Design Specification

Editor: Mikael Karlsson

Version 1.0

Status 3

Reviewed	Mikael Karlsson	2015-11-04
Approved	Antonios Pitarokoilis	2015-11-04

Project Identity

HT 2015
Linkoping University, ISY

Name	**Responsibility**	**Phone**	**Email**
Atheeq Ahmed	Group Member	070-4131687	athah298
Martin Andersson	Documentation Manager	073-3912657	maran703
Björn Ekman	Software Manager	070-2420137	bjoek586
Mikael Karlsson	Project Manager	076-2257939	mikka789
Valens Nsengiyumva	Hardware Manager	070-4131684	valns738
Oscar Silver	Test Manager	073-0589005	oscsi278

Customer: ISY, Linkoping University, 581 83, Linkoping

Customer contact: Hien Ngo, hien.ngo@liu.se
Examiner: Danyo Danev, danyo.danev@liu.se
Tutor: Antonios Pitarokoilis, antonios.pitarokoilis@liu.se

Contents

DOCUMENT HISTORY

Version	**Date**	**Changes**	**Sign**	**Reviewed**
0.1	2015-10-12	First Draft	Martin A	Mikael K
0.2	2015-10-20	Second Draft	Martin A	Mikael K
1.0	2015-11-04	First Version	Martin A	Mikael K

1 Introduction

We plan to design and realize a system that will demonstrate the Zero Forcing capabilities and possible applications of Massive Multiple Input Multiple Output (Massive MIMO). Zero Forcing is a type of beamforming technique used in Massive MIMO in a multi-user environment where the signal energy towards an intended user is maximised under the constraint that no interference is caused to the remaining users. The purpose of this document is to specify the design of the system and how we will be implementing said design.

1.1 MIMO

MIMO is an essential element of wireless communication that uses multiple antennas at the transmitter and receiver to enhance the capacity of the radio link. Massive MIMO is a new innovative version of MIMO which uses a very large number of transmitter antennas that are operated in a completely coherent and adaptive manner to focus the transmission of signal energy into small regions of space. Massive MIMO can potentially increase the capacity by at least 10 times due to its aggressive spatial multiplexing while simultaneously improving the energy-efficiency with a factor 10-100.

1.2 Beamforming

Beamforming is one of the techniques that can be used in an MIMO system in which signals from the multiple antennas as transmitted in such a way that the signal energy gets focused at certain points in space by constructive combining. In the case of Zero Forcing beamforming the signal energy is focused at the relevant user while the other users essentially receive no signal at all. This is an important property for a wireless communication system since, in the ideal case, transmission to one user does not causes interference to other users.

1.3 The Project

For the project course TSKS05 CDIO Communication Systems, we plan to demonstrate the Zero Forcing capabilities of Massive MIMO building upon the work done by a previous project Massive Audio Beamforming [5]. Using the hardware built in the previous project we will design and implement the Zero Forcing algorithm for the existing system. The Zero Forcing and MIMO technique will ultimately be demonstrated by sending different data to the two different terminals, simultaneously.

1.4 Aim and goals

See section 2.1 in the document Project Plan [3].

1.5 Definitions

See Table 1 for definitions of words used in this document.

Table 1: Definitions of words used in this document.

Word	**Definition**
MIMO	Multiple Input Multiple Output
A/D	Analog to Digital
D/A	Digital to Analog
L/M unit	Loudspeaker/Microphone unit
L/M-pair	A pair of Loudspeaker/Microphone units
MIMO-array	The array of L/M pairs used to generate the Zero Forcing beam
Terminal	One of the two units in the receiving L/M pair.
OS	Operating System
Subsystem	A part of the whole system

2 Overview of the System

The system will basically be an upgrade of the existing system which was built in the previous project. The system will consists of two subsystems, namely the Software and the Hardware. The Hardware of our project is the same as that of the existing system, so this project's focus is just the Software. The new system should be able to demonstrate Zero Forcing by simultaneous data transmission to two different terminals, both receiving different data at the same time.

The Hardware consist of eight L/M pairs of which seven act like a MIMO-array while the last pair recievies the signals from the MIMO-array. An overview of the system is depicted in Figure 1.

The Software of the existing system consists of MATLAB scripts for channel estimation, signal generation and spectral analysis. It also includes drivers for communication between a Windows operative system and the Hardware.

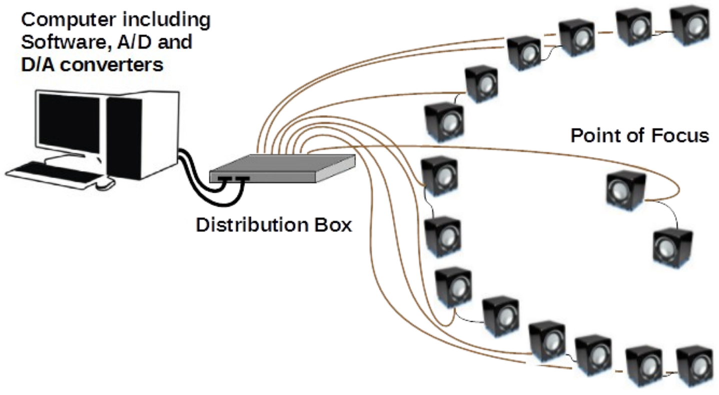

Figure 1: Overview of the Hardware in the system. [5, Fig. 1]

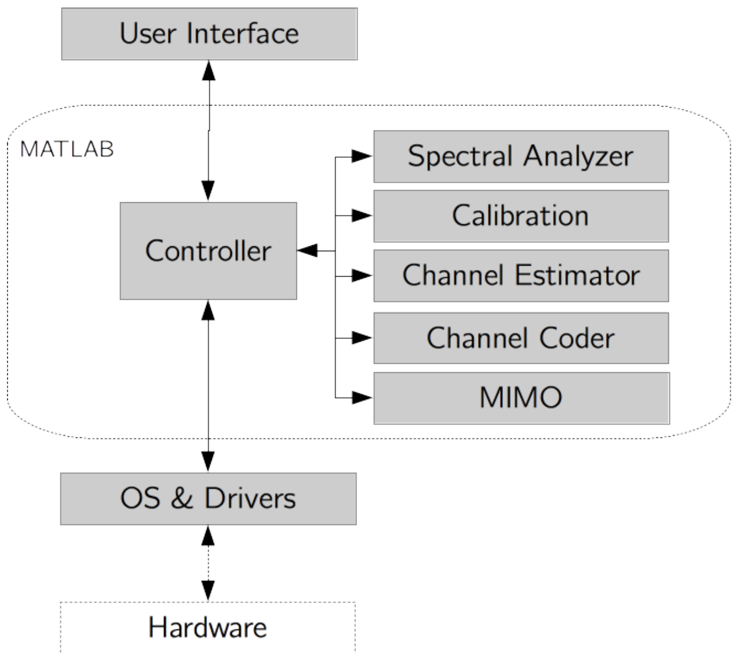

Figure 2: Overview of the software.

2.1 Validation of the Existing System

The current system will be checked to analyse its performance. To improve the performance of the system the hardware has to be fine tuned to work harmoniously with the software. The system will be tested to validate if the inputs to the hardware produce the expected outputs. MIMO Zero Forcing requires accurate measurements of the channel and the audio output of the L/M array also has to be precise. Any sensitivity differences in the L/Ms need to be estimated and compensated for in the software.

3 Software

The Software will be built in a modular fashion with well defined interfaces in between and an overview is depicted in Figure 2. This enables developers to easily tweak or replace individual modules. Below follows a description of each module and its task.

3.1 General Implementation of the Modules

The different modules in Figure 2 will all be coded in MatLab, in the form of different functions and scripts.

All scripts uses the underlying workspace while each function uses its own private workspace. This gives scripts access to all variables already in the general workspace and all variables declared in a script will end up in the general workspace. Functions on the other hand can only use copies of variables sent along with the function call. The output of the function can be several variables, but all need to be stated in the function signature and the caller can opt not to add them to its workspace. This difference makes scrips ideal for start-up, initialisation and small tests while functions better provide concrete interfaces, modularity and flexibility.

This project will mainly use functions. Scripts will be used when appropriate (initialisation and small tests) and will be named with a "*Script" suffix.

To get a clear and concise code the following coding guidelines will apply:

- Clear and concise commenting.
- Comments should be written in such a way that the built in Matlab-functions "help" and "doc" are able to display information about the function.
- All in and out parameters should be well defined: purpose and expected dimensionality.
- All global variables used should also be well defined at the top of the function, clearly marked as globals.
- Data-type is only needed for parameters that are not the Matlab default type (double-precision floating-point).
- Try to split functionality into separate rows and avoid "one-row-magic". When unavoidable (e.g. Matlab efficiency reasons): break it done into pseudo-code and put that in comments above the magic line.
- Use uppercase for constants, e.g. *CONSTANT_VARIABLE*.
- Use camel case for variable names, functions and scripts, e.g. *variableName*.
- Avoid hardcoded numbers - declare a constant instead.
- Some constants, or read-only variables, could be declared as globals.

3.2 Controller

The Controller will be responsible for the general flow of the program and supplies data to and from the other modules. It will also contain the functions necessary to communicate with the hardware drivers. The controller will contain the main loop of the system.

As for the controller functions, these are functions to communicate with the hardware drivers. Thus the functions for sending and receiving data lies under the Controller, since these are functions that need to interact with the drivers for the D/A and the A/D converters.

3.2.1 Implementation

The Controller module consists of one main script and four subfunctions.

controllerMainScript

controllerMain is the main script of the software. It will start with letting the user choose his/her own values or the default values for different initial variables. It will then continue by one at a time communicating to and in between the different subsystems, in such a way that the end system does what is required. The main script can therefore be read as a time line, from start to finish, of the scheduling of the different tasks in the system. In Listing 1 you can see the pseudo-code of the Controller's main function.

Listing 1: Pseudo code for the `controllerMainScript` script

```
controllerInitScript();
pilot = constructPilot(carrierFreq, timeDuration, sampleFreq);
receivedPilotVec = sendReceivePilots(pilotVec);
channelInformation = channelEstMain(pilots, receivedPilotVec);
bitVec = createBitVec();
errControlBitVec = errorControl(bitVec);
symbolVec = mapToSymbols(errControlBitVec);
MIMOSymbolVec = MIMOCombining(symbolVec, channelInformation);
outputSampleVec = upConvert(MIMOSymbolVec);
receivedSampleVec = sendReceiveData(outputSampleVec);
receivedBitVec = demodulate(receivedSampleVec);
decodedBitVec = decode(receivedBitVec);
errorRate = calcErrorRate(bitVec, decodedBitVec);
```

controllerInitScript()

controllerInitScript() is the function which will take care of the initialization of a lot of the system's different variables and constants. It will make the user able to set the values of different variables through input but the user will also be able to set variables to default values. It will further create the output and input channels for communication with the D/A and A/D converters.

Table 2: Variables which can be chosen by the user, and their default values.

Variable	Description	Default Value
CARRIER_FREQ	The carrier frequency used by the system for communication purposes	502
SAMPLE_FREQ_TERMINALS	The sample frequency used for the terminals	6024
SAMPLE_FREQ_ARRAY	The sample frequency for the MIMO-array	6024
ARRAY_PAIRS	Vector of the numbers of the LM-pairs in the MIMO array	[1, 2, 3, 4, 6, 7, 8]
TERMINAL_PAIRS	Vector of the numbers of the LM-pairs working as terminals	[5]

Table 3: Constant Variables used in the system.

Variable	Description	Value
AMOUNT_LMGROUPS	Number of LM-groups, groupmembers work in the same operation mode	2
ARRAY_GROUP	Which of the LM-groups are the MIMO array, used as flag	0
TERMINAL_GROUP	Which of the LM-groups that are the two terminals, used as flag	1
CARRIER_FREQ_MIN	The minimum possible carrier frequency	300
CARRIER_FREQ_MAX	The maximum possible carrier frequency	3012
AO	Analog output object for the D/A converter	-
AI	Analog input object for the A/D converter	-

sendReceivePilots()

The high level function used in controllerMain to send and receive the pilots.

Listing 2: Pseudo code for the `sendReceivePilots()` function

```
function [receivedPilotVec] = sendReceivePilots(pilotVec)
    % Setup variables
    ...
    receivedPilotVec = sendReceive(
        pilotVec,
        AO,
        AI,
        TERMINAL_PAIRS,
        ARRAY_PAIRS,
        SAMPLE_FREQ_TERMINALS,
        SAMPLE_FREQ_ARRAY);
end
```

sendReceiveData()

The high level function used in controllerMain to send and receive the pilots.

Listing 3: Pseudo code for the `sendReceiveData()` function

```
function [receivedDataVec] = sendReceiveData(dataVec)
    % Setup variables
    ...
    receivedDataVec = sendReceive(
        dataVec,
        AO,
        AI,
        ARRAY_PAIRS,
        TERMINAL_PAIRS,
        SAMPLE_FREQ_ARRAY,
        SAMPLE_FREQ_TERMINALS);
end
```

sendReceive()

A general, low level function for sending and receiving data. Will be used by sendReceivePilots() and sendReceiveData().

Listing 4: Pseudo code for the `sendReceive()` function

```
function [receivedSignalVec] = sendReceive(
    signalVec,
    analogOutput,
    analogInput,
    sendingLMPairsVec,
    receivingLMPairsVec,
    sampleFreqSending,
    sampleFreqReceiving)
    ...
end
```

3.3 Calibration

The amplitude parameters sent to the L/M units will be calibrated to ensure that all the L/M pairs perform as uniformly as possible and any differences between them are compensated for in software. This will be accomplished by using one pair as reference microphones in order to be able to calibrate the rest. The reference pair should be the same pair that later on will be used as the pair that the MIMO array will be sending to.

3.3.1 Implementation

The amplitude characteristics will be done by measuring the audio outputs of each L/M unit for linearly increasing amplitude parameters in the software input. This test will be done for each of the MIMO array units individually where each unit will act as a speaker and the one of the terminal unit at the point of focus will act as the receiver. A graph will be plotted which will give us the mapping of audio signal amplitude to the amplitude parameters used in the software. Similarly the sensitivity characteristics of the MIMO array units when they act as receivers is determined. The amplitude calibration will then be done using the obtained mapping.

3.4 Channel Estimator

The Channel Estimator will be used to estimate the impulse responses of each channel between the two terminals to each L/M-unit in the L/M-array, the channels are depicted in Figure 3. An element $h_{i,j}$ of the true impulse response matrix H is stated in Equation (2), where α is the complex attenuation and τ is the phase shift. H is defined in equation (1).

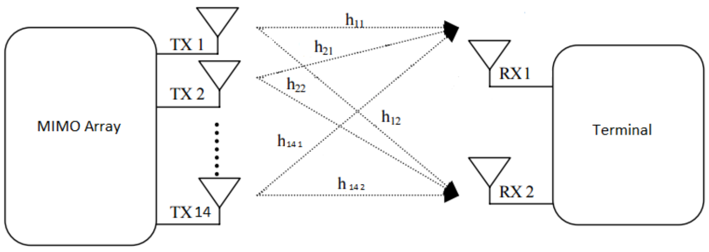

Figure 3: Channels in the MIMO system.

$$H := \begin{bmatrix} h_{1\ 1} & h_{1\ 2} \\ h_{2\ 1} & h_{2\ 2} \\ \vdots & \vdots \\ h_{14\ 1} & h_{14\ 2} \end{bmatrix} \tag{1}$$

$$h_{i,j}(\tau) = \sum_{l=1}^{(l)} \alpha_{i,j}^{l} \delta(t - \tau_{i,j}^{(l)}) \tag{2}$$

With the assumed flat fading channel most of the energy from the reflections will arrive within one symbol interval. Therefore the individual reflections are not resolvable and only an estimate of their superposition is possible. The true H will be estimated with $\hat{H}$ which will consists of elements defined in equation (3).

$$\hat{h}_{i,j}(\tau) = \sum_{l=1}^{(l)} \hat{\alpha}_{i,j}^{l} \hat{\delta}(t) \tag{3}$$

The channels, which are assumed to be flat fading, will be estimated by sending a known signal from the two terminals and record what the L/M-array receive. By comparing the known sent signal with the received one, the characteristics of the channel can be estimated.

The input to the Channel Estimator module will be the sent pilot signal and the received signals for all the units in the L/M-array together with information about their respective sampling frequencies. The output from the Channel Estimator module will be the estimated relative amplitude change and phase shift for each channel.

Cross-correlation will be used to estimate the phase shift introduced by the channel. It will also be investigated whether the zero crossings of the signals can be used for phase estimation since this would give a continuous scale rather than the discrete one that the cross-correlation will give. To estimate the relative amplitude change the recorded signals will be integrated over a characteristic section and then compared.

The Channel Estimator module is also responsible for generating the pilot to be sent during training. This is achieved with the `constructPilot()` function detailed below.

3.4.1 Implementation

The Channel Estimator module will consist of a main function, `channelEstMain`, with two subfunctions, `channelEstAmp` and `channelEstPhase`, to estimate the amplitude and phase respectively. The main function is responsible for executing the two subfunctions for all the received signals and allocate the output matrix containing information about amplitude and phase estimates and the pseudo code for it is given in Listing 5.

Listing 5: Pseudo code for the `channelEstMain` function

```
function [estimationMatrix] = channelEstMain(pilot, recordedSignals)
    amplitudeEst = channelEstAmp(recordedSignals)
    phaseEst = channelEstPhase(pilot, recordedSignals)
    estimationMatrix = generateMatrix(amplitudeEst, phaseEst)
end
```

channelEstAmp()

The amplitude of the received signals will be estimated using the samples from a selected window of the received signal and taking an average of the integration.

Listing 6: Pseudo code for the `channelEstAmp` function

```
function [ampEst] = channelEstAmp(recordedSignals)
    for i in recordedSignals
        ampEst[i] = integrate(recordedSignals)/time
    end
end
```

channelEstPhase()

The cross-correlation between two functions, f and g, is given in Equation (4) for the time discrete case. The phase shift τ between two signals can then be estimated as the argument of the maximum of the cross-correlation, as stated in Equation (5). If this method is found to not be accurate enough then Fast Fourier Transform (FFT) Maximum Likelihood Estimation (MLE), as discussed in [2], might be a possible way of improving the accuracy.

$$(f \star g)[n] = \sum_m f^*[m]g[m+n] \tag{4}$$

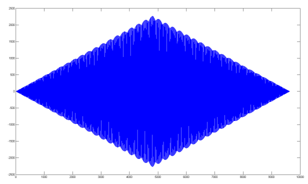

Figure 4: Auto-correlation of a sinus signal.

$$\tau = \arg\max_n (f \star g)[n] \tag{5}$$

MATLAB's function `xcorr` will be used to cross-correlate the pilot with the received signals and thereby get an estimate of the phase shift introduced by the channel. To avoid transients a window of the signals will be used. Note that we are only interested in relative phase shifts between the signals and not in shifts introduced due to delays in software or hardware. If the sent pilot and recorded signals have different sampling frequencies then the pilot will have to be downsampled before the cross correlation.

Listing 7: Pseudo code for the `channelEstPhase` *function*

```
function [phaseEst] = channelEstPhase(pilot, recordedSignals)
    if (different sampling frequencies)
        pilot = downsample(pilot)
    end
    pilotWindow = extractWindow(pilot)
    for i in recordedSignals
        indexLag[i] = xcorr(pilotWindow, extractWindow(recordedSignals[i]))
    end
    % Convert the indexLag to phase
    phaseEst = 2*pi*indexLag*carrierFreq/sampleFreq
end
```

constructPilot()

The pilot used for estimating the channel will be a chirp with frequencies around the frequency which will be used as carrier frequency. See Listing 1 for the full function signature. The channel is assumed to be flat in this interval and the chirp will thereby provide better characteristics than a simple sinusoid for the phase estimation since with the chirp delays larger than 2π can be detected as well. Furthermore, the chirp's auto-correlation gives a more distinct peak as seen in Figures 4 and 5. Sending the pilot with a high and constant amplitude will ease the amplitude estimation and hopefully overpower most of the noise. By assuming reciprocity for the channel, the pilot only has to be sent from the two terminals instead of all the units in the MIMO array. The output of the `constructPilot()` function will be a vector with samples to be sent.

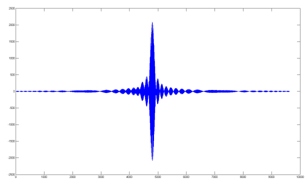

Figure 5: Auto-correlation of a chirp signal.

3.5 Spectral Analyser

The Spectral Analyser module will be responsible for estimating and plotting the power spectral density and sound intensity at the terminals. The main purpose of the module is testing and validation of input/output.

3.5.1 Implementation

The module will take the signal and the frequency with which it was sampled as input and output the main frequency component of the signal. Plots can also be created by setting the relevant flags.

The main frequency component will be calculated using the Fast Fourier Transform (FFT) to transform the signal to the frequency domain and then locate the maximum and convert the found index to the corresponding frequency.

3.6 MIMO

This module will combine the results from the Channel Estimator with the data from the Channel Coder according to a MIMO-combining technique, in this case Zero-Forcing, in order to send signals to each terminal. The output will be the time signals for each L/M-unit in the MIMO array. An overview of this setup is depicted in figure 6.

3.6.1 Implementation

The implementation of this module consists of a main function and a help function used to perform necessary matrix calculations.

MIMOCombining()

Performs MIMO combination with Zero-Forcing on input data. Returns a vector with signals for each L/M-pair.

Listing 8: Pseudo code for the `MIMOCombining()` function

```
function [MIMOComplexVec] = MIMOCombining(complexVec, channelEstimationMatrix
     )
    W = calculatePseudoInverse(channelEstimationMatrix)
    MIMOComplexVec = W*complexVec
end
```

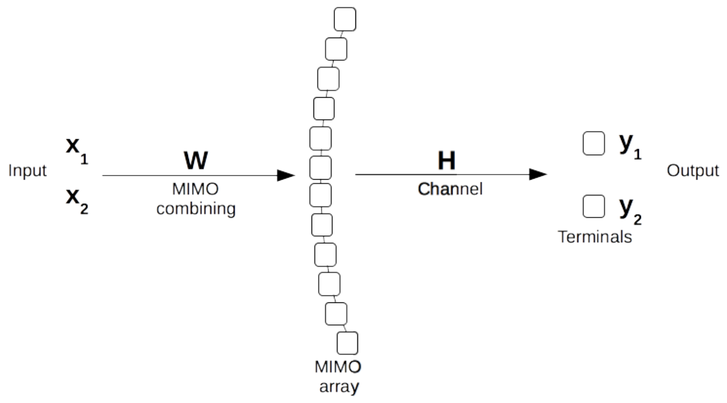

Figure 6: Overview of MIMO system with Zero Forcing.

$$\begin{aligned} &Y = HWX, \textit{ where} \\ &\qquad H = \text{Channel}, \\ &\qquad W = \text{Pseudo Inverse of estimated channel} \\ &\qquad X = \text{Input vector}, \\ &\qquad Y = \text{Output vector}, \end{aligned} \tag{6}$$

We want to realize that the received signal Y equals our sent signal X at each terminal as in equation 6. Assuming no noise on the channel and that we have the true channel estimation matrix we can precode our input vector X by multiplying it by the inverse W of the channel estimation matrix which is given by equation 7.

$$W = H^*(HH^*)^{-1} \tag{7}$$

Then we get our precoded vector WX which yields $Y = HWX = X$ at the the receiving terminals.

However we have noise on our channel which gives us an imperfect channel estimation matrix. So a better representation of our situation is given by equation 8 where H is multiplied by the pseudo-inverse of estimated H. In the implementation of zero-forcing we will use our imperfect channel estimation matrix like if it were the true channel estimation matrix.

$$Y = H\hat{H}^*(\hat{H}\hat{H}^*)^{-1}X \tag{8}$$

We also have to consider the hardware limitations on the amplitude of the signals which will restrict the amplitudes we can send from the sender. This will further complicate the calculation of a perfect pseudo-inverse.

3.7 Channel Coder

The Channel Coder will produce symbols to send to the terminals. This includes functions for generating data, error-control, modulation, demodulation and decoding.

3.7.1 Implementation

The block structure of what the Channel Coder will do can be seen in Figure 7. Here you can see the general flow of the different functions and data types, how to go from an original bit stream to an actual output signal to send to the different speakers. The different variables in the figure are:

- n: amount of bits in the created bit vector
- R: efficiency of the error control code
- m: number of symbols in the symbol constellation
- t: number of samples/symbol

The different functions (except `MIMOCombining()` which can be found under Section 3.6) are described further under this section.

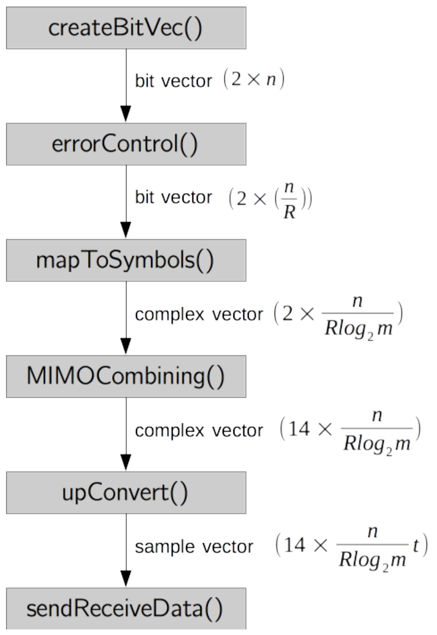
createBitVec()
bit vector $(2 \times n)$
errorControl()
bit vector $(2 \times (\frac{n}{R}))$
mapToSymbols()
complex vector $(2 \times \frac{n}{R\log_2 m})$
MIMOCombining()
complex vector $(14 \times \frac{n}{R\log_2 m})$
upConvert()
sample vector $(14 \times \frac{n}{R\log_2 m} t)$
sendReceiveData()

Figure 7: Overview over the flow of data from creation to sending.

createBitStream()

Creates a random bitstream consisting of ones and zeros, given the size of the wanted bit vector.

Listing 9: Pseudo code for the `createBitStream()` function

```
function [bitVec] = createBitStream(bitLength)
    ...
end
```

errorControl()

Extends the bits given by the data generation with extra redundancy for error control. Aim is to go for something simple, for example Hamming code, and get something that works.

Listing 10: Pseudo code for the `errorControl()` function

```
function [errControlBitVec] = createBitStream(bitVec)
    ...
end
```

mapToSymbols()

Maps a bit vector to a symbol vector. The symbols are expressed as complex values which represent amplitude and phase shift of the carrier signal. The modulation tried at first will be BPSK.

Listing 11: Pseudo code for the `mapToSymbols()` function

```
function [complexVec] = modulate(bitVec)
    % read m bits
    % map them to a complex value using an m-ary constellation

    ...
end
```

upConvert()

This function up converts the complex symbols given as input into an actual signal represented by discrete samples. Input is a complex symbol vector and output will be a vector of discrete samples.

Listing 12: Pseudo code for the `upConvert()` function

```
function [sampleVec] = upConvert(complexVec)
    %map each symbol into a set of discrete samples representing one waveform
    %output a vector of discrete samples representing a signal
    ...
end

\paragraph{demodulate()}
Using an appropriate detector scheme to retrieve bits given a received wave-
     form.

\begin{lstlisting}[caption={Pseudo code for the \texttt{demodulate()}
     function}, label={lst:demodulate}]
function [demodBitVec] = demodulate(receivedSampleVec)
    % correlate received signal with possible sent signals.
    % pick the best one as the sent symbol
    % translate into bit block
    ...
end
```

decode()

Decodes the error controlled bit vector to a clean bit vector with no redundancy bits.

Listing 13: Pseudo code for the `decode()` function

```
function [decodedBitVec] = decode(codedBitVec)
    ...
end
```

calcErrorRate()

Inputs the sent as well as the received bit vector and checks the error rate.

Listing 14: Pseudo code for the `getErrorRate()` function

```
function [errorRate] = calcErrorRate(origBitVec, recBitVec)
    ...
end
```

3.8 User Interface

The User Interface will be the user's main way of running the program and modify its behaviour. The idea is to build upon the previous project's command line tool running in the MATLAB console. Through the interface the user should be able to modify some of the parameters controlling the execution of the program.

3.9 Operating System and Drivers

In order to control the L/M-units the A/D and D/A card used by previous project will be used this year too. These uses PCI-connections and are controlled using their respective Windows drivers. The drivers are accessed from MATLAB by use of an add-on called

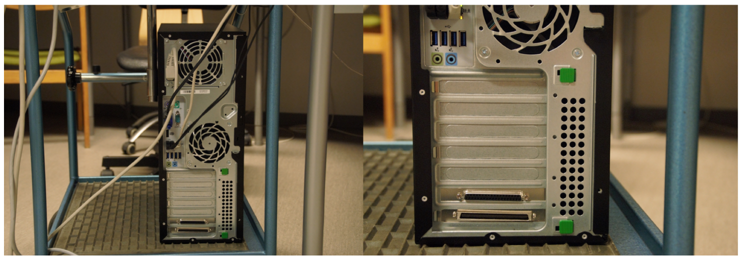

Figure 8: The computer back panel with A/D and D/A slots. [6, Fig. 2]

"Data Acquisition Toolbox" (DAQ) as well as a MATLAB library called "MATLAB-compliant Data Acquisition Library" (MLDAQ) [5]. We expect most of this to be in place already from last year.

4 Hardware

The hardware from the previous project will not be altered and the description given here is therefore heavily based on the technical report produced by the previous project [5]. The hardware is only described to make this document complete but no development will be done regarding the hardware.

The existing hardware consists of a computer, an A/D converter, a D/A converter, a distribution box and L/M-pairs. All these entities are described in this chapter. An overview of the hardware is given in Figure 1.

4.1 Computer

A computer is an electronic device for manipulating information or data, which can be stored, retrieved and processed. The model of the computer used in the project is a Hp Compaq Elite 8300 running Windows 7 as its operating system.

4.2 L/M pairs

A L/M unit is a modified loudspeaker that can also act as a microphone. The L/M unit can operate either as a loudspeaker or a microphone based on the input from the user. A L/M pair is a set of two L/M units in which one is a master unit and the other a slave unit. The system has 8 L/M pairs and one of these pairs is depicted in Figure 9. The master unit is made of an original amplifier board and an additional detection board designed by Mikael Olofsson. The detection makes it possible for the L/M pair to operate as both loudspeaker and a microphone. The 9-pole D-sub connector (DB-9) mounted on top of the master unit serves as interface for signal to, or from, the L/M pair. The power supply to the L/M pair is done through the USB cable, connected to the wall via a USB adapter.

4.3 A/D and D/A converters

An A/D converter is an electronic circuit used to convert an electrical signal into binary numbers to be used in a digital controller (computer). A D/A converter circuit is used to convert binary numbers to analog voltage or current. One A/D converter (Contec AD12-64) and one D/A converter (Contec DA12-16) are attached to the computer motherboard through PCI slots to ensure the communication between the computer and the distribution box. With the help of the internal sample clock in each card, resolution of 12V and highest conversion speed of 100 kilosamples/s are attained for both converters. The voltage levels for both converters are in the interval [-10, 10] V. Of the 64 analog input channels only 16 are used in the product. The A/D converter also has 4 digital inputs and 4 digital outputs for TTL level signals (Transistor-Transistor-Logic). In the product only two of the digital outputs of the A/D converter are utilized and these are responsible for switching between the two operation modes of the L/M pairs.

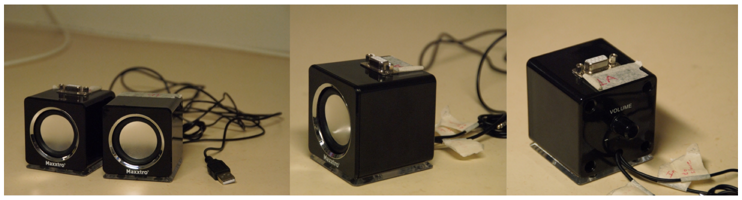

Figure 9: The L/M pair and the master unit from different angles. [6, Fig. 6]

4.3.1 Detection Board

Microphone mode operation for the L/M pair is possible by means of the detection board. When the L/M pair is working in this mode the circuit amplifies the audio signals received using a differential-in-differential-out amplifier with a voltage gain of 23 dB. Thereafter, the amplified signals are forwarded to the collection board for further amplification.

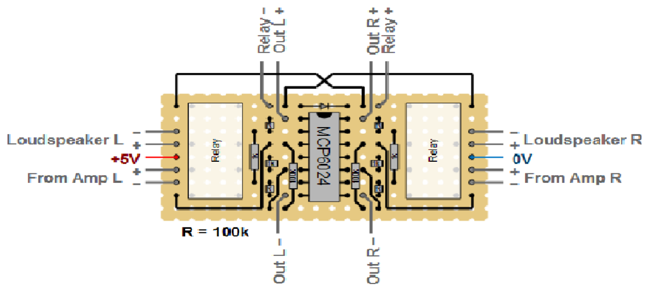

Figure 10: Detection board. [5, Fig. 3]

4.3.2 Maxxtro Mini Speaker 4W

A master loudspeaker and a slave loudspeaker constitute the main parts of the L/M pair. Both speakers, joint by a stereo cable of length 0.3m, are supplied 5V by the USB cable of 1m length connected to an adapter. The adapter, called *Euro-USB-laddare*, has the stock number 25-249-98 at ELFA. The USB cable and volume controller are attached only to the master loudspeaker.

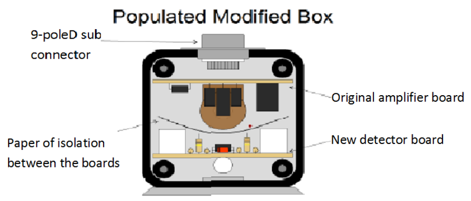

Figure 11: Overview of the L/M master unit. [5, Fig. 7]

4.4 Distribution Box

The distribution box depicted in Figure 12 works as a hub and distributes the signals between the computer and the L/M pairs. It has 11 connections: eight for the L/M pairs, one for the A/D and D/A converters respectively and a power supply connection. For the purpose of letting the user choose which control group a L/M pair is part of the distribution box also has a control board consisting of eight physical switches.

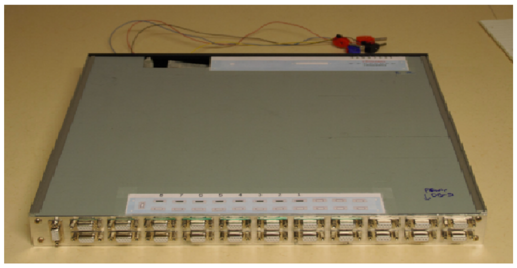

Figure 12: The Distribution Box. [6, Fig. 3]

4.5 Limitations on available hardware

Limitations on the hardware available from the previous project are presented below in the different sections.

4.5.1 Sampling frequency

The maximum sampling frequency that can be used for the A/D and D/A converter is 100 kHz. These 100 kHz are divided among all channels in use. Thus, when all 16 L/M units are in use the maximum sampling frequency is limited to 6250 Hz in theory. But the sampling frequency is fixed to certain values so in practice the maximum sampling frequency is 6024 HZ.

4.5.2 Data transmission

The limit of the amount of data that can be put through the D/A engine is limited to 250 000 samples, which translates to about 2.5 seconds of continuous sound, when using a sampling frequency of 6024 Hz and 16 channels.

4.5.3 A/D converters

The A/D converter has a voltage range of [-10,+10] V. However the power supply can deliver voltages in the interval [-11,11] V which means that the A/D converter can be damaged by a too high sound level from the loudspeakers.

References

[1] Tomas Svensson, Christian Krysander, *Projektmodellen LIPS*. Studentlitteratur, 2011.

[2] D. Rife and R. Boorstyn, *Single-tone parameter estimation from discrete-time observations*, IEEE Transactions on Information Theory, vol. 20, no. 5, pp. 591–598, 1974.

Unpublished References:

[3] Mikael Karlsson et al., *Project Plan*. ISY, Linkoping University, 2015.

[4] Mikael Karlsson et al., *Requirement Specification*. ISY, Linkoping University, 2015.

[5] Fredrik Stenmark et al., *Technical Report*. ISY, Linkoping University, 2014.

[6] Fredrik Stenmark et al., *User Manual*. ISY, Linkoping University, 2014.

[7] Fredrik Stenmark et al., *Project Plan*. ISY, Linkoping University, 2014.