(former Helsinki University of Technology)
Faculty of Electronics, Communications and Automation
Degree Programme of Communications Engineering

Marjukka Kokkonen

A high-level model of an embedded controller for an RF transceiver

Master's Thesis submitted in partial fullfillment of the requirements for the degree of Master of Science in Technology

Espoo, 29th April 2010

Supervisor: Professor Heikki Saikkonen, Aalto University
Instructor: D.Sc.(Tech.) Vesa Hirvisalo, Aalto University

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communications and Automation
Degree Programme of Communications Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Marjukka Kokkonen
Title of thesis:	A high-level model of an embedded controller for an RF transceiver
Date:	29th April 2010 **Pages:** xii + 65
Professorship:	T-106 Software Technology
Supervisor:	Professor Heikki Saikkonen
Instructor:	D.Sc.(Tech.) Vesa Hirvisalo

A mobile hand-held may contain nowadays many different radio devices, typically one for each of the radio protocols it supports. Each of these makes an additional cost and takes space on the device. As small cost and device size are important design factors for a mobile hand-held, any chance for reducing the number and size of needed components is warmly welcomed.
Digital basebands can be seen to be mergeable to one generic processor due to increase in current processor speeds. A generic baseband processor is able to handle baseband processing for multiple radio connections concurrently, but it is yet to be seen how much integration can be done to the RF parts. With currently available technologies, we cannot get rid of all of the parallelism.
The merged components require control different from their predecessors. Merged components should still contain the functionality of the preceding components, but be more flexibly controllable, as the callers are more diverse than before. Ultimately we would want to end up to a very few, fully flexible components, usable by any radio protocol or radio system. Cognitive Radio and Software Defined Radio are the aims the radio related research now strives for.
In this thesis, we present a high-level system model of part of a futuristic RF radio, including controller software for a futuristic RF radio transceiver. The system models controller software for RF transceiver, whose sub-parts are concurrently usable by any radio protocol, instead of dedicated RF parts for each radio protocol or radio system. In the model, the control functionality that connects protocol-level commands to RF parts was modelled.
The model is based on two interfaces. These are the Lyra model of the radio device interface, and an interface for a generic RF transceiver module. The model was made with SystemC.
As the result, our model shows that a generic multiradio RF transceiver can be made controllable by several concurrently running radio protocols. The constructed model may be further used as a model for more accurate model-based radio system designs.

Keywords:	radio system, controller, RF, model, SystemC
Language:	English

TEKNILLINEN KORKEAKOULU
Elektroniikan, tietoliikenteen ja automaation tiedekunta
Tietoliikenteen koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Marjukka Kokkonen
Työn nimi:	Korkean tason malli sulautetun radiotaajuuslähetinvastaanottimen ohjaimesta
Päiväys:	29. huhtikuuta 2010 **Sivumäärä:** xii + 65
Professuuri:	T-106 Software Technology
Työn valvoja:	Professori Heikki Saikkonen
Työn ohjaaja:	TkT Vesa Hirvisalo

Mobiililaitteessa voi nykyään olla useita radiolaitteita, tyypillisesti yksi tuettua radioprotokollaa kohti. Kukin radio lisää kustannuksia ja vie laitteesta tilaa. Koska halpa hinta ja pieni koko ovat usein tärkeitä mobiililaitteen suunnittelukriteerejä, kaikki mahdollisuudet tarvittavien komponenttien vähentämiseksi ja niiden koon pienentämiseksi otetaan innolla vastaan.

Radiojen digitaalisten kantataajuuskomponenttien toiminnot voidaan parhaimmillaan yhdistellä yhden, riittävän nopean, yleiskäyttöisen prosessorin tehtäväksi. Tämä on nykyisin mahdollista, koska nykyisin on saatavilla riittävän nopeita prosessoreja. Yleiskäyttöinen prosessori pystyy nopeutensa vuoksi käsittelemään useiden samanaikaisten radioyhteyksien kantataajuiset datat. Sen sijaan radiotaajuuksia käsittelevien komponenttien osalta yhdistelymahdollisuudet ovat vielä pitkälti selvittämättä. Nykyisillä teknologioilla kaikesta komponenttien rinnakkaisuudesta ei vielä päästä eroon.

Yhdistellyt komponentit tarvitsevat edeltäjistään poikkeavan ohjauksen. Yhdistelystä huolimatta komponenttien tulee edelleen tarjota sama toiminnallisuus kuin edeltäjiensä, mutta niiden täytyy olla joustavampia, sillä yhdisteltyjen komponenttien käyttäjät ovat vaihtelevampia kuin edeltäjien. Pitkän tähtäimen päämääränä komponentit halutaan karsia lukumääräisesti vähäisiksi, mutta toiminnoiltaan täysin joustaviksi, siten, että satunnainen radiojärjestelmä tai -protokolla voisi käyttää niitä. Kognitiivinen radio ja softaradio (Software Defined Radio, SDR) ovat joustavia radiojärjestelmiä, joihin radioihin liittyvä tutkimus tähtää.

Tässä työssä rakensimme korkean tason simulaattorimallin futuristisen radiotaajuuksia käsittelevän lähetinvastaanottimen osasta, erityiseseti sen kontrolleriohjelmistosta. Ohjelmisto mallintaa perinteisen, aina yhtä radiojärjestelmää vastaavan yhden signaalipolun sijaan radiotaajuuskomponentteja kontrolloivan ohjelmiston toiminnallisuutta sellaiselle radiolaitteelle, jossa kaikki radion osat ovat samanaikaisesti kaikkien radioprotokollien käytettävissä. Ohjelmisto mallintaa kontrollia protokollatasolta radiotaajuuskomponenttien digitaaliseen rajapintaan saakka.

Simulaattori rakentuu kahden annetun rajapinnan varaan. Nämä ovat Lyra-metodilla tehty radiolaitteen rajapinta sekä yleiskäyttöisen radiotaajuuslähetinvastaanottimen rajapinta. Simulaattori rakennettiin SystemC-kirjastoa käyttäen.

Mallin perusteella näyttää siltä, että protokollasta riippumaton, yleiskäyttöinen multiradiolähetinvastaanotin voidaan todella rakentaa siten, että useat samanaikaisesti aktiivisena olevat radioprotokollat voivat käyttää yleiskäyttöistä radiotaajuuslähetinvasaanotinta samanaikaisesti. Rakennettua mallia voidaan käyttää tarkempien radiomallien suunnittelun pohjana.

Avainsanat:	radiojärjestelmä, ohjain, RF, malli, SystemC
Kieli:	Englanti

Contents

List of Figures

Preface

A brief history of my thesis writing project follows, notes to self mostly...

This thesis was constructed, and mostly written, when working at former Helsinki University of Technology (HUT), at Department of Computer Science and Engineering, at Embedded Software Group. For the last edition rounds, HUT already merged into the Aalto University, which is unified school of three former Finnish universities.

Thesis is written as a side effect of a project we called RIM, standing for Radio Interfce Modelling. This was a very interesting dive into the diverse world of mobile radio technology. Into the world of future radio technologies, of hardware design, of system modelling, of circuit technology. Into the world of endless number of definitions of a radio and several of a baseband, and at least two of a service.

The whole process took a long time, over two years of calendar time.

But much was learned along the way, and the book got written at last. This was a good trip, I say.

Acknowledgements

I want to thank the project organization - TKK departments of Computer Science and Engineering (CSE), and Micro and Nano Technology (MNT), and Nokia SDR team. Thank you all for making the RIM project exist and the required resources available, for giving me the opportunity to participate in a very interesting project and for providing me with the subject to write this thesis about.

Naming names...

Professor Heikki Saikkonen (TKK, CSE), thank you for your supervision.

Dr. (tech.) Vesa Hirvisalo (TKK, CSE/ESG), thank you for encouragement and patience, advice and instructions, for explaining the backgrounds and sharing visions. Thanking you also for the warm hearted and open minded talks and discussions in general - they have been of a great help, for the project called life.

The rest of the Embedded Software Group of my thesis writing time - Aleksi Aalto, M.Sc. Sami Kiminki, Jaakko Kotimäki, Juhani Peltonen, Jyry Suvilehto, Timo Töyry, and Juho Äyräväinen. (As well as the nearly group members and wannabes. ;) ... names left out for hardness to decide which all to include...) For I am gregarious one, thank you all for being the most friendly, homy herd for me. Thank you for aiding with the daily pitfalls, keeping company, and having a lot of fun, too. Yeah, and thanking Sami, Timo and SVN problems for inducing me eventually to use LaTeX instead the wysiwyg editors with not-so-handy reference generating system.

MNT guys, thank you for co-operating. M.Sc. Mikko Talonen for keeping the project running when no one still knew what was to be done, Nasir Kadiri for sharing the pain of a Master's Thesis writer for this project, and Professor Jussi Ryynänen for staying up late with us. ;)

M.Sc. (tech.) Antti Immonen and the rest of the SDR team, thank you for providing us with the information about whatever we could think to ask, and for the generally supportive attitude towards us during the project.

The generic group called friends, thank you for providing me with many cheerful moments, and changing thoughts. Laughing gets one to relax, and different points of view are always warmly welcomed...

My parents and parents-in-law, for all of your support, be that practical, financial, emotional or whatever, there has been plenty of those.

My family Jarno and Lassi, for sharing the days going by, let those be sunny or rainy. It's great I have you.

Otaniemi 29th April 2010

Marjukka Kokkonen

Abbreviations and Acronyms

AD Analog to Digital

ADC/DAC Analog to Digital and Digital to Analog converters

AGC Automatic Gain Control

AM Amplitude Modulation

ASIC Application Specific Integrated Circuit

BB Baseband

BT Bluetooth

CMOS Complementary Metal Oxide Semiconductor

CIM Computation Independent Model

CPU Central Processing Unit

CR Cognitive Radio

DA Digital to Analog

DMA Direct Memory Access

DSP Digital Signal Processing

GFSK Gaussian frequency-shift keying

GSM Global System for Mobile Communications

FE Front-end

FIFO First In, First Out

FM Frequency Modulation

FPGA Field Programmable Gate Array

HAL Hardware Abstraction Layer

HW Hardware

IC Integrated Circuit

IF Intermediate Frequency

IP Intellectual Property

LAN Local Area Network

LNA Low Noise Amplifier

LO Local Oscillator

MBD Model-Based Design

MIMO Multiple In, Multiple Out

MRC Multi Radio Controller

MSC Message Sequence Chart

NIC Network Interface Card

OS Operating System

QPSK Quarature Phase Shift Keying

PA Power Amplifier

PIM Platform Independent Model

PLL Phase Locked Loop

PSAP Provided Service Access Point

PSM Platform Specific Model

RF Radio Frequency

RF-FE Radio Frequency engine front-end

RM Resource Manager

RTOS Real-Time OS

RX receiver

SDR Software Defined Radio

SDR model Software Defined Radio functional architecture model

SW Software

SX clock circuit

TDM Time Division Multiplexing

TX transmitter

UML Unified Markup Language

USAP Used Service Access Point

VHSIC Very High Speed Integrated Circuit

VHDL VHSIC Hardware Description Language

WLAN Wireless LAN

Chapter 1

Introduction

In this thesis, we present a high-level model of a multiradio device, and a model of controller software at functional level for a Radio Frequency (RF) transceiver. The model represents controller software functionality that could be found in a futuristic radio device.

With the model, we can see what we can learn about radio control software requirements, what structures are needed by the radio controller, and are there similarities to already matured controller software technology, operating systems.

Usage of a set of generic radio interface functions is simulated. The interface functions are of a kind that controller of a futuristic radio device could offer. The interface functions are to be generic in a way that they could be used by any radio system, that is, independent of the radio protocol using them, in a platform where multiple radio systems could use the same radio hardware concurrently. With our simulator, we put the interface functions to a test, to see if they really can be used to control a generic radio device by multiple radio protocols concurrently.

1.1 Motivation

A mobile hand-held may contain nowadays many different radio devices embedded in them. For example, Nokia model N97 supports GSM850/900/1800/1900, Bluetooth, WLAN, HSDPA, WCDMA, GPRS, and GPS systems[25].

Previously, each of the radio protocols have typically had dedicated radio tube(s)[1] including digital baseband and RF chips [28]. Each of these makes an additional cost and takes space on the device.

Small cost and device size are typically important design factors for embedded systems in general. The same goes for mobile hand-helds [19, 17], which also need to be as energy efficient as possible. As the number of the radio systems integrated into a single device has increased, and the number of the possible radio systems has grown, a need has emerged to make the usage of the growing number of radio tubes more efficient. Any chances for reducing the number and size of the needed components are warmly welcomed, as well as ways to minimize energy consumption. These have brought up a question about possibilities to merge at least some parts of some radio tubes.

Possibilities to reduce the number of RF components come from increasing the flexibility of components, either hardware or software ones [38].

When component integration increases in radio devices, the applicability of the integrated radio tubes change, changing their usage at several levels of the system. Both the design process and the way the resulting platform is used at each layer, change from the traditionally used.

Would there become changes to the hardware side, the software layers controlling the radio must change, too. Though the number of the layers of control might stay the same as in the previous approach, the information content sent through the layers would have to change, at least.

The traditional way of controlling a radio tube has been as seen in Figure 1.1a. The radio can be thought of as a peripheral device inside the "main" device. The application needing the radio device uses the device through a driver of that particular radio. The driver may be used either directly, or through an operating system of the main device. The radio driver then commands the radio tube through Hardware Abstraction Layer (HAL). Commands given to the radio tube would most probably be writings into simple hardware registers. As the radio tube is constructed for a particular radio standard, the tube handles the given data automatically according to the protocol it knows.

[1] Radio tube stands coarsely to the path the radio signal takes when propagating through the radio. Digital baseband may or may not be included, depending on who is speaking.

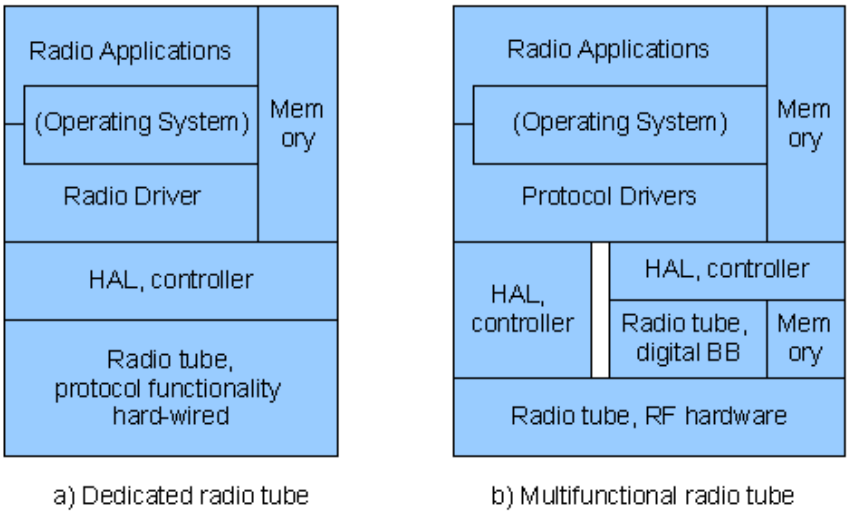

Figure 1.1: Radio control layers. Dedicated tube vs. multiradio tube.

Now, when the radio tube changes to one with flexible components, becoming able to accomplish the tasks of more than one radio standard or radio protocol, things change, as seen in Figure 1.1b. The way to use the tube changes in several parts of the system. We still have an application, wanting to use the radio. It still uses the radio functionality either through an operating system, or directly. However now, as the radio tube is not dedicated to a single radio protocol any more, we must take some parts of protocol-level information up in the stack, to be able to tell the radio components, how they should behave with the given data or signal, instead of telling them simply to process the data (the only way they know). Instead of a radio driver, we can say we now have a protocol driver.

We could say, simplifying things a bit, that the digital baseband and RF parts made up a First In, First Out (FIFO) radio tube formerly. The signal or the data was put in, and data or signal came out as a result. This changes with multiradio, a radio tube able to serve multiple radio systems concurrently, and/or parallelly. The timely usage of the digital baseband and RF parts becomes only loosely linked. As the protocol is the one to know what to do and when, it is up to it to tell the baseband and the RF modules when to act. The control commands thus come from the protocol layer, now outside of the tube, instead of the tube knowing on

its own what to do and when.

1.2 Research problem

Within this thesis, we are interested in the control of a multiradio device. In a multiradio device, there will be some controller software for the blocks of the device. But what kind of software would there be, within the controllers controlling the usage of the flexible multiradio tube parts? How complex software structures are needed there, and what is the functionality laying between the radio protocols, and the digital baseband, and RF hardware?

The questions relate to the research of Software Defined Radio (SDR), and Cognitive Radio (CR), yet futuristic systems for implementing and organizing radio devices, and the usage of frequency of the radio systems. Much has been studied already, but there is still a long way to go until we have them available in our every day customer products. Multiple In, Multiple Out (MIMO) radio techniques, radio hardware integration, and multiradios (available as various definitions), are all examples about steps towards SDR, and CR.

Previously, generic desktop processing units have been shown to be fast enough to be able to implement some computing intensive functionality formerly implemented by fixed dedicated analog hardware[16]. This shows that processing power of some affordable general processing devices is already sufficient to cope with current radio system needs, though we supposedly must still wait for some years before we have alike processors for mobile devices available.

A SDR has been fully implemented as a low-frequency radio [38]. This shows the idea of SDR being eligible. Small in size low power radio front-end capable of frequencies between 174 MHz and 6 GHz is implemented by IMEC [15], showing that generic, flexible RF components are within reach.

High-level multiradio functionality has been considered and taken into functional blocks in [4]. Control protocols for multiradios have been considered *e.g.,* in [36] for link-layer protocol for multiple WLAN Network Interface Cards (NICs), and in [2] for network router nodes. Control software for a digital baseband of a multiradio has been considered in [29], including a hardware implementation.

We took into consideration the control for the RF part of a multiradio, as this was

not found to have been studied.

1.3 Results

This thesis presents a high-level functional model of a radio device, and controller software for a futuristic RF radio transceiver, which is referred here as "the radio simulator". The radio simulator models the control of a generic RF module that is concurrently usable by any radio protocol. The RF module is seen as being fully independent of the digital baseband of the multiradio, what comes into controlling them.

The radio simulator models the control functionality that connects protocol-level commands to RF parts. It is made with SystemC, and it is based on two given interfaces. The interfaces are the Lyra model of a radio device interface, and an interface for a generic RF transceiver module.

A hardware implementation of the RF radio was not available, but a software model of a generic RF transceiver was got instead, to test the control software functionality with. The radio simulator models parts of the control traffic needed to command a generic RF front-end hardware device through a dedicated interface.

1.4 Thesis' structure

This thesis is divided into six chapters.

In this chapter, the topic has been introduced.

Chapter 2 sets background for this work, looking at mobile hand-helds (our target platform for an embedded radio device), operating systems (much studied and tested in the context of control software), peripheral devices (which are the devices to control) and control in general.

Chapter 3 provides a general description of our particular peripheral device to control, the RF radio transceiver. In Section 3.5, SDR and CR are introduced, concepts that widely motivate and guide the radio device research today, including this work.

Chapter 4 contains a brief look into system modelling. Also, the modelling tools relating to this work are introduced, Lyra modelling technique in Section 4.2, and SystemC in Section 4.5. The two radio models we used as the base for this work are introduced, too. Nokia SDR model is introduced in Section 4.6, and radio hardware model of the RF part in Section 4.7.

Chapter 5 introduces the radio control simulator built. We look at simulator structure, and skim through some code examples.

Chapter 6 wraps up the work.

Chapter 2

Background

In this chapter, we gather up some basics needed in order to understand peripheral controller modelling. Peripheral control usually differs from complex system control somewhat. Usually, we do not need a full sized operating system in a peripheral device, let alone in an embedded system with just one or a few clearly defined, quite simple, repeating tasks.

We begin our background review by looking into the structure of mobile handhelds, which are typical devices to contain multiple radio devices.

We look into operating systems, and peripheral devices. Operating systems are a much studied and already matured branch of control software, and peripheral devices a way to think about the device to control to.

Last, we look into what kind of structures the control software consists of, and how the structures typically evolve, as the software gets more mature. Some control is found at various levels in a device, from the lowest physical levels to the level of an application. Standards and regulations state still something more, for the single device as well as for the surrounding network environment. We do not have space to consider these here more closely, but some related information can be found in, e.g., [22] for embedded systems design, [30] for RF electronics control, and [12] for mobile protocols.

2.1 Mobile hand-helds

Mobile hand-helds have started their triumph at 90's from analog hand-held wireless telephones, devices that had microphone, speakers, a modulator and a radio frequency engine[26]. Since then, mobile hand-helds have developed much. Analog mobile phones have changed to digital ones, and one-purpose devices have evolved to multi-purpose devices. Nowadays, you may use your mobile hand-held to take pictures or videos, to browse the web, to make a call, to play games, and to check your location with GPS, only to mention a few.

A mobile hand-held of today is a miniaturized, low-power optimized computing system, with the main processor chip and several peripheral chips for various purposes integrated on the same board.

Mobile hand-helds are also a major genre of embedded systems today. Embedded systems are systems that have hardware and controller software together in a single device. This controller software is also called firmware. The user of the device with firmware does not even need to know that there is software inside the device[1].

A mobile hand-held has an operating system in it, taking care of the management of the functionality of the device, and access to its peripherals, and offering the user the handle to the part of the software the user is aware of, a (usually graphical) user interface. The operating systems of mobile devices are typically not as complex as those found on a full-sized personal computer, but simpler software to control the device, fitted to be able to accomplish just enough to make the system run.

The peripherals inside the mobile hand-held may have some firmware of their own, too, or why not even an operating system, if the required functionality is complex. The whole functionality of a peripheral accomplishing some tasks may consist of plain firmware running on a general purpose chip. There is often no more technological reasons for having fixed circuits to accomplish jobs, but manufacturing costs, design traditions, and partial re-usability may still be good enough reasons to not yet move into full software implementation.

Despite the vast growth of their complexity, the mobile hand-helds have had a tendency to get smaller. Mobile hand-held being a device that needs a screen and

[1] E.g. the washing machines and cars of today typically are embedded systems containing at least some firmware.

an input system, the useful minimum size for them has generally been reached. Still, the miniaturization continues, as growing amount of features is now wanted to fit in the same space. The current evolution goes on the one hand towards integrating many subsystems into the same board or even chip, but on the other hand towards some dedicated hardware acceleration modules, where e.g. the processing speed or power resistance of the system would not be sufficient enough otherwise [26].

2.2 Operating systems and peripheral devices

Operating systems technology has already matured, as it has history of some decades, from 1950's [32]. In the arrangements of control of operating systems, we can probably find useful hints about how to arrange the control of our device, what ever that would be.

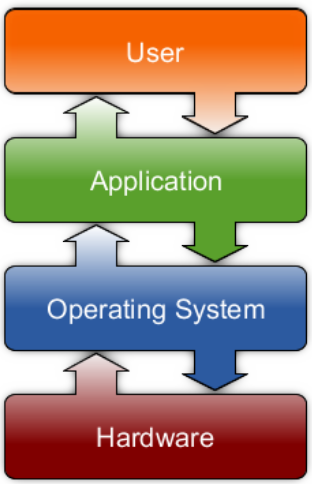

Figure 2.1: Logical placement of an operating system.
Figure from: http://en.wikipedia.org/wiki/Operating_system, licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

An Operating System (OS) is software made to control and abstract the resources the computing device has to offer. Logically it resides between the hardware and the application program (see Figure 2.1). It usually aims to simplify the usage of the underlying hardware, seen by the application programmer. An operating system hides much of the low-level details crucial for system functionality, usability, and extendability, but not directly relating to the functionality of the end user application itself.

OSs typically handle tasks like scheduling of the processes, threads or tasks, memory space allocation, regulating the usage of different system parts or peripheral devices, and handling user accounts to provide the multiple users the feeling of using the device alone. An OS also typically offers abstracted interfaces for different types of devices, taking much of the burden of device specific functionality coding away from an application programmer.

In this section, we look briefly at OSs, and how they relate to peripheral devices. If more detail is needed, the reader may familiarize with suitable textbooks (e.g., [14, 32, 34]).

2.2.1 General purpose operating systems

General-purpose Operating Systems are the most familiar ones for the majority of the people. These are systems called Linux, Microsoft XP, Vista, MacOS, Unix variants, or the like. Each of these has further distributions tailored for slightly different user profiles and various platforms. In following, we look at some functionality typical of these operating systems.

Nowadays any advanced OSs offers a way to run multiple programs concurrently. The user might want to e.g. write an email, listen to music, download a file and run a virus scanner at the same time. This is possible because the OS allows driving multiple processes seemingly at the same time, that is, concurrently. Concurrency is independent of the number of available processors. Software structures related to accomplishing concurrency safely and fluently include e.g. scheduler, processes, threads, locks, and interrupts.

General purpose OSs provides an application program with a way to use the memory address space of the computing device in isolation from other applications.

Every application sees only its own isolated memory sandbox where it is allowed to play freely. This makes the whole computer system more stable. As the programs are not allowed to use other memory space directly, it becomes much harder for them to corrupt the memory spaces of the other programs, let it be intentional or not. Common functionality usable by all applications, offered by the OS, is accessed through traps to the so called kernel code. The kernel can be thought of as the core to an OS. Unlike application program code, the kernel code may be executed in kernel mode, having full access to the whole memory space of the device.

OSs regulate the usage of permanent memory devices such as hard disks or flash sticks, and provide a file system to use them a structured way. This makes the storage usage easier, by offering the user tools to keep her files in order and keeping them from corrupting each other. Permanent memory devices can also be used for virtual memory. Virtual memory is an OS functionality used to extend available memory space beyond physical memory limits.

The OSs offer abstraction layers for peripheral device usage. An OS usually offers generic device interfaces for various different peripheral devices, to be used by the application programmer. The peripherals might be pointing devices, typing devices, printers, monitors, sound devices, network devices, graphic processing devices, measuring devices etc. There are uniform interfaces at least for the most often used peripheral device types available, letting the application programmer free from the details each peripheral device would need to its operation. The device specific control code is then placed in a device driver, residing logically between the abstraction layer the operating system offers, and the device itself.

2.2.2 Operating systems for different purposes

There are different OSs for different devices, made to answer the various needs of the different gadgets.

In the complex end of OS family, there are OSs built for generic systems which might have multiple users operating at the same time, having various needs for the usage of the system, and wide extendability.

Mobile hand-helds and other devices with less computing power or reduced means

of use have usually simplified OSs offering a subset of the functionality of that of a full-sized OS.

In a real-time system, where acting at certain points of time, or fast reacting times are important, a Real-Time OS (RTOS) could take extra care of correct timings. A factory robot might utilize a RTOS, as well as a radio transceiver. Both of these usually require precise timings.

There are also simple controller programs for simple systems, which may offer only a few fixed things to do. This kind of software, that you may also think of as a very reduced OS, can be found usually in an embedded system, e.g. a washing machine.

2.2.3 Peripheral devices

Peripheral devices are attached to the main computer, to widen its abilities. Peripheral devices typically add some functionality to the device, such that the main computer in itself is not capable of accomplishing, or they may increase the computing power of the device. Peripheral devices include e.g. screens, keyboards and other input devices, radio devices, sound devices, graphic devices and external storage.

A peripheral device consists of the hardware doing the dirty job, and wirings connecting it to the main device and to the power source. The peripheral may have a controller chip or a few in it where needed. Usually a dedicated driver, and in some cases a user application, is shipped with the device, to allow and ease its usage and configuration.

Formerly, the hardware for most of the peripherals has been implemented as Application Specific Integrated Circuits (ASICs), whose functionality is fixed at the factory[37]. Technological advancements have enabled electronic systems, also peripheral devices, to move from fixed hardware to more flexible solutions. These include everything from reconfigurable Field Programmable Gate Array (FPGA) hardware to general-purpose processor chips with a suitable software stack, all of which enable configurating or programming the system later. This allows the manufacturers to fix bugs, configure, and to introduce new functionality to the product later through firmware updates, even after the product launch.

Peripherals must be connected to a main computer in order to be of any use. Connecting is done through what are called buses. Buses consist of wirings, and control arrangements to manage the data transfers. Wirings have previously consisted of control traffic lines, address lines, and data lines, and control arrangements have handled timings and buffering of the data. Newer bus architectures are becoming conceptually close to networks. They prefer flexibility at least when it comes to bus speeds, physical connectivity, or the types of devices able to connect to the bus.

Peripheral hardware is often used and/or configured simply by setting values at hardware registers, which often have the width of one to some bytes or words[2]. The states of the registers may define the operation of the hardware directly, or there might be an interpreter inside the peripheral, interpreting the register values into action statements.

2.2.4 Control software of peripheral devices

Operating System is software intended to control and abstract the usage of the resources the device offers. Depending on the device, the needed constructs vary widely. A device may not need a full size OS, but some control software is for good. A simple loop of control may suffice if the system is a simple one. More mature methods are used when needed.

There are various ways to make a relatively simple control software for relatively simple devices[5]. When the system to be controlled grows larger and more complex, more functionality must be put into the operating system so it can meet the requirements.

Control loops could be named as the simplest type of control software. Anything they do, they do within a single program loop. The program loop takes turns to visit all the functions of the system, one after another, within the certain program loop. Usually, the systems that use control-loop must have no strict timing requirements. They are generally poor when it comes to responsiveness, as they just do their loop, but do not prioritize or respond to much to anything. **Cooperative multitasking** can be seen as a more developed version of control loops.

[2]Typical widths: byte = 8 bits, word = 16, 32 or 64 bits.

Interrupt-based control software works best at the systems whose functionality consists mostly of responses to service requests. An interrupt controlled system has a way to notice and react sufficiently fast to interrupts that tell there is something to react to. The tasks are executed mainly as responses to interrupt requests.

Pre-emptive multitasking control software introduces time based interrupts, and low-level code for switching between the running threads. Thus, it allows several threads to really run concurrently, unlike all the previously presented control schemes. Pre-emptively multitasking systems are that complex, that they can be thought to have an operating system.

Microkernels are stripped down kernels that take care of multitasking and memory allocation, but no other OS tasks.

A **RTOSs** is used in a device which needs precise timing. A RTOS must have sufficiently accurate knowledge about time, and they must complete the given tasks within certain time limits. In systems where a RTOS is needed, the tasks must be completed within time limits, or doing it is of no use no more. RTOSs are further divided to *soft RTOSs* and *hard RTOSs*.

With the soft RTOSs, the timing is important, but the given time-limits are still somewhat flexible. The missed time limit does not make the completion of the task useless immediately, but the usefulness of the computation result degrades fast when the given time limit has passed.

Hard RTOSs need to complete their tasks in a precise time, or at the latest at given time, otherwise the result of the computation is just a waste of computing power. As an example, take a factory robot hand, designed to operate with products moving on the factory line. If the robot hand is late with its function, it fails to operate on the product which already went further on the line. There is no use to do anything late - operating on the empty space instead of the product is of no use.

2.3 Modelling control

In a peripheral communication device, control is all about making electric signals flow the wanted ways. There are transmission lines to get the signals from a place to another. At the end of each transmission line, there is a way to catch the signal.

At the end point of a simple transmission line, the way may be just the detection of a signal state, when there is one wire. In a more complex transmission lines specialized devices can be used to handle the usage of the line.

At the software level, control is all about commands or instructions given within the system in order to get the device to function properly as wanted. It is about making choices about what is done next, according to the available knowledge. Control propagates through interfaces, using data structures and algorithms on the way.

Data structures are the format of the data, telling how the data is arranged. E.g., arrays, objects, and class definitions are data structures.

Algorithms tell how the given data is handled.

Interfaces make using the algorithms and data structures easier and regulate the usage. Interfaces hide low-level details of the sub-system from the developer.

Control paths are code structures that pass information from one interface to another. They usually appear as execution paths or function call chains.

When modelling the control, these aforementioned are the structures that are modelled. The modelling tool must be able to present these as clearly as possible. Modelling methodology and model structures must highlight these, and make them clear. In Section 4, we look at these in more details.

Control, or controller software, in the context of this thesis, is about hardware management according to protocol based information. Manageable things include decisions about when to act (time), if there are alternative traits of acting, what to do, and in some cases, if the system is flexible enough, how to do it as well.

A controller needs to get the system parts work together, usually following some kind of protocol. At the software level, controller software is sending messages flowing through interfaces, using appropriate data structures on the way. Near to physical hardware in the control stack, the data structures may be simple hardware registers to set. When higher in the software stack, the used data structures are usually more complex, and they may have more advanced access routines.

A typical controller consists of a processor together with auxiliary control logic, connected to the controllable hardware. The processor takes commands from outside (e.g., from Central Processing Unit (CPU)) and interprets them to instructions

of how to use the hardware. Other way said, the processor of the controller listens to the commands given to it, executes the control logic software, and command the hardware by the logic it uses.

Aim of this work is in getting understanding about embedded controllers, by generating a high-level model of a radio controller. A high-level model can not accurately describe a fully functional, readily manufactured product but depending on the model it could be used, e.g., to decide what system parts to put into the product, and how to arrange the control, (ways to use model-based computing[11]) or to build a more accurate model based on it (Model-Based Design (MBD)[27]).

2.3.1 Typical control software evolution

Whatever gadget we have, the control is usually implemented first more or less ad-hoc. That is, on what ever way, to provide just enough control for what we have. How the control is organized, depends on the people creating it, their abilities and preferences.

If the system evolution goes on, the control structures evolve. Typically more is wanted from them. Someone needs to get a new number out of the system. The system is needed to be faster. Maybe there is emerging a bunch of new systems for similar usage, where using the same controlling system, or at least parts of it, would be beneficial.

Through experience, it becomes better known what it is what is to be controlled, what the things are needed from the control. What the things are, need to be told by the user or by outside of the controller. This leads to simplifying and streamlining the system, driven by the need to hide the huge amount of details the developers should otherwise know in order to be able to make things work. As details are hidden, control gets modularized, layered, and structured. Relatively rigid interfaces of some kind are formed, since usage is easier through a fixed, hopefully well documented interface, than through a random bunch of ever changing functionality.

2.4 Summary

Operating systems technology offers tools for concurrent processing. However, the usage of the most powerful technologies in mobile devices is limited by the restricted computing power and energy available with the mobile devices. The processors available for mobile devices are not as powerful as their desktop cousins, and the processor speed is usually set to be just adequate, partly because of the cost, partly because the slower pace usually means longer battery life.

The platform the control is made for, naturally affects much the organization of the control. Complex systems usually need more control than simple ones. In order to be manageable, complex systems usually need better structuring, clearer interfaces and more modularization than simple ones. The reasons are diverse. Better structured, and well layered control is usually simpler to use by a software developer, and makes only the needed part of the control directly available, lessening the possibilities of data corruption due to programming errors.

Chapter 3

Radio

This section is about the functionality and control of a radio device. We first look at the generic functionality needed in order to have a working radio device. We look, how the data is transferred into radio waves and vice versa, and what kind of modules does an implemented transceiver tend to have. As we are particularly interested in modelling the control near the antenna end of the transceiver, we take the Radio Frequency engine front-end (RF-FE) functionality under more detailed examination. Futuristic radio systems, CR and SDR are briefed at the end of the chapter. These are simplified presentations about these subjects, see [13, 30, 6] for more.

3.1 Radio device

A radio device is one that uses radio waves[1] to communicate. A radio device may be able to *transmit* or *receive*, or both. A radio device which can both transmit and receive is called a *transceiver*. For examples, FM radio systems only need powerful transmitters at antenna towers, and a receiver only at consumer radio devices. Any radio system designed for two-way communication, e.g. Global System for Mobile Communications (GSM) or Wireless LAN (WLAN), has transceivers in both ends of the communications channel.

[1]Radio waves are electromagnetic radiation with wavelengths longer than those of visible light, that is, of lower frequency than light.

At the dawn of radio technology, the radios were all analog, and the sent data was all analog signal - most often human speech. The receivers and transmitters were separate devices, only later were they packed into a transceiver form, containing both. Usage of the radio frequencies was much unregulated, as there were just a few users.

Since then, digital technology has advanced, and radio communication has become more popular. Nowadays, vast majority of the radio traffic is digital, and the usage of the radio frequencies has become strictly controlled subject, leaving only a few "free" frequency bands for experimental devices to use.

The typical radio device is no more a simple analog device with a tangent to press, when wanting to speak. The typical radio device of today is a complex system having a probably complex antenna subsystem, the still yet analog RF front-end, the Analog to Digital (AD) and Digital to Analog (DA) converters, some baseband processing chips in order to get the digital information out of the analog signal, a software protocol stack for the device to know when and how to transmit or receive, and on the top of it all, some kind of a user interface to let the end user use this all, preferably with a few button pushes and an informative monitor to look at.

Radio devices nowadays usually reside in a portable handheld device. They may be embedded in computer devices such as GPS navigator or a mobile phone, or in something with less functionality, such as car keys with which you may remotely open the car doors. Radio devices may as well be peripheral devices for personal computers, e.g. WLAN cards.

3.2 Data path

The job of a radio transmitter is to convert the data to the stream of bits, and further into an analog radio frequency signal in the transmitter. Correspondingly, the receiver must convert the analog signal to the digital stream of bits, and further, the data must be extracted from the bits.

In Figure 3.1, we have block-level description of a radio device offering this data format transformation service, and in Figure 3.2 we have a brief description of the stages the data goes through when is gets transferred from bits to signal. In the

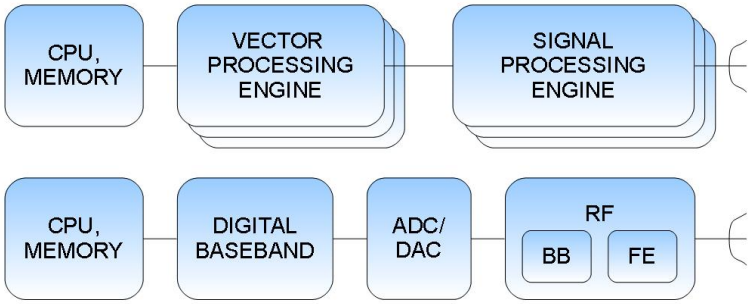

Figure 3.1: Radio gadget block structure.

following, we take a look into these stepwise - what happens to the data on a radio device, and what are the blocks that do the transformation.

3.2.1 Memory and application data

To be able to transmit any data, we must first have the data somewhere[2]. In a digital system, the data is always first stored in a memory device of some kind. From the memory, it is transferrable to the radio device when needed. When transmitting with a radio, from the memory the data is sent to a digital baseband, or more generically speaking, to a vector processing engine.

3.2.2 Digital baseband and data modulation

Digital baseband handles applying the modulations to the data. Modulations are the way to code the data into the electromagnetic signal. The simplest modulations would be Amplitude Modulation (AM) or Frequency Modulation (FM), but most radio standards nowadays use more complex modulations. They are more complex to implement, but are used due to more efficient frequency band usage, better power efficiency, or for they are less prone to transfer errors due to an unideal

[2]It may be a file saved to the device, a user may type the data it in, or it might be collected from microphone input.

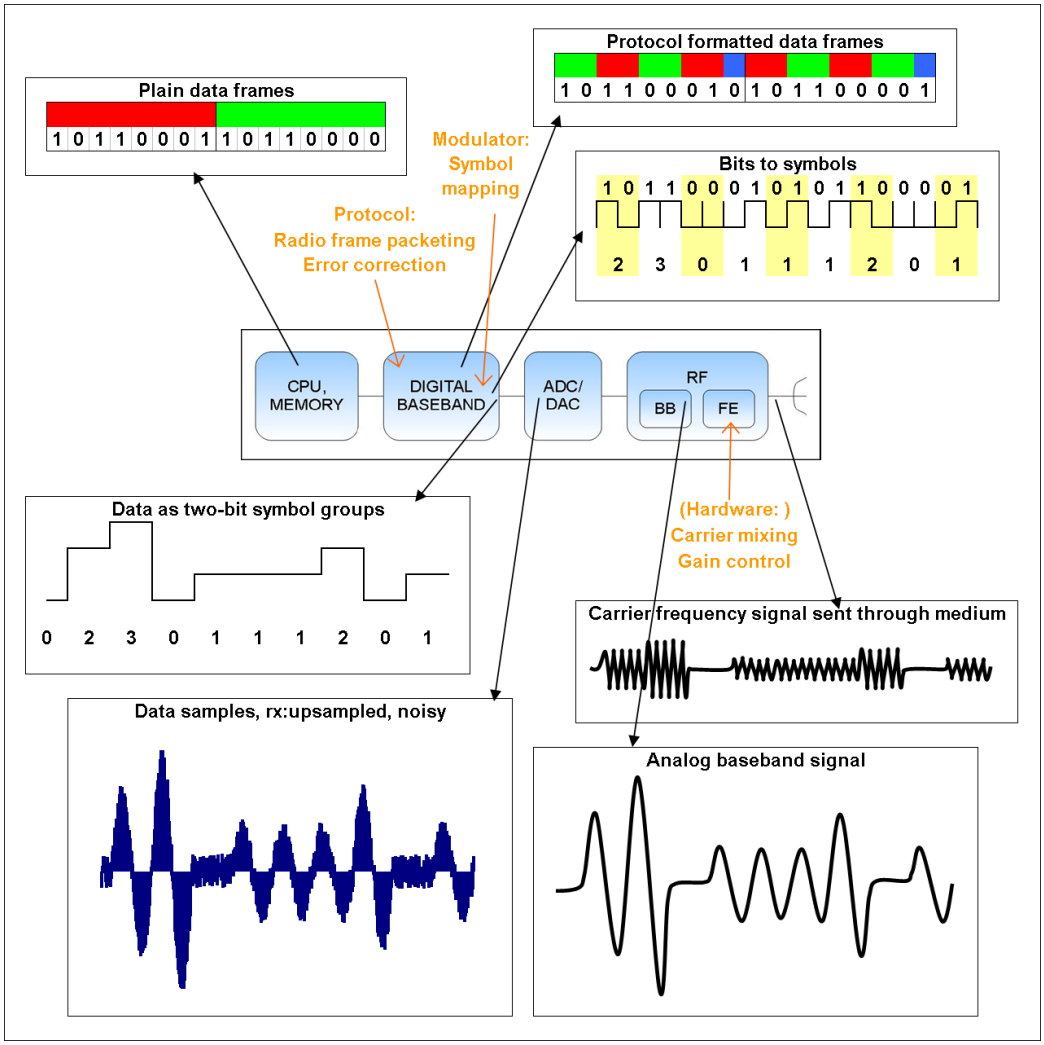

Figure 3.2: Transfer from data bits to RF signal. Modulation resembles that of two-bit AM.
As radio architectures very a lot, the presented stages are for reference only. E.g., placement of ADC/DAC depends heavily on the architecture.
Orange texts pick some funcionalities the radio gadget parts typically do.

channel. E.g. GSM standard requires Quarature Phase Shift Keying (QPSK) modulation to be used, and Bluetooth uses Gaussian frequency-shift keying (GFSK) [13, 9].

Baseband processing may also contain e.g. some checksum calculations, error correction, or bit (de-)interleaving for the protocol [8, 31]. In the digital baseband, the data bits may be upsampled[3], turned around, re-organized, or what ever is needed to implement the modulation scheme.

3.2.3 ADC/DAC

Analog to Digital and Digital to Analog converters (ADC/DAC) are there to transform digital bits to analog signals, or vice versa.[18] In different radio implementations, position of ADC/DAC may vary. The previously mentioned digital baseband could be implemented as an analog one, or ADC/DAC could be pushed yet closer to the antennae.

In DA conversion, the bits are just mapped to a signal, one or a few bits at a time. Bits or bit groups are encoded as certain voltage levels. There are several possible encodings for doing this. AD conversion, made during reception, is trickier than DA conversion. In AD conversion, used when receiving, we first collect samples out of the arriving signal. This is done at higher frequency than the wanted signal. The samples are then averaged to get out the data at correct rate. Averaging is to cancel out some noise. It reduces effects of high frequency noise. Our sampled stream of bits may still contain erroneously received bits due to the noise added to the original signal both in the transmit channel[4], and due to the noise our own receiver makes, intentionally or not.

3.2.4 Analog baseband processing

Analog baseband is the part of the radio where the data flows in format of an analog baseband frequency signal. That is, where the signal is a data rate signal.

Analog baseband may contain e.g. some filterings for the signal, to keep the signal from leaking from the wanted frequency band.

[3] that is, bits multiplied
[4] The transmit channel is usually air

3.2.5 RF front-end

RF-FE is the part of the radio near antenna, where the signal is at carrier frequency. Next we look into RF-FE functionality in more details.

3.3 RF front-end

The role of an RF front-end in a radio transceiver is to transfer the signal at base frequency[5] to the wanted intermediate or carrier frequency band(s) or vice versa. Within the RF-FE, the signals handled are all analog. The information carried within the signal mediated through RF-FE in might be ultimately digital or analog, but that is all the same for the RF front-end. To get the transmitted and received signals out of the RF-FE correctly, various filterings must be done. RF-FE takes care that the signal keeps in the frequency band intended, and that it does not leak the signal to unwanted frequencies. [13]

Other functionality the RF-FE does, is power control. The correct power levels for transmissions and receptions are set in RF-FE.

As technology advances, the ADC/DAC are taken closer and closer to the antenna, but for now, we still have analog RF front-ends. Ultimately, the signal that enters the air is analog any way – in the air, we have zeros or ones no more, but only the signal flowing in time.

To be able to complete the frequency transformation, the RF front-end has a bunch of things to be taken care of.

The *transmitter* must be able to send the signal at the carrier frequency band. The transmitter takes care that the signal emitted is strong enough, so that it can be heard. The transmitter must not distort the signal too much in any way, so the information is still understandable at the receiver end of the radio channel when caught. Also the transmission must not leak to the surrounding frequencies, which usually are strictly regulated by national or international laws or authorities. E.g., in Finland, the radio frequencies (from 0HZ to 400GHz) are regulated by the Finnish Communications Regulatory Authority's (FICORA). Frequency

[5]Base frequency is that of the transmitted data bit stream. That is, how much data bits are flowing through the RF-FE.

allocations are publicly available from their site, [10].

The *receiver* must be able to pick the signal from carrier frequency. There is usually need for gain control so that the amplitudes for the received signal are proper to give forward to the rest of the radio. A too weak signal can not be detected at the end, and with a signal too strong, information is lost due to signal cutting off at the high amplitudes. Signal to noise ratio must be kept good enough in order to be able to decode the signal properly.

3.3.1 RF front-end building blocks

The tasks of an analog radio front-end of a radio transceiver are implemented by various mixers, filters and amplifiers. Usually there is an Automatic Gain Control (AGC) block in addition, to adjust the gain in reception. AGC can be implemented in hardware, but is today often implemented Digital Signal Processing (DSP) software. In Figure 3.3, we have an example of a radio transceiver structure, including RF-FE blocks.

Mixers are there to add the carrier frequency with the base frequency signal when transmitting, or to take the two apart when receiving. Until recently, the mixing of the signals in transceivers has been made in two or more steps due to technological challenges. Theses so called superheterodyne transceivers use one or more intermediate frequencies between the base frequency signal and the signal at the carrier frequency. As technology has advanced, it has become possible to do the transfer in one step. These transceivers are called direct conversion transceivers. [13, 1]

A **synthesizer** is required to get the carrier frequencies in to the radio. The frequency oscillations of the synthesizer are formed e.g. with crystals vibrating at their natural resonance frequencies. The frequencies the crystals produce, are not exact, but they tend to jitter. A Phase Locked Loop (PLL) is a technique and construct to lock a crystal frequency to a wanted frequency. The frequency can be altered this way, as well as kept more constant than the one the oscillator would be able to produce on its own.

Filters restrict the frequency bands the transceiver uses. They are there to restrict the transmitted signal to the band we want to use, so that our transmission does

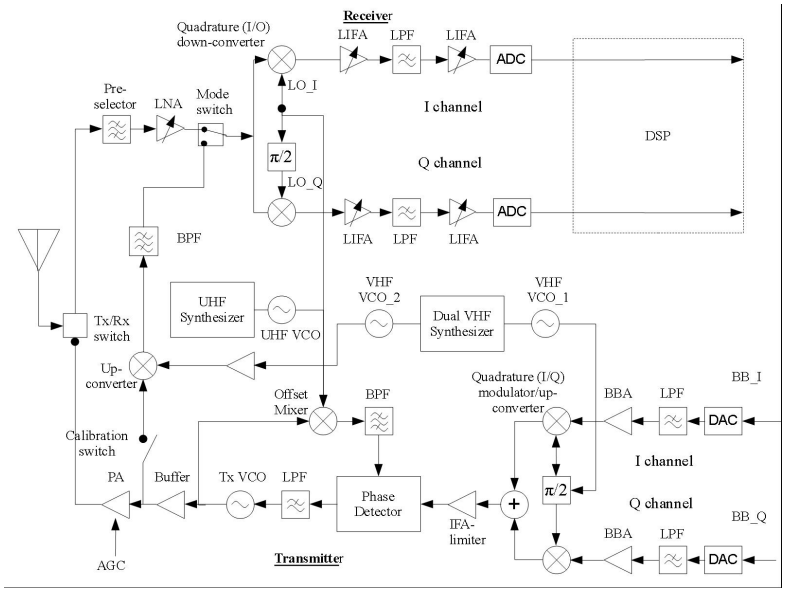

Figure 3.3: Example of radio transceiver structure.
Block diagram of low IF receiver and superheterodyne transmitter.
Figure based on: [13], p. 174

not disturb the other users of the radio path. There are also filters on the receiver side of the radio. Filters help in selecting the needed information from the wanted band, and in suppressing the signals at the frequencies not needed at the moment.

Amplifiers are needed in various stages of both the transmission and reception. As the signal arrives at the receiver, it might be very weak already, and not amplifying it would mean an immediate data loss. With Low Noise Amplifiers (LNAs), the signal is amplified as much as needed (within certain limits) without much loss in the signal to noise ratio. Also, when the signal proceeds through various stages of the transceiver, in many stages there is some energy loss, which lowers the signal amplitude. In order to keep the signal strong enough, it is amplified where needed.

A **Power Amplifier (PA)** is used at the antenna end of the transceiver Front-end (FE), or in the antenna system, ensuring enough power for the outgoing

transmission.

Automatic Gain Control, unlike the previously introduced hardware blocks, is usually a acDSP block. It is used to control the strength of the received signal. AGC strengthens very weak signals and suppressess very strong signals adaptively during the communication in order to maintain an appropriate level of the received signal.

3.4 Controlling RF transceivers

A plain RF transceiver is conventionally controlled simply by changing some registers values. Register values then control directly whatever functionality needed to twitch - PA gain, filter settings, Local Oscillator (LO) frequency, and so forth.

While the registers might be commanded straight from the application, some driver functionality is usually written to map the register twitchings into more abstract commands. This makes the usage of the RF engine more straightforward.

This far, a single RF transceiver has usually been used by a single radio system only. Therefore the control schemes used within a product have been widely known in advance. Even while still at the manufacturing state, the control interface may have been constructed as a quite straightforward mapping of protocol defined channels to register mappings. "Put the GSM radio on at channel 43, use power setting 4."

As there become more radio standards that utilize the same freely usable (or otherwise utilizable) frequency resources[6], and different data encoding techniques, simply mapping the protocol commands into register values is no more sane. Some control layers to abstract the usage of the RF engine would help to use the same hardware for different uses.

It would be useful, if any radio system could ask to use all the applicable radio parts. Digital baseband systems can already be implemented as more generic processors, usable by several radio systems, but by now, RF hardware has still been dedicated to a single radio system only. In the future, it could be possible for the radio system to say "Use any RF transceiver capable of transmitting at

[6]At the moment, there are a few freely usable frequency bands. Most commonly used is the one at 2,4GHz, used e.g. by Bluetooth and WLAN.

frequency 2,045GHz with 200kHz bandwidth, 100ms from now"[7] - and get its job done.

3.4.1 Protocols

All the widely used radio systems have been standardized to some extent, and each of them offer some sort of protocol to be followed. The radio system definitions vary much, as different systems have been planned for different contexts and environments, and so have their protocols.

Though any protocol answers questions about how to send the data and how to receive it, there is much variance in what they define. There are, anyhow, some things they all define. A radio protocol knows answers to the following questions: To whom, and how to listen to, when to listen, to whom, and how to transmit to, and when. What carrier frequencies and frequency bands are to be used, what is the used modulation, and what timing schemas are used.

3.5 Future radios

Until these days, each RF transmitter has been dedicated to a single radio system. Today, radio research is striving for Software Defined Radio (SDR) and Cognitive Radio (CR). SDR and CR are futuristic ways to build a radio system, both still under research. The original ideas of SDR and CR are described in [23, 24].

SDR in its purest form would be a generic processor with a software module attached directly to ADC/DAC conversion modules, which would be attached directly to the antenna system. With an SDR, any radio system could be implemented, by just picking up the wanted frequency and processing it with suitable software.

CR is a concept going still further than SDR. It has various definitions, or other way said, there are many different expectations to it. CR would be an 'intelligent' radio system able to adapt to its current environment on the fly on its own, detecting the changes in its environment, changing its operation accordingly to best suit

[7] Artificial numbers in the example. Just to show the difference in the way to command the radio.

the current needs, whatever they are. The CR could take advantage of the white spectrum[8] found in the environment. With CR, a mobile device could switch to another way of communication, another communication channel or communication protocol, when it ever saw a way better one than that currently in use. A cheaper connection, a connection with better quality, a connection with higher speed, or how ever we would want to define the goodness of a connection. SDR is usually seen as a suitable platform for CR.

Yet we do not have either of them, though MIMO techniques, (relatively) freely usable frequency bands made available, advances in RF and antenna technology, and the increase in the available computing power, for examples, these all bring SDR and CR closer to us. Still, there are many open questions to solve.

Many possibilities are offered by the increase in available processing power. The available processing power for a mobile device is growing due to more efficient data processing chips and better batteries. More of the signal processing can thus be moved from the traditional fixed printed circuits to more flexible FPGA chips, or even to general data processing chips.

The digital basebands of different radio systems can be seen to be mergeable to one generic processor. This is due to increase in current processor speeds, and relatively similar tasks the radio protocols require from the baseband. A sufficiently powerful generic processor for baseband functionality is able to handle baseband processing for multiple radio connections for several radio protocols concurrently. [26]

We will see, if the ADC/DACs can be taken next to the antenna system in the future for an every day products, but in the meanwhile, we may concentrate on looking the possibilities to reduce and re-use the components in current systems. Possibilities of merging digital basebands are known, but it is yet to be seen how much integration can be done to the RF parts. At least with currently available technologies, we can not get rid of all of the parallelism, but something can be done. One way is using programmable RF ICs [26], (or more generally by using software,) where there has previously been hardware (usually as fixed circuits).

Another way to reduce parallelism in RF circuits would be some technological advancements, which would offer possibilities to make the analog radio components more widely tunable[38], offering e.g., virtually unlimited ranges of frequencies and

[8] the RF spectrum not in use at that particular moment.

bandwidths, without considerable losses in signal quality, and without much extra power needed.

3.6 Summary

Different radio devices in one main device have until now been implemented as separate radio tubes. Evolution goes slowly towards systems where all the radios are implemented as one. Technologies include analog hardware development and utilizing generic processor speedup and more complex software.

However, the basic functionality of the radio stays the same. The data must be sent out as signals at wanted radio frequencies. To accomplish this, the same basic functionality must be implemented into the radio, be that in analog or digital form within the device.

Chapter 4

Modelling & used models

In this chapter, we look into modelling and simulation, and introduce both the modelling tools, and radio models utilized within this work.

We start by defining some basic concepts relating to modelling and simulation. Then we introduce Lyra, which is a modelling technique for communication systems. Lyra was used to build one of the models we got as a starting point for our work. We continue into world of model-based analysis, and specifically discrete-event simulations. We introduce SystemC, a C++ library, that allows discrete-event, or transaction level modelling, which we chose for the toolkit for our simulator. We end this chapter by introducing the two models we had as the base for this work, Software Defined Radio functional architecture model (SDR model) and RF hardware model.

General modelling sections of this chapter (4.1, 4.3, 4.4) lean on [20].

4.1 Systems

A system is a bunch of entities enacting together to reach a collective goal. An entity can be anything, depending on the particular system. It could be e.g. a human, a city, a valve, an ant, a processor.

A state of the system can be described by a collection of variables needed to catch the relevant information of the situation the system is currently in.

Systems can be categorized into continuous and discrete systems. In a continuous system, the state changes happen gradually and all the time. In a discrete system instead, the state changes are happening only at certain points of time, and they may contain however big changes in the state - and meanwhile, the state of the system stays constant. Of most of the systems both continuous and discrete attributes can be found, but usually either one is dominant, thus making the categorization still eligible.

4.2 Lyra modelling technique

Lyra is service oriented[1], four phased top-down modelling technique developed for modelling communication systems[21]. Lyra has been developed and utilized at Nokia. It aims in supporting the parallel development of software and hardware, and working as a verifying tool that may be used to formally verify the correct functionality of the system throughout the design process.

In principle, Lyra is language and tool independent technique, stating just some phases of the modelling, telling what kinds of things are to be thought of at each design stage. In practice, Unified Markup Language (UML) 2.0 and Telelogic Tau have been used with success as tools in various phases.

4.2.1 Lyra phases

Lyra modelling technique consists of four design phases that may be thought of as recursive modelling steps. The phases are called Service Specification, Service Decomposition, Service Distribution and Service Implementation. In Lyra, the high-level functionality is defined first. In later phases, it is sliced to smaller pieces, defining the functionality at ever more accurate level, ultimately reaching the implementable system.

Service Specification

In service specification phase, the task at hand is to define the services the whole system offers to the outside of the system. This is accomplished by specifying

[1] "service" is used within Lyra a bit unconventionally. That is, there are no servers offering services and reacting to requests. Instead, the offered functionality is called services. Thus, "service" must be thought of as "function" or "functionality" in this scope.

services as Computation Independent Model (CIM). Aim of the specification phase is to define, what the system should be able to accomplish. Particularly, it tells only *what*, but it does not tell *how* to do that.

Service Decomposition
In Service Decomposition phase, the internal architecture and implementation of the services are specified. This is formulated as Platform Independent Model (PIM). After this phase, the logical architecture of the system-level services should be clear. This should be done by decomposing the services to system components. These may use some external services, Used Service Access Points (USAPs), to implement the required functionality. Services defined at Service Specification phase are left untouched - this is all about defining them more precisely.

Service Distribution
In Service Distribution phase, the distribution of the services to a given platform is made. The services are implemented as Platform Specific Model (PSM). This phase is done bottom-up. Hardware (HW)-Software (SW)-partitioning is done at this stage, as well as choosing between different possible implementation techniques, e.g. direct conversion vs. superheterodyne transceiver in RF-FE, or between a DSP processor, and a FPGA implementation of an Intellectual Property (IP) block.

Service Implementation
In Service Implementation phase, the planned structural elements are integrated to the target environment. This is the phase for the things like routing and data encoding.

4.2.2 Lyra usage in this work

SDR model we used as the starting point for our radio simulator (described in Section 4.6) is made as Lyra model of phases Service Specification / Service Decomposition [3]. One of the interfaces we built the simulator on, was taken directly from SDR model. The presentation formats used by the SDR model - interface function definitions, and the structures of the code implied by them - were adopted and used in our simulator, where seen applicable.

4.3 Modelling

We have several ways to study a system. The most relevant way depends on the system at hand.

If we have an actual system, we are able to experiment it, and if experimenting is seen feasible - that is, cheap and fast enough - we usually want to do so.

In many cases, this is not so. Experimenting with the system may be too costly, or we might have not an actual system. If, in spite of these, we want to study that particular system, we may make a model of it, and experiment with the model instead of the real one.

A model can be made to a coarse grained one, or an accurate one. The model may take into account only a part of the bigger system, or try to grab on the whole system.

Models try to somehow represent the system they model. But however the model is made, it is always worth to ask, whether the model correctly represents the system in the aspects we want to study, and how well.

Models may be further divided into physical models and mathematical ones. For information processing systems, the physical models are not usually of much use. They can visualise the structures needed, but usually can not tell much about how the system operates. Mathematical models instead, can always be manipulated and tested to see how the system does react.

Simple enough mathematical models can be represented as equations, such as $E = mc^2$. However, in many cases, there is no way to stumble on a simple enough mathematical model, to be able to study it so that we would get an analytic solution for it. In this case, we may still take the model and simulate it running.

Simulating is a way to gather information about systems that are so complex, that we can not find an analytic solution for them. Either making a realistic model of the system would be impossible, or running it would not give us any answers, at least not within reasonable time limits. Simulating is done by testing the model with some (often numerical) input, and studying how the system does react to it, that is, what we do get as output.

Simulation models may be further divided into dynamic vs. static, These differen-

tiate systems by whether the time plays a role in the model or not, whether there is some randomness or probabilistic components built in the model or not, and whether the system changes happen continuously or only at certain points of time.

Our protocol-initiated radio system under research can be seen mainly as static, deterministic, and discrete-time system. Protocol initiates commands that then traverse discretely through the system. The state of the system does not change outside of the events, all is digitally static and deterministic. Static model means absence of gradual changes, and determinism absence of stochastic modelling, but for the discrete-event simulation computer model, we can state a typical structure. The next subsection describes this briefly.

4.4 Discrete-event simulation model

A discrete-event simulation model contains typically at least the following parts, if it is based on approach where the simulation advances directly to the next event in simulation time (adapted from [20]):

System state - a collection of variables needed to describe the system.
Simulation clock - variable keeping track of current simulation time.
Event list - list of events with information about when they will occur.
Statistical counters - variables storing statistical information about system performance.
Initialization routine - a subprogram for initializing the simulation model.
Timing routine - a subprogram choosing the next event from the event list and advancing simulation clock correspondingly.
Event routine - a subprogram changing the sytem state after a particular event.
Library routines - a set of subprograms making random observations from probability distributions seen as part of the simulator.
Report generator - computes estimates of the wanted measures and produces a report.
Main program - a subprogram running timing routine and event routines.

4.5 SystemC

SystemC is an extension of C++ programming language. It is a combination of simulation kernel and library functions. It is a tool for making discrete-event simulations. Intended use for it is electronic system-level modelling, design and verification of an electronic system. It can be used to model projects from a very abstract system functionality down to VHSIC Hardware Description Language (VHDL)-like hardware-level implementation. SystemC is an open source project lead by the Open SystemC Initiative (OSCI). For further information about SystemC beyond this section, see [33], and [7].

SystemC simulation kernel (later "kernel") offers the tools for running parts of the code virtually in parallel. That is, it offers a kind of threading system for the developer. The kernel has also an internal simulation clock, tracking the passing of simulated time.

With SystemC, you create subsystem parts that are intended to run parallel to each others, connect them with the kernel, put some timing functionality to the code if needed, and then let the kernel handle the running of the parallel system parts for you.

The kernel has a process queue, and a round-robin scheduler. The processes are executed in zero simulation time, and are thus not ever interrupted by the scheduler. Any timing information must be explicitly built in the process subroutines.

4.5.1 SystemC simulation kernel functionality and usage

The kernel requires the subsystems and functions of the subsystems that are to be driven by it to register with it. Each subsystem is basically just a C++ class, and the registration is done by wrapping them inside a macro that comes with SystemC.

The functionality of the subsystems may be freely constructed within the limits of C++. After registration, the subsystems are commandable by the kernel. In addition to kernel registration, the subsystems must be connected to each other via one or more SystemC ports or other SystemC channels where needed, and

the subsystems must be told to react to wanted events, in order to any kernel functionality to take effect. There are two kinds of events the subsystems may react to. These are actions happening through the pipes, and the simulation clock events.

When using SystemC for building a simulator, `sc_main`[2] function is to be implemented instead of the normal C/C++ main function[3]. The constructed `sc_main` is called by the real main function, which is embedded within the kernel. The `sc_main` is used to build the whole simulator with all its subsystems, and to finally to pass the command to the kernel. The pipes and their connections are created in `sc_main`, and the real main function makes the final connecting of the subsystems before starting the simulation with the model.

The kernel keeps track of internal simulated time with the simulator clock. At kernel start, kernel runs once all such functions registered to it that are not explicitly set to not to start at startup. Functions are run one at a time, in undetermined order. A function runs until the end of it, or until a SystemC wait function[4] is called, whichever comes first. Now the control is handed back to the kernel, which selects another function into execution. When all the functions to start at a startup (and those triggered to start at the same time by the other functions) have run, the kernel picks up the next moment in the simulation time when something is going to happen at the registered subsystems. Kernel then runs all the functions to be waken at this SystemC time step - one at a time, and at undetermined order. A function is waking either from a sleep[5], or because it got an event to its port to react to. This way, the functions are run at the needed SystemC times, and those only, up to the SystemC time limit that has been set at start, or up to when there is nothing to run anymore.

[2] `int sc_main(int argc, char* argv[])`
[3] `int main(int argc, char* argv[])`
[4] `sc_wait(...)`
[5] wherever `sc_wait(...)` was set to wait for a limited time

4.6 Starting point: SDR model

Software Defined Radio functional architecture model discussed in [3, 4] was used as a starting point for the radio control simulator. SDR model describes the wanted functionality but leaves architecture and timings undefined. SDR model is made as a Lyra Service Specification / Service Decomposition model. It describes functionality of a radio platform. The platform offers the possibility to use multiple *radio systems* at the same time; at the same time both in a sense of concurrently used radio tube and of parallel execution in multiple tubes.

A radio system stands for the functionality and/or protocols which define a way to use the radio hardware. Frequency and timely usage of the resources, data encoding schemes, and some minimum requirements for the radio device are often defined in standards. For example GSM, WLAN and Bluetooth (BT), are radio systems.

Our aim, considering the SDR model, was to get a proof of concept for part of refined functionality for a Lyra Service Decomposition phase modelled Devices block (see Figure 4.1). Thus, we were to find whether the functionality description offered by the Devices model is adequate and given at suitable accuracy.

The functionality offered by any block in a Lyra model is referred to as services. For this work, we got a set of these services for Devices block to implement. These included services about e.g. timing issues, data access, and radio system initiation.

4.6.1 SDR model blocks

Figure 4.1 shows the SDR model building blocks. In the following, we give short descriptions about the most important parts of SDR model, as seen by our radio simulator.

Devices
Devices is the block whose inner organs we were interested in within this work, and whose inside we were to model. This block contains radio baseband and RF functionalities, plus the probably needed controller functionality. RF engine hardware resides logically within Devices, too.

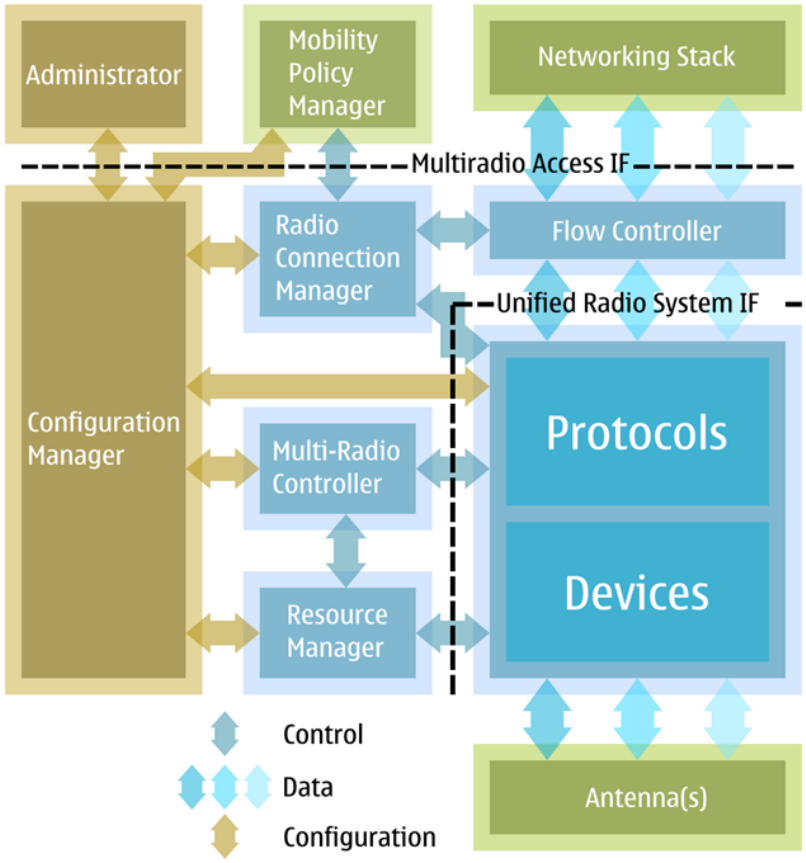

Figure 4.1: SDR functional architecture. Adapted from [35]. Courtesy of Nokia.

Protocols

Protocols is a logical block taking care of radio system specific functionality. An instance of Protocols knows about timing and frequency-related behavior of a radio system. Protocols gives Devices the orders for transmissions and receptions, after asking, and getting permission from MRC.

Multi Radio Controller

Multi Radio Controller (MRC), is responsible for fine-grained resource managing. MRC decides if a radio system may use the radio frequency hardware or other resources at the requested time. MRC handles scheduling of the different radio system transmissions and receptions with its own, dedicated time concept. Time

concept is used to relate the different radio system times to each other.

Resource Manager
Resource Manager (RM) is responsible for coarse grained resource managing. RM decides, e.g., whether a new radio system can be put run at the same time with already running ones, and whether a radio system may be granted certain amount of the resources. Resources of a single radio device might be under control of a distinct device specific sub-RM.

Antenna
Antenna is connected to the RF hardware of the device. It provides the way to access the medium. Controlling the antenna was not considered within this work.

Other blocks
The rest of the functional blocks are not directly connected to the Devices, and are thus unimportant within this work.

4.7 Starting point: RF transceiver model

The interface between SDR blocks Devices and Protocols was the starting point for the work, and the most important interface for our simulator. The other interface the simulator utilizes, faces RF. We did not have a physical concrete functional radio platform as our RF hardware within this work, to test our software with. Instead, we got a transaction-level model of the RF transceiver hardware (the "RF model"), one whose interface was still evolving while constructing the simulator.

The RF model we got is constructed with SystemC. It functions as a sink and source, or more precisely as a loopback, for the modelled radio traffic going through the SDR Devices.

The RF model consists of transmitter (TX) receiver (RX) clock circuit (SX) AD-C/DAC parts. These parts of the RF model further consist of functional models of radio blocks. E.g. amplifiers, power switches, mixers, and AGC functionality can be found.

The RF model models changes in the physical features of the radio signal format that occur when the signal traverses through the radio pipe. At the antenna end of the receiver, the data signal is described with transmission frequency range and

some tracking information about the modeled data signal. Timely information is encoded into the model, too, though it is hardly seen in the signal model itself. At the ADC/DAC end of the RF model, the data is no more described as a signal, but instead as the description of a data frame, with information about the number of bits carried, and some state information about whether the frame is intact or corrupted by noise (if receiving).

4.7.1 RF interface

The RF model offers us an interface that has services (methods) for setting the radio transceiver power states, carrier frequencies, frequency bands, and ADC/DAC resolution and clock frequency.

This level of hardware abstraction was seen as a good compromise between generic radio transceiver usability through simple unified interface, and possibility to control the radio transceiver as directly as possible. This abstraction level leaves some things to be done on the RF side of the radio interface. Examples of functionality left out of our controller simulator are e.g. AGC[6], and accurate radio specific power control schemas[7].

At the SDR Devices block level, we want to be able to command the RF hardware at unified way, instead of twitching the multiple different implementation dependent variations of the hardware. It is preferred that the gain control just works so that no receivable bit gets lost due to incorrectly set gain level, but in such a way that any upper level controller does not need to take care about different gain levels. Also, it is preferred that the power is saved as much as possible, but that the RF is still ready to work whenever needed, without the Device needing to know the explicit implementation of the hardware power saving schemas.

[6] AGC is usually made as firmware in a simple software loop that controls gain settings. These settings are individual for each RF implementation. A generic radio controller does not want to touch these.

[7] For power control, any radio transceiver offers at least two states, on and off. However, there might be many more, letting different parts of the radio to go to sleep independently. Deciding how to manage these is left for the transceiver to do on its own, as a higher level software does not want to know about tens of different variants of the same functionality.

4.8 Summary

Modelling is the way to examine systems that are too complex, or otherwise hard or impossible to examine directly. Characteristics of the still yet unimplemented system may be approximated this way as well.

Computer systems, running on a clock, are typically discrete systems. Discrete-event models suit well for examining them. This case is no exception.

We got two different models to deal with within this work. The SDR model is a Lyra model of a radio system, and the starting point for this work. The RF transceiver model is a RF radio model playing the hardware for us.

Our radio simulator model is a discrete event model, having its roots in a Lyra model of a radio system. It is implemented by SystemC. SystemC kernel readily offers us parts of a typical discrete-event simulation model, namely simulation clock, initialization routine, timing routine, and the fixed part of the functionality of the main program.

Chapter 5

Radio control simulator

In this chapter, we describe the radio control simulator implemented.

We state the questions we aimed to answer with the simulator, and after this, we look at the structure of the constructed simulator, look in its architectural decisions, and see some excerpts from the implementation.

5.1 Simulator scoping

The simulator implementation is constructed on two cornerstones, both introduced in Chapter 4. These are the the SDR model and the RF model, or particularly their interfaces - the SDR Devices interface implementation in the SDR model, and the generic RF radio hardware interface in the RF model.

Studying and understanding these models was a considerable part of this work. These two models set the framework for our simulator, stating what is to be modelled and what left out the simulator.

The SDR model gives us an interface to be implemented. The functionality offered by the interface is however, complex, and only parts of it were chosen to be implemented within our simulator. A complete implementation of SDR Devices interface would take care of loading used radio systems in and out, handling link synchronization with the base stations (if required by the radio system), and handling receptions and transmissions. Implementing recepting and transmitting was

considered a large enough piece of work alone, and thus the other parts of the functionality were left out from the simulator where possible.

The RF hardware interface we got, on the other hand, must be fully implemented, if the RF is to do anything correctly. The given interface offers us just functions that take in numeric arguments resembling physical quantities, e.g. frequency, of the radio signals that are wanted to use. Frequencies as well as other physical quantities must all be set correctly before the signal can correctly pass the RF hardware.

What was left as our simulator scope, is basically two things. The first thing is translating given transmission or reception related commands (given by SDR Devices interface) into a form that the RF engine understands. The second thing is giving the orders to RF engine at correct enough timings, so that it can accomplish its job.

5.2 Implementation framework

For our simulator, we searched for tools that allow us model the control at suitable level of abstraction, and to connect us with the provided RF model.

The implementation framework was chosen amongst Telelogic Tau with UML, Java, and C++ with SystemC. SystemC was chosen.

Telelogic Tau with UML was used for the SDR model construction. These would allow the modelling of the SDR Devices block based directly on the SDR model already constructed. On the other hand, Tau was an unknown tool, probably needing much effort to familiarize with. Also, connecting the Tau model (one with a very high abstraction level) to other available, probably low-level models, was considered hard, if directly possible at all.

Java is a language that would allow probably shorter development times than C++-based SystemC, partly because the language structure, and partly because is was already a familiar tool. On the other hand, as far as we know, Java lacks connections to suitable low-level modelling tools. Also, Java does not support low-level coding, would that have been required.

SystemC is a C++ library designed to make system simulations, and it offers a

readily built simulation kernel aiding with the simulator construction. Also, there are means to connect the SystemC code to various levels of code, from C++ code to VHDL-level modelled code. SystemC was an unknown library for us. However, it has fairly good documentation available, and it is C++ library, a library of already familiar language.

We chose SystemC as the implementation framework for the simulator. Integrating the control side of the simulator with the RF hardware model is easy enough with SystemC, independent of the abstraction level of the RF HW model we have[1].

As SystemC is a C++ library, high-level or abstract models made with SystemC/C++ are easily attachable with it. But also some signal-level modelling tools offer us a readily made SystemC interface, allowing us to attach our high-level model of the controller software to a low-level model of the RF hardware easily, would that be required.

5.3 Implementation

The created simulator model is characteristically a SystemC model. Much of the modelling made with SystemC is about creating the subparts of the system, called modules, and ports with suitable functionality, and connecting these together. Our simulator consists of several modules representing different functional parts of the system.

Our modules have quite a simple structures inside them in addition to ports and connecting them. That is, most of the functionality is built within ports. In our implementation, a SystemC ports function either as handles to the module functionality, or as buffers for data or commands - or both.

Structures for radio system dependant functionality are very complex if fully implemented. They would require e.g. data link models to really work. Due to scope of this thesis, radio system structures were stripped down to amount of information just enough to run the simulator with the fixed input.

The remaining data structures used in the implementation are mostly various

[1]Before getting the transaction-level model we now have for the RF hardware, a lower level (or a more detailed, or a hardware oriented, or a logic port level) model of RF hardware was considered by our partners.

tables. These contain much of the translation information, needed to translate e.g. from time format to another, or from radio channel information to frequency information.

Implemented algorithms are typically mappings from values to others, finally leading either to setting of some physical properties of the radio tube, or to launching a certain event at the wanted time. Logic in the algorithms is mostly at the level of if clauses, like "If port A gives you value X, put value Y in the port B". Somewhat more complex algorithms are found when dealing with time concepts. Time translations from device system time to radio system (base station) times are mainly linear translations, maybe added with some resets of the clock counters.

The commands are arriving at the SDR Devices in order of execution, as the operating schedule for the RF HW is made outside of SDR Devices. As this results in using translation tables for the given arguments, and in some time calculations, but not in complex or long-lasting calculations, there seem to be no reasons for using scheduler, or processes, or the like more complex concurrency handling. Interrupts may be needed, depending on the actual real implementation. In the simulator, the event-based functionality of SystemC simulates interrupts.

The commands are executed in order, and are in a basic case no longer needed after usage[2]. This states, that a filesystem functionality is not required for the functionality, but simple buffers[3] will be required at the borders of the modules.

What is needed, is some kind of accurate enough latcher that passes the translated commands for the RF at just correct time. If the RF HW itself has a buffer big enough[4], the RF could be able to handle the latching correctly on its own.

5.3.1 Simulator block structure

In the following, we briefly look at the structure of the simulator. Figure 5.1 presents the block structure of the simulator.

System

System "is" the simulator, being the only block needed to call to construct the

[2] a more advanced system might re-use the commands e.g. in periodically utilized systems such as GSM.

[3] circular or other FIFOs will do

[4] "big enough" does not to be that big really, some bytes should be enough

whole simulator. It is a container and connector of more detailed blocks.

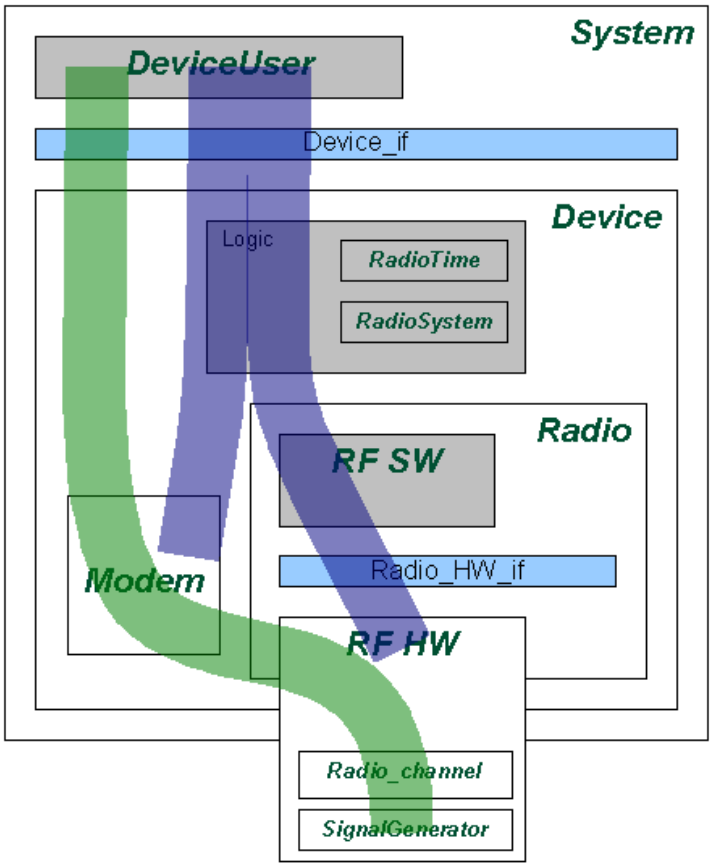

Figure 5.1: Simulator block structure and control and data paths. Control flow is presented as blue line, data flow as green line.

System connects DeviceUser and Device blocks, which represent the protocols and all the upper level functionality as needed, and the peripheral device, respectively. DeviceUser and Device communicate using service requests and responses defined by SDR Devices. These are mapped into the Device_if interface, which consists of two interface classes, one for each caller.

DeviceUser

DeviceUser is a construct simulating any parts of the SDR model that communicate with the SDR Devices. The parts of SDR model which use SDR Devices services are SDR Protocols and SDR Multi Radio Controller.

The DeviceUser functions as a fixed load generator of the simulator. Since simulating any particular radio protocol was not at the bullseye for this work, it contains no generator algorithms, but just pre-defined command lists that are then fed to the Device. DeviceUser sends GSM and WLAN related commands for the Device.

Device

Device block is a container for SDR Devices implementation. It contains RF and baseband parts of the radio device, as well as controller structures needed to translate SDR Devices requests to the format the further contained blocks can understand.

The control structures of the Device save information about the radio systems that are in use, and the algorithms for translating the commands from the Protocols to the format the Radio understands. In addition to giving commands from User to Radio, some commands must be given to the Modem, too, so that the data gets flowing.

Modem

Modem stands for a digital baseband, or similar functionality. It is a skeleton model of a device that could do e.g. framing of the data, and calculating and addition of the error correction bits to the data frames. Functionality offered by our Modem is minimal. The "data" flowing through it is simply some numeric descriptions about what would there be, were the real data there. Modem does quite simple type conversions for the data it gets, reflecting the processes there would be in a real situation.

Modem was initially thought to be fully left out of this work, as it was seen to be out of the scope of this work. However, including it was finally seen as a must, since we were required to validate the radio control by driving at least some dummy data frames through the system. In the end, what we want the radio to do, is to get some data flowing in and/or out. A control that fails to provide a connection to the radio network for the user, for DeviceUser in our case, is of no use.

Radio

Radio block is a container for the signal handling RF front-end plus ADC/DAC

functionality. RF front-end further contains the hardware for the RF front-end, and the software logic for the radio to translate the given generic radio commands for the hardware.

Radio is accessed through a generic radio interface. This may be thought of as a HAL to group up some control events given to the RF-FE that are repeatedly called as certain patterns.

Radio Logic (RF SW)
Radio logic translates the generic radio usage commands given by the Device logic into the form the RF HW understands.

Radio logic handles things like translating generic radio commands such as "transmit on" and "use band 2,014 to 2,034 GHz" to the correct functionality of the RF HW. A simple generic radio interface command may translate to e.g. setting some registers to states that correspond to the wanted functionality.

Radio Hardware (RF HW)
RF HW, described in 4.7, is the radio hardware we ultimately wish to control, letting the radio user connect to the radio network, transmitting and receiving the messages to and from the network as radio signals.

The interface offered by this external module was used "as is". A control logic programmer takes the RF HW as a black box, just using the offered interface.

ADC/DAC functionality (not seen in the figure) was added to the given analog RF hardware model for simplicity of usage and for a bit of modelling accuracy.

5.3.2 Example: sc_module construction

In Figures 5.2 and 5.3 we have an excerpt of code about typical usage of SystemC, used to create a SystemC module.

In Figure 5.2, we have the header file of a module.

SystemC functionality is first made accessible by including *systemc.h* at line 2. Starting at line 10, we have a definition of a class inheriting SystemC module class *sc_module*. This makes our class connectible to SystemC kernel.

At lines 13-15, we have the subsystems of our *System*, namely *Device*, *DeviceUser*,

```
1 ///// System.hh /////
2 #include <systemc.h>
3
4 namespace simulator {
5
6 //forward declarations for pointers
7 class DeviceUser;
8 class Device;
9
10 class System : public sc_module {
11
12   /// System parts ///
13   DeviceUser*     user;      // System outside of SDR Devices
14   Device*         device;    // SDR Devices implementation
15   Radio_channel*  channel;   // ``air''
16
17   // Connections between system parts
18   sc_fifo<Data_frame>       fifo_user2dev;
19   sc_fifo<Data_frame>       fifo_dev2user;
20   sc_signal<RF_frame_list> dev2channel; //Contains RF frames
21
22   SC_HAS_PROCESS(System);    // Inform SystemC kernel
23                              // about this module
24 public:
25   System(sc_module_name name);
26   ~System();
27
28 };
29
30 } // namespace simulator
```

Figure 5.2: Typical SystemC module code, header file.

and *Radio_channel*. These are all SystemC module inheritors. They are defined as pointers for faster compiling and possibility to change modules on the fly, without the need to compile the whole system again. The classes are forward declared at lines 5-6 to avoid compile time notifications about non-declared data structures[5].

[5]In a real life implementation situation, a module could be an IP block, made by some other implementor, and not yet available. However, it is often enough for us only to know the interface it uses, and thus the functionality it must implement. Our parts of the program can still be compiled, without the exact implementation available.

```
1 ///// System.cpp /////
2 System::System(sc_module_name name) : sc_module(name) {
3
4   user      = new DeviceUser("user");
5   device    = new Device("device");
6   channel   = new Radio_channel("channel1");
7
8   user->make_connections(device, fifo_dev2user, fifo_user2dev);
9   device->make_connections(user, fifo_user2dev, fifo_dev2user);
10  device->to_channel(dev2channel);
11  channel->inOwnTx(dev2channel);
12 }
```

Figure 5.3: SystemC module code, .cpp file.
Example of module creation and sc_interface connecting.
Namespace and includes (systemc.h, and header classes for the submodules) are left out for simplicity.

After the modules we define the channels that are to be connect our subsystem modules together. This is done at lines 18-21.

At line 22, we register our module with SystemC kernel. *SC_HAS_PROCESS(...)* is SystemC defined macro that makes the connecting for us, and the argument given to it is the name of the connected class.

The constructor in the format seen at line 25, taking argument *sc_module_name*, is required by SystemC. The destructor at line 26 is required for destructing submodules when they are not needed anymore.

In Figure 5.3, we see an excerpt of the module implementation file.

The constructor of the inherited *sc_module* is called at line 2. At lines 4-6, the submodule constructors are called, and so do the pointers to the submodules get their contents. At lines 8-11, the modules are connected together. Lines 10-11 represent the common way to connect the modules (*to_channel* and *inOwnTx* being ports), while lines 8-9 represent a self created, custom way to connect multiple ports with a single function. Each submodule, if containing further submodules, takes care of connecting them on its own.

By creating an instance, we have a module connected to SystemC kernel, ready to be used.

5.3.3 Example: sc_interface usage

In Figures 5.3 (familiar from the previous example), 5.4, and 5.5, we have code excerpts about the usage of *sc_interface*s in this simulator.

```
1  ///// Device_if.hh /////
2  class if_device : virtual public sc_interface {
3  public:
4    sc_port<if_deviceUser> U_if;
5    . . .
6    virtual void
7    make_connections(if_deviceUser* user, . . .) = 0;
8    void reqFunction(Param_a &p);
9    . . .
10 }
11
12 ///// Device.hh /////
13 class Device : public sc_module, public if_device,  . . . {
14   . . .
15 };
16
17 ///// Device.cpp /////
18 void Device::make_connections(if_deviceUser* user,  . . .){
19   this->U_if(*user);
20   . . .
21 }
22
23 void Device::reqFunction(Param_a &p) {
24   Param_b p_(p.getParamPart());
25   U_if->doSomething(p_);
26 }
```

Figure 5.4: sc_interface usage example (1/2).

We have *Device* and *DeviceUser* modules, that are connected to each other through *sc_interface*s. Both of them have an interface they implement, and a port to attach the other module to.

Connecting is initiated at *System* module, calling *make_connections* functions at lines 8-9 in Figure 5.3. *make_connections* is a function declared in an interface

class, as in figure 5.4 at line 7 in *Device_if.hh*, and implemented correspondingly in *Device.cpp* (line 16 in the same figure). Within *make_connections*, SystemC port(s), also defined in the interface, are connected to the ports given as arguments. Interface ports are ports that are used to call the communications partner, and ports are given as pointers to the communication partner. In this extract, we have an *sc_port* of type *if_deviceUser* defined in *if_device*, and correspondingly *if_device* typed port in *if_deviceUser*. Device and DeviceUser then connect the ports in *make_connections*. The real connection mechanism in itself resides in readily made SystemC libraries, in *sc_port*.

Interface usage in itself is then easy. The interface functions are called for the port the communications partner is connected to, as in line 17 in Figure 5.3 and in line 25 in Figure 5.5.

The rest of the Figures 5.4, and 5.5 are presented for required inheritances for the code to work with SystemC kernel.

```
1  ///// DeviceUser.hh /////
2  class DeviceUser : public sc_module, public if_deviceUser{
3    void make_connections(if_device* device, . . .){
4      D_if(*device);
5      . . .
6    }
7    void ackFunction(Param_b &p){ . . . }
8  }; // class DeviceUser
9
10 ///// DeviceUser.cpp /////
11 DeviceUser::DeviceUser(sc_module_name name)
12 : sc_module(name), . . .{
13   SC_THREAD(aFunction);
14 }
15
16 void DeviceUser::callingFunction(){
17   c_Param p();
18   D_if->reqFunction(p);
19 }
```

Figure 5.5: sc_interface usage example (2/2).

5.4 Simulation propagation

The simulator can be run by calling the binary from the command line. The length of the simulated time period can be given to the simulator as an argument, if there is need to restrict it.

The simulation run starts at SystemC kernel code, where the main function resides. After the initialization of the simulator, SystemC kernel runs the simulator modules, feeding them radio protocol commands, until there is nothing to do anymore, or until the given simulation time limit has been reached, and then returns.

In the following, we look more closely into the details of the simulator run.

5.4.1 Initialization

Simulator initialization starts in `sc_main` function. The simulator parts are created and connected, and connections checked.

Start at sc_main

The simulator initialization starts by SystemC kernel calling the user generated starting function, `sc_main`. It is up to this function to register the simulator components into SystemC. (Subcomponents can connect their further subcomponents on their own.) When the `sc_main` returns, the execution continues at SystemC kernel code, starting the simulation with the registered components.

In Figure 5.6, we have the required parts of our `sc_main` function. Our `sc_main` creates our simulator container module *System*, and after it calls the `sc_start`, which initiates the simulation of all of the modules at SystemC kernel. Simulation time limit is also handled within this function, if set.

```
1 int sc_main(int  argc, char* argv[]) {
2   ... // check arguments for the simulation_time
3   simulator::System ras("System");
4   sc_start(simulation_time);
5 }
```

Figure 5.6: Starting the simulator.

System creation

System module "is" the simulator in a sense that it gathers up all the simulator specific data structures and functionality. All of the submodules are created recursively by the *System* creation (Figure 5.6, line 3). Within each module, the module structures (class structures) are created just as in a traditional C++ class. Due to SystemC, there is also something else to be done. The module itself is to be registered to SystemC kernel, and so are the interfacing functions that are either to be attached to ports, or to be initiated by the SystemC kernel[6]. Submodule ports are also connected at module instantiation.

Connectivity check

Once SystemC kernel has got the instruction to start the simulation (*sc_start* at line 4 of Figure 5.6), it begins driving the modules given to it. A check is made to see that all the ports of the modules are connected. An unconnected port typically stands for a design flaw, and any of these must be corrected before the simulation run is started. SystemC kernel prints an error message and refuses to proceed, if unconnected ports are found. When port check is passed, the simulation may starts.

5.4.2 Simulation

The simulation is controlled by SystemC kernel. The SystemC kernel has a simple scheduler in it, and it also keeps track of simulation time.

Kernel schedules the methods and threads given to it to run, finding the one with the earliest starting time and letting it run. The methods and threads given to the kernel either have a time stamp for when to run, or are started by an event happening at a port. Events occurring at the same moment in simulation time are run in undefined order. Once a method or thread is started, it is run until it stops executing. This is done either by leaving wait until a certain simulation time, or until reaching the end of its code. Whatever happens within the method or thread, is considered happening in zero time, so the timely proceeding, if any, must be built within the system if needed.

[6]Functions attached to ports are implemented as *SC_METHOD*. Kernel initiated functions are run only once, often at te system startup, and are implemented as *SC_THREAD*.

5.4.3 Control

Control commands of the simulator are given by DeviceUser to Device through the Device_if interface. They always contain a way to identify the radio system that the command is for. What other information is given in addition to the identification information, depends on the service used.

Operations service is responsible for recepting and transmitting data signals. Propagation of control for this service is given as an example.

Subservices of Operations commands carry with them a notion of time and the channel of the radio system which is to be used as arguments. The time is there to tell when the command is to ultimately make something flow out of or into the Radio. The channel contains the information about the frquencies that are to be used. The command must lead to a correct transmission or reception in the RF engine.

Inside the Device, device controller software looks into its inner states to see what does the given radio system identifier mean. It looks if there is such a system at all, if it is ready to use, or must it be initialized first.

Some kind of a time conversion takes place here, too. The different radio systems have their own notions of time, and ways to structure and arrange all the communication taking place. The point in time given through Device_if is given as a generic device system time, in a generic time format, used for any radio system residing in this particular physical device. Now the Device control software must decide, what this does mean in reference clock cycles, to be able to command the radio engines at the correct times.

The channel information, given in the service request, must also be dug from the radio system specific inner structures of the control logic of the Device block. As it is not a thing for the RF engine to know what the particular channel means, the channel information is translated to frequency related knowledge, e.g. to carrier frequency and bandwidth.

All the translated information the Device logic now has, is then given to the Radio, particularly to RF SW. Radio translates the frequency information further to suitable register values, saves them and stays waiting to another command which tells it to turn the hardware (either transmitter or receiver) on (and later

off), according to these saved settings. The timely information is not given to the Radio as a value to process anymore, but the Radio just proceeds immediately doing what it is commanded to do, at its own pace[7].

5.4.4 Data propagation & Data models

Study of data propagation was not aimed at within this work, but it was given some attention in order to understand the other functionality needed by the control in the Device.

Any real data is not required in order to validate the control. It is enough that we get some notion about our "data" passing some control points. In this work, the data is described as objects that contain some information about the format the data currently has within the data pipe, and some information about the chain of steps the data has flown through.

In TX direction, data propagates from DeviceUser to modem, from Modem to Radio, and inside the Radio from DAC to RF HW.

We call the objects representing data the DeviceUser sends to the Modem (or vice versa) *DataFrames*. These objects represent the plain data stream, where packaging or framing, is omitted.

Data objects that exit the Modem in order to enter Radio are called *SampleFrames*. SampleFrames are to track and describe the changes the Modem makes to the data. The data packets may be e.g. interleaved, or they may have error correction bits added to them.

SampleFrame continues its journey towards the RF HW inside the Radio. In the ADC/DAC it gets transformed into the form of *RF-Frame*. RF-frame is model for a data signal, transformed into analog format in DA converter. The data stops being a digital data frame, saveable in a memory, but starts being a signal.

The same data path and data formats are used at RX, but naturally to the reversed direction.

[7]This is an architectural decision leading to a requirement that the speeds of the Radio being able to tune itself, or to switch itself on or off, must be known at the Device logic level.

5.5 Output & Results

Program output has the appearance of Message Sequence Charts (MSCs). From these, we can track down the propagation of the control commands running through the simulator.

From the runs, we see that the control is able to reach the RF HW, and give it commands that make it transmit and/or receive the signal at the frequency we wanted. Control travels from DeviceUser[8] through the Device interface to Device controller software, and further into Modem and Radio.

As the result, we find that the control interfaces given to us, namely SDR Device interface and RF hardware interface, seem to be sufficient to drive the generic radio hardware model. Differing protocol skeletons, those of GSM and WLAN, were used to drive the RF model. This shows us that the interfaces do work well for a generic multiradio.

5.6 Summary

Radio controller simulator described here is based on two external models, Software Defined Radio functional architecture model (SDR model) and RF transceiver model. It is implemented with SystemC. SystemC was chosen due to co-design reasons.

The simulator size is about 6000 lines of code[9]. Simulation runs, including simulator initialization with SystemC kernel, take a few seconds for the test sequences we have.

The simulator consists largely of SystemC structures, as is typical of SystemC simulators. There are modules that implement the functionality, and ports (channels) that connect the modules together. The modules coarsely represent chips on a device, but more at the level of thought than that of a real implementation. The simulator is too simple to be used for model-based computing, but it can be used as an aid for further development of the radio model, that is, it can be used to model-based design.

[8] representing SDR Protocols and SDR MRC
[9] The RF hardware implementation excluded

We ended up with a simulator that has hierarchical module structure, describing coarsely the chips on a board. Modules are constructed as SystemC classes, classes that inherit basic module class of SystemC, `sc_module`. Module hierarchy is there both to help understand the functionality the Devices block implements, and encouraged by SystemC, which readily offers threading system for the modules.

There are practically no operating systems like structures, but this is more due to the high abstraction level of the simulator, and simplifications, than due to that the system would not require such. Some buffers and timers/latchers were required, but implemented almost implicitly with SystemC. With a more complete implementation, some basic memory space allocation schemes would be required. If the radio system loading into the device would have been implemented, some way to allocate memory from the controller memory space safely would probably have been required. Loading a new radio system into the device was out of our scope for us, so we did not need memory space control either.

Abstraction level of the given radio interface seem to be wisely chosen. They could indeed serve both of the tested (though simplified) radio protocols, GSM and WLAN. Based on this experiment, they could be used as interfaces for a generic multiradio.

Chapter 6

Conclusions

In this work, we were to create a simulator model of a generic radio device (RF device) controller. The simulator was to test given generic radio interface functions for their usability at controlling a generic RF transceiver. We were also to see, what kind of control software would there be in a radio controller, and if there is any match for the operating systems technology.

The abstraction level of the implemented simulator was lifted up during the project, when the complexity of the system was revealed. Abstraction level of the simulator is too high for us to make much conclusions about the similarities between the generic radio controller software technology and operating systems technology. Not much operating systems technology was required for the simplified functionality of SDR Devices now implemented. The most complex structures required were buffers, timers and latchers. This implementation can not, however, state whether more sophisticated system structures would be preferred or required. E.g., realistic schedules with even approximate execution times are not implemented here, unlike planned at the beginning, making it hard to say anything about required concurrency related software.

The constructed simulator can be used further for model-based design of a more accurate model of the multiradio system, continuing the development according to Lyra or other modelling technologies.

Controlling the RF engine through the given SDR Devices interface was shown to be possible for concurrently running radio systems with the given RF engine

model, and RF hardware interface. That is, the SDR devices interface offered the required functionality for RF radio usage, and RF hardware interface using physical quantities is seen to be as generic as possible, leaving implementation dependent functionality for the RF harware block to implement.

The SDR model sets some requirements for the implementing software platform. SDR Devices block[1] must have either built-in knowledge about the radio system functionality it runs, or a way to get the radio system related data loaded inside it, and time concepts of each radio system must be translated into a uniform time format that ticks the time of a common, system wide, clock. RF engine functions must be executed on that time.

6.1 Future work

The implemented simulator is a functional model only, concentrating on configuring the states of the RF engine correctly. The next step would be to implement the time information into it. Tackling the timing issues is crucial for a system that has to be in correct time in a margin of a few microseconds. With a fully implemented timely behavior, it could be seen, whether there are any issues requiring some modifications to the interface, for multiple concurrently functioning radio systems.

Time handling was originally meant to be implemented into the simulator during the project, as handling the multiple time concepts of each radio system and integrating these into the simulator was thought to be, though not trivial, but doable part of the SDR Devices implementation. Implementing the time into the model was first started, but later it was left for future work, as it became clear that this is a larger task than expexted[2].

A model simulating the SDR Devices services on a real hardware platform as a place to run the control code for the radio would be the ultimate proof of concept for the SDR model.

[1] In our implementation, Device module resembles SDR Devices

[2] Besides us needing to construct the SystemC model differently, proper modelling would have required link models, more exact protocol models, and knowledge about typical hardware timings, for examples.

Control inside the RF engine itself would be worth to study as well. There are interesting questions about e.g. power control and component selection to study, left to be done on the other side of the RF hardware interface. Power controlling schemas and power usage optimization relate to the durability of a battery. Decisions about component usage, in case of multiple radio components being able to perform the same transmissions or receptions, relates to performance the gadget offers for concurrent operations.

Bibliography

[1] Abidi, A. A. Direct-conversion radio transceiver for digital communications,. *IEEE J. Solid-State Circuits 30*, 12 (1995), 1399–1410.

[2] Adya, A., Bahl, P., Padhye, J., Wolman, A., and Zhou, L. A multi-radio unification protocol for ieee 802.11 wireless networks. In *in BroadNets* (2004), pp. 344–354.

[3] Ahtiainen, A., Berg, H., Lücking, U., Pärssinen, A., and Westmeijer, J. Architecting software radio. *SDR Forum Conference* (November 2007).

[4] Ahtiainen, A., van Berkel, K., van Kampen, D., Moreira, O., Pipponen, A., and Zetterman, T. Multiradio scheduling and resource sharing on a software defined radio computing platform. *Proceedings of the '08 Technical Conference and product Exposition* (November 2008).

[5] Auslander, D. M., Ridgely, J., and Ringgenberg, J. *Control Software for Mechanical Systems: Object-Oriented Design in a Real-Time World.* Prentice Hall, Jun 2002.

[6] Berlemann, L., and Mangold, S. *Cognitive Radio and Dynamic Spectrum Access.* Wiley, 2009.

[7] Black, D. C., and Donovan, J. *SystemC: From The Ground Up.* Kluwer Academic Publishers, Downloadable from http://www.embeded.cn/down_list/down_7218.htm, 2004. [Accessed 2009-12-09].

[8] Bluetooth Special Interest Group (SIG). Bluetooth specification version 3.0 + hs [vol 2]. http://bluetooth.com/Bluetooth/Tech nology/Build-

ing/Specifications/Default.htm, 2009. Part B, Baseband specification, pages 55–201 [Accessed 2009-12-17].

[9] Calhoun, G. *Digital Cellular Radio.* Artech House, 1988.

[10] Finnish Communications Regulatory Authority (FICORA). Frequency allocation table. http://www.ficora.fi/en/index/saadokset/m aarayksset.html. [Accessed 2009-12-17].

[11] Fromhertz, M. P., Bobrow, D. G., and Kleer, J. d. Model-based computing for design and control of reconfigurable systems. *AI magazine 24*, 4 (June 2003), 120–130.

[12] Golding, P. *Next Generation Wireless applications, 2nd edition.* Wiley, 2008.

[13] Gu, Q. *RF System Design of transceivers for wireless communications.* Springer, 2005.

[14] Hennessy, M., and Pattersson, D. *Computer architecture – a quantitative approach.* Morgan-Kaufman, 2005.

[15] IMEC realized a cost-efficient fully reconfigurable software-defined radio transceiver IC (press release). http://www2.imec.be/be_en/press/imec-news/archive-2007/imec-realized-cost-efficient-fully-reconfigurable-softw-defined-radio-transc-ic.html. [Accessed 2010-01-14].

[16] Ismail, M. A high speed wireless LAN radio receiver in software. In *Proceeding of the SDR 07 Technical Conference and Product Exposition* (2007).

[17] Karjaluoto, H., Karvonen, J., Kesti, M., KoivumÃki, T., Manninen, M., Pakola, J., Ristola, A., and Salo, J. Factors affecting consumer choice of mobile phones: Two studies from finland. *Journal of Euromarketing 14(3)* (2005). Digital Object Identifier: 10.1300/J037v14n03_04.

[18] Kester, W. *The Data Conversion Handbook.* Newnes, 2005. Available online, http://www.analog.com/library/analogDialogue/archives/39-06/data$_c onversion_h andbook.html.[Accessed 2010 - 04 - 28]$.

[19] Koopman, P. J. Embedded system design issues (the rest of the story). *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors* (1996), 310. ISBN:0-8186-7554-3.

[20] Law, A. M., and Kelton, W. D. *Simulation modeling & analysis.* Mc Graw-Hill, 1991.

[21] Leppänen, S. *Rigorous Service-oriented Development of Communicating Distributed Systems.* PhD thesis, Tampere University of Technology, 2008. Publication 718.

[22] Marwedel, P. *Embedded System Design.* Springer, 2006.

[23] Mitola, J. The software radio,. *IEEE National Telesystems Conference* (1992). Digital Object Identifier 10.1109/NTC.1992.267870.

[24] Mitola, J., and Maguire, G. Cognitive radio: Making software radios more personal. *IEEE Personal Communications Magazine 6*, 4 (1999), 13–18.

[25] Nokia N97 Data Sheet. http://events.nokia.com/nokiaworld08/assets/p df/-Data_Sheet_Nokia [Accessed 2009-12-09].

[26] Neuvo, Y. Cellular phones as embedded systems. *In IEEE International Solid-State Circuits Conference* (2004).

[27] Nicolescu, G., and Mosterman, P. J. *Model-Based Design for Embedded Systems.* CRC Press, 2010.

[28] Org, E. L., Cyr, R. J., Dawe, G., Kilpatrick, J., and Counihan, T. Software defined radio - different architectures for different applications. In *Proceeding of the SDR 07 Technical Conference and Product Exposition* (2007).

[29] Raiskila, K. Design and implementation of the control software for a radio frequency modem. Master's thesis, Helsinki University of technology, 2006.

[30] Razavi, B. *RF Microelectronics.* Prentice Hall, 1998.

[31] Scourias, J. Overview of the global system for mobile communications (gsm) [online document], October 1997. [Accessed 2009-12-17].

[32] Stallings, W. *Operating Systems, fourth edition.* Prentice Hall, 2001.

[33] Systemc homepage. http://www.systemC.org. [Accessed 2009-12-09].

[34] Tannenbaum, A. S. *Modern Operating Systems, second edition.* Prentice Hall, 2001.

[35] van Berkel, K., van Kampen, D., Moreira, O., Kourzanov, P., Splunter, M., Piipponen, A., Raiskila, K., Slotte, S., and Zetterman, T. Multiradio scheduling and resource sharing on a software defined radio computing platform. *SDR'09 Technical Conference and Product Exposition* (2009).

[36] Wang, J., Fang, Y., and Wu, D. O. A power-saving multi-radio multi-channel mac protocol for wireless local area networks. In *INFOCOM* (April 2006), IEEE, pp. 1–12. ISSN: 0743-166X, ISBN: 1-4244-0221-2, Digital Object Identifier: 10.1109/INFOCOM.2006.277, Current Version Published: 2007-04-10.

[37] Wolf, W. *FPGA-based System design.* Prentice Hall, 2004.

[38] Zhang, C., Anderson, C. R., and Athanas, P. M. All digital FPGA based FM radio receiver. In *Proceeding of the SDR 07 Technical Conference and Product Exposition* (2007).