Modular Software Radio

L.Gasser, EPFL

14th April 2004

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.**??** 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.

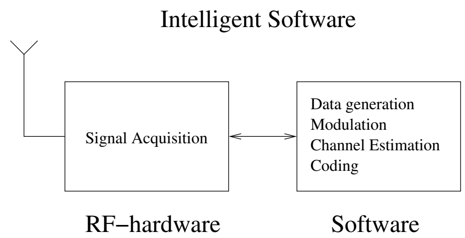

Figure 2.1: Dumb hardware and intelligent software

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) 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.

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.

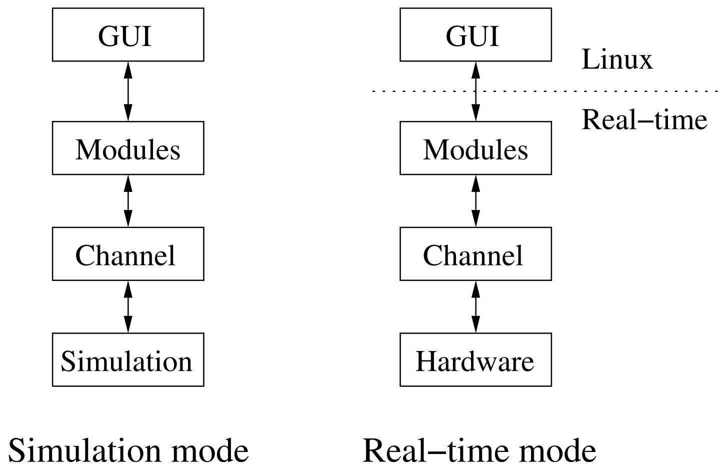

Figure 3.1: Structure of the software-radio in both modes

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 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.

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.

A general view can be seen in fig. @overview GUI/Signal Processing/Antenna@. For each of these elements, you can find a chapter that describes it in more detail. The list of chapters include also more general descriptions of the software-radio:

- GUI 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

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 fig. @blow-up of GUI:User IO/Mapper/FifoCmd@, we distinguish three main-parts:

- User IO 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 IO

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

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.

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 displaying them correctly.

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 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
- 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

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 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

8.2.1 Modules

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. 8.1.

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.

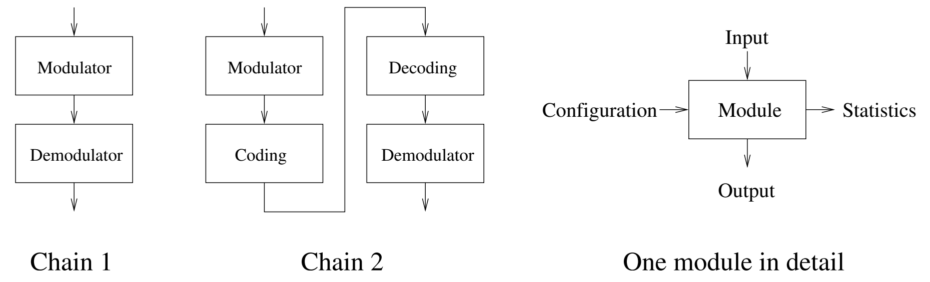

Figure 8.1: Two simple chains and a module in detail

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. 8.2. It shows part of a running software-radio that has two transmitting and two receiving slots. [1] 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 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.

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

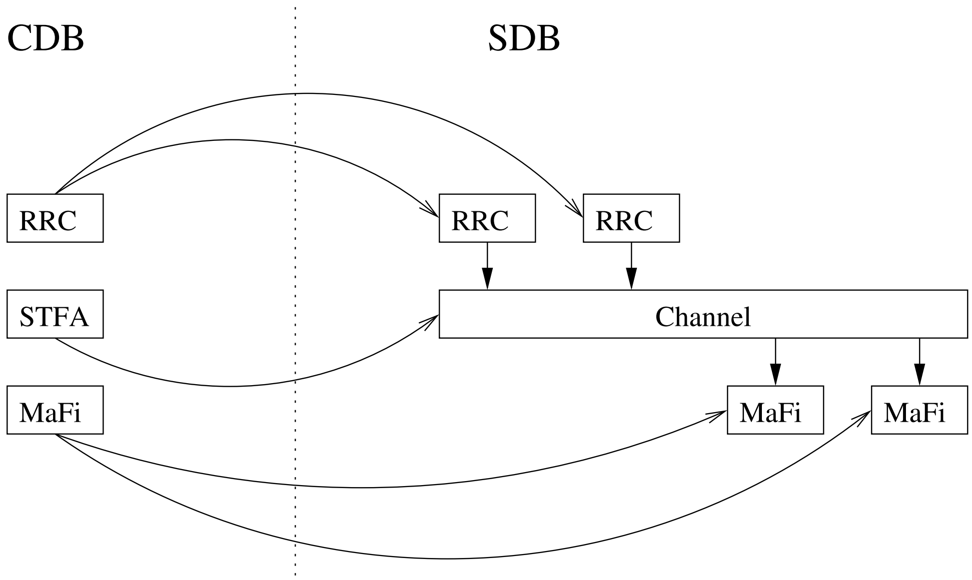

Figure 8.2: The CDB and SDB

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]	1.9	2.4-2.48
Bandwith [MHz]	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 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.

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.

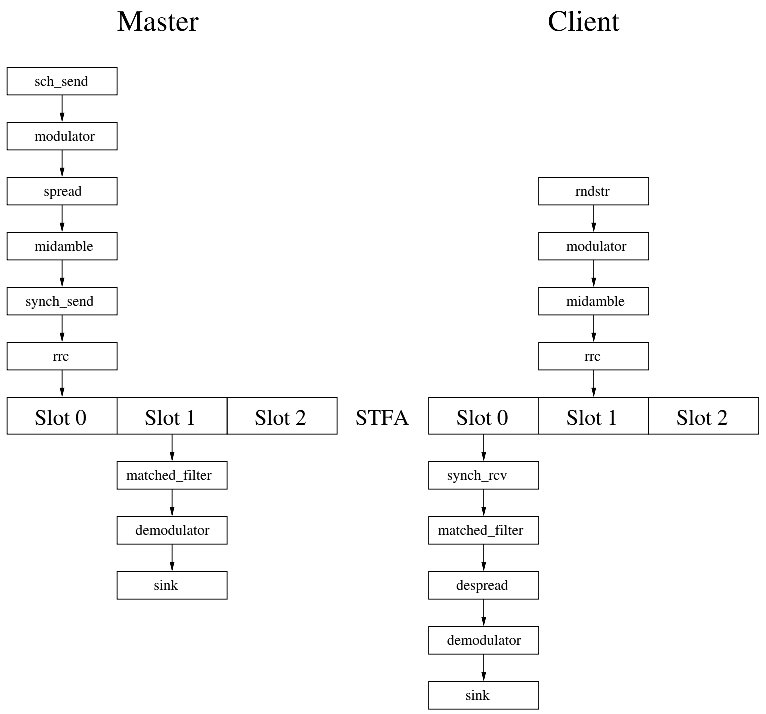

Figure 11.1: The most simple two-way communication example

When running in 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*. 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[1]. The communication channel in this example consists of three slots, of which only two are occupied[2]. 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 **??**, 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:

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

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

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.

midamble 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.

matched_filter The counterpart to *midamble*, 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 4x4 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 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

Chapter 15

Abbreviations and terms used

CDB Class DataBase

MSR Modular Software Radio

RRC Root Raised Cosine

SDB Subsystem DataBase

STFA Slot To Frame Allocation, basic slot-based frame structure for interfacing with the RF-hardware

Chapter 16

GUI

16.1 User Interfaces

16.1.1 Plots

16.1.1.1 Plot XY

16.1.1.2 Plot X(t)

16.1.1.3 Signal

16.2 Mapper

16.3 Modules

Chapter 17

Signal Processing

17.1 CDB

17.2 SDB

17.3 Subsystem

17.4 Module

This section gives an overview of the module-creation and the use of it. Even though **??** 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.

17.4.1 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.

17.4.2 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

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

before being able to use *config->something*. To free the structure, use

```
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.

17.4.3 Data Types

17.4.3.1 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

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

SYMBOL_COMPLEX_S32

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

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

$$Re_o \quad Im_0 \quad -Im_0 \quad Re_0$$

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:

$$Re_o \quad Im_0 \quad Re_0 \quad Im_0$$

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

17.4.4 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.

17.4.4.1 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_DOUBLE adds a double-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_DOUBLE adds a double-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 17.4.3.1

UM_INPUT adds an input-port, for the types see **??**, and allows to define a flag

UM_OUTPUT adds an output-port, for the types see **??**, and allows to define a flag

17.4.4.2 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

Chapter 18

Makefile

18.1 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.

18.1.1 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
cvs_commit	Commits *all* changes to the SRadio/*-tree

18.1.2 Radios

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

bsms	Starts channel-server and both BS and MS part. To stop, run *make kill*
show_bsms	Like *bsms*, but also runs the visualisation-tool

18.1.3 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
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

Chapter 19

DBG-interface

19.1 Command-syntax

19.1.1 list

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.

19.1.2 show

gives more detail

Arguments the value for the reference

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.

19.1.3 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.

19.1.4 get_config

Used to fill in the config-window

Arguments module_id,config_index

Returns value

19.1.5 set_config

Changes a configuration. Changes take effect as soon as possible

Arguments module_id,config_index,value

Returns OK or FAILED

19.1.6 get stats

Reads out a statistical value.

Arguments module_id,stats_index

Returns value. For complex values, this is a (real,imag) value, for blocks, it is handled in the same way as get_output, so it's size,type,values

Chapter 20

TODO

20.1 Tests

20.2 Radios

20.3 Visualize

20.4 Signal Flow

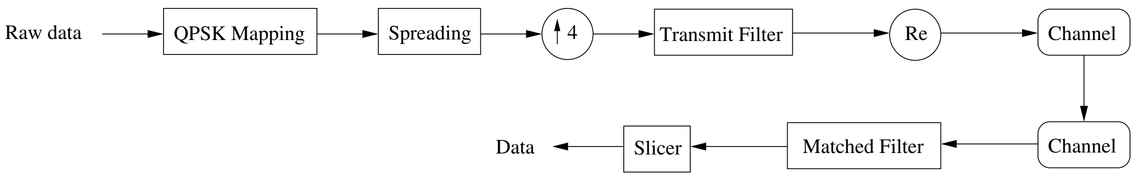

Figure 20.1: Signal Flow Block Diagram

(data->symbols->samples, frequencies and all) The raw data bits are grouped to form message symbols in the mapper. Each such symbol can be regarded as a member of a finite alphabet set containing M members. For every two raw data bits, we map them to a symbol in the QPSK signal space as shown in Figure 20.2.

The symbols are then spread by means of a spreading sequence which is independant of the data. The symbols are thus coded over more signaling coordinates. This operation involves multiplying each symbol with a predetermined sequence of ± 1. The length of the ± 1 sequence is the spreading factor(SF). So, when we are finished spreading, we have SF times the number of symbols we started out with.

Then a midamble (training-sequence) is inserted into the signal. This sequence is known from the receiver that uses it to estimate the channel and to create the matched-filter.

Next a synchronisation sequence is added to the signal. This sequence is formed such as to be extracted at the receiver end with as less calculations as possible. Next, we upsample the symbols by four and pass them through a transmit filter. A root-raised cosine filter is used, 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 final step before transmission is to take the real part of the transmit filter output, since we can only transmit real values through a real-life channel.

For an application involving RF transmission, the next important step is bandpass modulation; it is required whenever the transmission medium will not support the propagation of pulse-like waveforms. The

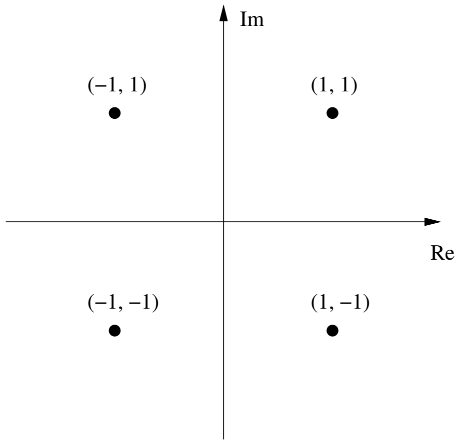

Figure 20.2: QPSK signal space

term bandpass is used to indicate that the baseband waveform is translated by a carrier wave to a frequency that is much larger than the spectral content waveform. The Figure 20.3 illustrates the upsampling, filtering and taking the real part operations.

At the receiver end, the demodulator provides frequency down-conversion for each bandpass waveform. Then the received samples are passed through a matched filter, which is matched to a cascade composed of the spreading sequence and the channel estimate. The matched filter produces points on the QPSK signal space in Figure 20.2. Due to channel effects, theses points will not likely lie exactly on one of the four signal points. Instead we will need to perform some sort of detection or demapping to guess which signal point was transmitted. This is done by the slicer, which output we want to be as close as possible to the raw data that we wanted to send at the beginning of the chain.

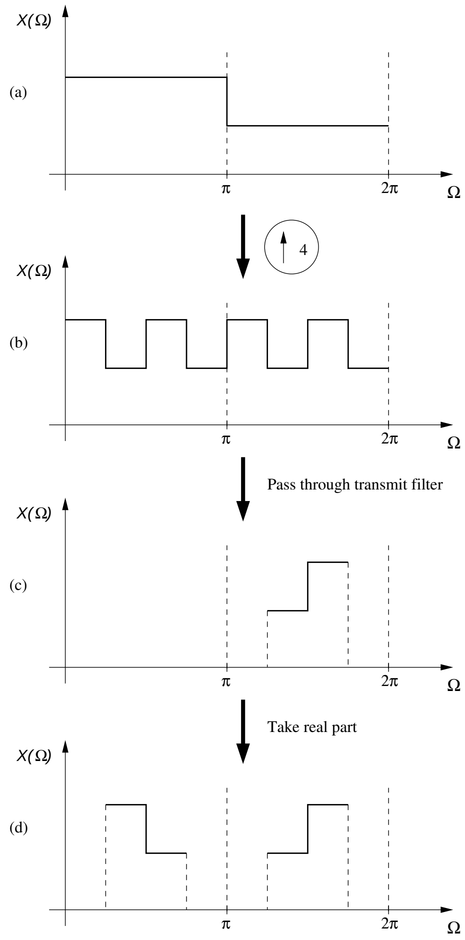

Figure 20.3: Illustration of the upsampling and bandpass conversion in the Frequency Domain

Chapter 21

Antenna

21.1 Driver

21.2 Channel-Server

Chapter 22

Hardware

Chapter 23

Modules

23.1 STFA

23.2 RRC

23.3 Midamble

Part IV

HOWTOs

Chapter 24

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 **??**. You should also already have run the example in chapter **??**. This part is a bit heavy on coding, but you won't be able to write modules without a good knowledge of C.

24.1 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. 24.1. 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 x, 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.

Random	Midamble	Random

Figure 24.1: The example slot

24.1.1 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
```

24.1.1.1 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.

24.1.1.2 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;
  // The QPSK-type, 0->in the corner, 1->on the axes
  int type;
} 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 **??** 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" );
```

24.1.1.3 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;
```

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

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

[1] For further information, look at **??**

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 x so that we can implement the formula given above. Instead of calculating x and then taking the sign of it, we directly calculate x 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" );
```

24.1.1.4 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
```

24.1.2 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.

24.2 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.

24.2.1 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
```

24.2.2 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 midamble rrc block
```

24.2.3 test_snr.c

Let's have a look at the documentation:

```
Make a simple chain:
random - snr_send - midamble - rrc - block -
         matched_filter - snr_rcv
and additionally:
random - snr_rcv(2)
```

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

24.2.3.1 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 **??**. 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( ''midamble'' ),
                        NEW_SPC( ''rrc'' ),
                        NEW_SPC_VAR( ''block'' ),
                        NEW_SPC_VAR( ''matched_filter'', mafi ),
                        NEW_SPC_VAR( ''snr_rcv'', snr_rcv ),
                        END_CHAIN );
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

midamble inserts the training-sequence in the middle of the stream

rrc Root Raised Cosine pulse-shape filtering

block a very simple channel-simulation

matched_filter 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*.

24.3 Going Over the Air

OK, now that the module is written, a simple test-case shows that our module works (NOT), 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.

24.3.1 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 midamble random \
          rndstr spread sink cch macro_sch snr
```

and in *MS/Makefile* reads:

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

24.3.2 README

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

24.3.3 MS/radio_ms.c

There is a lot of things to say about the basic system. You can find an introduction in **??**. 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( "midamble" ),
                       NEW_SPC( "rrc" ),
                       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 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.

24.3.4 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_VAR( "matched_filter", 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 );
```

24.3.5 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.

24.3.6 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* 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 25

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.

25.1 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.

25.1.1 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.

25.1.2 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.

25.1.3 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.

25.1.4 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

25.1.5 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.

25.2 Channel-server

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

25.2.1 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.

25.2.2 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*.

25.3 LDPC-code generation

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

25.3.1 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.

25.3.2 Known bugs

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

25.4 Output data-blocks

A small program called *convert* has the ability to read out any port or any stats-block that is defined in a running software-radio, be it a simulation or a real-time run. It does this by reading out the tree generated by the software-radio. This tool is now more or less obsolete, as the visualize-tool has the ability to export data-structures from any module.

25.4.1 Syntax

convert dir [-h] [-o file] [-p] [nr]

The *dir* parameter has to point to the appropriate directory, which is */proc/sradio/sdb* or */tmp/proc.n/sradio/sdb*. Upon running convert with only the directory, it outputs a list of available modules in this directory. You have to add the id of the desired module to the path, and then you can run *convert* again. If you run it without the *-p* option, it will tell you what stats-blocks are available, with the *-p* option, it will output a list of all available output-ports and their types. Now you can run *convert* one last time, adding the index of the stats-block or the output-port you want to look at, and probably redirecting the output into a file. This file can be read directly by *octave*. You can also add *-o file* to set the output-filename.

Chapter 26

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.

26.1 Debugging in user-space

While running either a test-case or a radio in user-space, it is quite simple to debug it, as all other radios can be stopped during the debugging session. This is of course not possible to do in real-time...

26.1.1 Using gdb

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*

26.1.1.1 Known bugs

It's gdb. So it's an extremly powerful environment but sometimes difficult to use. Always try the *help* command and read everything that is on the screen.

26.1.2 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!

26.1.2.1 Known bugs

On mandrake-linux, libpthreads is badly compiled, so that you can't use ddd in this distribution. Sorry. Install a debian or something...

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->Run Again*. Optionally you can use the F3-key.

Chapter 27

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.

27.1 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/testing and newer. 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.

27.2 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.

27.2.1 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.

27.2.2 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.

27.2.3 Common errors

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

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

27.3 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 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!

Chapter 28

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.

28.1 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.

28.2 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 29

TODO

29.1 Using CVS

describe the most important things about cvs, how to use lussh, how to update, commit, join and create branches, some important issues a.s.o.

29.2 Writing a new Module

29.3 Writing a Test

29.4 Writing a new Radio

Part V

Tips ’n Tricks

Part VI

Future thoughts