Modular Software Radio

Linus

February 23, 2007

Contents

Part I

Introduction

Chapter 1

How to read this document

Before going into details about what is a software-radio, and what it can be used for, I will give an overview of this document, so that you know where to start first.

This document is separated into six parts:

- Introduction - this is what you're reading right now. It gives some basic definitions and ideas about the signal-processing part as well as the chosen implementation.
- Architecture - here you'll learn more about the design of the different aspects of the software-radio
- Reference Manual - when you need to know about a certain function how to use it or what it does, this is the place to go. Usually you'll need the knowledge from the *Technical Documents* to know everything.
- How-to - a more practical approach, this could also be called *tutorial*, as you learn how to use the software-radio step-by-step, without an explanation for the gory details.
- Tipsntricks - a collection of common pitfalls and how to avoid them, plus some help on how to do more unusual things.
- Future Thoughts - lots of things I wanted to do with the software-radio, both technically and experimentally, but that I didn't have any time left. Wanna go for it?

At the beginning of each part, you will find a short overview of the different chapters and what they talk about.

1.1 Overview of this part

- Motivation - why we want to have a software-radio
- System Overview - the basic building blocks of the software-radio
- Usage - what we are doing with it right now
- Outlook - possible future enhancements

Chapter 2

Motivation

This introduction describes our motivation for building a transceiver based on software-radio, hereafter called *software-radio testbed,* and gives an overview of the general philosophy.

2.1 Why Software-Radio?

> We talk about Software Radio when the map between the data (sending and receiving) and the data-carrying antenna signal are completely (within hardware limits) specified by the software. Any map that conforms with the hardware limitations (power, bandwith, hardware imperfections) may be implemented by means of an appropriate code. (B. Rimoldi, 2003)

If you like the idea of a flexible transceiver and are not too concerned with size and energy consumption, then you want your transceiver to be software-radio based. For instance, let us say that you have a software radio mobile phone. This mobile phone is a general purpose communication device with a piece of software that makes it behave like a mobile phone. You can turn your mobile phone into a GPS receiver, or a TV receiver, or a Wi-Fi interface, just by down-loading a piece of software (assuming there is a server that has the software you need).

For the technically oriented person: in a software-based transmitter, the software creates the samples corresponding to the signal to be transmitted. A general purpose hardware converts these samples into the signal that will be sent to the antenna. Similarly, in a software-based receiver, the general purpose hardware takes the signal captured by the antenna and produces the corresponding samples. The software does the rest. The hardware is not aware of the standard you are using: it just converts back and forth between samples and waveforms.

Fig.2.1 shows the two main components of a software-defined transceiver. The hardware implements a two-way mapping between waveforms and samples. Except for the possibility of controlling the power of the transmitted signal, the amplification of the received signal, as well as some other parameters that are not relevant for this discussion, this mapper performs the same operation regardless of the standard implemented by the transceiver.

img /var/www/html/ipgwww/data/media/dumb_hardware.ps

Figure 2.1: Dumb hardware and intelligent software

Chapter 3

System Overview

The software-radio helps to make it possible to implement a signal-processing algorithm which works on samples that are transmitted and received over the air.

Because the debugging is an important part of the implementation of a signal-processing algorithm, the software-radio can be run in either *simulation mode* or in *real-time mode*. Fig. 3.1 shows the software-radio in both modes.

The *Graphical User Interface* (GUI[1]) is the only visible part of the software-radio and allows the user to visualize the state of the different *Modules* as well as to change their configuration. The *Channel* is a general interface that represents either a *Simulation* or has access to the *Hardware*.

3.1 Simulation Mode

In simulation mode no hardware is used, and the whole transmission is simulated in software, including Gaussian noise and multi-path fading. There are no real-time constraints which makes it very easy to debug the algorithm to be implemented.

Of course, if you don't have access to the right hardware, this is the only possibility to use the software-radio. However, the *channel-server* which links multiple channels together, is written to simulate a real channel with high enough accuracy to test and verify signal-processing algorithms.

3.2 Real-Time Mode

In real-time mode only the Graphical User Interface runs on Linux, while the rest of the software-radio runs in Real-Time mode, made available through the use of RTLinux. This is necessary, as the transmission and reception of the samples has to meet time-constraints that are not possible to meet in simple Linux.

As of spring 2004, there exists two hardware-platform that allow the software-radio to do actual transception of samples over the air. The older one, produced by STMicroelectronics, offers a simple SISO-interface, that is, one antenna at each end of the transmission. The newer interface, produced by ICS-Ltd, allows the software-radio to take advantage of a MIMO-channel, with up to four antennas at each end of a transmission. A MIMO-channel is defined as a channel that has more than one transmitting antenna and more than one receiving antenna. These channels have very interesting properties, mainly the possibility to multiply the available channel-capacity by a function of the available antennas.

[1] Graphical User Interface

Figure 3.1: Structure of the software-radio in both modes

3.3 Communication

The software-radio is built to have a two-way communication. So, if you have more than one instance of a software-radio running, they can communicate together.

If the software-radio is run in real-time mode, only one instance of a software-radio can run on a computer. So if you want to communicate in real-time, you need at least two computers.

In simulation mode, the number of instances per computer is only limited by its calculation-power (and the patience of the user ;). A *Channel-Server* connects all channels of all instances of the software-radio together and makes it possible that everybody can listen to what the other radios are sending.

Chapter 4

Usage

At EPFL, the Federal Institute of Technology in Lausanne, Switzerland, we use the software-radio both in class and for research purposes.

This chapter gives an overview of what we did with the software-radio until the end of 2003, what we are doing now in winter/spring 2004, and what we are planning to do during the rest of this year.

4.1 Past

In class, it has been used to demonstrate the different parts of a radio-transmission, such as modulation, spreading, coding and matched filtering.

For research, we used it succesfully to demonstrate the usability of LDPC-codes over the air and to verify their theoretical performance.

4.2 Present

We are looking in the challenges arising from MIMO transmission that is coded with LDPC-codes. There are timing constraints to be solved, as well as theoretical challenges with regard to the MIMO channel to be met.

4.3 Future

Different projects for the software-radio are in preparation. These include a better matched filter (channel estimation), ZigBee implementation and the obligatory GPS-decoder.

Chapter 5

Outlook

For the time being, the software-radio is taking a direction towards point-to-point communications in MIMO-channels. We would very much like to study the implications of multi-point to multi-point communications, as well as low-tech implementations of a communication.

5.1 Multi-point to Multi-point

Point-to-point communications are quite well known. In fact, every commercially available transmission technology today only works in a point-to-point configuration, usually surrounded with a method to be sure that only one person is sending at the same time on the same frequency.

Different theories describe the possiblity of having more than one sender at the same time and being able to reconstruct the signal at the other end. It would be very interesting to study these theories in a real environment in order to give a feed-back about problems arising when implementing such theories.

5.2 Low-Tech Communication

The current hardware in use is capable taking advantage of several MHz[1] of spectrum to transmit and receive. HAM-radios only have a couple of kHz of spectrum available for the transcpetion. It would be interesting to study transmission using a sound-card and a HAM-radio, perhaps to propose a better and faster transmission than AX.25.

[1] Megahertz

Part II

Architecture

Chapter 6

Overview

In this part you'll learn about the architecture behind the software-radio. If you're looking for a precise information, you may be better off by looking at the *reference*-part of this document.

We like to split the software-radio in three parts: GUI[1], Signal Processing and the Antenna, as can be seen in figure 6.1. For each of these elements, you can find a chapter that describes it in more detail. Additionally to these aspects, there are more general ones that don't fit that nicely into the pictures. Here is an overview of the chapters in this part of the documentation:

- GUI[1] Is the Graphical User Interface, that allows to interact with the software-radio on a user-level
- Signal Processing is the ensemble of all interchangeable participants that make the active part of the software-radio
- Antenna represents the transmission and reception part of the software-radio, either in simulation or in real-time mode
- Operating system how the different parts of RTLinux and linux play together
- Modes of Operation gives an overview of the different modes of the software-radio
- Hardware which shows the architecture of the ICS-hardware
- Code the different directories in the tar-ball of the software-radio

[1] Graphical User Interface

Figure 6.1: The three main components and their respective subdivisions

Chapter 7

GUI

For taking measurements and changing the behaviour of the software-radio we developed this Graphical User Interface, called *Visualize*. It is capable of showing the internal states of all active modules (parts of the software-radio), their signals and changing the configuration of these modules while the software-radio is running.

By looking at figure 6.1, we distinguish three main-parts of the GUI[1]:

- User I/O is the input and output towards the user. It shows the chains, updates the statistics and offers windows to configure the modules
- Mapper arranges the modules in the software-radio to chains
- FifoCmd interfaces with the software-radio

7.1 User I/O

Different ways of interfacing the software-radio exist. The user can display:

- Chains which is an overview of the general state of the software-radio
- Stats representing internal values of the modules, single values or plots
- Output-ports that are the signals that pass from one module to the next
- Plots tracing stats of different modules against each other or in time

There exist two ways of actively interacting with the software-radio:

- Re-configuration by changing parameters of one or more modules
- Process Data which informs a module to immediately do something by processing it's input

7.1.1 Chains

When the *Visualize*-tool is started, it displays an overview of all active chains for the first antenna of the first radio it finds. In fig. @screenshot of visualize start-up@ you see the STFA in the middle, surrounded by a sending and a receiving chain. Each module in the chain has its name displayed, as well as up to two parameters of its internal state.

[1] Graphical User Interface

7.1.2 Stats

A stats can either be a single value or a *block* of values. Examples of single values include SNR, variance, BER or counters, while *blocks* of values can include channel-vectors, slots or a whole frame.

7.1.3 Output-ports

Each module has zero to many output-ports that can be displayed in a seperate window. As these signals can be of complex nature, one can chose to display only the real, only the imaginary, or the absolute part. Additionally, one can chose to display the FFT of the first $2^{\lfloor log_2 \rfloor}$ samples of the signal.

7.1.4 Plots

You may also be interested in a plot of one stats-variable in time or of two stats-variables one against the other. The *Visualize*-tool opens each new plot in a seperate window.

7.1.5 Export

All plotting-windows can be exported either as postscript-file or as a Matlab-function (which are also compatible with Octave).

7.1.6 Re-configuration

When asking for a re-configuration window, the *Visualize*-tool will update the software-radio with every new value you fill in. This is very useful for educational and debugging purposes.

7.1.7 Process-Data

This is mainly a debugging-function and allows to send a *Data-msg* to one of the modules, which will then process whatever is in its input.

7.2 Mapper

This is an important part of the *Visualize*-tool, as it's not a straightforward task to identify chains and display them correctly. It is basically the original version written by two 4th year students, but extended to work with more than one STFA, or other main-modules.

7.3 FifoCmd

This is the interface to the software-radio and takes care about the communication between the two. Everything is driven by the *Visualize*-tool, which means that the software-radio does only reply to requests from the GUI[1] and never initiates a request by itself.

Chapter 8

Signal Processing

Our goal was to find a suitable tradeoff between code re-use and performance of the software-radio. This is why we chose to have a modular framework, running on a RTLinux-platform. In the following sections, you will learn about the following items:

- DBG the interface to the GUI[1]
- Framework which makes it possible that modules can interact with each other and the outer world
- STFA the interface to the channel

8.1 DBG

The debug-interface allows the user to interact with the software-radio. It accepts commands and queries from the GUI[1][2] and replies accordingly. Looking at figure 8.1 you can see the Fifos used in both RF- and simulation-mode.

The replies of the dbg-module can be either status-informations or data directly from the software-radio. An overview of the different queries/replies can be found in 18.

8.2 Framework

The modules, which define the specific task of the software-radio, are surrounded by a framework that takes care of the book-keeping tasks necessary to guarantee a good working together of the different modules.

There are three parts:

- Modules and Chains the specific description of different signal-processing parts, like pulse-shape filter, coding, mapping, and others
- CDB Class Data Base, where a reference to every available module is stored
- SDB Subsystem Data Base, which contains a reference to every instantiated and thus active module, as well as the glue that makes it work

Because of the complexity of the subsystem alone, it has its own subsection where an overview of its capabilities is shown. The same goes for the STFA.

[1] Graphical User Interface

[2] the FifoCmd class encapsulates all queries and commands

Figure 8.1: The debug-interface in RF- and simulation-mode

img /var/www/html/ipgwww/data/media/modular.ps

Figure 8.2: Two simple chains and a module in detail

img /var/www/html/ipgwww/data/media/cdb_sdb.ps

Figure 8.3: The CDB and SDB

8.2.1 Modules and Chains

The communication system is built out of modules. Each of the modules has a classroom-style function. As an example, in a simple one-way communication system, the transmitter could consist of a first module that maps bits into signal space points and a second module that maps signal space points into samples. In the corresponding receiver you may find a module that implements the matched filter, and another module that decides what was transmitted.

Signal processing modules have inputs and outputs for the signals being processed. They also have a configuration-part to control the behavior of the module (e.g. the desired amplification for an amplifier) and statistics to display relevant information (e.g. internal variables). This is shown in fig. 8.2.

Each of the modules can exist in multiple copies. The framework (composed of CDB and SDB) makes sure that each copy can work independently of the other copies, not unlike the class/instance-behaviour of C++ or other object-oriented languages.

When one or more modules are linked together, we speak about a *chain* of modules. The software-radio knows how to pass the data from one module to another, and will call each module at the appropriate time, that is when it has some data to process.

8.2.2 CDB and SDB

The Class Data Base (CDB) and Subsystem Data Base (SDB) make it possible to use the modules in an object-oriented approach. While the CDB holds the static information about a module, such as the names and types of the configuration-parameters, the SDB holds an actual implementation of a module, with the specific configuration-values that may differ from one implementation to another.

An overview of the CDB and SDB can be found in fig. 8.3. It shows part of a running software-radio that has two transmitting and two receiving slots[3]. On the left side you see the CDB that holds a description of each module that has been loaded in the software-radio. On the right-hand side, you see the actual instantiations of some modules present in the CDB.

The loading of the modules happens usually at start-up, but theoreticaly it's also possible to load further modules once the software-radio has been started. While loading, a module informs the CDB about it's presence, and includes the input/output signal-types, it's name, configuration and statistics-names and -types. After that, the module is inscribed in the CDB, but not yet not active.

Once a module is needed in the software-radio, it is *instantiated*, that is, a running instance of the module is created. This includes reservation of memory needed for the different variable parts, as well as initialisation of these parts. After a module is instantiated, it can be connected to other modules and can perform signal processing.

8.2.3 Subsystem

Usually the subsystem is a part of the SDB. But over time it has become quite complex and would deserve an own directory in the Base/-directory [4]. One can think of the subsytem as the base-class for all modules. It offers a handful of virtual functions that allow it to interact with the modules.

[3] For clarity, only part of the chains are shown

[4] Future work hint…

img /var/www/html/ipgwww/data/media/architecture_subsystem.ps

Figure 8.4: Two modules and all possible connections

The main goal of the subsystem is to allow interaction between two modules. In figure 8.4 all the possible interactions between two modules are shown.

It is thus responsible for the following tasks:

- Allocating Memory for the Data, Config- and Stats-blocks
- Messaging between modules and do most tasks
- Threading the modules if necessary

Most changes in the state of a module include a message sent to the attached modules. An overview and short description of every message can be found in 16.3.

8.2.4 STFA

The Slot To Frame Allocator (STFA) is a module that makes the connection between the antenna and the rest of the modules. It offers a frame-based, slotted TDD interface to the rest of the signal-processing modules.

There exist two STFAs, one for the old hardware and one for the new, ICS-based hardware. The first is called stfa, while the second is called stfa_ics.

Chapter 9

Antenna

The antenna-part of the software-radio has the structure as shown in fig. @Antenna-blowup: Common/-Driver/HW/Server@ Their respective functions are as follows:

- Common is the interface towards the signal processing part
- Driver implements a certain hardware or simulation
- Hardware or Simulation the actual transmission system

9.1 Common

The interface of the antenna offers the following parameters to the signal-processing part:

- DMA-region a place where the received samples are written to and the samples to be sent are read from
- RF-parameters frequency, amplitude and other config-parameters
- Timing an function is available that tells about the actual timing of the RX/TX part

9.2 Driver

This is the implementation of a certain way to transmit and receive samples. It has to take care about all initialisations and correct handling of all exceptions. The following drivers are functional:

- RF interfaces the old, STMicroelectronics based RF-system
- ICS interfaces the new, ICS-based RF-system that is capable of MIMO-transmission
- Simul for simulations of the RF-system
- Simul_ics for simulations of the ICS-system
- Emul does a simple copy of the data to be transmitted to the next slot
- NOP does nothing, for debugging purposes

9.3 Hardware or Simulation

The final part of the software-radio defines the channel. The *Emul* driver, for example, implements a flat single-tap channel with no noise. The *Simul* drivers need a channel-server that takes multiple radios together, mixes their signal, and sends back the calculated signal.

The most interesting parts are the RF and ICS hardware, because they offer a real channel to test the transmission with.

9.3.1 Hardware

In the following table you can find a comparison of the two hardware-systems available.

	RF	ICS
Max. number of antennas	1	4
Frequency [GHz[1]]	1.9	2.4-2.48
Bandwith [MHz[2]]	3.8	1
Resolution	12	14

9.3.2 Simulation

In simulation-mode, a channel-server accepts connections from different radios, as can be seen in fig. @channel-server with two radios@. The channel-server can simulate multi-tap channels and add gaussian noise to the transmitted signals. This allows for easy simulation of real-world signals, before taking the modules on the air.

Chapter 10

Operating System

One very important aspect of a software-radio is it's real-time capability. In our implementation, we transmit and receive slots of a duration of about 1ms. If we want to make sure that there is no blank in the transmission, we need to make sure to do our calculations in this short time-span and to do it at the right moment.

In a modern operation system, lots of things are happening at the same time: graphics, sound, network, disk-access and more. A normal program will have to wait for these tasks to finish, before it can do it's work. This means that it's nearly impossible for a normal program to meet sub-ms precision. Different approaches exist to bring a solution to this problem. We chose RTLinux because of it's stability, availability and because it is licensed under the GPL[1] which means that other people can use this solution without having to pay high software-costs.

In short, RTLinux allows to meet time-constraints of a couple of μs, the precision-constraint that is given by todays hardware. It does this by running a real-time aware micro-kernel which is principally responsible for scheduling. One of the default tasks that runs with the lowest priority is the linux-kernel. This makes sure that even if the kernel is busy doing one of the not-so important things, RTLinux may put it to sleep, execute the real-time task, and resume the linux-kernel.

[1] GNU General Public License

Chapter 11

Modes of Operation

As the signal-processing part and the framework is very flexible, different operating-modes are possible:

- Test which includes just a simple chain that is run a limited number of times
- Simulation or Real-Time modes which are possible for both Local-Loop and the Two-Radio System
- Local-Loop where a radio 'talks' to itself by receiving every sample sent
- Two-Radio System a set of two radio, where each one talks to the other

In the following sections, the advantages of each of these are listed.

11.1 Test

As the name indicates, this is used to test the basic functionality of a module. Usually it includes a simple chain that has only the most basic components in it in order to test the module. Like this one can test the module in a simple environment, before going through the more real and more complete test in a real-time communication.

To test a mapper-module that maps bits into complex symbols, it would be enough to have the chain as depicted in fig. @show modules: source→mapper→block→slicer→sink@ For convenience, the *source* can contain readable text-messages that are printed by the *sink* and can be verified by hand.

The *block* module exists in different variants, where MIMO channels and multi-tap fading channels can be simulated, both in a convenient, deterministic manner.

11.2 Simulation or Real-Time

As already described in section 9.3, the software-radio can be run either locally without the RF-hardware and in a user-space mode, or it can be run in real-time using special RF-hardware for transmission and reception. While the former is much more easy to debug, only the latter allows to make real-world measurements and confirmation of theoretical results.

In the software-radio, both modes are transparent to the user, as the decision between the simulation or the real-time mode only has to be taken when running it. For the user, in either case, the channel is represented by the STFA.

11.3 Local-Loop

Sometimes it is too complicated to take care about all the synchronisation and fading-problems. Then you can chose to run your modules to test in a local-loop, and thus always be synchronised. When running in

img /var/www/html/ipgwww/data/media/simple_setup.ps

Figure 11.1: The most simple two-way communication example

simulation-mode, the channel-server makes sure that the sent samples are received at the same time. If you run the radio in real-time mode, then you have to make sure to connect the output of the cards with the input through a cable.

11.4 Two-Radio System

TODO: update for ICS-example

This section describes a basic system with two radios, following the example of the radio found in *Radios/Simple/BS* and *Radios/Simple/MS*[1]. It is important to know about this if you want to do more than just run the examples. You will learn about the most important modules, how to put them together and what make the thing going.

11.4.1 Setup

Looking at fig. 11.1, you can see two parts: a master and a client [2]. The communication channel in this example consists of three slots, of which only two are occupied [3]. The part in the middle, where the tree slots reside, is called STFA, which means Slot To Frame Allocation. This is the most basic module that you will find in mostly all of the software-radio. The input of the STFA are sent through the antenna, while the received signal from the antenna is passed through the output of the STFA. As you can see in figure 6.1, the antenna is a placeholder for either a real channel or just a simulation.

11.4.2 Modules

This is a short overview of the different modules:

sch_send Synchronisation CHannel. This module generates data that is used to communicate the configuration of the other clients. This includes the configuration about which slot to use for sending, which slot to use for receiving, and gain-control. Most of it is not used in a two-radio case.

modulator Takes bits as inputs and creates complex symbols. In its default-configuration, it outputs QPSK symbols.

spread Spreads the input-symbols with a given sequence. Usually this is used to have more than one radio sending during the same slot, in this example it is used to give some protection to the data, as a spreader may act as a simple coder.

chest_send Inserts a training-sequence into the signal. This sequence is known at the receiver-end, which uses it then to estimate the channel and to create the matched-filter.

synch_send Adds a synchronisation-sequence to the signal. The sequence is done in a special way so as to make it possible to retrieve the synchronisation-signal with as less calculation as possible.

rrc The Root Raised Cosine pulse-shape filter. Takes the complex signal, upsamples by a factor of two and generates a real output.

stfa Slot To Frame Allocation, takes the slots and prepares them to be sent over the channel.

rrc_rcv Applies again a Root Raised Cosine filter

[1] Microsoft

[2] Historically, the master is called BS (for BaseStation) and the client is called MS[1] (for MobileStation).

[3] There are three slots in order to allow for a more relaxed timing.

chest_rcv The counterpart to *chest_send*, calculates the channel-estimation and the matched-filter, and applies it to the input-signal.

demodulator Makes a hard decision on the received signals.

sink Prints to the screen the received sequence of bits.

synch_rcv Keeps up the synchronisation with the master.

despread Undos the operation introduced by the spreader, and offers a cleaner signal. As coding module this is suboptimal.

sch_rcv Decodes the synchronisation-channel and sets the tx-amplitude according to it's information.

11.4.3 Master

The masters task is to send out the synchronisation-signal on it's slot 1, combined with the data-signal that tells an eventual client its required tx-gain. The tx-gain of the client is calculated with the power received on slot 2. If it is below a certain threshold, the master considers that no client is sending, and puts the tx-power to 0. If the receiving-power is above a certain threshold, the tx-gain is adjusted to what the master would like to hear. This task in fact is done automatically by the sch_send module.

11.4.4 Client

While the master is quite static, the client has to do lots more:

1. Search for the synchronisation-signal
2. Set up the synch-slot and uplink-slot
3. Keep the synchronisation

The first point is necessary because the client doesn't know beforehand the time-frame of the master. So, in order to get it, the mobile sets up a *synch_rcv* module on each slot, and choses the one that has the highest probability of a successful synchronisation. After this, it updates the offset of the STFA, so that it is in synch with the master, and keeps one *synch_rcv* module active, to allow for further synchronisation. All this is done in a macro-module called *macro_synch*.

Once the primary synchronisation is achieved, it will set up some modules to decode and demodulate the synchronisation-channel, as well as set up a tx-channel on slot 2.

After this it has to keep up the synchronisation, because the master and the client don't have exactly synchronised clocks.

Chapter 12

Hardware

The current hardware is composed of three parts, as can be seen in fig. @figure of layout with ICS-rx and tx, as well as RF-cards@ This setup is optimized towards a 4×4 MIMO-system at 2.4GHz.

Chapter 13

Code

Once you untar the SRadio-*.tar.gz file, you find a directory in the form of *SRadio-1.0.0*[1] under which all the code is placed.

13.1 Directory Structure

Base CDB, SDB and the channel implementations are found here, as well as some general helping functions to the MSR.

Conventions template files that can be copied to new projects and then be filled in

Modules all user-written modules are found in here, put into different categories: Coding, Channel, Data, General, Macro, Signal

Test simple chains that are used to test the basic functionalities of the modules

Radios full-fledged two way transmission parts

User place for all compiled user-libraries

Kernel place for all compiled kernel-libraries

Documentation where you find this manual as well as some presentations

[1] the name has to start with SRadio, or else the Makefiles won't work!

Part III

Reference-Manuals

Chapter 14

Overview

In this part you find a complete as possible reference to all parts of the software-radio. If you're interested in one of these components, be sure to read about the corresponding subject in the architectural part.

- GUI[1] shows an overview of the classes used in the *Visualize* program
- Signal Processing gives more details about the CDB and SDB, as well as the subsystem and the modules
- Antenna what a driver has to implement in order to be used
- Hardware how to set up the hardware
- Modules a detailed description of the most often used modules

[1] Graphical User Interface

Chapter 15

GUI

The Graphical User Interface for the software-radio is called *Visualize*, as it visualizes the internal structure and state of the modules. Furthermore it's possible to change configuration-parameters of the modules in real-time, while it is running.

This chapter is split into three sections: one for the general interface, one for the interaction windows, and one for the more internal structures.

15.1 General Interface

The main classes involved in displaying the main view are shown in figure 15.1. Only the main subclassing from Qt is shown, subclassing from QObject and such is not shown.

There is one main-window, even if there are multiple radios to display. The main window, created by *Interface*, contains a *QTabWidget* with a tab for every active radio. If there is only one active radio, no tab will be shown.

Each active radio that is show is served by a *RadioView* module which is the active *QWidget* for the corresponding tab. The RadioView updates the display once a second, and stops updating if it is not the active tab. While updating it checks for new or removed modules and asks each module to update the values shown in it's body.

Each RadioView instance has it's own *ModuleGenerator*. On instantiation, the ModuleGenerator lists all available module and tries to determine which one is the main-module. In the most common situation, this will be a *stfa*, but it might be another module. The main-module is very important for the Mapper. For further updates, only added modules are taken into consideration.

All modules are drawn upon a *CanvasView*, which is subclassed from *QCanvasView*. CanvasView's main job is to create the context-menu when the user clicks the right mouse-button, as well as to move the canvas when the user drags it with the left mouse-button.

The *Visualize!Classes!Mapper* has a reference to all modules. He also figures out the position and connections of all modules, so that they fit nicely onto the CanvasView [1]. The main-module is called *stfa*. As we only show one channel at a time, the Mapper needs also to know which is the current channel.

Finally the *Module* represents the software-radio module with respect to all needed functionalities. It draws itself with the name and the chosen stats-parameters, including the pads for the connections; it can bring up windows for output- and stats-signals; it can ask the software-radio module to perform the signal-processing[2].

Between all these modules a number of signals are passed in order to minimize the cross-module method calls.

[1] Here is a possibility to rewrite a class

[2] This is only useful in debugging-mode

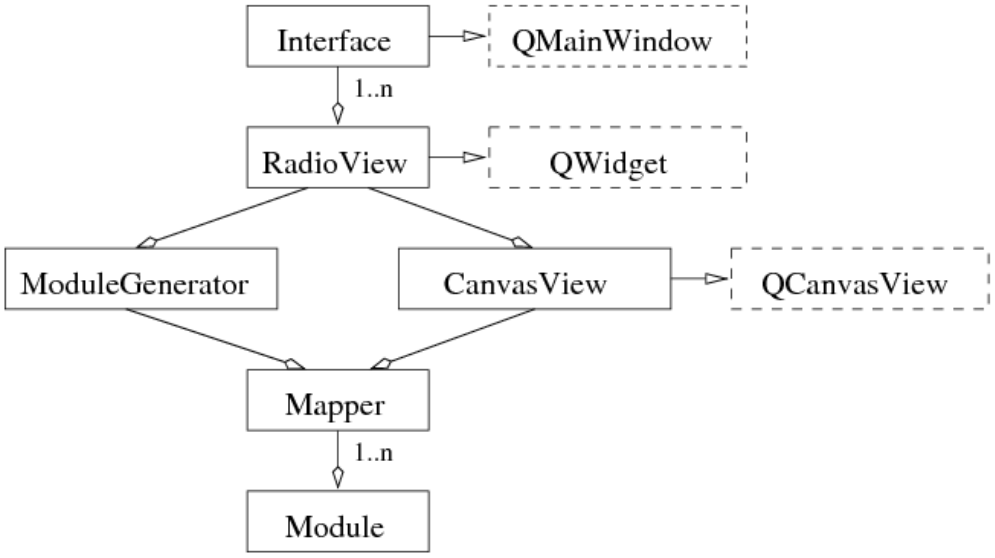


Figure 15.1: The classes involved in bringing up the main view

15.2 Interaction

There are a number of ways the user can interact with the GUI[3]:

- Chosing the stats to be displayed on the module
- Showing a stats-graphic
- Showing a graphic of an output-port
- Configuring values of a module
- Plotting stats-values

The active classes in doing this are shown in figure 15.2. The following list gives an overview of the used modules and their function, for a more detailed description, refer to the following subsections.

- Interface is the head of the visualize-tool and is the only one to have access to the menus
- Module represents a software-radio module
- Block is the virtual class for the (output)port, stats and plot
- Port knows how to read data from an output-port
- Stats reads a stats-'block' that represents some data
- Plot has a flexible data-part that can grow over time
- ConfWind shows a window with all configuration-options of the module
- Image is a special stats-'block' representing an image
- Show allows a Block to be displayed, complete with all control-widgets necessary
- PlotWin takes care of chosing the stats to be displayed

[3]Graphical User Interface

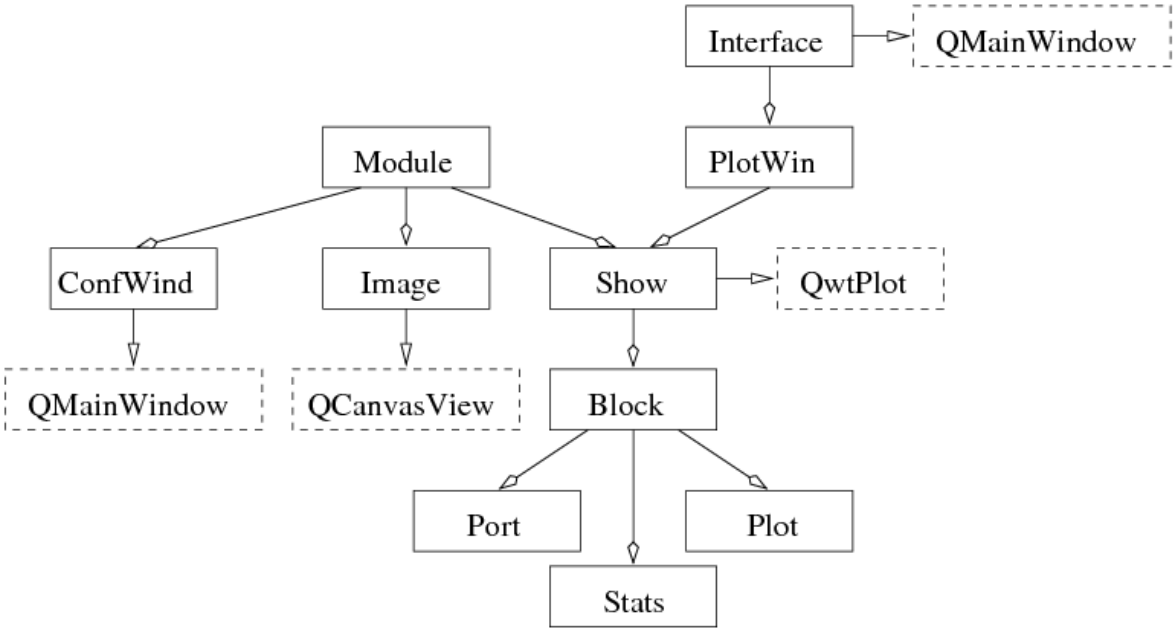


Figure 15.2: The different display-options

15.2.1 Plotting

There are two possibilities of plotting: Y(t) and XY. The former takes only one stats-argument and displays it in time, while the latter displays one stats-argument as the function of a second one.

Once the user has chosen one of the two plotting-methods, the Interface class instantiates a *PlotWin* and sets up the signals so that the PlotWin will be informed whenever the user clicks on a module.

It is the software-radio that takes care of reading the stats-values and putting them into a list. The PlotWin class reads this list once a second and updates its internal values with the values read from the software-radio.

In a clean implementation, PlotWin would be a subclass of Show, but as PlotWin needs an initialised window to work with, this is not possible.

15.2.2 Configuration

If the user requests a reconfiguration of a module, a *ConfWind* class is instantiated and given the authority to change configuration parameters. The ConfWind acts independantly on changes from the user and transmits them to the software-radio.

15.2.3 Signal and Outputs

Both require the Module to instantiate a *Show*, but give it either a Port or a Stats as argument. Show takes care of letting the user chose the method to display the signal (real, imaginary, complex, absolute, fft), zoom in and out, freezing and exporting to postscript and matlab.

15.2.4 Image

Althogh *Image* is a stats-variable, it is implemented as a class on its own[4]. When the module is asked to show a stats, it decides whether it has to put an Image in a QMainWindow or a Show.

The Image-class takes care itself about updating and preparing the data for display.

[4]One could subclass it from Block and tell Show how to treat an image

15.3 Internal

These classes deserve some more specific treatement.

15.3.1 Mapper

The mapper is described in a report of the students who wrote it. The report can be found in the software-radio tree under *SRadio/Documentation/Report/Visualize.ps.bz2*

Although the report is not up-to-date with most of the software-radio, the mapper has never been updated in the meantime. So everything described in the report with regard to the mapper is still accurate.

15.3.2 FifoCmd

This is the link with the software-radio. The counterpart is in SRadio/Base/DBG/*. All possible requests and changes to the configuration are described by this class.

15.3.3 Module

Together with the Mapper, this is one of the main classes. Its tasks consist of:

- Draw itself on the canvas with the name and the chosen stats-parameters, including the pads for the connections and eventual performance-measurements
- Show windows for output- and stats-signals, as well as configuration
- Process some data of the module. This is mostly useful in debugging-mode and for test-cases. It corresponds to a *call_module* in the software-radio.

Chapter 16

Signal Processing

16.1 CDB

There are two classes of functions in the CDB: defining a module and requesting informations about defined modules. While the latter is only used internally of the software-radio, the former is also used in the module-definition. Both type of functions are also described briefly in *Include/cdb.h*.

The Class Data Base has a reference to all announced modules in the system. Only a module that is written in here can be instantiated, get into the SDB and be connected to other modules. These are the functions used to add a new module to the CDB. Every function has also a counterpart-macro that is defined in *Include/spc.h*.

16.1.1 swr_spc_get_new_desc

First, it has to ask for a new descriptor, using the following function:

```
1 swr_spc_desc_t *swr_spc_get_new_desc(
2 int nbr_inputs,         // The maximum number of inputs
3 int nbr_outputs,        // The maximum number of outputs
4 int nbr_config_params, // How many configuration parameters
5 int nbr_stats_params); // How many statistic parameters
```

This function allocates the necessary memory to store the required parameters and makes sure there is a place in the internal database. All required in- and outputs need to be defined, as well as all config- and stats-parameters. This can be done using the functions that are described hereafter. Each of this function is also used in a macro defined in *Include/spc.h* for easier reference.

16.1.2 swr_spc_define_config_parameter

Every configuration-parameter has to be defined in the order of appearance in the *config_t*-structure. The types have to be the same. Instead of using the following function, you can resort to the *UM_CONFIG_**-functions. So, to define an INTeger, you can use *UM_CONFIG_INT("Name");*.

```
1 int swr_spc_define_config_parameter(
2   swr_spc_desc_t *cdb_desc,  // The description received
3   parameter_type_t type,     // The type of the variable
4   unsigned long flags,       // Eventual flags
5   const char *name);         // The name to use in the software-radio
```

If you have the following configuration-structure:

```
1 typedef struct{
2   int slots;
3   double amplitude;
4 } config_t;
```

then you need to define the two configuration-parameters with the following macros:

```
1 UM_CONFIG_INT( "slots" );
2 UM_CONFIG_DOUBLE( "amp" );
```

Note that the order and the type need to correspond, but not the name!

16.1.3 swr_spc_define_stats_parameter

Correspondingly to defining the config-parameters, all stats-parameters need to be defined, too. The same restrictions with regard to order and type apply. There are also macros that call this function. They are called *UM_STATS_**, where the types are the same as for the config-parameters.

```
int swr_spc_define_stats_parameter(
  swr_spc_desc_t *desc,  // The description-handler
  parameter_type_t type, // The type of the stats
  unsigned long flags,   // Eventual flags
  const char *name);     // Name visible in the software-radio
```

If you have the following stats-structure:

```
typedef {
  complex double channel;
  double SNR;
} stats_t;
```

Then you have to define them with the following macros:

```
UM_STATS_DOUBLE_COMPLEX( "H" );
UM_STATS_DOUBLE( "SNR" );
```

Note again how the order and the types need to correspond, but not the names!

16.1.4 Flags for define_*_parameter

There are two flags for the config- and stats-parameters that can be defined when defining these parameters:

- PARAMETER_DEBUG shows a parameter only when debug-mode is active in visualize
- PARAMETER_HIDE never shows a parameter in the visualize-tool

16.1.5 Types for define_*_parameter

These are the valid types for the stats- and the config-parameters:

	UM_CONFIG_	UM_STATS_
int	INT	INT
double	DOUBLE	DOUBLE
complex double	DOUBLE_COMPLEX	DOUBLE_COMPLEX
char[128]	STRING128	STRING128
block_t	-	BLOCK
image_t	-	IMAGE
void *	POINTER	POINTER
SYMBOL_COMPLEX	COMPLEX	COMPLEX

16.1.6 swr_spc_define_input

All inputs to the module need to be defined using this function. Again, a macro exists that englobes this function. Contrary to the config- and stats- definitions, there is only one macro, and you have to give it a paramater for the type. Even if you intend to use only a subset of the inputs at any given time, you have to define all of them.

```
int swr_spc_define_input(
  swr_spc_desc_t *desc,          // A pointer to the description
  swr_signal_type_t signal_type, // The signal type
  unsigned long flags);          // Eventual flags
```

All outputs to the module need to be defined using this function. Again, a macro exists that englobes this function. Contrary to the config- and stats- definitions, there is only one macro, and you have to give it a paramater for the type. Even if you intend to use only a subset of the outputs at any given time, you have to define all of them.

```
int swr_spc_define_output(
  swr_spc_desc_t *desc,
  swr_signal_type_t signal_type,
  unsigned long flags);
```

```
swr_spc_id_t swr_cdb_register_spc(
  swr_spc_desc_t **desc,
  const char *name);
```

16.1.7 Port Types

These are the valid signal-types for the in- and output and their respective type-name:

	UM_(IN/OUT)PUT(SIG_
U8	U8
SYMBOL_S16	SYMBOL_S16
int	S32
double	DOUBLE
complex double	DOUBLE_COMPLEX
SYMBOL_COMPLEX	SYMBOL_COMPLEX
SYMBOL_COMPLEX_S32	SYMBOL_COMPLEX_S32
SYMBOL_MMX	SYMBOL_MMX
SAMPLE_S12	SAMPLE_S12

16.1.8 Port Flags

The port-flags are described in 16.3.3.1 and can either be directly added when creating the ports, or in the _init-block of the module by a line like

```
  port_in(0).flags = SWR_PORT_OWN_MALLOC;
```

16.2 SDB

Once a module is defined in the CDB, it can be *instantiated*. When this is done, the SDB allocates private space for each instantiation and makes sure that each time an instance is called, it has the reference to its own private space.

Furthermore the SDB offers functions to have access to the config- and stats-structures from both the inside and the outside of a module. The difference is, that inside of the module you have an exact knowledge of the structure to change, while outside of the module you don't know anything else than the type and the name of the parameter to change or read.

Then the SDB offers some more special functions to access internal structures like the port in- and outputs and the profiling.

16.2.1 Instantiation

16.2.1.1 swr_chain_create

Usually you will instantiate a *chain* of modules, using the function *swr_chain_create*. It takes a number of arguments to modules that should form a chain. The following arguments are valid:

- NEW_SPC(”name”) Instantiates the module ”name” and connects its input 0 to the previous output (if any) and its output 0 to the next input (if any)
- NEW_SPC_IN(”name”, in) as NEW_SPC, but the input port ’in’ is connected to the output-port of the previous module
- NEW_SPC_OUT(”name”, out) as NEW_SPC, but the output port ’out’ is connected to the input-port of the next module
- NEW_SPC_IN_OUT(”name”, in, out) as NEW_SPC, but the input-port ’in’ is connected to the previous module, and the output-port ’out’ is connected to the next module
- NEW_SPC_VAR(”name”, var) same as NEW_SPC, but the id of the instantiated module is stored in ’var’
- NEW_SPC_VAR_IN(”name”, var, in) same as NEW_SPC_IN, but the id of the instantiated module is stored in ’var’
- NEW_SPC_VAR_OUT(”name”, var, out) same as NEW_SPC_OUT, but the id of the instantiated module is stored in ’var’
- NEW_SPC_VAR_IN_OUT(”name”, var, in, out) same as NEW_SPC_IN_OUT, but the id of the instantiated module is stored in ’var’
- OLD_SPC(var) same as NEW_SPC, but instead of instantiating a new module, takes an alread instantiated module ’var’
- OLD_SPC_IN(var) same as NEW_SPC_IN, but instead of instantiating a new module, takes an alread instantiated module ’var’
- OLD_SPC_OUT(var) same as NEW_SPC_OUT, but instead of instantiating a new module, takes an alread instantiated module ’var’
- OLD_SPC_IN_OUT(var) same as NEW_SPC_IN_OUT, but instead of instantiating a new module, takes an already instantiated module ’var’
- CHAIN_END indicates that the chain is finished.

The function returns an identifier to the created chain, so that the whole chain can be deleted when not in use anymore. In order to function correctly, at least two modules must be given as arguments.

16.2.1.2 swr_sdb_instantiate_name

If you only want to create one instance, for example for a module that will be connected to multiple other modules, or for a module that doesn’t have any in- or ouputs, you can use *swr_sdb_instantiate_name*. It takes as argument the name of the module and returns a sdb-id or -1 if an error occurs.

16.2.1.3 swr_connection_add

Once you created a chain or some single modules, you might want to create connections ’by hand’. Then you need the following function:

```
1 swr_conn swr_conn_add(
2   swr_sdb_id sender,   // The id of the sending module
3   int output,          // The output-port of the sending module
4   swr_sdb_id receiver, // The id of the receiving module
5   int input );         // The input-port of the receiving module
```

16.2.2 Manipulating stats- and config-structures

There are two possibilities: either a module wants to change its own structures, or a part of the software-radio outside of the module wants to read or write one of the structures. This can be another module, or the visualize-tool. Due to the different sources that may be using the config- and stats-structures, they have to be mutex'ed, so that the information read is always up-to-date and doesn't change in a critical way.

16.2.2.1 Accessing own Structures

For the modules own structures, a pair of functions exist to request the mutex to either the config- or the stats-structure. Once the mutex is aquired, the module can access it as it likes, before giving back the mutex. In order to force the user not to access the structures outside of a mutexed environment, a pointer to the structures is passed and is initialised with the address of the structure, or with NULL when the mutex is released. In this way the software-radio will immediatly show a bug when the structures are accessed outside of a mutexed environment.

In order to aquire the mutex, one of the two functions has to be called:

```
1 int swr_sdb_get_config_struct( swr_sdb_id id, void **str );
2 int swr_sdb_get_stats_struct( swr_sdb_id id, void **str );
```

As this is usually done in the module, the *context*-variable is available, so it is used like this:

```
1 config_t *config;
2 swr_sdb_get_config_struct( context->id, &config );
```

Now the module can read and write to the configuration-structure. The stats-structure is used in the same way.

To release the mutex, also two functions exist:

```
1 int swr_sdb_free_config_struct( swr_sdb_id id, void **str );
2 int swr_sdb_free_stats_struct( swr_sdb_id id, void **str );
```

As described above, they take a pointer to a pointer of the struct. In this way, using the pointer incorrectly causes a segmentation-fault (or a kernel-Oops in real-time).

16.2.2.2 Accessing other Structures

When accessing structures from other modules, one not only has to take care about mutual exclusion, but also one has to notify the module that something changed, at least in the case of a configuration-change.

To change the configuration of another module, one has to know the id of that module, the name of the configuration-variable and the type. Then one can set the value using one of the *swr_sdb_set_config_**-functions:

```
1 int swr_sdb_set_config_pointer( swr_sdb_id id, char *name, void *value );
2 int swr_sdb_set_config_int( swr_sdb_id id, char *name, int value );
3 int swr_sdb_set_config_complex( swr_sdb_id id, char *name, complex double value );
4 int swr_sdb_set_config_double( swr_sdb_id id, char *name, double value );
5 int swr_sdb_set_config_symbol( swr_sdb_id id, char *name, SYMBOL_COMPLEX value );
```

There is a special case where one wants to wait for the reconfiguration of the module, for example when changing a range of configurations from the same module. For this reason, the 'id'-parameter of the above functions can be chosen to be negative. Internally, all sdb-ids are positive, so a negative id always points to a unique module and stands for: "don't reconfigure right now". The other case is when reading stats-structures of other modules. The function-names are:

```
1 void* swr_sdb_get_stats_pointer( ... );
2 int swr_sdb_get_stats_int( ... );
3 complex double swr_sdb_get_stats_complex( ... );
4 double swr_sdb_get_stats_double( ... );
5 SYMBOL_COMPLEX swr_sdb_get_stats_symbol( ... );
6 block_t swr_sdb_get_stats_block( ... );
7 image_t swr_sdb_get_stats_image( ... );
```

All functions have \texttttswr_sdb_id id, char *name as parameter. Here a negative sdb-id as 'id'-parameter is invalid.

16.2.3 Other Functions

Most other functions from the SDB are only used internally and are documented in *Include/sdb.h.* The only exception is the function to read and display the profile of a module:

```
1 int swr_sdb_show_profile( swr_sdb_id id );
```

which displays all available profiles of the given module complete with number of calls, total time and average time.

16.3 Subsystem

As written in 8.2.3, the subsystem is the base-class for all modules. As such it is responsible for correct message-passing and cleaning up of the modules. Furthermore it keeps track and acts upon different flags that may be set in the subsytem and the ports. So there are three places that describe more or less entierly the state of the subsystem:

- Messages which are passed between subsystems[1]
- Subsystem-flags reflecting the state of the subsystem
- Port-flags showing the state of each port individually

In the following three subsections you'll find a description of each of these systems.

16.3.1 Messages

Each message that is passed to a subsystem has three arguments: message-id, data and return-id. The message-id tells the subsystem what it needs to do. The data-part is a (void*)-pointer, and should be set to NULL when it's not used. The return-id is used when a return-message could be generated, and should contain the sender-id. If the sender has no id (is not a module), the sender-id should be set to -1.

The messages defined in the message-id can be divided in three groups:

- Basic handling involves everything to set-up the module and is rarely or never called during life-time
- Reconfiguration of the module is also pretty rare for most of the modules
- Data Propagation is the workhorse of the subsystem and modules

Each group is described in more detail in the following sections.

16.3.1.1 Basic Handling

After the initialisation of a subsystem, everything is ready to connect this subsystem to another subsystem.

Connecting is done by sending the message *SUBS_MSG_CONNECT* to both subsystems that are to be connected together. As payload for the message one should give a structure of type swr_propagation_t. This structure contains all necessary information: port-#, size, flags, block-address, sdb-id of the other end and the direction. If one of the ports is already defined with regard to its size, it will communicate this to the connect-function, which will inform the other port of the desired size.

The *SUBS_MSG_DISCONNECT* message works in a similar way. One has to take care that both messages don't inform the other subsystem of the change. A function wanting to connect or disconnect two modules has to inform both of the action to take.

The user can ask for tracking of certain values. Whenever a subsystem is asked to track its values, it is sent a *SUBS_MSG_NEW_TRACK* message, after which the subsystem will check the tracking-list on each data-processing to update the corresponding tracks. Similarly, *SUBS_MSG_NO_TRACK* is sent to tell the

[1] Subsystem and Module are interchangeable in this context

subsytem to stop searching the tracking-list. This pair of messages exists because tracking is quite rare and asks for some processing-power in order to update all necessary lists. So, as long as the subsystem didn't receive a SUBS_MSG_NEW_TRACK-message, it won't search through the list.

Even though the software-radio is conceived as a real-time radio, some modules take more time because of their complexity. In order to assure that the rest of the software-radio is not affected by a complex module, it is possible to put the module in a thread by sending it a *SUBS_MSG_THREAD*[2]. When receiving this message, the subsystem sets up a thread and will activate this thread whenever it receives a SUBS_MSG_DATA message. For all other messages, the subsystem will run in the context of the calling function.

Finally, a subsystem will stop working upon receiving a *SUBS_MSG_EXIT*-message. All input- and output-ports have to be cleaned up before sending this messasge, otherwise undefined behaviour might occur.

16.3.1.2 Data Propagation

In a multi-threaded real-time environment one has to take care that things don't get mixed up. For this reason, before asking a module to do some calculations on data, one has to send it a *SUBS_MSG_PREPARE*-message. This message is propagated to all connected outputs where it is further propagated. If any of the connected modules is still working, the message returns a SUBS_STATUS_WORKING, and the caller should wait for a later time.

If the prepare-message returned 0 (for OK), that means that all modules in the chain are prepared and can be called by sending a *SUBS_MSG_DATA*-message to the top module. This message will test for SUBS_STATUS_MULTI_IN and SUBS_STATUS_THREAD and react accordingly. If appropriate, it will call the pdata-function of the module. Upon returning, the output-ports are checked for new data, and the modules connected to output-ports containing new data are sent a SUBS_MSG_DATA-message.

A small test-message that survived from the depths of the development is the *SUBS_MSG_PING*-message, which has no direct effect on the subsystem.

In order to allow for user-defined messages to the modules, the *SUBS_MSG_USER*-message exists. The payload of the message can contain whatever is accurate. Upon reception of this message, the user_msg-function of the module is called, with the payload as argument.

16.3.1.3 Reconfiguration

Whenever a part of the software-radio thinks that the configuration might have changed, it sends a *SUBS_MSG_RECONFIG*-message to the corresponding module. If the receiving module has the flag SUBS_STATUS_RECONFIG set, it will call the reconfig-function of the module. Furthermore the configure_inputs or configure_outputs-function is called, depending on whether the SUBS_STATUS_RESIZE_UP or -DOWN flag is set.

Upon arrival of a message, the subsystem stores all input- and output-port addresses, as well as the sizes. If something changes during the execution of the message, a *SUBS_MSG_RESIZE*-message is sent to all ports that changed size or the data-pointer.

16.3.2 Subsytem-Flags

These flags reflect the internal state of the susbsystem and are split in these groups:

- Propriety reflect a general state of this subsystem which is more or less static
- User-defined, that is, set in the _init-part of the module
- State for transient information about the module

[2] This is not the default setting, because threading of a module gives a sensible overhead

16.3.2.1 Propriety

All these flags are set internally by the software-radio and change very rarely.

- SUBS_STATUS_THREAD module has been threaded
- SUBS_STATUS_TRACKED there is a stats_track list with this module
- SUBS_STATUS_RESIZE_DOWN resize-messages go down
- SUBS_STATUS_RESIZE_UP resize-messages go up

The RESIZE-flags are set the first time a module receives a resize-message. This is done to know in the future which port-sizes have precedence, because in some situations it's not straightforward to decide what to do if there is not a clear preference for a certain resize-direction.

16.3.2.2 User-defined

All these flags can be set in the _init-part of the module by inserting a line

```
SET_STATUS( RESIZE_NONE );
```

One has to note that with the SET_STATUS-command the SUBS_STATUS_-part of the flag has to be omitted.

A module like the STFA only generates resize-requests, and will never receive one. The usual logic of the subsystem forbids this, but if you set the *SUBS_STATUS_RESIZE_NONE*-flag the subsystem will honor this behaviour.

While some modules don't want to receive resize-requests, other modules like the test_data_rcv need to be informed by changes on both the input and the output. If this is the case, you have to set the *SUBS_STATUS_RESIZE_BOTH*-flag. Afterwards the module will be alerted by any size-change on its input- and output-ports, and the subsystem won't complain about this strange behaviour.

The SUBS_MSG_PREPARE-message traverses all attached modules. Of course it has to stop at the STFA, else every module will be in *prepare*-status. If a module has the *SUBS_STATUS_PREPARE_SWALLOW*-flag set, then it will silently drop all requests to prepare and it will not inform other modules attached to itself.

If you have a module with multiple intputs, and you want to make sure that all connected inputs contain up-to-date data, you can set the *SUBS_STATUS_MULTI_IN*-flag. This will tell the subsystem to make sure that all inputs contain data before calling the pdata-method of the module.

An important issue when using the MULTI_IN-flag is the fact that the subsystem will try to make sure that all inputs are from the same time-instant. For this reason, the inputs of the module that has the MULTI_IN-flag set need to arrive in chronological order. Taking the example of a MIMO_LDPC-decoder, the first input has to come from the first STFA, the second input from the second STFA and so on. This is the only way that the subsytem can make sure that all inputs come from the same frame.

16.3.2.3 State

The states described here are very short-lived. They usually indicate a work in progress or a needed action.

- SUBS_STATUS_RECONF is set when the configuration-parameters have been changed, but before the module's reconfig-method has been called.
- SUBS_STATUS_WORKING indicates a module that is in it's pdata-method
- SUBS_STATUS_PREPARE is a module that is 'locked' and ready to process data.
- SUBS_STATUS_LISTED in conjunction with the debug-interface, indicates a module that is known to the visualize-tool

16.3.3 Port-Flags

These flags are individual for each input- and output-port. They can be combined together, although not all combinations make sense. There are mainly two groups of port-flags:

- Block-related which define how the block is allocated and who takes care about malloc/free
- Data-passing which describe when a block of data is ready or when it needs to change

16.3.3.1 Block-related

Besides the usual block- (port-)handling, some modules need a more special handling. These flags help define such special ports.

- SWR_PORT_OWN_MALLOC this means that the module wants to keep track on it's own about the different malloc/free
- SWR_PORT_OTHER_FREE another port is responsible for freeing this data
- SWR_PORT_OTHER_MALLOC another port is allocating the memory
- SWR_PORT_THIS_FREE this port is responsible for freeing the data
- SWR_PORT_PASSED_THROUGH this port passes the data through

16.3.3.2 Signal-passing

The flag *SUBS_PORT_DATA* is set whenever a module requests a buffer by using *buffer_out(port)*. When terminating the subsystem-call, it checks for this flag on all output-ports and makes sure that the appropriate attached modules are called.

The *SUBS_PORT_GOT_RESIZE*-flag is only used internally to mark a port that already has been resized. Without it, one could have a ping-pong of two ports that try to resize each other mutually.

16.4 Module

This section gives an overview of the module-creation and the use of it. Even though 25 gives an example of how to create a new module, it is a good idea to read at least this introduction, so that you know what it is about.

16.4.1 General introduction

Before a module can be used, it usually has to go through the following steps:

1. Registration with the CDB, usually in *module_init*, this happens when loading the module into memory
2. Instantiation, which means setting up the needed memory and calling *init*
3. A call to *reconfig* to assure that everything is OK

The points 2 and 3 are done automatically when calling *swr_sdb_instantiate_** and may happen more than once, where a new memory-block is allocated for each instantiation, in order to make sure that all copies of the module work in an independent way.

Once this has been done, a module can be asked to do one of the following tasks:

pdata	Process an incoming data-block and eventually produce some output-data
reconfig	Reconfigure itself because one of the configuration variables have been changed

resize Re-calculate its input- and output-sizes

custom-msg React to a user-message

finalize Clean up allocated values

The names to the left are the internal names used in the module-definition. You will never call these functions directly, but rather ask the MSR to do something that will then call one of these functions. So if you reconfigure one module using *swr_sdb_set_configure_int* you ask the MSR to set the configuration of this module-instance to a certain value and to call the appropriate *reconfig* function.

16.4.2 Data Structures

A module has three different data-structures:

config Where other modules may ask for a change in the behaviour

stats Results from the signal-processing

private Internal structure that is not available to the outside

While the first two have already been discussed a bit, the third is new. It may be used for internal tables built depending on the configuration, it may contain a copy of important config-parameters or anything else needed for a module to function correctly. An important point: the private-structure is personal to each copy of the module, so it is not suited to keep 'global' options.

The config and stats structures are protected by mutexes, as they are open to all other modules to use. So in order to use a config-structure, one has first to call

```
1 swr_sdb_get_config_struct( context->id, (void**)&config );
```

before being able to use *config-\textgreatersomething*. To free the structure, use

```
1 swr_sdb_free_config_struct( context->id, (void**)&config );
```

after which other modules can alos access this structure. The same goes for the stats-structure. You don't have to make this extra effort with the private-structure, as they are local to each instance anyway.

16.4.3 Data Types

16.4.3.1 For Config and Stats

Blocks

Blocks are a defined in the following way:

```
typedef struct {
  void *data;
  int size;
  swr_signal_type_t type;
} block_t;
```

They can be used to give a window into an internal vector. The matched-filter module for example has a block that points to the matched-filter used, so the user can see the matched-filter in real-time, using the visualisation tool.

The *data* pointer has to point to the vectory you want to display, *size* is the size in units of *type*, which is one of the Data-Types described in here (w/o *Block*, of course).

SYMBOL_COMPLEX

```
1 typedef struct {
2   short int real;
3   short int imag;
4 } SYMBOL_COMPLEX;
```

SYMBOL_COMPLEX_S32

```
typedef struct {
  int real;
  int imag;
} SYMBOL_COMPLEX;
```

DOUBLE_COMPLEX

```
typedef struct {
  double real;
  double imag;
} double_complex;
```

This structure is compatible with the *complex double* declaration from C. So, if you include ”complex.h”, you can declare a *complex double* and tell the subsystem to use it as such.

SYMBOL_MMX

Describes one complex symbol in a special format. It is done like this:

$Re_0 Im_0 - Im_0 Re_0$

The utility of this is that if we want to do a complex multiplication, we can arrange the second complex number in the following way:

$Re_0 Im_0 Re_0 Im_0$

And then a special MMX-operation on these two complex numbers yields directly the result, separated into real and imaginary part. This is very useful for convolutions that need to be optimised.

Simple Data-types

U8 Unsigned 8-bit

S8 Signed 8-bit

U32 Unsigned 32-bit

S32 Signed 32-bit

SAMPLE_S12 Signed 12-bit, where the 12 upper bits are used. For the available hardware, the lower 4 bits signal RX/TX

SYMBOL_S16 Signed 16-bit real symbol

DOUBLE a double floating-point value

16.4.4 Macros

Each function that is defined in a module takes at least one argument: *swr_sdb_t *context* In there all necessary information to distinguish one instance of another is stored. As this information may be a bit difficult to access, a lot of macros allow easy access to this information. These macros are defined in *spc.h* which is already included in the templates.

16.4.4.1 module_init

This function is a bit special in that it only registers the module with the CDB and doesn’t do any actual signal-processing. So these are the macros you can use:

UM_CONFIG_INT adds an int-parameter to the configuration

UM_CONFIG_COMPLEX adds a complex-parameter to the configuration

UM_CONFIG_DOUBLE adds a double-parameter to the configuration

UM_CONFIG_DOUBLE_COMPLEX adds a double_complex-parameter to the configuration

UM_CONFIG_STR128 adds a char[128] parameter to the configuration

UM_CONFIG_POINTER adds a void* parameter to the configuration

UM_STATS_INT adds an int-parameter to the statistics

UM_STATS_COMPLEX adds a complex parameter to the stats

UM_STATS_DOUBLE adds a double-parameter to the statistics

UM_STATS_DOUBLE_COMPLEX adds a double_complex-parameter to the statistics

UM_STATS_STR128 adds a char[128] parameter to the statistics

UM_STATS_POINTER adds a void* parameter to the statistics

UM_STATS_BLOCK adds a block_t parameter to the statistics, see <ref>par:Blocks</ref>

UM_STATS_IMAGE adds an image to the stats

UM_INPUT adds an input-port, for the types see <ref>sub:Data-types</ref>, and allows to define a flag

UM_OUTPUT adds an output-port, for the types see <ref>sub:Data-types</ref>, and allows to define a flag

16.4.4.2 other functions

private allows access to this modules private-structure

size_in(n) returns the input-size of the port *n*. This may also be used to assign a size to a port, so *size_in(0)=256;* is valid.

size_out(n) returns the output-size of the port *n*. Allocating sizes is possible as with *size_in*.

data_available(n) returns true if the input-port *n* has some new data

buffer_in(n) returns a pointer to the input-buffer *n* and clears the data-flag on this input-port

buffer_out(n) returns a pointer to the output-buffer *n* and sets the data-flag on this output-port

call_module sends a MSG_DATA to the module

make_thread puts a module in a thread

Chapter 17

Makefile

17.1 Make Arguments

Whenever you are in a sub-directory of the software-radio, you can give some arguments to the *make* command. There are arguments that may be used everywhere in the tree, some that are only valid in the *Radios/** subdirectories and some arguments that are only valid in the subdirectories that contain code.

17.1.1 Common

These arguments may be used anywhere in the tree (except the *Tools*-directory):

clean	Remove all object-files in all sub-trees
whole	Re-compile the whole tree
base	Re-compile base only
tools	Re-compile tools only
modules	Re-compile modules only, additionally *mod_coding*, *mod_data*, *mod_general*, *mod_macro*, *mod_signal* re-compile only this special modules-directory
show	Starts the visualisation-tool
server	Start the channel-server
kill	End all simulations as well as the channel-server
cleanproc	Remove all simulation-directories from */tmp*
rmall	Unloads all real-time modules and stops RTLinux
cvs_up	Updates the *whole* SRadio/*-tree using CVS[1]
cvs_commit	Commits *all* changes to the SRadio/*-tree

17.1.2 Radios

Arguments that can be used in the subdirectories of *Radios*/

bsms	Starts channel-server and both BS and MS[2] part. To stop, run *make kill*

[1] Concurrent Versions System
[2] Microsoft

show_bsms Like *bsms*, but also runs the visualisation-tool

wait_bsms Like *bsms*, but stops the MS^2 after 20 seconds and the BS after 30 seconds. Most useful to check whether a radio exits nicely, before trying it in real-time

short_wait_bsms Like *wait_bsms* but for the impatient: BS waits for 10 seconds, MS^2 for 5 seconds. Attention: things might not be correctly initialised after 5 seconds!

mc Starts channel-server and both Server and Client part. To stop, run *make kill*

show_mc Like *mc*, but also runs the visualisation-tool

wait_mc Like *mc*, but stops the Client after 20 seconds and the Server after 30 seconds. Most useful to check whether a radio exits nicely, before trying it in real-time

short_wait_mc Like *wait_mc* but for the impatient: Server waits for 10 seconds, Client for 5 seconds. Attention: things might not be correctly initialised after 5 seconds!

17.1.3 Code

Useful arguments when you are developing code

user Loads the modules defined in the Makefile for simulation and unloads them

user_show Like *user*, but also starts the visualisation-tool

user_wait Like *user*, but doesn't unload the modules

user_wait_5 Like *user*, but waits for 5 seconds before unloading

user_wait_10 waits for 10 seconds before unloading

user_wait_20 waits for 20 seconds before unloading

user_wait_30 waits for 30 seconds before unloading

user_wait_60 waits for 60 seconds before unloading

ddd Start the graphical debugger in simulation-mode

debug Start gdb in simulation-mode

rf Starts the radio in real-time mode

rf_tail Like *rf* but also tails */var/log/messages* where *PR_DBGs* will be written to

rf_show Like *rf_tail* but launches the visualisation-tool

rmall Unloads all modules from real-time mode

Chapter 18

DBG-interface

18.1 Command-syntax

18.1.1 list_modules

returns a list of all available modules

Arguments none

Returns a module_id,name list of all modules available, where module_id is to be used for reference, while name reflects the spc-name of the module. The returned list is sorted on module_id.

18.1.2 list_tag_modules

returns a list of all available modules and tags all modules as 'seen'. See also *list_new_modules*.

Arguments none

Returns a module_id,name list of all modules available, where module_id is to be used for reference, while name reflects the spc-name of the module. The returned list is sorted on module_id.

18.1.3 list_new_modules

only returns modules that are not tagged as 'seen'. Useful only in conjunction with *list_tag_modules*. All returned modules are tagged as 'seen', too.

Arguments none

Returns a module_id,name list of all modules available, where module_id is to be used for reference, while name reflects the spc-name of the module. The returned list is sorted on module_id.

18.1.4 show_all

gives the whole description of a module

Arguments the id of the module

Returns

input	number of inputs, followed by a type,len - list for every input
output	number of outputs, followed by a conn_id,conn_index,type,len list for every output, where conn_id and conn_index point to the module and port connected. If this port is not connected, conn_id and conn_index are both -1.
config	number of configs, followed by a name,type,value - list for every configuration-item.
stats	number of stats, followed by a name,type,value - list for every configuration-item.

18.1.5 show_*

Returns only part of the description. ”*” can be one of input, output, config, stats and will return the corresponding information.

Arguments the id of the module

Returns Like *show_all*, but only the asked argument

18.1.6 get_output

returns a given output of a given module

Arguments module_id,port_nbr

Returns size,type,values where values are decimal, comma-seperated values. For complex numbers, each value is a (real,imag)-pair.

18.1.7 get_block

Returns the values of a block. Contrary to ”show_stats” and ”show_config”, ”get_block” returns the values in their binary form.

Arguments module_id, stats_index

Returns The block of data in binary representation.

18.1.8 get_image

Returns an image that is stored in a stats. Read *Returns* for a description of the values returned.

Arguments module_id, stats_index

Returns The image in binary representation. The size of the returned block is of $width \cdot height \cdot \lceil \frac{bpp+7}{8} \rceil$. That means that a 20 x 20 black/white image ($bpp = 1$) will return 400 bytes.

18.1.9 set_config

Sends a new config-value

Arguments module_id, config_index, value value is in human-readable form.

Returns ”Reconfigured” on success

18.1.10 new_list

A list is used when one wants to track a certain value in the software-radio, or a value-pair. The software-radio tries its best to make sure that all value-pairs are correlated, but it may happen that an older value gets paired with a new value.

Arguments $module\_id_1$, $stats\_index_1$, $module\_id_2$, $stats\_index_2$ If $module\_id_2$ is -1 then only a single value will be tracked and the values returned by *read_list* will contain a value,time pair.

Returns The id of the list, in ascii

18.1.11 read_list

Returns the so far collected value-pairs. The cache is of length 1024, that means that you should collect the data before 1024 are stored. In the most busiest scenario, this means once every second.

Arguments list_id

Returns The first line contains the total number of value-pairs that will be sent. Then follow either ($value_1$, $value_2$) or (value, time) pairs, each one followed by a ”\n”.

18.1.12 close_list

Finishes tracking of the values from this list.

Arguments list_id

Returns OK or error on error.

18.1.13 process_data

Tells a module to immediatly start processing. If the module has inputs, all connected inputs will be activated before processing.

Arguments module_id

Returns OK on success

18.1.14 get_profiling

Returns all profiling-data from a module. The software-radio keeps track of the number of calls and the total execution-time of the following parameters: user-messages, data-processing and total time.

Arguments module_id

Returns Three lines of profiling with the time spent and the number of calls seperated by a space. The numbers are 64-bit integers. The time spent is returned in μ s.

18.1.15 ping

To test whether the software-radio is still running and replying to requests.

Arguments none

Returns ”pong”

Chapter 19

Signal Flow

For a correct understanding of what happens in the software-radio and where to insert a new module, it is very good to have an overview of the signal-flow that goes through the radio. As of the writing of this chapter, new hardware is being installed in our lab. For this reason, this chapter is seperated into three sections:

- Common: the common signal-flow
- ICS-hardware: the signal-flow specific to the ICS-cards
- STM-hardware: the signal-flow specific to the STM-cards

19.1 Common

In figure 19.1 you see an overview of the most common architecture when building a software-radio. On the left-hand side you see the transmitting modules while on the right-hand side the receiving modules are located. Each transmission is built around a *slot* which is a constant time-slice in the transmission.

19.1.1 Transmitting

The most common implementation starts with two blocks that have bits as output. The *Source* may be anything, from a pseudo-random sequence to a part of a network-transmission.

These bits go through a *Coding* block, where redundant information is added, in order to assure some error-resistance when receiving the data. The ouput of this operation are again bits. For the coding we have ldpc-codes, convolutional-codes or spreading-sequences which allow also for to seperate multiple users if they send at the same time-instant.

After the coding, the bits get *Mapped* into symbols. The most common mapping is a QPSK-mapping, where two bits define one symbol, as can be seen in figure 19.2. The mapper-module supports also other PSK-mappings or QAM-mappings. But commonly the QPSK-mapping is used.

In the middle of this block, a test-sequence is inserted which is called *Midamble*, because of it's position in the block, as can be seen in figure 19.3. The goal of this test-sequence is to be able to estimate the channel at the receiving end and to perform a matched filtering afterwards, cancelling out any effects due to the channel.

Once these four basic operations are done, the block composed of complex symbols goes through the hardware-specific part.

19.1.2 Receiving

Out of the hardware-specific part, we get again a block composed of complex symbols. If we have a flat fading channel with only one tap, that is a direct line-of sight, as well as a perfect synchronisation between

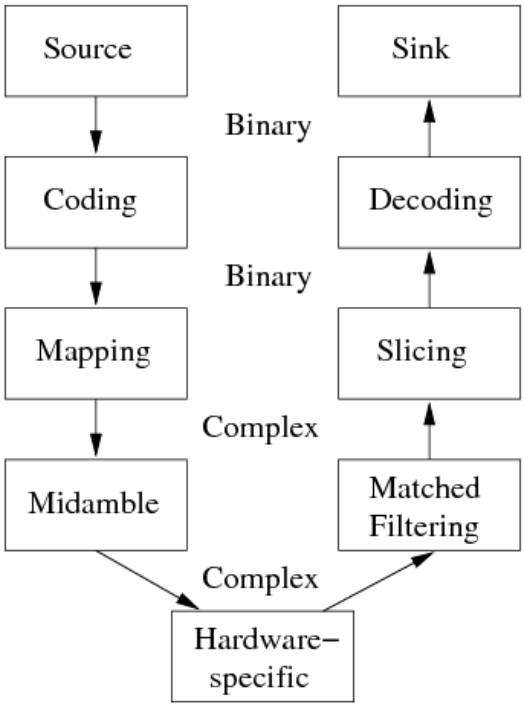


Figure 19.1: The common part of the signal-flow

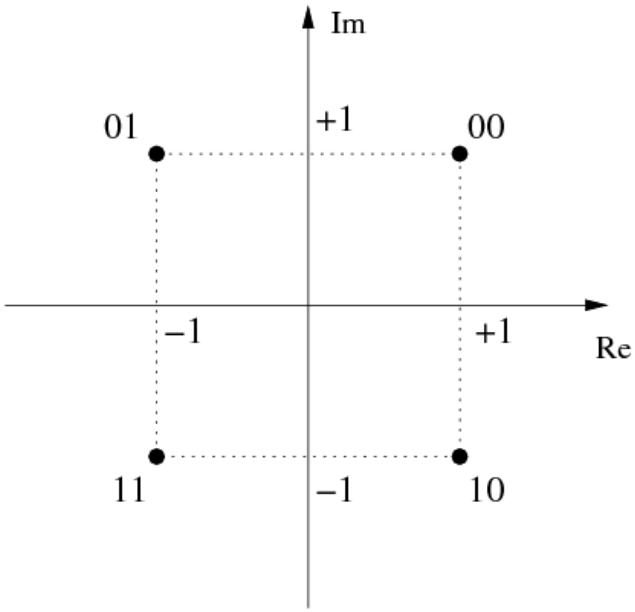


Figure 19.2: QPSK signal space

Figure 19.3: Position of the midamble

the sending and the receiving part (which is usually NOT the case), then this receivd symbols would be the same as the sent ones.

In common transmissions, this is not the case. For this reason we have the *Matched Filtering*, where the midamble from the sending chain is used to estimate the channel-parameters. Using these channel-parameters, one can improve the received quality of the signal. The output of this module is the filtered signal without the midamble. This module usually produces also information used by other modules, such as *SNR* or *amplitude* of the signal.

Now that the received signal is filtered, it is ready to be sliced. *Slicing* denotes the fact of taking a hard decision on the received symbols. For a QPSK-signal, the quadrant of the symbol gives directly the two bits. If the signal-alphabet is bigger, one has to calculate the distances between the received symbol and all possible emitted symbols, and then taking the smallest distance. A hard decision is usually the worst thing to do. For example the LDPC-decoder takes the complex symbols directly from the matched filter and achieves much better results.

After the slicing, we have again a block of bits, which run through the *Decoder*, where an algorithm tries to correct for transmission errors. As noted before, decoding on bits is not ideal, but it is what happens in most school-book examples...

The decoder ouputs again a block of bits that should contain no errors anymore. This block can now be either used for the network-transmission, reception of an image, or just to count the number of residual errors, in order to evaluate a code/decoder-pair.

19.2 Hardware

The hardwares job is to take the signal at it's sampling frequency, something around 1-10MHz, and to mix it so that it falls in the carrier-frequency, 1.9GHz, or 2.4-2.48GHz. In order to relieve the hardware of some very difficult filtering, it is important that the signal sent to the hardware does not occupy the whole sampling-frequency bandwith, but rather just a portion of it.

Furthermore it is important that, as the outgoing signal is filtered, as less as possible intersymbol interference is produced. For this reason, we apply a root-raised cosine filter, whose Fourier transform is the square root of the commonly used raised-cosine spectrum. If a root-raised cosine filter is used at both the transmitter and the receiver, the product of the transfer functions will be a raised cosine that will give rise to an output having a minimal inter-symbol interference at the receiver.

The ICS- and STM-hardware differ mostly in two aspects:

- Band while the ICS-hardware wants to have the signal in baseband, the STM-hardware needs the signal in passband
- Signal Unit the ICS-hardware works with complex samples, while the STM-hardware only works with real samples

The figures 19.4 and 19.5 depict the steps done to the signal from the fourier-transform point of view. The range of the fourier-transform has been chosen to be $-\frac{1}{2}..\frac{1}{2}$, but one could have $-\frac{f}{2}..\frac{f}{2}$ or $-\frac{\pi}{2}..\frac{\pi}{2}$ without changing the meaning at all.

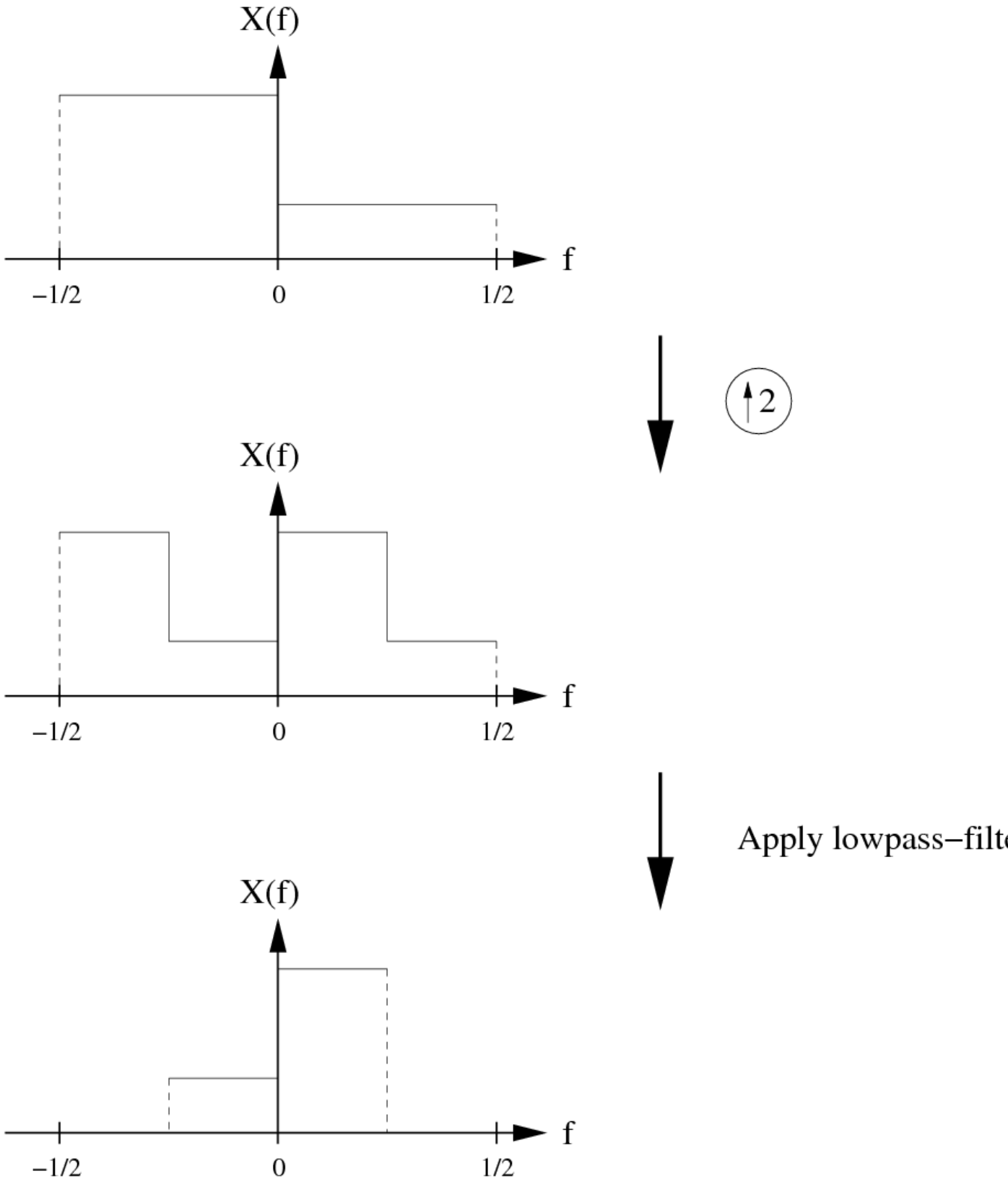
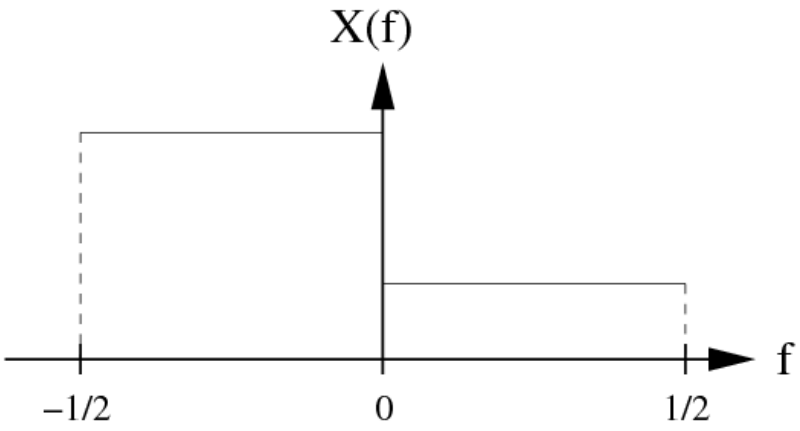
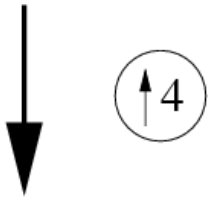
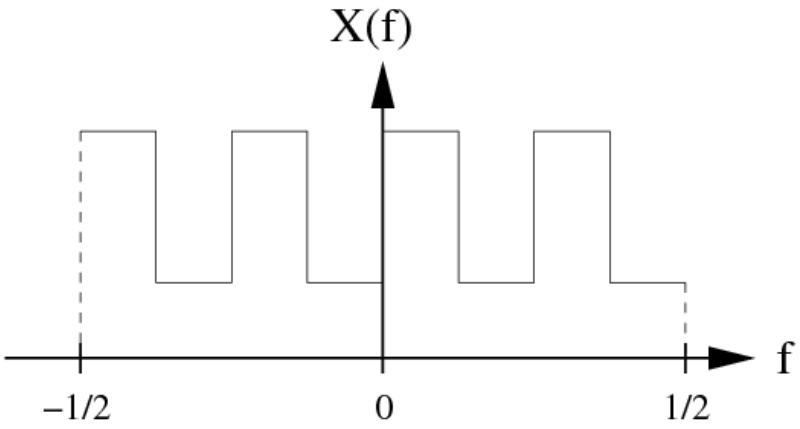
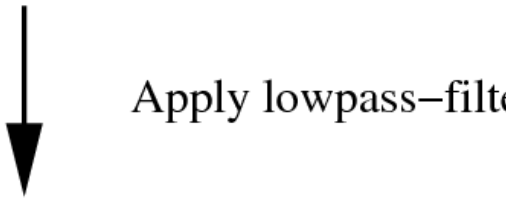


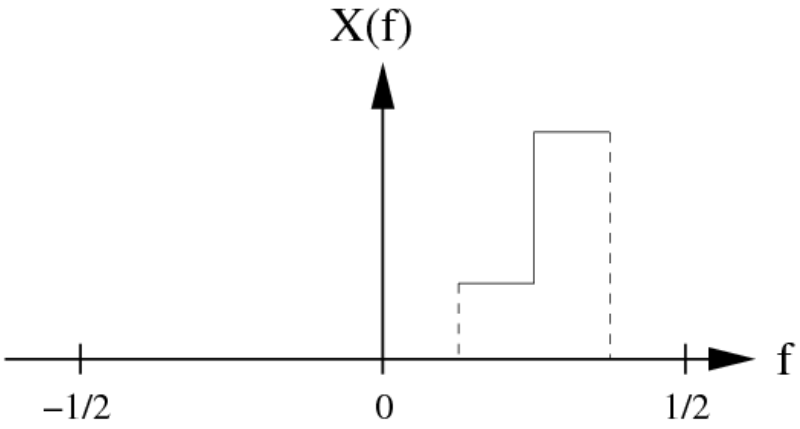
Figure 19.4: The signal preparation for ICS

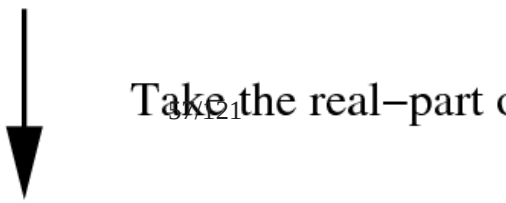

X(f)
f
−1/2
0
1/2

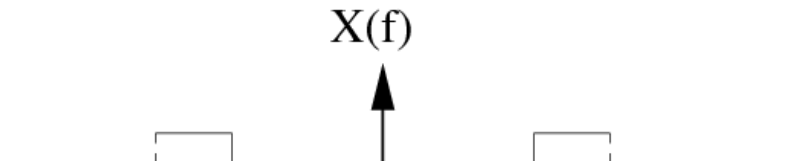

4


X(f)
f
−1/2
0
1/2


Apply lowpass−filte


X(f)
f
−1/2
0
1/2


Take the real−part o


X(f)

19.2.1 ICS-hardware

In figure 19.4, you can see the preperation necessary for sending the signal to the ICS-hardware. The signal is first upsampled by a factor of two. This is done by inserting a zero in between two complex symbols.

Next we apply the lowpass RRC-filter and now we have a signal that occupies half the bandwith, but at double the symbol rate. Thus, we didn't loose information.

On the receiving side, exactly the contrary needs to be done: first we apply the RRC-filter, then we downsample by a factor of two. The filtering is necessary, as there might be some other signal next to ours. And, because the reception sample-rate is twice the symbol-rate, we have to downsample by a factor of two.

19.2.2 Philips-hardware

The latest hardware that is not yet installed has been developed by Philips. It contains the A/D- and D/A-converters directly on the same board as the RF-part. The connection to the PC is made via two SCSI[1]-cables, and the PCI-card only contains simple glue logic to put the data on the PCI-bus or to read it from there.

This hardware works at 1.9GHz and has a bandwith of 5MHz.

19.2.3 STM-hardware

Looking at figure 19.5, you can see that for the STM-hardware, we need to upsample by a factor of four. The filter chosen for the pulse-shaping is a passband-filter, that has a bandwith of 1/4.

Because the hardware accepts only real samples, we can't keep the signal at baseband, as a loss of the signal would happen. You can see that by taking the real-part of the signal, it gets mirrored at the axis. If you do this with a baseband signal, information loss occurs.

These three operations are done using advanced MMX-operations that manage to take advantage of the special structures of the filters and the signal.

[1] Small Computer System Interface

Chapter 20

Important Modules

In this chapter you'll learn about some of the most important modules in the software-radio. They represent the basic functionality and should be known by everyone that wants to handle the software-radio.

20.1 STFA

STFA stands for Slot To Frame Allocator. Its main purpose is to map the RX- and TX-slots to the correct moment in time. First of all you need to understand the principles of a slotted TDD transmission using frames. In figure 20.1 you can see a transmission of 3 frames. Each frame takes the same amount of time. Normally the frame-structure stays the same during the transmission. Of course this wouldn't be the case in a multi-user environment, where a frame contains the structure of the up- and downlink slots, which would change during connection and disconnection of users.

The STFA represents this frame-structure. It is a module with inputs and outputs, where is the number of slots per frame. The purpose of the STFA is to calculate the slots so that they are sent at the correct moment in time. To understand figure 20.2 correctly, it helps to imagine yourself the STFA as a representation of the channel. So the inputs of the STFA are the inputs to the channel, which corresponds to the TX-part. The outputs of the STFA are what comes out of the channel, which is the RX-part.

Internally, the STFA has two large sample-buffers that hold one complete frame in memory. One buffer is read continously [1] from by the D/A converter, while the other buffer is written to continously [2] by the A/D converter. For this reason, it is important that the result of one slot is computed before the D/A converter needs it. It is also important that a RX-slot is processed before the next frame.

[1] the transmission is done in blocks using DMA
[2] the transmission is done in blocks using DMA

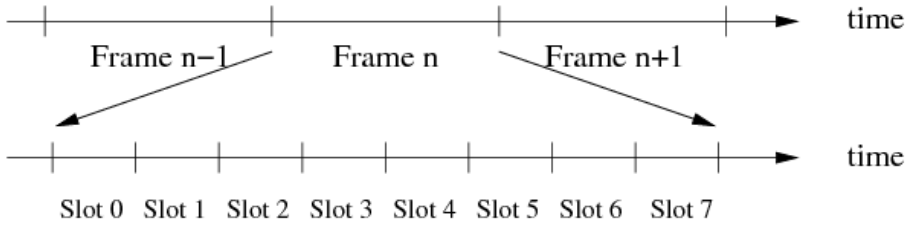


Figure 20.1: The frames and slots

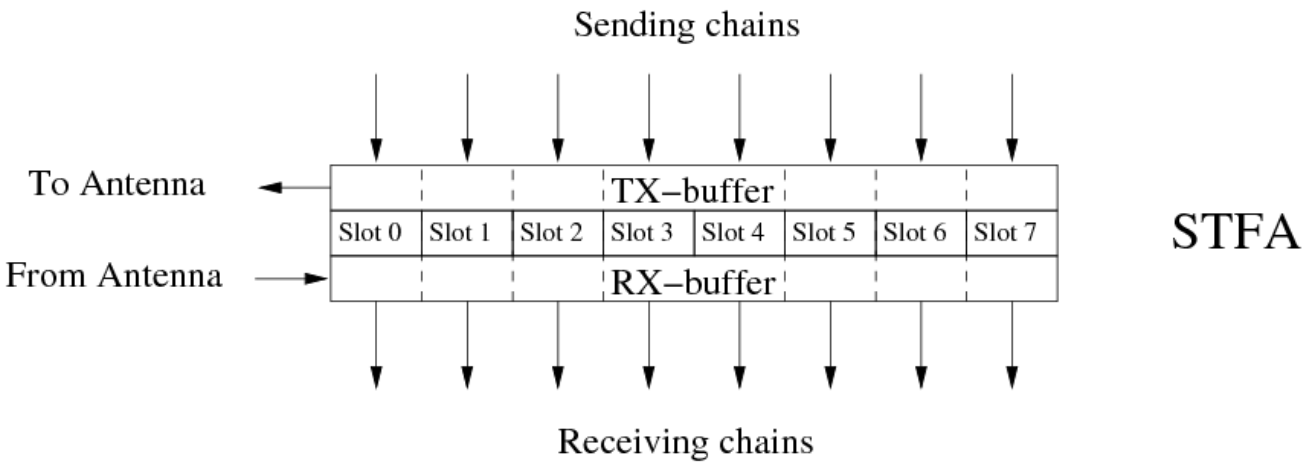


Figure 20.2: Inputs and outputs of the STFA

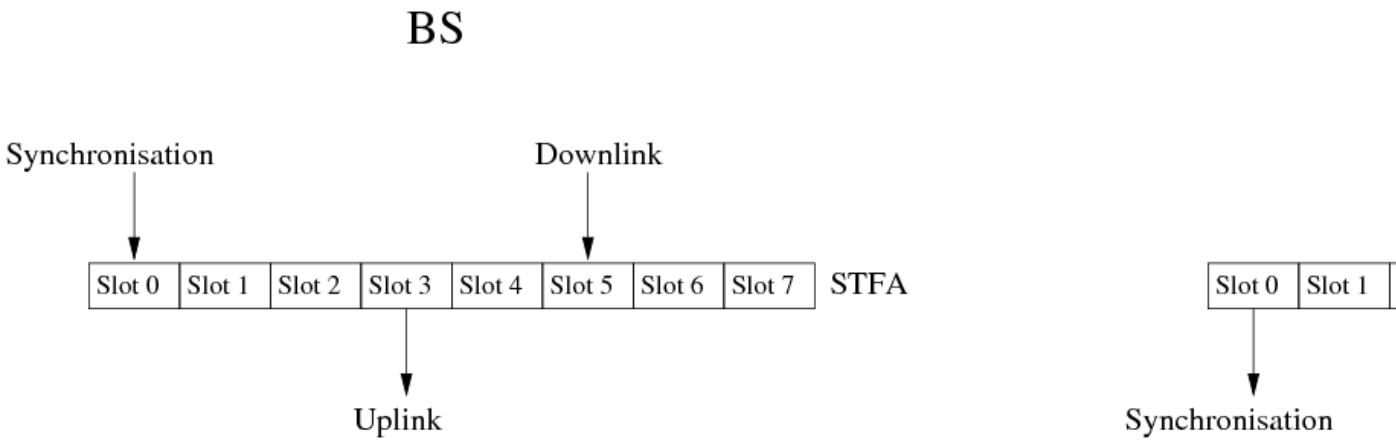


Figure 20.3: A typical set-up of the STFA

20.1.1 Synchronisation

A common problem in a slotted TDD-environment is the synchronisation of two stations. If we consider two participants in a communication, called BS and MS[3] (for Base Station and Mobile Station), one has to define at what instant in time the first frame starts. If we call this time-instant $t_{frame}(0)$, then all consecutive frames will start at $t_{frame}(n) = t_{start} + nd_{frame}$ where is a positive integer and d_{frame} is the time-duration of one frame. This also defines all slots, as $t_{slot}(m) = t_{frame}(n) + md_{slot}$ with d_{slot} the time-duration of one slot and $0 < m < slots$.

We can't know beforehand $t_{frame}(0)$, and also d_{frame} is only known up to a certain δt , because of clock-drifts between the two stations. One solution is to send a synchronisation-signal in slot 0, so that one can know $t_{slot}(0)$ for every frame. Then we can find $t_{slot}(m)$ for all the other slots of the frame, supposing that the error in d_{slot} is negligible when calculating the time-instants of one slot.

This setup is shown in figure 20.3. In a real system, the MS[3] starts out with synchronisation modules attached to all its STFA-outputs. The reason for this is that upon startup, we don't have any information about $t_{slot}(0)$, so we have to expect the synchronisation-signal on any slot. Once the synchronisation-signal is found on a given slot, the buffers are adjusted so that the synchronisation-signal falls into $slot_0$

[3]Microsoft

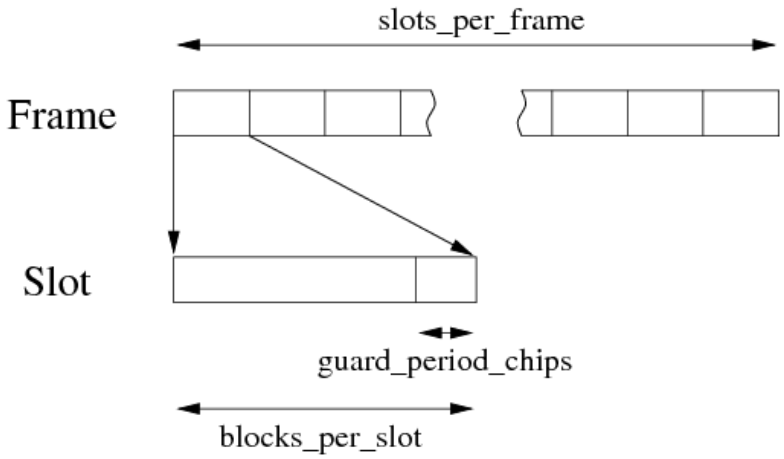


Figure 20.4: The different size-parameters

. Then, each time $slot_0$ is received, the buffer is adjusted again, so that all $t_{slot}(m)$ are accurate and in synchronisation with the BS again.

20.1.2 Important Parameters

There are three groups of parameters in the STFA:

- Structural which define the size of the different parts of the STFA
- Timing everything that got to do with preparation of slots and synchronization
- RF the parameters of the RF-part are also reflected in the STFA

20.1.2.1 Structural

These are quite important, as they define the basic structure and size of the STFA. You can't change these parameters once the STFA has been started, as the DMA-transfer would be disturbed greatly by this. An overview of the parameters is given in figure 20.4.

The *blocks_per_slot* parameter has a unit of 128 symbols. By taking the default value of 20, this gives a slot-length of 2560 symbols. Substracting the guard-period, we get a useable slot-length of 2470 symbols.

To make things even more complicate, the total number of *blocks* has to be a multiple of 16. This is due to the fact that the DMA-transfer is done in blocks with a size of 16 * 128 symbols [4]. Taking a smaller block-size for the DMA-transfer would result in more interrupts and thus a higher system-load.

As we have a TDD-system, TX and RX slots are stacked up in time. Without any special handling, the RF-system would have to be able to switch off the transmit-chain from one symbol to another. Because this is very difficult to do, a *guard-period* has been inserted. During this time, the state of the RF-cards is not defined, and no useful data is transmitted.

20.1.2.2 Timing

Usually you don't have to change these parameters. They are taken into account by the synchronisation-macro module.

[4] This is defined in Base\/Antenna\/ICS/ics_dev.h

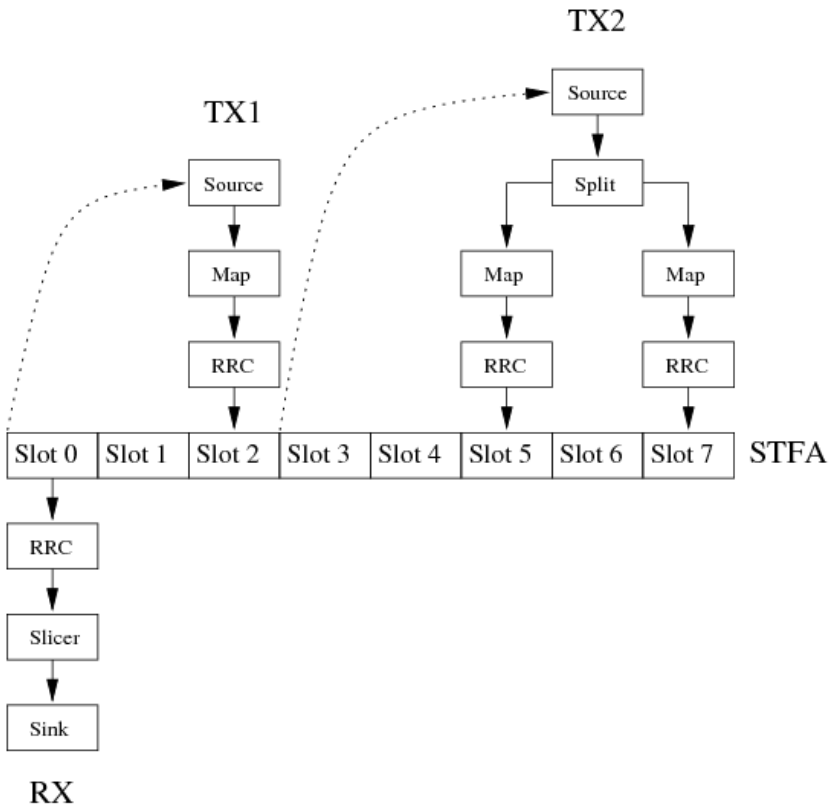


Figure 20.5: Two transmit and one receive-chain as an example

20.1.3 Attaching Chains

The STFA on its own doesn't do anything. The mode of operation and the kind of mapping done over the air is defined by attaching chains to it. If a chain is attached to an input of the STFA, we talk about a transmit-chain, or TX-chain. A receive-chain or RX-chain is attached to the output of the STFA.

In figure 20.5 you see a picture of some example-chain. In the middle is the STFA that is responsible to alert all modules as soon as there is some data to proceed. Looking at the RX-chain, you see that it is very simple for the STFA to know when to alert the RRC-module of the chain. The right moment is when this slot has been received.

For the TX1-chain, the right moment to alert the source-module is $t_{alert} = t_{slot}(0) - d_{calc_chain_{tx1}}$. Unfortunatly the $d_{calc_chain_{tx1}}$ is not known in advance and might change during the run of the software-radio. One way to tackle this problem is to put an upper time-limit on the calculation-duration, and call the top-module that much in advance. In the actual STFA-implementation, $d_{calc_chain_{tx_max}} = 2 * d_{slot}$, so that the first module of the TX1-chain is called at the beginning of slot 0.

Another problem arises when we have constructs as in the TX2-chain [5]. Although it is not optimal (See 34.2), we applied the same reasoning as with the TX1-chain, that is, we call the top module of the TX2-chain two slots in advance. The disadvantage is, that the whole chain has to finish in the time $2 * d_{slot}$.

20.1.3.1 Overcoming the Time-Limits

In an ideal setup, you'll be able to calculate every slot in a frame fast enough, so that it can be sent on time over the air. Unfortunatly, this is not always the case. Sometimes you want to trade in some of the

[5] The two branches may also fall into two different STFAs

real-time with the possibility to do some more calculations than the time allows. The problem is different on the sending- and on the receiving side.

The only time you'd want to send a slot that takes a very long time to calculate is when you want to do repeated measurements on the slot. So what you can do is to calculate the slot once, and then send it again and again. You can do this by setting the notice-point not to the top-module, but rather to the RRC-module only. Like this only the pulse-shape will be done, taking care of an eventual desynchronisation between the two radios [6].

On the receiver side, the only reason for taking more than the allocated time is about the same. But you may easily take up more time than $2 * d_{slot}$, if the total amount of all receiving chains is not bigger than one frame. If this might be the case, then you can use the following command:

```
make_thread( rrc_id );
```

This means that every time the STFA has data that needs to be processed, it checks whether the chain is still working, and only notifies the top-module, if no work is done. If the chain is still working, the STFA doesn't send the request, and waits for the next frame to check back again.

Here at EPFL we use these two techniques to measure code-qualities, especially LDPC-coding schemes. It is important to us that we get an actual transmission, but analyzing 10 slots per second is good enough for our case.

[6] Remember that the MS[3] moves in time

Chapter 21

Subsystems

The software-radio has a reception and a transmission part. Each one of these has its specialities and quirks. Here I list the names used in programming, the restrictions and how they come across. Only the reception-part is treated in this document, as the transmission-part is quite simple.

21.1 Nyquist

It is very important to understand the implication of Nyquists theorem for this reception-chain. Nyquist wrote that the sampling-frequency must be twice the bandwith of the signal to sample. He also described the aliasing that happens if you sample a signal that is out of this bound. For the reception-part of the software-radio, we rely on this aliasing to capture a signal at an intermediate frequency of 70MHz using a sampling-frequency of 100MHz. In a most general way, given the parameters:

fi_rx the intermediate frequency

wi_rx the intermediate used bandwith

f_adc the sampling-frequency of the analog-to-digital converter

then f_adc must be chosen so that

$$\frac{f_adc}{2}N \notin \left[fi_rx - \frac{wi_rx}{2}, fi_rx + \frac{wi_rx}{2} \right]$$

is fulfilled. Else we get an overlapping of the aliased signal and we loose information.

For the software-radio, $fi_rx = 70MHz$, $wi_rx = 20MHz$, $f_adc = 100MHz$, which fullfills the above equation.

21.2 Reception-chain

The different parameters of fig.21.1 and fig.21.2, their boundaries, and a short description:

f_rf [2300..2500]MHz[1], the transmission-frequency

attn_rx [0-41]dB, the attenuation of the Rx-chain (0 gives stronges output, 41 gives weakest)

f_adc [1-100]MHz[1], the sampling frequency of the ADC.

fi_rx [1-500]MHz[1], the intermediate frequency. Every frequency above $\frac{f_adc}{2}$ will be attenuated due to the sinc of the ADC.

[1] Megahertz

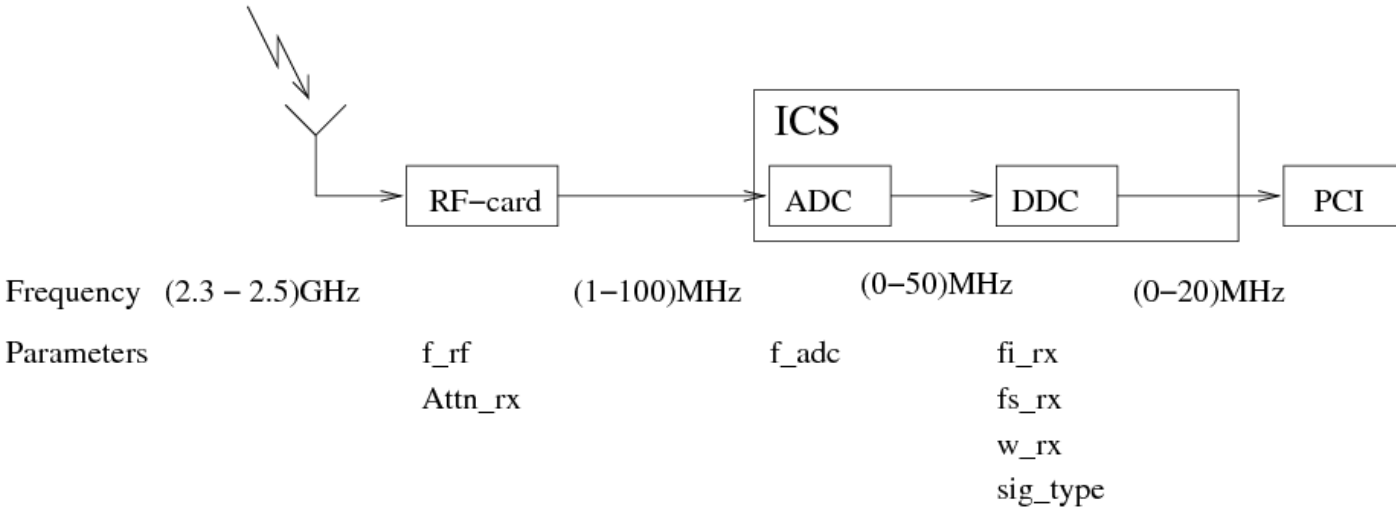


Figure 21.1: Reception-chain

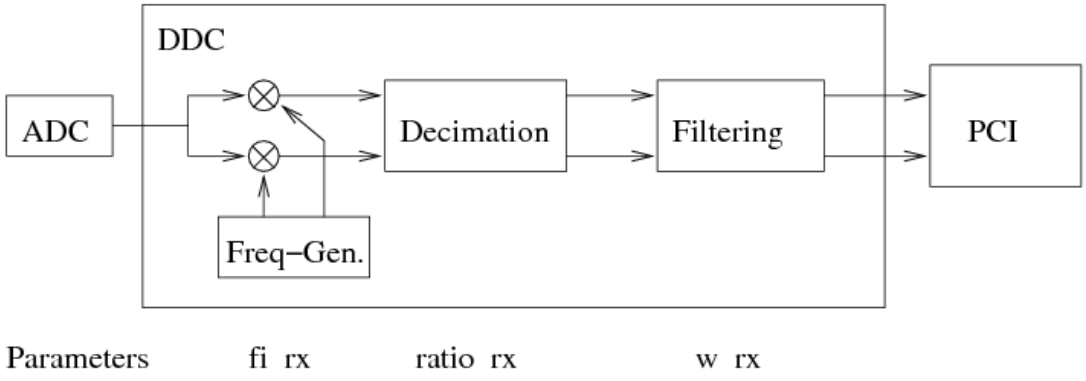


Figure 21.2: The whole chain and the most important part of it

fs_rx [0-$\frac{f_adc}{8}$]MHz[1], the final sampling frequency (the DDC does down-sampling). Contrary to f_adc, fs_rx is measured in complex samples.

w_rx [0.08-0.75], the portion of the final sampling frequency fs_rx which is not cut off by the DDC filters.

sig_type SIG_COMPLEX_ICS, SIG_S16 for FPGA_DEFAULT
SIG_COMPLEX_S16 for FPGA_S16

21.3 More detail

The final sampling frequency is limited by the use of the DDCs in the card. A simplified structure of the DDCs can be seen in fig.21.2. Due to internal limitations, the smallest useable decimation factor of the DDCs is 8, and the highest is 4096. For more detail, see the Graychip-documentation on page 11, 3.4.2 and references.

21.3.1 w_rx

This parameter is also dependant on the DDC-chips. In fig.21.2 this is simplified by a *Filtering* block. The DDC itself does implement this filtering with two filters, called *cfir* and *ffir*. The exact nature of these filters is subject to a subsequent work and thus only pre-calculated filters have been used. This is why the bandwith can only be one of 0.085, 0.17, 0.34, 0.40, 0.50, 0.75.

Another point here: the filters in *filters.c* are defined for twice this bandwith. This is because of the internal workings of the DDCs and can only be understood through a thorough study of the Graychip-documentation, see also the document about the DDC by Ignace.

21.3.2 sig_type

The signal-types are defined for the software-radio and can be chosen to be one of the following. The choice of the signal-type influences also whether the DDCs are used or not.

SIG_COMPLEX_ICS is the standard mode of the ICS-card. In this mode, DDCs are enabled and the signal is outputted in baseband.

SIG_S16 bypasses the DDCs and transfers the direct output of the ADCs. This has been tested with one ADC only. It is useful for processing high-bandwith signals.

SIG_COMPLEX_S16 is only possible with a re-programmed FPGA on the ICS-cards. DDCs are enabled, only one channel is transmitted, but not in the SIG_COMPLEX_ICS format that takes 128bits per sample, but in a more simple SIG_COMPLEX_S16 format, that only takes 32bits per complex sample, thus allowing 4 times more storage.

Chapter 22

Interface

The following files have been rewritten:

```
Include/antenna.h
Base/Antenna/ICS/*
Base/Antenna/Common/antenna.c
```

In table 22.0.3 you can see the newly defined commands.

22.0.3 New commands defined

Name	Short description
swr_ant_ics_init	Initialises the ics-cards, should be called first
swr_ant_ics_get_fs_rx	Returns the real sampling-frequency
swr_ant_ics_get_fs_tx	Returns the real sampling-frequency
swr_ant_ics_rx	Sets the bandwith and the if-frequency
swr_ant_ics_rx_freq	Sets the if-frequency of the reception, also while the radio is running
swr_ant_ics_tx	Sets the if-frequency of the transmission, also while the radio is running
swr_ant_ics_clk	Sets the speed of the clock-reference, as well as the multiplication-factor of the DACs
swr_ant_ics_start	Starts the transmission
swr_ant_ics_stop	Stops the transmission
swr_ant_ics_io	Returns the time left to reach a certain position in the reception
swr_ant_ics_set_synth	Sets the synthesizer on the RF-card
swr_ant_ics_write_ddcs	Writes the calculated values to the DDCs, can't be called while the radio is running!

22.1 int swr_ant_ics_init(fs_rx, fs_tx, ch_rx, ch_tx, sig_type);

This is the first function to call to initialise the cards. This function can only be called when the cards are not running, as some very basic parameters are defined in here.

22.1.1 fs_rx

Type: double

Description:

sampling frequency for the reception, in Hz. Must be between 10kHz and 3MHz. The sampling-frequency on the reception has some constraints, described in the function *swr_ant_ics_clk*. This sampling-frequency is given in complex samples, so that a sampling frequency of 10MHz gives a theoretical bandwith of 10MHz! See also *swr_ant_ics_rx*

22.1.2 fs_tx

Type: double

Description:

sampling frequency for the transmission, in Hz. Must be between 10kHz and 3MHz. The sampling-frequency of the transmission has some constraints that are described in the function *swr_ant_ics_clk*. This frequency is in complex samples! Please be aware that the DACs have an in-built filter that cuts the useful signal to about 75% of the bandwith of fs_tx. So if fs_tx is 1MHz, the useful bandwith is about 750kHz.

22.1.3 ch_tx

Type: uint

Description:

how many rx-channels, between 0 and 4 —

22.1.4 ch_tx

Type: uint

Description:

how many tx-channels, between 0 and 4

22.1.5 sig_type

Type: swr_signal_type_t

Description:

the desired type of the rx-channels, it depends on the program in the FPGA. A version exists for a one-channel, 32-bit complex reception mode (16bits real and 16bits imaginary), using the DDCs but allowing for a longer aquisition time. In this mode, *ch_tx* must be 0 and *ch_rx* 1. The different signal types correspond to the following:

SIG_COMPLEX_ICS is the signal used by most of the radio-systems, as it allows to use the DDC and works with complex signals on the reception path

SIG_S16 is used for the 802.11. Here the DDCs are completely bypassed and raw ADC-material is received.

SIG_S32 not implemented yet: DDCs work in wideband real-mode only

SIG_COMPLEX_S16 uses the modified FPGA to implement a 1-channel 32 to 16 bit conversion, including the DDCs. Used for GPS-reception.

After this function has been called, everything is set up so that *swr_ant_init* can be called. All other funtions defined in here are only for more special needs.

swr_ant_ics_init calls *swr_ant_ics_clk(100e6, 4, 50e6)* which initialises the multipliers on the ICS-cards.

fs_rx and fs_tx have to meet certain criterias, so the final values may differ from the chosen ones! To get the real values, use the functions *swr_ant_ics_get_fs_rx* as well as the function *swr_ant_ics_get_fs_tx*, which return the re-calculated values.

22.2 double swr_ant_ics_get_fs_rx(void);

22.3 double swr_ant_ics_get_fs_tx(void);

These function get the actual sampling frequency as calculated by *swr_ant_ics_clk*. They return the actual frequency in Hz, which may differ from the settings, because of the limitations of the cards.

22.4 void swr_ant_ics_rx(ch, fc, W);

22.4.1 ch

Type: int
Description: The affected channel, 0-3

22.4.2 fc

Type: double
Description: The new center-frequency

22.4.3 W

Type: double
Description: The bandwith of the final samples that contain data, relative to fs_rx set by swr_ant_ics_init. W $\in$ 0.085, 0.17, 0.34, 0.5, 0.75. If a non-existant bandwith is selected, the next-higher (or highest available) will be chosen.

fc is re-calculated to fit into $[0..\frac{fs_rx}{2}]$, using the availability of different Nyquist windows. First it is converted relative to the sampling-frequency:

$$f = \frac{fc}{fs_rx}$$

Then it is converted using:

$$f_p = \frac{1 - |(f - \lfloor f \rfloor) * 2 - 1|}{2}$$

f_p is calculated with 31 bits precision. For a fs_rx of 100MHz this gives a precision of $\frac{1}{20}$ Hz.

22.5 void swr_ant_ics_rx_freq(ch, fc);

Function to set only the intermediate-frequency of the reception-part.

22.5.1 ch

Type: int
Description: The affected channel, 0-3

22.5.2 fc

Type: double
Description: The new intermediate-frequency, see also the function swr_ant_ics_rx

Because it is very difficult to re-program the DDC-chip while the software-radio is running, this function has been written to only re-program the intermediate-frequency and nothing else. This function only calculates the new value to be programmed in the DDC-chip. The final programmation of the chip is done in the DMA-interrupt so that the transmission is not interrupted.

22.6 void swr_ant_ics_tx(ch, fi_tx);

This sets the intermediate frequency 'fi_tx' for the tx-channel. For the re-calculation of the intermediate frequency, see *swr_ant_ics_rx*.

This function can be used without problems during runtime.

22.6.1 ch

Type: int

Description: The affected channel, 0-3

22.6.2 fi_tx

Type: double

Description: center-frequency, in Hz

22.7 void swr_ant_ics_clk(f_adc dac_mult, f_dac);

The ICS-cards usually are clocked by an external 50MHz-clock. If you set a new clock-frequency, fs_rx and fs_tx are re-calculated and might differ. A frequency set with *swr_ant_ics_init* might be possible with f_dac=50MHz, but not fit correctly into f_dac=25MHz!

22.7.1 f_adc

Type: double

Description: the external frequency in Hz

22.7.2 dac_mult

Type: int

Description: the multiplier. One of 1, 4..16

22.7.3 f_dac

Type: double

Description: the external frequency in Hz

22.8 void swr_ant_ch_start(void);

Here the channel begins to send/receive data. All initialised antennas begin at once to send/receive.

22.9 void swr_ant_ch_stop(void);

Here the channel interrupts sending/receiving data. It may be that later on another transmission starts.

22.10 int swr_ant_ch_io(slot);

This function returns the time to wait so that the next slot will be sent/received.

22.10.1 slot

Type: int
Description: read/write up to this block

22.10.2 return

Type: int
Description: The time in micro-seconds (10^{-6}) to wait

22.11 void swr_ant_ch_set_synth(ch, RF, side);

Sets the synthesizer on a given RF-board. Due to some constraints, the RF-boards take quite some time to behave stable in a new frequency. This is in the order of a couple of ms.

22.11.1 ch

Type: int
Description: The affected channel, 0-3

22.11.2 RF

Type: double
Description: The frequency in Hz: [2.3..2.5] * 10^9

22.11.3 side

Type: int
Description: low- or high-injection:

0	Auto
1	low-injection
2	high-injection

22.12 void swr_ant_ch_set_freq_diff(ch, freq_diff);

As described in the function *swr_ant_ics_tx*, the value for the intermediate tx-frequency has a resolution of 32bits. With this function, you can set an offset to the 'official' value.

22.12.1 ch

Type: int
Description: The affected channel, 0-3

22.12.2 frec_diff

Type: long int
Description: An offset to the set value. The step is $\frac{fs_tx}{2^{32}}$, where fs_tx includes an eventual multiplicator (usually 4).

22.13 write_ddcs(void);

A function called *ics554_ddc_set_cic* is used to set the decimation to the desired value. The main problem is the calculation of the attenuation of the signal. This function has been tested for a wide range of decimations and should work without problems.

Chapter 23

FPGA

For the work on the software-radio, we have two versions of the FPGA:

STD which is the standard off-the-shelf version, that works for SIG_COMPLEX_ICS and SIG_S16

GPS which is used for the GPS SIG_COMPLEX_S16

Both of these modes can be programmed using a simple Tool. This can be found in the directory FPGA. If all you want is to reprogram the FPGA, here is how to proceed:

1. cd into the directory
2. ./compile
3. program :
   - for the GPS-version:
     ./program gps
   - for the standard-version:
     ./program std

Of course, if you want to switch back and forth between versions, you don't have to enter the "./compile"-command every time.

23.1 Directories

Module The kernel-module to talk to the ics554-card

Program The actual programmer for the FPGA

Api The Application Programming Interface, the glue between *Module* and *Program*

inc The include-files necessary

All these files are nearly identical to the software delivered on the ICS-CDs of the software-development pack. The only change has been done to the *Program*, so that it accepts the command-line switches and doesn't wait for a key.

23.2 Testing the version

Before running the radios it is a good idea to verify the version of the program in the FPGA. This can be done by running the following program:

```
cd
$SRADIO/Test/FPGA
make test
```

This will tell you whether the actual program supports either STD or GPS mode.

Chapter 24

Tidbits

24.1 DMA-considerations

Size of the slots and FIFOs:
$\frac{512k}{2*16} = 16k$ $\frac{512k}{2*4} = 64k$

For 1 slot	Rx - ics554	Tx - ics564
Samples	2560*2=5120=5k	2560*2=5120=5k
Bytes	5120*16=81920=80k	5120*4=20480=20k
FIFO-size [bytes]	65536*8=512k	65536*4*2=512k
Max samples		

The samples per slot is given by the pseudo-UMTS system of the software-radio and is fixed for the moment at 2560 samples. As we do an oversampling by a factor of 2, this gives 2560*2=5120 samples per slot, for both the ics554 and ics564.

The total bytes is calculated as $samples * bytes_per_samples$, where $bytes_per_samples$ is:

4 = 16bit real + 16bit imaginary

16 = 32bit real + 32bit imaginary, but for two channels, as we can't treat only one channel at a time in the ics554-card

According to the ics-554 documentation E10681 Rev.B p. 55, the FIFO is sized at 65536 * 64bit-values, which gives 512kBytes. The ics-564 documentation E10734 Rev.- p. 9 gives the size 65536 16bit-samples as "approximately one quarter of the FIFO-size". So the total is 65536*4*2 = 512kBytes.

24.1.1 Conclusion

The DMA-size will be dependant on the slot-size. So, there will be 1 slot per DMA-transfer. This limits the slot-length to 16kSamples for the Rx-part, but that should be OK.

Care has to be taken for small slot-sizes, as the DMA-transfer happens in two steps:

1. Fill up the FIFOs on the ics564
2. Start the transfer
3. Wait for underflow of the FIFOs and re-transmit

24.2 Server

The old implementation of the server takes DMA_BLOCK-sized packets and works on these. In order to rewrite only what is necessary, I adjusted the Base/Antenna/Simul_ics in a way to chop down the slots into DMA_BLOCK-sized packets. Too bad if it is not a multiple of these. There is no check against that.

24.3 Resampler

A big mess that nobody understands. Hopefully the work of the student this summer sheds some light on this.

24.4 Samples, Chips and Symbols

The current setup has 2 Samples for 1 Chip, and 1 Chip for 1 Symbol (without a spreading-sequence). This means, that there is a 2 x oversampling.

Part IV

HOWTOs

Chapter 25

From Conception to Measurement

Of course the main idea behind the MSR is that you're able to write new modules for it. This chapter will give an introduction with an actual example of a module as well as an implementation of a radio-transmission. After this you should be able to create your own modules and put them into use. An important introduction can be found in chapter 31. You should also already have run the example in chapter 28. This part is a bit heavy on coding, but you won't be able to write modules without a good knowledge of C.

25.1 Defining New Modules

In here you will learn the most important things about a module, how it works, how to use it, and how to extend it. This example is already present in the tree, but you can read this section to get a feel of it.

The goal of this module will be to measure the SNR of the signal. Even though this functionality is already implemented in a module, it is a nice idea to have a possibility to compare the results of the two approaches. The existing module compares the received training sequence with the original in order to calculate the SNR. As the training sequence is only part of a transmitted slot, it is a good idea to compare this SNR with the SNR computed in here.

In order to calculate the SNR differently, we will transmit a random sequence and then compare it after the transmission. This is depicted in fig. 25.1. In order to know the exact amplitude, it is important to know the random sequence in advance. This is done by setting the *seed* parameter of the random-module.

Once we have the received signal $y = y_r + iy_i = [y_{0r}y_{1r}...y_{n-1r}] + i[y_{0i}y_{1i}...y_{n-1i}]$ and the transmitted signal , we can calculate the amplitude:

$a = \frac{\sum_{l=0}^{n-1} y_{lr} sign(x_{lr}) + y_{li} sign(x_{li})}{n}$

and also the variance:

$v = \frac{\sum_{l=0}^{n-1} (y_{lr} - a\,sign(x_{lr}))^2 + (y_{li} - a\,sign(x_{li}))^2}{2n}$

and the signal to noise ratio is then

$SNR = 10 * \log(\frac{a^2}{v})$

The correctness of this assertion is left as an exercise to the reader.

Figure 25.1: The example slot

25.1.1 The Files

Now that we know what it is about, let's have a look at the written files. For your information you will learn where the templates for the files come from in every section. The discussion then is only about the parts that have been added. In the directory *Modules/Signals*, there is a directory called *SNR*. in there you find the code for the SNR-module. The MSR knows about this directory because of the file *Modules/Signals/Makefile* that has an entry *SNR* in the list of *DIRS*. You can have a look at this Makefile to see it.

The template files come from the *Conventions* directory and are called:

```
Conventions/multi_*.c
Conventions/Makefile.module
```

Most of the modules come in two parts: one sending part and one receiving part. But as they usually are used in a pair, they are put together into one module. So as not to be confused with different modules, they are renamed to:

```
multi_template.c -> snr.c
multi_template_send.c -> snr_send.c
multi_template_rcv.c -> snr_rcv.c
Makefile.module -> Makefile
```

25.1.1.1 snr.c

Now look first at the file *snr.c* and look at the places that contain some documentation. It is really important to keep this documentation up-to-date, so that other people know what it's about:

```
SNR - measure the signal to noise ratio of a transmitted slot.
In order to do this, we send a slot of known random data that
is measured on the other side.
```

This is all that is needed. Not a big description, just enough to know what it's about.

What it does is the following: once the module *snr* is loaded into the memory, the function *spc_module_init* is called, which in turn calls *rcv_module_init* from *snr_rcv.c* and *send_module_init* from *snr_send.c*. These two functions are responsible to tell the CDB about their name, their input and output as well as their paramters.

25.1.1.2 snr_send.c

This is the part that prepares the slot.

documentation. At the top of the file, you see again a short description of the module

```
snr_send.c - sends a slot of random symbols
```

and a bit further down (after the copyright message) a bit more of description.

```
This module expects some random input that is then modulated
using QPSK modulation so that the noise can be mesured. It can
send the QPSK symbols either on the axis or in the corners.
```

config-structure We want the user to be able to change the amplitude of what we send over the channel. So, edit the config-structure, and make it something like:

```
typedef struct {
  // The amplitude of the generated QPSK-signal
  int amplitude; // 32767
  // The QPSK-type, 0->in the corner, 1->on the axes
  int type; // 0
} config_t;
```

It may seem strange that we take an integer for the amplitude, but you have to know that the signals are all in 16-bit integers, so what usually is between $+1$ and -1 is now between $+32767$ and -32768 .

private-structure Once the user changed the configuration, we will store it in our private variable. This is more for convenience than anything else:

```
typedef struct {
  int amplitude;
  int type;
} private_t;
```

The other structure may be left empty, we don't need it for this example. Just after the structures is a function called

send_init Why another initialisation function, you might ask. Well, remember from chapter 8.2 that first the module is registered with the CDB, before it is possible to instantiate it. So, this function is what is called each time this module is instantiated. In our example, we just want to put a default-value in the amplitude-part of the configuration, so add this line:

```
config->amplitude = 32767;
config->type = 0;
```

32767 is the maximum number that we can have in a 16-bit signed variable.

send_configure_input This function is called whenever the MSR wants to know what the size of the input should be, given the size of the output. So, for each 2 bit of input, we create one symbol with the QPSK representation. This means that two bits of input create one symbol of output, at least for an even number of symbol-outputs. The only tricky part here is that the input is not counted in bits, but rather in bytes. So $size_{input} = \frac{2 * size_{output}}{8}$, which is written in this function as

```
size_in(0) = ( size_out(0) + 7 ) * 2 / 8;
```

This assures that we always have enough bits to write to the output.

send_configure_output The same as before, but this time the opposite direction:

```
size_out(0) = size_in(0) * 8 / 2;
```

send_reconfig We use this function to copy the configuration-data to our private structure:

```
private->amplitude = config->amplitude;
private->type = config->type;
```

send_pdata Now comes finally the processing function. This is where the main action takes place. Let's first define some variables:

```
// Definition of variables - don't touch
stats_t *stats;
int i, amp;
U8 *in;
SYMBOL_COMPLEX *out;
```

To get to the input and output-buffers, we have to do the following:

```
in = buffer_in(0);
out = buffer_out(0);
```

The first line deletes the *data* bit on the input-port, signaling the MSR that the input-port is free again to receive some data. Furthermore it returns a pointer to the input-buffer of this module. The second line gets the output-buffer of this module and sets the *data* bit on the output-port. Once this function returns, the MSR checks for the *data*-bit in the output-port and, if it is set, handles the processing to the next module.

OK, now we just have to process the data:

```
// Fill the slot with random QPSK symbols
for ( i=0; i<size_out(0); i++ ){
  switch( 2 ){
  case 1:
    // The amplitude in this case is
    // Sqrt( Re^2 + Im^2 ) and thus the
    // desired amplitude has to be divided by
    // sqrt(2)
    amp = private->amplitude / sqrt( 2 );
    out[i].real = ( 2 * ( *in & 1 ) - 1 ) * amp;
    *in = *in >> 1;
    out[i].imag = ( 2 * ( *in & 1 ) - 1 ) * amp;
    *in = *in >> 1;
    break;
  case 2:
    amp = private->amplitude;
    if ( *in & 1 ){
      *in = *in >> 1;
      out[i].real = ( 2 * ( *in & 1 ) - 1 ) * amp;
      out[i].imag = 0;
    } else {
      *in = *in >> 1;
      out[i].imag = ( 2 * ( *in & 1 ) - 1 ) * amp;
      out[i].real = 0;
    }
    *in = *in >> 1;
    break;
  }
  // Get the next input-byte of random
  if ( i && !( i % 4 ) ){
    in++;
  }
}
```

You might not be completely fond of this example, but it works (yet to check).

send_custom_message This function is not needed and can be deleted

send_module_init Registers this module with the CDB. The CDB first wants to be informed about the type of module to be attached [1]. In our case, we have one input, one output, one config-parameter and 0 stats-parameter:

```
desc = swr_spc_get_new_desc( 1, 1, 2, 0 );
```

Then we have to tell about the config-parameter, the input-type and the output-type:

```
UM_CONFIG_INT( ''amplitude'' );
UM_CONFIG_INT( ''type'' );
UM_INPUT( SIG_U8, 0 );
UM_OUTPUT( SIG_SYMBOL_COMPLEX, 0 );
```

In order for the MSR to know what functions to call in what case, we have to define 'call-back functions'. As these are always the same, they are already pre-defined in the templates, and we only have to delete the *send_custom_msg* entry. Now the module-description is complete, save for the name:

```
send_id = swr_cdb_register_spc( &desc, "snr_send" );
```

25.1.1.3 snr_rcv.c

Let's start with the comment in the beginning of the file:

```
This module receives the stream from the matched filter
and the stream of random-signals that are supposed to be
the same that have been used by the snr_send. It then
calculates the amplitude, the variance and the snr.
```

This means that this module has two inputs: one from the channel, and another one from the random-module.

config-structure Again we have the possibility to change the type:

```
typedef struct {
  // The QPSK-type, 0->in the corner, 1->on the axes
  int type;
} config_t;
```

stats-structure So we're able to retrieve the SNR from the outside, we have to write it in this structure:

```
typedef struct {
  double snr;
} stats_t;
```

[1] For further information, look at 8.2.2

rcv_init Let's just start with a SNR of -2.3:

```
stats->snr = -2.3;
config->type = 0;
```

rcv_reconfigure

```
private->type = config->type;
```

The functions *rcv_configure_input* and *rcv_configure_output* can be deleted, as this module is at the end of the chain.

rcv_pdata Here goes the working function. It's just about implementing the above formula. Let's go through step by step. Definition of variables:

```
stats_t *stats;
SYMBOL_COMPLEX *in, *buf_rnd;
U8 *in_rnd;
double signal = 0., noise = 0.;
int i;
```

Then we have to make sure that we have both the signal from the channel and the random-data:

```
if ( !data_available(0)    !data_available(1) ){
  PR_DBG( 4, "Not all data available yet\n" );
  return 0;
}
```

Now we reconstruct so that we can implement the formula given above. Instead of calculating and then taking the sign of it, we directly calculate with an amplitude of ± 1 . Again, once for the QPSK-signals on the axes, and once for the signals tilted by $\frac{\pi}{4}$

```
in_rnd = buffer_in(1);
buf_rnd = swr_malloc( size_in(0) * sizeof( SYMBOL_COMPLEX ) );
for ( i=0; i<size_in(0); i++ ){
  switch( private->type ){
  case 0:
    buf_rnd[i].real = ( 2 * ( *in_rnd & 1 ) - 1 );
    *in_rnd = *in_rnd >> 1;
    buf_rnd[i].imag = ( 2 * ( *in_rnd & 1 ) - 1 );
    *in_rnd = *in_rnd >> 1;
    break;
  case 1:
    if ( *in_rnd & 1 ){
      *in_rnd = *in_rnd >> 1;
      buf_rnd[i].real = ( 2 * ( *in_rnd & 1 ) - 1 );
      buf_rnd[i].imag = 0;
    } else {
      *in_rnd = *in_rnd >> 1;
      buf_rnd[i].imag = ( 2 * ( *in_rnd & 1 ) - 1 );
      buf_rnd[i].real = 0;
```

```
    }
    *in_rnd = *in_rnd >> 1;
    break;
  }
  // Get the next input-byte of random
  if ( i && !( i % 4 ) ){
    in_rnd++;
  }
}
```

Now we can calculate the amplitude of the signal:

```
in = buffer_in(0);
// Calculate signal energy
for ( i=0; i<size_in(0); i++ ){
  signal += (double)( in[i].real ) * buf_rnd[i].real +
    (double)( in[i].imag ) * buf_rnd[i].imag;
}
signal = signal / size_in(0);
```

And the noise-variance:

```
for ( i=0; i<size_in(0); i++ ){
  noise += pow( (double)in[i].real - signal * buf_rnd[i].real, 2 ) +
    pow( (double)in[i].imag - signal * buf_rnd[i].imag, 2 );
}
noise = noise / size_in(0) / 2;
```

Finally we can calculate the snr:

```
PR_DBG( 2, "signal_amp: %i, noise_amp: %g\n",
        (int)signal, noise );
// And write the snr
swr_sdb_get_stats_struct( context->id, (void**)&stats );
if ( noise > 0 ){
  stats->snr = 10 * ( log10( signal ) * 2 - log10( noise ) );
}
swr_free( buf_rnd );
```

Again, the *rcv_user_msg* is not used and can be deleted. **rcv_module_init** We have only 1 input, no output, 1 config-variable and 1 stats-variable, and lots of functions are not used:

```
desc = swr_spc_get_new_desc( 1, 0, 1, 1 );
UM_STATS_DOUBLE( ''snr'' );
UM_INPUT( SIG_SYMBOL_COMPLEX, 0 );
desc->fn_init              = rcv_init;
desc->fn_reconfigure       = rcv_reconfig;
desc->fn_process_data      = rcv_pdata;
desc->fn_finalize          = rcv_finalize;
rcv_id = swr_cdb_register_spc( &desc, "snr_rcv" );
```

25.1.1.4 Makefile

In the makefile we have to tell the final name of the module, as well that we use the math-library:

```
MODULE_NAME = snr
MATH = true
```

25.1.2 Compile it

Now you can try to compile it by typing *make* on the command-line. If there are any errors, try to fix them, the above lines should work, they have been tested. In order to include this module even better in the MSR, you can add the name of the directory to the file *Modules/Signals/Makefile* in the line *DIRS* = . Like this a top-level *make* will also update the SNR-module.

25.2 Testing

Up to now only the module has been written. It is not yet in a usable state, as it is only registered with the CDB, but not yet instantiated. Theoretically we could write everything in the module to make an instance, but this would turn upside-down the idea of modules. So what we need is an own program that implements the chain and runs it, just to look how good it runs.

Perhaps as a surprise, this program will again be a module, but this time a module that does actually something. Implementing a simple chain. So there is a function called *um_module_init* that will be called upon inserting the module. This function itself creates a new thread that will be used to create the chain. In order to be compatible for further RTLinux implementation, we have to do this two-step calling.

25.2.1 The Directory

In the MSR, there is a directory called *Test* which holds already different tests. The test for the SNR is of course in a directory called *Test/SNR*. The templates for the test-module are in

```
Conventions/test_template.c
Conventions/Makefile.module
```

Again, for easier handling they are renamed:

```
test_template.c -> test_snr.c
Makefile.module -> Makefile
```

25.2.2 Makefile

The makefile wants to know the name of the module, which is *test_snr*, as well as the modules to load in order for the MSR to function correctly. In our case, these are the modules *random, snr, midamble, rrc* and *block*:

```
MODULE_NAME = test_snr
DEPENDS = random snr chest rrc block
```

25.2.3 test_snr.c

Let's have a look at the documentation:

```
Make a simple chain:
random - snr_send - chest_send - rrc - block -
         rrc_rcv - chest_rcv - snr_rcv
and additionally:
random - snr_rcv(2)
```

Then we can create our main-function, which is called *start_it* for the test-program.

25.2.3.1 start_it

The first thing we have to do is to create a *chain* of modules. A chain is a logical suit of signal-processing modules, that take some input and produce some output that is handled further down the chain.

When using the *swr_chain_create* functionwe give a list of all modules, that will be automatically connected together, and finish the list with END_CHAIN. In this call to *swr_chain_create*, you see three different kind of macros, *NEW_SPC_VAR*, *NEW_SPC* and OLD_SPC_IN all of which are described in 8.2.2. In short, while the former allows you to give a variable where a reference to the module will be stored, the latter just creates the module and connects it, without giving the reference of the created module. The third takes an already defined module for further connections.

```
swr_sdb_id rnd, mafi, snr_rcv;
test_chain = swr_chain_create(
                     NEW_SPC_VAR( ''random'', rnd ),
                     NEW_SPC_VAR( ''snr_send'' ),
                     NEW_SPC( ''chest_send'' ),
                     NEW_SPC( ''rrc'' ),
                     NEW_SPC_VAR( ''block'' ),
                     NEW_SPC_VAR( ''rrc_rcv'', mafi ),
                     NEW_SPC_VAR( ''chest_rcv'', mafi ),
                     NEW_SPC_VAR( ''snr_rcv'', snr_rcv ),
                     END_CHAIN );
swr_sdb_set_config_int( mafi, "cacl_taps", 8 );
test_chain_2 = swr_chain_create(
                     NEW_SPC_VAR( ''random'', rnd2 ),
                     OLD_SPC_IN( snr_rcv, 1 ),
                     END_CHAIN );
```

So we have created a chain. The modules have the following function:

random	create random bytes
snr_send	our module created above, takes some random bytes as input, and creates a QPSK output
chest_send	inserts the training-sequence in the middle of the stream
rrc	Root Raised Cosine pulse-shape filtering
block	a very simple channel-simulation
rrc_rcv	Applies again the Root Raised Cosine filtering
chest_rcv	uses the training-sequence to make a channel-estimation and does a matched-filtering on the received samples

snr_rcv our module to calculate the SNR

If we compile and test our module with *make; make user*, this chain will be created and the program will exit. What we forgot is to really use this chain. For this, the *random* module listens to user-messages, and begins creating a random-output whenever it receives such a user-message. But first we have to make sure that both random-modules create the same values:

```
swr_sdb_set_config_int( rnd, "seed", 0x1234 );
swr_sdb_set_config_int( rnd2, "seed", 0x1234 );
```

Then we adjust a bit the amplitudes, so that we don't run all the time on the edge:

```
swr_sdb_set_config_int( snr_send, "amplitude", 16384 / 4 );
swr_sdb_set_config_int( mid, "amplitude", 16384 / 4 );
```

To make it a bit more nice, we have a look at different values, using the *sigma* parameter of the *block* module:

```
for ( i=0; i<50; i+=5 ){
  swr_sdb_set_config_double( block, "sigma", i );
  swr_sdb_send_msg( rnd, SUBS_MSG_USER, NULL, -1 );
  swr_sdb_send_msg( rnd2, SUBS_MSG_USER, NULL, -1 );
  PR( "Amp: %2i:%2i, Noise: %3i:%3i  SNR : %5.5g - %5.5g = %5.5g\n",
      swr_sdb_get_stats_int( mafi, "mid_amp" ),
      swr_sdb_get_stats_int( snr_rcv, "amp" ),
      swr_sdb_get_stats_int( mafi, "noise_var" ),
      swr_sdb_get_stats_int( snr_rcv, "var" ),
      swr_sdb_get_stats_double( mafi, "snr" ),
      swr_sdb_get_stats_double( snr_rcv, "snr" ),
      swr_sdb_get_stats_double( mafi, "snr" ) -
      swr_sdb_get_stats_double( snr_rcv, "snr" ) );
}
```

And after a *make; make user* you should see something like:

```
Amp: 63:63, Noise:   1:1   SNR : 34.40 - 34.24 =  0.15426
Amp: 62:62, Noise:   5:6   SNR : 28.52 - 28.10 =  0.42007
Amp: 62:62, Noise:  27:26  SNR : 21.50 - 21.58 = -0.077267
Amp: 62:62, Noise:  66:63  SNR : 17.61 - 17.83 = -0.21601
Amp: 62:61, Noise: 122:111 SNR : 14.96 - 15.34 = -0.37944
Amp: 61:61, Noise: 190:187 SNR : 12.89 - 13.07 = -0.18071
Amp: 58:59, Noise: 179:219 SNR : 12.72 - 12.01 =  0.70692
Amp: 65:64, Noise: 317:322 SNR : 11.23 - 11.06 =  0.16854
Amp: 66:63, Noise: 421:393 SNR : 10.14 - 10.15 = -0.016712
Amp: 65:65, Noise: 528:555 SNR : 9.026 -  8.83 =  0.18867
```

So you see that our method gives more or less the same results as the *snr* calculated in the *matched_filter*.

25.3 Going Over the Air

OK, now that the module is written, a simple test-case shows that our module works, we can go on and write a simple radio that transmits the SNR-slot and then receives it and shows the result. We will make a simple radio that has a master, the BaseStation, that transmits the synchronisation-signal, and a client, the MobileStation, that synchronises to it and sends a SNR-slot in return.

25.3.1 The Directories and Files

It will be a radio, so we find the code in the directory *Radios/SNR*. The base for this radio is the simple-radio that can be found at *Radios/Simple* and contains the following files:

```
simple_radio.h
Makefile
BS/Makefile
BS/radio_bs.c
MS/Makefile
MS/radio_ms.c
```

To reflect the fact that it isn't the *simple* radio anymore, we have to adjust the names in the Makefiles. The first lines in *BS/Makefile* contains:

```
MODULE_NAME = radio_snr_bs
DEPENDS = rrc synch energy mapper chest random \
          rndstr spread sink cch macro_sch snr
```

and in *MS*[2]*/Makefile* reads:

```
MODULE_NAME = radio_snr_ms
DEPENDS = rrc synch energy mapper chest random \
          rndstr spread sink macro_synch macro_sch snr
```

25.3.2 README

This file also reflects the changes and has a very small documentation in it.

25.3.3 MS/radio_ms.c

There is a lot of things to say about the basic system. You can find an introduction in 31. Here we just try to focus on the things necessary to run our SNR-module over a real channel.

A very short simplification: when the mobile synchronises for the first time to the base-station, it creates the necessary chains. This happens in the function *synchronised*. Near the end of this function, you have to replace the declaration of the UP-chain with the following chain:

```
// And setup a simple UP-chain...
ch_up = swr_chain_create(
                      NEW_SPC_VAR( "random", rnd ),
                      NEW_SPC( "snr_send" ),
                      NEW_SPC( "chest_send" ),
                      NEW_SPC( "rrc" ),
```

[2] Microsoft

```
                    OLD_SPC_IN( stfa, 1 ),
                    CHAIN_END );
swr_stfa_notice_sdb( stfa, 1, rnd );
PR( "Ready to go" );
```

You see a new macro in the function *swr_chain_create* here, which is called *OLD_SPC_IN* and uses the *stfa* module. The macro tells the function that this module has already been created before. The module *stfa* is quite important in the MSR: it creates the link between the modules and the actual hardware. So an input to the stfa is like a connection to the antenna.

As we don't know exactly when the mobile will be synchronised with the base-station, we set the seed of the *random* module every frame to the same value. Like this we're sure that both the BS and the MS[2] have the same random-values. To achieve this, the function *do_send_up* is handy. It is called once in a frame, and we can put the following line in there:

```
swr_sdb_set_config_int( rnd, "seed", 0x1234 );
```

That's it for the mobile-station part.

25.3.4 radio_bs.c

This part of the radio sets up the chains for reception anyway and then just waits on what happens. As it gives the synchronisation and doesn't need to wait for it, it is much more simple than *radio_ms.c*. So we can directly change the construction of the UP-chain in the function *start_it* to:

```
PR( "Setting uplink in slot 1\n" );
ch_rach = swr_chain_create(
                OLD_SPC_OUT( stfa, 1 ),
                NEW_SPC( "rrc_rcv" ),
                NEW_SPC_VAR( "chest_rcv", mafi ),
                NEW_SPC( "snr_rcv" ),
                CHAIN_END );
ch_rach_2 = swr_chain_create( NEW_SPC_VAR( "random", rnd ),
                              OLD_SPC_IN( snr_rcv, 1 ),
                              CHAIN_END );
swr_sdb_set_config_int( sch, "mafi0", mafi );
while ( looping++ ){
  usleep( 1000000 );
  PR( "mafi0: %g, mafi1: %g\n",
      swr_sdb_get_stats_double( mafi, "snr" ),
      swr_sdb_get_stats_double( snr_rcv, "snr" ) );
}
```

One last important thing we don't have to forget: once a frame we have to tell the random-module to generate some data. This is best done in the function called *do_send_down*, which is used once a slot. So we can insert there:

```
swr_sdb_set_config_int( rnd, "seed", 0x1234 );
swr_sdb_send_msg( rnd, SUBS_MSG_USER, NULL, -1 );
```

25.3.5 Running it with the channel-simulation

Again, to help track down errors, it is more adviseable to run it first in simulation-mode, like this you can track down errors much more simple. In order to do so, change to the directory *Radios/SNR* and type *make* to compile it, and *make server; make show_bsms* which should bring up two windows, one showing the mobilestation and another showing the basestation. If something goes wrong with the compilation, fix it and run *make* again. If something goes wrong with the display, type *make kill; make cleanproc* which should clean-up the directories, and then you can try *make server; make show_bsms* again.

25.3.6 Running the real thing

Now that you did all this work and the modules returned some decent values, you can be pretty sure that it shouldn't run havoc in real-time mode. So let's try it. First, you have to run the radio on the basestation, issuing a *make rf_show* from the *Radios/SNR/BS* directory. Then, on the other machine, you can run *make rf_show* from the *Radios/SNR/MS*^{2} directory. If everything is set up correctly, the hardware is OK and all things are nicely connected and plugged in (this will give another chapter or two, installing the hardware), you should again see two windows, one from the basestation and one from the mobilestation, and the values this time are REAL values. If you come this far, congratulations!

Chapter 26

Tools

A couple of different tools exist for the MSR, to show the internal state of the MSR, to act as a channel-server or build LDPC-codes. In this chapter you'll learn about the different tools and how to use them more accurately. If you have trouble with a tool, you can read here if you find some help.

26.1 Visualize

This tool is used to show the internal state of the MSR. It depends on the STFA-module to draw the other signal-processing modules. So, if there is no STFA module in the chain (which is the case for most of the programs in the *Test*-directory), Visualize can't show the complete chain.

26.1.1 Starting it

To run it, simply type *make show* and enter. This sets up the path so that the *qwt* library can be found. The executable first searches for a real-time MSR that puts its data into */proc/sradio*, then it searches for the most recent entry in */tmp/username/proc.** If it doesn't find any of these, it stops with an error. Optionally you can also give a path to the *sradio/sdb/* directory on the command-line.

Once the correct path has been found, the different modules are displayed on the screen. Optional modules that are not connected to anything are not shown on the screen. The whole display is updated once a second.

26.1.2 Mouse handling

With the left mouse-button you can drag around the whole screen, which is mostly useful on small screens, when not all chains fit on the screen.

The right mouse-button opens an option to exit the program when pressed on an empty part of the visualize-display. When pressed over a module, a menu pops up, where you can choose different display-options: stats and outputs. If a module has more than two stats-entries, you can choose which ones to display by choosing the corresponding entries. Each selected entry is put on top of the list of stats inside the block representing the module. Some of the stats-entries represent blocks of data, which will be displayed in a window apart.

26.1.3 Plotting of values

There is a possiblity to plot the *stats*-values into a seperate window. This can be used to log values of a certain module, or even to draw plots of one value agains the other. In order to create a plot, go to the *File* menu and chose *Stats Plot XY* or *Stats Plot Y(t).* Now you can click on a module in order to get a list of *stats* that shall be plotted. If you chose *Stats Plot XY*, you have to chose a second module and a second *stats.* Once the stats have been chosen, the plot-window updates once a second with a new value.

You can chose a new value for the udpate-time. The time is given in 1/100s of seconds. Be aware that for performance reasons the screen-update is only twice a second, even if the data-update-value is less than 50. No samples will get lost, only the update will appear slow. To enable long measurements without degrading performance of the *visualize* tool, only the 1000 most recent samples are shown. This assures that you can have 1 million or more samples, and still have *visualize* react to your requests. If you chose to *export to matlab*, all samples will be exported. If you chose *export to ps*, only the visible samples will be plotted.

26.1.4 Exporting values

Each plot-window has the possibility to export the values either as a postscript-file or to a matlab-file. In order to have a small preview of the data you'll export, you can click on the graphic. This will freeze the update, and allow you to 'chose' which data you want to export.

If there is lots of data, a general update will only show 1000 samples. When you click on the button.

26.1.5 Known bugs

During simulation-mode, it may be that an update of the screen occurs at the same time as an update from the MSR, which results in broken or incomplete chains. Usually this should 'heal' during the next update. If the same happens with a plot-window, it may be that you have to close this particular window, and reopen it again.

26.2 Channel-server

The channel-server takes the inputs of different radios, mixes them together and sends them back to the different radios.

26.2.1 Starting it

When you're in the *Main/* structure somewhere, usually it is enough to type *make server* to start the channel-server. This is only necessary once.

26.2.2 Known bugs

For the moment the channel-server and the simulations have to be run on the same computer. If you change something in the implementation of the channel-server, you have to restart it. The most simple way is to type *killall server* followed by a *make server*.

26.3 LDPC-code generation

In *Tools/LDPC* you find a program that takes descriptions of LDPC-codes and puts them in a module-readable way.

26.3.1 Starting it

First you need some descriptions of LDPC-codes. For this you have to ask Abdelaziz on how to do this. Then you can type *write code1 code2* for putting *code1* and *code2* into a file called *graphs.c* which has to be copied into *Modules/Coding/LDPC*. After a recompilation you're ready to use the new codes.

26.3.2 Known bugs

The length of the code is fixed for the moment at 4000 bits.

Chapter 27

Debugging

In a perfect world we wouldn't need this chapter. In a realistic world, however, one has to take into account possible errors. While every care has taken to make the framework of the MSR as error-free as possible, there still are bugs left. For sure. But usually they are difficult to find. So, if your module doesn't work, chances are big that you don't use the framework as it's supposed to be. This chapter will help you finding where the bug is. It is then up to you to find how it has to be fixed.

27.1 Debugging in user-space

When the software-radio runs in real-time, it is very difficult to debug the radio. This is due to the fact that no delay is permitted in either radio, or else synchronisation will be lost. On the other hand, in user-space, we don't have this issue. If one radio is stopped, all other radios will be stopped, too, and synchronisation will NOT be lost. For this reason, debugging in user-space is usually the only way to debug a misbehaving module.

This section gives an overview of how to use the GDB to detect some simple errors, like division by zero, $\log(0)$ or other exceptions.

27.1.1 Using Gdb with Tests

The first thing to do when a module doesn't run and stops with a *segmentation fault* is to use *make debug* instead of *make user*. This will call the GNU-debugger, run the module and stop on the offending instruction. The most common commands after this are:

ba	show the backtrace, where the most recent function is on top
print	prints a variable of the current context
up	moves one function deeper on the stack. If it stops on a function you don't know, you can use *up* a couple of times until you are in the MSR
list	lists the current context. Takes as argument a file and it's line-number, like *list sdb.c:234*

27.1.2 Debugging a Simulation

When running a simulation of two radios, the above doesn't work anymore. You have to start a channel-server, and the two radios. The most simple way to so is to run the server and the BS in one window, and the MS[1] in another window.

Sometimes the two radios behave nicely, but it is only after changing some configuration-value with the visualize-tool that the system crashes. For this reason we start a visualize-tool in a third window, after the BS and MS[1] are started. shell1, shell2 and shell3 denote three different windows or shells.

[1] Microsoft

```
shell1:SRadio/Radios/Simple/BS$ make server debug
shell2:SRadio/Radios/Simple/MS$ make debug
shell3:SRadio/Radios/Simple$ make show
```

This starts the server and the BS, then starts the MS[1]. The third line starts the visualize-tool, if you wish to do so. If an error is encountered, gdb will stop in one of the two windows, and you can use the commands described above to see what is wrong.

27.1.3 Using ddd

For better debugging, for example if your module doesn't crash, but doesn't do what you expect it to do, you can use *ddd.* To start it, you go to the same directory where usually you type *make user* and type *make ddd.* After this, you have to type *break main.c:36* and press F3 to start the program. Once it stops on the breakpoint, all necessary libraries have been loaded and also the modules you want to debug (given that you didn't make any errors in setting up the *Makefile*).

Now you can enter the name of a function to debug on top in the white box and press on *lookup* to the right of the box. Clicking with the right mouse-button on a line you can set a breakpoint. Repeat this procedure as often as necessary, then click on *cont* and watch what happens. *next, step*, right-clicking on variables are other nice options to take. Try it, break it. Debugging is an art!

27.1.3.1 Known bugs

DO NOT type *run* in the command-line of ddd. This runs the program without any command-line options that have been carefully crafted to work with ddd. Do use the run-button or the menu *Program-\textgreaterRun Again.* Optionally you can use the F3-key.

Chapter 28

Getting Started

In this chapter you'll learn how to set-up your computer so as to be able to run the MSR in simulation mode. If you want to work with the MSR, you should read this chapter. First you have to make sure that all necessary software is installed on your system, then you have to download the necessary package, compile it, and finally you can run the example.

28.1 Prerequisites

The MSR runs without problems on a linux-system. For the simulation-part, an installation of the RTLinux is not necessary. The MSR relies on some new packages, like a new linker and a QT-package. These should be available with RH7.3 and newer, Mandrake 8.2 and newer as well as Debian 3.1/sarge. The most common missing programs are:

linker — check with *ld version* whether it is something newer or equal than 12.13.90

qt — check with *qmake version* and see if the version of QT is bigger or equal than 3.0

libqwt — is used to display the graphics on screen. If you don't have it installed, ask your system-administrator to install it. Alternatively you can also install it in your home-directory.

If one of these is missing, please contact your system administrator or install the missing packages.

28.2 Installing the MSR

It is possible to use the MSR without the proper RF-hardware. For this you will use a channel-simulation. In this section you learn how to download and compile the MSR.

28.2.1 Download the software

From http://lcmpc10.epfl.ch/Menu/Download you can download the latest version of the MSR, which has a name like msr-*.tgz Once it is on your computer, you can place it in a convenient directory and untar it using *tar xzf msr-*tgz* This will create a directory named *Main* and a lots of subdirectories and files in there. Generally you will find README files in most of the directories. They are useful if you need to know what goes on in this special place. Do not hesitate to read them.

28.2.2 Compile the software

The software consists of two parts: the graphic display and the signal-processing modules. For the graphical part, you have to change into *Main/Tools/Visualize* and run *qmake visualize.pro* followed by a *make*. If everything goes well, you should have an executable called visualize. After this you can change in the *Main* directory and run *make* there. Supposing that everything goes well, you're ready to run the examples.

28.2.3 Common errors

28.2.3.1 While compiling 'Visualize' I get 'libqwt not found'

Make sure that libqwt is installed and check eventually the path in visualize.pro.

28.3 Running the examples

There are a couple of pre-defined radios in the subdirectory *Radios*/. The most simple is in *Radios/Simple*/. Using the following commands, you can display both the BS and the MS[1] of this simple example:

```
cd Radios/Simple
make server
make show_bsms
```

If the installation of the MSR has been carried out successfully, you should see now two windows popping up, showing the basestation that emits the synchronisation-signal, as well as the mobile-station that listens to this signal.

You can also run the other examples that you find in this directory, namely *Multiuser* and *LDPC* just to get an idea what the software-radio is all about. And remember: the same c-code also runs in real-time on RF-hardware!

[1] Microsoft

Chapter 29

Testing

Once a new module has been written, or if something new should be tried, a new testing-module should be written. A testing-module usually consists of a simple chain with some output that tells the user whether the test has succeeded or not. It is the first step towards writing a Radio and using a new or modified module in real-time.

29.1 Files

If you want to start a new test, create a new directory under *Test*/ and add the name to the line

```
DIRS = FirstChain Memory ...
```

of the file *Test/Makefile*. Like this your test will be automatically built when using *make whole*. Next *cd* in your new directory and copy some files:

```
cp ../../Conventions/test_template.c ../../Conventions/Makefile.module .
```

The *test_template.c* should be renamed to something fitting the pattern *test_*.c*, and this name should be appended to the line

```
MODULE_NAME ==
```

of the *Makefile* (which has been renamed from *Makefile.module*). Now you are ready to modify your *test_*.c* file and test it out, using

```
make
make user
```

You can also have a look into different files that you find in the directory *Testing/** to see different techniques on how to do strange things.

29.2 Main

The main-part of the module is in the function *start_it*. It is called once when the module is loaded. For a small testing-module you want to make perhaps only a small chain, something like

```
swr_chain_create( NEW_SPC_VAR( ''random'', rnd ),
                  NEW_SPC( ''modulator'' ),
                  NEW_SPC( ''demodulator'' ),
                  NEW_SPC_VAR( ''sink'', sink ),
                  CHAIN_END );
```

This leads to an empty chain, because all modules (with the exception of the *STFA* and the *block* module) have an input- and output-size of 0. So we need to set one module to a given size:

```
swr_sdb_set_config_int( sink, ''size'', 128 );
```

And while we're at it, we can tell the *sink* module, that it should count the occurences of the different values:

```
swr_sdb_set_config_int( sink, ''flag'', 2 );
```

For further explanation, you can turn to the explanation of the *sink* module.

Now that the chain has been created and configured, it still needs to be activated at least once, so that something happens:

```
swr_sdb_seng_msg( rnd, SUBS_MSG_USER, NULL, -1 );
```

Which sends a user-defined message to the *random* module. It will now traverse the whole chain and the *sink* module will output the occurances of the different values. To compile and start it, type the following commands:

```
make
make user
```

As the size of the sink-input is only 128, this will not be very representativ for the random-number generator. You may increase the size of the sink-input to something like 65536 or even more, to see how the random-number generator works.

Chapter 30

Using CVS

In this chapter you'll get an overview of the CVS[1]-structure we're using for our software-radio. It is assumed that you already worked sometimes with CVS[1] and that you know about the basic ideas and advantages of CVS[1].

CVS[1] stands for Concurrent Versioning System, and is the most widespread used tool to make sure that a group of developers can access the code at the same time without creating havoc. While it has some disadvantages, it is a well-tested, stable tool that does its job right.

The first section talks about the structure used, while the other sections talk about how to use the CVS[1] with this structure.

30.1 Structure

Usually one has a structure with one primary *Branch* where all developers commit their changes to. In our case this was not desired, as sometimes a developer changes huge structures during their work, and while the changes last, the code is unstable. Furthermore the code has to run on more than one machine at the same time, while needing localized compiling under *root.* This made it impossible to have a shared directory on all machines. So, what we need is:

- Seperate Development by giving each developer an own Branch
- Distributing of the new code using CVS[1]-command 'up'
- Easy merging through the CVS[1]-command 'join'

The first two points are easily done in CVS[1]. You can create branches, check them out, and then work in these quite comfortably. The difficulty arises when one tries to merge the changes back into the main-branch, perhaps even from different developers.

Here is an overview about what is happening in a case when we want to synch two branches, called *Main* and *Nicou.* "a", "b" and "c" are the original versions of three files, while "b'" and "c'" are modified versions of these files. "d" and "e" are new files that have been added later.

The 'server'-column displays what is stored on the cvs-server, that is, whenever you call "cvs commit", your 'local' changes are stored on the server, and with "cvs update", the changes stored in 'server' are written to 'local'.

Main		Nicou	
server	local	server	local
a b c	a b c	a b c	a b c
tag: lswN		tag: lswM	

[1] Concurrent Versions System

Both Main and Nicou start synchronized. They have a tag called *lswN* for the Main-Branch and *lswM* for the Nicou-Branch. Now both work in their respective Branch and check in their changes, and we get the following picture:

a b' c d	a b' c d		a b c' e	a b c' e

The tag is still with the first version, that is "a b c". Now let's get the changes from Main to Nicou:

```
SRadio.Nicou> Conventions/synch
```

By taking the difference between *lswN* and what is actually stored in the Main-Branch, we get this:

a b' c d	a b' c d		a b c' e	a b' c' d e

Now it is very important not to commit these changes, because first we need to take everything that has been changed in the Nicou-Branch to the Main-Branch:

```
SRadio.Main> Conventions/synch
```

And we get:

a b' c d	a b' c' d e		a b c' e	a b' c' d e

Now we can commit on both sides, writing the changes from 'local' to the 'server'. Then, we update the tags:

```
SRadio.Nicou> Conventions/tag
```

and finally we get this picture:

Now we can start again with changing in both branches.

30.2 Starting a new Branch

There are two things to do when starting a new branch: first the new branch has to be created, then the appropriate tags have to be written to the cvs-tree.

Let's say we want to create a new branch called 'Brian'. For the first part, it is enough to write in a checked-out Main-branch the following line:

```
SRadio.Main> cvs tag -b Branch_Brian
```

Then we have to make sure that everything is correctly tagged:

```
SRadio.Main> Conventions/tag Brian
```

This is all that is needed. Now you can proceed to 30.3 to see how to set up the system for the new user.

a b’ c’ d e	a b’ c’ d e		a b’ c’ d e	a b’ c’ d e
tag: lswN			tag: lswM	

30.3 Checking Out for the First Time

It is a good idea to have the following set-up if you’re not familiar with CVS[1]:

```
brian@radio1: > echo "export CVS_RSH=ssh" >>  /.bashrc
brian@radio1: > echo -e "cvs -q\nup -dP \ncommit -m ’’" > /.cvsrc
```

Then you have to log out and log in again, so that these parameters are available to your bash-shell. These commands help in everyday cvs. It might also be more comfortable to run the tool SRadio.Brian/Conventions/lussh and follow it’s directions to create a password-less login to user ’sradio’ on lcmpc10.epfl.ch.

Now we can actually check-out a version of this new branch. For this, change into the home-directory of brian, and write this:

```
brian@radio1: > cvs -d sradio@lcmpc10.epfl.ch:/home/sradio/cvs co -r
Branch_Brian SRadio
brian@radio1: > mv SRadio SRadio.Brian
```

This gives you a check-out of the branch Brian in the directory SRadio.Brian. All changes that are done in this directory are kept seperate from the other parts of the software-radio, so you can commit and update at will, without disturbing other developers.

Chapter 31

Creating a simple radio

In this chapter you'll learn how to create a very simple radio consisting of a transmitter and a receiver. You should know the very basics of a digital transmission. For a short introduction in this matter, you can also go and read chapter 19.

In figure 31.1 you see the goal of this tutorial: two radios, called *Master* and *Client* with the following functions:

- Master: send the synchronisation, training-sequence and an image
- Client: synchronize, do a channel estimation and decode the image

Both the master and the client can be represented by a simple chain. So it will be very easy to implement these two radios as shown in the rest of this tutorial.

After explaining some more general things about the software-radio, we'll explain first the master, then the client. You can find the basic files with all the basic setup already done in the sub-directory *Radios/ICS_Simple* of the SRadio-project. *Master/radio_master.c* contais the basic setup of the master and *Client/radio_client.c* the basic setup of the client.

31.1 General Setup

This section is just a very brief introduction to the software-radio. For a more complete overview, follow the references. First some overview of the software-radio, then an explanation of the C-files.

31.1.1 Overview

As explained in chapter 11, the software-radio can be run either in simulation- or in real-time-mode. While it is possible to have both the master and the client on the same computer in simulation mode, this is not possible in real-time mode. The first part of this HOWTO works only in simulation-mode, so you need neither hardware nor two computers to run the examples.

In simulation mode, the master and the client connect to a *channel-server* which simulates a simple wireless connection with some noise and a fading-channel. Both the master and the client run independently of each other and the only connection is the channel-server. Each one has a complete software-radio environment with a CDB and a SDB (see also 8.2.2), so that they can communicate only through the channel-server. This corresponds to the real transmission where the two radios can only communicate through the channel.

31.1.2 Files

Because the master and the client are independent entities, they are put in a seperate directory. In each directory a small template-file can be found that contains the initialisations necessary for the software-radio. For this tutorial, two functions are important:

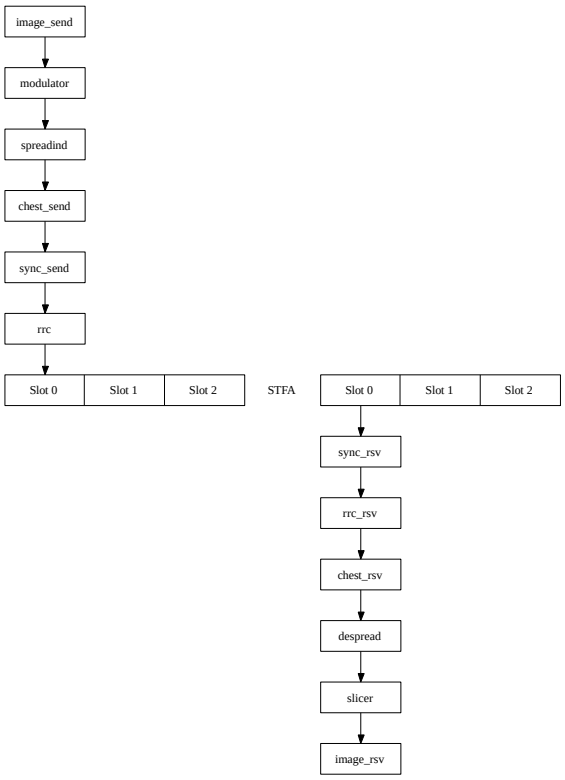


Figure 31.1: A very simple radio

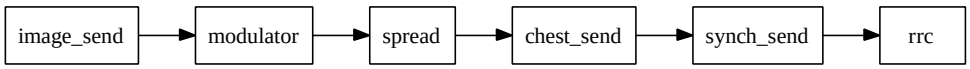


Figure 31.2:

- start_it comparable to the *main*-function in C-programs or the *constructor* in a C++-class
- um_module_exit which corresponds to the *destructor*-function in a C++-class.

In the start_it-function the radio initialises some modules and/or chains (see 8.2.1) which are cleaned-up in the um_module_exit-function.

31.2 Master

The master needs to send an image, encode it using a spreading-sequence, add a trainig-sequence for the channel estimation and put a synchronisation-signal on top of this, so that the client can synchronise. Looking at figure 31.1, we can directly deduce the necessary chain:

The most simplest way to do that is to create a chain with all the modules inside. In order to do that, open the file for the master in *Radios/ICS_Simple/Master/radio_master.c* and put the following text in the function *start_it*:

```
swr_sdb_id img, spread;
PR( "Setting up send-chain\n" );
send_ch = swr_chain_create(
  NEW_SPC_VAR( "image_send", img ),
  NEW_SPC( "mapper" ),
  NEW_SPC_VAR( "spread", spread ),
  NEW_SPC( "chest_send" ),
  NEW_SPC( "synch_send" ),
  NEW_SPC( "rrc_complex_send" ),
  NEW_SPC_VAR( "stfa_ics", stfa ),
  CHAIN_END );
```

For more explanation on the NEW_* commands, see 16.2.1.1. In short we use *NEW_SPC* to instantiate a new module and *NEW_SPC_VAR* to instantiate a module and get a reference-id back. Now that this chain is created, we need to tell the STFA[1] that this is a transmission-chain. We use the following command:

```
swr_stfa_notice_sdb( stfa, 0, img );
```

Now everytime the STFA wants to send slot 0, it will call the img-instantiation of the ”image_send”-module. Finally we set up the STFA in transmission-mode and tell the spread-module to use a spreading-factor of two[2]:

```
swr_sdb_set_config_int( stfa, "tx", 1 );
swr_sdb_set_config_int( spread, "factor", 2 );
```

Now we’re ready to start the STFA:

[1] read 8.2.4 for more information
[2] which is equivalent to a simple repetition code

```
  swr_stfa_go( stfa );
```

That's it for the start_it function. Because we use the *send_ch* and the *stfa* globally, we need to define these above the start_it-function like this:

```
struct chain_t *send_ch;
swr_sdb_id stfa;
```

And, last but not least, we need to clean up the chain in case the radio gets stopped. This is done by inserting the following lines at the end of the *um_module_exit*-function:

```
  if ( stfa ){
    PR( "Stopping the STFA$\backslash$n" );
    swr_stfa_stop( stfa );
    PR( "Deleting Sending-chain$\backslash$n" );
    swr_chain_destroy( send );
  }
```

31.2.1 Testing the Master

Now that the master is written, we can test it. First of all, make sure that you can compile it, using

```
make
```

and that there are neither warnings nor errors. Then you can call

```
make show_local
```

and if everything is OK, you should see a window coming up that displays the STFA as a horizontal white bar and the chain attached to it. Right-clicking on the *image_send*-module you can chose *Display Data* and then *picture* which will pop up a window with the picture sent in it. Keeping this window and right-clicking on *rrc_complex_send*, then *Display outputs* and *port_out_0* shows the output of this chain in complex format. In this new window you can click on *Complex* and chose *FFT* to see a fast-fourier transform of the same output. You can also change the behaviour of some of the modules. For example you can change the *factor*[3] of the *spread*-module. Right-click on the *spread*-module and chose *Change config*. Now you can increase or decrease the *factor*-value. If you still have the image-window open, you can see how the image changes when increasing or decreasing the *factor*-value. The next subsection explains why this happens.

31.2.2 Slots and Blocks in the Software-Radio

This is a small excurse in the internals of the software-radio. It's goal is to give you some more understanding of how our implementation works. It is not elemental but quite useful to understand the rest of the software-radio.

[3] this is the spreading-factor or the spreading-length of the code

The software-radio works with fixed blocks of data. Everything that goes over the air has to fit into one slot. In the current implementation, a slot is of length 2560 symbols[4]. Every time the software-radio wants to send a slot, the output of the chain has to fit in 2560 symbols.

Starting from the STFA, the software-radio calculates the maximum size available by asking each module how much symbols it needs:

$\frac{1}{n}\ \frac{2078}{n} * 2\ \frac{4156}{n}$

module	use	symbols	left
stfa	guard-period	90	2470
synch_send	synchronisation	256	2214
chest_send	training-sequence	136	2078
spread	spreading-sequence		
mapper	bit-to-symbols		bits

This means that the image_send-module has a variable-sized output that can vary from 4156 to $\frac{4156}{32} = 129$ bits[5], depending on the spreading-factor chosen. Every time the output-size of the image_send-module changes, it adjusts the image-size so that it fits in the given place.

For this reason you see the image changing when you change the spreading-factor of the spreading-module.

31.3 Client

Looking at figure 31.1 we can see that we have to implement the following chain:

There are two small subtelties when implementing this receiver-chain: first, we put the *synch_rcv*-module in front of the *rrc_rcv*-module. Looking at the master, we'd expect to first do the rrc-reception and then the synchronisation. But as the rrc_rcv-module downsamples by a factor of two, it is better to do the synchronisation at the higher sample-rate, in order to have a more exact synchronisation.

Second, the actual module-names are quite long: sync_rcv is called *synch_complex_ics_rcv* and rrc_rcv is called *rrc_complex_ics_rcv*. This is due to the development-process of the software-radio and the wish to keep old things running.

Besides these small details, the client is built more or less in the same way as the server. All these lines go in *Radios/ICS_Simple/Client/radio_client.c*. Before the *start_it*-function, we declare the *stfa* and the *rcv_ch*:

```
swr_sdb_id stfa;
struct chain_t *rcv_ch;
```

In the *start_it*-function, we declare the needed variables and the reception-chain:

```
  swr_sdb_id synch, mafi, despread, slicer;
  rcv_ch = swr_chain_create( NEW_SPC_VAR( "stfa_ics", stfa ),
    NEW_SPC_VAR( "synch_complex_ics_rcv", synch ),
```

[4] this is an artefact of the first UMTS-implementation. You can change the slot-length in multiples of 128 symbols

[5] The spreading-module has a maximum spreading-length of 32

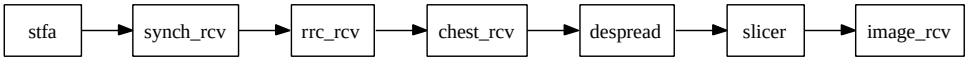


Figure 31.3:

```
  NEW_SPC( "rrc_complex_ics_rcv" ),
  NEW_SPC_VAR( "chest_rcv", mafi ),
  NEW_SPC_VAR( "despread", despread ),
  NEW_SPC_VAR( "slicer", slicer ),
  NEW_SPC( "image_rcv" ),
  CHAIN_END );
```

This chain is the same as in figure 31.1 and described above. Some of the modules need some configuration so that the radio works correctly. We have to do these things:

- Gain: adjust the gain of the RF-cards, so that the cards don't saturate
- Synchronisation: tell the module the id of the STFA so it can adjust for the desynchronisation
- Despreading: set the despreading-factor to 2
- Matched Filter: calculate 8 taps and align them
- Slicer: set the id of the matched filter so that it can know the amplitude of the signal

All these values are in the configurable part of the modules, so we can set them using the swr_sdb_set_config*-functions:

```
swr_sdb_set_config_int( stfa, "attn_tx", 31 );
swr_sdb_set_config_int( stfa, "attn_rx", 15 );
swr_sdb_set_config_int( synch, "stfa_id", stfa );
swr_sdb_set_config_int( despread, "factor00", 2 );
swr_sdb_set_config_int( mafi, "calc_taps", 8 );
swr_sdb_set_config_int( mafi, "align", 1 );
swr_sdb_set_config_int( slicer, "mafi_id", mafi );
```

Now everything is ready to start the STFA:

```
swr_stfa_go( stfa );
```

We need to do some initialisation, just in case something will go wrong. Insert this line in the beginning of the function *um_module_init*:

```
stfa = 0;
```

And at the end we need to clean things up, so insert these lines at the end of *um_module_exit*:

```
if ( stfa ){
  PR( "Stopping stfa$\backslash$n" );
  swr_stfa_stop( stfa );
  PR( "Destroying rcv-chain$\backslash$n" );
  swr_chain_destroy( rcv_ch );
}
```

31.3.1 Testing the Client

Compile it with *make* and correct eventual errors. Once everything compiles nicely, you can go in the directory *Radios/ICS_Simple* and type

```
make show_mc
```

This will start the master, then the client and will bring up the visualize-tool so that you can see the modules in action. Once the visualize-tool is started, you can see that there are now two tabs, one for the master and the other for the client. Clicking on these tabs you can switch between the two radios.

31.3.2 Testing the transmission

If everything up to here works as described, congratulations. You just finished your first very simple radio-transmission using the software-radio. While running the master and the client, you can now visualize different modules and change the configuration-parameters, in order to see what happens.

31.3.3 Synchronisation

A small word on synchronisation: the implementation we did in this tutorial is the most simplest possibility. Every frame the synchronisation-module will search for a synchronisation-signal. If it decides that there is no synchronisation, it shifts half a slot and will try again in the next frame.

The worst-case scenario is that the synchronisation-signal is just on the opposite side of the searching-direction, which will make the synchronisation-module searching the whole frame. In real time, one frame corresponds to 30ms and holds 15 slots, so that it will be searched completely in 1sec.

For a radio-application, 1 second is quite a long time. For this reason, a second method exists which is a bit more complicated but finds the synchronisation-signal on the first go. A *macro-module* called *macro_synch_ics* attaches a synchronisation-module to each slot of the stfa. Once a whole frame is received, it loops over all synchronisation-modules and choses the one with the strongest synchronisation-signal, discarding all others.

This allows the software-radio to synchronise in much more efficient way. As this method includes callback-functions and handling the macro-module, we decided to go the easy way and just scan the whole frame.

31.4 RF-transmission

Now everything should be ready to be able to transmit over the air. Once the simulation works, there are usually no big bugs left that can hinder the test over the air.

As you will have to transmit from one computer to the other, the code needs to be installed on two computers where each computer has the appropriate hardware installed. Once this is done, it is usually more efficient to work on one computer only and to *ssh* in the other computer and run the software-radio remotely. This gives you the advantage of having all the output on one screen.

To start the master, simply go into the *Radios/ICS_Simple/Master*-directory and call

```
make rf_show
```

which should start the real-time part of the software-radio and show up the transmission-window. On the other computer, go into the *Radios/ICS_Simple/Client* directory and issue the same command:

```
make rf_show
```

If all goes well, you should have now two windows where one shows the master and the other the client. You can now play around with the values to see how the software-radio reacts. Interesting configuration-parameters include the attn_tx on the master-side, the attn_rx on the client side. Both can be found in the stfa, the horizontal white line in the middle of the window.

Other things to experiment with is the calc_taps of the chest_rcv-module or the factor and sequence configuration of the spread and despread-module.

31.4.1 Going Further

Now that you have this simple radio running, it is possible to change the spreading-module with something else, for example a convolution-module or even an ldpc-module. You can also add other chains to the stfa in a similar manner than the ones we did. If you do so, don't forget that you don't need a synchronisation-module anymore on the additional chains. One synchronisation-module per frame is enough.

Part V

Practice

Chapter 32

Introduction

This document started out as a work-description on what has been going on with the software-radio. It now is a reference for the RF-interface of the software-radio as well as a setup-guide for the hardware-part of the software-radio. For the software-part, see the document ”A tutorial to the software-radio”.

32.1 Motivation

During Summer/Fall 2004 different projects have been done on the software-radio that asked for special implementations on the level of the ICS-cards. Four useful branches co-existed:

1. Normal MIMO at 2.4GHz with 1MS/s
2. Radar-application at 2.4GHz with 100kS/s
3. 802.11-reception at 2.4GHz with 20MS/s
4. GPS-reception at IF of 24MHz and 4MS/s at 32 bits per complex sample

1-3 were done using the normal FPGA-programm, while 4 is only possible with a rewritten FPGA. This is due to the fact that the PCI can’t transfer the data representing a raw GPS-signal. The rewritten FPGA-program represents one sample using 32 bits, whereas the original FPGA-program represents one sample using 128 bits.

In order to have a program that can handle all these situations, a new interface to the ICS-cards was built. The goal of this interface is to set the basic parameters of the ICS-cards in a more user-friendly way. These parameters are:

- Sampling-frequency
- Bandwith
- Data-type
- Number of channels
- Center-frequency
- Detect the FPGA-programm type

All frequencies are given in Hz and the sampling-rates in Complex Samles per second (CSs). 1 CS corresponds to 2 Real Samples.

32.2 Intended reader

This document has been written to keep track of the changes in the programming and to help further engineers keeping track of what is going on. Depending on which part of the document you're interested in, the necessary background differs. In order to build the different target applications, you should have good knowledge in informatics and a good understanding of the UNIX-shell. For the reference-part you have to be a good programmer and have some background of the software-radio. A good start is "A tutorial on the software-radio".

32.3 Parts

The different parts in this document include:

- Motivation what you are reading
- Usecases describes the four goals and how to test them
- Subsystems which gives an overview of the different parts of the software-radio hardware
- Interface is the main part and describes in detail the new functions used to interact with the hardware
- Tidbits collects different thoughts about the project that came up during the writing of this document

32.4 Conventions

32.4.1 Directories

All given directories are relative to your SRadio.*Branch*-directory. The position of this branch is marked *$SRADIO*. This has to be installed for you by the system administrator, as well as a CVS[1]-access to the source-code. So if your system-administrator installed a branch under

```
/home/foo/SRadio.Bar
```

and a command asks you to

```
cd $SRADIO/Test/Radar
```

Then this means you have to enter the following directory:

```
/home/foo/SRadio.Bar/Test/Radar
```

32.4.2 Commands

Commands are written in typewriter-code, like this:

```
cvs -d sradio@lcmpc10.epfl.ch:/home/sradio/cvs \
  co -b Branch_Report SRadio
```

Special meanings are explained in brackets, for example:

```
make rf_tail
[ wait for 5 seconds ]
make rmall
```

[1] Concurrent Versions System

32.4.3 Radio-platforms

There are two radio-platforms, called *radio1* and *radio2*. The work represented in this document requires root-privileges to be run, so ask your system-administrator for the root-password and how to log in.

If you are in the local network of the software-radio, you can login to *radio1* and *radio2* with

```
ssh root@radio1
```

or

```
ssh root@radio2
```

Chapter 33

Test Configurations

This section describes the sample configurations. Both soft- and hardware-setup is described for each configuration. It is a good starting point for new users.

The following configurations are described here:

- GPS for the reception and storage of a GPS-signal
- Radio demonstrates a simple send/receive-setup
- Radar is a simple game using radar-measurements of a moving ball
- WLAN captures one or more 802.11-packets

33.1 Setup

This subsection contains some general overview of the different hardware-subsystems.

33.1.1 RF-cards

Fig. 33.1 shows the input/output connectors of the RF cards (power supply omitted).

33.1.2 Rohde & Schwarz

All settings given in the figures start from a presetted state of the signal-generator or -analyzer. You find a button labeled *Preset* that sets the machine into a well-known state.

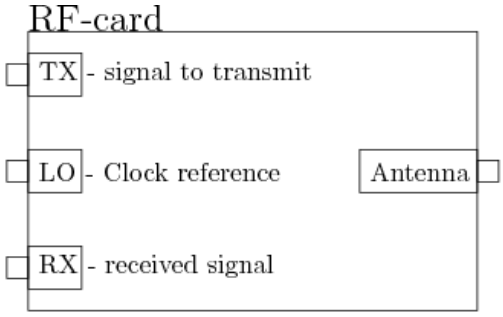


Figure 33.1: The important connections on a RF-card

33.1.3 ICS-cards

In the following picture (TODO) you can see the ICS-cards on the back of the computer. All the input and output-labels are visible.

33.1.4 FPGA

As described in section 23, the FPGA can be programmed with two versions of the code: one for the GPS-reception, and another one for everything else. Please be sure to chose the appropriate code, or the given application will not work.

Refer to section 23 on how to reprogram the PFGA.

33.1.5 Power supply

Wrong settings of the power-supply may destroy the attached hardware, so take care and be sure to follow these instructions:

1. Switch up 1 & 2
2. Connect the power supply
3. Switch down 1
4. Adjust voltage and current using 3 & 4
5. Switch down 2

33.2 GPS

For a clean reception of a GPS signal, it is advised to have a view as wide as possible of the open sky. Chose a roof-top or a wide field with no obstruction. Then you can start with the reception of a 60-second piece of data:

Before doing so, test this given setup with a much shorter sequence, 2 seconds, as described below. Once everything is working all right, use the longer sequence.

33.2.1 Hardware-setup

The setup of the hardware for the GPS-acquisition can be seen in Figure **??**. If you extend the cable of the GPS-antenna, the received signal may be too weak!

33.2.2 Software-setup

First you have to re-program the FPGA, so that the correct code is loaded in the ICS-card. Refer to Section 23 on how to do this. For a first test, it is advised to use only a short testing sequence, see the paragraph below. This short sequence can be used to test whether enough satellites are visible. Once the FPGA is repgrammed do the following:

```
cd
<latex>SRADIO/Test/GPS
make rf_tail

[ wait for the message "*** Acquisition done ***" ]

../../Tools/Dbg/dbg 2 0 > ../../Matlab/GPS/gps_yymmdd_hhmm.dat
```

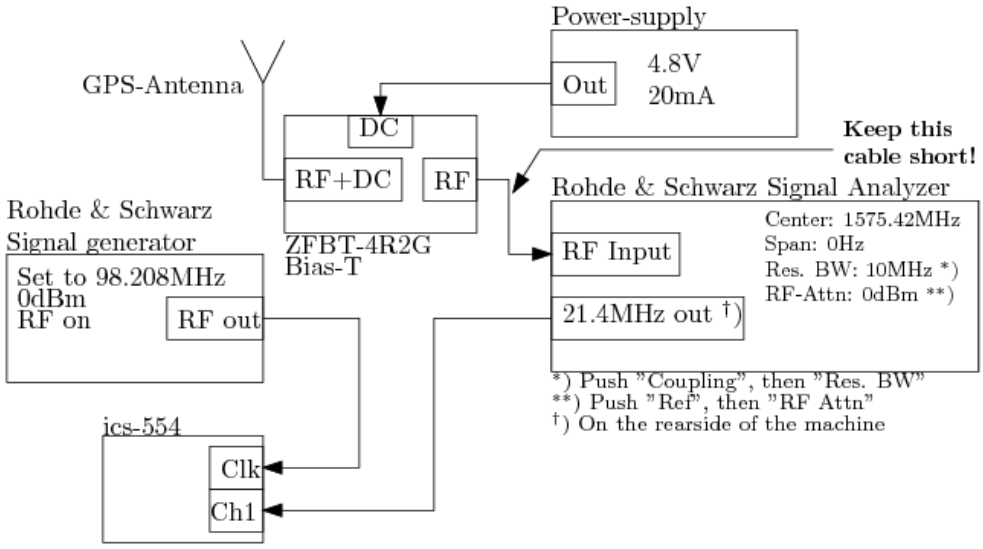


Figure 33.2:

Where *yymmdd_hhmm* stands for the current date and time. This is just a convenient way to remember when the samples were collected. Be aware that each second of measurement takes about 25MBytes. So 60 seconds of measurement take 1.5GByte!

Then you can check which satellites are visible from the antenna location using using the program in *\$SRADIO/Matlab/GPS/*:

```
matlab -nojvm -nosplash
[ wait for Matlab to start up ]
acquisition( 'gps_yymmdd_hhmm.dat', 100000 );
```

this should show you a plot with a noise-floor around 10000 and a number of points sticking distinctively out to $14000 - 20000$. Then chose the highest point and run the following command:

```
acquisition_scan( 'gps_yymmdd_hhmm.dat', 100000, #sat# );
```

where *sat* corresponds to the highest peak seen in the previous plot.

33.2.2.1 Short testing sequence

Before getting a 60-second piece of data, it is adviseable to test it on a short sequence to see whether enough satellites are visible or if there is a problem with the setup.

In the file *\$SRADIO/Test/GPS/test_gps.c* adjust the *buffer_len*:

```
// Now buffer_len is in seconds of recording
swr_sdb_set_config_int( gps, "buffer_len", #2# );
```

Now the radio will only record 2 seconds of data which will be much shorter to write to the disk using the above-mentioned method.

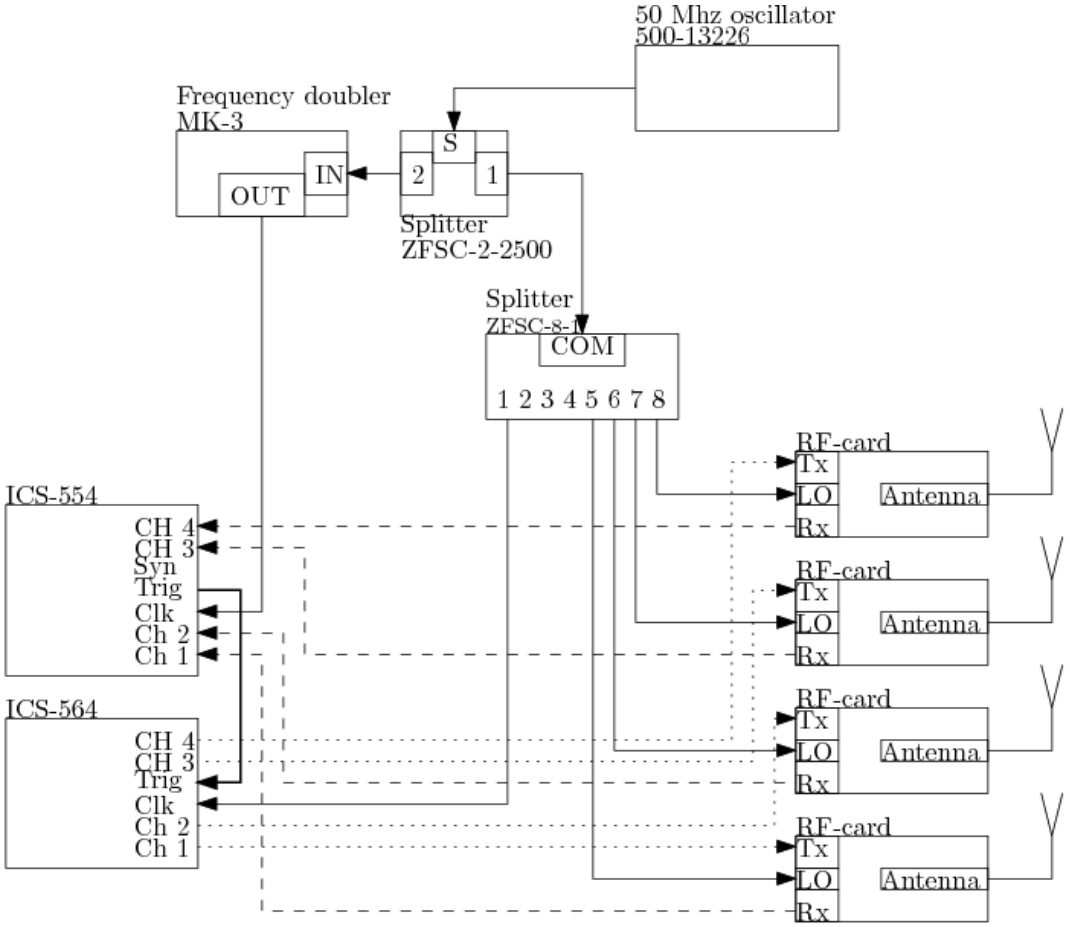


Figure 33.3: Radio-setup

33.3 Radio system

The first goal of the software-radio was to enable a two-way transceiver on a 2.4GHz link. Use the setup described here:

33.3.1 Hardware-setup

For a radio transmission, the setup in Figure 33.3 must be done on both radios. It is necessary to have frequency doubler exactly as shown in the figure and not attached to the output of the 8-way splitter, as one may try to do to get rid of the 2-way splitter. Otherwise the clock-signal for the ICS-554 card would be too weak.

33.3.2 Software-setup

In order to be sure that the STD-software is loaded in the FPGA, process as described in section 23. If that is not the case, reprogram the FPGA with the STD-software.

The software needs to be run on both radios, *radio1* and *radio2*. On one radio you have to run the sender (BS), on the other radio the receiver (MS[1]). To start the sender, run the following commands:

[1] Microsoft

```
cd $SRADIO/Radios/Image/BS

make rf_show
```

For the receiver you have to log in to the other radio and run the following commands:

```
cd $SRADIO/Radios/Image/MS

make rf_show
```

on the receiver side. If it doesn't work, go through the following checklist:

- all cables correctly connected
- power is applied to all elements (50MHz clock and RF-cards)
- RF-cards are connected with the right flatband-connector
- Using the Rohde \& Schwarz signal analyzer to check for a singal at 2.38GHz

33.4 Radar-system

The radar-system uses only one radio-platform, and the setup is similar to 33.3.

33.4.1 Hardware-setup

For this setup, the duplicator is not needed, and only two RF-cards are necessary. Align the two antennas, so that they point in the direction of the object. If possible, the player should be behind the antennas, so that his movement is hidden to the radar. Else it is difficult to tell apart the movement of the player with the movement of the object.

33.4.2 Software-setup

Be sure to have the correct code loaded into the FPGA, else refer to 23 on how to program the correct code.
When everything is set up, you can start the radio on the software-platform:

```
cd \$SRADIO/Test/Radar
make rf_tail
```

Then, on a computer connected to the software-platform, connect the camera, make sure it is detected, and run:

```
cd \$SRADIO/Tools/PlayRadar
./run
```

Eventually you have to adjust the following two variables:

REMOTE points to the software-radio platform

RADIO_DIR is the remote directory

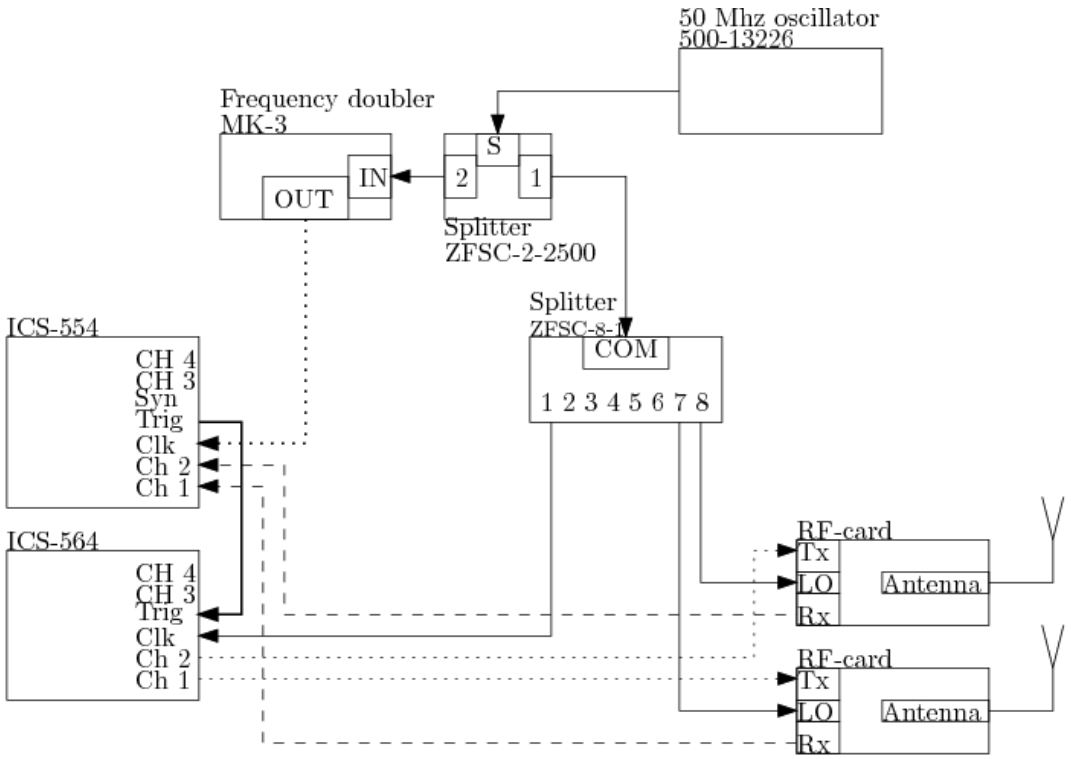


Figure 33.4: Radar-system

33.4.3 Camera-setup

For this radar-system you can have a camera attached to the computer, so that the different people using the system can be shown live! Unfortunatly, this requires some setup to your computer. The source code for the driver can be found under:

```
http://www.saillard.org/linux/pwc/debian/
```

and once it is installed, things should run nicely.

33.4.4 Amplitude settings

The amplitude settings are done in case the two antennas are 1m apart and on the same height, pointing in the same direction. If you chose to use them in a different setting, it can be necessary to adjust the values *attn_rx* and *attn_tx* in the file *\$SRADIO/Test/Radar/test_radar.c* to more convenient values. Lower values mean lower attenuations. So if you put the antennas further apart, it may be useful to chose a lower value for *attn_rx*.

You can also run

```
cd \$SRADIO/Test/Radar
make rf_show
```

and use the configuration-window from the *radar_rf*-module while looking at the output of *radar_rf* and the *fft*-stats of the *radar_fft*-module. Once you find a good pair of values where the output shows a dotted circle and the fft shows a clean peak, copy these values to *test_radar.c* and re-run it.

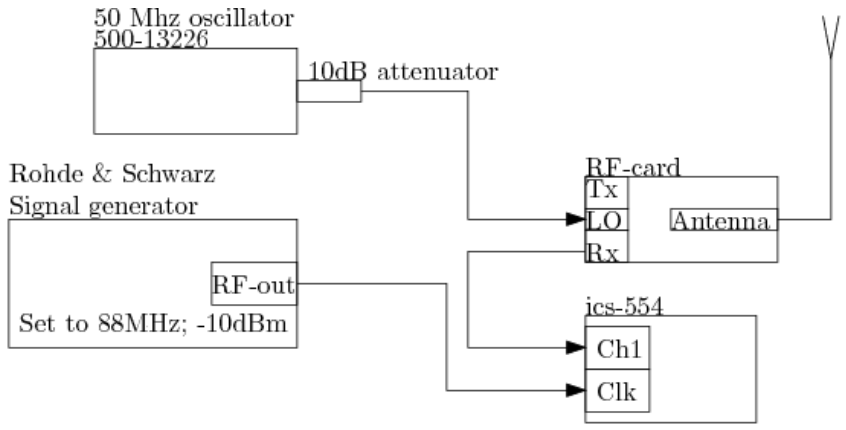


Figure 33.5: WLAN-setup

33.5 WLAN

The goal is to aquire a part of a WLAN-signal and to decode the most important bits.

33.5.1 Hardware-setup

For this test, take a wireless access point and set it to channel 6 at a center frequency of 2437MHz. This takes care of EPFLs network that is tuned to channel 1 and 12 at 2412MHz and 2467MHz. The bandwith of the signal is 22MHz.

The setup can be seen in fig. 33.5.

33.5.2 Software-setup

If you used the GPS beforehand, you have to re-program the FPGA first. For this, refer to 23. Once this is done, you can start the aquisition:

```
cd \$SRADIO/Test/WLAN
make rf_tail
../../Tools/Dbg/dbg 2 0 > ../../Matlab/WLAN/wlan_capture.dat
```

Then you can use the programs in *\$SRADIO/Matlab/WLAN* to decode the received WLAN-signal.

Part VI

Future thoughts

Chapter 34

STFA

The STFA being a complex module and having evolved over two years, it would be a good idea to re-write it once completely. Here are some ideas on what should be mended and what thoughts should go into a re-structuration of the STFA.

34.1 Antenna

The antenna should be a module on its own with an output for the received samples and an input for the sent samples. These two ports should then be connected to the stfa. Advantages:

- Configuration If for a given antenna we have special parameters, we can define them as easily as in any module
- Clarification It gives a more constant concept in the software-radio
- Universality For the moment the software-radio is very TDD-based with the STFA. If the antenna is really an independant module, it would be much more easy to implement other stuff

34.2 Chains

The way chains are linked to the STFA is very kludgy. While things are quite OK for RX-chains, it is really not optimal for TX-chains. Consider figure 34.1 where on the left-hand side you see the actual implementation. The user has to define which modules have to be called at which moment with two function-calls like:

```
swr_stfa_notice_sdb( stfa, 2, source1 );
swr_stfa_notice_sdb( stfa, 5, source2 );
```

In order to simplify things and thus taking away possible sources of error, it would be much better, if the STFA would consider the chains on its own, like show in the right-hand part of the figure.

For the TX1-chain, not much would change, only the two above-mentioned function-calls could be left out, because the STFA would call the RRC-module with a message like *SUBS_MSG_PRODUCE_DATA*. The RRC-module in turn would call the Map-module, which would call the Source1-module. Then, the Source1-module produces some data, and everything is like before.

img /var/www/html/ipgwww/data/media/stfa_tx_kludge.ps

Figure 34.1: Actual and proposed design of the TX-chain alertion

For the TX2-chain, the distribution of the time of calculation would even be more uniform. If you consider the left-hand side, all the calculations for all the modules is done during slot 3 and 4, even though about half of the calculations are not needed before slot 7! Now, if the STFA would be smart enough to not only call the modules with a message *SUBS_MSG_PRODUCE_DATA*, but also tag the modules passed this way, only the necessary calculations would be carried out.

So, at the beginning of slot 3, the RRC would be notified with *SUBS_MSG_PRODUCE_DATA* and tagged active. Then the Map-, Split-, and finally the Source2-module would be notified. Once the Source2-module produced its data, the inverse way will be taken. An important point is to note that the Split-module produces both branches of data, but that only the left branch is followed.

Then, at the beginning of slot 5 [1], the RRC of slot 7 would be notified, which would notify the Map-module, which would see that it has some data to process, and stop the notification of further modules. It then processes the data, hands it to the RRC, which processes its data, too, and hands it to the STFA.

34.2.1 Implementation

In fact, it is not really necessary to tag the data. Each module could just

- Check inputs for whether all necessary inputs are filled with data
- Notify upper modules if this is not the case
- Process the data

The idea is that the notification of the upper modules happens synchronously, that is, the notification only returns once it has been carried out. The module can then assume that the data is available.

There are a lot of special cases, just a few here:

- Multiple inputs of a module, where one or more may be inactive, that is, not connected
- Loops for creating incremental coding

[1] Coincidence: this is the same slot as the result of the left branch

Chapter 35

Visualize

Reading through the reference-part of the visualize-tool, one gets the impression that this tool has been glued together (like oh so many others) more randomly than anything else. Well, unfortunatly, this is true. One project would be to re-design the classes of this tool, and then re-write a more nice version of it.

The steps to do so could include the following:

- Understand and look at the existing code. Using a tool like Umbrello, one can get quite fast a basic understanding of which classes use which classes.
- Design a new class-model, where each class has a well-designed purpose.
- Rewrite the whole tool, doing mostly copy-paste constructs. [1]

[1] Important: go step-by-step with easy testable parts

Part VII

Index

Index

List of Figures