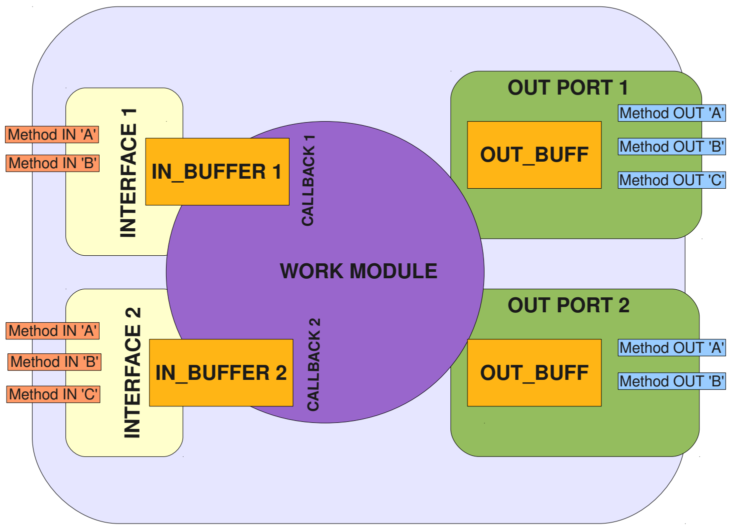


Figure 3.10. Internal component structure

This component is the same component previously described in appendix B, but the code was manually modified (according to the spirit of SCA, the interface is unchanged, but the internal implementation is different). The main properties of the customized code are listed below:

- I created a class `buffer` to easily instantiate buffers inside components. This class inherits from the `Thread` class because, after the `start` method is called, a thread checks for new data in the buffer. Each buffer object has an internal queue, which is created with the Python `Queue` library by specifying the maximum number of elements. Different kinds of subclasses can be instantiated (for example `queue_buffer` or `circular_buffer`) which allow the developer to change the behaviour of the buffer.

- The `MyComponent.py` source file also has some changes. For example, in this implementation the `start` and `stop` methods control the execution of the buffers on the output ports and in the work module. Another difference is that the work module is created when the component is instantiated, rather than when the `configure` method is called.

- The port implementation is not in `MyComponent.py`, but has been moved to another source file (`portImpl.py`) so that new port types can easily be added.

- Output ports are derived from the `use_port` class, so that the methods to connect ports in the SCA framework do not need to be rewritten for every port. Output ports have an optional internal buffer which allows to temporarely store data when sending data to other components.

- When an output port is instanciated, a callback function for the buffer has to be declared: this function is called when new data is put into the buffer, causing the data to be processed. Also the methods on the port interface have to be implemented such that it is possible to send data to a port of the same type on another component.

- Input ports have an interface with the available methods, but the methods are implemented in the `WorkModule` class so that it is easier to exchange data between different processes.

- The `WorkModule.py` source file contains a buffer for each input port and a callback function for each buffer to process input data. Therefore, a call to a method available on the interface corresponds to putting the data into a buffer, this causes a callback function to be invoked and the data retrieved from the buffer and processed. This implementation allows several processing threads to work in parallel within the same component.

3.6 Assembly controller

OSSIE components inherit from the Resource interface. The key functions introduced by the Resource interface are the `start` and `stop` operations, which provide the top-level mechanism for controlling the Resource that is implemented.

According to the SCA specification, once a resource is instantiated and initialized, the `start` operation is used to place it into an operational mode. The `stop` operation halts the processing performed by the Resource. This means that the operational functions of the Resource are stopped, but it does not terminate the Resource or remove it from the system.

As stated in section 3.4, the `start` and `stop` methods are empty by default when the component is first generated by the OSSIE tool. For example, the components available in the OSSIE library become active after they are initialized and configured. They can process input signals until the `releaseObject` method is called. When a waveform is run the `start` and `stop` method can be invoked only on the Assembly Controller while all the other components, already active, remain in an active state waiting for data to process.

A discussion thread found in the OSSIE discuss archive [13] says: “There is the right way, and there is the OSSIE way......” This discussion concerned the design of the assembly controller in OSSIE waveforms. According to the SCA specification, calling start on `CF::Application` shall delegate the implementation of the inherited Resource operations (`runTest`, `start`, `stop`, `configure`, and `query`) to the Application’s Resource component (Assembly Controller) identified by the Application’s Software Assembly Descriptor (SAD). So it would be appropriate for the designated Assembly Controller to in turn call `start` and `stop` on other Resources within the waveform. Therefore, the Assembly Controller, as a component, is not meant to be an active participant within the waveform, but rather it is a controller. The reason a component is designated as the Assembly Controller is so that modeling tools can generate appropriate code to meet the behaviorial requirements of the specification. However, OSSIE does not provide, as of the current release (0.7.4), any support to automatically generate an Assembly Controller!

The idea suggested by Michael Ihde, et al., in the discussion thread is to add a `getResource` method (listing 3.1) to the Assembly Controller so that it is possible to retrieve the reference to all the components in the waveform [13]. This approach assumes that the Assembly Controller knows the names of the other components. However, this seems to be a fair compromise given that the component is already tightly coupled to the waveform specification. OSSIE should automatically generate the code for the Assembly Controller. For example, when the Assembly Controller is generated, all the references to the other components in the waveform should be retrieved so that the waveform can be controlled through the Assembly Controller.

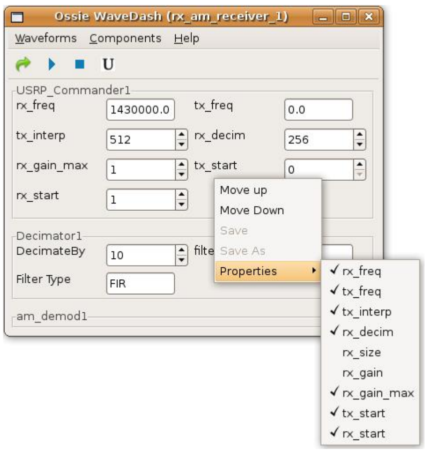
```
def getResource(self, name):
        ns_obj = self.prnt_orb.resolve_initial_references("NameService")
        rootContext = ns_obj._narrow(CosNaming.NamingContext)

        compRef = [CosNaming.NameComponent(x, '')
                        for x in self.naming_service_name.split("/")][0:2]
        compRef.append(CosNaming.NameComponent(name, ''))

        compRsrcObj = rootContext.resolve(compRef)._narrow(CF.Resource)
        return compRsrcObj
```

Listing 3.1. `getResource` method used to retrieve the reference to a resource

Another issue related to the Assembly Controller is the `configure` method. It would be useful to call the `configure` method on the Assembly Controller in order to modify the properties of the waveform. In this approach the Assembly Controller would in turn call the `configure` method on the other components, setting specific properties on each one. This enables the control of the waveform's behaviour at run time through the Assembly Controller. In order to do this, the Assembly Controller must know the list of properties for each component and be able to call the configure method on each one. Currently, OSSIE does not provide any support to do this, but there is a tool, called WaveDash [23], which is an interactive configurable GUI used to work with OSSIE SDR waveforms (figure 3.11) that might be used.


Ossie WaveDash (rx_am_receiver_1)

USRP_Commander1
rx_freq 1430000.0
tx_freq 0.0
tx_interp 512
rx_decim 256
rx_gain_max 1
tx_start 0
rx_start 1

Decimator1
DecimateBy 10
Filter Type FIR

am_demod1

Figure 3.11. WaveDash GUI for configuring a waveform

Through WaveDash, users can install/uninstall, start/stop waveforms, or configure the component properties at run time. This eliminates the need to restart or rebuild the waveform every time a component's properties need to be changed. Users can also customize the GUI to view only the components and properties in which they are intersted. Further, it also allows users to change the widget type of a property making the configuration process more interactive. For further information about WaveDash refer to the OSSIE user guide [6].

3.7 Custom interfaces

When a port is to be defined, the first step is to choose a port type. The type determines the kind of data that can be handled (transmitted or received) through the port and the methods available on the interface, i.e., the methods that can be called by other components. OSSIE comes with quite a large set of predefined port types to manage different kinds of signal, such as *realChar*, *complexShort*, etc. The list is available when you create a component using the OSSIE plugin for Eclipse (see figure 3.8 on page 44).

Using IDL it is possible to describe new custom interfaces, then compile them with an IDL compiler. The predefined data types are well suited for signal processing, but there is no support to manage link layer frames or complex data structures.

3.7.1 The interfaces

The waveform to be implemented requires a MAC layer and a physical layer. Both can be represented as SCA components. Both the MAC and physical layers have to follow the protocol described in the IEEE 802.15.4 standard. Fortunatly, there are already some available methods for each layer, but they must be called in the correct order according to a handshake protocol. The definition of the interface and the methods is done during the first phase of component creation; while the protocol is implemented later, inside the component.

As shown in figure 3.12 six different interfaces have to be defined (the source code is included in appendix D)[3]:

- *sap_app_to_link* and *sap_link_to_app*
- *sap_link_to_mac* and *sap_mac_to_link*
- *sap_phy_to_mac* and *sap_mac_to_phy*

A general requirement for the interfaces in my waveform is that an interface should contain all unidirectional methods between two layers. Looking at figure 3.12 and considering the physical layer and the MAC layer, there are two interfaces between these two components: *sap_mac_to_phy* contains all the methods that the MAC layer can call on the physical layer and *sap_phy_to_mac* contains all the methods that can be called on the MAC layer. Both the MAC and the physical layer have a port with type *sap_mac_to_phy*, but it is an input port on the physical layer and an output port on the MAC layer; correspondingly the physical layer has an output port with type *sap_phy_to_mac* and the MAC layer has an input port *sap_phy_to_mac*.

[3]The Service Access Point (SAP) is a conceptual location at which one Open Systems Interconnection (OSI) layer can request the services of another OSI layer (see section 4.1.1).

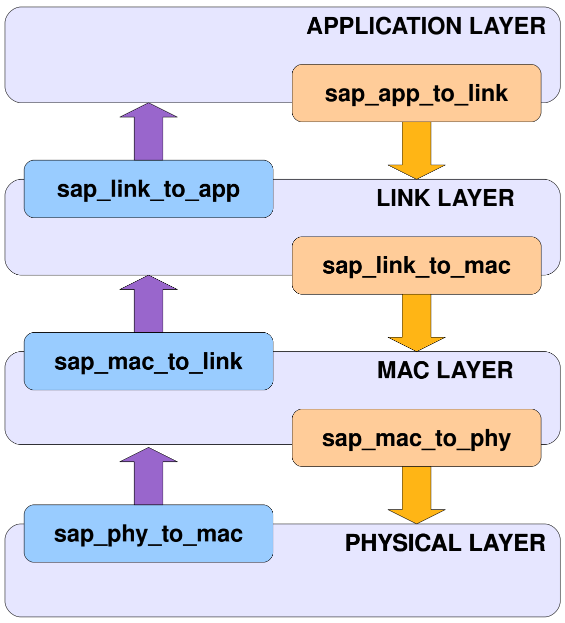


Figure 3.12. Custom interfaces in 802.15.4 waveform

Since in my waveform I need six interfaces, I created the six IDL files listed above (the IDL is included in appendix D). After creating these six files, I need to compile them to produce the stub code. The procedure for this is described in the next subsection.

3.7.2 IDL compilation

In order to be used, the custom IDL interfaces need to be compiled to produce the code that will be used in the various components. The procedure to compile custom interfaces is:

1. You need the source files of the OSSIE distribution you are currently using.
2. Change to the root directory of the folder containing the OSSIE source files (in my case *ossie-0.7.4*)
3. Change to *ossie-0.7.4/system/customInterfaces/*
4. Copy the IDL files describing your custom interfaces into this directory.
5. Edit the file *customInterfaces.idl* to include the IDL files copied in the previous step.

6. Edit the file *Makefile.am* to add the names of the IDL files to *dist_pkginclude-_DATA*. This will enable the compiled interfaces to appear in the OSSIE menu along with the other port types.

7. Execute the command `./bootstrap`, then `./configure`

8. Now you can execute `make` and if everything is OK (no error messages were output during compilation) you can run `make install` as the root user.

9. The custom interfaces have been compiled and are now installed in your OSSIE distribution.

This process is very simple, but is incompletely described in the OSSIE user guide [6]. For this reason, at the beginning of my thesis project, I spent a lot of time trying to understand how to use the OSSIE tool. The lack of documentation is a major disadvantage of the OSSIE environment as compared with GNU Radio that has extensive documentation. I have submitted my additions to the OSSIE documentation as part of my work.

Chapter 4

Analysis

Chapter 3 illustrated the SCA waveform development in OSSIE. The results of this analysis were used to implement the target waveform. In this chapter, a section is dedicated to analyzing each design phase.

4.1 The MAC Layer

The first step in the thesis project was to understand IDL and learn how to create custom interfaces. The general procedure has been presented in section 3.7. Below a practical example is described in detail. We begin with the definition of a custom interface for the MAC layer component in the target waveform. Following this some implementation issues are analyzed, regarding both Python, the language used to write the component code, and the functionality of this component.

4.1.1 Interface

The IEEE 802.15.4-2006 standard specification [24] was the main reference for the MAC layer interface. The IEEE 802.15.4 architecture defines a number of blocks – called layers – in order to simplify the standard. Each layer is responsible for one part of the standard and offers services to the higher layers. This architecture is based on the open systems interconnection (OSI) seven-layer model.

A low-rate wireless personal area network (LR-WPAN) device comprises a physical layer (PHY), which contains the radio frequency (RF) transceiver along with its low-level control mechanism, and a MAC sublayer that provides access to the physical channel. These concepts are illustrated in figure 4.1.

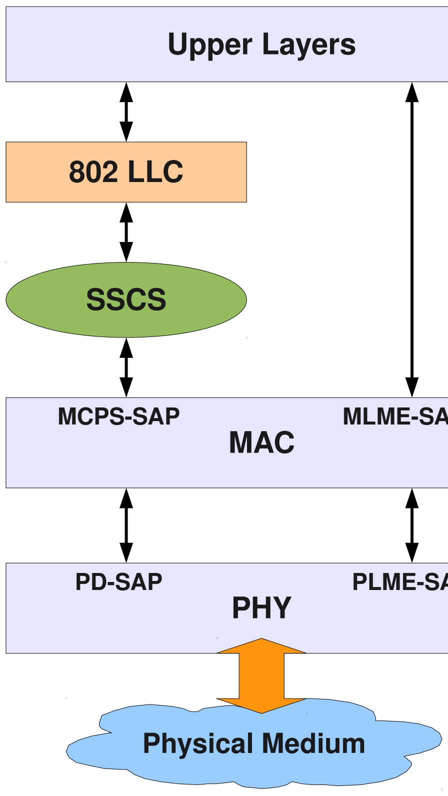


Figure 4.1. LR-WPAN device architecture (adapted from [24])

The MAC sublayer specification divides the functions into two groups, the MAC data service and the MAC management service:

- The *MAC data service* enables the transmission and reception of MAC protocol data units (MPDUs) across the PHY data service.
- The *MAC management service* provides an interface to the MAC sublayer management entity (MLME) service access point (SAP) (known as MLME-SAP); the MLME-SAP allows the transport of management commands between the next higher layer and the MLME.

This thesis project focused on the MAC data service because the goal was to implement only the basic function of the protocol. The MAC data service requires three types of methods:

- *request*: On receipt of a *MCPS-DATA.request* primitive, the MAC sublayer entity creates a frame containing the supplied MSDU and transmits it (the frame is passed to the physical layer).
- *confirm*: The *MCPS-DATA.confirm* primitive reports to the upper layers the result of a request to transfer a MAC service data unit (MSDU).
- *indication*: The *MCPS-DATA.indication* primitive indicates the transfer of a MSDU from the MAC sublayer to the upper layer.

In the target waveform, the layer above the MAC sublayer is represented by the LLC layer (which in our case simply splits the network packet into multiple link layer frames, as said in section 3.1). The primitives listed above represent the function available to the LLC layer. For this reason, the interface between the MAC layer and the LLC layer has to be defined. The IDL files for this interface are `sap_mac_to_link.idl` and `sap_link_to_mac.idl` (included in appendix D). The other interface is between the MAC layer and the physical layer – `sap_mac_to_phy.idl` and `sap_phy_to_mac.idl` (IDL files included in appendix D). Figure 4.2 illustrates how these interfaces are used. The interfaces `sap_mac_to_link` and `sap_link_to_mac` represent the MAC layer functionalities that are available to the upper layer, therefore they are illustrated in this section; while `sap_mac_to_phy` and `sap_phy_to_mac` represent the functionalities exported by the physical layer, they will be illustrated in section 4.2.

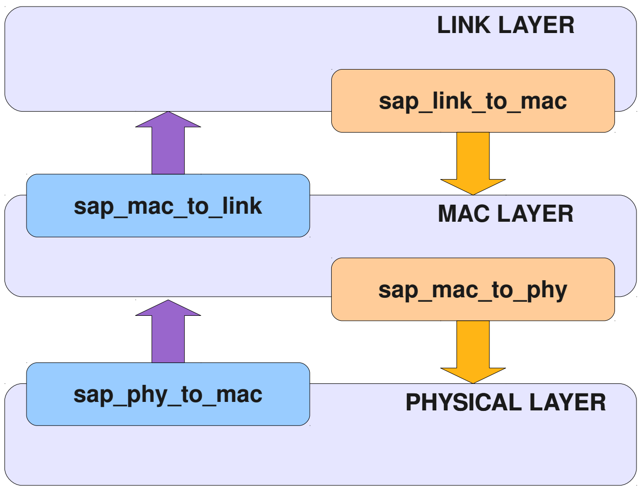


Figure 4.2. IDL interfaces for the MAC layer

The procedure for defining the interfaces for a SCA component is described in the steps below:

1. Pick the components which the selected component is connected to.
2. Consider one adjacent component at a time, for each component follow the steps 'a' to 'c'.
   a Select the services to be implemented between the selected component and the adjacent one, representing each service as a primitive function call.
   b Divide the selected functionalities into input and output primitives for the selected component.
   c Represent the input primitives in an IDL file and the output primitives in another IDL file.
3. Collect all the IDL files created at the previous step and follow the procedure described in section 3.7.2 to compile the interfaces and make them available in OSSIE.

One observation is that during the SCA waveform development process the definition of SCA component interfaces is flexible and the creation of SCA custom interfaces is easy. Given a protocol specification, its interfaces can be directly translated into IDL files. The only limitation was the lack of clear documentation for this process using the design tool, i.e., OSSIE allows creation of custom interfaces but the procedure to do so was initially not clear.

4.1.2 Functionalities

As said in the previous section, this thesis project focused on the MAC data service. For this reason, most of the functionality consists of creating frames, packing the received information into frames, and sending the created frames to the physical layer. Hence, we will begin with an analysis of the MAC frame structure, followed by an examination of the service primitives that can be applied to these frames.

4.1.2.1 MPDU structure

Each MAC frame consists of the following basic components:

- A MAC header (MHR), which includes frame control, sequence number, address information, and security-related information.
- A MAC payload, of variable length, which contains information specific to the frame type. Acknowledgment frames do not contain a payload.
- A MAC footer (MFR), which contains a Frame Check Sequence (FCS).

The format of a generic MAC frame is illustrated in table 4.1. Some fields, such as the source and destination addresses, have variable length. The number of octets used to represent this information is specified in the *frame control* field (see table 4.2).

Table 4.1. MAC frame format

Octets: 2	**1**	**0/2**	**0/2/8**	**0/2**	**0/2/8**	**0/5/6/ 10/14**	**variable**	**2**
Frame Control	Sequence Number	Destination PAN Identifier	Destination Address	Source PAN Identifier	Source Address	Auxiliary Security Header	Frame Payload	FCS
		Addressing fields						
MHR							**MAC Payload**	**MFR**

Table 4.2. Format of the Frame Control field

Bits: 0-2	**3**	**4**	**5**	**6**	**7-9**	**10-11**	**12-13**	**14-15**
Frame Type	Security Enabled	Frame Pending	ACK Request	PAN-ID Compression	Reserved	Destination Addressing Mode	Frame Version	Source Addressing Mode

The 16 bits of the *frame control* field are used to:

- define the frame type;
- specify the length of the addressing fields;
- set control flags to activate/deactivate some functionalities.

Refer to the IEEE 802.15.4 standard [24] for additional information about each of these fields and the meaning of different values in these fields.

4.1.2.2 Data service primitives

The MAC layer in the target waveform supports *data* and *acknowledge* frames. According to the standard specification, the primitives to be implemented are:

- *MCPS-DATA.request*
- *MCPS-DATA.indication*
- *MCPS-DATA.confirm*

MCPS-DATA.request is an input primitive for the MAC layer. The semantics of this service primitive are illustrated below. The *msdu* parameter contains the data to be transmitted. The MAC layer, on receipt a a *request* from the upper layer, creates a data frame and copies the *msdu* into the frame payload. All the other fields in the frame must be properly set, according to the values of the *request* parameters; the *request* parameters also indicate the number of octetets required to represent these values. When the frame is ready, it can be sent to the physical layer using the available methods provided by the PHY interface (this interface is described section 4.2).

MCPS-DATA.request (	SrcAddrMode,	DstAddrMode,	DstPANId,
	DstAddr,	msduLength,	msdu,
	msduHandle,	TxOptions,	SecurityLevel,
	KeyIdMode,	KeySource,	KeyIndex)

MCPS-DATA.indication is an output primitive for the MAC layer. The semantics of this service primitive are illustrated below. The *MCPS-DATA.indication* primitive is generated by the MAC sublayer and issued to the upper layer on receipt of a data frame from the physical layer. The received data frame is parsed according to the bits of the frame control field, which determine the length of the addressing fields and of the security options in the frame header. The information retrieved from the frame fields is then copied into the parameters of this primitive and passed to the upper layer.

MCPS-DATA.indication (	SrcAddrMode,	SrcPANId,	SrcAddr,
	DstAddrMode,	DstPANId,	DstAddr,
	msduLength,	msdu,	mpduLinkQuality,
	DSN,	Timestamp,	SecurityLevel,
	KeyIdMode,	KeySource,	KeyIndex)

MCPS-DATA.confirm *MCPS-DATA.confirm* is an output primitive for the MAC layer. The semantic of this service primitive is illustrated below. The *MCPS-DATA.confirm* primitive is generated by the MAC sublayer entity in response to an *MCPS-DATA.request* primitive. The *MCPS-DATA.confirm* primitive returns a status of either `SUCCESS`, indicating that the request to transmit was successful, or the appropriate error code (refer to the IEEE 802.15.4 specification [24] for the status values).

MCPS-DATA.confirm (	msduHandle,	status,	Timestamp)

4.1.2.3 Operational mode

Figure 4.3 illustrates the order in which the MAC data service primitives must be used in order to successfully transfer data between two devices.

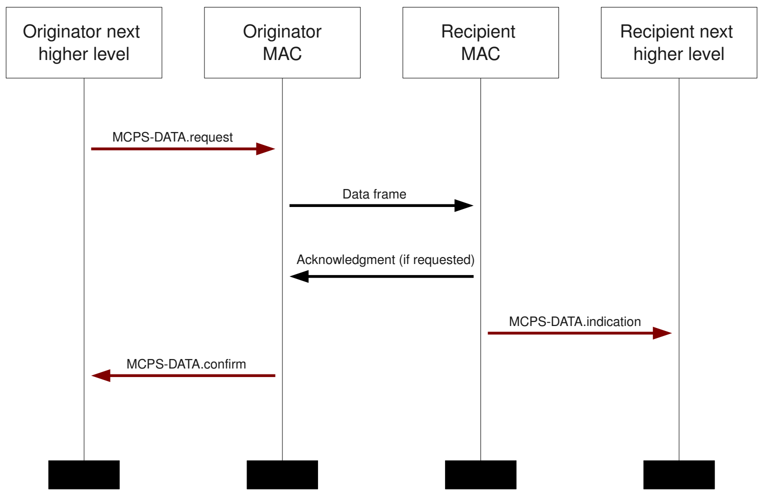


Figure 4.3. Message sequence chart describing the MAC data service (adapted from [24])

4.1.3 Component implementation

OSSIE allows the waveform developer to write component implementations in C++ and Python. For the implementation of the MAC layer component in the target waveform, Python has been used for two reasons:

- The processing required at this level is very limited, compared to components in charge of modulating and demodulating the signal. The overhead of using an interpreter-based language for the implementation of the MAC layer does not affect the overall performance.
- Although the target waveform implements only the data service, using Python the functionality of the MAC layer component can be rapidly extended or modified.

To generate the Python code, the option *py_comp* is selected in OSSIE when defining the component (see section 3.4). The component ports use the interfaces described above. The internal architecture is the same as presented in section 3.5.

What is specific to the MAC layer component is the marshaling of the information to be transmitted and demarshaling of the frames received from the physical layer. This is done using a python class called `frame`. The `frame` class provides methods for creating frames, checking frame integrity, and retrieving information from frames.

The frame is considered as an array of bytes (a string) and the information fields are accessed with the methods `pack` and `unpack` offered by the python module `struct`. This module performs conversions between Python values and C structs represented as Python strings. It uses format strings as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

For the de-marshaling process the *frame control* field has to be read first, since it has a standard layout and contains information about the format of the following addressing fields. The format information is then used together with the `unpack` method to retrieve the values of the addressing and security fields. Similarly, for the creation of frames, the `pack` method is used and the bits in the *frame control* field are set according to the format used to represent the fields in the frame header.

4.1.4 Simulation and final considerations

The OSSIE development tool does not provide, as of the current release (0.7.4), any testing technique for OSSIE components. Therefore, an *ad hoc* technique has been used, inherited from the VHSIC hardware description language (VHDL, VHSIC: very-high-speed integrated circuit).

In VHDL, the circuit to be tested is referred to as unit-under-test (UUT) and it is considered as a black box, that is only the input and output connections can be accessed. VHDL allows the circuit designer to test the UUT by applying some signals to the input connections and observing the signals on the output connections. This can be done automatically by creating a *testbench* component, a circuit entity[1] without input or output connections. Inside the *testbench* component, the UUT is instantiated and its external ports are connected to internal signals. A processes controls these signals and generates a sequence of *stimuli*. Another process observes the values of the output signals of the UUT and compares them with the expected values (according to the design specification).

Similarly, a *testbench* OSSIE component (referred to as TEST component) was created for testing the target waveform. This is illustrated in figure 4.4. The advantage of this technique is that the same TEST component can be used for testing the lower layers of the target waveform. Figure 4.3 earlier illustrated how the data service primitives available on the MAC layer interface must be called by the upper layer in order to successfully transfer data between two devices. Here our TEST component represents the layer above the MAC layer, thus it must call

[1] *Entity* is a technical word in VHDL used to refer to a digital circuit, which represents the same concept as *component* in SCA

the *request* primitive on the MAC layer interface and verify that the MAC layer component returns the correct values with the *indication* and *confirm* primitives. This results in the *ad hoc* testing technique shown in figure 4.5.

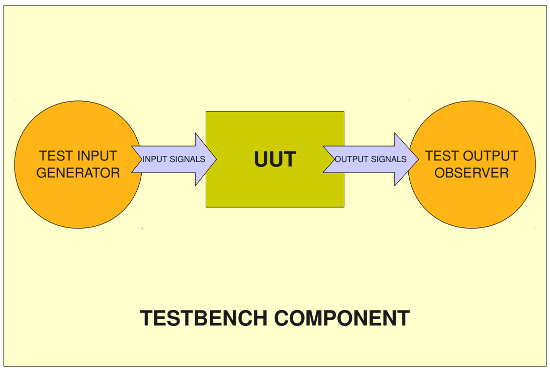


Figure 4.4. Testbench component for simulation of digital circuits in VHDL

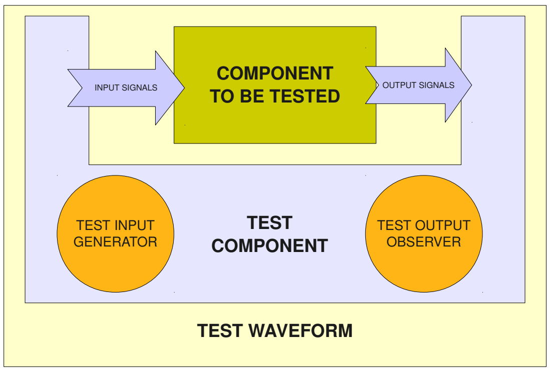


Figure 4.5. *Ad hoc* tecnique for testing OSSIE component

At this stage of the thesis project only the MAC layer component is available. Hence the test waveform uses the TEST component and two MAC layer components. The two MAC components are interconnected through the interfaces *sap_mac_to_phy* and *sap_phy_to_mac* (described in section 4.2): the output ports of one MAC layer component are directly connected to the input ports of the other one. SCA makes it possible to abstract the underlying OSI levels and to test the correct functionment of the MAC layer component (specially in terms of message sequence and marshaling/demarshaling of the information).

The TEST component uses the *sap_link_to_mac* and *sap_mac_to_link* interfaces (the IDL files are included in appendix D) to communicate with the MAC layer component. For example, the TEST component can request *MAC1* to send some data to the other device and wait for a *confirm* from *MAC1*; on the other side, the TEST component waits for an *indication* from *MAC2* containing the data supplied to *MAC1*. This is illustrated in figure 4.6.

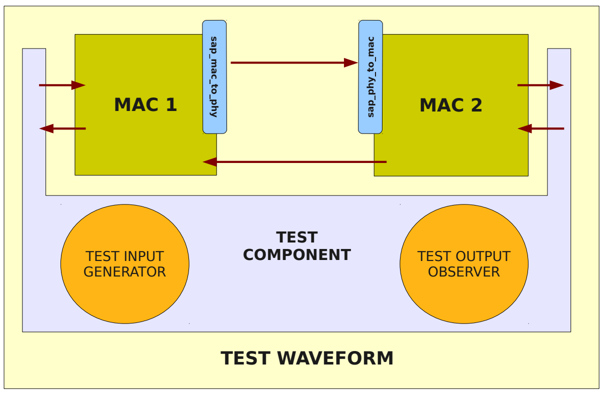


Figure 4.6. Waveform for testing the MAC layer component

Using this model, it is possible to test different scenarios and functions of the MAC layer by simply changing the implementation of the TEST component. Note that there is no need for a physical layer component to do testing at this stage; in fact, the MAC components can be connected using the stubs and scheletons available on the interfaces *sap_mac_to_phy* and *sap_phy_to_mac*. In a next phase, the same TEST component will be used to test the lower layers of the protocols; in fact, the components implementing the lower layers will be inserted between the two MAC layer components, in the same order as they appear in the OSI model.

4.2 The Physical Layer

The implementation of the physical layer of the target waveform was the second step in this thesis project. As said in section 3.1, modulation and demodulation were delegated to separate components, thus the physical layer in the target waveform represents an interface for sending and receiving frames. Therefore, the functionality implemented in this component is minimal.

The design method for this component is the same used for the MAC layer component (section 4.1), thus this section focuses on the features specific to the physical layer; while the reader can refer to the previous section for a more detailed description of the design method.

It is important to note that the IEEE 802.15.4 standard specifies four different physical layers, with different operating frequency ranges and modulation schemes. The physical layer implemented in this thesis project uses 2450 MHz direct sequence spread spectrum (DSSS) PHY with offset quadrature phase-shift keying (OQPSK) modulation.

4.2.1 Interface

The physical layer of the IEEE 802.15.4 standard provides an interface between the MAC layer and the physical radio channel, via the RF hardware represented by the USRP in the target waveform. Like the MAC layer, the physical layer provides two services, accessed through two service access points (SAPs): the data service, accessed through the physical layer data SAP (PD-SAP), and the management service, accessed through the physical layer management entity service access point (PLME-SAP).

In the target waveform, the PD-SAP is implemented in the PHY component, thus it is described in this section. The PLME-SAP, instead, is in charge of configuring the RF hardware of the USRP, using the GNU Radio libraries. Since the PLME-SAP is (partially) implemented in the lower layer components, it is illustrated in the next sections.

The IDL files *sap_mac_to_phy.idl* and *sap_phy_to_mac.idl* (included in appendix D) represent the functions the physical layer makes available to the MAC layer. These interfaces export the same primitives as the MAC layer (*request*, *indication*, and *confirm*). These primitives are described in the next section.

The PHY component has also to interface with the lower layer components, as they are in charge of modulating/demodulating the signal. Therefore, this component also uses the *realChar* standard interface, available in the OSSIE library. The *realChar* output port sends the data to be modulated to the next component; the *realChar* input port, instead, receives the demodulated frames (see figure 4.7). In this implementation, the frames are seen as messages consisting of a set of bytes, because the PHY layer is only transmitting when there is a frame to transmit.

4.2.2 Functionalities

The physical layer component has very little functionality in the target waveform, only creating PHY protocol data units (PPDUs). However, it has been included in order to make it possible, as future work, to extend its functionality.

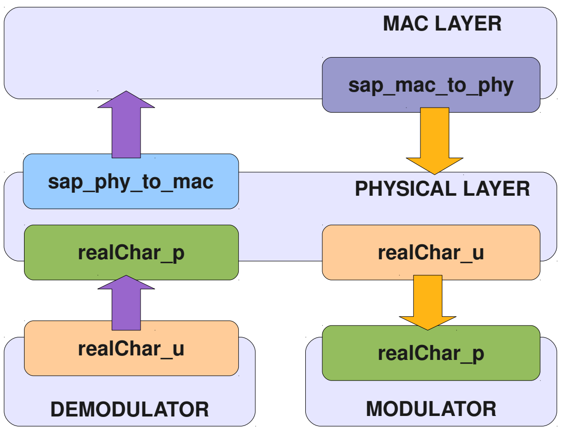


Figure 4.7. IDL interfaces for the physical layer

4.2.2.1 PPDU structure

Each PPDU packet consists of the following basic components:

- A synchronization header (SHR), which allows a receiving device to synchronize with the bit stream; the start-of-frame delimiter (SFD) octet indicates the end of the SHR and the start of the frame.
- A PHY header (PHR), which contains the frame length; since 7 bits are used to represent this information, the maximum frame size is 128 bytes.
- A variable length payload, which carries the MAC sublayer frame.

The size of the preamble and of the SHD depend on the frequency range and the modulation scheme used. The PPDU structure is formatted as illustrated in table 4.3; the size, in number of octets, of the SHR field refers to the PHY used in the target waveform, i.e., the 2450 MHz DSSS PHY employing OQPSK modulation.

Table 4.3. The PHY protocol data unit (PPDU) format

Octets: 4	**1**	**1**		**variable**
		(7 bits)	(1 bit)	
Preamble	SFD	Frame length	Reserved	PSDU
SHR		**PHR**		**PHY Payload**

4.2.2.2 Data service primitives

The PD-SAP supports the transport of MAC protocol data units (MPDUs) between peer MAC sublayer entities. The physical layer provides the following data service primitives to the MAC layer:

- *PD-DATA.request*
- *PD-DATA.confirm*
- *PD-DATA.indication*

PD-DATA.request is an input primitive for the physical layer. The *PD-DATA.request* primitive is generated by a local MAC sublayer entity and issued to its PHY entity to request the transmission of a MPDU. The PHY first constructs a PPDU, containing the supplied MPDU, then transmits the PPDU. Once the PHY entity has completed the transmission, it will issue a *PD-DATA.confirm* primitive with a status of `SUCCESS`. The semantics of the request service primitive are illustrated below.

PD-DATA.request (psduLength, psdu)

PD-DATA.confirm is an output primitive for the physical layer. The *PD-DATA.confirm* primitive is generated by the PHY entity and issued to its MAC sublayer entity in response to a *PD-DATA.request* primitive. The *PD-DATA.confirm* primitive will return a status of either `SUCCESS`, indicating that the request to transmit was successful, or an error code. The semantics of this service primitive are illustrated below.

PD-DATA.confirm (status)

PD-DATA.indication is an output primitive for the physical layer. The *PD-DATA.indication* primitive is generated by the PHY entity and issued to its MAC sublayer entity to transfer a received MPDU. The semantics of this service primitive are illustrated below.

PD-DATA.indication (psduLength, psdu, ppduLinkQuality)

4.2.3 Implementation

Since only the PHY data service is implemented in the PHY component, most of the functionality consists in creating the PPDU, that is adding the synchronization header (SHR) and PHY header (PHR) to the MPDU to be transmitted. All the considerations of the MAC layer component in the previous section (4.1) are still valid for the PHY component, thus the PHY component was also implemented in Python. As for the MAC component, a python class `packet` is in charge of creating a PPDU on receipt of a MPDU; both PPDUs and MPDUs are considered as data units (messages consisting of a set of bytes).

4.2.4 Simulation and final considerations

The same technique illustrated in the previous section (section 4.1) was used to test the physical layer component, called PHY. The test waveform is illustrated in figure 4.8; in this case, the two PHY components can directly communicate with each other, while the modulation/demodulation and the transmission of the radio signal are not yet implemented. The TEST component is the same used for testing the MAC component, but some tests were added to verify the correct behaviour of the lower layer components (in this waveform the PHY component) in some particular cases; for example, when sending a MPDU larger then the maximum size or with size zero (the PHY component should not transmit any PPDU). By adding tests at each design step, at the end of the thesis project the TEST component provided an extensive set of tools for debugging the target waveform. In addition, due to the partitioning of the testing - different groups could be designing, implementing, and testing their components in parallel.

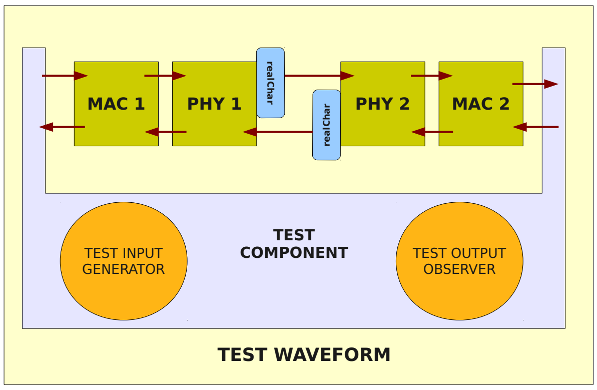


Figure 4.8. Waveform for testing the PHY layer component

The TEST component must verify the correct behaviour of the target waveform according to the initial specification. Therefore, the internal implementation of the TEST component is a formal representation of the waveform requirements. It would be useful, as future work, to study some more advanced technique to automatically generate the implementation of this component: given the requirements of the target waveform represented in a standard format, a tool could translate this formal specification into a TEST component for the waveform. Hence, the design would start with testing: this approach is called *test-driven development* and requires developers to create automated unit tests that define code requirements before writing the code itself.

As said before, the IEEE 802.15.4 standard provides four different PHYs, with different operating frequency ranges and modulation schemes; while the interface to/from the physical layer is always the same, the underlying modulation/demodulation components and RF hardware may be different. For this reason, the PHY component in the target waveform is only an interface, providing to the MAC layer methods to transmit MPDUs. The advantage of SCA is that it is possible to change modulation schemes and operating frequency range by using alternative components for these functions; while the upper layers in the waveform do not need to be changed. The 2450 MHz DSSS PHY with OQPSK modulation is the modulation scheme used in the target waveform.

Sections 4.3 and 4.4 illustrate two different approaches to the design of the modulation/demodulation components, along with two possible ways of sending/receiving data from the USRP. The goal is to analyze the possibility of using the existing GNU Radio libraries inside OSSIE components to perform signal processing. One solution will acces the USRP using the OSSIE device interface while the other one will represent the USRP as a GNU Radio source/sink component and will access it without CORBA.

4.3 The GNU Radio solution

This section illustrates how to use existing GNU Radio components in OSSIE.

4.3.1 Description

At the beginning of this thesis project, a basic GNU Radio implementation of the target waveform already existed, which used the *UCLA Zigbee PHY* library. The GNU Radio *UCLA Zigbee PHY* library was presented in section 3.2. Figure 4.9 illustrates the structure of the physical layer in that implementation. The python class `phy` is derived from `gr.top_block`, as it is the container for the flow graph.

The GNU Radio components used in the waveform are listed below:

- `gr.message_source`, `gr.multiply_const_cc`, `gr.pwr_squelch_cc`, `usrp.sink_c`, `usrp.source_c`: these components are available in the standard GNU Radio library (the latest release during the thesis project was version 3.2.2);
- `ucla.ieee802_15_4_packet_sink`, `ieee802_15_4.ieee802_15_4_demod`, `ieee802_15_4.ieee802_15_4_mod`: the *UCLA Zigbee PHY* library provides these components.

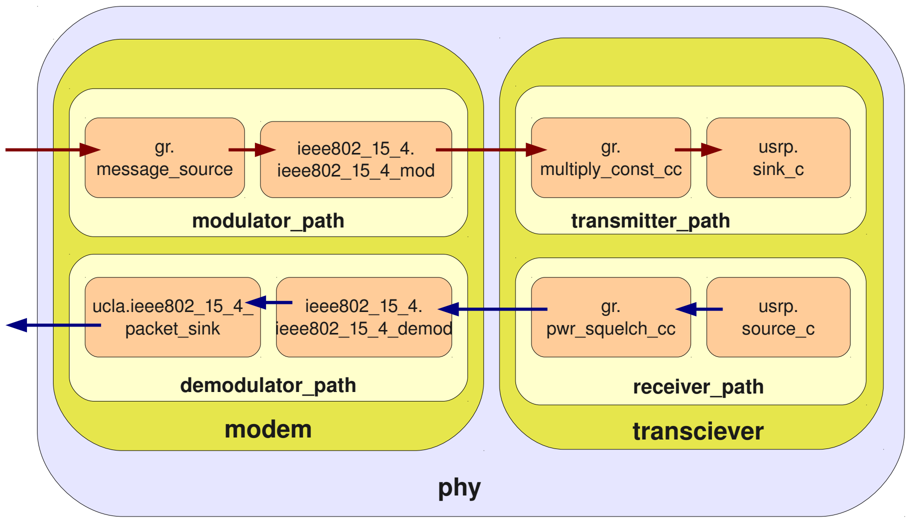


Figure 4.9. Physical layer structure in the GNU Radio IEEE 802.15.4 waveform

Since all the components are already available in libraries, the PHY component (`phy`) can be easily implemented by interconnecting these components; this can be done by writing some Python code or using the graphical tool GRC (as described in section 2.3). The `phy` component has three functions:

1. as a **sender**, modulating frames to be sent and transmitting the modulated signal through the USRP;
2. as a **receiver**, reading the signal from the USRP and demodulating the received frames;
3. as a **controller**, configuring the RF-hardware on the USRP.

Sender The `phy` component provides a method, `send_packet`, which allows the upper layer to transmit a PPDU (of type *string*, passed as argument to `send_packet`) with the USRP.

Receiver The constructor method for the class `phy` requires a callback function as parameter. The callback function must have a parameter of type *string*. Inside the `phy` component, a process running in a separate thread checks for available frames in the `ucla.ieee802_15_4_packet_sink` queue: when a frame is available, the callback function is called and the frame is passed as argument. In this way, the upper layer can receive frames from the USRP.

Note that the USRP does not have any concept of frame. In fact, the component `usrp_source_c` simply reads a sequence of bytes from the buffer internal to the USRP and passes this data to the next component as a stream of samples; the frame are actually built by the component `ucla.ieee802_15_4_packet_sink`, which searches for the PHY synchronization header to know when a PPDU starts and then uses the length field in the PPDU header to know when the frame (the content of the PPDU payload, refer to table 4.3 on page 66) ends.

Controller As said in section 4.2, the PHY component in the target waveform does not implement the management service for the physical layer described by the IEEE 802.15.4 standard specification. Some of these functionalities, in fact, are implemented inside the `phy` GNU Radio block. The GNU Radio classes `usrp_sink_c` and `usrp_source_c` provide methods to configure the USRP, for example setting the operating frequency and the amplifier gain.

4.3.2 OSSIE component

The goal is to create a OSSIE component, called MODEM_USRP. Inside this component we instanciate the existing GNU Radio `phy` block. Hence, in order to send and receive PPDUs from the USRP, the ports of the OSSIE components would simply use the functionality available in the existing GNU Radio block.

As port type, the *realChar* type available in the standard OSSIE library was used, since the PPDUs are represented as streams of bytes. The option *py_comp* was selected when creating the component, to make it possible to use the `phy` python class (section 4.4 will show that it is also possible to create C++ GNU Radio blocks in order to use C++ code for implementing OSSIE components).

Figure 4.10 illustrates the internal structure of the MODEM_USRP OSSIE component. Since the `phy` block already has internal structures for buffering data, this OSSIE component does not contain any buffer on the ports. In fact, the `send_packet` method available in the `phy` class creates an object of type `gr.message` and copies the PPDU into it; then this object is inserted into a first-in-first-out (FIFO) queue before being modulated. Similarly, the queue included in the GNU Radio component `ucla.ieee802_15_4_packet_sink` implements the buffering needed for the output port.

Some properties are defined for this component, such as the *frequency* (by default 2.45 GHz), the *gain*, and the *usrpId* (used to select the target USRP, if several USRPs are connected to the same computer). The *usrpId* property must be specified when the component is created, since it is passed as parameter to the constructor method of the `phy` class and it is used to instantiate the USRP sink/source components. While for the *frequency* and *gain* properties, the corresponding get/set method in the GNU Radio `phy` class can be called also at runtime. As seen in section

3.6 (figure 3.11 on page 51), the tool WaveDash allows the waveform designer to configure the components while the waveform is running.

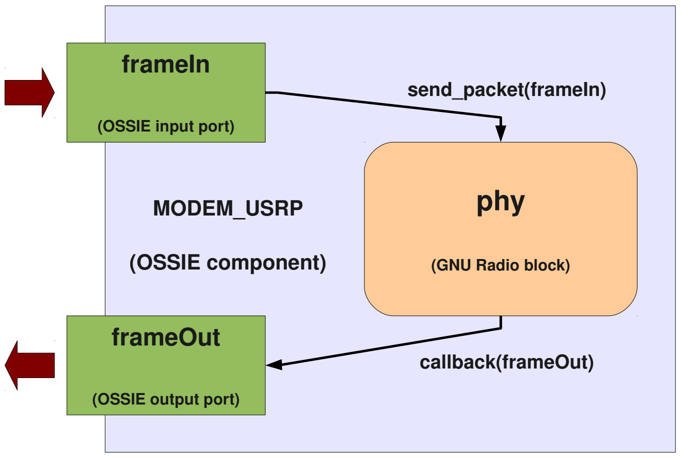


Figure 4.10. GNUARDIO_MODEM component structure

As noted in section 3.3.2, SCA waveforms have a flat structure. However, the development method presented in this section shows that it is possible to implement a hierarchical structure by instanciating GNU Radio components internal to an OSSIE component.

4.3.3 Test

The TEST component described in sections 4.1 and 4.2 can be used to verify the correct behaviour of the MODEM_USRP component. The waveform is configured as illustrated in figure 4.11. Two USRPs are connected to the computer where the waveform was deployed. The components MODEM_USRP1 and MODEM_USRP2 are configured with *usrpId* 0 and 1, respectively; while the *frequency* and the *gain* properties are set to the same values (2.45 GHz and 20 db).

In order to execute this test, a OSSIE waveform must be deployed on a node. The OSSIE plugin for Eclipse provides four possible nodes and one of them must be selected when the waveform is created. The nodes used in this thesis project are either:

- `default_GPP_node`, the OSSIE components run on the CPU of the computer;
- `default_GPP_USRP_node`, the OSSIE components run on the CPU of the computer, but can also access the USRP.

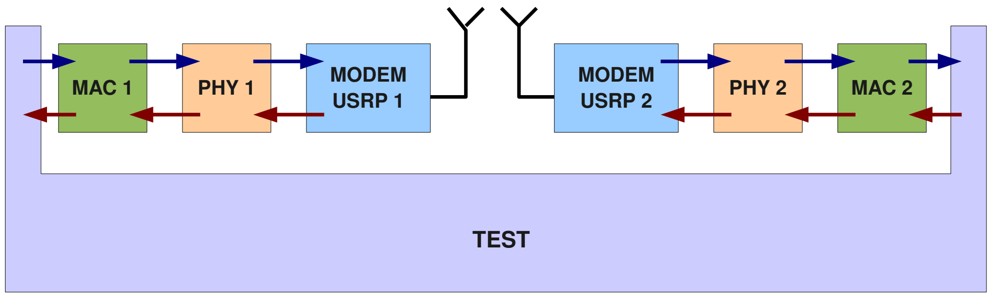


Figure 4.11. Test waveform for MODEM_USRP component

The MODEM_USRP OSSIE component accesses the USRP internally, as an internal subcomponent, rather than as an OSSIE device; hence, the waveform presented in this section must be deployed on a `default_GPP_node`.

In sections 4.1 and 4.2, the simulated waveform used a virtual channel, while this realization of the waveform actually uses the physical wireless channel. Note that the same TEST component used for simulation can be also used to test the real wireless communication. This is another advantage of using the *ad hoc* UUT technique that we have used.

4.3.4 A chat application

The MODEM_USRP component, together with the MAC and PHY components, implement a complete receiver and transmitter. To complete the testing process we designed an application component to connect to the MAC layer component. This specific OSSIE component implements a chat application.

The chat application uses the *Tkinter* python module to show a graphical user interface (GUI). The user can:

- set the destination address,
- write the messages to be sent,
- read the received messages.

The OSSIE component has two ports, which use the interfaces *sap_link_to_mac.idl* and *sap_mac_to_link.idl*; hence, the CHAT component can be connected to the MAC component and use its data service primitives. The waveform is illustrated in figure 4.12. When a message is typed in, the CHAT component calls the *request* method on the MAC interface to send the message; while on receipt of a message, the message is displayed via the user's GUI.

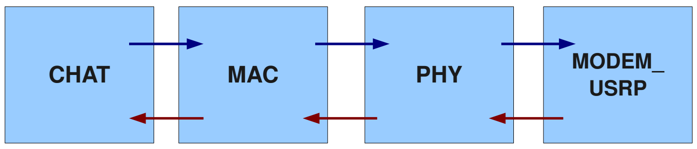


Figure 4.12. OSSIE waveform using the components MODEM_USRP and CHAT

To test the MODEM_USRP component, the waveform shown above was used together with the existing GNU Radio implementation of the same waveform (illustrated in figure 4.13). The test configuration was as follow:

- two USRPs connected to the same computer;
- the GNU Radio waveform using USRP 0 (*usrpId*=0), destination address set to 1;
- the OSSIE waveform using USRP 1 (*usrpId*=1), destination address set to 0;
- in both waveforms, the operating frequency was set to 2.45 GHz and gain set to 20 db.

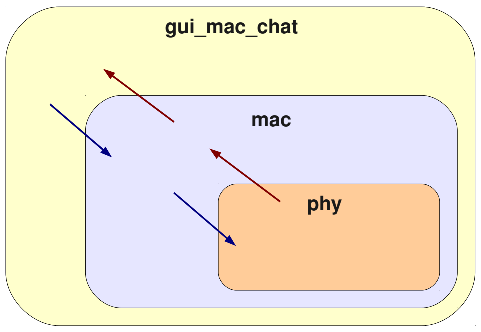


Figure 4.13. GNU Radio implementation of the chat waveform

4.3.5 Results

The implementation of the MODEM_USRP component was straight forward and it was possible to run the target waveform in a short time. The design method consisted in defining a OSSIE interface for the GNU Radio block. The mechanism for passing data from the OSSIE input ports to the GNU Radio block was by inserting the received data into the GNU Radio queues. Conversely, when some data is available in the output queue of the GNU Radio block, this data is passed to the output ports of the component through a callback method.

The solution illustrated in this section utilizes the USRP via a GNU Radio component (*usrp_sink_c* and *usrp_source_c*). As a result, the USRP is accessed within a OSSIE component. As stated in section 2.4, the aim of SCA is to abstract the underlying hardware platform, allowing waveform development *without* having to worry about the technical details of the physical radio hardware. This approach makes SCA components platform independent, that is they can run on different SCA platforms. However, in this case the OSSIE component described in this section directly accesses a hardware resource, the USRP, hence this component will not be independent of the underlying platform. It should be noted that this component could be replaced with another component to realize this protocol operating on a different physical radio – with a different modulation scheme.

The goal in the next phase of this thesis project was to modify the waveform in order to access the USRP as a SCA resource. This is possible because OSSIE provides a SCA device interface for the USRP, but requires the components implementing the signal modulation and demodulation be modified in order to be connected to the USRP interface via the SCA device interface.

4.4 The OSSIE solution

The implementation illustrated in the previous section allowed the waveform developer to use existing GNU Radio blocks in OSSIE. However, this leads to the USRP being used as a component rather than being accessed as an SCA device. The goal of the next design step, illustrated in this section, is to modify the previous waveform – specifically, the MODEM_USRP component – in order to access the USRP through the OSSIE device interface.

4.4.1 Description

Figure 4.10 (page 72) illustrated the internal structure of the MODEM_USRP component. The transmitter and receiver functionalities are separated, thus the signal processing performed on the transmitter path is independent of the signal processing on the receiver path. Therefore, this component was split into two OSSIE components:

- **MODULATOR**: implementing the modulator path and the transmitter path;
- **DEMODULATOR**: implementing the receiver path and the demodulator path.

Next, the `usrp_sink_c` and `uspr_source_c` components were removed from the design, in their place the OSSIE interface for the USRP device was used. The waveform to be implemented is illustrated below in figure 4.14.

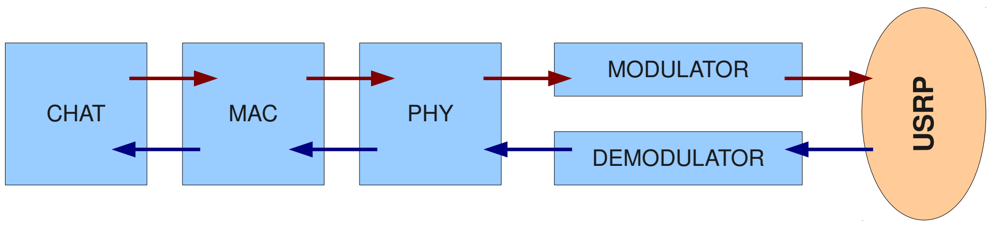


Figure 4.14. MODULATOR and DEMODULATOR components in the target waveform

The MODEM_USRP component was split into two components for two reasons:

- as seen in section 3.1, modularization makes possible the reuse of components;
- two different design methods were used in this thesis project. Here one method will be illustrated for the the MODULATOR and the other one for the DEMODULATOR.

4.4.2 MODULATOR component

The MODULATOR component implements both the modulator and the transmitter path. The difference from the previous solution using the GNU Radio is that the block `usrp_sink_c` must be removed and replaced with another sink; in this case realized as a GNU Radio block implementing a queue. This queue works as a buffer for the output signal (represented as a stream of pairs of shorts) to be sent via the SCA device interface to the USRP. Figure 4.15 illustrates the internal structure of the MODULATOR component. The `symbol_sink` component does not exist in the GNU Radio standard library. Therefore, a GNU Radio library – `mylib` – was created; the C++ block for the `symbol_sink` component was created and included in `mylib`. The details of how this was done is described next.

4.4.2.1 How to write a signal processing block

The title of this paragraph refers to a guide available on the GNU Radio website (see [5]) with the same title. As said in section 2.3, if you need to build a block that is not (yet) available in the GNU Radio library, you have to create it. C++ is the language used to create signal processing blocks in GNU Radio and the guide describes the procedure to do so in detail.

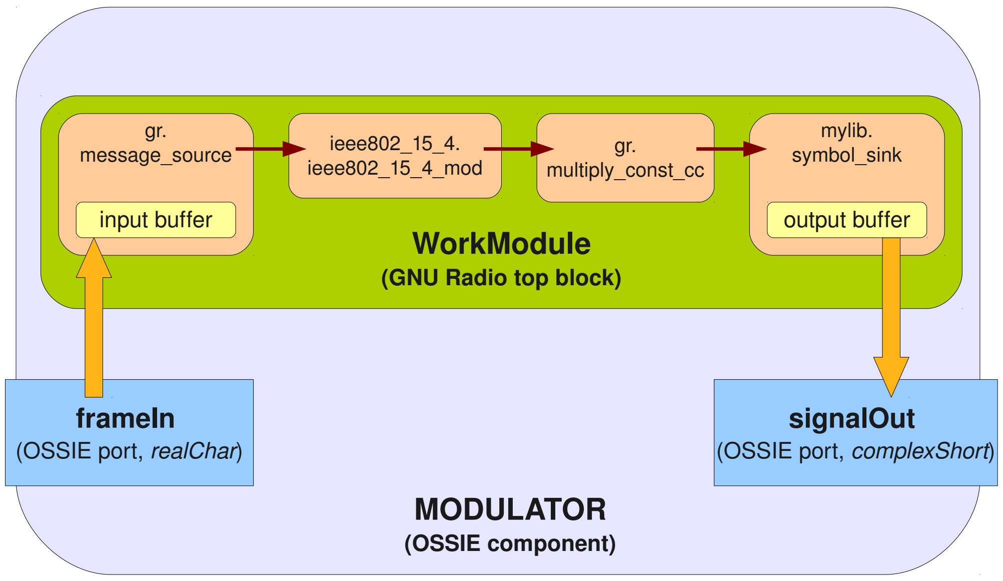


Figure 4.15. Internal structure of the MODULATOR component

To implement the component `symbol_sink`, the design of `gr.message_sink` was adapted by making some changes:

1. the input port of `symbol_sink` is defined to be of type `gr_complex` in order to receive the modulated signal;

2. the internal queue stores arrays of floating point numbers instead of strings;

3. a method – `get_short_I_and_Q_vectors` – allows the top block to retrieve data from the queue;

4. the data returned by `get_short_I_and_Q_vectors` is an object of type *std::pair* containing two vectors of shorts; therefore, it was necessary to write some wrap code in order to make the SWIG library pass this object to the top block as a Python tuple `(I, Q)`.

This component was included in a library called `mylib`. Following the guide [5] cited above, the `mylib.i` file was edited, then compiled. The `symbol_sink` component, implemented as a Python module, can be called as `mylib.symbol_sink` by importing `mylib`.

4.4.2.2 Component structure

The MODULATOR component was designed as an OSSIE component with the following properties:

1. Python implementation: by implementing it in Python, it is possible to write Python code for interconnecting GNU Radio components;
2. OSSIE standard types for the ports: the input port must be able to receive the PPDUs to be modulated, thus the input port was defined to use the type *realChar*; while the output port must be connected to the USRP interface, hence it is defined as the type *complexShort*;
3. as shown in figure 4.15, the `WorkModule` class represents the GNU Radio top level block and this class is derived from the class `gr.top_block`;
4. `WorkModule`, implementing a GNU Radio waveform, already includes input and output queues to buffer data; therefore, no additional buffers are used in the OSSIE component;
5. on receipt of a PPDU on the input port, this data is inserted into the `gr.message_source` component; since the PPDU is represented as an array of chars, a `gr.message` object is created calling the method `gr.make_message_-from_string` and then inserted into the queue;
6. a thread inside `WorkModule` checks for available data in `symbol_sink` and a callback function is used to pass this data to the output port.

4.4.2.3 Considerations

The method used to design the MODULATOR component is easy and flexible. Once the custom library `mylib` was created, the sink/source components could be instantiated in different waveforms, making it possible to use GNU Radio libraries inside OSSIE components to perform signal processing. The DEMODULATOR component was initially implemented in this way, therefore a `symbol_source` component was also created and included in `mylib`. However, another method to implement the DEMODULATOR will be illustrated in the next section.

The disadvantage of this method is that it reduces the portability of OSSIE components. In fact, the implementation illustrated in this section for the MODULATOR and DEMODULATOR components introduce the following dependencies:

- Python 2.5
- GNU Radio 3.2.2
- UCLA Zigbee PHY library
- `mylib` library

In order to be able to run this waveform, the platform must be configured with all the libraries and programs listed above, in addition to the OSSIE distribution.

4.4.3 DEMODULATOR component

The development method illustrated in this section aims to increase the portability of the OSSIE components, while still using GNU Radio libraries to perform signal processing.

4.4.3.1 Description

For the definition of the interface, the considerations in the previous section are still valid for this component:

- since the DEMODULATOR must be connected to the OSSIE device interface for the USRP, the input port is defined to be of type *complexShort*.
- as the demodulated frames are represented as arrays of bytes, the output port uses the *realChar* type.

Also the internal architecture (illustrated in figure 4.16) is similar to that shown in the previous section. However, this OSSIE component was implemented in C++. To generate the C++ interface the option *basic ports* was selected for compilation (see section 3.4). Next we will described how to implement this component in C++.

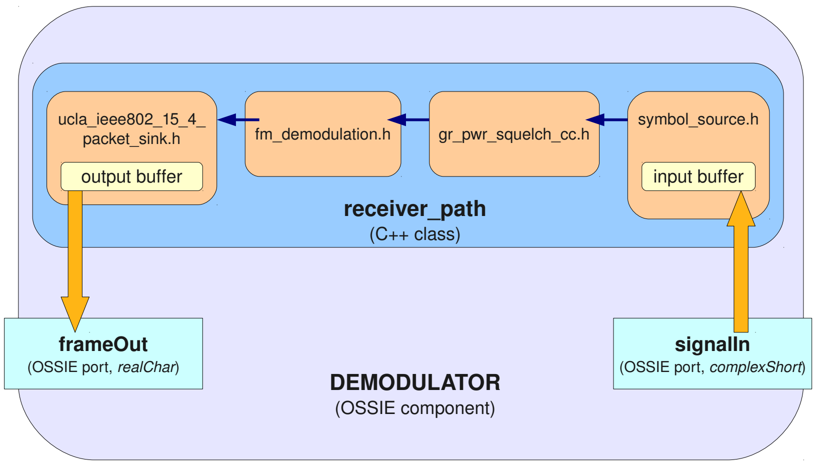


Figure 4.16. Internal structure of the DEMODULATOR component

4.4.3.2 GNU Radio C++ libraries

For the transmitter path in the MODULATOR component (previous section), the standard approach for developing GNU Radio components was followed. The basic processing blocks are implemented in C++ (for the MODULATOR, the `symbol_sink` component) while higher layer blocks can be implemented in Python to interconnect basic blocks (for example, the `WorkModule` class). The design of the DEMODULATOR component does not require any Python code.

The GNU Radio library components can be used as C++ classes. The top-block C++ class (`<gnuradio/gr_top_block.h>`) allows interconnecting library components and implementing the waveform in C++. Eric Blossom's guide "How to write a signal processing block" [5] helps to understand smart pointers, which are often used to instantiate C++ GNU radio blocks.

4.4.3.3 Component structure

In figure 4.16 the components `symbol_source`, `gr_pwr_squelch_cc.h`, and `ucla_ieee802-_15_4_packet_sink.h` are already available, either because they are included in the libraries or were designed in the previous steps. Therefore, the design of the DEMODULATOR component focused on the component `fm_demodulation`.

In the target waveform, `fm_demodulation` replaces the library component `ieee802_15_4_demod`, included in the *UCLA Zigbee PHY* library. Therefore, this Python library had to be translated in C++, using the C++ classes for the GNU Radio blocks. Then the component `fm_modualtion` was connect to the other components inside `receiver_path`, the class that inherits from `top_block`.

Hence, the demodulation functionality was implemented in the `receiver_path` class and could be instaciated and used within a C++ OSSIE component. The class `receiver_path` provides a method that allows the DEMODULATOR class to insert the received signal into a queue; while a thread running in the DEMODULATOR component checks for available demodulated frames and passes them to the output port.

4.4.3.4 Considerations

This approach to implementation requires only the standard GNU Radio library. In fact, for compiling this component, only the library `libgnuradio-core` needs to be linked. This is done by modifying the configuration file (`configure.ac`) in the OSSIE project folder for the DEMODULATOR component. The changes to the file `configure.ac` are shown in listing 4.1: the variables `GNURADIO_HEADERS`, `GNURADIO_LIBS`, and `GNURADIO_DEFINES` must be manually added in order to allow the compiler to link to the GNU Radio standard library.

```
GNURADIO_HEADERS="-I/usr/local/include/gnuradio"
GNURADIO_LIBS="-lgnuradio-core"
GNURADIO_DEFINES="-DOMNITHREAD_POSIX"

export PKG_CONFIG_PATH="$PKG_CONFIG_PATH:/usr/local/lib/pkgconfig"
CXXFLAGS="$CXXFLAGS $OSSIE_CFLAGS $GNURADIO_HEADERS $GNURADIO_DEFINES"
LIBS="$LIBS $OSSIE_LIBS $GNURADIO_LIBS"
```

Listing 4.1. Changes to the file *configure.ac* in the OSSIE project

Because GNU Radio is the only dependency for this OSSIE component, this DEMODULATOR component can be compiled and run on any OSSIE platform provided with GNU Radio[2].

Another aspect of this solution is that the resulting OSSIE component is an executable binary file and not a python module. The major reason why the DEMODULATOR was designed with this method was to test if avoiding the use of the Python interpreter would give better performance. The DEMODULATOR, in fact, is directly connected to the USRP interface and receives the signal from the ADC, which operates at a sampling rate of 64 Msps. See section 4.6 for further information about this performance evaluation.

4.4.4 An alternative method

The thesis project designed and evaluated the target waveform following the approaches presented in the previous sections. The goal of both the approaches was to use the existing GNU Radio libraries to perform signal processing inside OSSIE components. This is very useful since the current OSSIE library is, in fact, small, when compared with the standard GNU Radio library; in addition, the Comprehensive GNU Radio Archive Network (CGRAN) is a free open source repository for third party GNU Radio applications that are not officially supported by the GNU Radio project.

Section 4.6 presents the results of evaluation of the target waveform, using several different configurations. In order to understand these results, it was useful to evaluate a particular configuration, which used another component to implement the modulation. This particular implementation avoided linking to any GNU Radio library, thus the libraries for the IEEE 802.15.4 standard modulation were re-written in C++. This section describes this method (effectively giving a third approach that can be evaluated and compared to the two earlier approaches).

Re-implementing the modulation in C++ and debugging it took a long time since the waveform developer needs to understand all the details of the signal pro-

[2]At the end of this thesis project, OSSIE 0.8.0 was released: the compatibility problem with GNU Radio 3.2.2 had been fixed, thus OSSIE 0.8.0 and GNU Radio 3.2.2 can be installed on the same platform.

cessing. However, the time available for this thesis project was limited. Therefore, only modulation was implemented with this method; the resulting component was named FM_MODULATION and can be used in place of MODULATOR.

The compilation of FM_MODULATION does not need any external library and produces an executable file about half the size of MODULATOR (C++ implementation with GNU Radio libraries). In comparison with figure 4.15, `gr.message_source` and `mylib.symbol_sink` were removed, while the library `ieee802_15_4_mod` (a python module, included in *UCLA Zigbee PHY* library) was re-implemented in C++; and the component `gr.multiply_const_cc` was replaced with a multiplication.

This design required less computational resources (in terms of both memory and CPU cycles) since only one thread was running, to control the buffer on the input port. Additionally, the code for this component can be compiled for use on any OSSIE platform, because it does not need any external library (even not the standard GNU Radio libraries). One of the clear conclusions from this effort was that OSSIE library components should be implemented with this method, for the benefit of performance and portability.

4.4.5 Simulation

Once the MODULATOR and DEMODULATOR components were ready, the simulation waveform was configured and run in order to test the system for the correct behaviour. The MODULATOR and DEMODULATOR components can replace the MODEM_USRP component (see figure 4.14 at the beginning of this section). The TEST component illustrated in section 4.1 was also used in this simulation, as illustrated in figure 4.17.

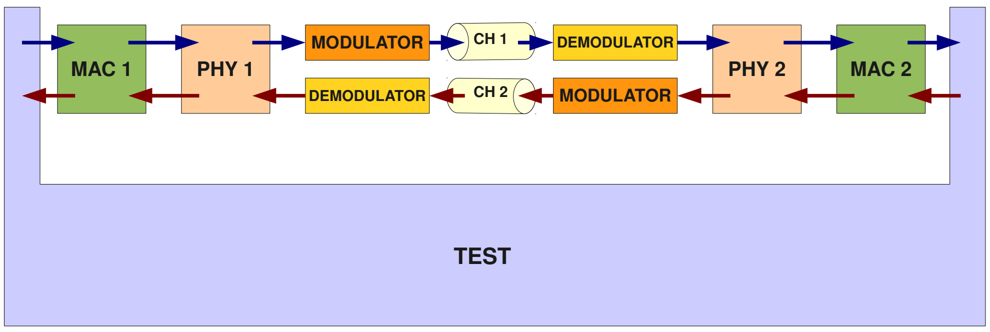


Figure 4.17. Waveform for testing MODULATOR and DEMODULATOR

The MODULATOR and DEMODULATOR components are interconnected through the *channel* component, which has an input port and an output port, both of type *complexShort*. This component is available in the standard OSSIE library and simulates the wireless physical channel; in fact, it is possible to set parameters such as added noise. Simulating the wireless physical channel makes it possible to deploy the waveform on the GPP of the computer (in OSSIE, `default_GPP_node`) and to test the MODULATOR and DEMODULATOR implementation without needing a USRP.

4.4.6 A chat application

Finally, the target waveform that was illustrated in the beginning of this section (figure 4.14 at page 76) was deployed on the OSSIE node `default_GPP_USRP_node`, which accesses the USRP.

The USRP_commander is a component available in the standard OSSIE library; it allows the waveform developer to configure and control the RF hardware on the USRP. This functionality implements (partially) the management service for the physical layer, as required by the IEEE 802.15.4 standard specification. This component can be easily modified and other functions can be added in order to satisfy all the requirements for the IEEE 802.15.4 management service for the physical layer. The USRP_commander, combined with the WaveDash tool, allows the waveform developer to control the waveform, by calling at runtime the *configure* method to set the frequency and gain properties. The final waveform is illustrated in figure 4.18.

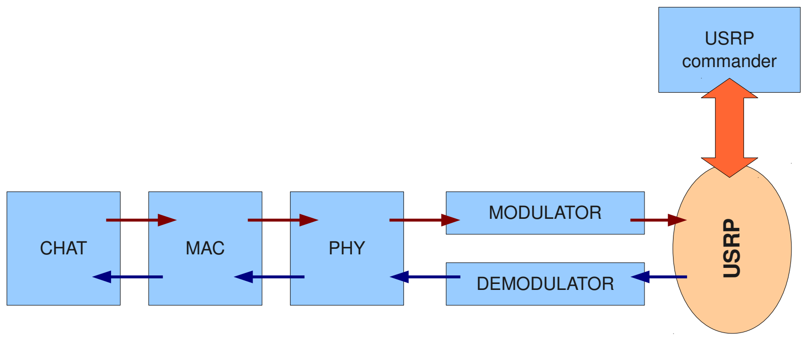


Figure 4.18. OSSIE waveform for the chat application

4.5 The TUN/TAP functionality

In chapter 3, when the goals of this thesis project were presented, the target waveform included two additional components, the LINK and the TUN/TAP components.

4.5.1 Description

TAP (as in network tap) emulates an Ethernet interface and it operates with layer 2 packets such as Ethernet frames. TUN (as in network TUNnel) emulates a network layer interface, by operating on layer 3 packets (such as IP packets). TAP can be used to create a network bridge, while TUN can be used together with routing (i.e., you can route packets to a TUN instance the same way that you can route them to an other network destination).

In this thesis project, TAP was used to implement tunneling of Ethernet frames over a point-to-point connection between two USRPs. More specifically we have used the IEEE 802.15.4 standard protocol to realize radio communication between two TAP interfaces (thus effectively building a IEEE 802.15.4 bridge – but *without* the computation of the bridge topology).

4.5.2 The TAP component

In order to determine the maximum size of the data payload in a IEEE 802.15.4 MPDU, a waveform designer must take into account the following considerations:

- according to the IEEE 802.15.4 standard specification, the maximum MAC frame size is 127 bytes (as described in section 4.2 the frame length field in the physical header of a PPDU is represented on 7 bits);
- the standard MPDU header fields and the checksum field require 5 bytes;
- within the MAC frame, the addressing fields have variable length: as a design choice in this project, only the source and destination fields were used and each address was represented using 2 bytes, thus 4 bytes were required for the addressing fields (for the MAC frame structure, see table 4.1 at page 59).

Therefore, in the target waveform, the maximum MPDU payload size is 127 - 5 - 4 = 118 bytes. Since TAP works with Ethernet frames and the maximum size of an Ethernet frame is 1518 bytes, each Ethernet frame can require up to 13 MPDUs in order to be transmitted through the USRP.

The original idea was to use two different components, the LINK component and the TUN/TAP component. However, the functionality of the LINK component in the target waveform was limited to splitting Ethernet frames into a set of smaller frames and doing re-assembly at the receiver. Since the transmission of each MPDU requires a CORBA call to the MAC layer, the LINK component would have introduced additional CORBA overhead without offering any advantage. Given these considerations and in order to reduce the complexity of the target waveform, the TAP functionality and the splitting Ethernet frames were implemented in the same OSSIE component, in this case the TAP component. Hence this augmented TAP component was connected directly to the MAC layer component.

4.5.3 Simulation

In order to test the correct implementation of the TAP component, the waveform illustrated in figure 4.19 was deployed. This waveform can run on the GPP node and does not need any USRP, since the physical radio channel is emulated by the *channel* component.

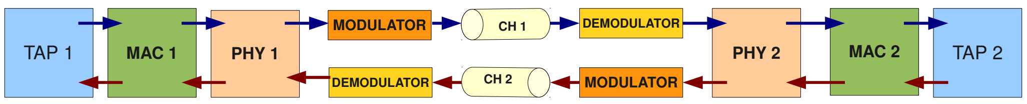


Figure 4.19. Waveform used to test the TAP component

An IP address must be specified for each TAP network interface. This address allows the TAP interface to be reached by other applications. For testing purposes the component TAP1 was configured with the IP address `192.168.0.1` and TAP2 with `192.168.0.2`. The connection between the two TAP components was tested with the *ping* program. By executing `‘‘ping 192.168.0.1’’` it was possible to verify that the IP packets were received on TAP2. Similarly, by executing `‘‘ping 192.168.0.2’’` the internet control message protocol (ICMP) packets were received on the TAP interface of component TAP1.

The `ping` program generates ICMP requests encapsulated in IP packets and sends them to the IP address `192.168.0.1` which causes them to be sent to the virtual network device created by component TAP1. The TAP interface allows the TAP component to read this data from the user memory space. Since TAP is a layer 2 network interface, Ethernet frames are read from the TAP interface; as seen above, these frames are then split into smaller frames to fit a MPDU and sent to the MAC layer interface. The (emulated) radio connection between the two TAP components allows the TAP2 component to receive the ICMP packets.

Referring to the semantics of the MAC request call in table 4.1.2.2 on page 60, the sub-frame number can be specified in the *msduHandle* argument. Thus, a future improvement could do Automatic Repeat-reQuest (ARQ) with retransmission of the sub-frames rather than having to retransmit the full frame.

4.5.4 The waveform

Once the implementation was successfully tested, the *channel* component was replaced by the physical radio channel between two USRPs. Figure 4.20 shows the configuration of the target waveform.

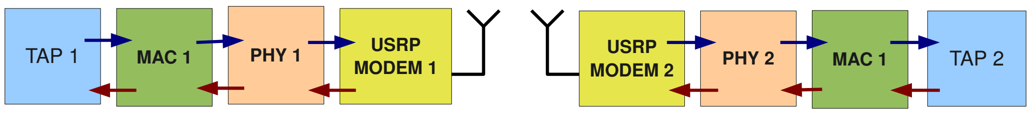


Figure 4.20. The target waveform with the TAP component

4.6 Results

Each design step described in the previous sections produced a OSSIE component to be added to the target waveform. After being designed and implemented, each component was then simulated, which could be done using only the GPP, i.e., without any RF hardware; as the *channel* component made it possible to simulate the physical wireless channel. Once the correct behaviour of the waveform components had been verified, the target waveform could be deployed upon the USRP.

4.6.1 The GNU Radio solution

4.6.1.1 The development method

The target waveform was first implemented using the MODEM_USRP component to modulate the signal and to access the USRP. The method for importing and using GNU Radio libraries within an OSSIE component was simple and easy to apply to the target waveform. This solution also made it possible, in the beginning of this thesis project, to solve the compatibility issues between OSSIE 0.7.4 and GNU Radio 3.2.2 (see secion 3.2).

This method is flexible and can be applied to other GNU Radio waveforms. The practical result was a running OSSIE waveform in a short time. The OSSIE chat waveform, in fact, could communicate with the GNU Radio chat waveform, using the configuration described in the end of section 4.3.

The MODEM_USRP component was also used with the TAP component in order to implement the point-to-point radio connection between two TAP interfaces (the configuration is illustrated in figure 4.20): the resulting waveform worked as expected.

4.6.1.2 The GNU Radio framework

The ease of development is not the only advantage of this solution:

- Accessing the USRP within the same component that modulates/demodulates the signal requires fewer CORBA calls, thus requiring less CPU and memory resources.
- The `top_block` class implements data transfer synchronization and realtime scheduling for the USRP source/sink component and the demodulator/demodulator component.

The last point is very important, for SDR development. In the MODEM_USRP solution, the USRP sink/source block and the MODEM block run within the `top_block` class and the GNU Radio framework hides the low level implementation details of the real-time scheduling. The waveform developer only has to call the `connect` method to interconnect the subcomponents within the `top_block` component.

Each GNU Radio block defines a `work` function that operates on its input to produce output streams. In order to help the scheduler to decide when to call the `work` function, blocks also provide forecast functions that tell the runtime system the number of input items it requires to produce a number of output items and how many output items it can produce given a number of input items. At runtime, blocks tell the system how many input (output) items they consumed (produced). Blocks may consume data on each input stream at a different rate, but all output streams must produce data at the same rate.

The `connect` function specifies how the output stream(s) of a processing block connects to the input stream of one or more downstream blocks. A key function during flow graph construction is the allocation of data buffers to connect neighboring blocks. The buffer allocation algorithm considers the input and output block sizes used by blocks and the relative rate at which blocks consume and produce items on their input and output streams. The flowgraph is then implemented as a single thread that loops over all the blocks in the graph, executing each block sequentially until all the data has been consumed.

However, GNU Radio is not designed to work in packet mode, rather the scheduler is designed to operate on continuous data streams. This, of course, is not a good fit for the packet oriented protocol implemented in this thesis project. In [9] Rahul Dhar, et al., report their experience of adding a Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) MAC protocol to GNU Radio: “GNU Radio was designed to support signal processing on continuous data streams. There is no concept of (fixed or variable sized) packets or frames. The stream-centric design is most prominent in the flowgraph mechanisms, specifically buffer management and scheduling”.

4.6.2 The OSSIE solution

4.6.2.1 The development method

SCA allows the waveform developer to structure a waveform and implement each function in a different component. Within each component, it is possible to instanciate the GNU Radio `top_block` class and interconnect different subcomponents: hence, the functionality implemented by an OSSIE component can be de-composed into a GNU Radio flowgraph. Since each GNU Radio flowgraph represents one processing thread, flowgraphs within different OSSIE components will be independent threads with separate scheduling.

The possibility to define custom interfaces makes it easy to implement a communication protocol, because IDL allows the waveform developer to define both new methods and data types. Thus, given a communication protocol based on the OSI model, each layer can be implemented as a SCA component. For each component, a custom interface is defined; the methods available on this interface represent the functions exported to the upper layer.

OSSIE supports Python and C++ implementations, however other languages can be used to implement SCA components. CORBA, in fact, makes it possible to interconnect components written in different languages. The advantage of OSSIE, from this point of view, is that it is possible to rapidly develop a waveform prototype using Python; once the design is verified, then the Python implementation can be translated in C++ in order to improve performance.

4.6.2.2 Simulation

Before deploying the target waveform, the MODULATOR and DEMODULATOR components were simulated (see figure 4.17 at page 82). This design phase made it possible to debug the waveform components. The most common problem for the MODULATOR and DEMODULATOR components was memory leaks. When started the waveform could run, but after a while (depending on the transmitter's rate) the waveform crashed. It was possible to observe (for example by running the program "`top`" in a shell) the memory usage for each OSSIE component, since each component runs in a separate thread.

Another kind of problem concerned the buffers. Both input and output ports use buffers to make up for possible delays in CORBA calls. However, in a SCA waveform the processing blocks are independent from each other. Since the data transfer is implemented by CORBA, there is no synchronization between the blocks. Therefore, unbounded buffers should be avoided, because they would cause a high rate processing thread to never release the CPU.

The simulation of the modulation scheme and of the TAP functionality showed that OSSIE is a useful tool for prototyping and learning about SCA waveforms. The *channel* component made it possible to simulate the radio channel and deploy the whole waveform (transmitter and receiver) on a GPP (i.e., on a PC). The WaveDash utility makes it possible to configure the waveform at runtime, for example to analyze the behaviour by changing the data transmission rate.

4.6.2.3 The USRP OSSIE device

Using the OSSIE interface for the USRP device turned out to be challenging. The waveform illustrated in figure 4.18 did not work as expected. In fact, only a few messages could be correctly received and demodulated, depending on the transmission rate. In order to understand the problem, different configurations were deployed. The parameters that were modified in each configuration were:

- **buffer size**: the `symbol_sink` has a queue for buffering the data to be sent to the USRP, while the *symbol_source* component has a queue to store the received signal;
- **transmitter rate**: a component was designed, to generate PPDUs at a constant rate;
- **receiver packet size**: this parameter can be set by the *USRP_Commander* component and determines how many bytes must be read from the USRP buffer (it must be a multiple of 512 bytes);

Since the OSSIE framework ran over Ubuntu, other parameters affected the waveform execution. In fact, the waveform components run on the GPP together with other user applications, such as the X Window System, the Java Virtual Machine (JVM) runtime, Eclipse, etc.

This problem seemed not to depend on the implementation of the MODULATOR and DEMODULATOR components. Section 4.4.4 illustrated an alternative implementation for the modulation, which did not use any GNU Radio library. Even using this component, the waveform presented the same undesired behaviour. Because of the poor transmission capabilities of this waveform, the TUN/TAP functionality could not be implemented in this configuration.

4.6.2.4 The OSSIE framework

Using the device interface for sending/receiving data from the USRP seems to imply that the problem is due to making CORBA calls to transfer data (arrays of shorts) between the modulator/demodulator component and the USRP interface. However, this should not be a problem since CORBA uses shared memory to implement local invocations. The configuration file for omniORB – the CORBA framework used by OSSIE – allows the user to set a priority for the transport rules (by default, the local filesystem is used, without using TCP/IP). This may mean that the problem is due to the thread scheduling.

A result was that by using a high transmission rate, the signal could be received and correctly demodulated. While in GNU Radio the `top_block` class is in charge of synchronizing the data transfer between components, the processing threads for OSSIE components are suspended when no data is available on the input ports, but are reactivated when new data is received. Another difference is that the GNU Radio flowgraph is implemented as a single thread, while in OSSIE waveforms several threads run in parallel, at least one for each component.

R. Dhar, et al., had a similar problem with implementing a MAC layer in GNU Radio: “We observed during test runs that the first frame was always transferred correctly while later frames were often corrupted. We discovered that this problem was caused by the basic TX cards. Since the cards were designed for stream-oriented communication, they continue to transmit, even after all the data in the USRP transmit queue has been sent. This transmission interferes with later frame transmissions by other nodes, often resulting in the corruption of those frames”[9].

4.6.3 PHY/MAC design in SDR

Both GNU Radio and OSSIE are designed to operate in stream mode and it is difficult to develop waveforms for transmitting packets. In fact, traditional SDRs receive a stream of samples from the RF hardware, captured at a specific decimation rate. The radio hardware (in this project, the USRP) cannot determine when packets for the destination MAC layer will arrive; as a result, the radio must remain in receiving state. The downside of this solution is that the demodulation process uses significant memory and processor resources despite the low rate of incoming packets. Thus, the solution to the problem would be to implement a packet mode for both GNU Radio and OSSIE.

Another solution would be to use the *split-functionality* approach for SDR development – proposed by George Nychis in [17], in opposition to the *host-based* approach, such as GNU Radio. While the *host-based* approach implements everything on the GPP of the host computer, the *split-functionality* approach is based on the principle that a set of core MAC functions must be implemented on the radio hardware for performance and efficiency reasons, while mantaining control on the host CPU through an API.

The goal of this solution is to achive fast-packet recognition at the radio hardware in order to demodulate only when necessary (CPU intensive); fast-packet recognition can also be used to trigger pre-modulated *dependent packets*, such as ACK messages. “Our results show three orders of magnitude greater precision for the scheduling of packets and carrier sense, along with a high level of accuracy in fast packet detection”, compared with a GNU Radio implementation of the same MAC layer [17].

4.6.4 Measurements

The target waveform did not work as expected when using the OSSIE interface for the USRP, but worked using GNU Radio to access the USRP. In order to understand the reason for this undesired behaviour, some measurements were made on both the GNU Radio waveform and the OSSIE waveform. Only the results for the receiver path are considered in this section because the demodulation of the received signal is the most CPU intensive function in the waveform.

The goal is to evaluate the rate of read operations from the USRP buffer, thus calculating the input data rate for the demodulator path. This was done by measuring the delay between two successive read operations from the USRP buffer – the physical buffer, on the USRP hardware. The measurements were done using the function `gettimeofday`, available in the library `time.h`. This function returns the absolute time with precision of microseconds. Note that it was not needed to make an exact measurement (for this purpose, a tool like *oprofile* would have been used), while to compare the OSSIE and the GNU Radio implementations.

The GNU Radio implementation of the chat application accessed the USRP through the component *usrp_source_c*. This component is derived from the class `usrp_basic_source`, which internally calls the *read* method of the USRP driver to read data from the USRP buffer. The `read` function is called within the `work` method of the `usrp_basic_source` class, hence the GNU Radio scheduler is in charge of calling this method.

The OSSIE implementation accesses the USRP through the OSSIE interface. This interface uses a separate thread to read data from the USRP buffer. The receiver thread is always running and its execution is independent of the other components in the waveform. Within this thread, the same *read* method is called, since OSSIE uses the GNU Radio driver to access the USRP.

This test was run using a Pentium M processor with clock speed of 1.73 GHz and 1 GB RAM. The results show the following average input data rate for the receiver path in the waveform.

Table 4.4. Input data rate for the receiver path

	GNU Radio	**OSSIE**
Data rate	8 MB/s	15.3 MB/s

4.6.4.1 Data transfer size

The plot in figure 4.21 shows that the number of bytes read from the USRP is constant in OSSIE: this value is specified in the *packet_size* option of the component USRP_commander, which is in charge of controlling the RF hardware on the USRP.

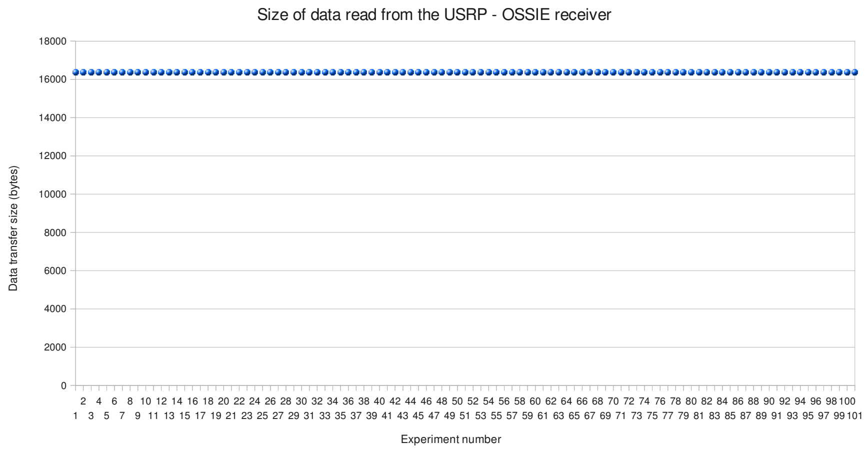


Figure 4.21. Data transfer size for the USRP (OSSIE implementation)

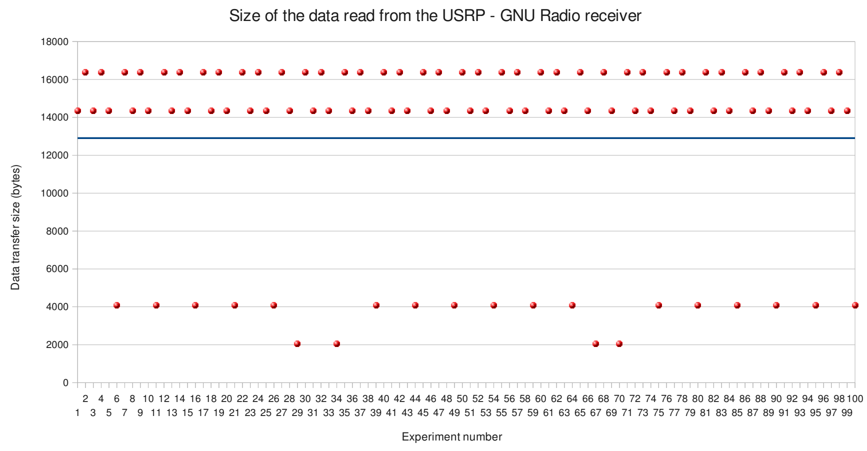


Figure 4.22. Data transfer size for the USRP (GNU Radio implementation)

In the GNU Radio implementation, instead, the data transfer size is variable, since it is calculated at runtime by the GNU Radio scheduler, according to the buffer availability. Note that the data size can assume only four different values: 2048 KB, 4096 KB, 14336 KB, and 16384 KB. The plot in figure 4.22 also shows the average value, that is 12844 KB.

4.6.4.2 Data transfer delay

The average data transfer delay in the OSSIE receiver is 1 ms and the plot in figure 4.23 shows the measurements made in this test. These measurements indicate that the data transfer delay was almost constant during the test.

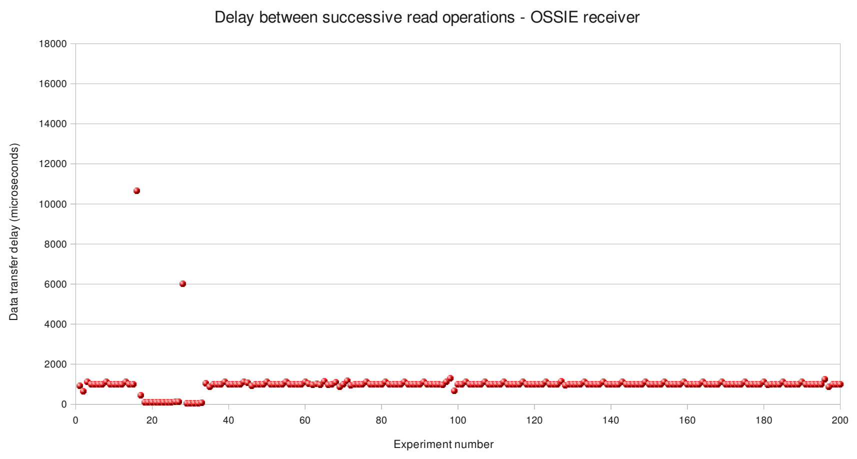


Figure 4.23. Data transfer delay for the USRP (OSSIE implementation)

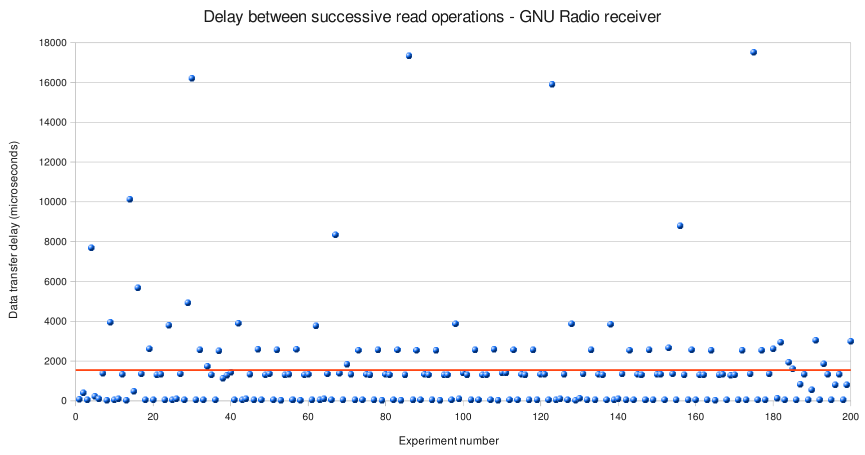


Figure 4.24. Data transfer delay for the USRP (GNU Radio implementation)

The plot in figure 4.24 shows the measurements of the data transfer delay for the GNU Radio implementation. There are two differences between the OSSIE and the GNU Radio receiver:

- The average value of the data transfer delay for the GNU Radio receiver (1.5 ms) is higher than in OSSIE (1 ms).
- The distribution of the measurements is uniform in OSSIE, since the data transfer delay is almost constant; while in GNU Radio the data transfer delay has an higher variance from the average value.

As described earlier, a GNU Radio flowgraph is a single processing thread and there are synchronization techniques for intercomponent communication. Therefore the delay between two read operations depends not only on the time required to access the USRP (via USB) but also on the GNU Radio scheduling.

In the OSSIE receiver, the delay between two read operations depends (i) on the CORBA delay to send the data to the next component (in charge of demodulating the signal), (ii) on the time required to read data from the USRP buffer, and (iii) on the scheduling of the threads on the CPU. However, the second parameter is the same for both OSSIE and GNU Radio, thus it does not have to be considered; while the CORBA delay is minimal compared to the data processing performed within the components, since in this test CORBA uses Unix domain sockets to transfer data.

Thus, the main difference between the two implementations seems to be the scheduling algorithm: in fact, the GNU Radio scheduler considers at runtime the buffer availability between all the components in a flowgraph and calculates the optimal data transfer size; while in OSSIE there is not any scheduler, thus the OSSIE receiver uses a separate thread to read data from the USRP, which has a scheduling independent of the other components in the waveform. This allows OSSIE waveforms to have an higher data transfer rate but at the same time may cause some problems because of the lack of synchronization.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Both GNU Radio and OSSIE are free toolkits for developping SDR applications. Both include a framework which provides the waveform developer a high level interface making it possible to develop a waveform by interconnecting library components; both GNU Radio and OSSIE include graphical tools for doing this.

However, the waveform developer needs in some cases to understand how the SDR framework works, since some unexpected problems may occur. In this thesis project, the goal was to use a USRP for sending and receiving an IEEE 802.15.4 frame as an RF signal. The target waveform was successfully realized using GNU Radio to connect the USRP device component to the modulator/demodulator component: the waveform implemented a radio point-to-point connection between two TAP interfaces, as illustrated in figure 4.20 at page 86. Unfortunatly, the OSSIE device interface for the USRP could not be used to realize the same waveform: the lack of synchronization between the USRP interface and the demodulator/modulator component was considered to be the reason for the unexpected behaviour of the waveform.

This thesis showed a general method for implementing a OSI-based communication protocol in OSSIE. SCA allows the waveform developer to create a component for each OSI layer. Internally, a GNU Radio block can be instanciated and connected to the ports of the OSSIE component. Hence, several subcomponents can be connected together to implement a more complex functionality, for example modulation and demodulation of the signal. The advantage is that it is possible to use existing GNU Radio components, thus making the OSSIE waveform development more rapid.

At the same time, portability – one of the key features of SCA – is still guaranteed: as shown in this report, the only requirement to run the target waveform is a platform configured with the OSSIE framework and the standard

GNU Radio library. The latest OSSIE package, released at the time of the end of this thesis project, is fully compatible with the latest GNU Radio distribution.

OSSIE is a young open-source project, thus it lacks the documentation and library components of more mature projects (such as GNU Radio). As a result these represent the main obstacles to waveform development. The aim of this thesis project was to explore an alternative method of implementation, by using GNU Radio libraries within OSSIE components to leverage the existing GNU Radio libraries. This should hopefully contribute to the OSSIE project and help waveform developers new to OSSIE.

5.2 Future Work

The initial directives for this thesis project where not restrictive, the goal was to evaluate the OSSIE toolkit by implementing a prototype waveform. Thus, the design was carried out in order to achieve a flexible solution and to explore methods of developing waveforms, which would allow a designer to implement any other waveform. Specially, the waveform design took into account existing GNU Radio solutions and used existing GNU Radio blocks within OSSIE components.

Another approach to the design would have been to implement the components needed for the target waveform without using any existing library. This would have been a valuable attempt to provide the OSSIE library with new components. Such an addition would be valuable as the OSSIE library is much smaller than the GNU Radio library, therefore as of today waveform development is more rapid in GNU Radio. The GNU Radio library was used to guide an implementation of a modulation scheme directly in C++ for OSSIE: it took longer to implement the modulation algorithm in this way, since the waveform developer needs to understand all the signal processing details. However, the resulting executable file was half the size of the equivalent GNU Radio component and needed only one processing thread.

A lot of time was spent in trying to fix the initial compatibility problem between OSSIE 0.7.4 and GNU Radio 3.2. However, the OSSIE developer team provided a patch to make OSSIE 0.7.4 work with GNU Radio 3.2. Lots of effort was also dedicated to understanding the reason for the problem with the OSSIE interface for the USRP. This problem required studying the source files for the USRP for both OSSIE and GNU Radio and examining the differences between them. Instead of spending time to understand the details of OSSIE and GNU Radio, this time could have been spent on the MAC layer and the management service described in the IEEE 802.15.4-2006 standard.

In the end of this thesis project, OSSIE version 0.8.0 was released by the OSSIE team. A continuation of this project sould first analyze any improvement in the OSSIE toolkit and trying to use the USRP interface provided by this new distribution.

Bibliography

[1] David Ascher. Dynamic languages: ready for the next challenges, by design. *ActiveState*, July 2004.

[2] P. Balister, M. Robert, and J. Reed. Impact of the use of CORBA for intercomponent communication in SCA based radio. In *SDR Forum Technical Conference*, Orlando, Florida, USA, November 2006.

[3] Philip Balister. A software defined radio implemented using the OSSIE core framework deployed on a TI OMAP processor. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, December 2007.

[4] John Bard and Vincent J. Koverik Jr. *Software Desined Radio: the Software Communication Architecture*. John Wiley and Sons, April 2007. ISBN 978-0470865187. 462 pages.

[5] Eric Blossom. *How to Write a Signal Processing Block*. GNU Radio, July 2006. URL http://gnuradio.org/redmine/wiki/gnuradio.

[6] Matt Carrick, Drew Cormier, Christopher Covington, Carl B. Dietrich, Joseph Gaeddert, Benjamin Hilburn, C. Ian Phelps, Shereef Sayed, Deepan Seeralan, Jason Snyder, and Haris Volos. *OSSIE 0.7.4 Installation and User Guide*, September 2009. URL http://ossie.wireless.vt.edu.

[7] Jacob A. DePriest. A practical approach to rapid prototyping of SCA waveforms. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, April 2006.

[8] A. Devlic, A. Graf, P. Barone, A. Mamelli, and A. Karapantelakis. Evaluation of context distribution methods via Bluetooth and WLAN: Insights gained while examining battery power consumption. In *Fifth Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2008)*, July 2008. Dublin, Ireland.

[9] Rahul Dhar, Gesly George, Amit Malani, and Peter Steenkiste. Supporting integrated MAC and PHY software development for the USRP SDR. In

1st IEEE Workshop on Networking Technologies for Software Defined Radio Networks, September 2006.

[10] Matt Ettus. *USRP family brochure.* Ettus Research. http://www.ettus.com.

[11] B. Foster and F. Andreasen. Basic media gateway control protocol MGCP packages. Technical Report 3660, IETF, Network Working Group, December 2003. Request for Comments.

[12] José A. Gutiérrez, Edgar H. Callaway Jr., and Raymond L. Barrett Jt. *Low-Rate Wireless Personal Area Networks.* Institute of Electrical & Electronics Engineers (IEEE), November 2003. ISBN 978-0738135571.

[13] Michael Ihde, Philip Balister, and Shereef Sayed. How should assembly controller start other components. URL `http://listserv.vt.edu/archives/open-source.html`. OSSIE-discuss forum, September 2009.

[14] Joseph Mitola III. Software radios survey, critical evaluation and future directions. In *IEEE National Telesystems Conference*, Washington, DC, USA, May 1992.

[15] Alaelddin Mohammed. Studying media access and control protocols. Master's thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology, Stockholm, Sweden, November 2009.

[16] James Neel. Simulation of an implementation and evaluation of the layered radio architecture. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, December 2002.

[17] George Nychis, Thibaud Hottelier, Zhuocheng Yang, Srinivasan Seshan, and Peter Steenkiste. Enabling MAC protocol implementations on software-defined radios. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, number 10.1109/VETECS.2007.18, pages 91–105, Berkeley, CA, USA, 2009. USENIX Association.

[18] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly Media, 1st edition, January 2001. ISBN-10: 0596001088 and ISBN-13: 978-0596001087.

[19] Max Robert, Shereef Sayed, Carlos Aguayo, Rekha Menon, Karthik Channak, Chris Vander Valk, Craig Neely, Tom Tsou, Jay Mandeville, and Jeffrey H. Reed. OSSIE: Open source SCA for researchers. *SDR Forum Technical Conference*, 47, 2004.

[20] Thomas Schmid. The UCLA Zigbee PHY library. The Comprehensive GNU Radio Archive Network (CGRAN). http://www.cgran.org/wiki/UCLAZigBee.

[21] Thomas Schmid. GNU Radio 802.15.4 En- and Decoding. *NESL Technical Report*, September 2006. University of California, Los Angeles, USA.

[22] Douglas C. Schmidt, Nanbor Wang, and Steve Vinoski. Object Interconnections - Collocation Optimizations for CORBA. *SIGS C++ Report magazine*, September 1999.

[23] Deepan N. Seeralan, Stephen Edwards, and Carl Dietrich. The Waveform DashBoard: An interactively configurable GUI for prototype SCA-based SDR waveforms. In *SDR'09 Technical Conference and Product Exposition*, December 2009. Washington, DC.

[24] IEEE Computer Society. IEEE std 802.15.4-2006. IEEE standard, Institute of Electrical & Electronics Engineers (IEEE), June 2006.

[25] Thomas Sundquist. Waveform development using software defined radio. Master's thesis, Linköping Universitet, Linköping, Sweden, April 2006.

[26] T. Tsou, P. Balister, and J. Reed. Latency profiling for SCA software radio. In *SDR Forum Technical Conference*, Denver, CO, November 2007.

[27] T. Ulversøy and J. Olavsson Neset. On workload in an SCA-based system, with varying component and data packet sizes. In *Papers presented at the RTO Information Systems Technology Panel (IST) Symposium held in Prague, Czech Republic, on 21-22 April 2008*, number RTO-MP-IST-083 AC/323(IST-083)TP/221. NATO Research and Technology Organisation, July 2008. ISBN 978-92-837-0065-4.

[28] Jongwon Yoon, Hyogon Kim, and Jeong-Gil Ko. Data fragmentation scheme in IEEE 802.15.4 wireless sensor networks. In *IEEE 65th Vehicular Technology Conference, 2007*, number 10.1109/VETECS.2007.18, pages 26–30. VTC2007-Spring, April 2007. ISBN 1-4244-0266-2.

Appendix A

Acronyms and Abbreviations

ADC	**A**nalog to **D**igital **C**onverter
CF	**C**ore **F**ramework
CGRAN	**C**omprehensive **G**NU **R**adio **A**rchive **N**etwork
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
CPU	**C**entral **P**rocesing **U**nit
DAC	**D**igital to **A**nalog **C**onverter
DSSS	**D**irect **S**equence **S**pread **S**pectrum
EDA	**E**uropean **D**efence **A**gency
ESSOR	**E**uropean **S**ecure **S**oftware **D**efined **R**adio
FCS	**F**rame **C**heck **S**equence
FIFO	**F**irst **I**n **F**irst **O**ut
FM	**F**requency **M**odulation
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GUI	**G**raphical **U**ser **I**nterface
ICMP	**I**nternet **C**ontrol **M**essage **P**rotocol
ICT	**I**nformation and **C**ommunication **T**echnology
IDL	**I**nterface **D**escription **L**anguage
ISO	**I**nternational **O**rganization for **S**tandardization
GIOP	**G**eneral **I**nter-ORB **P**rotocol
GNU	**G**NU is **N**ot **U**nix
GPP	**G**eneral **P**urpose **P**rocessor
GSM	**G**lobal **S**ystem for **M**obile communications
GSPS	**G**iga**S**amples **P**er **S**econd
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
JTRS	**J**oint **T**actical **R**adio **S**ystem
LAN	**L**ocal **A**rea **N**etwork
LLC	**L**ocal **L**ink **L**ayer
LR-PAN	**L**ow **R**ate **P**ersonal **A**rea **N**etwork
MAC	**M**edia **A**ccess **C**ontrol

MCPS	MAC **C**ommon **P**art **S**ublayer
MFR	MAC **F**oote**R**
MHR	MAC **H**eade**R**
MLME	MAC sub**l**ayer **m**anagement **e**ntity
MPDU	MAC **P**rotocol **D**ata **U**nit
MSDU	MAC **S**ervice **D**ata **U**nit
MSPS	**M**ega**S**amples **P**er **S**econd
OE	**O**perating **E**nvironment
OQPSK	**O**ffset **Q**uadrature **P**hase-Shift **K**eying
ORB	**O**bject **R**equest **B**roker
OSI	**O**pen **S**ystems **I**nterconnection
OSSIE	**O**pen **S**ource **S**CA **I**mplementation - **E**mbedded
PAN	**P**ersonal **A**rea **N**etwork
PC	**P**ersonal **C**omputer
PD-SAP	**P**hysical layer **D**ata - **S**ervice **A**ccess **P**oint
PHR	**P**hysical **H**eade**R**
PLME	**P**hysical **L**ayer **M**anagement **E**ntity
PPDU	**P**HY **P**rotocol **D**ata **U**nit
POS	**P**ersonal **O**perating **S**pace
QoS	**Q**uality **o**f **S**ervice
RAM	**R**andom **A**ccess **M**emory
RF	**R**adio **F**requency
SAP	**S**ervice **A**ccess **P**oint
SCA	**S**oftware **C**ommunications **A**rchitecture
SDR	**S**oftware **D**efined **R**adio
SFD	**S**tart-of-**F**rame **D**elimiter
SoC	**S**ystem **o**n **C**hip
SFDR	**S**purious-**F**ree **D**ynamic **R**ange
SHR	**S**ynchronization **H**eade**R**
SPDU	**S**SCS **P**rotocol **D**ata **U**nits
SSCS	**S**ervice-**S**pecific **C**onvergence **S**ublayer
SWIG	**S**implified **W**rapper and **I**nterface **G**enerator
USB	**U**niversal **S**erial **B**us
USRP	**U**niversal **S**oftware **R**adio **P**eripheral
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
VHSIC	**V**ery-**H**igh-**S**peed **I**ntegrated **C**ircuit
WCDMA	**W**ideband **C**ode-**D**ivision **M**ultiple-**A**ccess
WLAN	**W**ireless **L**ocal **A**rea **N**etwork
WPAN	**W**ireless **P**ersonal **A**rea **N**etwork
XML	e**X**tensible **M**arkup **L**anguage

Appendix B

Standard OSSIE component structure

Source file `MyComponent.py`

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

'''
/****************************************************************************

Copyright 2009 by your_name_or_organization, all rights reserved.

****************************************************************************/
'''

from omniORB import CORBA
from omniORB import URI
import CosNaming
from ossie.standardinterfaces import standardInterfaces__POA
from ossie.custominterfaces import customInterfaces__POA
from ossie.cf import CF, CF__POA
import sys

import WorkModule   # module found in the component directory.
                    # this module is where the main processing
                    # thread resides.
import threading
import time         # primarily availble for time.sleep() statements

#-------------------------------------------------------------
# MyComponent_i class definition (main component class)
#-------------------------------------------------------------
class MyComponent_i(CF__POA.Resource):
    def __init__(self, uuid, label, poa):
        CF._objref_Resource.__init__(self._this())
        print "MyComponent_i __init__: " + label
        self.naming_service_name = label
        self.poa = poa

        self.inPort0_servant = dataIn_complexShort_i(self, "dataIn")
        self.inPort0_var = self.inPort0_servant._this()

        self.outPort0_servant = dataOut_realChar_i(self, "dataOut")
        self.outPort0_var = self.outPort0_servant._this()
        self.WorkModule_created = False

        self.propertySet = []
        self.work_mod = None

    def start(self):
        print "MyComponent start called"

    def stop(self):
        print "MyComponent stop called"

    def getPort(self, id):
        if str(id) == "dataIn":
            return self.inPort0_var
        if str(id) == "dataOut":
            return self.outPort0_var

        return None  #port not found in available ports list

    def initialize(self):
        print "MyComponent initialize called"

    def configure(self, props):
        ''' The configure method is called twice by the framework:
        once to read the default properties in the component.prf
        file, and once to read the component instance properties
        storred in the waveform.sad file.  This method should be
        called before the start method.  This method is where
        the properties are read in by the component.
        '''

        print "MyComponent configure called"
        buffer_size = 0

        for property in props:
            if property not in self.propertySet:
                self.propertySet.append(property)
            if property.id == 'DCE:4f047ae2-cad1-11de-b541-00014ac5c4dc':
                self.prop1 = int(property.value.value())
            if property not in self.propertySet:
                self.propertySet.append(property)
            if property.id == 'DCE:638b6fca-cad1-11de-b541-00014ac5c4dc':
                self.prop2 = float(property.value.value())

        # make sure that only one WorkModule thread is started,
        # even if configure method is called more than once
        if not self.WorkModule_created:
            self.work_mod = WorkModule.WorkClass(self, buffer_size)
            self.WorkModule_created = True

    def query(self, props):
        return self.propertySet

    def releaseObject(self):
        # release the main work module
        self.work_mod.Release()

        # release the main process threads for the ports
        self.outPort0_servant.releasePort()

        # deactivate the ports
        iid0 = self.poa.reference_to_id(self.inPort0_var)
        oid0 = self.poa.reference_to_id(self.outPort0_var)

        self.poa.deactivate_object(iid0)
        self.poa.deactivate_object(oid0)


#-------------------------------------------------------------------
# dataIn_realChar_i class definition
#-------------------------------------------------------------------
class dataIn_realChar_i(standardInterfaces__POA.realChar):
    def __init__(self, parent, name):
        self.parent = parent
        self.name = name

    # WARNING:  I and Q may have to be changed depending on what data you
    # are receiving (e.g., bytesIn for realChar)
    def pushPacket(self, I, Q):
        self.parent.work_mod.AddData(I, Q)


#-------------------------------------------------------------------
# dataOut_realChar_i class definition
#-------------------------------------------------------------------
class dataOut_realChar_i(CF__POA.Port):
    def __init__(self, parent, name):
        self.parent = parent
        self.outPorts = {}
        self.name = name
        self.active = False

        self.data_buffer = []
        self.data_event = threading.Event()
        self.data_buffer_lock = threading.Lock()

        self.is_running = True
        self.process_thread = threading.Thread(target = self.Process)
        self.process_thread.start()

    def connectPort(self, connection, connectionId):
        port = connection._narrow(standardInterfaces__POA.realChar)
        self.outPorts[str(connectionId)] = port
        self.active = True

    def disconnectPort(self, connectionId):
        self.outPorts.pop(str(connectionId))
        if len(self.outPorts)==0:
            self.active = False

    def releasePort(self):
        # shut down the Process thread
        self.is_running = False
        self.data_event.set()

    # WARNING:  I and Q may have to be changed depending on what data you
    # are receiving (e.g., bytesIn for realChar)
    def send_data(self, I, Q):
        self.data_buffer_lock.acquire()
        self.data_buffer.insert(0, (I,Q))
        self.data_buffer_lock.release()
        self.data_event.set()
    def Process(self):
        while self.is_running:
            self.data_event.wait()
            while len(self.data_buffer) > 0:
                self.data_buffer_lock.acquire()
                new_data = self.data_buffer.pop()
                self.data_buffer_lock.release()

                for port in self.outPorts.values():
                    port.pushPacket(new_data[0], new_data[1])

                self.data_event.clear()



#-------------------------------------------------------------------
# ORB_Init class definition
#-------------------------------------------------------------------
class ORB_Init:
    """Takes care of initializing the ORB and bind the object"""

    def __init__(self, uuid, label):
        # initialize the orb
        self.orb = CORBA.ORB_init()

        # get the POA
        obj_poa = self.orb.resolve_initial_references("RootPOA")
        poaManager = obj_poa._get_the_POAManager()
        poaManager.activate()

        ns_obj = self.orb.resolve_initial_references("NameService")
        rootContext = ns_obj._narrow(CosNaming.NamingContext)

        # create the main component object
        self.MyComponent_Obj = MyComponent_i(uuid, label, obj_poa)
        MyComponent_var = self.MyComponent_Obj._this()

        name = URI.stringToName(label)
        rootContext.rebind(name, MyComponent_var)

        self.orb.run()

#-------------------------------------------------------------------
# Code run when this file is executed
#-------------------------------------------------------------------
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print sys.argv[0] + " <id> <usage name> "

    uuid = str(sys.argv[1])
    label = str(sys.argv[2])

    print "Identifier - " + uuid + "  Label - " + label

    orb = ORB_Init(uuid, label)
```

Source file WorkModule.py

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

'''
/*******************************************************************************

Copyright 2009 by your_name_or_organization, all rights reserved.

*******************************************************************************/

'''

#!/usr/bin/env python
import threading

class WorkClass:
    """This class provides a place for the main processing of the
    component to reside."""

    def __init__(self, MyComponent_ref, buffer_size):
        '''Initialization.  Sets a reference to parent.
        Initializes the buffer.  Starts the Process data
        thread, which waits for data to be added to the buffer'''

        self.MyComponent_ref = MyComponent_ref
        self.buffer_size = buffer_size

        self.data_queue = []
        self.data_queue_lock = threading.Lock()
        self.data_signal = threading.Event()

        self.is_running = True

        self.process_thread = threading.Thread(target=self.Process)
        self.process_thread.start()

    def __del__(self):
        '''Destructor'''
        pass

    def AddData(self, I, Q):
        '''Generally called by parent.  Adds data to a buffer.
        The Process() method will wait for this buffer to be set.
        '''
        self.data_queue_lock.acquire()
        self.data_queue.insert(0, (I,Q))
        self.data_queue_lock.release()
        self.data_signal.set()

    def Release(self):
        self.is_running = False
        self.data_signal.set()

    def Process(self):
        while self.is_running:
            self.data_signal.wait()  # wait for data to be aded to the
                                     # buffer in self.AddData()
            while len(self.data_queue) > 0:
                # get the data from the buffer:
                self.data_queue_lock.acquire()
                new_data = self.data_queue.pop()
                self.data_queue_lock.release()

                # get data out of tuple
                I = new_data[0]
                Q = new_data[1]

                newI = [0 for x in range(len(I))]
                newQ = [0 for x in range(len(Q))]


                # Insert code here to do work
                # Example:
                #for x in range(len(I)):
                #    newI[x] = I[x]
                #    newQ[x] = Q[x]


                # Output the new data
                if self.MyComponent_ref.outPort0_servant.active:
                    self.MyComponent_ref.outPort0_servant.send_data(newI, newQ)


            self.data_signal.clear()  # done reading the buffer
```

Appendix C

Custom OSSIE component

Source file `MyComponent.py`

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

from ossie.standardinterfaces import standardInterfaces__POA
from ossie.cf import CF, CF__POA

from omniORB import CORBA
from omniORB import URI
import CosNaming

import sys

import WorkModule
import portImpl

buffer_size = 16

#-------------------------------------------------------------
# MyComponent_i class definition (main component class)
#-------------------------------------------------------------
class MyComponent_i(CF__POA.Resource):

    def __init__(self, uuid, label, poa):
        CF._objref_Resource.__init__(self._this())
        print "MyComponent_i __init__: " + label
        self.naming_service_name = label
        self.poa = poa

        self.dataIn_servant = portImpl.dataIn_realChar_i(self, "realChar")
        self.dataIn_var = self.dataIn_servant._this()

        self.dataOut_servant = portImpl.dataOut_realChar_i(self, "realChar",
                                                            buffer_size)
        self.dataOut_var = self.dataOut_servant._this()

        self.work_mod = WorkModule.WorkClass(self, buffer_size)

        self.propertySet = []

    def start(self):
        print "MyComponent start called"
        self.dataOut_servant.start()
        self.work_mod.start()

    def stop(self):
        print "MyComponent stop called"
        self.work_mod.stop()
        self.dataOut_servant.stop()

    def getPort(self, id):
        if str(id) == "dataIn":
            return self.dataIn_var
        if str(id) == "dataOut":
            return self.dataOut_var

        return None  #port not found in available ports list

    def initialize(self):
        print "MyComponent initialize called"

    def configure(self, props):
        print "MyComponent configure called"

        for property in props:
            if property not in self.propertySet:
                self.propertySet.append(property)
            if property.id == 'DCE:4f047ae2-cad1-11de-b541-00014ac5c4dc':
                self.prop1 = int(property.value.value())
            elif property.id == 'DCE:638b6fca-cad1-11de-b541-00014ac5c4dc':
                self.prop2 = float(property.value.value())

    def query(self, props):
        return self.propertySet

    def releaseObject(self):
        # deactivate the ports
        iid0 = self.poa.reference_to_id(self.dataIn_var)
        oid0 = self.poa.reference_to_id(self.dataOut_var)

        self.poa.deactivate_object(iid0)
        self.poa.deactivate_object(oid0)

#-------------------------------------------------------------------
# ORB_Init class definition
#-------------------------------------------------------------------
class ORB_Init:
    """Takes care of initializing the ORB and bind the object"""

    def __init__(self, uuid, label):
        # initialize the orb
        self.orb = CORBA.ORB_init()

        # get the POA
        obj_poa = self.orb.resolve_initial_references("RootPOA")
        poaManager = obj_poa._get_the_POAManager()
        poaManager.activate()

        ns_obj = self.orb.resolve_initial_references("NameService")
        rootContext = ns_obj._narrow(CosNaming.NamingContext)

        # create the main component object
        self.MyComponent_Obj = MyComponent_i(uuid, label, obj_poa)
        MyComponent_var = self.MyComponent_Obj._this()
        name = URI.stringToName(label)
        rootContext.rebind(name, MyComponent_var)

        self.orb.run()

#-------------------------------------------------------------------
# Code run when this file is executed
#-------------------------------------------------------------------
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print sys.argv[0] + " <id> <usage name> "

    uuid = str(sys.argv[1])
    label = str(sys.argv[2])

    print "Identifier - " + uuid + "  Label - " + label

    orb = ORB_Init(uuid, label)
```

Source file `WorkModule.py`

```
#! /usr/bin/env python
from buffer import circular_buffer

class WorkClass:

    def __init__(self, MyComponent_ref, buffer_size=0):
        self.parent = MyComponent_ref
        self.dataIn_buffer = circular_buffer(self.dataIn_callback, buffer_size)

    def start(self):
        self.dataIn_buffer.start()

    def stop(self):
        self.dataIn_buffer.stop()

    def pushPacket(self, dataIn):
        self.dataIn_buffer.put(dataIn)

    def dataIn_callback(self, data):
        self.parent.dataOut_servant.send_data(data)
```

Source file `portImpl.py`

```
from ossie.custominterfaces import customInterfaces__POA
from ossie.cf import CF__POA

from buffer import circular_buffer

#------------------------------------------------------------------
# use port class definition
#------------------------------------------------------------------
class use_port(CF__POA.Port):

    def __init__(self, parent, idl, buffer_size=None):
        self._parent = parent
        self._outPorts = {}
        self._active = False
        self._interface = eval("customInterfaces__POA." + idl)
        self._buffer = None
        if buffer_size != None:
            self._buffer = circular_buffer(self.buffer_callback, buffer_size)

    def connectPort(self, connection, connectionId):
        port = connection._narrow(self._interface)
        self._outPorts[str(connectionId)] = port
        self._active = True

    def disconnectPort(self, connectionId):
        self._outPorts.pop(str(connectionId))
        if len(self._outPorts)==0:
            self._active = False

    def start(self):
        if self._buffer != None:
            self._buffer.start()

    def stop(self):
        if self._buffer != None:
            self._buffer.stop()

    def buffer_callback(self, data):
        print "ERROR: method buffer_callback not implemented"

#-------------------------------------------------------------------
# dataIn_realChar_i class definition
#-------------------------------------------------------------------
class dataIn_realChar_i(standardInterfaces__POA.realChar):

    def __init__(self, parent, idl):
        self.parent = parent
        self.idl = idl

    def pushPacket(self, dataIn):
        self.parent.work_mod.pushPacket(self, dataIn)

#-------------------------------------------------------------------
# dataOut_realChar_i class definition
#-------------------------------------------------------------------
class dataOut_realChar_i(use_port):

    def send_data(self, dataOut):
        if self._buffer != None:
            self._buffer.put(dataOut)
        elif self._active:
            for port in self._outPorts.values():
                port.pushPacket(dataOut)

    def buffer_callback(self, data):
        if self._active:
            for port in self.outPorts.values():
                port.pushPacket(data)
```

Source file `buffer.py`

```
import Queue
from threading import Thread

class buffer(Thread):

    def __init__(self, callback, buffer_size):
        Thread.__init__(self)
        self._callback = callback
        self._queue = Queue.Queue(buffer_size)
        self._running = False

    def run(self):
        self._running = True
        while self._running:
            data = self._queue.get()
            if self._running:
                self._callback(data)

    # start() method inherited from Thread class

    def stop(self):
        self._running = False
        try:
            self._queue.put_nowait(None)
        except Queue.Full:
            pass
        # reinitialize the thread so it is possible to call start again
        Thread.__init__(self)


class queue_buffer(buffer):

    def put(self, data):
        try:
            self._queue.put_nowait(data)
        except Queue.Full:
            print "ERROR - buffer overflow: the newest data will be discarded!"
            pass


class circular_buffer(buffer):

    def put(self, data):
        try:
            self._queue.put_nowait(data)
        except Queue.Full:
            print "ERROR - buffer overflow: the oldest data will be discarded!"
            self._queue.get()
            self.put(data)
            pass
```

Appendix D

IDL custom interfaces

Source file `sap_app_to_link.idl`

```
#include "ossie/PortTypes.idl"

module customInterfaces{

        /* link layer SAP for applications */
        interface sap_app_to_link{

                void request(   in unsigned short source_address,
                                in unsigned short destination_address,
                                in PortTypes::CharSequence data,
                                in unsigned short priority      );

        };
};
```

Source file `sap_link_to_app.idl`

```
#include "ossie/PortTypes.idl"

module customInterfaces{

        /* application layer SAP for link */
        interface sap_link_to_app{

                void indication(in unsigned short source_address,
                                in unsigned short destination_address,
                                in PortTypes::CharSequence data,
                                in unsigned short priority      );

        };

};
```

Source file `sap_link_to_mac.idl`

```
#include "ossie/PortTypes.idl"

module customInterfaces{

        /* mac layer SAP for link layer */
        interface sap_link_to_mac{

                void mcpsData_request(  in octet SrcAddrMode,
                                        in octet DstAddrMode,
                                        in unsigned short DstPANId,
                                        in unsigned long long DstAddr,
                                        in octet msduLength,
                                        in PortTypes::CharSequence msdu,
                                        in octet msduHandle,
                                        in octet TxOptions,
                                        in octet SecurityLevel,
                                        in octet KeyIdMode,
                                        in unsigned long long KeySource,
                                        in octet KeyIndex );

        };
};
```

Source file `sap_mac_to_link.idl`

```
#include "ossie/PortTypes.idl"

module customInterfaces{

        /* link layer SAP for mac layer */
        interface sap_mac_to_link{

                void macpsData_confirm( in octet msduHandle,
                                        in octet status,
                                        in unsigned long Timestamp );

                void mcpsData_indication(       in octet SrcAddrMode,
                                                in unsigned short SrcPANId,
                                                in unsigned long long SrcAddr,
                                                in octet DstAddrMode,
                                                in unsigned short DstPANId,
                                                in unsigned long long DstAddr,
                                                in octet msduLength,
                                                in PortTypes::CharSequence msdu,
                                                in octet msduLinkQuality,
                                                in octet DSN,
                                                in unsigned long Timestamp,
                                                in octet SecurityLevel,
                                                in octet KeyIdMode,
                                                in unsigned long long KeySource,
                                                in octet KeyIndex );
        };
};
```

Source file `sap_phy_to_mac.idl`

```
#include "ossie/PortTypes.idl"

module customInterfaces{

        /* mac layer SAP for physical layer */
        interface sap_phy_to_mac{

                void pdData_indication( in octet psduLength,
                                        in PortTypes::CharSequence psdu,
                                        in octet ppduLinkQuality );

                void pdData_confirm(in octet status);

        };

};
```

Source file `sap_mac_to_phy.idl`

```
#include "ossie/PortTypes.idl"

module customInterfaces{

        /* physical layer SAP for mac layer */
        interface sap_mac_to_phy{

                void pdData_request(    in octet psduLength,
                                        in PortTypes::CharSequence psdu );

        };

};
```

TRITA-ICT-EX-2010:27