



Corporate Sector
Centre de Recherche de Motorola - Paris

**Development of an Asymmetric Digital
Subscriber Line (ADSL) Simulator in C/C++
and
Examination of Time and Frequency
Domain Equalization Methods**

Author: Markus Mück
Approval: Marc de Courville

DISTRIBUTION: (1) Public
(2) ENST
(3) University of Stuttgart

MOTOROLA PROPRIETARY INFORMATION

Last name: Mück
First name: Markus

Report version: 1.0

Date of report version: 22/04/1999

Topic of the diploma thesis: Development of an Asymmetric Digital Subscriber Line (ADSL) Simulator in C/C++ and Examination of Time and Frequency Domain Equalization Methods

Time frame: 01/11/1998 to 30/04/1999

Responsible person: Professor Joseph Jean Boutros
(ENST) Ecole Nationale Supérieure des Télécommunications (ENST)
Département COMELEC
Email: boutros@com.enst.fr
Tel.: +33-1-45817678

Responsible person: Dr. Marc de Courville
(Motorola) Research Engineer
Email: courvill@crm.mot.com
Tel.: +33-1-6935-2518

Responsible Person: Professor Joachim Speidel
(University of Stuttgart) University of Stuttgart
Institut fuer Nachrichtenerbertragung
Email: speidel@inue.uni-stuttgart.de
Tel.: +49-711-685 8016/8017

Contents

Acknowledgement	10
List of symbols	12
Institutions involved in this diploma thesis	16
1 Introduction	18
2 Introduction to OFDM/DMT Systems	20
2.1 A mathematical presentation of OFDM/DMT	20
2.2 A discrete channel model	25
2.3 The guard interval	26
2.4 A complete OFDM/DMT system	28
3 Introduction to ADSL	30
3.1 General description of an ADSL system	30
3.2 Some technical details	31
3.3 An ADSL system in detail	32
4 The ADSL simulator in C/C++	34
4.1 The structure of the ADSL simulator	34
4.2 The different modules of an ADSL system	34
4.2.1 The ADSL-frame builder	36
4.2.2 CRC generation and CRC check	36
4.2.3 Energy scrambler and descrambler	36
4.2.4 Reed-Solomon encoder and decoder	36
4.2.5 Interleaver and deinterleaver	36
4.2.6 Tone ordering and re-ordering	37
4.2.7 Trellis coding	37
4.2.8 VITERBI decoding	37



4.2.9	Constellation encoder and decoder	37
4.2.10	DFT and IDFT	37
4.2.11	Time Domain Equalizer (TEQ)	37
4.2.12	Frequency Domain Equalizer (FEQ)	37
4.2.13	Transmission channel	38
4.3	Description of the ADSL simulator program design	38
4.3.1	Flowchart of the main function	38
4.3.2	Flowchart of the transmitter functions	38
4.3.3	Flowchart of the receiver functions	38
5	Equalization methods for OFDM/DMT systems	43
5.1	Equalization using a multiplication in the frequency domain	43
5.2	Equalization using a Target Impulse Response (TIR) filter	45
5.3	Equalization in the time domain without guard interval	47
5.4	Discussion	48
6	A new algorithm updating the Time Domain Equalizer	49
6.1	The filter structure	49
6.2	Choosing the adaptive step size	51
6.3	The update algorithm for the filter taps	52
6.3.1	A proposition for a practical implementation	56
6.4	Complexity evaluation	59
6.4.1	The complexity of basic operations	59
6.4.2	The complexity of Fast Fourier Transformation (FFT) algorithms	59
6.4.3	The complexity of convolution and correlation algorithms	60
6.4.4	The complexity of the tap update calculation	61
6.4.5	The complexity of the Time Domain Equalization (TEQ) convolution	67
7	Simulation results	69
7.1	Parameters of the simulation	69
7.2	Convergence properties without noise	72
7.3	Convergence properties with Additive White Gaussian Noise	73
7.4	The resulting TEQ filters	76
8	Conclusions	79
A	Convergence properties of the LMS algorithm	81
A.1	Some definitions and properties	81

A.2	Convergence properties of the LMS algorithm	82
B	A practical implementation of the WSAF algorithm	85
B.1	Switching the inputs to the FFTs/IFFTs	85
B.2	Standard definitions of FFT/IFFT operations	86
C	Software description of the ADSL simulator in C/C++	87
D	Top-level diagram of the ADSL simulator	89
E	Module-level description of the ADSL simulator	90
E.1	simulator.C/h	90
E.2	algor_enc.C/h	90
E.3	channel.C/h	90
E.4	conv_encoder.C/h	91
E.5	coset_select.C/h	91
E.6	CRC.C/h	91
E.7	cvector.C/h	91
E.8	decoder.C/h	91
E.9	DEINTERL.C/h	91
E.10	Descrambler.C/h	91
E.11	FEC.C/h	91
E.12	fft.C/h	92
E.13	generate.C/h	92
E.14	INTERL.C/h	92
E.15	ivector.C/h	92
E.16	my_types.C/h	92
E.17	receiver_home_part1.C/h	92
E.18	receiver_home_part2.C/h	92
E.19	routines.C/h	92
E.20	Scrambler.C/h	93
E.21	tone_order.C/h	93
E.22	tools.C/h	93
E.23	transmitter_CO_part1.C/h	93
E.24	transmitter_CO_part2.C/h	93
E.25	viterbi_decoder.C/h	93
E.26	wei_encoder.C/h	93



E.27	constants.h	93
E.28	debug.h	94
E.29	define.h	94
E.30	switch.h	94
F	Function-level description of the ADSL simulator	95
F.1	How to read this description	95
F.2	Transmitter	95
F.2.1	algor_enc.C	95
F.2.2	conv_encoder.C	97
F.2.3	coset_select.C	98
F.2.4	CRC.C	99
F.2.5	FEC.C	101
F.2.6	fft.C	104
F.2.7	INTERLEAVER.C	106
F.2.8	Scrambler.C	107
F.2.9	tone_order.C	108
F.2.10	transmitter_CO_part1.C	109
F.2.11	transmitter_CO_part2.C	110
F.2.12	wei_encoder.C	110
F.3	Channel and TEQ update	112
F.3.1	channel.C	112
F.4	Receiver	116
F.4.1	decoder.c	116
F.4.2	DEINTERL.C	117
F.4.3	Descrambler.C	118
F.4.4	receiver_home_part1.C	120
F.4.5	receiver_home_part2.C	120
F.4.6	viterbi_decoder.C	121
F.5	Supporting functions	125
F.5.1	cvector.C	125
F.5.2	generate.C	127
F.5.3	ivector.C	129
F.5.4	my_types.C	131
F.5.5	routines.C	134
F.5.6	tools.C	135

F.6	INCLUDE files containing some useful #defines	137
F.6.1	constants.h	137
F.6.2	debug.h	143
F.6.3	define.h	144
F.6.4	switch.h	144
G	The Parameter Files of the ADSL simulator	147
G.0.5	FILE_BITS_PER_TONE	147
H	The comments of the ADSL simulator during a simulation	148
	Reference	151



Acknowledgement

First of all, I am very thankful to Professor Joseph Jean Boutros for the supervising of this diploma thesis and to Marc de Courville, Véronique Buzenac and Sébastien Simoens for their help and encouragement.

Likewise, I would like to thank Jean-Noël Patillon for his technical aid and support, Paul Courbis and Mathieu Cousin for their help in the informatics as well as Patrick Labbé, Bertrand Muquet, Mickaël Batarriere, Gregoire Hourlier, Sylvain Jaume, Arianna Filoramo and Sandrine Vialle for their cooperation.

Last but not least, I want to give my sincere thanks to Professor Joachim Speidel of the University of Stuttgart who accepted to supervise this diploma thesis in the framework of the double diploma program for the University of Stuttgart.



List of symbols

$\mathbf{b}(k)$	Vector of noise
b_n	Noise samples
$\mathbf{B}(k)$	Fourier transformation of the vector of noise $\mathbf{b}(k)$
$c(t)$	Channel impulse response of a linear channel
c_n	Channel taps
c_n^i	Channel taps in the frequency domain
\mathbf{C}	Vector containing the channel taps
C_c	Matrix of channel taps used in order to calculate the received symbols
$C_0(N)$	Matrix containing the channel taps (influence of the previous symbol)
$C_1(N)$	Matrix containing the channel taps (influence of the latest symbol)
C_i	The elements of the fourier transform of \mathbf{C}
$\mathcal{C}_C^{TIME}(N)$	Abbreviation for a convolution/correlation performed in the time domain
$\mathcal{C}_{FC}^{FAST}(N)$	Abbreviation for a fast convolution/correlation performed in the frequency domain domain <i>without</i> transformation back into time domain
$\mathcal{C}_{TC}^{FAST}(N)$	Abbreviation for a fast convolution/correlation performed in the frequency domain domain <i>with</i> transformation back into time domain
d_n	Transmitted samples convolved with the TIR filter
d_n^i	Transmitted samples convolved with the TIR filter in the frequency domain
D	Size of the guard interval
$\text{DFT}_C(N)$	Abbreviation for a complex N -points FFT
e_{any}	Discrete impulse response of any Time Domain Equalizer
$e_{classic}$	Discrete impulse response of a classical Time Domain Equalizer
e_{DMT}	Discrete impulse response of a Time Domain Equalizer optimized for DMT systems
e_n	Residual error
e_n^i	Residual error in the frequency domain
E_n	Vector of the residual error
E_n^i	Vector of the residual error in the frequency domain
f_0	Lower bound frequency
f_m	Frequency shift for the orthogonal subcarriers
F_N	Matrix performing the fourier transformation

\tilde{F}_N	Matrix performing the fourier transformation multiplied with a constant
$F_{N,i}$	The i th line of the matrix performing the fourier transformation
$F_{N \times \frac{N}{2}}$	The first $\frac{N}{2}$ lines of the Matrix performing the fourier transformation
$g_k(t)$	Orthogonal subcarrier
$g_m(z)$	The m th SFB filter
g_n	TIR filter taps
g_n^i	TIR filter taps in the frequency domain
$G(z)$	Polyphase matrix associated with the synthesis filter bank performing the modulation
G_i^{MMSE}	Equalization coefficients using a <i>Minimum Mean Square Error</i> (MMSE) approach
G_i^{ZF}	Equalization coefficients using a <i>Zero Forcing</i> (ZF) approach
$G_m^l(z)$	Type-I l th polyphase component of $G_m(z)$
G_n	Vector containing the P TIR filter coefficients
$h_p(t)$	Matched filter
$I(k)$	Incoming data stream
I_n	Transmitted data
\tilde{I}_n	Received data
$I_n(k)$	The incoming data stream $I(k)$ is multiplexed into the sub-streams $I_n(k)$
I_N	Identity matrix
I_n^p	The n th symbol conveyed by carrier p
\tilde{I}_n^p	The estimated n th symbol conveyed by carrier p
J	Cost function
\hat{J}	Cost function estimated at the reception site
k	The latest iteration step number
K	Number of different orthogonal subcarriers $g_k(t)$
L	Number of TEQ filter taps
$mod_{\mathbb{C}}$	Abbreviation for complex modulus
N	Number of samples per DMT symbol (without guard interval)
P	Number of TIR filter coefficients
P_c	Length of the channel impulse response
P_{DMT}	Energy per DMT symbol divided by the time duration of a DMT symbol
P_{\times}	Cross-Correlation vector of received samples and desired filter response
$\mathbf{r}(k)$	Transmitted data convolved by the channel
$r(t)$	Rectangular function
r_n	Arriving samples
r_n^i	Arriving samples in the frequency domain
$r(n)$	The last N arriving samples
$r_k(n)$	The k th element of the vector R_{nN} , i.e. r_{nN-k}
$\mathbf{R}(k)$	Transmitted data convolved by the channel in the frequency domain
R_n	Vector of the last N arriving samples
R_n^i	Vector of the last N arriving samples in the frequency domain
\mathcal{R}_n	Matrix containing the last L vectors of the arriving samples

$\mathbf{S}(k)$	Vector containing $S_n(k)$ for K different n
$\mathbf{s}(k)$	Vector containing the inverse fourier transform of $\mathbf{S}(k)$
$\mathbf{s}^{ig}(k)$	Corresponds to $\mathbf{s}(k)$ with a cyclic prefix being added
$S_n(k)$	Information to be transmitted on DMT symbol subcarrier n at time k
$t_r[k]$	Autocorrelation of arriving samples
T	Time distance between two samples of a DMT symbol
T_A	Autocorrelation matrix of the received samples
T_s	Time duration of an OFDM symbol
$u(t)$	Normalized rectangular function
U	Modal matrix of the autocorrelation Matrix T_A
w_n	TEQ filter taps
w_n^i	TEQ filter taps in the frequency domain
W_n	Vector containing the L TIR filter coefficients
W_o	Vector containing the optimum L TIR filter coefficients
W_N	Constant for the fourier transformation
x_n	Transmitted samples
x_n^i	Transmitted samples in the frequency domain
X_n	Vector of the last N transmitted samples
X_n^i	Vector of the last N transmitted samples in the frequency domain
\mathcal{X}_n	Matrix containing the last P vectors of the transmitted samples
y_n	Received samples convolved with the TEQ filter
y_n^i	Received samples convolved with the TEQ filter in the frequency domain
y_p^n	Matched filter outputs
$\alpha_{\mathbb{C}}$	Number of complex additions
$\alpha_{\mathbb{R}}$	Number of real additions
$\delta_{i,i'}$	Kronecker symbol
ΔW	TEQ tap update
ϵ_n	Error between the transmitted data I_n and the received data \tilde{I}_n
λ_i	Weighting factor
Λ	Matrix of weighting factors
μ	Factor adapting the step size
$\mu_{\mathbb{C}}$	Number of complex multiplications
$\mu_{\mathbb{R}\mathbb{C}}$	Number of real \times complex multiplications
$\mu_{\mathbb{R}}$	Number of real multiplications
$\sigma_{B_i}^2$	Variance of the noise
$\sigma_{r^i}^2$	Variance of the arriving samples in the frequency domain
σ_s^2	Variance of the elements of $\mathbf{s}(k)$
$\tau(\mathbf{C})$	Sylvester matrix containing C_0 and C_1
\mathbb{C}	Complex numbers
\mathbb{R}	Real numbers

Operators

$(\cdot) * (\cdot)$	Convolution Operator
$(\cdot)^*$	Complex Conjugate Operator
$(\cdot)^t$	Transposition Operator
$(\cdot)^H$	Operator for $((\cdot)^*)^t$
$\nabla(\cdot)$	Nabla Operator, Derivation Operator
$(\cdot) \odot (\cdot)$	Component by Component Product (<i>Product of Schur</i>)
$Diag(X)$	Diagonal matrix whose diagonal elements are the components for vector X
$E(\cdot)$	Statistical Expectation Operator
$FLIP(X)$	Operator turning the vector X , i.e. the first element of X comes last in $FLIP(X)$, etc.
$\mathcal{Z}\{\cdot\}$	z -Transformation Operator
$\mathcal{Z}^{-1}\{\cdot\}$	Inverse z -Transformation Operator

Abbreviations

ADSL	Asymmetric Digital Subscriber Line
AWGN	Additive White Gaussian Noise
BLMS	Block Least Mean Square
CIR	Channel Impulse Response
dB	Dezibel
DIF	Decimation in Frequency
DMT	Discrete Multitone Transmission
DSP	Digital Signal Processor
EC	Echo Cancelling
FDM	Frequency Division Multiplex
FEXT	Far-End Crosstalk
FFT	Fast Fourier Transformation
FIR	Finite Impulse Response
GI	Guard Interval
IFFT	Inverse Fast Fourier Transformation
LMS	Least Mean Square
MIMO	Multiple-Input-Multiple-Output
MMSE	Minimum Mean Square Error
NEXT	Near-End Crosstalk
OFDM	Orthogonal Frequency Division Multiplex
QAM	Quadrature Amplitude Modulation
SFB	Synthesis Filter Bank
SNR	Signal-to-Noise Ratio
TDM	Time Division Multiplex
TEQ	Time Domain Equalizer
TIR	Target Impulse Response filter
WSAF	Weighted Sub-band Adaptive Filter
WSS	Wide Sense Stationary
ZF	Zero Forcing



Institutions involved in this diploma thesis

This diploma thesis was elaborated in the framework of the *double diploma program* between the *University of Stuttgart* and the *Ecole Nationale Supérieure des Télécommunications de Paris* in cooperation with the *Motorola Center of Research CRM* in Paris. These institutions are presented in the following.

Motorola

Motorola is one of the world's leading providers of wireless communications, semiconductors and advanced electronic systems, components and services. Major equipment businesses include cellular telephone, two-way radio, paging and data communications, personal communications, automotive, defense and space electronics and computers. Motorola semiconductors power communication devices, computers and millions of other products.

Motorola maintains sales, service and manufacturing facilities throughout the world, conducts business on six continents and employs more than 139,000 people worldwide.

CRM

CRM is one of Motorola's Corporate Labs conducting research projects on a wide variety of topics of concern to Motorola, often in cooperation with other distinguished public labs in France or elsewhere in Europe. The center is quite new, having started in 1996, but it is growing rapidly.

The Lab's mission is to serve the needs of Motorola operations, particularly in Europe, by developing technology and participating in standards and regulatory developments in the region.

The CRM staff is working on various projects, involving the collaborations of researchers with expertise in the fields of Radio Technologies, Signal Processing and Speech, Telecommunication systems and Standards, Networks, Advanced Services and applications, Software and Hardware technologies, Devices and materials.

The center is located just to the south of Paris in an area close to several distinguished schools (Supelec, Ecole Polytechnique, ENST) and Universities (Orsay,...) and are able to draw on their outstanding faculty and students to help us in our work. Several cooperative projects have been started and some are under discussion, and this is expected to become a significant part of activity of the Lab.

Ecole Nationale Supérieure des Télécommunications (ENST) de Paris

To be educated at ENST is first a matter of acquiring basic knowledge in the sciences and techniques that make up the foundation of information and communications: telecommunications, electronics, computing, networks, signals, and images. It is also discovering the engineering profession and acquiring competency in economics and management as well as in communication and expression. Languages also constitute an essential part of the degree program. Students will not only have lectures but also practicals, projects, cases studies, visits of industries, internships. Strong emphasis is led on team work.

Because of the constant development of its research potential, ENST ranks among the best institutions in Europe and world-wide. In qualitative terms, the research potential of the school is represented by more than 300 full-time researchers and almost 150 doctorate students in 1992. The continual interaction between instruction and research in the laboratories of the school ensures excellence in teaching and a sound knowledge of fields that are in constant evolution. The ENST has close relations with the other "Grandes Ecoles" (France's top higher education institutes), particularly in the Group of Engineering Institutes of Paris, among them some of the most prestigious schools in France.

University of Stuttgart

The University of Stuttgart is located in the southwest of Germany which is well-known for its high-tech industry such as car manufacturing, machine tools, electronics, information and communication technology and hosts about 18,000 students and 140 institutes in 14 departments ranging from architecture, natural sciences, mathematics, computer science, history, languages, philosophy, social sciences to almost all of the engineering sciences. It is one of the oldest technical universities and has been repeatedly ranked among the very top universities in Germany. Its annual budget amounts to 500 million DM with more than 220 million DM provided for research by public and industrial sponsors. It hosts 14 centres of excellence, various technology transfer centres, four graduate research programmes, and the first federal supercomputing centre. Various large research centres (Max-Planck Institut, Fraunhofer Gesellschaft, Deutsches Zentrum für Luft- und Raumfahrt) have strong connections to the University of Stuttgart - truly a future-oriented place for research and advanced education.



Chapter 1

Introduction

The aim of this diploma thesis is to provide a simulator of a complete *Asymmetric Digital Subscriber Line* (ADSL) system in C/C++. For this, the elementary parts of an ADSL simulator were available as stand-alone modules. Moreover, equalization methods are examined and a new *Time Domain Equalization* (TEQ) algorithm is proposed and implemented into the simulator.

Examining new equalization algorithms is motivated by the fact that latest propositions (→ Jacky Chow [5, 6], Peter Chow [7], Cioffi [2], Vandendorpe [32, 33]) suffer either from slow convergence or extremely high arithmetical complexity. We intend to find a solution that improves the convergence speed considerably on the one hand with a reasonable increase of the complexity on the other hand. The general idea behind the new algorithm is the combination of Cioffi's idea (→ Cioffi [2]) of using two filters with de Courville's idea (→ de Courville [10, 11]) to introduce weighting factors in order to speed up the convergence properties.

The report is organized as follows. *Chapter 2* presents a general introduction to *Orthogonal Frequency Division Multiplex* (OFDM) systems.

Chapter 3 discusses an ADSL system in general.

Chapter 4 presents the C/C++ ADSL simulator that has been developed during this diploma thesis.

Chapter 5 discusses different equalization methods proposed by latest publications.

Chapter 6 presents the derivation of the new *Weighted Sub-band Adaptive Filter* (WSAF) Time Domain Equalization algorithm adapted to the ADSL system. An optimized structure for an implementation is proposed and the arithmetical complexity is calculated.

chapter 7 presents simulation results based on both, an ideal environment without noise and a realistic environment with noise, with an erroneous estimation of the channel impulse response, etc. The properties of the WSAF algorithm compared to the *Least Mean Square* (LMS) algorithm are discussed.

chapter 8 resumes and interprets the results.

Appendix A demonstrates why the new WSAF algorithm converges faster than the standard algorithms.

Appendix B discusses implementation problems.

Appendix C to *H* present details of the ADSL simulator.



Chapter 2

Introduction to OFDM/DMT Systems

This Chapter presents a mathematical model describing *Orthogonal Frequency Division Multiplex* (OFDM) systems. In the context of *Digital Subscriber Line* (xDSL) systems, OFDM is better known under the acronym *Discrete Multitone Transmission* (DMT). A discrete model of an OFDM/DMT system and a model for a time-discrete channel covering the effect of interference between two OFDM/DMT symbols are presented. In the end, a short discussion of the guard interval will end up this theoretical part. Afterwards, some general information concerning OFDM/DMT systems is presented.

2.1 A mathematical presentation of OFDM/DMT

In an OFDM/DMT system an incoming data stream $I(k)$ is multiplexed into several sub-streams $I_n(k) = I(k \cdot K + n)$, $0 \leq n \leq K - 1$. There are K different orthogonal *sub-carriers* $g_k(t)$ which form altogether an *orthogonal* set of *sub-carriers*:

$$\langle g_k(t - iT_s), g_{k'}(t - i'T_s) \rangle = \int_{-\infty}^{+\infty} g_k(t - iT_s) g_{k'}^*(t - i'T_s) dt \quad (2.1)$$

$$= \delta_{i,i'} \delta_{k,k'} \quad (2.2)$$

with T_s being the time duration of an OFDM/DMT symbol and $\delta_{i,i'}$ being the Kronecker symbol

$$\delta_{i,i'} = \begin{cases} 1 & \text{for } i = i' \\ 0 & \text{for } i \neq i' \end{cases} \quad (2.3)$$

Traditional systems use the following set of orthogonal sub-carriers:

$$g_k(t) = u(t) \cdot e^{j2\pi\frac{k\cdot t}{T_s}} \quad (2.4)$$

with $u(t)$ being

$$u(t) = \frac{1}{\sqrt{T_s}} \cdot r\left(\frac{t}{T_s}\right) \quad (2.5)$$

and

$$r(t) = \begin{cases} 1 & \text{for } 0 \leq t < 1 \\ 0 & \text{otherwise} \end{cases} \quad (2.6)$$

Now, we define the vector $\mathbf{S}(k)$

$$\mathbf{S}(k) = (S_0(k), \dots, S_{K-1}(k))^t \quad (2.7)$$

where $S_n(k)$ contains the information to be transmitted during an OFDM/DMT symbol of duration T_s on sub-carrier n at time k (i.e. block number k). After modulation, the time domain signal can be expressed as

$$s(t) = \sum_{k=0}^{K-1} \sum_{i \in \mathbb{Z}} S_k(i) \cdot \delta(t - iT_s) * g_k(t) \quad (2.8)$$

$$= \sum_{k=0}^{K-1} \sum_{i \in \mathbb{Z}} S_k(i) \cdot g_k(t - iT_s) \quad (2.9)$$

with “*” being the convolution operator. After low-pass filtering and sampling $s(t)$ at a sample frequency $f = \frac{1}{T} = \frac{1}{T_s/N}$ the N time domain samples $s_n(k)$ of block k are

$$s_n(k) = s(kT_s + nT), \quad 0 \leq n \leq N - 1 \quad (2.10)$$

$$\stackrel{!}{=} s[(kN + n)T] \quad (2.11)$$

$$= \sum_{m=0}^{K-1} \sum_{i \in \mathbb{Z}} S_m(i) \cdot g_m[(k - i)NT + nT]. \quad (2.12)$$

In order to prepare the z -transformation we define

$$G_m^l(z) = \sum_{n \in \mathbb{Z}} g_m[(nN + l)T]z^{-n}, \quad (2.13)$$

$$G(z) = \left[G_m^k(z) \right]_{0 \leq k \leq N-1, \leq m \leq K-1}. \quad (2.14)$$

With

$$\mathcal{Z}\{x_n\} = \sum_{k \in \mathbb{Z}} x_k z^{-k} \quad (2.15)$$

denoting the z -transformation of any series $(x_n)_{n \in \mathbb{Z}}$, it can be demonstrated (\rightarrow Courville [9]) that

$$s_n(z^N) = \sum_{m=0}^{K-1} G_m^n(z^N) S_m(z^N) \quad (2.16)$$

$$= (G_0^n(z^N), \dots, G_{K-1}^n(z^N)) \cdot \mathbf{S}(z^N) \quad (2.17)$$

with

$$\mathbf{S}(z) = \mathcal{Z}\{\mathbf{S}(n)\} \quad (2.18)$$

$$= [S_0(z), \dots, S_{N-1}(z)]^t \quad (2.19)$$

and

$$\mathbf{s}(z) = \mathcal{Z}\{\mathbf{s}(n)\} \quad (2.20)$$

$$= [s_0(z), \dots, s_{K-1}(z)]^t \quad (2.21)$$

$$= G(z) \cdot \mathbf{S}(z). \quad (2.22)$$

An addition of all samples per symbol results in (\rightarrow Courville [9])

$$s(z) = \sum_{k=0}^{N-1} z^{-k} \cdot s_k(z^N) \quad (2.23)$$

$$= (G_0(z), \dots, G_{K-1}(z)) \mathbf{S}(z^N). \quad (2.24)$$

So, we use the following expression for the digital filters of the synthesis bank in order to produce $s(z)$:

$$g_m(z) = \sum_{l=0}^{N-1} G_m^l(z^N) \cdot z^{-l}, \quad 0 \leq m \leq K-1. \quad (2.25)$$

In the literature (\rightarrow Vaidyanatha [30])

- $G(z)$ is denoted as the *polyphase matrix* associated with the *synthesis filter bank* (SFB) performing the modulation,
- $g_m(z)$ is the m th SFB filter and
- $G_l^m(z)$ is the Type-I l th polyphase component of $G_m(z)$.

As mentioned before, traditional systems use a set of orthogonal filters like

$$g_m(t) = u(t) \cdot e^{j2\pi f_m t}, \quad \{f_m = f_0 + \frac{m}{T_s}, 0 \leq m \leq K-1\} \quad (2.26)$$

with f_0 being a kind of lower bound frequency. In a baseband-model (as applicable to ADSL systems) there is $f_0 = 0$. Therefore, we obtain from (2.12) with $T_s = NT$

$$s_n(k) = \sum_{i \in \mathbb{Z}} u[(k-i)T_s + nT] \cdot \sum_{m=0}^{K-1} S_m(i) \cdot e^{j2\pi f_m((k-i)T_s + nT)} \quad (2.27)$$

$$= \sum_{i \in \mathbb{Z}} u[(k-i)N + n] T \cdot \underbrace{\sum_{m=0}^{K-1} S_m(i) \cdot e^{j2\pi \frac{nm}{N}}}_{\text{IDFT of } \sqrt{K} \cdot S_m(i)} \quad (2.28)$$

$$= \frac{1}{\sqrt{NT}} \cdot \sum_{m=0}^{K-1} S_m(k) \cdot e^{j2\pi \frac{nm}{N}} \text{ with } T_s = NT. \quad (2.29)$$

The symbol sequence $\{S_n(k)\}_n$ must be enlarged by $N_z = N - K$ zeros for the *IDFT* operation. In matrix representation, this may be written as

$$\mathbf{s}(k) = \frac{1}{\sqrt{T}} F_N^{-1} \mathbf{S}(k) \quad (2.30)$$

with

$$F_N = \frac{1}{\sqrt{N}} \left(W_N^{lk} \right)_{0 \leq l \leq N-1, 0 \leq k \leq N-1}, \quad (2.31)$$

where

$$W_N = e^{-j\frac{2\pi}{N}} \quad (2.32)$$

and

$$\mathbf{S}(k) = (S_0(k), \dots, S_{K-1}(k), \overbrace{0, \dots, 0}^{N_z=N-K})^t. \quad (2.33)$$

Now, all the necessary steps for modulating an incoming data-stream $I(k)$ are presented. The OFDM/DMT symbols to be transmitted are given by (2.30). Fig.2.1 presents the different orthogonal carriers of an OFDM/DMT symbol in the frequency domain. Since all $g_k(t)$ are rectangular functions, there is a superposition of $\frac{\sin(x)}{x}$ functions in the frequency domain. It is clear that this result must be band-limited to $\Delta f = \frac{1}{T_s}$ in order to be able to sample with $f_s \geq \frac{1}{T_s}$.

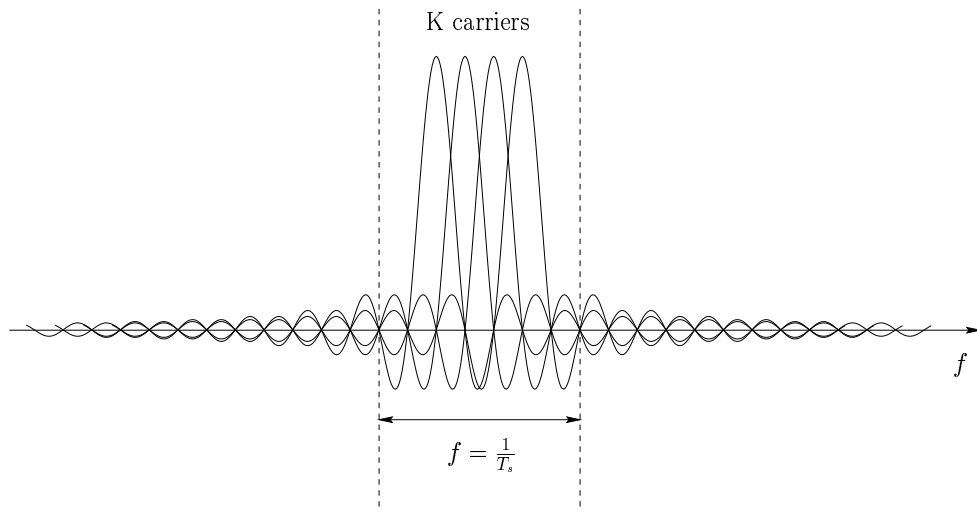


Figure 2.1: An OFDM/DMT symbol in the frequency domain.

2.2 A discrete channel model

We are going to assume that the latest OFDM/DMT symbol will only be influenced by the previous symbol. In other words, the channel impulse response will be shorter than one OFDM/DMT symbol. So, the channel can be modeled as proposed by Fig.2.2.

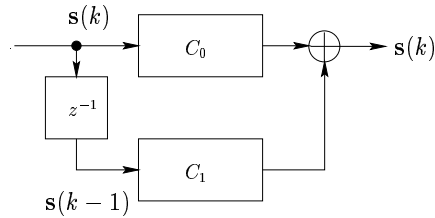


Figure 2.2: Discrete channel model.

Using a matrix representation and a channel with $M = N$ coefficients

$$\mathbf{C} = (c_0, \dots, c_{N-1})^t, \quad (2.34)$$

the received vector $\mathbf{r}(k)$ can be expressed by

$$\mathbf{r}(k) = C_0(N)\mathbf{s}(k) + C_1(N)\mathbf{s}(k-1) \quad (2.35)$$

$$= [C_1(N), C_0(N)] \begin{bmatrix} \mathbf{s}(k-1) \\ \mathbf{s}(k) \end{bmatrix} \quad (2.36)$$

$$= \begin{pmatrix} \underbrace{\begin{matrix} 0 & c_{N-1} & \cdots & c_1 \\ \downarrow & \searrow & \searrow & \vdots \\ \downarrow & & \searrow & c_{N-1} \\ 0 & \rightarrow & \rightarrow & 0 \end{matrix}}_{\text{influence of the previous symbol}} & \underbrace{\begin{matrix} c_0 & 0 & \rightarrow & 0 \\ c_1 & \searrow & \searrow & \downarrow \\ \vdots & \searrow & \searrow & 0 \\ c_{N-1} & \cdots & c_1 & c_0 \end{matrix}}_{\text{influence of the latest symbol}} \end{pmatrix} \begin{bmatrix} \mathbf{s}(k-1) \\ \mathbf{s}(k) \end{bmatrix} \quad (2.37)$$

$$(2.38)$$

$$= \tau(\mathbf{C}) \begin{bmatrix} \mathbf{s}(k-1) \\ \mathbf{s}(k) \end{bmatrix}. \quad (2.39)$$

In the literature, $\tau(\mathbf{C})$ is denoted as *Sylvester matrix*.



2.3 The guard interval

The guard interval will allow to decrease or even to totally eliminate the influence of a previous OFDM/DMT symbol to the latest symbol. In order to understand the idea, let's take a look at the influence of the previous OFDM/DMT symbol to the latest symbol, as presented by Fig.2.3

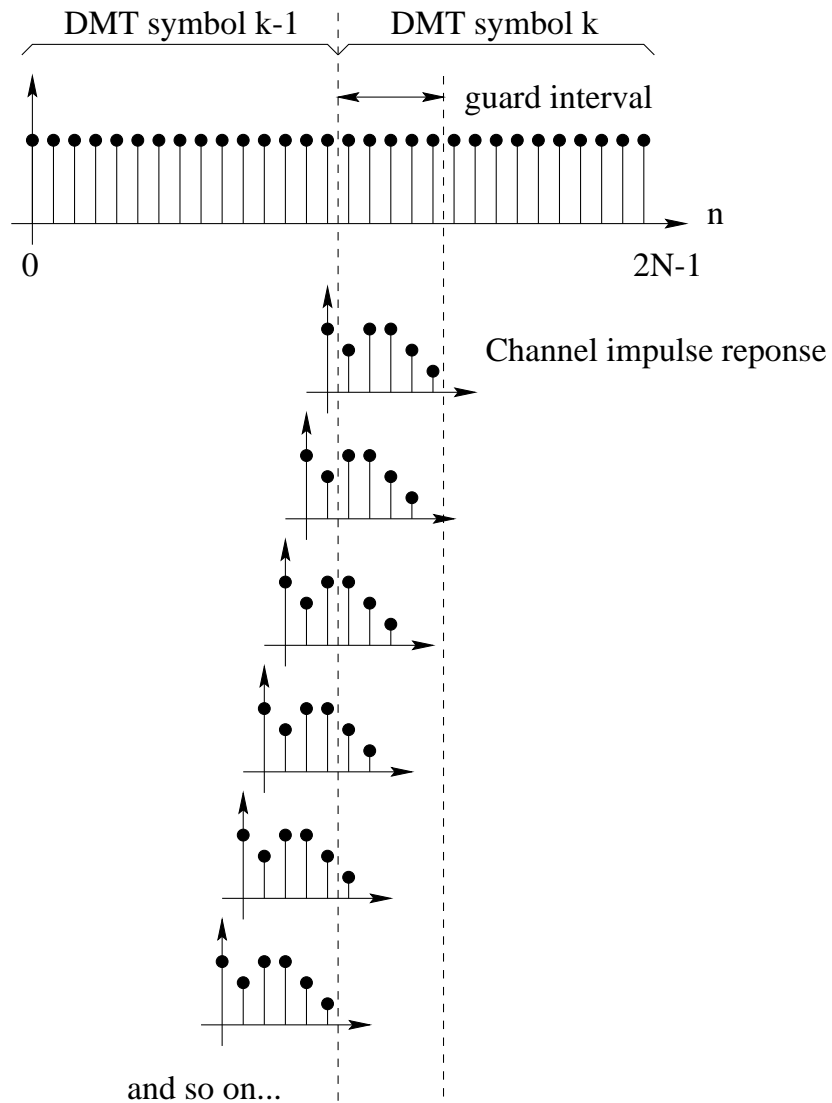


Figure 2.3: Influence of the previous DMT symbol.

Hereby, the channel impulse response shall be limited to M samples and is represented by the vector

$$\mathbf{C} = (c_0, \dots, c_{M-1}, 0, \dots, 0)^t. \quad (2.40)$$

The idea behind a guard interval is to add to a OFDM/DMT symbol

$$\mathbf{s}(k) = (s_0(k), \dots, s_{N-1}(k))^t \quad (2.41)$$

$D > M$ samples representing a cyclic prefix:

$$\mathbf{s}^{\text{ig}}(k) = (s_{N-D}(k), \dots, s_{N-1}(k), s_0(k), \dots, s_{N-1}(k))^t. \quad (2.42)$$

So, only the guard interval of one symbol is influenced by the previous symbol. The guard interval may be skipped at the reception site, since the information herein is redundant. In the end, we receive the circularly convolved data

$$\mathbf{r}(k) = C_c(N)\mathbf{s}(k) \quad (2.43)$$

where

$$C_c(N) = \begin{pmatrix} c_0 & c_{N-1} & c_{N-2} & \cdots & c_1 \\ c_1 & c_0 & \searrow & \searrow & \vdots \\ \vdots & \searrow & \searrow & \searrow & c_{N-2} \\ c_{N-2} & & \searrow & \searrow & c_{N-1} \\ c_{N-1} & c_{N-2} & \cdots & c_1 & c_0 \end{pmatrix} \quad (2.44)$$

$$= C_0(N) + C_1(N). \quad (2.45)$$

The reconstructed vector after the FFT at the reception site is (\rightarrow chapter 6.3 discusses the calculation of correlations/convolutions in the frequency domain)

$$\mathbf{R}(k) = F_N \mathbf{r}(k) \quad (2.46)$$

$$= F_N C_c F_N^{-1} \mathbf{S}(k) \quad (2.47)$$

$$= \sqrt{N} F_N F_N^{-1} ((F_N \mathbf{C}) \odot (F_N \mathbf{s}(k))) \quad (2.48)$$

$$= \sqrt{N} \text{Diag}(F_N \mathbf{C}) \mathbf{S}(k) \quad (2.49)$$

$$= \sqrt{N} [C_0 S_0(k), \dots, C_{N-1} S_{N-1}(k)]^t \quad (2.50)$$

with

$$(C_0, \dots, C_{N-1})^t = F_N (c_0, \dots, c_{M-1}, 0, \dots, 0)^t. \quad (2.51)$$

Therefore, the transmitted information is perfectly received, beside a constant multiplication factor for each value. These coefficients must be determined by a learning sequence known to the transmitter and receiver.

2.4 A complete OFDM/DMT system

Figure Fig.2.4 presents the model of a complete OFDM/DMT system as we are going to use it in this diploma thesis (→ Courville [9]).

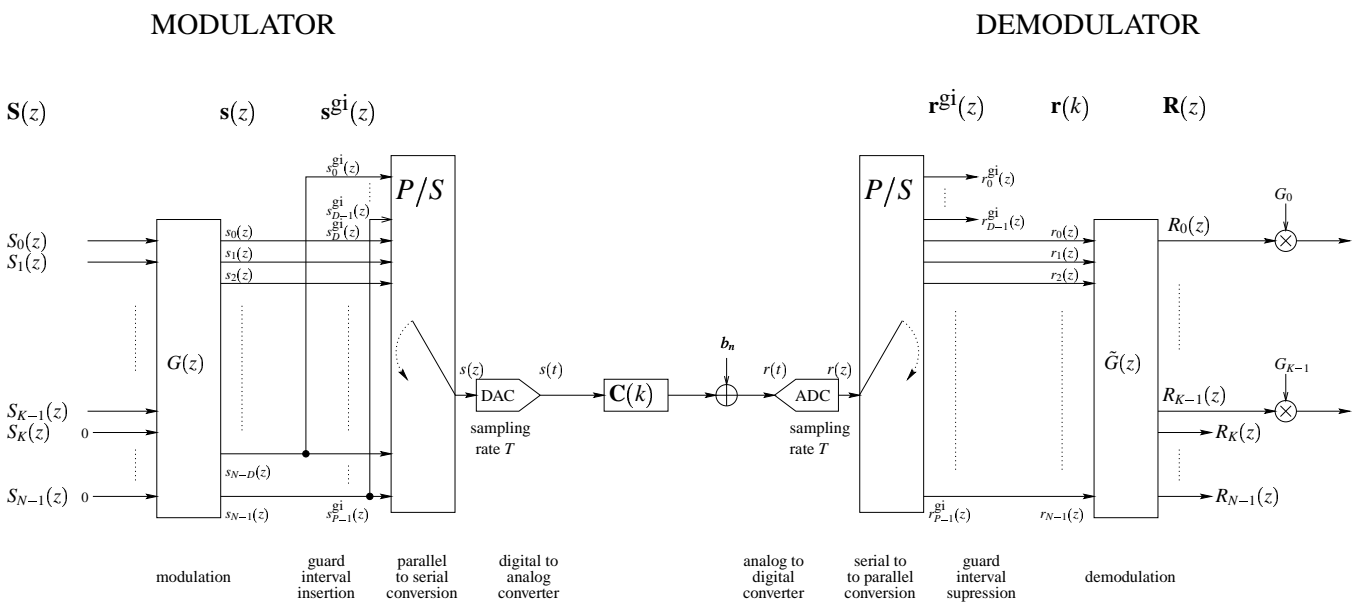


Figure 2.4: General OFDM/DMT transmission system.

The parameters used in Fig.2.4 are briefly presented:

- $\mathcal{Z}^{-1}\{\mathbf{S}(z)\} = \mathbf{S}(k) = (S_0(k), \dots, S_{K-1}(k))^t$ is a vector where $S_n(k)$ contains the information to be transmitted during a OFDM/DMT symbol duration T_s on sub-carrier n at time k (i.e. block number k).
- $G(z)$ is denoted as the *polyphase matrix* associated with the *synthesis filter bank* (SFB) performing the modulation. It can be inverted, since the filter bank is lossless: $G(z)\tilde{G}(z) = I_N$.
- $\mathbf{s}(k) = \frac{1}{\sqrt{T}}F^{-1}\mathbf{S}(k)$ is the output vector of the IDFT unit (\rightarrow (2.30)).
- $\mathbf{s}^{\text{ig}}(k) = (s_{N-D}(k), \dots, s_{N-1}(k), s_0(k), \dots, s_{N-1}(k))^t$ corresponds to $\mathbf{s}(k)$ with a cyclic prefix (guard interval) being added.
- $\mathbf{C}(k)$ describes the discrete channel and contains the coefficients of the vector $\mathbf{C} = (c_0, \dots, c_{M-1}, 0, \dots, 0)^t$.
- b_n describes the noise being added to the data after the channel.
- $\mathbf{r}^{\text{ig}}(n)$ contains the data arriving at the reception site, including the guard interval which has been corrupted by the channel.
- $\mathbf{r}(n)$ corresponds to $\mathbf{r}^{\text{ig}}(n)$ without guard interval. If the channel impulse response is shorter than the guard interval, the transmitted information can be recovered at 100% (with the assumption that no noise has been added).
- $\mathbf{R}(z) = \mathcal{Z}\{\mathbf{R}(n)\}$ corresponds to the arriving data after the DFT unit.
- $\mathbf{G}(n) = (G_0, \dots, G_{K-1})$ correspond to the equalizing coefficients in the frequency domain. They are sufficient if the channel impulse response is shorter than the guard interval (with the assumption that no noise has been added).



Chapter 3

Introduction to ADSL

In this chapter a short overview of the *Asymmetric Digital Subscriber Line* (ADSL) Metallic Interface standard (\rightarrow T1E1 [3]) is presented. Due to the limited size of this report, it contains only the most elementary information which is absolutely indispensable for the understanding of the chapters to follow. For further information the ADSL standard T1E1 [3], Chen [4], Saarela [28], the ADSL Forum [1] and Young [36] are recommended.

3.1 General description of an ADSL system

The ADSL standard (\rightarrow T1E1 [3]) defines transmission technology for simultaneous use of normal telephone services (*POTS, Plain Old Telephone Set*), data transmission of max. 6 Mbit/s in the downstream, max. 640 kbit/s in the upstream and Basic-Rate Access (BRA). In order to allow bidirectional data transmission, three possible solutions have been taken into account: *Frequency Division Multiplex* (FDM), *Time Division Multiplex* (TDM) and *Echo Cancelling* (EC), \rightarrow Saarela [28], Chen [4], ADSL standard T1E1 [3]. We are going to consider only the *Echo Cancelling* option, since it offers the best dynamic and isn't too costly due to efficient *Echo Cancelling* solution in VLSI technology.

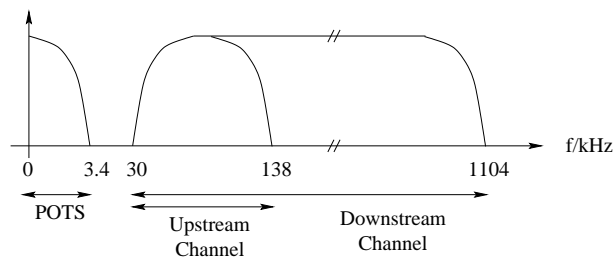


Figure 3.1: *The frequency spectrum of ADSL.*

The ADSL system reference model (\rightarrow T1E1 [3]) contains the basic blocks of an ADSL system:

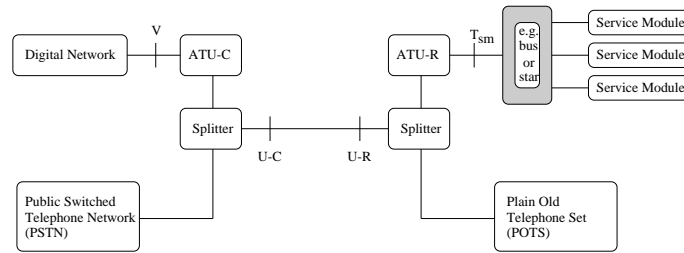


Figure 3.2: *The ADSL system reference model.*

In Fig.3.2,

- V represents the logical interface between ATU-C and a digital network element such as one or more switching systems,
- $U-C$ is the abbreviation for *loop interface - central office end*,
- $U-R$ is the abbreviation for *loop interface - terminal end*,
- $ATU-C$ is the abbreviation for *ADSL transceiver unit, central office end*,
- $ATU-R$ is the abbreviation for *ADSL transceiver unit, remote terminal end* and
- T_{sm} stands for interface(s) between $ATU-R$ and *Service Module(s)*.

3.2 Some technical details

The ADSL standard T1E1 [3] demands *Discrete Multitone* (DMT) modulation as presented in chapter 2 and works directly in the baseband. So, f_0 as defined by (2.26) is $f_0 = 0$. The downstream channels are divided in 256 4.3125-kHz wide tones, the upstream channels are divided into 32 subchannels. The frequency spectrum where the upstream channels are placed into, may also be used by the downstream channels. Therefore, we cannot use a simple bandpass filter in order to separate the upstream and downstream channels. That's why reflections caused by the hybrid circuit demand the use of an *echo cancellation unit*. Additionally, there is usually a very long channel impulse response in an ADSL system (\approx half the size of a symbol). A guard interval that is longer than the channel impulse response would largely limit the transmission rate. The solution is to introduce a linear equalizer for reducing the length of the channel impulse response. So, a guard interval longer than the resulting impulse response *channel * equalizer* will be sufficient.

In order to create real values in the time domain, the inputs of the IFFT unit at the transmission site must have *hermitian symmetry*. So, we use an IFFT unit with a *double* capacity (2×256 carriers for the downstream channel and 2×32 carriers for the upstream channels at the remote terminal end). The upper half of the carriers is defined as $Z_{i'} = (Z_{512-i'})^*$, $i = 257, \dots, 511$ (downstream channels) and $Z_{i'} = (Z_{64-i'}^*)$, $i = 33, \dots, 63$ (upstream channels).

The transceiver parameters of a DMT system are summarized in Table Tab.3.1 for the downstream channel (→ Chen [4]).

Type	Value
Symbol rate	4 kHz
FFT size	512 samples
Cyclic prefix	32 samples
Synchronization	Average 8 samples/symbol
Sampling rate	2.208 MHz
Transmit power	20 dBm
Time-Domain Equalizer	16 taps

Table 3.1: *DMT downstream parameters.*

With these parameters, the downstream DMT sub-carriers are spaced at 4.3125 kHz intervals. The lowest carrier available is at 12.938 kHz. The highest carrier available is at 1099.6875 kHz. The parameters for the upstream channel are presented by Table Tab.3.2 (→ Chen [4]).

Type	Value
Symbol rate	4 kHz
FFT size	64 samples
Cyclic prefix	4 samples
Synchronization	Average 1 samples/symbol
Sampling rate	276 kHz
Transmit power	7 dBm
Time-Domain Equalizer	32 taps

Table 3.2: *DMT upstream parameters.*

3.3 An ADSL system in detail

A whole ADSL system is presented by Fig.3.3. It has been developed using the ADSL standard [3] and Chen [4]. The different blocks are discussed in detail by chapter 4 which presents the ADSL simulator that has been developed in C/C++ during this diploma thesis.

In some blocks of Fig.3.3 there are references to the ADSL standard [3]. (6.2.1.3), for example, stands for chapter 6.2.1.3 of the ADSL standard [3].

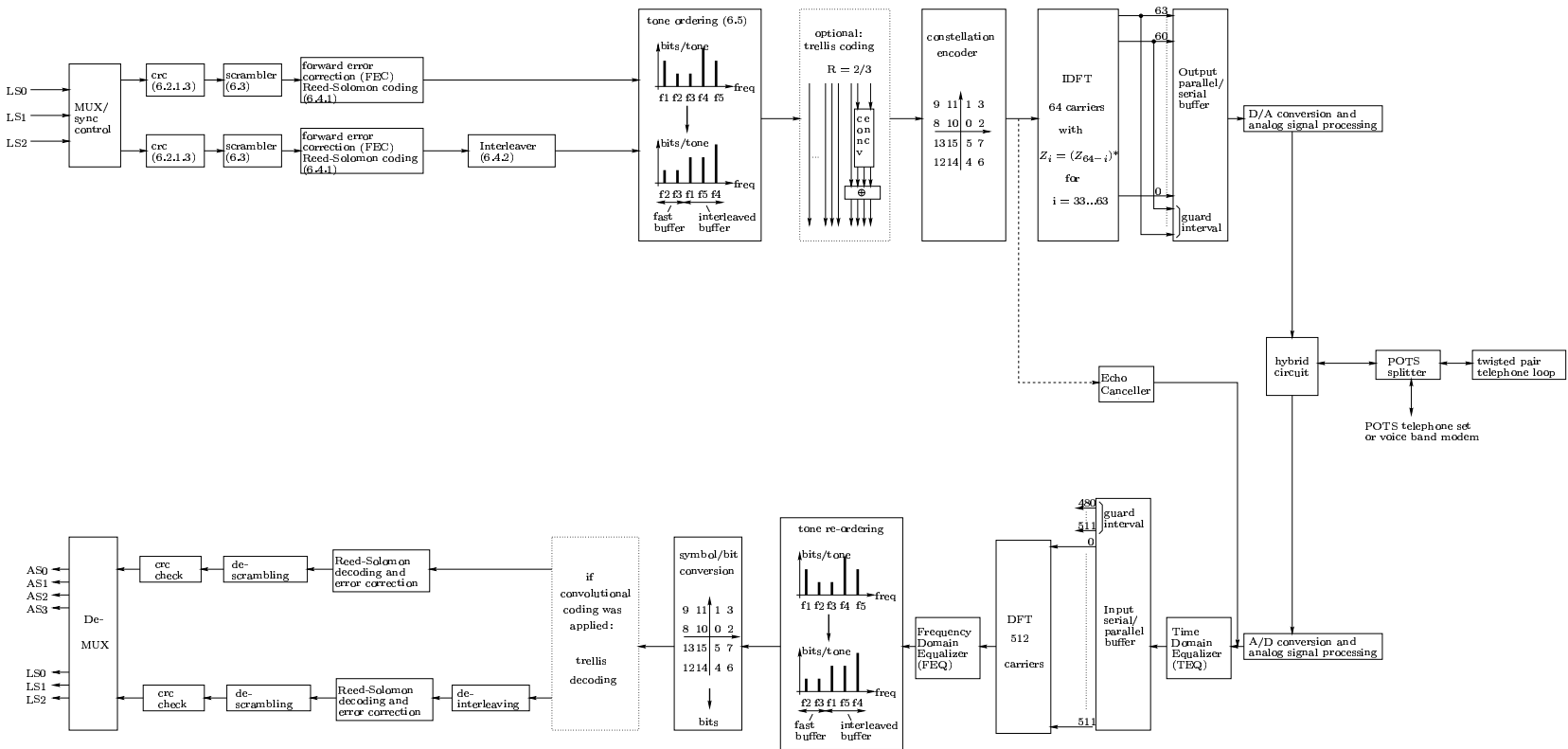


Figure 3.3: A complete ADSL transceiver, remote terminal end.



Chapter 4

The ADSL simulator in C/C++

During this diploma thesis an *Asymmetric Digital Subscriber Line* (ADSL) simulator in C/C++ has been developed. But, the system didn't have to be designed from the beginning. Some modules have been written before as *separate, stand-alone programs*. They had to be modified and - in some cases - to be corrected. The VITERBI decoder, for example, didn't work properly and quite some effort was necessary to build up a working simulator. In this chapter, the different modules of the ADSL system are briefly mentioned and the structure of the ADSL simulator is presented and discussed. A summary of all functions can be found in the appendixes C to H.

4.1 The structure of the ADSL simulator

The ADSL-SIMULATOR intends to simulate the *Time Domain Equalizer* (TEQ) training, two different kinds of transmitters (*central-office transmitter* and *home transmitter*), a channel and the *home* receiver. Therefore, the program can be divided into 5 parts (→ appendix C to H):

- the *central office* transmitter;
- the *home* transmitter;
- the equalizer training;
- the channel (for the *central-office* data);
- the *home* receiver.

4.2 The different modules of an ADSL system

The transmission part of an ADSL transceiver consists of the following modules (→ Fig.3.3):

- Serial-to-Parallel conversion and ADSL-frame builder,
- CRC generation (→ T1E1 [3] section 6.2.1.3),

- Energy scrambler (→ T1E1 [3] section 6.3),
- Forward Error Correction (FEC), Reed-Solomon coding (→ T1E1 [3] section 6.4.1),
- Interleaver (→ T1E1 [3] section 6.4.2),
- Tone ordering (→ T1E1 [3] section 6.5),
- Optional: Trellis coding (→ T1E1 [3] section 6.6),
- Constellation Encoder (→ T1E1 [3] section 6.6.4),
- Inverse Discrete Fourier Transformation (IDFT) (→ T1E1 [3] section 6.9.2, 7.9.2) and
- Adding of the guard interval (→ T1E1 [3] section 6.10, 7.10).

The reception part of an ADSL transceiver consists of (→ Fig.3.3):

- Serial-to-Parallel conversion,
- Time Domain Equalization (TEQ) (→ Chapter 6 and Chen [4]),
- Discrete Fourier Transformation (DFT),
- Frequency Domain Equalization (FEQ),
- Tone re-ordering,
- Symbol-Bit conversion (Constellation decoder),
- If convolutional coding was applied: Trellis Decoding (VITERBI decoder),
- Deinterleaving,
- Reed-Solomon decoder and error correction,
- Energy-Descrambling and
- CRC error check.

The following modules were available as *stand-alone* programs: CRC generation/check, energy scrambler/descrambler, Reed-Solomon coding/decoding with error correction, convolutional encoding, VITERBI-decoder, Discrete Fourier Transformation (DFT) and Inverse Discrete Fourier Transformation (IDFT).

Beside an adaption of these parts, the following modules had to be newly developed: The ADSL-frame builder, the Interleaver, the deinterleaver, the tone ordering, the tone re-ordering, a channel, the noise generator, the Time Domain Equalizer (using a weighted LMS algorithm, → chapter 6) and the frequency domain equalizer (using a simple multiplication with one coefficient per carrier). Additionally, the whole simulator had to be designed carefully in order to ensure portability and to allow further extensions. An *echo cancellation* is not yet implemented.

The following sections present rapidly the different modules of the ADSL simulator. For general information, the ADSL standard T1E1 [3] and Chen [4] are recommended. Detailed information concerning the simulator is presented in annex C to H.

4.2.1 The ADSL-frame builder

The ADSL standard T1E1 [3], section 6.2 describes the *framing*. A superframe structure is presented, containing *68 frames* and *one synchronization symbol* and having a periode of $250\mu\text{sec}$. One *frame* is placed into one DMT symbol and is split up into a *fast data buffer* and a *interleaved data buffer*. The *fast data buffer* is not interleaved and will be used for delay-sensitive applications. The *interleaved data buffer* will be used for video-on-demand services, for example. In each buffer there are up to four simplex channels *ASX* (*AS0, AS1, AS2, AS3*) and three duplex channels *LSX* (*LS0, LS1, LS2*).

4.2.2 CRC generation and CRC check

At the transmission site, a *cyclic redundancy check* (CRC) codeword is calculated for each DMT frame. So, the receiver can easily determine whether a transmission error occurred or not. The error itself cannot be corrected by this codeword.

4.2.3 Energy scrambler and descrambler

Energy scrambling is used in order to obtain an energy distribution without any energy peaks in the ideal case.

4.2.4 Reed-Solomon encoder and decoder

The size of a Reed-Solomon codeword is $n = r + k$ where r is the degree of the generator polynomial, n is the codeword size depending on the number of bits assigned to either fast or interleaved buffer and k is the message sequence size. For Reed-Solomon codes the minimum distance d_{min} is $d_{min} = r + 1$. The number of errors that can be corrected is therefore $t = \frac{d_{min}-1}{2} = \frac{r}{2}$ (\rightarrow Chen [4], Lin [23]). In the ADSL standard there are the following constellations defined: 16 FEC redundancy bytes for 194 interleaved data bytes, 12 FEC redundancy bytes for 146 interleaved data bytes, 16 FEC redundancy bytes for 196 interleaved data bytes or 16 FEC redundancy bytes for 192 interleaved data bytes.

4.2.5 Interleaver and deinterleaver

Usually, the OFDM/DMT systems do not perform better than single carrier modulation systems if used *without any coding or frequency interleaving*. Since a given frequency or the whole spectrum are not likely to be strongly attenuated by a channel fading during a long period of time, the symbols are transmitted at *different times and frequencies*. So, a small number of them can *simultaneously* be degraded by fading. This reordering is performed by the *interleaver*. The performance of the system is largely increased (\rightarrow Courville [9]). The *interleaver* is implemented in a circular shift register, *deinterleaver* is implemented using a circular buffer.

4.2.6 Tone ordering and re-ordering

The *tone ordering algorithm* assigns to the different carriers first the *fast buffer data* beginning with the carriers with the smallest number of bits assigned to them. Then, the *interleaved buffer data* is assigned the remaining carriers. Again, the carriers with the smallest number of bits assigned to them are used at first. The *tone re-ordering algorithm* has to re-establish the correct data order.

4.2.7 Trellis coding

The ADSL standard T1E1 [3] defines a 16-state 4-dimensional trellis code (*Wei's encoder*). The coding rate is $R = \frac{2}{3}$. With the added redundancy, the *minimum distance* between two constellation points is improved (\rightarrow Proakis [27]).

4.2.8 VITERBI decoding

The *VITERBI algorithm* allows a *maximum likelihood decoding* (\rightarrow Min [24], Proakis [27]). The sequence will be found that has most probably been sent.

4.2.9 Constellation encoder and decoder

The *constellation encoder* selects for each carrier an odd-integer point (X, Y) from the square-grid constellation based on b bits that have to be transmitted. It is distinguished between *odd* and *even* values of b (ADSL standard T1E1 [3]). The *constellation decoder* converts the received odd-integer points (X, Y) into the originally sent data.

4.2.10 DFT and IDFT

The *Discrete Fourier Transformation* (DFT) and the *Inverse Discrete Fourier Transformation* (IDFT) is performed using the *Fast Fourier Transformation* (FFT) algorithm.

4.2.11 Time Domain Equalizer (TEQ)

In the classical case (\rightarrow Chen [4]), the *Time Domain Equalizer* (TEQ) is implemented using a *Target Impulse Response* (TIR) filter as proposed by chapter 5.2. So, after truncating the guard interval, the arriving symbol is supposed not to be influenced by any previous symbols. The coefficients of the filters are determined during the *learning sequence*.

4.2.12 Frequency Domain Equalizer (FEQ)

In the classical case (\rightarrow Chen [4]), the frequency domain equalizer is implemented using a multiplication of each carrier with a coefficient. The coefficients are determined during the *learning sequence*.



4.2.13 Transmission channel

For the transmission channel, a *discrete model* is used as proposed by chapter 2.2.

4.3 Description of the ADSL simulator program design

This section describes the design of the ADSL simulator program design by presenting the following flowcharts:

- Flowchart of the main function,
- Flowchart of the *central office* transmitter, part 1 (preparation of one superframe),
- Flowchart of the *central office* transmitter, part 2 (preparation of one DMT symbol),
- Flowchart of the *remote terminal end* receiver, part 1 (reception of one DMT symbol),
- Flowchart of the *remote terminal end* receiver, part 2 (decoding of one superframe),
- Flowchart of the *channel*.

The *Time Domain Equalizer* (TEQ) will be discussed in detail by chapter 6.

4.3.1 Flowchart of the main function

Fig.4.1 presents the flowchart of the main function of the ADSL simulator. Here, the functions for *preparing a superframe for transmission* (→ Fig.4.2), for *preparing one DMT symbol for transmission* (→ Fig.4.3), for *receiving one DMT symbol at the reception site* (→ Fig.4.4), for *treating one superframe at the reception site* (→ Fig.4.5) and for *emulating the channel* are called. In the end, the received and transmitted data are compared and the number of errors is counted.

4.3.2 Flowchart of the transmitter functions

The transmitter modules are placed into two different functions. The function *transmitter_CO_part1* (→ Fig.4.2) contains all operations applied to the whole superframe and *transmitter_CO_part2* (→ Fig.4.3) all operations that are applied to the different DMT symbols separately.

4.3.3 Flowchart of the receiver functions

The receiver modules are placed into two different functions. The function *receiver_home_part1* (→ Fig.4.4) contains all operations applied to the different DMT symbols separately and *receiver_home_part2* (→ Fig.4.5) all operations that are applied to the whole superframe.

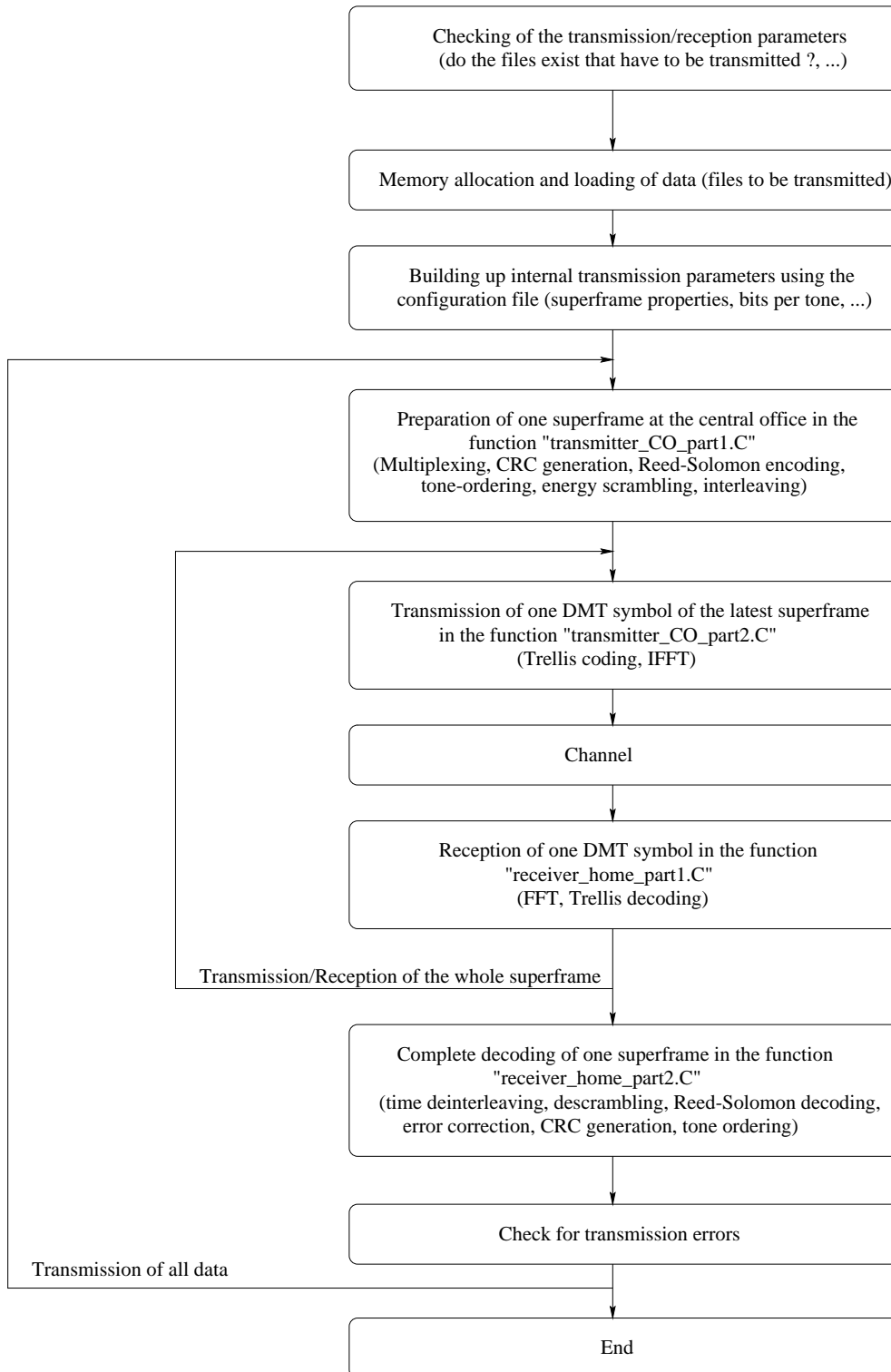


Figure 4.1: The main function of the ADSL simulator.

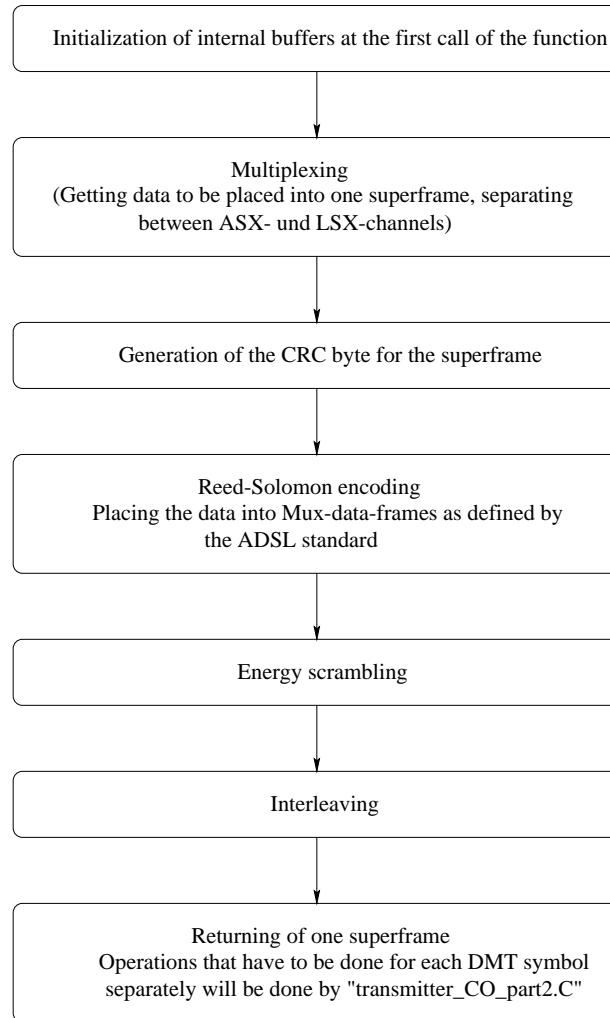


Figure 4.2: *The function preparing one superframe for transmission.*

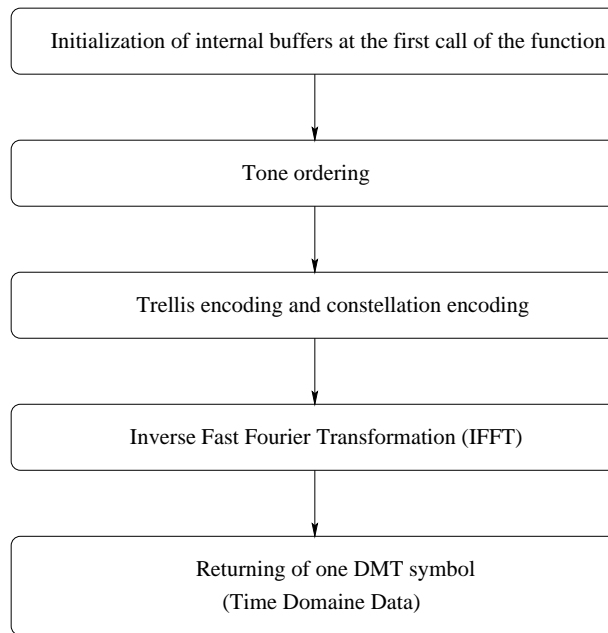


Figure 4.3: *The function preparing one DMT symbol for transmission.*

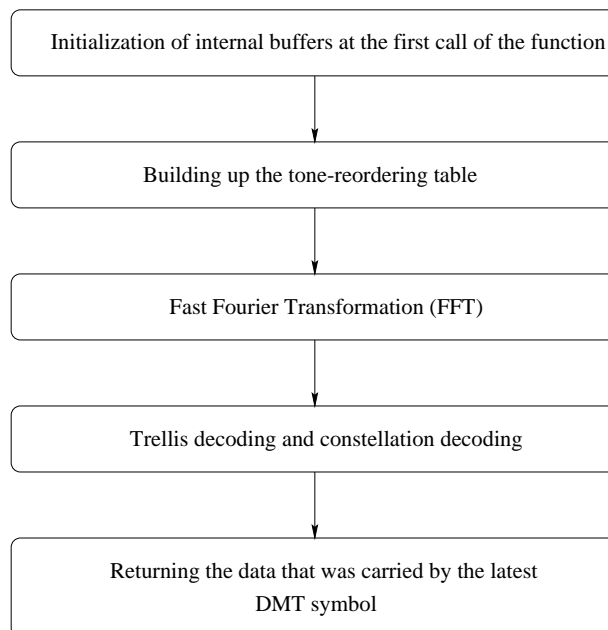


Figure 4.4: *The function receiving one DMT symbol.*

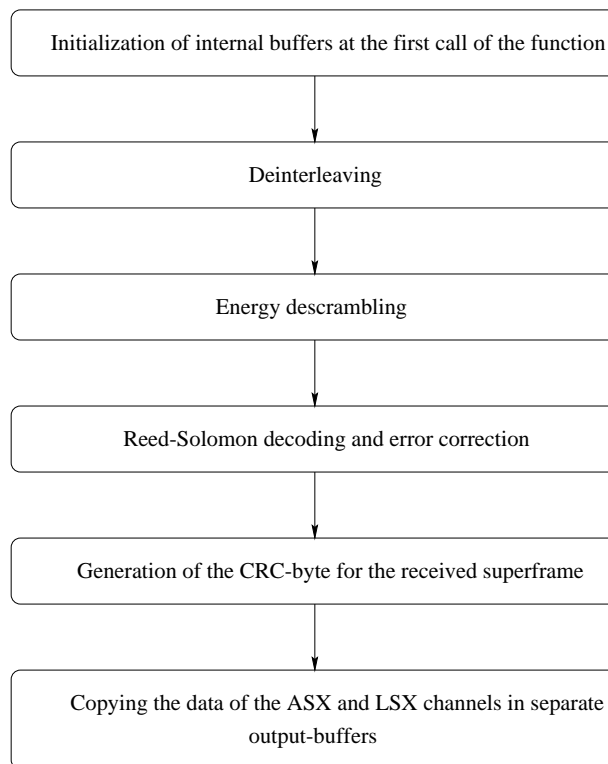


Figure 4.5: *The function treating one superframe at the reception site.*

Chapter 5

Equalization methods for OFDM/DMT systems

This chapter presents the different equalization methods for OFDM/DMT proposed by the literature and latest publications. The advantages and disadvantages of the different propositions are presented and discussed.

5.1 Equalization using a multiplication in the frequency domain

The equalization can be done using a multiplication of each carrier with a complex coefficient depending on the channel when the guard interval is longer than the memory of the channel. Fig.5.1 explains why.

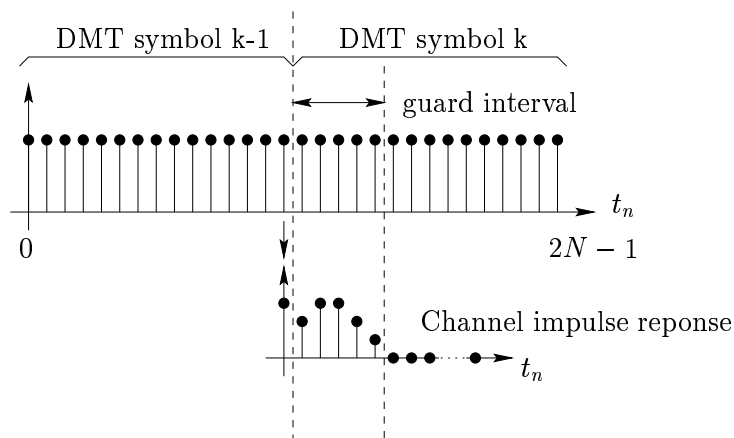


Figure 5.1: A guard interval larger than the memory of the channel.

With the guard interval being shorter than the memory of the channel, the distortion provoked by a OFDM/DMT symbol $DMT(k-1)$ in the OFDM/DMT symbol $DMT(k)$ does only effect the guard interval of $DMT(k)$.

This interval contains redundant information (it's a cyclic extension of the corresponding OFDM/DMT symbol) and is truncated at the reception site as demonstrated in chapter 2.3.

Depending on the data sent at the transmission site (\rightarrow (2.7))

$$\mathbf{S}(k) = (S_0(k), \dots, S_{K-1}(k))^t \quad (5.1)$$

and the channel coefficients described by the vector

$$\mathbf{C} = (c_0, \dots, c_{M-1}, 0, \dots, 0)^t, \quad (5.2)$$

the received information $\mathbf{R}(k)$ is (\rightarrow (2.50))

$$\mathbf{R}(k) = \sqrt{N} \text{Diag}(F_N \mathbf{C}) \mathbf{S}(k) \quad (5.3)$$

$$= \sqrt{N} [C_0 S_0(k), \dots, C_{N-1} S_{N-1}(k)]^t. \quad (5.4)$$

A simple multiplication of each output of the DFT unit at the transmission site with a constant coefficient is sufficient for the equalization. Desiring a *Zero Forcing (ZF)* equalization, the equalizing coefficients are

$$G_i^{ZF} = \frac{1}{C_i}, \quad (5.5)$$

in the case of a *Minimum Mean Square Error (MMSE)* equalization, the coefficients are

$$G_i^{MMSE} = \frac{C_i^*}{|C_i|^2 + \sigma_{B_i}^2 / \sigma_s^2} \quad (5.6)$$

with $\mathbf{B}(k) = F_N \mathbf{b}(k)$ (\rightarrow de Courville [11]) where $\mathbf{b}(k)$ are the noise samples. Hereby, $\sigma_{B_i}^2$ is the variance of the *Additive White Gaussian Noise (AWGN)*, σ_s^2 is the variance of the transmitted symbols.

5.2 Equalization using a Target Impulse Response (TIR) filter

In this section we are going to discuss the case of a channel impulse response being longer than the guard interval. It will be explained why this problem can be resolved with a combination of a *Target Impulse Response* (TIR) filter and a *Time Domain Equalization* (TEQ) filter (\rightarrow Chow [5]). The approach presented here requires a learning phase in order to train the two filters. The TIR filter will only be used during that training sequence. Afterwards, the resulting *Time Domain Equalizer* (TEQ) will allow to confine the channel impulse response.

In the OFDM/DMT context, the classical idea of minimizing the *Mean Square Error* (MSE)

$$E [|\epsilon_n|^2] = E [|I_n - \tilde{I}_n|^2] \stackrel{!}{=} \min, \quad (5.7)$$

with I_n representing the transmitted data and \tilde{I}_n the received data, isn't the best solution. For a channel impulse response which is longer than the guard interval, it will be sufficient to shorten the channel impulse response to a size smaller than the guard interval. Then, the remaining error can be easily corrected by a frequency domain equalization with a multiplication as explained in chapter 5.1.

In other words, our Time Domain Equalizer must *not* correct the received data in a way that

$$I_n \approx \tilde{I}_n * e_{classic,n}, \quad (5.8)$$

with $e_{classic,n}$ being the time discrete impulse response of the classical Time Domain Equalizer. The Time Domain Equalizer in our OFDM/DMT system will have the optimal coefficients when the equalized data

$$I_{equalized,n} = \tilde{I}_n * e_{DMT,n} \quad (5.9)$$

correspond to the result of the original data I_n convolved with *any* channel impulse response $e_{any,n}$ which is *shorter than the guard interval* and constant:

$$I_{equalized,n} = \tilde{I}_n * e_{any,n}. \quad (5.10)$$

The remaining distortion can be corrected by a multiplication of each carrier with a constant coefficient which must be calculated during a learning process (\rightarrow chapter 5.1).

The general idea of the approach with a TIR-Filter is demonstrated by Fig.5.2.

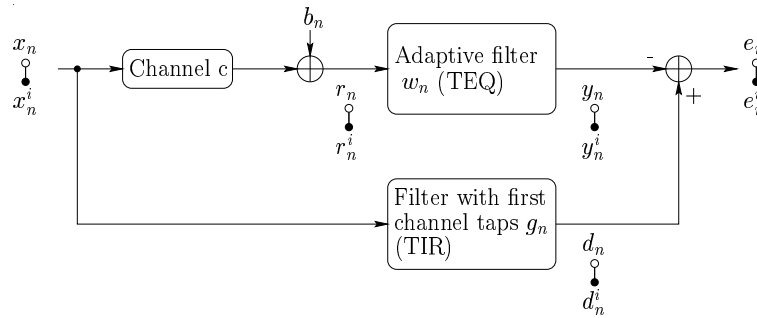


Figure 5.2: Equalization using a Target Impulse Response (TIR) filter.

Chow [7] has demonstrated that the mean square error R_{ee} is

$$R_{ee} = E[e(k)e^*(k)] \quad (5.11)$$

$$= g^* \cdot (R_{xx} - R_{xr}R_{rr}^{-1}R_{rx}) \cdot g \quad (5.12)$$

$$= g^* \cdot R_{x|r} \cdot g. \quad (5.13)$$

Hereby,

$$r_n = x_n * c_n + b_n \quad (5.14)$$

with b_n being the samples of the added noise. Now, the optimal TIR coefficients g must be found. This results in an eigenvalue problem which is quite costly in computation time. The time equalizer coefficients are then updated using

$$w = g^* \cdot R_{xr} \cdot R_{rr}^{-1} \quad (5.15)$$

as also shown by Chow [7].

5.3 Equalization in the time domain without guard interval

Vandendorpe [32] and [33] has analyzed this problem. His aim is to determine an optimal prediction \tilde{I}_n of the transmitted data I_n in the time domain using a separate matched filter and a *Multiple-Input-Multiple-Output* (MIMO) equalization filter for each used carrier.

The transmitted signal is

$$x(t) = \sqrt{\frac{2 \cdot P_{DMT}}{N}} \cdot \sum_{p=0}^N \sum_{n=-\infty}^{\infty} I_n^p \cdot u(t - nT) \cdot e^{j2\pi \cdot \frac{p \cdot t}{T}}, \quad (5.16)$$

with I_n^p being the n th symbol conveyed by carrier p , $P_{DMT} = \frac{N \cdot E_b}{T}$ and E_b being the energy per bit. $u(t)$ is the symbol shape which is assumed to be rectangular in our case. Having a linear channel $c(t)$ the received signal is

$$r(t) = \sqrt{\frac{2 \cdot P_{DMT}}{N}} \cdot \sum_{p=0}^N \sum_{n=-\infty}^{\infty} I_n^p \cdot \left(u(t - nT) \cdot e^{j2\pi \cdot \frac{p \cdot t}{T}} \right) * c(t) + \quad (5.17)$$

$$\text{noise} \quad (5.18)$$

$$= \sqrt{\frac{2 \cdot P_{DMT}}{N}} \cdot \sum_{p=0}^N \sum_{n=-\infty}^{\infty} I_n^p \cdot h_p(t - nT) + \text{noise}. \quad (5.19)$$

Using a different matched filter $h_p^*(t - nT)$ for each carrier p , the matched filter outputs are

$$y_p^n = \frac{1}{T \cdot \sqrt{2 \cdot P_{DMT}}} \cdot \int_{-\infty}^{\infty} r(t) \cdot h_p^*(t - nT) dt. \quad (5.20)$$

Using an equalizer with $2 \cdot K + 1$ coefficients for each matched filter output, the estimated sequence is

$$\tilde{I}_n^q = \sum_{p=0}^{N-1} \sum_{m=-K}^K c_{q,p}^m \cdot y_p^{n-m}. \quad (5.21)$$

The $N \times N \times (2 \cdot K + 1)$ filter coefficients can be calculated by using the orthogonal principle (\rightarrow Moulines, Boutros [25])

$$E \left[\epsilon_q^k \cdot (y_p^{k-l})^* \right] = 0 \quad (5.22)$$

where

$$\epsilon_q^k = \tilde{I}_q^n - I_q^n. \quad (5.23)$$

This results in a linear equation system with $N \times N \times (2 \cdot K + 1)$ equations. It must be solved during the learning procedure. Afterwards, there is still a large number of multiplications to be performed with every arriving OFDM/DMT symbol. Vandendorpe [32] uses for his examples a OFDM/DMT symbol with $N = 4$ carriers. Hereby, a system with $N \times N \times (2 \cdot K + 1)$ different equations can be easily solved, compared to the number of equations in the case of an ADSL OFDM/DMT-symbol with $N = 256$ different carriers.

5.4 Discussion

The most simple solution for a OFDM/DMT system is to choose a guard interval that is larger than the channel impulse response. In this case, equalizing with a simple multiplication with a coefficient for each carrier in the frequency domain can be applied (\rightarrow chapter 5.1). Unfortunately, this method limits the transmission speed considerably. Since the guard interval is shorter than the channel impulse response for the ADSL system, this method is not suitable.

Equalization using a Target Impulse Response (TIR) filter (\rightarrow chapter 5.2) is the classical method for ADSL systems. The coefficients are determined during a learning sequence where the transmitted data are already known to the receiver. Even if the calculation of the coefficients might be costly in computation time (Chow's solution results in a costly eigenvalue problem), after having determined these values, no further update is necessary and there is a simple FIR filter with a small number of taps at the reception site.

Vandendorpe's solution (\rightarrow chapter 5.3) is interesting, but too costly for systems with a large number of carriers. It may be used in future when DSPs become more powerful.

In the end, the classical solution with a *Target Impulse Response (TIR) filter* seems to be the most promising idea.

Chapter 6

A new algorithm updating the Time Domain Equalizer

This chapter presents the derivation of the update algorithm for the *Time Domain Equalization* (TEQ) filter using the *Weighted Sub-band Adaptive Filter* (WSAF) algorithm (\rightarrow de Courville [10]). With the WSAF algorithm we use our knowledge about the mean energy of the different orthogonal carriers in order to speed up the convergence properties (\rightarrow appendix A).

6.1 The filter structure

Fig.6.1 presents the general idea of a system using an adaptive TEQ filter and a TIR filter containing always the first taps of the latest estimation *Channel Impulse Response* (CIR) * *TEQ*.

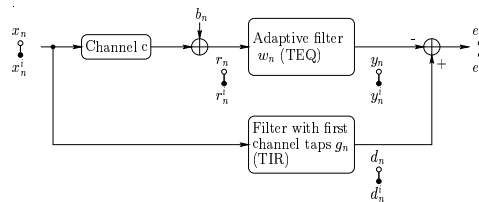


Figure 6.1: The TEQ and TIR filters.

Based on the idea of Cioffi [2], we are going to use a *Target Impulse Response* (TIR) filter whose impulse response is updated after each iteration step with the first taps of the *Channel Impulse Response* (CIR) of the ADSL channel convolved with the latest TEQ filter coefficients. The number of taps for the TIR filter will correspond to the length of the guard interval of the *Discrete Multi-Tone* (DMT) symbols. The classical approach adapts the TEQ filter by using the square of the resulting impulse response $CIR * TEQ$ after the guard interval as the criterion to be minimized (*Least Mean Square* (LMS) algorithm), where “*” denotes the convolution operator. We are taking up the proposition of de Courville [10] to introduce additionally a *weighting factor* for the square errors in each sub-band (*Weighted Sub-band Adaptive Filter* (WSAF), \rightarrow appendix A).

Hereby,

- b_n is the noise added to the samples after the channel.
- c_n are the channel coefficients.
- d_n are the transmitted samples convolved with the TIR filter.
- e_n is the difference between the samples d_n and y_n , called *error*.
- g_n are the TIR filter coefficients.
- r_n are the received samples.
- w_n are the TEQ filter coefficients.
- x_n are the transmitted samples.
- y_n are the received samples convolved with the TEQ filter.
- $(\cdot)^i$ denotes the corresponding samples in the frequency domain.

During the *learning sequence* the TEQ filter will be adapted by using the weighted square error

$$\hat{J} = \sum_{i=0}^{\frac{N}{2}-1} \lambda_i \cdot |d_n^i - y_n^i|^2 \quad (6.1)$$

as the criterion to be minimized.

We calculate the sum from $i = 0$ up to $i = \frac{N}{2} - 1$, i.e. only half the energy spectrum is used for the error calculation. This is reasonable, since the ADSL standard [3] demands hermitian symmetry in the frequency domain in order to create *real* time domain values (\rightarrow chapter 3):

$$Z_i = (Z_{N-i})^*, i = \frac{N}{2} + 1, \dots, N - 1. \quad (6.2)$$

We are going to see later in this chapter, that a summation up to $i = N$ just limits the solution space to real numbers. So, the calculation complexity can be optimized.

In the ideal case, the resulting impulse response $c_n * w_n$ will be shorter than the guard interval of the DMT symbol. In reality, the energy of the resulting impulse response will be concentrated in the first taps, but the remaining taps will still not be exactly zero (\rightarrow Fig.6.2).

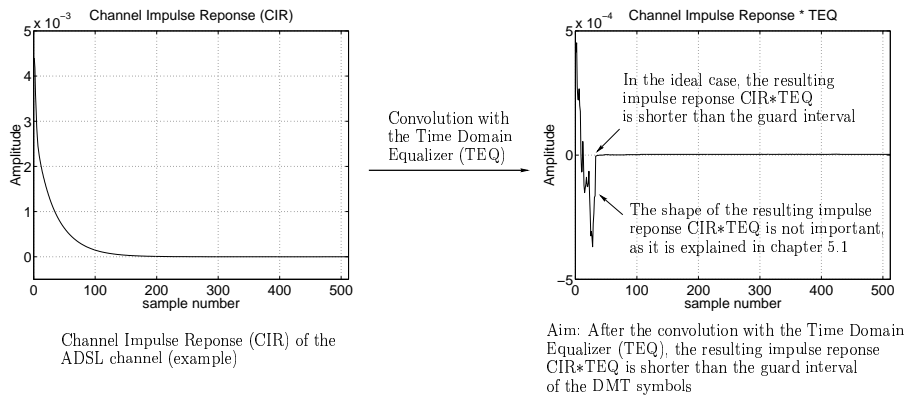


Figure 6.2: Shortening the Channel Impulse Response (CIR) by a Time Domain Equalizer (TEQ).

6.2 Choosing the adaptive step size

As discussed by Widrow [34], under white noise input the fastest convergence will be achieved for the *Least Mean Square* (LMS) algorithm by choosing the adaptive step size

$$\mu_i = \frac{1}{L \cdot \sigma_{r_i}^2}. \quad (6.3)$$

In the case of the WSAF algorithm, we obtain:

$$\mu \cdot \lambda_i = \frac{1}{L \cdot \sigma_{r_i}^2}. \quad (6.4)$$

λ_i are the weighting factors used in order to speed up the convergence properties (\rightarrow appendix A), the constant factor μ may be used in order to obtain various tradeoffs between convergence rate and residual error, L corresponds to the number of taps of the TEQ filter and $\sigma_{r_i}^2$ to the variance of the received signal. In fact, there are many LMS algorithms working separately in each sub-band.

During the learning sequence, always the same DMT symbol x_n , $n \in (0, 1, \dots, N - 1)$ is transmitted. Receiving $r_n = x_n * c_n$, we choose the weighting factors for each sub-band separately:

$$\mu \cdot \lambda_i = \frac{1}{L \cdot |x_n^i \cdot c_n^i|^2}. \quad (6.5)$$

Here, we need an estimation c_n^i of the ADSL channel in the frequency domain. Since no guard interval is used during the learning sequence and always the same symbol is transmitted, the received symbols are actually *circularly convolved*. In other words, we can interpret the previous DMT symbol as guard interval of the latest DMT symbol.

Assuming the channel impulse response being shorter than one DMT symbol, it can be easily calculated in the frequency domain:

$$c_n^i = \frac{r_n^i}{x_n^i}. \quad (6.6)$$

The first taps of the channel impulse response in the time domain $c_n = IFFT(c_n^i)$ are used in order to initialize the TIR filter.

6.3 The update algorithm for the filter taps

At first, we are going to introduce some definitions. They are similar to the ones used by de Courville [10, 11].

- N is the number of samples per DMT symbol. At the same time, this is the number of inputs to the FFT (if no guard interval is used, as it is the case during the learning sequence).
- L is the number of taps of the *Time Domain Equalization* (TEQ) filter.
- P is the number of taps of the *Target Impulse Response* (TIR) filter.
- $R_n = (r_n, r_{n-1}, \dots, r_{n-N+1})^t$ contains the samples arriving at the receiver, i.e. the transmitted samples convolved by the channel and disturbed by noise b_n .
- $\mathcal{R}_n = (R_n, R_{n-1}, \dots, R_{n-L+1})$ contains the last L vectors of the arriving samples.
- $W_n = (w_0(n), w_1(n), \dots, w_{L-1}(n))^t$ contains the L TEQ filter coefficients.
- $X_n = (x_n, x_{n-1}, \dots, x_{n-N+1})^t$ contains the originally transmitted samples.
- $\mathcal{X}_n = (X_n, X_{n-1}, \dots, X_{n-P+1})$ contains the last P vectors of the transmitted samples.
- $G_n = (g_0(n), g_1(n), \dots, g_{P-1}(n))^t$ contains the P TIR filter coefficients.
- $X_n^i = F_{N,i} \cdot \mathcal{X}_n = (x_n^i, x_{n-1}^i, \dots, x_{n-P+1}^i)^t$ contains the i th carrier of the transmitted information in the frequency domain.
- $R_n^i = F_{N,i} \cdot \mathcal{R}_n = (r_n^i, r_{n-1}^i, \dots, r_{n-L+1}^i)^t$ contains the i th carrier of the received samples in the frequency domain.
- $r(n) = (r_0(n), r_1(n), \dots, r_{N-1}(n))^t = R_{nN}$ contains the received samples.
- $E_n = (e_n, e_{n-1}, \dots, e_{n-N+1})^t = \mathcal{X} \cdot G - \mathcal{R} \cdot W$ contains the error between the transmitted samples convolved by the channel and the TEQ and the transmitted samples convolved by the TIR filter.
- $E_n^i = (e_n^i, e_{n-1}^i, \dots, e_{n-N+1}^i)^t = F_{N,i} \cdot E_{kN}$ contains the i th carrier of the error in the frequency domain.

- $F_{N,i}$ is the i th line of the matrix $F_N = \frac{1}{\sqrt{N}} \cdot (W_N^{lk})$ performing the discrete fourier transformation with $W_N^{lk} = e^{-j\frac{2\pi}{N}lk}$ and $0 \leq l \leq N-1, 0 \leq k \leq N-1$.
Note: In our representation, the last symbol appears as first entry in the corresponding matrix (example: $R_n = (r_n, r_{n-1}, \dots, r_{n-N+1})^t$, where r_n was the last received symbol and r_{n-N+1} the first received symbol in the vector). The usual FFT and IFFT implementations, however, expect the order of the samples vice versa. This will slightly influence the implementation presented in this chapter and will therefore be discussed by section 2.4.
- $F_{N \times \frac{N}{2}}$ contains the first $\frac{N}{2}$ lines of F_N .
- k is the latest iteration step number.

Now we are going to calculate the derivate $\frac{\partial \hat{J}}{\partial W_{kN}^*}$ of

$$\hat{J} = \sum_{i=0}^{\frac{N}{2}-1} \lambda_i \cdot |e_k^i|^2. \quad (6.7)$$

We obtain

$$\frac{\partial \hat{J}}{\partial W_{kN}^*} = 2 \cdot \sum_{i=0}^{\frac{N}{2}-1} \lambda_i \cdot e_k^i \left(\frac{\partial e_k^i}{\partial W_{kN}^*} \right)^* \quad (6.8)$$

with

$$\frac{\partial e_k^i}{\partial W_{kN}^*} = \frac{\partial}{\partial W_{kN}^*} (F_{N,i} \mathcal{A}_{kN} G_{kN} - F_{N,i} \mathcal{R}_{kN} W_{kN}) \quad (6.9)$$

$$= -F_{N,i} \mathcal{R}_{kN} \quad (6.10)$$

$$= -R_{kN}^i. \quad (6.11)$$

This results in

$$\frac{\partial \hat{J}}{\partial W_{kN}^*} = -2 \cdot \sum_{i=0}^{\frac{N}{2}-1} \lambda_i \cdot e_k^i (R_{kN}^i)^*. \quad (6.12)$$

Here, it can be easily seen why only the $\frac{N}{2}$ first carriers are taken into account by the criterion \hat{J} . The reason lies in the fact that the ADSL standard [3] demands hermitian symmetry in the frequency domain in order to create *real* time domain values, as it was already discussed by chapter 6.1:

$$Z_i = (Z_{N-i})^*, i = \frac{N}{2} + 1, \dots, N-1. \quad (6.13)$$



Using all N carriers, there would be for each $\lambda_i \cdot e_k^i, 1 \leq i \leq 255$ a complex conjugate of the same value among $(\lambda_i \cdot e_k^i)^*, 257 \leq i \leq 511$. Likewise, for each $R_{kN}^i, 1 \leq i \leq 255$ there would be a complex conjugate of the same value among $(R_{kN}^i)^*, 257 \leq i \leq 511$. So, in reality $2 \cdot \text{Re} \{ \lambda_i \cdot e_k^i (R_{kN}^i)^* \}$ is minimized over $i = 0$ up to $i = \frac{N}{2} - 1$. It is sufficient to take only half the spectrum into account. Then, the imaginary part of this result is truncated and the remaining real part is multiplied with 2.

In the end, the tap update is

$$W_{(k+1)N} = W_{kN} - \mu \cdot \Delta W_{kN} \quad (6.14)$$

$$= W_{kN} - \mu \cdot \frac{1}{2} \frac{\partial \hat{J}}{\partial W_{kN}^*} \quad (6.15)$$

$$= W_{kN} + \mu \cdot \sum_{i=0}^{\frac{N}{2}-1} \lambda_i \cdot e_k^i (R_{kN}^i)^* \quad (6.16)$$

$$= W_{kN} + \mu \cdot \left(F_{N \times \frac{N}{2}} \mathcal{R}_{kN} \right)^H \Lambda F_{N \times \frac{N}{2}} E_{kN} \quad (6.17)$$

$$= W_{kN} + \mu \cdot \mathcal{R}_{kN}^H F_{N \times \frac{N}{2}}^H \Lambda F_{N \times \frac{N}{2}} E_{kN} \quad (6.18)$$

with

$$\Lambda = \text{Diag}(\lambda_0, \lambda_1, \dots, \lambda_{N-1})^t. \quad (6.19)$$

The most costly part in this structure is certainly the matrix multiplication with \mathcal{R}_{kN}^H . But, this matrix is *nearly* I/J-circular. If we could *make* it I/J-circular, the multiplication would correspond to a *circular correlation*. A circular correlation, however, can be performed in the frequency domain using the FFT (\rightarrow de Courville [11]):

$$M_I \cdot \mathbf{V} = F_N \left[\sqrt{N} \cdot F_N^{-1} \cdot (\text{Column}_{n_1}(M_I)) \right] \odot [F_N^{-1} \mathbf{V}] \quad (6.20)$$

$$= F_N^{-1} \left[\sqrt{N} \cdot F_N \cdot (\text{Column}_{n_1}(M_I)) \right] \odot [F_N \mathbf{V}], \quad (6.21)$$

$$M_J \cdot \mathbf{V} = F_N^{-1} \left[\sqrt{N} \cdot F_N \cdot (\text{Column}_{n_1}(M_J)) \right] \odot [F_N^{-1} \mathbf{V}] \quad (6.22)$$

$$= F_N \left[\sqrt{N} \cdot F_N^{-1} \cdot (\text{Column}_{n_1}(M_J)) \right] \odot [F_N \mathbf{V}], \quad (6.23)$$

with \mathbf{V} being a vector of dimension N , M_I being an I-circular matrix

$$M_I = \begin{pmatrix} m_0 & m_1 & \cdots & \cdots & m_{N-1} \\ m_{N-1} & \searrow & \searrow & & \vdots \\ \vdots & \searrow & & \searrow & \vdots \\ m_1 & \cdots & \cdots & m_{N-1} & m_0 \end{pmatrix}, \quad (6.24)$$

M_J being an J -circular matrix

$$M_J = \begin{pmatrix} m_0 & m_1 & \cdots & \cdots & m_{N-1} \\ m_1 & \searrow & \searrow & & m_0 \\ \vdots & \searrow & & \searrow & \vdots \\ m_{N-1} & m_0 & \cdots & \cdots & m_{N-2} \end{pmatrix}, \quad (6.25)$$

and \odot denoting the product of *Schur*:

$$(x_i)_{1 \leq i \leq N} \odot (y_i)_{1 \leq i \leq N} = (x_i \cdot y_i)_{1 \leq i \leq N}. \quad (6.26)$$

So, we try to *make* the matrix \mathcal{R}_{kN}^H

$$\mathcal{R}_{kN}^H = \left(\begin{array}{c|c|c|c} r_0(n) & r_1(n) & & r_{L-1}(n) \\ r_1(n) & r_2(n) & & r_L(n) \\ \vdots & \vdots & \cdots & \vdots \\ r_{N-1}(n) & r_0(n-1) & & r_{L-2}(n-1) \end{array} \right)^H \quad (6.27)$$

circular. In fact, this is not very difficult. It is sufficient to add to every line the values that are needed for circularity. In the end, we also have to add some lines in order to obtain matrix of dimension $2N \times 2N$. In fact, a matrix of the size $(N+L) \times (N+L)$ for updating the TEQ would be sufficient. But, that size wouldn't be very practical, since we have to perform a FFT just of that size. We choose a size of *two times* the old FFT size which is realistic for an implementation.

(6.30) presents \mathcal{R}_{kN} enlarged by some columns and some lines. In the end, the new matrix $\mathcal{R}_{kN}^{2N \times 2N, H}$ is J -circular. In order to perform the multiplication

$$\mathcal{R}_{kN}^{2N \times 2N, H} \cdot \left(F_{N \times \frac{N}{2}}^H \Lambda F_{N \times \frac{N}{2}} E_{kN} \right), \quad (6.28)$$

the vector $\left(F_{N \times \frac{N}{2}}^H \Lambda F_{N \times \frac{N}{2}} E_{kN} \right)$ must be enlarged by $N - L + 1$ zeros. In the end, we are performing the multiplication

$$\mathcal{R}_{kN}^{2N \times 2N, H} \cdot \left(\left(F_{N \times \frac{N}{2}}^H \Lambda F_{N \times \frac{N}{2}} E_{kN} \right)^t, \overbrace{0, 0, \dots, 0}^{N-L+1 \text{ zeros}} \right)^t. \quad (6.29)$$

It is obvious that only the first L entries of the resulting vector are part of the result. The rest is truncated.



6.3.1 A proposition for a practical implementation

Fig.6.3 presents the corresponding structure. Practical issues concerning the implementation are discussed in appendix B.

$$\mathcal{R}_{kN}^{2N \times 2N, H} = \left(\begin{array}{cccc|cccccccccccc}
r_0(n) & r_1(n) & \cdots & r_{L-1}(n) & r_L(n) & \cdots & r_{N-1}(n) & r_0(n-1) & \cdots & r_{L-2}(n-1) & r_{L-1}(n-1) & \cdots & r_{N-1}(n-1) \\
r_1(n) & r_2(n) & & r_L(n) & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & r_0(n) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
r_{N-1}(n) & r_0(n-1) & & r_{L-2}(n-1) & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
r_0(n-1) & r_1(n-1) & & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
r_1(n-1) & r_2(n-1) & & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
r_{N-1}(n-1) & r_0(n) & & r_{L-2}(n) & r_{L-1}(n) & \cdots & r_{N-2}(n) & r_{N-1}(n) & \cdots & r_{L-3}(n-1) & r_{L-2}(n-1) & \cdots & r_{N-2}(n-1)
\end{array} \right)$$

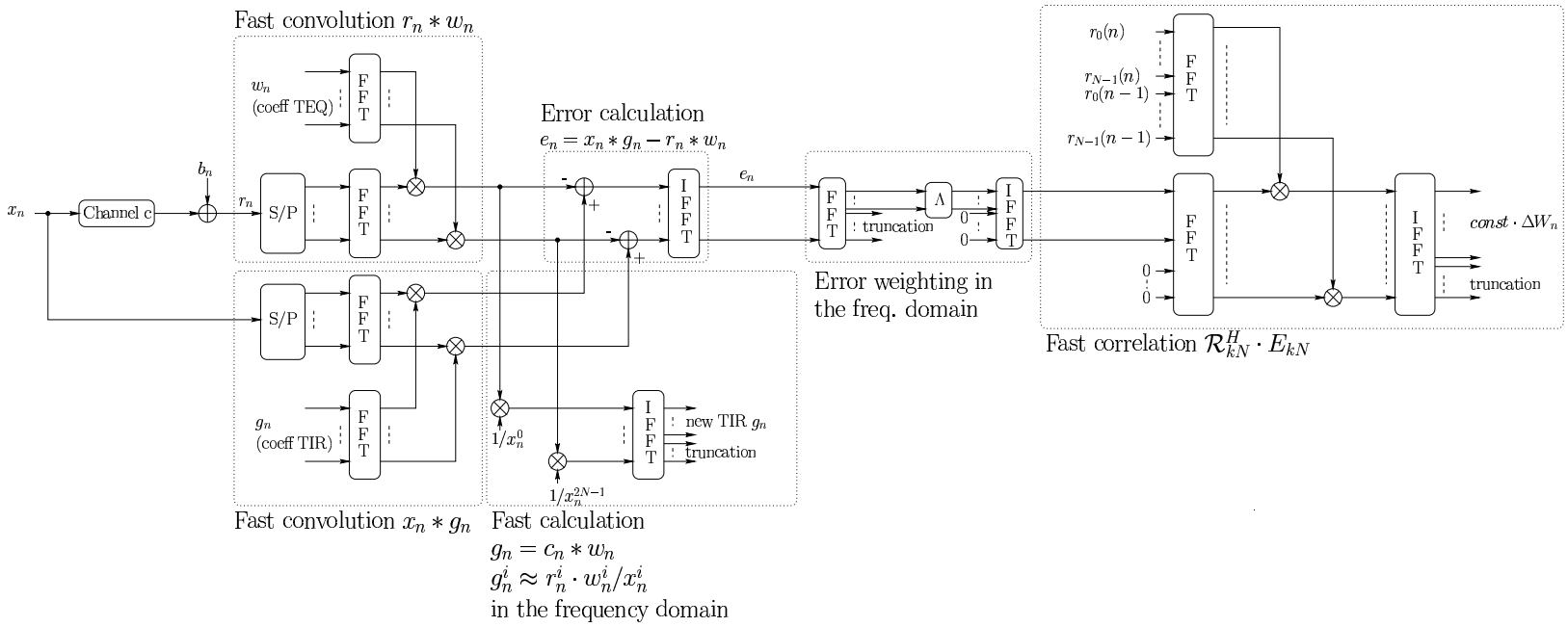


Figure 6.3: The optimal structure for the tap update.

6.4 Complexity evaluation

6.4.1 The complexity of basic operations

The complexity of the operations used is expressed by the number of *real multiplications* ($\mu_{\mathbb{R}}$) and *real additions* ($\alpha_{\mathbb{R}}$) necessary in order to perform the calculation. Tab.6.1 presents the complexity of basic complex operations in $\mu_{\mathbb{R}}$ and $\alpha_{\mathbb{R}}$ (\rightarrow Ifeachor [16], de Courville [10]).

Operations	Abbreviation	\mathbb{R} multiplications	\mathbb{R} additions
real multiplication	$\mu_{\mathbb{R}}$	1	–
real addition	$\alpha_{\mathbb{R}}$	–	1
complex modulus	$mod_{\mathbb{C}}$	2	1
complex multiplication	$\mu_{\mathbb{C}}$	3	3
complex addition	$\alpha_{\mathbb{C}}$	–	2
real \times complex multiplication	$\mu_{\mathbb{R}\mathbb{C}}$	2	–
complex N -points FFT	$DFT_{\mathbb{C}}(N)$	\rightarrow section 6.4.2	\rightarrow section 6.4.2
conv./corr. in time domain	$\mathcal{C}_{\mathbb{C}}^{TIME}(N)$	\rightarrow section 6.4.3	\rightarrow section 6.4.3
fast conv./corr.	$\mathcal{C}_{TC}^{FAST}(N)$	\rightarrow section 6.4.3	\rightarrow section 6.4.3
fast conv./corr. without transformation back into time domain	$\mathcal{C}_{FC}^{FAST}(N)$	\rightarrow section 6.4.3	\rightarrow section 6.4.3

Table 6.1: *The complexity of basic operations.*

6.4.2 The complexity of Fast Fourier Transformation (FFT) algorithms

Tab.6.2 presents the complexity of different FFT algorithms. Wherever possible, we use the *radix-4 FFT* (N must be a number that can be expressed as $N = 4^x, x \in \mathbb{N}^+$). The *split-radix* offers the lightest load whereas the *radix-4* is more power consuming; the *radix-2* comes last. On the other hand, however efficient the *split-radix* algorithm is, its memory access requirements are greater than for *radix-4* (\rightarrow Duhamel [12]).

FFT type	\mathbb{R} multiplications	\mathbb{R} additions	RAM access
radix-2	$\frac{3 \cdot N}{2} \cdot \log_2(N) - 5 \cdot N + 8$	$\frac{7 \cdot N}{2} \cdot \log_2(N) - 5 \cdot N + 8$	$4 \cdot N \cdot \log_2(N)$
radix-4	$\frac{9 \cdot N}{8} \cdot \log_2(N) - \frac{43}{12} \cdot N + \frac{16}{3}$	$\frac{25 \cdot N}{8} \cdot \log_2(N) - \frac{43}{12} \cdot N + \frac{16}{3}$	$2 \cdot N \cdot \log_2(N)$
split-radix	$N \cdot \log_2(N) - 3 \cdot N + 4$	$3 \cdot N \cdot \log_2(N) - 3 \cdot N + 4$	$3 \cdot N \cdot \log_2(N)$

Table 6.2: *The complexity of FFT operations.*

In most cases, we will need a FFT for $N = 2 \cdot 512 = 1024$ points. But, for the weighting of the calculated error (e_n^i) in the frequency domain, a FFT with $N = 512$ points is also required. In order to perform a 512-points FFT, the literature (\rightarrow Duhamel [12]) proposes a solution using a combination of *two radix-4 FFTs (256 points each)* and *radix-2 butterflies (512 points)* as presented by Fig.6.4. This solution is more efficient than a pure *radix-2* implementation.

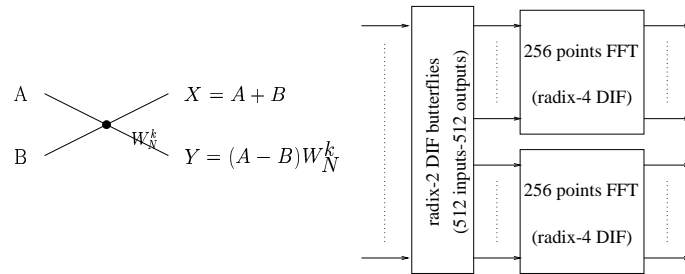


Figure 6.4: A radix-2 butterfly and an efficient solution for a FFT of 512 points.

The complexity of the proposed 512-points FFT and 1024-points FFT is presented by Tab.6.3. The *radix-2-DIF (Decimation in Frequency) butterflies (512 points)* for the 512-points FFT require 512 additions $\alpha_{\mathbb{R}}$ (since there are real time domain values in our case), 252 multiplications $\mu_{\mathbb{R}C}$ and 4 multiplications $\mu_{\mathbb{R}}$.

FFT size	FFT type	\mathbb{R} multiplications	\mathbb{R} additions
512 points	radix-2 butterflies combined with two radix-4 FFTs (256 points)	3292	11488
1024 points	radix-4	7856	28336

Table 6.3: The complexity of FFT operations.

The complexity of the *inverse fast fourier transformation (IFFT)* corresponds to the one of the FFT. Therefore, it won't be taken into consideration separately.

6.4.3 The complexity of convolution and correlation algorithms

The convolution/correlation operation can be performed in the time domain and in the frequency domain. In the case of a FIR-filter with L filter taps, the time domain solution uses $L - 1$ Buffers. For each entering value, L multiplications $\mu_{\mathbb{R}}$ and $L - 1$ additions $\alpha_{\mathbb{R}}$ are necessary (in the case of real time domain data and real filter coefficients).

The frequency domain solution requires a transformation of the L filter coefficients and of the entering data ($N > L$ values) into frequency domain, where the convolution/correlation is performed by simple multiplications $\mu_{\mathbb{C}}$ of the different coefficients. Afterwards, the frequency domain data must be transformed back into time domain. Naturally, the first $L - 1$ values of the fast convolution/correlation result are incorrect, since we have a circular convolution. This problem can be easily resolved. Instead of convolving/correlating the last N received values, we convolve/correlate the last $N + L - 1$ values. Additionally, we add *zeros* to the vector of the filter coefficients and to the vector of the received values. Now, $2N$ -points FFTs are used and the $L - 1$ first values of the convolution/correlation result are truncated. The rest is the result of a linear convolution/correlation ($N \gg L$).

The operations to be performed for the convolution/correlation operation is resumed by Tab.6.4, the complexity is presented by Tab.6.5. Hereby, *hermitian symmetry* in the frequency domain is already taken into account. That's why the multiplication operations in the frequency domain are only performed for half the carriers. The other half of the carriers $Z_i, i \in (\frac{N}{2}+1, \dots, N-1)$ can be easily determined by $Z_i = (Z_{N-i})^*$, $i \in (\frac{N}{2}+1, \dots, N-1)$ (\rightarrow ADSL standard [3]).

Operation	Abbreviation	Operations to be performed
conv./corr. in time domain	$\mathcal{C}_C^{TIME}(N)$	$N \cdot L \times$ real multiplication $\mu_{\mathbb{R}}$ $N \cdot (L - 1) \times$ real addition $\alpha_{\mathbb{R}}$
fast conv./corr.	$\mathcal{C}_{TC}^{FAST}(N)$	$3 \times \text{DFT}_{\mathbb{C}}(2N)$ $\frac{2N}{2} \times$ complex multiplication $\mu_{\mathbb{C}}$
fast conv./corr. without transformation back into time domain	$\mathcal{C}_{FC}^{FAST}(N)$	$2 \times \text{DFT}_{\mathbb{C}}(2N)$ $\frac{2N}{2} \times$ complex multiplication $\mu_{\mathbb{C}}$

Table 6.4: Operations to be performed for the convolution/correlation.

Operations	Abbreviation	\mathbb{R} multiplications	\mathbb{R} additions
conv./corr. in time domain	$\mathcal{C}_C^{TIME}(N)$	$N \cdot L$	$N \cdot (L - 1)$
fast conv./corr. (N=512, L<512)	$\mathcal{C}_{TC}^{FAST}(N)$	25104	86544
fast conv./corr. without transformation back into time domain (N=512, L<512)	$\mathcal{C}_{FC}^{FAST}(N)$	17248	58208

Table 6.5: Operations to be performed for the convolution/correlation (with $N = 512$ carriers and hermitian symmetry in the frequency domain).

The length of the TIR filter is $L = 32$ taps in ADSL, which corresponds to the length of the guard interval. Depending on the algorithm, WSAF or Block LMS (BLMS), the length of the TEQ algorithm will be either $L=64$ taps (minimum for WSAF, as it will be shown by simulations) or $L=16$ taps as proposed by Chen [4] for the LMS/BLMS.

6.4.4 The complexity of the tap update calculation

After having calculated the output data in the time domain of the TIR and TEQ filters, the calculation of the tap update ΔW_n is performed. Tab.6.6 indicates the operations to be performed for the block LMS (BLMS) algorithm performing all convolutions/correlations in the frequency domain. The operations to be performed for the BLMS algorithm using linear convolutions/correlations in the time domain is presented by Tab.6.7 and for the WSAF algorithm performing all convolutions/correlations in the frequency domain by Tab.6.8.

Tab.6.9 and Tab.6.10 resume the complexity of the different algorithms. In this table, the guard interval is fixed to $K = 32$, the length of one DMT symbol to $N = 512$ and the length of the channel impulse response to $P_c = \frac{N}{2} = 256$.

Operation	Steps to be performed	Calculation complexity
Fast convolution of received data with TEQ (the multiplications are performed only for half the spectrum due to hermitian symmetry)	$(y_n^i)_{2N} = \text{FFT} \begin{pmatrix} r_{n-N} \\ r_n \end{pmatrix} \odot \text{FFT} \begin{pmatrix} w_n \\ 0_{2N-L} \end{pmatrix}$	$2 \times \text{DFT}_{\mathbb{C}}(2N)$ $N \times \mu_{\mathbb{C}}$
Fast convolution of original data with TIR (original data are always the same, they must only once be transformed into frequency domain, therefore it isn't counted here)	$(d_n^i)_{2N} = (x_n^i)_{2N} \odot \text{FFT} \begin{pmatrix} g_n \\ 0_{2N-P} \end{pmatrix}$	$\text{DFT}_{\mathbb{C}}(2N)$ $N \times \mu_{\mathbb{C}}$
Error calculation in the frequency domain	$(e_n^i)_{2N} = (d_n^i)_{2N} - (y_n^i)_{2N}$	$N \times \alpha_{\mathbb{C}}$
Transformation of $(e_n^i)_{2N}$ back into time domain (since an overlapping is necessary for the convolution, a windowing must be performed in time domain)	$\text{IFFT}(e_n^i)_{2N}$	$\text{DFT}_{\mathbb{C}}(2N)$
Fast correlation (with transformation back into time domain)	$\text{IFFT} \left(\text{FFT} \left(\text{column}_1 \left(\mathcal{R}_{kN}^{2N \times 2N, H} \right) \right) \odot \text{FFT} \begin{pmatrix} e_n \\ 0_N \end{pmatrix} \right)$ $= \text{FFT} \begin{pmatrix} r_{n-N} \\ r_n \end{pmatrix}, \rightarrow \text{read}$ explication in this paragraph	$\mathcal{C}_{FC}^{\text{FAST}}(N)$
Multiplication of the first L values of the correlation result with μ	$\mu \cdot (\text{result of fast correlation})$	$L \times \mu_{\mathbb{R}}$
Filter update in the time domain	$w_{n+1} = w_n + \Delta w_n$	$L \times \alpha_{\mathbb{R}}$
Calculation of $(\text{CIR} * \text{TEQ})$ in the frequency domain $\left((r_n^i)_{2N} \odot (w_n^i)_{2N} = \text{FFT} \begin{pmatrix} r_{n-N} \\ r_n \end{pmatrix} \odot \text{FFT} \begin{pmatrix} w_n \\ 0_{2N-L} \end{pmatrix} \right)$ was already calculated before) and transformation back into time domain for the TIR-update	$\text{IFFT} \left((y_n^i)_{2N} \odot \left(\frac{1}{x_n^i} \right)_{2N} \right)$	$N \times \mu_{\mathbb{C}}$ $\text{DFT}_{\mathbb{C}}(2N)$

Table 6.6: The operations for the block LMS (BLMS) algorithm using fast convolutions/correlations.



Operation	Steps to be performed	Calculation complexity
Convolution of received data with TEQ (L = Number of TEQ filter taps)	$y_n = r_n * w_n$	$N \cdot L \times \mu_{\mathbb{R}}$ $N \cdot (L - 1) \times \alpha_{\mathbb{R}}$
Convolution of original data with TIR (K = Number of TIR filter taps)	$d_n = x_n * g_n$	$N \cdot K \times \mu_{\mathbb{R}}$ $N \cdot (K - 1) \times \alpha_{\mathbb{R}}$
Error calculation in the time domain	$e_n = d_n - y_n$	$N \times \alpha_{\mathbb{R}}$
Correlation	$\mathcal{R}_{kN}^{2N \times 2N, H} \cdot E_{kN}$	$N \cdot L \times \mu_{\mathbb{R}}$ $N \cdot (L - 1) \times \alpha_{\mathbb{R}}$
Multiplication of the correlation result with μ	$\mu \cdot$ (result of correlation)	$L \times \mu_{\mathbb{R}}$
Filter update in the time domain	$w_{n+1} = w_n + \Delta w_n$	$L \times \alpha_{\mathbb{R}}$
Calculation of (CIR * TEQ) in the time domain (P_c = length of the channel impulse response c_n , usually $P_c \approx \frac{N}{2}$)	$c_n * w_n$	$P_c \cdot L \times \mu_{\mathbb{R}}$ $P_c \cdot (L - 1) \times \alpha_{\mathbb{R}}$

Table 6.7: The operations for the block LMS (BLMS) algorithm using linear convolutions/correlations in the time domain.

Operation	Steps to be performed	Calculation complexity
Fast convolution of received data with TEQ	$(y_n^i)_{2N} = \text{FFT} \begin{pmatrix} r_{n-N} \\ r_n \end{pmatrix} \odot \text{FFT} \begin{pmatrix} w_n \\ 0_{2N-L} \end{pmatrix}$	$2 \times \text{DFT}_{\mathbb{C}}(2N)$ $N \times \mu_{\mathbb{C}}$
Fast convolution of original data with TIR	$(d_n^i)_{2N} = (x_n^i)_{2N} \odot \text{FFT} \begin{pmatrix} g_n \\ 0_{2N-P} \end{pmatrix}$	$\text{DFT}_{\mathbb{C}}(2N)$ $N \times \mu_{\mathbb{C}}$
Error calculation in the frequency domain	$(e_n^i)_{2N} = (d_n^i)_{2N} - (y_n^i)_{2N}$	$N \times \alpha_{\mathbb{C}}$
Transformation of $(e_n^i)_{2N}$ back into time domain (since an overlapping is necessary for the convolution, a windowing must be performed in time domain)	$\text{IFFT}(e_n^i)_{2N}$	$\text{DFT}_{\mathbb{C}}(2N)$
Weighting of e_n in the frequency domain and transformation back into time domain	$\hat{e}_n = \text{IFFT}(\Lambda \cdot \text{FFT}(e_n)_N)$	$2 \times \text{DFT}_{\mathbb{C}}(N)$ $\frac{N}{2} \times \mu_{\mathbb{R}\mathbb{C}}$
Fast correlation (with transformation back into time domain)	$\text{IFFT} \left(\underbrace{\text{FFT} \left(\text{column}_1 \left(\mathcal{R}_{kN}^{2N \times 2N, H} \right) \right)}_{= \text{FFT} \begin{pmatrix} r_{n-N} \\ r_n \end{pmatrix}, \rightarrow \text{read}} \odot \text{FFT} \begin{pmatrix} \hat{e}_n \\ 0_N \end{pmatrix} \right)$ explication in this paragraph	$\mathcal{C}_{FC}^{FAST}(N)$
Multiplication of the first L values of the correlation result with μ	$\mu \cdot (\text{result of fast correlation})$	$L \times \mu_{\mathbb{R}}$
Filter update in the time domain	$w_{n+1} = w_n + \Delta w_n$	$L \times \alpha_{\mathbb{R}}$
Calculation of $(\text{CIR} * \text{TEQ})$ in the frequency domain $\left((r_n^i)_{2N} \odot (w_n^i)_{2N} = \text{FFT} \begin{pmatrix} r_{n-N} \\ r_n \end{pmatrix} \odot \text{FFT} \begin{pmatrix} w_n \\ 0_{2N-L} \end{pmatrix} \right)$ was already calculated before) and transformation back into time domain for the TIR-update	$\text{IFFT} \left((y_n^i)_{2N} \odot \left(\frac{1}{x_n^i} \right)_{2N} \right)$	$N \times \mu_{\mathbb{C}}$ $\text{DFT}_{\mathbb{C}}(2N)$

Table 6.8: The operations for the WSAF algorithm using fast convolutions/correlations.



Algorithm	Calculation Complexity	\mathbb{R} multiplications	\mathbb{R} additions
LMS (performing convolutions/ correlations in time domain) $L = 16$ filter taps	$(2NL + NK + L + PL) \times \mu_{\mathbb{R}}$ $(2N(L - 1) + N(K - 1) + L + P(L - 1)) \times \alpha_{\mathbb{R}}$	36880	35088
LMS (performing convolutions/ correlations in time domain) $L = 64$ filter taps	$(2NL + NK + L + PL) \times \mu_{\mathbb{R}}$ $(2N(L - 1) + N(K - 1) + L + P(L - 1)) \times \alpha_{\mathbb{R}}$	98368	96576
LMS (fast convolution/ correlation) $L = 16$ filter taps	$7 \times \text{DFT}_{\mathbb{C}}(2N), 4N \times \mu_{\mathbb{C}}$ $L \times \alpha_{\mathbb{R}}, L \times \mu_{\mathbb{R}}$ $N \times \alpha_{\mathbb{C}}$	61152	205536
LMS (fast convolution/ correlation) $L = 64$ filter taps	$7 \times \text{DFT}_{\mathbb{C}}(2N), 4N \times \mu_{\mathbb{C}}$ $L \times \alpha_{\mathbb{R}}, L \times \mu_{\mathbb{R}}$ $N \times \alpha_{\mathbb{C}}$	61200	204688
WSAF (fast convolution/ correlation) $L = 64$ filter taps	$7 \times \text{DFT}_{\mathbb{C}}(2N), 2 \times \text{DFT}_{\mathbb{C}}(N),$ $4N \times \mu_{\mathbb{C}}, L \times \alpha_{\mathbb{R}},$ $L \times \mu_{\mathbb{R}}, N \times \alpha_{\mathbb{C}}, \frac{N}{2} \times \mu_{\mathbb{RC}}$	68296	228560

Table 6.9: The complexity of the different filter update algorithms.

Algorithm	\mathbb{R} multiplications (normalized by 36880)	\mathbb{R} additions (normalized by 35088)
LMS (performing convolutions/ correlations in time domain) $L = 16$ filter taps	1	1
LMS (performing convolutions/ correlations in time domain) $L = 64$ filter taps	2.667	2.752
LMS (fast convolution/ correlation) $L = 16$ filter taps	1.658	5.857
LMS (fast convolution/ correlation) $L = 64$ filter taps	1.659	5.834
WSAF (fast convolution/ correlation) $L = 64$ filter taps	1.851	6.513

Table 6.10: *The complexity of the different filter update algorithms normalized by the number of multiplications/additions of the LMS algorithm (performing convolutions/correlations in time domain, $L = 16$ filter taps).*

Now, we are going to explain what has to be considered concerning the reuse of

$$\text{FFT} \left(\text{column}_1 \left(\mathcal{R}_{kN}^{2N \times 2N, H} \right) \right) \quad (6.31)$$

for the calculation of

$$(y_n^i)_{2N} = \text{FFT} (r_{\times}) \odot \text{FFT} \left(\begin{array}{c} w_n \\ 0_{2N-L} \end{array} \right). \quad (6.32)$$

There are different constraints placed upon the shape of $(r_{\times})_{2N}$ and $\text{column}_1 \left(\mathcal{R}_{kN}^{2N \times 2N, H} \right)$.

We have to take $2N$ time-domain samples for the FFT calculating $(y_n^i)_{2N}$. Here, the consideration of the last $N + L$ arriving samples r_n would be sufficient in order to perform the convolution $r_n * w_n$ in the frequency domain, since the TEQ filter has only L taps. The remaining samples could be filled up with *zeros*:

$$(r_{\times})_{2N} = (r_{N-L}(n-1), \dots, r_{N-1}(n-1), r_0(n), \dots, r_{N-1}(n), 0, \dots, 0)^t. \quad (6.33)$$

We note by the way that it wouldn't be more costly to take instead of the additional *zeros* altogether the $2N$ last arriving samples at the input of the FFT. In this case, the desired result $c_n * w_n$ can be obtained as well.

In the case of $\text{column}_1 \left(\mathcal{R}_{kN}^{2N \times 2N, H} \right)$, we could also add some *zeros* in order to create a *J-circular* matrix. In this case, the first column of $\text{column}_1 \left(\mathcal{R}_{kN}^{2N \times 2N, H} \right)$ would look like

$$(r_0(n), r_1(n), \dots, r_{N-1}(n), r_0(n-1), \dots, r_{L-2}(n-1), 0, \dots, 0)^t. \quad (6.34)$$

As an alternative, the remaining values of $r(n-1)$ instead of the *zeros* can be used. The result will rest the same.

Now, we have some constraints and some liberties upon the two expression. We will use the *liberties* in order to find an expression corresponding to the *constraints* of both. In fact, this isn't very difficult. The transformation of the vector

$$(r_0(n), r_1(n), \dots, r_{N-1}(n), r_0(n-1), \dots, r_{N-1}(n-1))^t. \quad (6.35)$$

in the frequency domain may be used for both, for the calculation of $(y_n^i)_{2N}$ and for performing the correlation with $\text{column}_1 \left(\mathcal{R}_{kN}^{2N \times 2N, H} \right)$ using the properties of a *circular convolution/correlation*.

6.4.5 The complexity of the Time Domain Equalization (TEQ) convolution

The complexity of the *Time Domain Equalization (TEQ) convolution* ($r_n * w_n$) is even more important than the complexity of the tap update algorithm, since the tap update is only performed during the learning sequence. The convolution $r_n * w_n$, however, must always be performed for every arriving symbol.

Here, two possibilities for implementing the TEQ convolution are considered: A fast implementation using the *overlap-add* algorithm (\rightarrow Ifeachor [16]) and an implementation where the convolution is performed in the time domain using $L - 1$ Buffers for L TEQ filter taps. Tab.6.11 presents the complexity of the different implementations. For the fast convolutions, we consider two solution, one using a N -points FFT and one using a $2N$ -points FFT, since these two FFTs are already used by the WSAF update algorithm. The fast algorithms require *two* FFTs, one for calculating the arriving data in the frequency domain (r_n^i) and one for re-transforming the convolved data into time domain. After the tap update, the TEQ filter doesn't change any more. Therefore, the filter coefficients must only once be transformed into frequency domain.

Algorithm	Calculation Complexity	Number of filtered samples
convolution in the time domain	$N \cdot L \times \mu_{\mathbb{R}}, N \cdot (L - 1) \times \alpha_{\mathbb{R}}$	N
fast convolution (using a N -points FFT)	$2 \cdot \text{DFT}_{\mathbb{C}}(N), \frac{N}{2} \times \mu_{\mathbb{C}},$ $L \times \alpha_{\mathbb{R}}$ (for overlap-add)	$N-L+1$
fast convolution (using a $2N$ -points FFT)	$2 \cdot \text{DFT}_{\mathbb{C}}(2N), \frac{2N}{2} \times \mu_{\mathbb{C}},$ $L \times \alpha_{\mathbb{R}}$ (for overlap-add)	$2N-L+1$

 Table 6.11: *The complexity of the different convolution algorithms.*

The number of operations for the case of an ADSL system with $N = 512$ and $L = 64$ or $L = 16$ are presented by Tab.6.12.

Hereby, it has to be considered that only $N - L + 1$ values can be used per $\text{FFT}(N)$ operation. This is the case, since we need the results of a *linear convolution/correlation* in order to apply the *overlap add* algorithm. Therefore, the number of filtered samples per block is mentioned in the column *Number of filtered samples* (\rightarrow Tab.6.11).

Algorithm	\mathbb{R} multiplications per arriving sample	\mathbb{R} additions per arriving sample
convolution in the time domain, $L = 16$	16	15
convolution in the time domain, $L = 64$	64	63
fast convolution, $L = 16$ (using a N -points FFT)	14.8	47.8
fast convolution, $L = 64$ (using a N -points FFT)	16.5	53.0
fast convolution, $L = 16$ (using a $2N$ -points FFT)	17.1	57.7
fast convolution, $L = 64$ (using a $2N$ -points FFT)	18	60.6

 Table 6.12: *The complexity of the different convolution algorithms (ADSL).*

The most important point is that the number of real multiplications is practically the same for the 16-taps time-domain convolution algorithm and all fast convolution algorithms, even for 64-taps cases. By switching from a 16-taps TEQ filter to a 64-taps filter, only the number of real additions to be performed rises. However, real additions are much less complex than multiplications. In the end, the higher complexity of a 64-taps TEQ filter will still be acceptable.

Chapter 7

Simulation results

This chapter presents simulation results obtained with the algorithm described in chapter 6. The convergence speed is compared to the one of a simple LMS algorithm, once without noise and once with *Additive White Gaussian Noise* (AWGN). Moreover, the resulting TEQ filter taps and the resulting impulse response $c_n * w_n$ is presented. The remaining energy of the resulting impulse response after the guard interval $c_n * w_n|_{\text{after GI}}$ is calculated.

7.1 Parameters of the simulation

The channel is an important parameter of a simulation. We use a real channel impulse response corresponding to the *CSA-Loop #6 scenario* (9 kfeet length, 26 American Wire Gauge (AWG)), as it is defined by the ADSL standard [3]. It is presented by Fig.7.1.

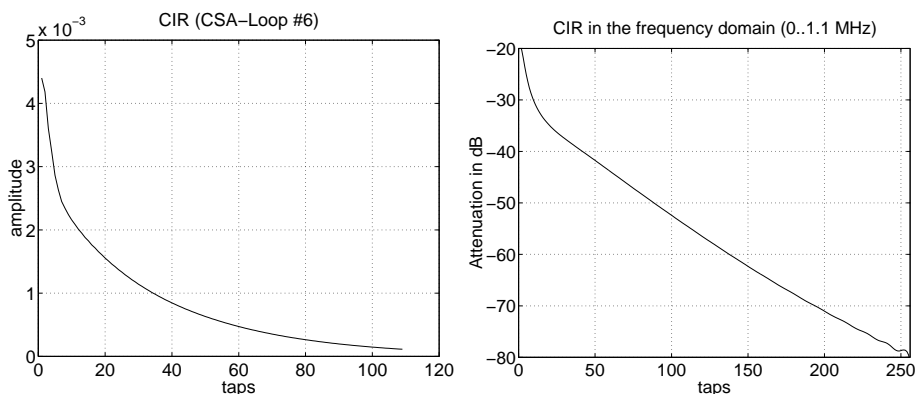


Figure 7.1: The channel impulse response (CIR) corresponding to CSA-Loop #6 in the time and frequency domain.

Each DMT symbol is loaded with the *C-REVERBI* symbol during the equalizer training, as defined by the ADSL standard [3], chapter 12.4.4 and 12.4.8. The *C-REVERBI* symbol uses a pseudo-random data pattern modulated on the carriers by a 4-QAM signal constellation (constellations $\{+, +\}$, $\{+, -\}$, $\{-, +\}$ and $\{-, -\}$).

A very important topic is the choice of the number of filter taps to be used for the TEQ filter. It may not be obvious at a first glance, but the WSAF algorithm will require a greater number of filter taps than the LMS algorithm. We can understand this fact by calculating the filter update in the frequency domain using (6.18):

$$W_{(k+1)N}^i = W_{kN}^i + \mu \cdot \lambda_i \cdot (R_{kN}^i)^* \cdot e_k^i. \tag{7.1}$$

Without weighting the different $(R_{kN}^i)^*$, $W_{(k+1)N}^i$ won't change very much for high frequencies, since the attenuation of the channel is extremely high for them. There are only some values changing in the lower frequency band. In the end, a small number of degrees of freedom will be sufficient in order to fulfill the 512 equations for $i = 0, \dots, 511$ approximately, i.e. a small number of filter taps may be used. Using the WSAF algorithm, all $(R_{kN}^i)^*$ have an important influence. So, updating a filter with a small number of taps will result in anything but a reasonable TEQ. The following simulations will demonstrate this behaviour.

We are performing the first iteration of the filter update with a 128 taps filter and a 16 taps filter with and without using a weighted criterion. The TEQ filter is initialized as indicated by Tab.7.1. The corresponding frequency domain properties are presented by Fig.7.2, it's just a constant value over all frequencies.

n	w_n
0	0.5
2...31	0

Table 7.1: Initialization of the TEQ filter taps.

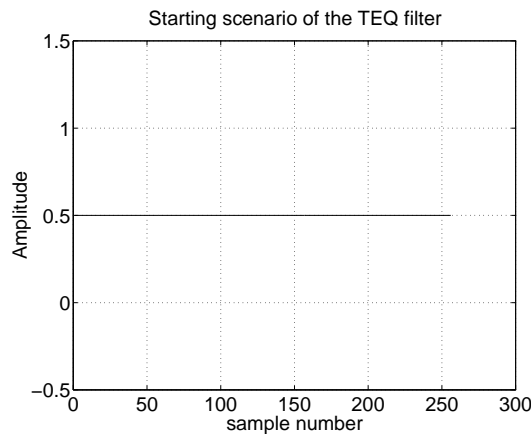


Figure 7.2: The initialization of the TEQ filter in the frequency domain.

Using a weighted criterion (WSAF algorithm), the resulting TEQ filter after the first iteration for 128 and 16 filter taps is presented by Fig.7.3. It can clearly be seen that the 16-taps filter (*dotted line*) has lost quite a lot compared to the 128-taps filter (*solid line*).

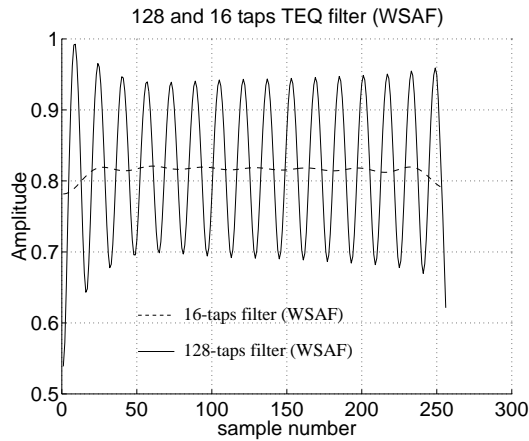


Figure 7.3: The TEQ filter after one iteration for 128 and 16 filter taps using the WSAF algorithm.

Without a weighted criterion, the results are quite different as it is demonstrated by Fig.7.4. The 16-taps filter (*dotted line*) is still similar to the 128-taps filter (*solid line*) and the result will therefore be satisfying.

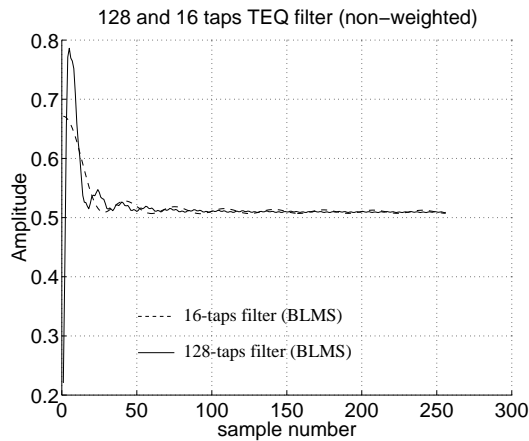


Figure 7.4: The TEQ filter after one iteration for 128 and 16 filter taps without a weighted criterion.

It is obvious that the LMS algorithm works better than the WSAF for a small number of filter taps, since quite some information is not used ($(R_{kN}^i)^* \approx 0$ for high frequencies). Using the WSAF algorithm, all $(R_{kN}^i)^*$ are taken into account. Therefore, more degrees of freedom, i.e. more filter taps are required.

7.2 Convergence properties without noise

Fig.7.5 and Fig.7.6 present the simulation results for an environment without noise.

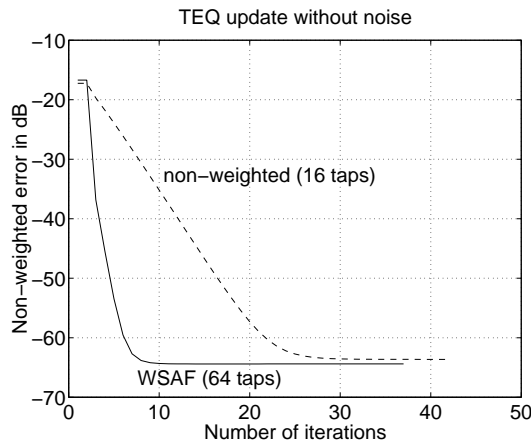


Figure 7.5: The convergence properties without noise, non-weighted (16 filter taps) and WSAF (64 filter taps).

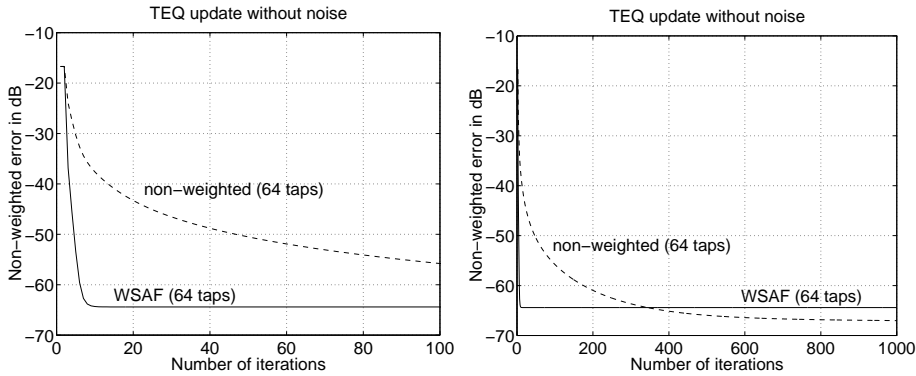


Figure 7.6: The convergence properties without noise, 64 filter taps for the non-weighted case and the WSAF (zoomed and over 1000 iterations).

The TEQ filter taps were initialized with the values presented by Tab.7.2.

n	w_n
0	0.5
2...31	0

Table 7.2: Initialization of the TEQ filter taps.

The simulations clearly reveal that during the first iterations the two algorithms converge with approximately the same speed. Then, the new WSAF algorithm converges rapidly. The classic LMS algorithm, however, converges after a large number of iterations.

7.3 Convergence properties with Additive White Gaussian Noise

Fig.7.7, Fig.7.8 and Fig.7.9 present the simulation results in an *Additive White Gaussian Noise* (AWGN) environment with a *Signal-to-Noise Ratio* (SNR) of 30dB, 40dB and 50dB respectively. The error presented here, is the error \hat{J} estimated at the reception site. The true error J cannot be calculated, since the exact channel impulse response is not known at the reception site.

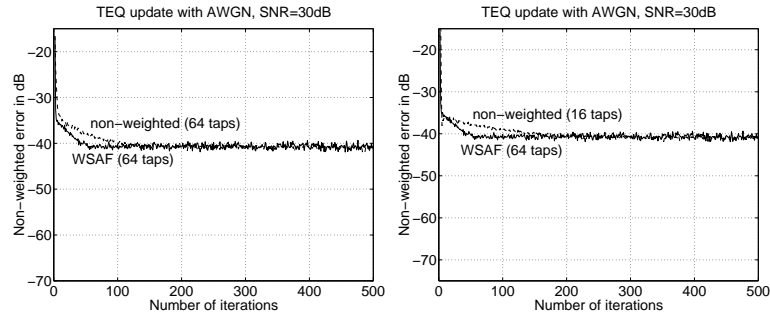


Figure 7.7: *The convergence properties with SNR=30dB.*

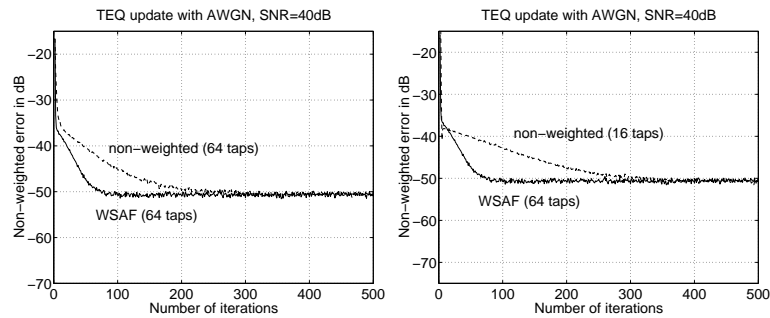


Figure 7.8: *The convergence properties with SNR=40dB.*

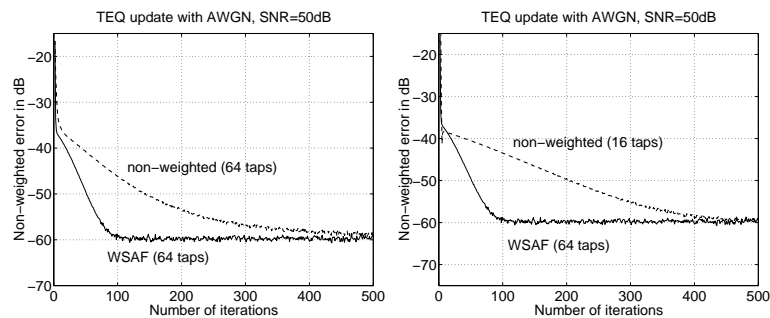


Figure 7.9: *The convergence properties with SNR=50dB.*

Fig.7.10, Fig.7.11 and Fig.7.12 present a zoom of the simulation results.

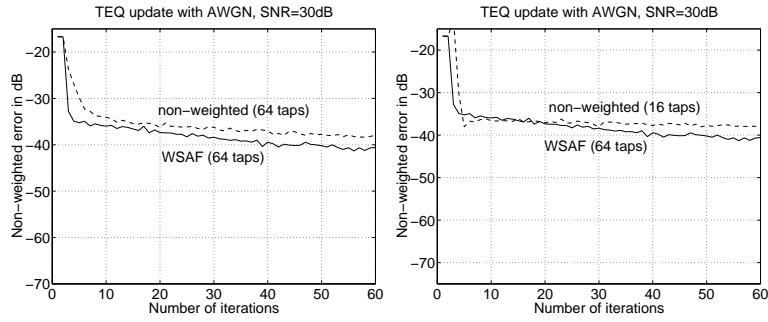


Figure 7.10: *The convergence properties with SNR=30dB (zoomed).*

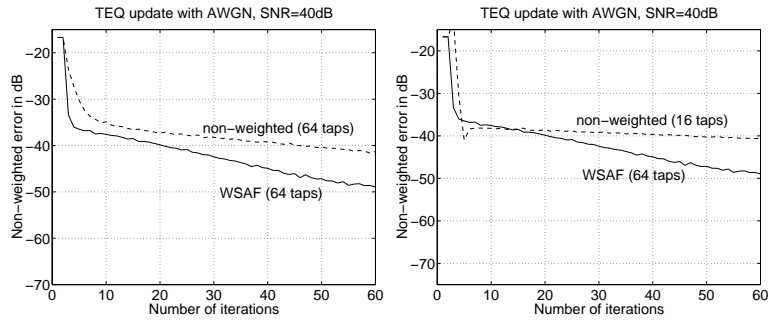


Figure 7.11: *The convergence properties with SNR=40dB (zoomed).*

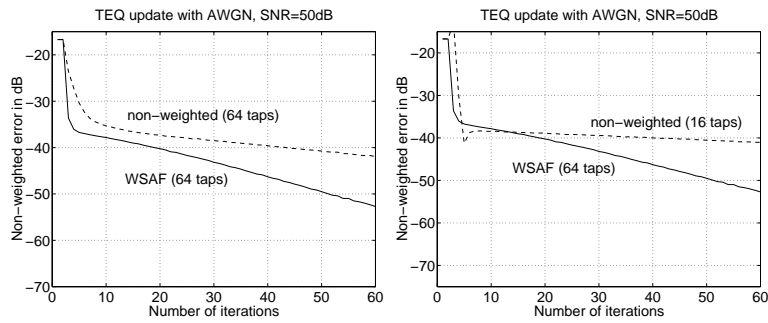


Figure 7.12: *The convergence properties with SNR=50dB (zoomed).*

As we mentioned before, only an estimated error can be calculated at the reception site, since the *true* channel impulse response is not known and there is noise added. These estimated errors were presented by Fig.7.7, Fig.7.8, Fig.7.9 and in a zoomed version by Fig.7.10, Fig.7.11, Fig.7.12. It is interesting to compare the *estimated error* \hat{J} with the *true error* J as it is presented by Fig.7.13, Fig.7.14 and Fig.7.15 (Reminder: The error presented by Fig.7.7, Fig.7.8, Fig.7.9, Fig.7.10, Fig.7.11 and Fig.7.12 is the error \hat{J} estimated at the reception site. The *true error* J presented by Fig.7.13, Fig.7.14 and Fig.7.15 normally cannot be calculated, since the exact channel impulse response is not known at the reception site). In order to visualize the instability of the *non-weighted 16-taps filter*, a linear scale is used.

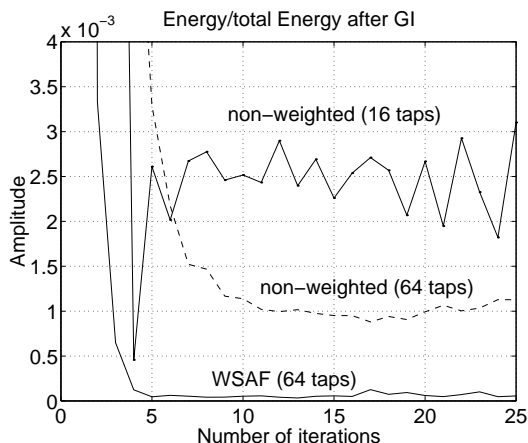


Figure 7.13: *The true error during the optimization with SNR=30dB.*

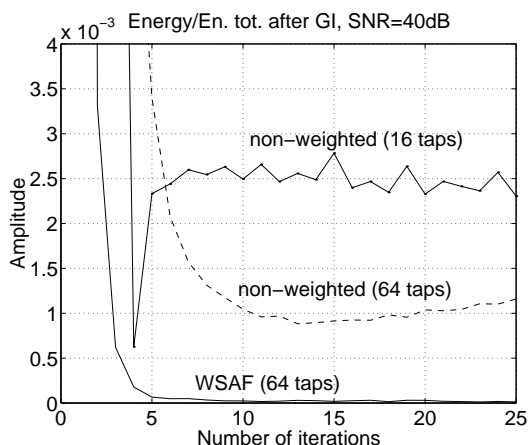


Figure 7.14: *The true error during the optimization with SNR=40dB.*

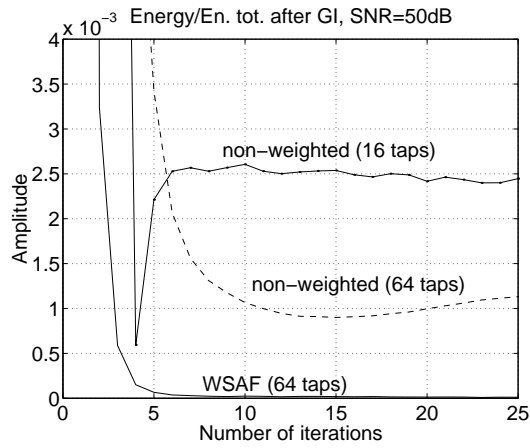


Figure 7.15: The true error during the optimization with SNR=50dB.

In the case of the 16-taps filter, the algorithm doesn't converge at the lowest true error found. The unprecise choice of the TIR filter after each iteration step is responsible for that behaviour. If we could always choose the exact value (the 32 first taps of CIR * TEQ) knowing the exact CIR, the algorithm would converge at a level near to the one of the 64-taps filter. Here, the update procedure is much more sensitive to unprecise values than the one using the WSAF algorithm in order to update a 64-taps filter.

7.4 The resulting TEQ filters

The most interesting result is the portion of energy remaining after the guard interval (32 taps) of the resulting impulse response $w_n * c_n$, the shape of $w_n * c_n$ itself and the shape of the TEQ filter impulse response. Fig.7.16 presents the shape of these two impulse responses for a simulation without noise using a weighted criterion.

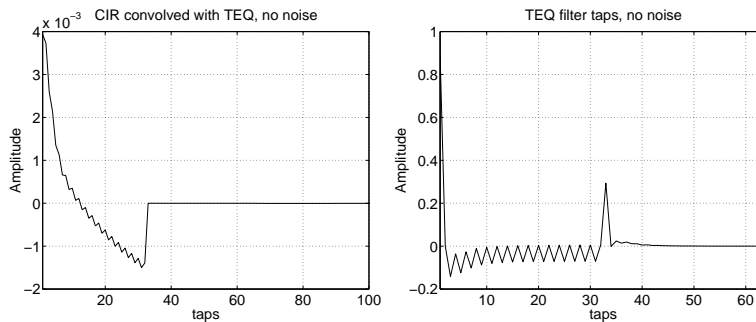


Figure 7.16: The resulting impulse response $w_n * c_n$ and the one of the TEQ filter (no noise) using a weighted criterion.

In the case of the 64-taps filter, the remaining $\frac{\text{Energy after the guard interval}}{\text{total Energy}}$ after the guard interval is so small $\left(\frac{\text{Energy after the guard interval}}{\text{total Energy}} \approx 10^{-6} \dots 10^{-3} \right)$ that the *inter-symbol interference* between two OFDM/DMT symbols using a guard interval of 32 taps will be nearly negligible.

Tab.7.3 presents some interesting values of achieved $\frac{\text{Energy after the guard interval}}{\text{total Energy}}$ (these are *true errors*, the error calculation is based on the *true* channel) and the corresponding number of iterations that was necessary in order to train the TEQ filter.

Simulation	$\frac{\text{Energy after guard interval (32 taps)}}{\text{Total energy (WSAF)}}$	$\frac{\text{Energy after guard interval (32 taps)}}{\text{Total energy (no weighting)}}$
No TEQ filter	0.11194	0.11194
16 taps TEQ, no noise (channel known)	no convergence	$4.979 \cdot 10^{-6}$ (≈ 25 iterations)
64 taps TEQ, no noise (channel known)	$5.1043 \cdot 10^{-6}$ (≈ 10 iterations)	$2.397 \cdot 10^{-6}$ (≈ 600 iterations)
16 taps TEQ, SNR=50dB (channel estimated)	no convergence	$1.9738 \cdot 10^{-3}$ (≈ 60 iterations)
16 taps TEQ, SNR=50dB (channel estimated)	no convergence	$3.8971 \cdot 10^{-5}$ (≈ 350 iterations)
64 taps TEQ, SNR=50dB (channel estimated)	$3.9014 \cdot 10^{-6}$ (≈ 60 iterations)	$1.92213 \cdot 10^{-5}$ (≈ 350 iterations) $1.0954 \cdot 10^{-3}$ (60 iterations)
16 taps TEQ, SNR=40dB (channel estimated)	no convergence	$2.0874 \cdot 10^{-3}$ (≈ 60 iterations)
16 taps TEQ, SNR=40dB (channel estimated)	no convergence	$3.7508 \cdot 10^{-5}$ (≈ 350 iterations)
64 taps TEQ, SNR=40dB (channel estimated)	$2.2184 \cdot 10^{-5}$ (≈ 60 iterations)	$1.9672 \cdot 10^{-6}$ (≈ 350 iterations) $1.1106 \cdot 10^{-3}$ (60 iterations)
16 taps TEQ, SNR=30dB (channel estimated)	no convergence	$1.9841 \cdot 10^{-3}$ (≈ 60 iterations)
16 taps TEQ, SNR=30dB (channel estimated)	no convergence	$3.2484 \cdot 10^{-5}$ (≈ 350 iterations)
64 taps TEQ, SNR=30dB (channel estimated)	$5.2470 \cdot 10^{-5}$ (≈ 60 iterations)	$2.0372 \cdot 10^{-5}$ (≈ 350 iterations) $1.0642 \cdot 10^{-3}$ (60 iterations)

Table 7.3: The portion of the remaining energy of $w_n * c_n$ after the guard interval.

Tab.7.4 presents the approximate number of iterations that is necessary in order to achieve certain values of $\frac{\text{Energy after the guard interval}}{\text{total Energy}}$ (these are *true errors*, the error calculation is based on the *true* channel). For the 16-taps TEQ filter, the peak down to very little energy values at the beginning of the equalizer training (\rightarrow Fig.7.13, Fig.7.14 and Fig.7.15) has not been taken into account.

$\frac{\text{Energy after guard interval (32 taps)}}{\text{Total energy}}$ of CIR * TIR	Nbr iterations for WSAF (64 taps)	Nbr. iterations for non-weighted (64 taps)	Nbr. iterations for non-weighted (16 taps)
10^{-3} , SNR = 30dB	4	12	119
10^{-4} , SNR = 30dB	5	195	271
10^{-5} , SNR = 30dB	735 (uncertain)	456 (uncertain)	340 (uncertain)
10^{-3} , SNR = 40dB	3	11	133
10^{-4} , SNR = 40dB	5	198	287
10^{-5} , SNR = 40dB	26	445	413
10^{-3} , SNR = 50dB	3	11	134
10^{-4} , SNR = 50dB	5	198	292
10^{-5} , SNR = 50dB	23	442	446

Table 7.4: Minimum number of iterations for a remaining energy of $w_n * c_n$ after the guard interval.

The tables Tab.7.3 and Tab.7.4 demonstrate the faster convergence speed of the new WSAF algorithm compared to the LMS solution. As it has been calculated in chapter 6, the price for this improvement is an increase of the tap update calculation complexity (\rightarrow Tab.6.10) and a slight increase of the convolution complexity $TEQ * \text{Received data}$ (\rightarrow Tab.6.12).

Chapter 8

Conclusions

The ADSL simulator in C/C++ works well and guarantees fast results. The TEQ filter tap update is, for example, approximately *50 times* faster than its MATLAB version. In the future, new modules can be easily tested and it won't be difficult to determine their performance.

The new filter tap update algorithm that has been discussed in the last chapters uses a *weighted criterion* in order to improve the convergence speed. The resulting performance boost is considerable, compared to the classical *least mean square* (LMS) algorithm. In our simulations, after approximately *5-10* iterations the new algorithm already delivers good results (→ Tab.7.4). The classical LMS algorithm needed *200* iterations or more. The price for the better convergence properties is that more filter coefficients must be used, otherwise the new *Weighted Sub-band Adaptive Filter* (WSAF) algorithm does not converge.

The *Target Impulse Response* (TIR) filter has been initialized with filter coefficients where only one coefficient is non-zero. So, at each iteration the TEQ filter is adapted to the latest TIR and the TIR filter is updated afterwards. Thanks to the improved convergence speed of the new algorithm, the main problem of the LMS algorithm is eliminated and its simple structure can actually be used for an implementation. It is much simpler and less costly in computation time than the propositions of Chow [7] which are widely used despite of the fact that they result in costly eigenvalue problems.

Our simulations are based on the *CSA-Loop #6 scenario* (9 kfeet length, 26 American Wire Gauge (AWG)), as it is defined by the ADSL standard [3] (→ Fig.7.1). The attenuations for the high frequencies are much more important for longer loops. Hereby, an even better performance is expected for the new algorithm using a weighted criterion compared to the classical LMS.

In the end, a new algorithm is proposed that combines the simple implementation structures of the LMS algorithm with a fast convergence speed. This proposition can be considered as an excellent solution for any xDSL system.

In the future, a *hybrid circuit emulation* and an *echo cancelling unit* may be implemented into the simulator. This will allow to get an even better impression of the performance of the proposed WSAF equalization algorithm. Moreover, a *field test* of the new algorithm in an existing ADSL environment is still to be done.

Appendix A

Convergence properties of the LMS algorithm

This report presents an adaption of the *Weighted Sub-band Adaptive Filter* (WSAF) algorithm proposed by de Courville [10] to an ADSL system. Weighting factors in the frequency domain are introduced in order to optimize the convergence speed. This appendix demonstrates theoretically why the convergence speed can actually be optimized by these factors. The information presented here is mainly based on Proakis [27], Haykin [14], Kay [19] and de Courville [10, 11].

A.1 Some definitions and properties

Here, some definitions and properties are presented which will help us to understand the WSAF properties (\rightarrow Haykin [14], Kay [19]).

Definition: A *discrete random process* r_n is a sequence of random variables defined for every integer n

Definition: A *wide sense stationary* (WSS) discrete random process has a mean $E(r_n) = \mu_r$ which does not depend on n and an autocorrelation function $t_r[k] = t_r[n, n - k] = E(r_n r_{n-k}^*)$ which depends only on the lag k between the two samples and not their absolute positions.

Definition: The N -by-1 *observation vector* R_n represents the elements of the time series r_n : $R_n = (r_n, r_{n-1}, \dots, r_{n-N+1})^t$.

Property 1: The correlation matrix of a stationary discrete-time stochastic process is hermitian: $T_A^H = T_A$ with

$$T_A = E(R_n R_n^H) \tag{A.1}$$

$$= \begin{pmatrix} t_r[0] & t_r[1] & \dots & t_r[N-1] \\ t_r[-1] & t_r[0] & \dots & t_r[N-2] \\ \vdots & \vdots & \ddots & \vdots \\ t_r[-N+1] & t_r[-N+2] & \dots & t_r[0] \end{pmatrix}. \tag{A.2}$$

Property 2: *Property 1* corresponds to: $t_r[-k] = t_r^*[k]$.

Proof: It is known that $t_r[k] = t_r[n, n - k] = E(r_n r_{n-k}^*)$. Now we try to rewrite $t_r^*[-k]$ in a similar way: $t_r^*[-k] = t_r^*[n, n + k] = E(r_n r_{n+k}^*)^* = E(r_n^* r_{n+k}) = E(r_{n-k}^* r_n) = E(r_n r_{n-k}^*) = t_r[n, n - k]$. In the end, $t_r^*[-k] = t_r[k]$ and therefore $t_r^*[k] = t_r[-k]$. q.e.d.

Property 3: Minimizing the cost function $J = E(e_n e_n^*) = E(|e_n|^2)$ is equivalent to $E(r_{n-k} e_n^*) = 0$ (*principle of orthogonality*). Hereby, w_k^* are the coefficients of an adaptive filter with the output $y_n = \sum_{k=0}^{\infty} w_k^* r_{n-k}$, the sampled error to be minimized is $e_n = d_n - y_n$ and the desired impulse response is d_n .

Proof: With the k th filter coefficient being $w_k = a_k + j b_k, k = 0, 1, 2, \dots$ a gradient operator ∇_k can be defined as $\nabla_k = \frac{\partial}{\partial a_k} + j \frac{\partial}{\partial b_k}, k = 0, 1, 2, \dots$. The multidimensional complex gradient vector is defined as $\nabla(J)$ whose k th element is $\nabla_k(J) = \frac{\partial J}{\partial a_k} + j \frac{\partial J}{\partial b_k}, k = 0, 1, 2, \dots$. For the cost function J to attain its minimum, all elements of $\nabla(J)$ must be zero. Under these conditions, we have the *optimum in the mean-squared-error sense*. With $J = E(e_n e_n^*)$, we obtain $\nabla_k(J) = E\left(e_n^* \frac{\partial e_n}{\partial a_k} + e_n \frac{\partial e_n^*}{\partial a_k} + j e_n^* \frac{\partial e_n}{\partial b_k} + j e_n \frac{\partial e_n^*}{\partial b_k}\right)$. Hereby, $\frac{\partial e_n}{\partial a_k} = -r_{n-k}$, $\frac{\partial e_n}{\partial b_k} = j r_{n-k}$, $\frac{\partial e_n^*}{\partial a_k} = -r_{n-k}^*$ and $\frac{\partial e_n^*}{\partial b_k} = -j r_{n-k}^*$ with $e_n = d_n - y_n$. Substituting all the partial derivatives in $\nabla_k(J)$, we obtain $\nabla_k(J) = -2E(r_{n-k} r_n^*)$. So, $\nabla_k(J) = 0$ is equivalent to $E(r_{n-k} e_n^*) = 0$. q.e.d.

Property 4: The *principle of orthogonality* $E(r_{n-k} e_n^*) = 0$ corresponds to the expression $T_A W_o = P_{\times}$ (*Wiener-Hopf Equation in matrix form*) where W_o denotes the N -by-1 *optimum tap-weight vector* of the transversal filter and P_{\times} the N -by-1 *cross-correlation vector* $P_{\times} = E(R_n d_n^*)$ between the tap inputs of the filter R_n and the desired response d_n .

Proof: With $y_n = \sum_{k=0}^{\infty} w_k^* r_{n-k}$, we obtain

$$E(r_{n-k} e_n^*) = E\left(r_{n-k} \left(d_n^* - \sum_{i=0}^{\infty} w_{oi} r_{n-i}^*\right)\right) = 0 \text{ with } W_o = (w_{o0}, w_{o1}, \dots, w_{o(N-1)}).$$

This corresponds to $\sum_{i=0}^{\infty} w_{oi} E(r_{n-k} r_{n-i}^*) = E(r_{n-k} d_n^*)$ or $T_A W_o = P_{\times}$. q.e.d.

A.2 Convergence properties of the LMS algorithm

Using the observations presented by the upper section, it won't be difficult to understand the convergence properties of the LMS algorithm with and without weighting factors.

The optimum equalizer coefficients W_o for minimizing the MSE are determined from the solution of the set of linear equations

$$T_A W_o = P_{\times}. \quad (\text{A.3})$$

As defined above, T_A is the autocorrelation matrix of the received signal, W_o is the optimum vector of equalizer tap gains and P_\times is the vector of cross-correlations (\rightarrow **Property 4**). Using the LMS algorithm, W_o is approximated iteratively:

$$W_{k+1} = W_k - \mu G_k \quad (\text{A.4})$$

$$= (I - \mu T_A) W_k + \mu P_\times \quad (\text{A.5})$$

with

$$G_k = \frac{1}{2} \cdot \frac{\partial J}{\partial W_k} \quad (\text{A.6})$$

$$= T_A W_k - P_\times \quad (\text{A.7})$$

$$= E(e_n R_n^*). \quad (\text{A.8})$$

R_n is the vector of the received signal samples, μ is the step size and J the cost function.

Now, we are going to decouple the equations by performing a linear transformation. We note that the matrix T_A is hermitian and, hence, can be represented as

$$T_A = U \Lambda U^H \quad (\text{A.9})$$

where U is the normalized modal matrix of T_A and Λ is a diagonal matrix containing the eigenvalues of T_A .

Now, we define the transformed (*orthogonalized*) vectors

$$W_k^o = U^H W_k \quad (\text{A.10})$$

and

$$P_\times^o = U^H P_\times. \quad (\text{A.11})$$

We obtain

$$W_{k+1}^o = (I - \mu \Lambda) W_k^o + \mu P_\times^o. \quad (\text{A.12})$$

Their convergence properties are determined from

$$W_{k+1}^o = (I - \mu \Lambda) W_k^o. \quad (\text{A.13})$$

The recursive relation will converge provided that

$$|1 - \mu\lambda_k| < 1. \quad (\text{A.14})$$

In other words, μ must satisfy the inequality

$$0 < \mu < \frac{2}{\lambda_{max}} \quad (\text{A.15})$$

where λ_{max} is the large eigenvalue of T_A . We obtain rapid convergence when $|1 - \mu\lambda_k|$ is small. This cannot be achieved if there is a large difference between the smallest eigenvalue λ_{min} and the large eigenvalue λ_{max} of T_A . In the end, even if we select μ near the upper bound given by (A.15), the convergence rate is determined by the smallest eigenvalue λ_{min} . Expressed in a different way, the ratio $\frac{\lambda_{max}}{\lambda_{min}}$ determines the convergence rate, since only if $|1 - \mu\lambda_k|$ is small for *all* $\mu\lambda_k$, the overall convergence will be fast.

Using the *weighted sub-band adaptive filter* (WSAF) algorithm, the weights play the role of normalization coefficients such that the eigenvalue spread is reduced. Therefore, faster convergence can be obtained. A more detailed discussion of the convergence properties can be found in de Courville [10, 11].

Appendix B

A practical implementation of the WSAF algorithm

Here, some practical issues with regard to an implementation of the WSAF update algorithm are discussed.

B.1 Switching the inputs to the FFTs/IFFTs

As it has been discussed in chapter 6, a practical implementation of the FFT or IFFT respectively requires the first transmitted value at the beginning and the last transmitted value at the end of the input vector (i.e. a scenario that is exactly vice versa).

This fact must be taken into account by switching the inputs to the FFTs/IFFTs, since the result of a convolution/correlation is unfortunately not *exactly* the same when all vectors are flipped:

$$IFFT \left(FFT \left(\begin{pmatrix} r_0(n) \\ r_1(n) \\ \vdots \\ r_{N-1}(n) \end{pmatrix} \right) \odot FFT \left(\begin{pmatrix} e_0(n) \\ e_1(n) \\ \vdots \\ e_{N-1}(n) \end{pmatrix} \right) \right) \quad (B.1)$$

\neq

$$FLIP \left(IFFT \left(FFT \left(\begin{pmatrix} r_{N-1}(n) \\ r_{N-2}(n) \\ \vdots \\ r_0(n) \end{pmatrix} \right) \odot FFT \left(\begin{pmatrix} e_{N-1}(n) \\ e_{N-2}(n) \\ \vdots \\ e_0(n) \end{pmatrix} \right) \right) \right), \quad (B.2)$$

where *FLIP* is an operator for turning a vector:



$$FLIP \begin{pmatrix} r_{N-1}(n) \\ \vdots \\ r_1(n) \\ r_0(n) \end{pmatrix} = \begin{pmatrix} r_0(n) \\ r_1(n) \\ \vdots \\ r_{N-1}(n) \end{pmatrix}. \quad (\text{B.3})$$

B.2 Standard definitions of FFT/IFFT operations

Our definition of the matrix performing the FFT operation

$$F_N = \frac{1}{\sqrt{N}} \cdot (W_N^{lk}) \quad (\text{B.4})$$

with

$$W_N^{lk} = e^{-j\frac{2\pi}{N}lk}, 0 \leq l \leq N-1, 0 \leq k \leq N-1 \quad (\text{B.5})$$

is usually substituted by

$$\tilde{F}_N = (W_N^{lk}). \quad (\text{B.6})$$

In this case, the matrix $(\tilde{F}_N)^H$ doesn't correspond to the inverse any more:

$$\tilde{F}_N \cdot \tilde{F}_N^H = N \cdot I_N \quad (\text{B.7})$$

as it did before:

$$F_N \cdot F_N^H = I_N. \quad (\text{B.8})$$

The matrix multiplication $F_{N \times \frac{N}{2}} \cdot (\dots)$ can be done by using a FFT. Hereby, the first $\frac{N}{2}$ values in the frequency domain are kept, the rest is truncated. The multiplication $F_{N \times \frac{N}{2}}^H \cdot (\dots)$ is done by an IFFT. Hereby, the $\frac{N}{2}$ values in the frequency domain are used for the $\frac{N}{2}$ first inputs of the IFFT, the remaining $\frac{N}{2}$ inputs are fed with zeros.

Appendix C

Software description of the ADSL simulator in C/C++

The appendixes C to H present a top-level description of the ADSL-SIMULATOR project. It provides an overview of the C/C++ program architecture. Each module is briefly described. A hierarchical diagram of the architecture is drawn-up.

The ADSL-SIMULATOR intends to simulate the *Time Domain Equalizer* (TEQ) training, two different kinds of transmitters (*central-office transmitter* and *home transmitter*), a channel and the *home* receiver. Therefore, the program can be divided into 5 parts (→ appendix D to H):

- the *central office* transmitter;
- the *home* transmitter;
- the equalizer training;
- the channel (for the *central-office* data);
- the *home* receiver.

The relevant part for DSP porting is of course the receiver function. An effort is made to separate the 5 parts of the program into independent modules. The following annexes will present the simulator in detail:

Appendix D presents a top-level description of the ADSL simulator.

Appendix E presents a module-level description of the ADSL simulator.

Appendix F presents a function-level description of the ADSL simulator.

Appendix G presents the parameter file determining the number of bits per OFDM/DMT carrier used by the ADSL simulator.

Appendix H comments the output of the ADSL simulator.

Appendix D

Top-level diagram of the ADSL simulator

On figure D.1, the architecture of the various modules is drawn-up.

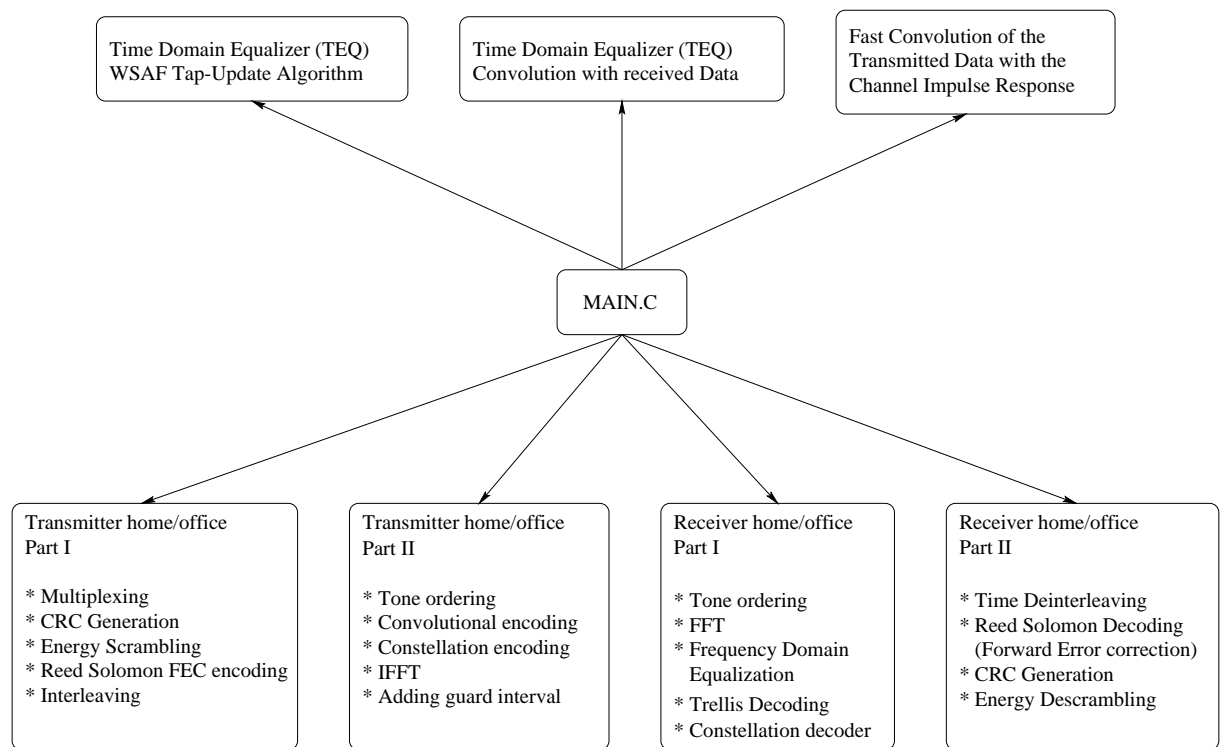


Figure D.1: Top level diagram of the various modules of ADSL-SIMULATOR.

The seven modules presented above are packed into sub-functions and called by the *main* function of the ADSL simulator.

Appendix E

Module-level description of the ADSL simulator

E.1 simulator.C/.h

This module contains only one function: main().
main() controls the sequencing of actions:

- parameters initialization with generate_Superframe_Properties_CO()/ generate_Superframe_Properties_Home() (transmission parameters) and generate_bits_per_tone_CO()/ generate_bits_per_tone_Home() (Defining the number of bits per available tone);
- memory allocations with allocate_memory();
- Defining the data in the command buffer using generate_LS_Buffer();
- Creating the time domain data, simulating a channel, calculation of the time/frequency domain equalizer filter taps using the WSAF algorithm, receiving the data;
- Error Control of the received and decoded data.

E.2 algor_enc.C/.h

This module contains the algor_enc class which performs the constellation encoding.

E.3 channel.C/.h

This module contains the Time Domain Equalizer (TEQ) taps calculation functions (using the WSAF algorithm) and the function performing the convolution *Received noisy data * TEQ*.

E.4 conv_encoder.C/h

This module contains the conv_encoder class which performs the convolutional encoding.

E.5 coset_select.C/h

This module contains the coset_select class which performs the encoding of the two bits per tone determining the coset.

E.6 CRC.C/h

This module contains the CRC class which performs the generation of the byte for the (c)yclic (r)edundancy (c)heck.

E.7 cvector.C/h

This module contains the cvector class which defines an complex vector class and some useful operators for it (addition, ...).

E.8 decoder.C/h

This module contains the decoder class which calls the viterbi decoder function in order to perform the maximum likelihood decoding of one DMT symbol.

E.9 DEINTERL.C/h

This module contains the DEINTERLEAVER class which performs the deinterleaving.

E.10 Descrambler.C/h

This module contains the Descrambler class which performs the energy descrambling.

E.11 FEC.C/h

This module contains the Reed-Solomon forward-error-correction encoder and decoder class.



E.12 fft.C/h

This module contains the *fast fourier transformation* (FFT) and *inverse fast fourier transform* (IFFT) functions.

E.13 generate.C/h

This module contains some initialization and memory allocation functions.

E.14 INTERL.C/h

This module contains the INTERLEAVER class which performs the interleaving.

E.15 ivector.C/h

This module contains the cvector class which defines an integer vector class and some useful operators for it (addition, ...).

E.16 my_types.C/h

This module defines a complex structure and some useful commands for it (get real part, get imaginary part, division, multiplication, ...).

E.17 receiver_home_part1.C/h

This module contains the first part of the home-receiver (Tone ordering, FFT, frequency domain equalization, Trellis Decoding, Constellation decoding). Here, only one DMT symbol is affected.

E.18 receiver_home_part2.C/h

This module contains the second part of the home-receiver (Deinterleaving, Reed Solomon Decoding (forward error correction), CRC Generation for the decoded (probably erroneous) data, Energy Descrambling). Here, one superframe is affected.

E.19 routines.C/h

This module contains some helpful commands like conversion bit->integer, ... and a substitution for the *memset* and *memcpy* command (to be (de)activated via the *USE_MEMMOVE_OF_STRING_H* #define in *switch.h*).

E.20 Scrambler.C/h

This module contains the Scrambler class which performs the energy scrambling.

E.21 tone_order.C/h

This module contains the functions for the tone ordering.

E.22 tools.C/h

This module contains the text-output functions which are called when not enough parameters are specified or in similar cases. Moreover there are some helpful functions like finding a certain bit in a string, etc...

E.23 transmitter_CO_part1.C/h

This module contains the central office transmitter, part 1 (Multiplexing, CRC Generation, Energy Scrambling, Reed-Solomon Encoding, Interleaving). Here, a whole superframe is affected.

E.24 transmitter_CO_part2.C/h

This module contains the central office transmitter, part 2 (Tone Ordering, Trellis encoding, Constellation encoding, IFFT, Adding the Guard Interval). Here, only one DMT symbol is affected.

E.25 viterbi_decoder.C/h

This module contains the viterbi_decoder class which performs a 4-D trellis decoding of a message being encoder with Wei's encoder.

E.26 wei_encoder.C/h

This module contains the wei_encoder class which performs the convolutional encoding using Wei's code.

E.27 constants.h

This h-file contains the constants used by the simulator.

E.28 debug.h

This h-file contains the Error()-#define which is very helpful for the debugging.

E.29 define.h

This h-file contains some general constant definitions and the definitions of the structures used by the simulator.

E.30 switch.h

This h-file contains all compiler-switches for the simulator.

Appendix F

Function-level description of the ADSL simulator

F.1 How to read this description

The description aims at helping the porting of the algorithms, at supporting the debugging of potential errors and at facilitating the adding of new functions. Therefore, several types of parameters are distinguished:

- The 'Input vars' field is a list of the variables which are not modified by the function (read-only parameters)
- The 'Output vars' field is a list of write-only variables
- The 'I/O vars' field gathers the other parameters passed to the function

F.2 Transmitter

F.2.1 `algor_enc.C`

`algor_enc::algor_enc(char *algor_name)`

Author: Kiaei/Markus Muck

Description: This is the constructor of the `algor_enc` class. Here, the coset file table is loaded and local variables reset to zero.

Input vars: `char *algor_name`: *Any name given to the object.*

Output vars: none

I/O vars: none



algor_enc::algor_enc()

Author: Kiaei/Markus Muck

Description: This is the constructor of the algor_enc class. Here, the coset file table is loaded and local variables reset to zero.

Input vars: none

Output vars: none

I/O vars: none

algor_enc::~algor_enc()

Author: Kiaei/Markus Muck

Description: This is the destructor of the algor_enc class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

void algor_enc::output(int input_vector[], int bits, int I[], int Q[],int& x_value, int& y_value)

Author: Kiaei/Markus Muck

Description: Here, the constellation encoding for one carrier is done.

Input vars: int input_vector[]: *Bits to be encoded*
int bits: *number of bits to be encoded*Output vars: int I[]: *Encoded bits real*
int Q[]: *Encoded bits imaginary*
int& x_value: *Encoded symbol real*
int& y_value: *Encoded symbol imaginary*

I/O vars: none

void algor_enc::display(int tonenumber)

Author: Kiaei/Markus Muck

Description: This function writes the results of the constellation encoder to the standard output device.

Input vars: int tonenumber: *Number of Tone/Carrier*

Output vars: none

I/O vars: none

F.2.2 conv_encoder.C**conv_encoder::conv_encoder(char *conv_encoder_name)**

Author: Kiaei/Markus Muck

Description: This is the constructor of the conv_encoder class. Here, the encoder states are reset to zero.

Input vars: char *conv_encoder_name: *Any name given to the object.*

Output vars: none

I/O vars: none

conv_encoder::conv_encoder()

Author: Kiaei/Markus Muck

Description: This is the constructor of the conv_encoder class. Here, the encoder states are reset to zero.

Input vars: none

Output vars: none

I/O vars: none

conv_encoder::~~conv_encoder()

Author: Kiaei/Markus Muck

Description: This is the destructor of the conv_encoder class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

conv_encoder::output(int *U, int *S)

Author: Kiaei/Markus Muck

Description: This function performs the convolutional encoding. The incoming data is U[1] and U[2] (U[0] contains a dummy value, → T1E1 6.6.2, Figure 10), the latest state S[0..3]. After the transition, the new encoder state is written into S[0..3].

Input vars: int *U: *Incoming data in U[1] and U[2] (→ T1E1 6.6.2, Figure 10)*

Output vars: none

I/O vars: int *S: *Latest/new encoder state*



conv_encoder::display()

Author: Kiaei/Markus Muck

Description: This functions is for debugging only. The latest encoder state is displayed.

Input vars: none

Output vars: none

I/O vars: none

F.2.3 coset_select.C**coset_select::coset_select(char *coset_select_name)**

Author: Kiaei/Markus Muck

Description: This is the constructor of the coset_select class. Here, the variables containing the latest incoming data U[0..3] (\rightarrow T1E1 6.6.2, Figure 10) are reset to zero.Input vars: char *coset_name: *Any name given to the object.*

Output vars: none

I/O vars: none

coset_select::coset_select()

Author: Kiaei/Markus Muck

Description: This is the constructor of the coset_select class. Here, the variables containing the latest incoming data U[0..3] (\rightarrow T1E1 6.6.2, Figure 10) are reset to zero.

Input vars: none

Output vars: none

I/O vars: none

coset_select::~~coset_select()

Author: Kiaei/Markus Muck

Description: This is the destructor of the coset_select class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

coset_select::output(int u[],int *v,int *w)

Author: Kiaei/Markus Muck

Description: Here, the coset of the symbol corresponding to the convolutionally encoded data $u[0..3]$ is calculated (\rightarrow T1E1, 6.6.2., the (v_0, v_1) and (w_1, w_0) are calculated corresponding to the two 2-D cosets).

Input vars: int u[]: *Incoming data in u[0..3]*

Output vars: int *v: *Only v[0], v[1] corresponding to (v_0, v_1) are written*
int *w: *Only w[0], w[1] corresponding to (w_0, w_1) are written*

I/O vars: none

coset_select::display()

Author: Kiaei/Markus Muck

Description: This functions is for debugging only. The latest incoming data is displayed.

Input vars: none

Output vars: none

I/O vars: none

F.2.4 CRC.C**CRC::CRC(char *CRC_name)**

Author: Kiaei/Markus Muck

Description: This is the constructor of the CRC class. Here, the shift registers are reset to zero.

Input vars: char *CRC_name: *Any name given to the object.*

Output vars: none

I/O vars: none

CRC::CRC()

Author: Kiaei/Markus Muck

Description: This is the constructor of the CRC class. Here, the shift registers are reset to zero.

Input vars: none

Output vars: none

I/O vars: none



CRC::~CRC()

Author: Kiaei/Markus Muck

Description: This is the destructor of the CRC class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

CRC::init(int value)

Author: Kiaei/Markus Muck

Description: This functions allows to set the shift registers of the cyclic redundancy check to a certain value.

Input vars: int value: *Value* $\in (0, 1, \dots, 255)$ of the encoder state.

Output vars: none

I/O vars: none

int CRC::output(int datain)

Author: Kiaei/Markus Muck

Description: An arriving bit *datain* $\in (0, 1)$ enters the encoder. The resulting encoder state is returned.

Input vars: int datain: *Bit entering the encoder*

Output vars: return-value: *Resulting encoder state*

I/O vars: none

int CRC::showreg()

Author: Kiaei/Markus Muck

Description: This functions returns the latest encoder state.

Input vars: none

Output vars: return-value: *Latest encoder state*

I/O vars: none

F.2.5 FEC.C**FEC::FEC()**

Author: Kiaei/Markus Muck

Description: This is the constructor of the FEC class. Right now, it is empty, but may be used in future.

Input vars: none

Output vars: none

I/O vars: none

FEC::~~FEC()

Author: Kiaei/Markus Muck

Description: This is the destructor of the FEC class. Right now, it is empty, but may be used in future.

Input vars: none

Output vars: none

I/O vars: none

int FEC::mul(int a, int b)

Author: Kiaei/Markus Muck

Description: This function performs a galois-field-multiplication.

Input vars: int a: *First operand*
int b: *Second operand*Output vars: return-value: *Result of multiplication a · b*

I/O vars: none

int FEC::inv(short int a)

Author: Kiaei/Markus Muck

Description: This function performs a galois-field-inversion.

Input vars: short int a: *Value to be inverted*Output vars: return-value: *Inverted value*

I/O vars: none



int FEC::add(int a , int b)

Author: Kiaei/Markus Muck

Description: This function performs a galois-field-addition.

Input vars: int a: *First operand*
int b: *Second operand*Output vars: return-value: *Result of addition a + b*

I/O vars: none

int FEC::Syndrome(unsigned char *Bad_Data, int N, int t)

Author: Kiaei/Markus Muck

Description: Calculates the error syndrome. If zero, there is no error to be found.

Input vars: unsigned char *Bad_Data: *Pointer to the received data*
int N: *Codeword length in bytes*
int t: *The redundancy per codeword is $2 \cdot t$* Output vars: return-value: *0 if no error detected, 1 if data erroneous*

I/O vars: none

FEC::Euclid()

Author: Kiaei/Markus Muck

Description: Calculates the (G)reatest (C)ommom (D)ivisor (GCD) of two polynomials.

Input vars: none

Output vars: none

I/O vars: none

int FEC::Chien()

Author: Kiaei/Markus Muck

Description: This function finds all error locations by stepping through all the 255 possible locations (0-254).

Input vars: none

Output vars: none

I/O vars: none

FEC::Value()

Author: Kiaei/Markus Muck

Description: uses the GCD remainder and the Error_locator_polynomial to find Values.

Input vars:

Output vars: none

I/O vars: none

FEC::Value_Correct(unsigned char *Bad_Data, int N)

Author: Kiaei/Markus Muck

Description: Given the locations and values found this function corrects the bad bytes.

Input vars: unsigned char *Bad_Data: *Pointer to the received data*
int N: *Codeword length in bytes*

Output vars: none

I/O vars: none

int FEC::Encode(int byte,int cont, int addr, int Rsel)

Author: Kiaei/Markus Muck

Description: This function performs the Reed-Solomon Encoding.

Input vars: int byte: *Latest byte to be encoded*int cont: *Control-sequence:*

$\left\{ \begin{array}{l} FEC_CONTROL_IDLE \\ FEC_CONTROL_READ_DATA_AND_DIVIDE \\ FEC_CONTROL_SHIFT_OUT_FROM_MEMORY \\ FEC_CONTROL_SHIFT_IN_FROM_MEMORY \\ FEC_CONTROL_SHIFT_OUT_CHECK_BYTES \end{array} \right.$	<i>idle</i> <i>read data and divide</i> <i>shift out to memory (add reg #0-19)</i> <i>shift in from memory</i> <i>shift out check bytes</i>
---	---

Output vars: return-value: *Redundancy bytes when using the
FEC_CONTROL_SHIFT_OUT_CHECK_BYTES control sequence*

I/O vars: none

int FEC::lookuplog(int value)

Author: Kiaei/Markus Muck

Description: Returns the log of a Galois field element.

Input vars: int value: *A value as parameter*Output vars: return-value: *The log of the parameter is returned*

I/O vars: none



int FEC::lookupvalue(int log)

Author: Kiaei/Markus Muck

Description: Returns a Galois field value given a log as input.

Input vars: int log: *A value as parameter*Output vars: return-value: *Galois field value given a log as input*

I/O vars: none

F.2.6 fft.C**void fft(cvector &x, int N, window_type window)**

Author: Kiaei/Markus Muck

Description: This function performs a windowing operation ($window \in (rectangle, Hamming, Hanning, Bartlett, Blackman, Blackman_Harris)$) and afterwards the *fast fourier transform* using *fftwor*.Input vars: int N: *Number of taps*window_type window: *window type $\in (rectangle, Hamming, Hanning, Bartlett, Blackman, Blackman_Harris)$*

Output vars: none

I/O vars: cvector &x: *Time domain data, will be overwritten with the frequency domain data***void ifft(cvector &x, int N)**

Author: Kiaei/Markus Muck

Description: This function performs the *inverse fast fourier transform* using *fftwor*.Input vars: int N: *Number of taps*

Output vars: none

I/O vars: cvector &x: *Frequency domain data, will be overwritten with the time domain data***void fftwork(int M, cvector &x, int inverse_flag, int N2)**

Author: Kiaei/Markus Muck

Description: This function *really* performs the (*inverse*) *fast fourier transform*.Input vars: int M: $M = \log_2(FFT\ size)$ int inverse_flag: *0 for FFT, 1 for IFFT*int N2: *Dummy, may have any value*

Output vars: none

I/O vars: cvector &x: *Time domain data, will be overwritten with the frequency domain data*

void Hamming_w(cvector &x, int n)

Author: Kiaei/Markus Muck

Description: This function performs the windowing operation using the *Hamming* algorithm.Input vars: int n: *Number of taps*

Output vars: none

I/O vars: cvector &x: *Data to be windowed. Will be overwritten with the windowed data***void Hanning_w(cvector &x, int n)**

Author: Kiaei/Markus Muck

Description: This function performs the windowing operation using the *Hanning* algorithm.Input vars: int n: *Number of taps*

Output vars: none

I/O vars: cvector &x: *Data to be windowed. Will be overwritten with the windowed data***void Bartlett_w(cvector &x, int n)**

Author: Kiaei/Markus Muck

Description: This function performs the windowing operation using the *Bartlett* algorithm.Input vars: int n: *Number of taps*

Output vars: none

I/O vars: cvector &x: *Data to be windowed. Will be overwritten with the windowed data***void Blackman_w(cvector &x, int n)**

Author: Kiaei/Markus Muck

Description: This function performs the windowing operation using the *Blackman* algorithm.Input vars: int n: *Number of taps*

Output vars: none

I/O vars: cvector &x: *Data to be windowed. Will be overwritten with the windowed data*

void Blackman_Harris_w(cvector &x, int n)

Author: Kiaei/Markus Muck

Description: This function performs the windowing operation using the *Blackman-Harris* algorithm.

Input vars: int n: *Number of taps*

Output vars: none

I/O vars: cvector &x: *Data to be windowed. Will be overwritten with the windowed data*

F.2.7 INTERLEAVER.C

INTERLEAVER::INTERLEAVER()

Author: Markus Muck

Description: This is the constructor of the INTERLEAVER class. Here, the interleaving memory is allocated and reset to zero.

Input vars: none

Output vars: none

I/O vars: none

INTERLEAVER::~~INTERLEAVER()

Author: Markus Muck

Description: This is the destructor of the INTERLEAVER class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

int INTERLEAVER::DO_INTERLEAVING(unsigned char *Data_out, unsigned char* Data, int nbr_frames, int Nb_Bytes_CodeWord, int depth)

Author: Markus Muck

Description: This function performs the interleaving using a circular buffer.

Input vars: unsigned char* Data: *Data to be interleaved*
int nbr_frames: *Number of codeword frames to be interleaved*
int Nb_Bytes_CodeWord: *Number of bytes per codeword frame*
int dept: *Interleaving depth*

Output vars: unsigned char *Data_out: *Interleaved data*

I/O vars: none

F.2.8 Scrambler.C

Scrambler::Scrambler(char *scrambler_name)

Author: Kiaei/Markus Muck

Description: This is the constructor of the Scrambler class. Here, the registers are reset to zero.

Input vars: char *scrambler_name: *Any name given to the object.*

Output vars: none

I/O vars: none

Scrambler::Scrambler()

Author: Kiaei/Markus Muck

Description: This is the constructor of the Scrambler class. Here, the registers are reset to zero.

Input vars: none

Output vars: none

I/O vars: none

Scrambler::~~Scrambler()

Author: Kiaei/Markus Muck

Description: This is the destructor of the Scrambler class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

Scrambler::init(int value)

Author: Kiaei/Markus Muck

Description: This functions allows to set the registers of the scrambler to a certain value.

Input vars: int value: *Value $\in (0, 1, \dots, 2^{23} - 1)$ of the scrambler state.*

Output vars: none

I/O vars: none



int Scrambler::output(int datain)

Author: Kiaei/Markus Muck

Description: An arriving bit $datain \in (0, 1)$ enters the scrambler. The resulting scrambler state is returned.Input vars: int datain: *Bit entering the scrambler*Output vars: return-value: *Resulting scrambler state*

I/O vars: none

int Scrambler::showreg()

Author: Kiaei/Markus Muck

Description: This functions returns the latest scrambler state.

Input vars: none

Output vars: return-value: *Latest encoder state*

I/O vars: none

int Scrambler::display()

Author: Kiaei/Markus Muck

Description: This functions writes the scrambler name to the standard output stream. It's for debugging only.

Input vars: none

Output vars: none

I/O vars: none

F.2.9 tone_order.C**void do_tone_ordering(unsigned int *tone_nr, unsigned int *nbr_bits, unsigned int *nbr_tones_used)**

Author: Markus Muck

Description: This function reads the file defining the number of bits per tone. The file name is defined by FILE_BITS_PER_TONE (\rightarrow *constants.h*).

Input vars: none

Output vars: unsigned int *tone_nr: *In the end, there are “unsigned int *nbr_tones_used” tones carrying at least two bits. All these tones are ordered, beginning with the tones carrying the least number of bits (but ≥ 2). “tone_nr[x]” contains the tone-number of the “x”th sorted tone*

unsigned int *nbr_bits: *In the end, this array contains the number of bits for each tone carrying at least two bits. **Attention:** If a tone carries less than two bits, it won't be used and is therefore **not** written into this array. In order to identify the different tones, the array “unsigned int *tone_nr” is used. “nbr_bits[x]” contains the number of bits of the “x”th sorted tone*

unsigned int *nbr_tones_used: *Number of tones carrying at least two bits*

I/O vars: none

void read_tone(FILE *Handle, unsigned int *Buffer)

Author: Markus Muck

Description: This function is used by *do_tone_ordering* and reads the number of bits per tone from a text file.

Input vars: FILE *Handle: *File handle*

Output vars: unsigned int *Buffer: *Buffer containing the number of bits per tone*

I/O vars: none

F.2.10 transmitter_CO_part1.C

void transmitter_central_offi ce_part1(unsigned char *InputDataAS, unsigned int DataToTransmitAS, unsigned char *InputDataLS, unsigned int DataToTransmitLS, unsigned char *Superframe_Memory_AS, unsigned char *Superframe_Memory_LS, unsigned int *DataTransmittedAS, unsigned int *DataTransmittedLS, unsigned char *Redundancy_AS, unsigned char *Redundancy_LS, unsigned char *Data_Interleaved, struct Sframe_Properties_CO Superframe_Properties_CO, unsigned short int *crc_check, int *trellis_states)

Author: Markus Muck

Description: This function performs the first part of the transmission process (*Multiplexing, (C)yclic (R)edundancy (C)heck calculation, Reed-Solomon encoding, Scrambling, Time Interleaving*). Here, always one superframe is treated at a time.

Input vars: unsigned char *InputDataAS: *Data to be transmitted in the ASX part*
 unsigned int DataToTransmitAS: *Number of bytes to be transmitted*
 unsigned char *InputDataLS: *Data to be transmitted in the LSX part*
 struct Sframe_Properties_CO Superframe_Properties_CO: *Structure containing the properties*

Output vars: unsigned char *Data_Interleaved: *Interleaved data, to be used by transmitter_central_offi ce_part2*

I/O vars: unsigned char *Superframe_Memory_AS: *Buffer used internally in order to build up a superframe*
unsigned int *DataTransmittedAS: *Data already transmitted in the ASX part*
unsigned int *DataTransmittedLS: *Data already transmitted in the LSX part*
unsigned char *Redundancy_AS: *Buffer used internally for the ASX redundancy data created by the Reed-Solomon encoder*
unsigned char *Redundancy_LS: *Buffer used internally for the LSX redundancy data created by the Reed-Solomon encoder*
unsigned short int *crc_check: *Latest state of the CRC encoder*
int *trellis_states: *Latest state of the convolutional encoder. Must be reset to zero for each DMT symbol.*

F.2.11 transmitter_CO_part2.C

```
void transmitter_central_offi ce_part2( unsigned char *Data_Interleaved,  
struct Sframe_Properties_CO Superframe_Properties_CO,  
unsigned int *nbr_bits_transmitted, double *TimeDomainData, int *trellis_states)
```

Author: Markus Muck

Description: This function performs the second part of the transmission process (*Tone ordering, Convolutional encoding, Constellation encoding, IFFT, Adding of the guard interval*). Here, only one DMT symbol is treated at a time.

Input vars: unsigned char *Data_Interleaved: *Interleaved data created by transmitter_central_offi ce_part1*
struct Sframe_Properties_CO Superframe_Properties_CO: *Structure containing the properties*

Output vars: double *TimeDomainData: *The time domain data of one DMT symbol. This buffer contains only the real parts, since the imaginary parts are all zero.*

I/O vars: unsigned int *nbr_bits_transmitted: *Number of already transmitted bits*
int *trellis_states: *Latest state of the convolutional encoder*

F.2.12 wei_encoder.C

```
wei_encoder::wei_encoder()
```

Author: Kiaei/Markus Muck

Description: This is the constructor of the wei_encoder class. Here, the encoder state is reset to zero and the standard class name “Wei” is defined.

Input vars: none

Output vars: none

I/O vars: none

wei_encoder::~~wei_encoder()

Author: Kiaei/Markus Muck

Description: This is the destructor of the `wei_encoder` class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

wei_encoder::clear_states()

Author: Kiaei/Markus Muck

Description: This function resets the state of the encoder to zero.

Input vars: none

Output vars: none

I/O vars: none

wei_encoder::set_states(int init_value)

Author: Kiaei/Markus Muck

Description: This function resets the state of the encoder to the value *init_value*.

Input vars: `int init_value`: *Desired state of the encoder* $\in (0, 1, \dots, 15)$

Output vars: none

I/O vars: none

int wei_encoder::output(int U[], int *S, int size1, int size2, complex *Point1, complex *Point2)

Author: Kiaei/Markus Muck

Description: This function performs the convolutional encoding of the arriving data $U[1..3] \in (0, 1)$ using the *conv_encoder* class (\rightarrow T1E1 6.6.2, Figure 10). One redundant bit is added to $U[0] \in (0, 1)$. Afterwards, the coset corresponding to $U[1..3]$ is calculated using the *coset_select* class. Finally, the arriving data $U[0..\max]$ are encoded into the symbols for two tones using the *algor_enc* class.

Input vars: `int U[]`: *Data to be encoded with* $U[] \in (0, 1)$
`int size1`: *Number of bits of the first tone*
`int size2`: *Number of bits of the second tone*



Output vars: complex *Point1: *Unscaled encoded symbol for tone 1*
complex *Point2: *Unscaled encoded symbol for tone 2*

I/O vars: int *S: *States of the encoder*

F.3 Channel and TEQ update

F.3.1 channel.C

void LoadChannelImpulseResponse(double **FreqDomainValues, double **FreqDomainValues_1024)

Author: Markus Muck

Description: This function loads the Channel Impulse Response (CIR) time domain samples, truncates the channel delay (i.e. all samples before the peak-sample) and performs a transformation of the samples into frequency domain. Once, the CIR is transformed on the basis of 512 samples (\rightarrow FreqDomainValues) and once, there are 512 zeros added in the time domain and a FFT performed based on 1024 samples. The path and name of the file containing the time domain samples of the CIR is defined by "FILE_CIR" in the file "constants.h".

Input vars: none

Output vars: double **FreqDomainValues: *Memory is reserved for "***FreqDomainValues**" and the frequency domain values based on 512 samples are copied into it*
double **FreqDomainValues_1024: *Memory is reserved for "***FreqDomainValues_1024**" and the frequency domain values based on 1024 samples are copied into it*

I/O vars: none

void Convolution_Test(double *FreqDomainValues)

Author: Markus Muck

Description: This function is used for testing only. The frequency domain data of the Channel Impulse Response (CIR) loaded by "LoadChannelImpulseResponse" are convolved with a sequence of samples which consist of two diracs and zeros for the rest. The resulting samples are re-transformed into time domain.

Input vars: double *FreqDomainValues: *The Channel Impulse Response (CIR) in the frequency domain based on 512 samples*

Output vars: none

I/O vars: none

void Generate_C_REVERB1(double **C_REVERB1_Freq, double **C_REVERB1_Time)

Author: Markus Muck

Description: This function generates the “C-REVERB1” symbol defined by the ADSL standard based on 512 samples. Memory is reserved for “*C_REVERB1_Freq” and “*C_REVERB1_Time” and the resulting frequency and time domain samples are copied into these two arrays.

Input vars: none

Output vars: double **C_REVERB1_Freq: *Frequency domain samples of the “C-REVERB1” symbol*
double **C_REVERB1_Time: *Time domain samples of the “C-REVERB1” symbol*

I/O vars: none

void Calculate_Noise_Amplitude(double SNR_dB, double *C_REVERB1_Freq, double *CIR_FreqDomainValues, double *Noise_Amplitude, double **Estimated_Channel_Freq)

Author: Markus Muck

Description: This function calculates (CIR convolved C-REVERB1) and its resulting energy. The energy is divided by the number of carriers (512 usually). The noise amplitude per time domain sample corresponding to the given SNR is calculated. Additionally, the channel estimation is performed (noisy environment). The estimated channel does therefore not EXACTLY correspond to the true channel. The channel is estimated using NBR_SYMBOLS_FOR_CHANNEL_ESTIMATION arriving symbols in order to make the influence of the noise smaller.

Input vars: double SNR_dB: *The desired SNR of the transmission in dB*
double *C_REVERB1_Freq: *The “C-REVERB1”-symbol in frequency domain*
double *CIR_FreqDomainValues: *The frequency domain values of the Channel Impulse Response (CIR) based on 512 samples*

Output vars: double *Noise_Amplitude: *Amplitude of the noise samples*
double **Estimated_Channel_Freq: *Memory is reserved for “*Estimated_Channel_Freq” and the channel - estimated in a noisy environment - is copied into it*

I/O vars: none

void Calculate_Filter_Coeff(double *C_REVERB1_Freq, double *C_REVERB1_Time, double *CIR_FreqDomainValues, double Noise_Amplitude, double **TEQ_FilterCoeff, double **TEQ_Filter_Coeff_1024_Freq, complex *Coeff_Equalization_Freq_Domain)

Author: Markus Muck

Description: This function determines the optimal filter coefficients of the Time Domain Equalizing (TEQ) filter using the WSAF algorithm in order to calculate the filter tap updates. The calculation is split up into 9 blocks: Initialization of the variables (calculation of

the weighting factors, initialization of the Time Domain Equalizing (TEQ) and Target Impulse Response (TIR) filters, etc.), Convolution *Channel Impulse Response (CIR) * Transmitted Data X*, Convolution *Received noisy data R * Time Domain Equalizing (TEQ) filter*, Calculation of the new Target Impulse Response (TIR) filter coefficients $\left(= \frac{R * TEQ}{X} \Big|_{32 \text{ first samples}} \right)$, Convolution *New Target Impulse Response (TIR) filter * Transmitted Data X*, Calculation of the error in the frequency domain based on 1024 samples $(= (TIR * X) - (TEQ * R))$ and re-transformation into time domain (here, the error is limited to 512 samples for the fast correlation), Fast correlation of the *Received noisy data R and the error*, Time Domain Equalizer (TEQ) update (the results of the correlation are multiplied with a constant), calculation of the frequency domain equalization coefficients (since we assume to have a Channel Impulse Response $CIR * TEQ$ which is smaller than the guard interval, there is a circular convolution performed in frequency domain which may be equalized by a coefficient for each carrier): $\frac{1}{TEQ * CIR}$ in the frequency domain.

Input vars: double ****C_REVERB1_Freq**: *Frequency domain samples of the "C-REVERB1" symbol*
 double ****C_REVERB1_Time**: *Time domain samples of the "C-REVERB1" symbol*
 double ***CIR_FreqDomainValues**: *Frequency domain samples of the Channel Impulse Response (CIR)*
 double **Noise_Amplitude**: *Noise amplitude calculated by "Calculate_Noise_Amplitude"*

Output vars: double ****TEQ_FilterCoeff**: *Resulting Time Domain Equalizing (TEQ) filter coefficients in the time domain, memory is reserved for "**TEQ_FilterCoeff"*
 double ****TEQ_Filter_Coeff_1024_Freq**: *Resulting Time Domain Equalizing (TEQ) filter coefficients in the frequency domain, memory is reserved for "**TEQ_Filter_Coeff_1024_Freq"*
 complex ***Coeff_Equalization_Freq_Domain**: *Array for the frequency domain equalization coefficients, the memory for this array is reserved by "TEQ_Filter_Taps_Calculation"*

I/O vars: none

**void Calculate_Energy_After_GI(double *CIR_FreqDomainValues,
 double *TEQ_FilterCoeff_Time)**

Author: Markus Muck

Description: This function may be called after the Time Domain Equalizing (TEQ) filter taps calculation. It performs the convolution $TEQ * CIR$ and calculates the remaining energy of the resulting impulse response after the guard interval (usually 32 taps). This information won't be used in any other function and is for testing purposes only.

Input vars: double ***CIR_FreqDomainValues**: *Frequency domain samples of the Channel Impulse Response (CIR)*
 double ***TEQ_FilterCoeff_Time**: *Resulting Time Domain Equalizing (TEQ) filter coefficients in the time domain*

Output vars: none

I/O vars: none

```
void Convolution_With_Channel_And_TEQ_Plus_AWGN_Noise(
double *Convolved_Data_Time_512, double *CIR_FreqDomainValues_1024, double
*TEQ_FilterCoeff_Freq_1024, double *Input_Data_Time_512, double Noise_Amplitude,
double *Overlap_Data)
```

Author: Markus Muck

Description: This function is used after the Time Domain Equalizing (TEQ) filter update calculation. It uses the resulting Time Domain Equalizing (TEQ) filter coefficients in order to perform the fast convolution operation *Received noisy data R * Time Domain Equalizing (TEQ) filter* using the Add-Overlap algorithm.

Input vars: double *CIR_FreqDomainValues_1024: *The Channel Impulse Response (CIR) in the frequency domain based on 1024 samples*
double *TEQ_FilterCoeff_Freq_1024: *The TEQ filter coefficients in the frequency domain based on 1024 samples*
double *Input_Data_Time_512: *Received noisy samples in the time domain*
double Noise_Amplitude: *Noise amplitude calculated by "Calculate_Noise_Amplitude"*

Output vars: double *Convolved_Data_Time_512: *Resulting samples R * TEQ*

I/O vars: double *Overlap_Data: *Buffer which is used for the Add-Overlap algorithm that is used for the convolution*

```
void TEQ_Filter_Taps_Calculation( double **TEQ_freq_1024,
double **ChannelImpulseResponse_Freq_1024, double **Overlap_Add_Buffer, dou-
ble *Noise_Amplitude, complex **Coeff_Equalization_Freq_Domain, double SNR)
```

Author: Markus Muck

Description: This function performs the calculation of the Time Domain Equalizing (TEQ) filter coefficients by calling the functions presented above: "Generate_C_REVERB1", "LoadChannelImpulseResponse", "Calculate_Noise_Amplitude", "Calculate_Filter_Coeff" and "Calculate_Energy_After_GI".

Input vars: double SNR_dB: *The desired SNR of the transmission in dB*

Output vars: double **TEQ_freq_1024: *The Time Domain Equalizing (TEQ) filter coefficients in the frequency domain based on 1024 samples*
double **ChannelImpulseResponse_Freq_1024: *The Channel Impulse Response (CIR) in the frequency domain based on 1024 samples*
double **Overlap_Add_Buffer: *Buffer which is used for the Add-Overlap algorithm that is used for the convolution*
double *Noise_Amplitude: *The resulting noise amplitude will be copied into this variable*
complex **Coeff_Equalization_Freq_Domain: *Array for the frequency domain equalization coefficients, the memory for this array is reserved by "TEQ_Filter_Taps_Calculation"*

I/O vars: none

F.4 Receiver

F.4.1 decoder.c

decoder::decoder(ivector output_bit_distribution, ivector pointer_table, char *decoder_name)

Author: Kiaei/Markus Muck

Description: This function is the constructor of the decoder class. Here, some memory is allocated for the class name is allocated and the output size ($= \sum (\text{Number of bits per tone})$) is calculated. This constructor initializes as well the *viterbi_decoder* class and two vectors (*bit_distribution* and *permutation*).

Input vars: ivector output_bit_distribution: *Number of bits per tone*
ivector pointer_table: *Real tone number of any ordered tone #x*
char *decoder_name: *Any name given to the decoder class*

Output vars: none

I/O vars: none

decoder::update_decoder(ivector output_bit_distribution, ivector pointer_table, char *decoder_name)

Author: Markus Muck

Description: This function reinitializes the decoder variables. Here, the once reserved memory for the class is freed and some memory is allocated for the class name is allocated and the output size ($= \sum (\text{Number of bits per tone})$) is calculated.

Input vars: ivector output_bit_distribution: *Number of bits per tone*
ivector pointer_table: *Real tone number of any ordered tone #x*
char *decoder_name: *Any name given to the decoder class*

Output vars: none

I/O vars: none

ivector decoder::output(cvector symbols, int number_4D_symbols)

Author: Kiaei/Markus Muck

Description: Performs the trellis decoding using the *tcm_decode* function.

Input vars: cvector symbols: *Symbols to be decoded*
int number_4D_symbols: *Number of useful tones*

Output vars: return-value: *Decoded data*

I/O vars: none

ivector decoder::tcm_decode(cvector symbols, ivector bit_distribution, ivector permutation, int number_4D_symbols)

Author: Kiaei/Markus Muck

Description: Here, the trellis decoding is performed using the functions defined in the *viterbi_decoder* class.

Input vars: cvector symbols: *Symbols to be decoded*
 ivector bit_distribution: *Number of bits per tone*
 ivector permutation: *Real tone number of any ordered tone #x*
 int number_4D_symbols: *Number of useful tones*

Output vars: return-value: *decoded data*

I/O vars: none

F.4.2 DEINTERL.C

DEINTERLEAVER::DEINTERLEAVER()

Author: Markus Muck

Description: This is the constructor of the DEINTERLEAVER class. Here, the deinterleaving memory is allocated and reset to zero.

Input vars: none

Output vars: none

I/O vars: none

DEINTERLEAVER::~DEINTERLEAVER()

Author: Markus Muck

Description: This is the destructor of the DEINTERLEAVER class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

int DEINTERLEAVER::DO_DEINTERLEAVING(unsigned char* Data_out, unsigned char *data_in, int nbr_frames, int Nb_Bytes_CodeWord, int depth, unsigned int *nbr_frames_decoded)

Author: Markus Muck

Description: This function performs the deinterleaving using a circular buffer.

Input vars: unsigned char* data_in: *Data to be deinterleaved*
int nbr_frames: *Number of codeword frames to be interleaved*
int Nb_Bytes_CodeWord: *Number of bytes per codeword frame*
int dept: *Interleaving depth*

Output vars: unsigned char *Data_out: *Deinterleaved data*

I/O vars: none

F.4.3 Descrambler.C

Descrambler::Descrambler(char *Descrambler_name)

Author: Kiaei/Markus Muck

Description: This is the constructor of the Descrambler class. Here, the registers are reset to zero.

Input vars: char *Descrambler_name: *Any name given to the object.*

Output vars: none

I/O vars: none

Descrambler::Descrambler()

Author: Kiaei/Markus Muck

Description: This is the constructor of the Descrambler class. Here, the registers are reset to zero.
The standard name "scram" is used as name for the descrambler class.

Input vars: none

Output vars: none

I/O vars: none

Descrambler::~~Descrambler()

Author: Kiaei/Markus Muck

Description: This is the destructor of the Descrambler class. Here, the memory reserved for the object name is freed.

Input vars: none

Output vars: none

I/O vars: none

Descrambler::init(int value)

Author: Kiaei/Markus Muck

Description: This functions allows to set the registers of the Descrambler to a certain value.

Input vars: int value: *Value* $\in (0, 1, \dots, 2^{23} - 1)$ of the Descrambler state.

Output vars: none

I/O vars: none

int Descrambler::output(int datain)

Author: Kiaei/Markus Muck

Description: An arriving bit *datain* $\in (0, 1)$ enters the Descrambler. The resulting Descrambler state is returned.Input vars: int datain: *Bit entering the Descrambler*Output vars: return-value: *Resulting Descrambler state*

I/O vars: none

int Descrambler::showreg()

Author: Kiaei/Markus Muck

Description: This functions returns the latest Descrambler state.

Input vars: none

Output vars: return-value: *Latest encoder state*

I/O vars: none

int Descrambler::display()

Author: Kiaei/Markus Muck

Description: This functions writes the Descrambler name to the standard output stream. It's for debugging only.

Input vars: none

Output vars: none

I/O vars: none



F.4.4 receiver_home_part1.C

void receiver_home_part1(double *TimeDomainData, struct Sframe_Properties_CO Superframe_Properties_CO, unsigned int received_frame_counter, unsigned char *Superframe_Memory_AS)

Author: Markus Muck

Description: This is the first part of the receiving procedure. This functions performs the *Removal of the guard interval, FFT, Tone ordering, Constellation decoding and Trellis decoding* using the corresponding classes.

Input vars: double *TimeDomainData: *Received time domain data, there are only the real parts. All imaginary parts are zero.*
struct Sframe_Properties_CO Superframe_Properties_CO: *Contains the superframe properties*
unsigned int received_frame_counter: *Which DMT-symbol is about being decoded out of all DMT symbols of one superframe*

Output vars: unsigned char *Superframe_Memory_AS: *The decoded data is written into this buffer*

I/O vars: none

F.4.5 receiver_home_part2.C

void receiver_home_part2(struct Sframe_Properties_CO Superframe_Properties_CO, unsigned char *OutputDataAS, unsigned int *DataTransmittedAS, unsigned char *OutputDataLS, unsigned int *DataTransmittedLS, unsigned char *Superframe_Memory_AS_reception, unsigned char *Superframe_Memory_LS_reception, unsigned short int *crc_check, unsigned int *nbr_bits_received_ext)

Author: Markus Muck

Description:

Input vars: struct Sframe_Properties_CO Superframe_Properties_CO: *Structure containing the properties*
unsigned int *DataTransmittedAS: *Not used right now*
unsigned int *DataTransmittedLS: *Not used right now*
unsigned char *Superframe_Memory_AS_reception: *ASX-data of one DMT symbol as delivered by receiver_home_part1.C*
unsigned char *Superframe_Memory_LS_reception: *LSX-data of one DMT symbol as delivered by receiver_home_part1.C*

Output vars: unsigned char *OutputDataAS: *Buffer for the decoded ASX-data*
unsigned char *OutputDataLS: *Buffer for the decoded LSX-data*
unsigned short int *crc_check: *CRC check byte of the decoded superframe*

I/O vars: unsigned int *nbr_bits_received_ext: *Number of received bytes*

F.4.6 viterbi_decoder.C

viterbi_decoder::viterbi_decoder(int num_4d_symbols, ivector pointer_table, ivector bit_distribution, char *viterbi_name)

Author: Kiaei/Markus Muck

Description: This function is the constructor of the viterbi_decoder class. Here, two vectors are initialized (*permutation*, *BIT_DISTRIBUTION*) and some memory for the class name is allocated. The viterbi_decoder metrics and the path-table is reset to standard values.

Input vars: int num_4d_symbols: *Number of tones carrying at least 2 bits per DMT symbol*
 ivector pointer_table: *Real tone number of any ordered tone #x*
 ivector bit_distribution: *Number of bits per tone*
 char *viterbi_name: *Any name given to the class*

Output vars: none

I/O vars: none

viterbi_decoder::update_viterbi(int num_4d_symbols, ivector pointer_table, ivector bit_distribution, char *viterbi_name)

Author: Markus Muck

Description: This function reinitializes the viterbi decoder.

Input vars: int num_4d_symbols: *Number of tones carrying at least 2 bits per DMT symbol*
 ivector pointer_table: *Real tone number of any ordered tone #x*
 ivector bit_distribution: *Number of bits per tone* char *viterbi_name: *Any name given to the class*

Output vars: none

I/O vars: none

viterbi_decoder::viterbi_decoder()

Author: Kiaei/Markus Muck

Description: This is the constructor of the viterbi_decoder class. The memory allocated for the class name is freed.

Input vars: none

Output vars: none

I/O vars: none

void viterbi_decoder::BMG_2d(double I,double Q, double *metric)

Author: Kiaei/Markus Muck

Description: The metric for a 2-D coset (\rightarrow 1 tone) is calculated.Input vars: double I: *Real part of a encoded constellation*
double Q: *Imaginary part of a encoded constellation*

Output vars: double *metric: Calculated branch metric

I/O vars: none

void viterbi_decoder::BMG_4d(double *metric1, double *metric2)

Author: Kiaei/Markus Muck

Description: Using the result of *BMG_2d* - done for two tones - the 4-D coset metric is calculated. The result is saved in an internal variable.Input vars: double *metric1: *2-D metric of the first tone*
double *metric2: *2-D metric of the second tone*

Output vars: none

I/O vars: none

void viterbi_decoder::ACS(int iteration_number)

Author: Kiaei/Markus Muck

Description: (A)ddition-(C)ompare-(S)elect function. The path table is build up, one column at each iteration.

Input vars: int iteration_number: *Latest iteration number*

Output vars: none

I/O vars: none

void viterbi_decoder::STD(int iteration_number, int number_4D_symbols)

Author: Kiaei/Markus Muck

Description: Tracing the survivor path and calculating the recovered cosets.

Input vars: int iteration_number: *Latest iteration number*
int num_4d_symbols: *Number of tones carrying at least 2 bits*

Output vars: none

I/O vars: none

void viterbi_decoder::CDP(double I1, double Q1, double I2, double Q2,int j)

Author: Kiaei/Markus Muck

Description: Decoding of the message carried by two tones. For the 2-D coset there are 4 decoding possibilities. We take the one which leads to the smallest metric (\rightarrow we use the recovered coset calculated by *STD*).

Input vars: double I1: *Real part of the encoded constellation of tone one*
 double Q1: *Imaginary part of the encoded constellation of tone one*
 double I2: *Real part of the encoded constellation of tone two*
 double Q2: *Imaginary part of the encoded constellation of tone two*
 int j: *Latest iteration number*

Output vars: none

I/O vars: none

ivector viterbi_decoder::output(cvector symbols, int number_4D_symbols)

Author: Kiaei/Markus Muck

Description: Output of the decoded data using the results of *CDP*.

Input vars: cvector symbols: *Encoded constellations of all tones of one symbol DMT*
 int num_4d_symbols: *Number of tones carrying at least 2 bits*

Output vars: none

I/O vars: none

void coset_members(int i,int j,int& offset1,int& offset2)

Author: Kiaei/Markus Muck

Description: Calculating the cosets of two tones ($\rightarrow v[0, 1], w[0, 1]$, T1E1 6.6.2, Figure 10).

Input vars: int i: *Integer value of $U[0:2]$ (\rightarrow T1E1 6.6.2, Figure 10)*
 int j: *Integer value of $U[3]$ (\rightarrow T1E1 6.6.2, Figure 10)*

Output vars: int& offset1: *Resulting coset of tone one as integer value, corresponds to bit-value $v[0, 1]$*
 int& offset2: *Resulting coset of tone two as integer value, corresponds to bit-value $w[0, 1]$*

I/O vars: none



int coset_ident(int I, int Q)

Author: Kiaei/Markus Muck

Description: Identification of the 2-D coset corresponding the encoded constellation I, Q .Input vars: int I: *Real part of the encoded constellation*
int Q: *Imaginary part of the encoded constellation*Output vars: return-value: *2-D coset number* $\in (0, 1, 2, 3)$

I/O vars: none

int Previous_State(int State, int Branch)

Author: Kiaei/Markus Muck

Description: Finding the previous state of the encoder.

Input vars: int State: *Latest state*
int Branch: *Latest branch decision*Output vars: return-value: *Previous decoder state*

I/O vars: none

int Coset_Number(int State, int Branch)

Author: Kiaei/Markus Muck

Description: Calculating the 4-D coset corresponding to the latest state.

Input vars: int State: *Latest state*
int Branch: *Latest branch decision*Output vars: return-value: *Coset number corresponding to the latest state*

I/O vars: none

void Modulo_ACS(double i, double j, double k, double l, double &min, int &branch)

Author: Kiaei/Markus Muck

Description: Performing the (A)ddition-(C)ompare-(S)elect operation. We take THE solution which guarantees the smallest metric.

Input vars: double i: *Metric of solution 0*
double j: *Metric of solution 1*
double k: *Metric of solution 2*
double l: *Metric of solution 3*Output vars: double &min: *The resulting branch metric*
int &branch: *The branch corresponding to the chosen solution* $\in (0, 1, 2, 3)$

I/O vars: none

int symbol_decode(int present_state, int previous_state)

Author: Kiaei/Markus Muck

Description: Calculating the 4-D coset number using the latest state and the previous state of the decoder.

Input vars: int present_state: *Latest decoder state*
int previous_state: *Previous decoder state*Output vars: return-value: *4-D coset number* $\in (0, 1, \dots, 7)$

I/O vars: none

double Mod16(double Sum)

Author: Kiaei/Markus Muck

Description: Performs a modulo-16 operation. No longer used.

Input vars: double Sum: *Integer value*Output vars: return-value: *Sum* - 16 if *Sum* \geq 16, else *Sum* is returned

I/O vars: none

int BIT_PACKER(int I, int Q, int bits)

Author: Kiaei/Markus Muck

Description: Decoding the bits corresponding to the encoded constellation *I, Q*.Input vars: double I: *Real part of a encoded constellation*
double Q: *Imaginary part of a encoded constellation* int bits: *Max. number of bits in I, Q*Output vars: return-value: Decoded data ($=[V, W]$, \rightarrow T1E1 6.6.2, Figure 10)

I/O vars: none

F.5 Supporting functions

F.5.1 cvector.C

Here, a complex vector class of a variable size and some operators are defined.



cvector::cvector(int xsize)

Author: Kiaei/Markus Muck

Description: Here, the memory for a new vector of size *xsize* and its name (standard-name is “?”) is allocated. The vector is initialized with zero values in the real and in the imaginary part.

Input vars: int xsize: *Vector size*

Output vars: none

I/O vars: none

cvector::cvector()

Author: Kiaei/Markus Muck

Description: Here, the memory for a new vector of size “1” and its name (standard-name is “?”) is allocated. The vector is initialized with zero values in the real and in the imaginary part.

Input vars: int xsize: *Vector size*

Output vars: none

I/O vars: none

cvector::cvector(const cvector& v1)

Author: Kiaei/Markus Muck

Description: Here, the memory for a new vector of the same size as “v1” and its name (same name as “v1”) is allocated. The values of the vector “v1” are used as initial values.

Input vars: const cvector &v1: *Vector to be copied*

Output vars: none

I/O vars: none

void cvector::init(double *initial_settings)

Author: Kiaei/Markus Muck

Description: This function initialized all values of the complex vector to zero if “double *initial_settings” = 0. Otherwise, the values in “double *initial_settings” are used as initial values (“initial_settings[index]” is used for both, the real and the imaginary part for “cvector[index]”).

Input vars: double *initial_settings: *Initial values*

Output vars: none

I/O vars: none

void cvector::grow(int additional_size)

Author: Kiaei/Markus Muck

Description: This function changes the vector size. The additional elements are initialized with zero. The old elements are kept.

Input vars: int additional_size: *Defines the additional vector size, may also be negative*

Output vars: none

I/O vars: none

int &cvector::operator[](int index)

Author: Kiaei/Markus Muck

Description: The [] operator allows to look for a value stored in the vector.

*Example: cvector_name[0] looks for the first entry in the vector (index 0).*Input vars: int index: *Defines the position in the vector*Output vars: return-value: *Complex value stored at position **index** in the vector.*

I/O vars: none

cvector &cvector::operator=(const cvector &v1)

Author: Kiaei/Markus Muck

Description: This operator allows to copy the elements of one vector into another vector of the same size.

Input vars: const cvector &v1: *Vector to be copied*Output vars: return-value: *Copied vector*

I/O vars: none

F.5.2 generate.C**void generate_bits_per_tone_CO(unsigned short int **BitsPerTone_CO)**

Author: Markus Muck

Description: Allocating the memory of the "Bits per Tone"-variable and defining the default values which are given by the define FILE_BITS_PER_TONE.

Input vars: none

Output vars: unsigned short int **BitsPerTone_CO: *Pointer to (not yet allocated) buffer containing the number of bits per tone*

I/O vars: none



**void generate_Superframe_Properties_CO(struct Sframe_Properties_CO
*Superframe_Properties_CO)**

Author: Markus Muck

Description: This functions defines some standard values for the simulation properties like *number of redundancy bytes per AS, interleaver deepness, etc. etc.*

Input vars:

Output vars: struct Sframe_Properties_CO *Superframe_Properties_CO: *Pointer to structure containing the properties*

I/O vars: none

void generate_LS_Buffer(unsigned char **InputDataLS, unsigned int *DataToTransmitLS)

Author: Markus Muck

Description: Generation of a random *LS Buffer* containing “SIZE_LSX_BUFFER” bytes. Memory for “unsigned char **InputDataLS” will be allocated.

Input vars: none

Output vars: unsigned char **InputDataLS: *Buffer for random LS Data*
unsigned int *DataToTransmitLS: *Number of bytes written*

I/O vars: none

void allocate_memory(unsigned char **InputDataAS, int DataToTransmitAS, unsigned char **Superframe_Memory_AS, unsigned char **Superframe_Memory_LS, double **TimeDomainData, unsigned char **Redundancy_AS, unsigned char **Redundancy_LS, unsigned char **Data_Interleaved, unsigned char ** Superframe_Memory_AS_reception, unsigned char ** Superframe_Memory_LS_reception, unsigned char OutputDataAS, unsigned char** OutputDataLS, unsigned char **Redundancy_AS_reception, unsigned char **Redundancy_LS_reception)**

Author: Markus Muck

Description: This function reserves the memory for some variables used during the simulation.

Input vars: int DataToTransmitAS: *Number of bits to be transmitted*

Output vars: unsigned char **InputDataAS: *Pointer to a buffer containing the data to be transmitted*
unsigned char **Superframe_Memory_AS: *Pointer to an ASX-Buffer*
unsigned char **Superframe_Memory_LS: *Pointer to an LSX-Buffer*
double **TimeDomainData: *Buffer for the time domain data created by the transmitter*

unsigned char **Redundancy_AS: *Buffer for the forward-error-correction redundancy data for the ASX*

unsigned char **Redundancy_LS: *Buffer for the forward-error-correction redundancy data for the LSX*

unsigned char **Data_Interleaved: *Buffer for the interleaved data*

unsigned char ** Superframe_Memory_AS_reception: *Buffer for the received AS-Data*

unsigned char ** Superframe_Memory_LS_reception: *Buffer for the received LS-Data*

unsigned char** OutputDataAS: *Buffer for the received and decoded AS-data*

unsigned char** OutputDataLS: *Buffer for the received and decoded LS-data*

unsigned char **Redundancy_AS_reception: *Buffer for the received forward-error-correction data concerning the AS-data*

unsigned char **Redundancy_LS_reception: *Buffer for the received forward-error-correction data concerning the LS-data*

I/O vars: none

F.5.3 ivector.C

Here, an integer vector class of a variable size and some operators are defined.

ivector::ivector(int ivector_size, char *ivector_name, int *initial_values)

Author: Kiaei/Markus Muck

Description: Here, the memory for a new vector of size *ivector_size* and its name is allocated. The values of the array *initial_values* are used as initial values.

Input vars: int ivector_size: *Vector size*
char *ivector_name: *Any name given to the object*
int *initial_values: *Array of ivector_size of the initial values*

Output vars: none

I/O vars: none

ivector::ivector(const ivector &v1)

Author: Kiaei/Markus Muck

Description: Here, the memory for a new vector of the same size as “v1” and its name (same name as “v1”) is allocated. The values of the vector “v1” are used as initial values.

Input vars: const ivector &v1: *Vector to be copied*

Output vars: none

I/O vars: none



void ivector::init()

Author: Kiaei/Markus Muck

Description: This function resets all values of the vector to zero.

Input vars: none

Output vars: none

I/O vars: none

void ivector::setname(char *ivector_name)

Author: Kiaei/Markus Muck

Description: This function changes the name of a vector.

Input vars: char *ivector_name: *New vector name*

Output vars: none

I/O vars: none

void ivector::update_ivector(int ivector_size, char *ivector_name, int *initial_values)

Author: Markus Muck

Description: This function *fre*es the allocated memory of the vector, changes the vector size and the vector name and allocates the necessary memory for the new vector. It is initialized by the values of the array *int *initial_values*.Input vars: int ivector_size: *Vector size*
char *ivector_name: *Any name given to the object*
int *initial_values: *Array of ivector_size of the initial values*

Output vars: none

I/O vars: none

void ivector::grow(int additional_size)

Author: Kiaei/Markus Muck

Description: This function changes the vector size. The additional elements are initialized with zero. The old elements are kept.

Input vars: int additional_size: *Defines the additional vector size, may also be negative*

Output vars: none

I/O vars: none

int &ivector::operator[](int index)

Author: Kiaei/Markus Muck

Description: The `[]` operator allows to look for a value stored in the vector.*Example: `ivector_name[0]` looks for the first entry in the vector (index 0).*Input vars: int index: *Defines the position in the vector*Output vars: return-value: *Integer value stored at position **index** in the vector.*

I/O vars: none

ivector &ivector::operator=(const ivector &v1)

Author: Kiaei/Markus Muck

Description: This operator allows to copy the elements of one vector into another vector of the same size.

Input vars: const ivector &v1: *Vector to be copied*Output vars: return-value: *Copied vector*

I/O vars: none

F.5.4 my_types.CHere, some basic operations for the *complex* structure defined in *my_types.h* are defined. They are also used by the *cvector* class.**The complex structure**For compatibility reasons, the complex variables used in the ADSL simulator do *not* use the *complex.h* library. The following structure is used for all complex variables:

```
typedef struct
{
double re, im;
}
complex;
```

complex make_complex(double real_v, double imag_v)

Author: Markus Muck

Description: This function converts two *double* variables into a complex variable.Input vars: double real_v: *Real part*
double imag_v: *Imaginary part*

Output vars: return-value: *Complex variable (real_v+ j-imag_v)*

I/O vars: none

double real(complex v)

Author: Markus Muck

Description: This function returns the real part of a complex variable.

Input vars: complex v: *A complex variable*

Output vars: return-value: *Real part of “v”*

I/O vars: none

double imag(complex v)

Author: Markus Muck

Description: This function returns the imaginary part of a complex variable.

Input vars: complex v: *A complex variable*

Output vars: return-value: *Imaginary part of “v”*

I/O vars: none

complex cminus(complex c1, complex c2)

Author: Markus Muck

Description: This function performs the “subtraction” operation for a complex variable.

Input vars: complex c1: *First complex variable*
complex c2: *Second complex variable*

Output vars: return-value: *Subtraction result (c1-c2)*

I/O vars: none

complex cplus(complex c1, complex c2)

Author: Markus Muck

Description: This function performs the “addition” operation for a complex variable.

Input vars: complex c1: *First complex variable*
complex c2: *Second complex variable*

Output vars: return-value: *Addition result (c1+c2)*

I/O vars: none

complex cmult(complex c1, complex c2)

Author: Markus Muck

Description: This function performs the “multiplication” operation for a complex variable.

Input vars: complex c1: *First complex variable*
complex c2: *Second complex variable*Output vars: return-value: *Multiplication result (c1·c2)*

I/O vars: none

complex cdiv(complex c1, complex c2)

Author: Markus Muck

Description: This function performs the “division” operation for a complex variable.

Input vars: complex c1: *First complex variable*
complex c2: *Second complex variable*Output vars: return-value: *Division result ($\frac{c1}{c2}$)*

I/O vars: none

complex c_exp(complex c1)

Author: Markus Muck

Description: This function performs the “exponential” operation for a complex variable.

Input vars: complex c1: *Complex variable*Output vars: return-value: *Result of e^{c1}*

I/O vars: none

complex c_conj(complex c1)

Author: Markus Muck

Description: This function calculate the “complex conjugate” of a complex variable.

Input vars: complex c1: *Complex variable*Output vars: return-value: *Result of $(c1)^*$*

I/O vars: none



F.5.5 routines.C**int bin_to_dec(int *bit_array, int size)**

Author: Kiaei/Markus Muck

Description: Converting a bit-array of size *size* “int *bit_array” containing one bit $\in (0, 1)$ per integer value into an integer value.Input vars: int *bit_array: *Bit array, one bit $\in (0, 1)$ per integer value*
int size: *Size of bit array*Output vars: return-value: *Integer value corresponding to “int *bit_array”*

I/O vars: none

void dec_to_bin(int *result, int size, int num)

Author: Kiaei/Markus Muck

Description: Converting an integer number “int num” into a series of “int size” bits. The result is stored in “int *result”, one bit $\in (0, 1)$ per integer value.Input vars: int size: *Number of bits*
int num: *Integer value to be converted*Output vars: int *result: *Bit array, one bit $\in (0, 1)$ per integer value*

I/O vars: none

int power (int basis, int n)

Author: Kiaei/Markus Muck

Description: This function calculates “ $(basis)^n$ ” without using the “pow”-command. “int basis” and “int n” must be integers.Input vars: int basis: *The basis of $(basis)^n$*
int n: *The exponent of $(basis)^n$* Output vars: return-value: *result of $(basis)^n$*

I/O vars: none

unsigned char *memmove(unsigned char* v1, unsigned char *v2, int n)

Author: Markus Muck

Description: This is a substitution for the *memmove* command of the string library. It must be activated via the USE_MEMMOVE_OF_STRING_H switch in switch.h. This substitution has been added for compatibility reasons.

Input vars: look at *memmove* in the C-manual

Output vars: look at *memmove* in the C-manual

I/O vars: none

void memset(unsigned char* v1, unsigned char value, int n)

Author: Markus Muck

Description: This is a substitution for the *memset* command of the string library. It must be activated via the `USE_MEMMOVE_OF_STRING_H` switch in `switch.h`. This substitution has been added for compatibility reasons.

Input vars: look at *memset* in the C-manual

Output vars: look at *memset* in the C-manual

I/O vars: none

F.5.6 tools.C

void simulator_usage (void)

Author: Markus Muck

Description: This function writes the command line parameters of the simulator to the standard output.

Input vars: none

Output vars: none

I/O vars: none

void show_version (void)

Author: Markus Muck

Description: This function write the version number and the release date to the standard output.

Input vars: none

Output vars: none

I/O vars: none



int find_number(char *Data, char Separator)

Author: Markus Muck

Description: This function is used in order to read in an integer parameter which is given in a text-parameter file. "char *Data" contains one line of text. This function skips all data up to a separator (for example ":"). The value after that separator is converted into an integer number.

Input vars: char *Data: *One line of text*
char Separator: *Separation character*

Output vars: return-value: *converted number*

I/O vars: none

void compare_data(unsigned char *P1, unsigned char *P2, int NbBits)

Author: Markus Muck

Description: This function compares "int NbBits" bits of two character arrays. In total, there are "int NbBits/8" bytes compared and an error message is written to the standard output if an error occurs (position of the first erroneous bit).

Input vars: unsigned char *P1: *Data array 1*
unsigned char *P2: *Data array 2*
int NbBits: *Number of bits to compare*

Output vars: none

I/O vars: none

void find_bits_in_char(int *bits, unsigned char *source, unsigned int already_sent, unsigned int to_be_sent)

Author: Markus Muck

Description: This functions copies "to_be_sent" bits of the source string "*source" into the buffer "*bits" beginning with the "already_sent"th bit of "*source" and the "0"th bit of "*bits".

Input vars: unsigned char *source: *Buffer where to take the bits from*
unsigned int already_sent: *Number of bits to be skipped in the "*source"*
unsigned int to_be_sent: *Number of bits to be sent*

Output vars: int *bits: *Buffer where to copy the bits*

I/O vars: none


```
void set_bits( unsigned char *out, int start_out, int bits[], int nr_bits)
```

Author: Kiaei/Markus Muck

Description: This function copies a certain number of bits from a int-array (one bit $\in (0, 1)$ per integer value) to a character array (eight bits per character).

Input vars: int start_out: *Number of first bit to be read*
int bits[]: *Array where to read the data from*
int nr_bits: *Number of bits to be copied*

Output vars: unsigned char *out: *Copied bits*

I/O vars: none

```
void memmove_bits( unsigned char *out, int start_out, unsigned char *in, int nr_bits)
```

Author: Markus Muck

Description: This function copies a certain number of bits from one byte-array (eight bits per character) to another.

Input vars: int start_out: *Number of first bit to be read*
unsigned char *in: *Array where to read the data from*
int nr_bits: *Number of bits to be copied*

Output vars: unsigned char *out: *Copied bits*

I/O vars: none

F.6 INCLUDE files containing some useful #defines

F.6.1 constants.h

```
#define ADDITIONAL_CIR_SAMPLES
```

Author: Markus Muck

Description: For the convolution *Channel Impulse Response (CIR)*Transmitted Data X* we take 512 samples AND the last “ADDITIONAL_CIR_SAMPLES” samples in order to guarantee a linear convolution for the last 512 samples.

```
#define CHANNEL_ESTIMATION_WITH_NOISE
```

Author: Markus Muck

Description: Should the estimation of the Channel Impulse Response (CIR) be done with or without noise? If active, it's done with noise (using the SNR defined by “SNR_Transmission”).



#define CRC_CHECK_FAST_DATA

Author: Markus Muck

Description: Should the CRC check be performed for the fast or interleaved data ?

#define ERROR_WEIGHTING_ON

Author: Markus Muck

Description: If active, the error $(TIR * X) - (TEQ * R)$ is weighted in the frequency domain (WSAF algorithm), with X being the transmitted data, R being the received noisy samples, TIR being the Target Impulse Response filter coefficients and TEQ being the Time Domain Equalizing filter coefficients. If not active, the error is not weighted and therefore a primitive LMS algorithm is applied for the filter taps update.

#define FEC_CONTROL_ADRESS_ENCODING

Author: Markus Muck

Description: Standard value for the *Reed-Solomon* class.

#define FEC_CONTROL_IDLE,
#define FEC_CONTROL_READ_DATA_AND_DIVIDE,
#define FEC_CONTROL_SHIFT_OUT_FROM_MEMORY,
#define FEC_CONTROL_SHIFT_IN_FROM_MEMORY,
#define FEC_CONTROL_SHIFT_OUT_CHECK_BYTES

Author: Markus Muck

Description: Possible values of "control_byte" for "FEC::Encode(Byte, control_byte, addr, nb_red_bytes)".

#define FILE_BITS_PER_TONE

Author: Markus Muck

Description: Name of the file containing the number of bits per tone.

#define FILE_CIR

Author: Markus Muck

Description: Here, the file name of the file containing the Channel Impulse Response (CIR) time domain samples is defined.

#define FILE_COSET_TABLE

Author: Kiaei

Description: A file containing some information on how to do the constellation encoding.

#define FILTER_TAP_UPDATE_WITH_NOISE

Author: Markus Muck

Description: Should the update of the Time Domain Equalizing filter be done with noise ? If active, it's done with noise (using the SNR defined by "SNR_Transmission").

#define FRAMES_PER_SUPERFRAME

Author: Markus Muck

Description: Number of frames per superframe.

#define LENGTH_GUARD_INTERVAL

Author: Markus Muck

Description: Here, this size of the guard interval is fixed (usually 32 samples).

#define MAX_BIT_NUMBER_PER_TONE

Author: Markus Muck

Description: Maximum number of bits per tone.

#define MAX_CARRIER_AMPLITUDE

Author: Markus Muck

Description: Right now, there is the same power for every carrier used, i.e. the maximum amplitude of each single carrier is limited to the value "MAX_CARRIER_AMPLITUDE".

#define MAX_INTERLEAVER_BUFFER_SIZE

Author: Markus Muck

Description: Maximum size of the (de)interleaver buffer.

#define MAX_NUMBER_COSETS

Author: Markus Muck

Description: Maximum number of 4-D cosets.



#define MAX_NUMBER_REDUNDANCY_BYTES_AS

Author: Markus Muck

Description: Maximum number of redundancy bytes in an ASX frame.

#define MAX_NUMBER_REDUNDANCY_BYTES_LS

Author: Markus Muck

Description: Maximum number of redundancy bytes in a LSX frame.

#define MAX_NUMBER_SYMBOLS_PER_CODEWORD

Author: Markus Muck

Description: Maximum number of symbols per codeword ($=S$, $\rightarrow T1E1$, 6.2.1.2).

#define MAX_SIZE_LSX_PER_SUPERFRAME

Author: Markus Muck

Description: Maximum size of all LSX Buffers in a superframe in bits.

#define MAX_SIZE_SUPERFRAME

Author: Markus Muck

Description: Maximum size of a superframe in bytes.

#define NBR_SYMBOLS_FOR_CHANNEL_ESTIMATION

Author: Markus Muck

Description: It determines, how many symbols should be taken in order to get an accurate estimation of the Channel Impulse Response (CIR).

#define NBR_TEQ_TAPS

Author: Markus Muck

Description: Here, the number of filter taps of the Time Domain Equalizing (TEQ) filter is fixed. Using the unweighted LMS algorithm, approx. 16 filter taps are sufficient, using the WSAF algorithm, more filter taps are needed (approx. 64).

#define NBR_TIR_TAPS

Author: Markus Muck

Description: Here, the number of filter taps of the Target Impulse Response (TIR) filter is fixed. Usually, the TIR size corresponds to the number of taps of the guard interval.

#define NOISE_DURING_TRANSMISSION_ON

Author: Markus Muck

Description: Should the transmission (after the Time Domain Equalizing filter update, etc.) be done with or without channel noise ? If active, it's done with noise (using the SNR defined by "SNR_Transmission").

#define NOTICE

Author: Markus Muck

Description: A text string containing some copyright information.

#define NUMBER_TIME_DOMAIN_DATA_CO

Author: Markus Muck

Description: Number of time domain data taps created by the central office transmitter.

#define NUMBER_TONES_CO

Author: Markus Muck

Description: Number of tones in a DMT symbol transmitted by the central office.

#define NYQUIST_REVERB1

Author: Markus Muck

Description: Carrier amplitude for the C-REVERB1 symbol at the Nyquist Frequency Carrier (#256).

#define PARAMETER_DIRECTORY

Author: Markus Muck

Description: Name of the parameter directory.



#define RELEASE_DATE

Author: Markus Muck

Description: A text string representing the release date.

#define REVERB1_AMPLITUDE_PLUS_MINUS_1

Author: Markus Muck

Description: Should the amplitude “ ± 1 ” be used for the REVERB1-QAM signal (\rightarrow ADSL standard [3]) or “ $\pm 1 \cdot \text{carrier_amplitude}$ ” (i.e. should the transmission power be fixed to -38dBm/Hz as proposed by the ADSL standard) ?? If active, the amplitude “ ± 1 ” is used for the carriers.

#define SCRAMBLER_INIT_VALUE

Author: Markus Muck

Description: Initial value of the scrambler.

#define SIZE_LSX_BUFFER

Author: Markus Muck

Description: Maximum size of the LSX Buffers in bits.

#define SNR_Transmission

Author: Markus Muck

Description: Here, the signal-to-noise ratio (SNR) for the transmission is fixed in dB.

#define STANDARD_IVECTOR_SIZE

Author: Markus Muck

Description: Standard size of an *ivector*.

#define SUPERFRAME_DURATION_SEC

Author: Markus Muck

Description: Duration of a superframe in seconds.

#define TEQ_FIRST_COEFF

Author: Markus Muck

Description: The Time Domain Equalizing (TEQ) filter is initialized with zeros before starting the filter update. The only exception is the first filter tap, it is initialized with this value.

#define TEQ_Update_Factor

Author: Markus Muck

Description: The calculated Time Domain Equalizing (TEQ) filter updates are multiplied with this constant factor. When the number of filter taps is changed or when there are problems with the convergence properties of the algorithm, the step size must be adjusted here.

#define TEQ_UPDATE_ITERATIONS

Author: Markus Muck

Description: Here, the number of iterations for the Time Domain Equalizing (TEQ) filter is fixed.

#define TWO_SYMBOLS_SIZE

Author: Markus Muck

Description: The size of two OFDM symbols, usually $2 \times 512 = 1024$ samples.

#define VERSION

Author: Markus Muck

Description: A text string representing the latest version number of the simulator.

F.6.2 debug.h**#define Error(msg)**

Author: Marc de Courville

Description: This #define writes an error message to the standard output stream including the message "msg", the line of code where the error occurred, the C-filename, etc..

Input vars: msg: *Error message to be sent to the standard output stream*

Output vars: none

I/O vars: none



F.6.3 **define.h**

Here, quite a lot of abbreviations are defined. We are going to mention only the ones used in the simulator.

#define OPENFILE(file_pointer,file_name,open_option)

Author: Marc de Courville

Description: This #define is used to open a file.

Input vars: file_name: *Name of file*
open_option: *File options like "rb", "wb", ...*

Output vars: file_pointer: *File handle*

I/O vars: none

#define CALLOCVAR(var,cast,number,size)

Author: Marc de Courville

Description: This #define is used to allocate memory.

Input vars: cast: *Pointer type like "char*", ...*
number: *Number of elements*
size: *Size per element*

Output vars: var: *Pointer variable*

I/O vars: none

struct Sframe_Properties_CO

Author: Markus Muck

Description: This is the structure containing the properties of a superframe like *number of redundancy bytes per AS, interleaver deepness, etc. etc.*

F.6.4 **switch.h**

Here, all compiler switches are defined. It is possible to define, whether the *trellis decoding, the energy (de)scrambling, etc. etc.* should take place.

#define ENERGY_SCRAMBLING_OFF

Author: Markus Muck

Description: This switch deactivates (if active) the *energy scrambling*.

#define FOR_DEBUG_ONLY_1_FRAME_RECEPTION

Author: Markus Muck

Description: If this switch is active, only one DMT symbol is transmitted and received per super-frame.

#define INTERLEAVING_OFF

Author: Markus Muck

Description: This switch deactivates (if active) the *interleaving* and *deinterleaving*.

#define REED_SOLOMON_ACTIVE

Author: Markus Muck

Description: This switch activates (if active) the *Reed-Solomon* forward error correction decoding.

#define TRELIS_TCM_ACTIVE

Author: Markus Muck

Description: If active, the arriving data is decoded using a maximum-likelihood decoder (viterbi algorithm). Otherwise, the data will be taken directly without using the error correction qualities of the convolutional codes.

#define USE_MEMMOVE_OF_STRING_H

Author: Markus Muck

Description: If this define is active, the *string.h* library is included and its commands like *memmove*, *memset*, etc. used. If not, the commands *memmove* and *memset* as defined in *routines.C* are used.

#define VERBOSE

Author: Markus Muck

Description: This switch activates (switch on) or deactivates (switch off) the text output to the standard stream during the simulation process. If active, some information concerning the simulation, like convolutional encoder state, etc. is displayed.



#define VERBOSE_PER_DMT

Author: Markus Muck

Description: Like *#define VERBOSE*, this switch enables (when active) the text output to the standard stream during the simulation process. Here, quite a lot of information is displayed for each transmitted and received DMT symbol.

Appendix G

The Parameter Files of the ADSL simulator

The simulation parameters can be easily adjusted by editing the parameter files using a text editor. The file names of the parameter files are defined in *constants.h*.

G.0.5 FILE_BITS_PER_TONE

The parameter file specified by FILE_BITS_PER_TONE contains the number of bits per tone $\in (2, 3, \dots, 15)$. Example for this parameter file:

```
Bits for tone 1: 0
Bits for tone 2: 0
Bits for tone 3: 0
Bits for tone 4: 0
Bits for tone 5: 0
Bits for tone 6: 0
Bits for tone 7: 0
Bits for tone 8: 8
Bits for tone 9: 8
Bits for tone 10: 8
Bits for tone 11: 8
Bits for tone 12: 8
:
Bits for tone 249: 11
Bits for tone 250: 11
Bits for tone 251: 8
Bits for tone 252: 8
Bits for tone 253: 8
Bits for tone 254: 8
Bits for tone 255: 8
```

Appendix H

The comments of the ADSL simulator during a simulation

After starting a simulation, the ADSL simulator always comments the operations it is performing. This facilitates the search for potential bugs in the simulator and gives a good impression of which calculation module is the most time consuming.

In the following, the comments that occurred during a typical simulation are presented. Hereby, a file named *out.txt* of size 8236 Bytes is transmitted.

```
=====
STARTING THE ADSL SIMULATOR, (c) 1998-9 by crm, Paris and
Markus Muck (University of Stuttgart, ENST Paris)
=====
File containing data to be transmitted:  out.txt
simulator:  Name of file to be transmitted:  out.txt, Size:  8236 Bytes
generate:  Bits per tone loaded from file './parameters/BITS_PER_TONE', 255 tones altogether
generate:  The max. amplitudes for each carrier have been calculated.
generate:  FEC Output Frame Size Nmi = 104
channel:  Time Domaine Equalizer (TEQ) Filter coefficients are calculated using the WSAF algorithm
within 600 iterations.
channel:  The ADSL channel impulse response has been loaded from file './parameters/csa6cr'.
channel:  The noise amplitude per sample is:  7.6298e-07 for a SNR of 60.00 dB for symbol
energy 2.9805e-04.
channel:  Resulting TEQ performance:  Energy after GI/Total energy of (CIR convolved with TEQ) =
3.15662e-04 (without TEQ: 1.13058e-01)
channel:  Time Domaine Equalizer (TEQ) Filter taps calculation done.
transmitter_central_office:  Initialization done, bytes per superframe = 6528.
transmitter_central_office:  Not yet all data encoded (DataTransmittedAS=0). Data remaining.
transmitter_central_office:  Multiplexing done.
transmitter_central_office:  Beginning mit FEC for fast MUX frame, Bytes to encode = 12,
Nr redund. Bytes = 4
transmitter_central_office:  Reed-Solomon-Encoding done for fast buffer.
transmitter_central_office:  Bytes per FEC_Frame:  192, FEC frames:  34, Bytes per MUX frame = 96,
Redundancy bits per MUX frame = 8.
transmitter_central_office:  Reed-Solomon-Encoding done.
transmitter_central_office:  Scrambling done, scrambler state 5a5122hex.
transmitter_central_office:  Time-Interleaving done, codeword-size = 208 bytes
incl. 16 bytes of redundancy.
simulator:  treating frame #0 out of #68 frames.
simulator:  treating frame #1 out of #68 frames.
simulator:  treating frame #2 out of #68 frames.
simulator:  treating frame #3 out of #68 frames.
simulator:  treating frame #4 out of #68 frames.
simulator:  treating frame #5 out of #68 frames.
simulator:  treating frame #6 out of #68 frames.
simulator:  treating frame #7 out of #68 frames.
simulator:  treating frame #8 out of #68 frames.
simulator:  treating frame #9 out of #68 frames.
simulator:  treating frame #10 out of #68 frames.
simulator:  treating frame #11 out of #68 frames.
simulator:  treating frame #12 out of #68 frames.
simulator:  treating frame #13 out of #68 frames.
simulator:  treating frame #14 out of #68 frames.
```

```

simulator: treating frame #15 out of #68 frames.
simulator: treating frame #16 out of #68 frames.
simulator: treating frame #17 out of #68 frames.
simulator: treating frame #18 out of #68 frames.
simulator: treating frame #19 out of #68 frames.
simulator: treating frame #20 out of #68 frames.
simulator: treating frame #21 out of #68 frames.
simulator: treating frame #22 out of #68 frames.
simulator: treating frame #23 out of #68 frames.
simulator: treating frame #24 out of #68 frames.
simulator: treating frame #25 out of #68 frames.
simulator: treating frame #26 out of #68 frames.
simulator: treating frame #27 out of #68 frames.
simulator: treating frame #28 out of #68 frames.
simulator: treating frame #29 out of #68 frames.
simulator: treating frame #30 out of #68 frames.
simulator: treating frame #31 out of #68 frames.
simulator: treating frame #32 out of #68 frames.
simulator: treating frame #33 out of #68 frames.
simulator: treating frame #34 out of #68 frames.
simulator: treating frame #35 out of #68 frames.
simulator: treating frame #36 out of #68 frames.
simulator: treating frame #37 out of #68 frames.
simulator: treating frame #38 out of #68 frames.
simulator: treating frame #39 out of #68 frames.
simulator: treating frame #40 out of #68 frames.
simulator: treating frame #41 out of #68 frames.
simulator: treating frame #42 out of #68 frames.
simulator: treating frame #43 out of #68 frames.
simulator: treating frame #44 out of #68 frames.
simulator: treating frame #45 out of #68 frames.
simulator: treating frame #46 out of #68 frames.
simulator: treating frame #47 out of #68 frames.
simulator: treating frame #48 out of #68 frames.
simulator: treating frame #49 out of #68 frames.
simulator: treating frame #50 out of #68 frames.
simulator: treating frame #51 out of #68 frames.
simulator: treating frame #52 out of #68 frames.
simulator: treating frame #53 out of #68 frames.
simulator: treating frame #54 out of #68 frames.
simulator: treating frame #55 out of #68 frames.
simulator: treating frame #56 out of #68 frames.
simulator: treating frame #57 out of #68 frames.
simulator: treating frame #58 out of #68 frames.
simulator: treating frame #59 out of #68 frames.
simulator: treating frame #60 out of #68 frames.
simulator: treating frame #61 out of #68 frames.
simulator: treating frame #62 out of #68 frames.
simulator: treating frame #63 out of #68 frames.
simulator: treating frame #64 out of #68 frames.
simulator: treating frame #65 out of #68 frames.
simulator: treating frame #66 out of #68 frames.
simulator: treating frame #67 out of #68 frames.
receiver_home_part2: Initialization done.
receiver_home_part2: First DEINTERLEAVING call, only 33 codeword(s) = 6864 bytes
could be deinterleaved.
receiver_home_part2: Time Deinterleaving done.
receiver_home_part2: FEC: Fast buffer decoded.
receiver_home_part2: FEC: Bytes per FEC_Frame: 192, FEC frames: 34.
receiver_home_part2: FEC: Reed-Solomon-Decoding done.
compare_data (tools): 65888 bits compared, 5961 errors found -> 9.04717% error rate.
compare_data (tools): The first error occurred at bit nr 50690.
transmitter_central_office: All data encoded. No data remaining.
transmitter_central_office: Multiplexing done.
transmitter_central_office: Beginning mit FEC for fast MUX frame, Bytes to encode = 12,
Nr redund. Bytes = 4
transmitter_central_office: Reed-Solomon-Encoding done for fast buffer.
transmitter_central_office: Bytes per FEC_Frame: 192, FEC frames: 34, Bytes per MUX frame = 96,
Redundancy bits per MUX frame = 8.
transmitter_central_office: Reed-Solomon-Encoding done.
transmitter_central_office: Scrambling done, scrambler state f4befhex.
transmitter_central_office: Time-Interleaving done, codeword-size = 208 bytes
incl. 16 bytes of redundancy.
simulator: treating frame #0 out of #68 frames.
simulator: treating frame #1 out of #68 frames.
simulator: treating frame #2 out of #68 frames.
simulator: treating frame #3 out of #68 frames.
simulator: treating frame #4 out of #68 frames.
simulator: treating frame #5 out of #68 frames.
simulator: treating frame #6 out of #68 frames.
simulator: treating frame #7 out of #68 frames.
simulator: treating frame #8 out of #68 frames.
simulator: treating frame #9 out of #68 frames.
simulator: treating frame #10 out of #68 frames.
simulator: treating frame #11 out of #68 frames.
simulator: treating frame #12 out of #68 frames.
simulator: treating frame #13 out of #68 frames.
simulator: treating frame #14 out of #68 frames.
simulator: treating frame #15 out of #68 frames.
simulator: treating frame #16 out of #68 frames.
simulator: treating frame #17 out of #68 frames.
simulator: treating frame #18 out of #68 frames.

```

APPENDIX H. THE COMMENTS OF THE ADSL SIMULATOR DURING A SIMULATION

```
simulator: treating frame #19 out of #68 frames.
simulator: treating frame #20 out of #68 frames.
simulator: treating frame #21 out of #68 frames.
simulator: treating frame #22 out of #68 frames.
simulator: treating frame #23 out of #68 frames.
simulator: treating frame #24 out of #68 frames.
simulator: treating frame #25 out of #68 frames.
simulator: treating frame #26 out of #68 frames.
simulator: treating frame #27 out of #68 frames.
simulator: treating frame #28 out of #68 frames.
simulator: treating frame #29 out of #68 frames.
simulator: treating frame #30 out of #68 frames.
simulator: treating frame #31 out of #68 frames.
simulator: treating frame #32 out of #68 frames.
simulator: treating frame #33 out of #68 frames.
simulator: treating frame #34 out of #68 frames.
simulator: treating frame #35 out of #68 frames.
simulator: treating frame #36 out of #68 frames.
simulator: treating frame #37 out of #68 frames.
simulator: treating frame #38 out of #68 frames.
simulator: treating frame #39 out of #68 frames.
simulator: treating frame #40 out of #68 frames.
simulator: treating frame #41 out of #68 frames.
simulator: treating frame #42 out of #68 frames.
simulator: treating frame #43 out of #68 frames.
simulator: treating frame #44 out of #68 frames.
simulator: treating frame #45 out of #68 frames.
simulator: treating frame #46 out of #68 frames.
simulator: treating frame #47 out of #68 frames.
simulator: treating frame #48 out of #68 frames.
simulator: treating frame #49 out of #68 frames.
simulator: treating frame #50 out of #68 frames.
simulator: treating frame #51 out of #68 frames.
simulator: treating frame #52 out of #68 frames.
simulator: treating frame #53 out of #68 frames.
simulator: treating frame #54 out of #68 frames.
simulator: treating frame #55 out of #68 frames.
simulator: treating frame #56 out of #68 frames.
simulator: treating frame #57 out of #68 frames.
simulator: treating frame #58 out of #68 frames.
simulator: treating frame #59 out of #68 frames.
simulator: treating frame #60 out of #68 frames.
simulator: treating frame #61 out of #68 frames.
simulator: treating frame #62 out of #68 frames.
simulator: treating frame #63 out of #68 frames.
simulator: treating frame #64 out of #68 frames.
simulator: treating frame #65 out of #68 frames.
simulator: treating frame #66 out of #68 frames.
simulator: treating frame #67 out of #68 frames.
receiver_home_part2: First DEINTERLEAVING call, only 34 codeword(s) = 7072
bytes could be deinterleaved.
receiver_home_part2: Time Deinterleaving done.
receiver_home_part2: FEC: Fast buffer decoded.
receiver_home_part2: FEC: Bytes per FEC_Frame: 192, FEC frames: 34.
receiver_home_part2: FEC: Reed-Solomon-Decoding done.
compare_data (tools): 65888 bits compared, 0 errors found -> 0.00000% error rate.
```

Bibliography

- [1] ADSL Forum *General Introduction to Copper Access Technologies*, ADSL Forum, WEB-Address <http://www.adsl.com>
- [2] N. Al-Dhahir, J.M. Cioffi *Optimum Finite-Length Equalization for Multicarrier Transceivers*, IEEE Transactions on Communications, vol. 44, no. 1, pp. 56-64, 1996
- [3] American National Standards Institute, Inc. *Asymmetric Digital Subscriber Line (ADSL) Metallic Interface*, T1E1.4/95-007R2
- [4] Walter Y. Chen *DSL, Simulation Techniques and Standards Development for Digital Subscriber Line Systems*, Macmillan Technology Series, 1998
- [5] Jacky S. Chow, John M. Cioffi, John A.C. Bingham *Equalizer Training Algorithms for Multicarrier Modulation Systems*, Proc. ICC, Geneva, pp. 761-765, 1993
- [6] Jacky S. Chow, Jerry T. Tu, John M. Cioffi *A Discrete Multitone Transceiver System for HDSL Applications*, IEEE journal on selected areas in communications, vol. 9, no. 6, 1991
- [7] Peter S. Chow *Bandwidth optimized digital transmission techniques for spectrally shaped channels with impulse noise*, PhD thesis, Stanford University USA, 1993
- [8] J.M. Cioffi, P. S. Chow *Line Code Complexity*, Amati Communications Corporation, California, USA, 1995
- [9] Marc de Courville, Pierre Duhamel *Orthogonal Frequency Division Multiplexing for Terrestrial Digital Broadcasting*, Ecole Nationale Supérieure des Télécommunications de Paris, 1998
- [10] Marc de Courville, Pierre Duhamel *Adaptive Filtering in Subbands Using a Weighted Criterion*, IEEE transactions on signal processing, vol. 46, no. 9, 1998
- [11] Marc de Courville *Utilisation de bases orthogonales pour l'algorithmique adaptative et l'égalisation des systèmes multiporteuses*, PhD thesis, Ecole Nationale Supérieure des Télécommunications, October 1996
- [12] Pierre Duhamel, Martin Vetterli *Fast Fourier Transforms: A Tutorial Review and a State of the Art*, Signal Processing, no. 19, pp. 259-299, 1990.
- [13] Robert F.H. Fischer, Johannes B. Huber *A new loading algorithm for discrete multitone transmission*, IEEE, 1996



- [14] Simon Haykin *Adaptive Filter Theory*, Prentice-Hall International Editions, Englewood Cliffs, New Jersey, USA, 1983
- [15] Minnie Ho, John M. Cioffi *High-Speed Full-Duplex Echo Cancellation for Discrete Multitone Modulation*, IEEE, 1993
- [16] Emmanuel C. Ifeachor, Barrie W. Jervis *Digital Signal Processing: A Practical Approach*, Addison Wesley publishing company, electronic systems engineering series, 1993
- [17] David C. Jones *Frequency Domain Echo Cancellation for Discrete Multitone Asymmetric Digital Subscriber Line Transceivers*, IEEE transactions on communications, vol. 43, no. 2/3/4, February/March/April 1995
- [18] Irving Kalet, *The Multitone Channel*, IEEE transactions on communications, vol. 37, no. 2, February 1989
- [19] Steven M. Kay, *Fundamentals of statistical signal processing estimation theory*, Prentice-Hall International Editions, Englewood Cliffs, New Jersey, USA
- [20] Philip J. Kyees, Ronald C. McConnell, Kamran Sistanizadeh *ADSL: A New Twisted-Pair Access to the Information Highway*, IEEE communications magazine, 1995
- [21] Inkyu Lee, Jacky S. Chow, John M. Cioffi *Performance Evaluation of a Fast Computation Algorithm for the DMT in High-Speed Subscriber Loop*, IEEE journal on selected areas in communications, vol. 13, no. 9, 1995
- [22] William C.Y. Lee *Mobile Communications Engineering*, McGRAW-HILL Telecommunications, 2nd edition, 1998
- [23] Shu Lin, Daniel J. Costello, Jr. *Error Control Coding: Fundamentals and Applications*, Prentice-Hall Series in Computer Applications in Electrical Engineering, Englewood Cliffs, New Jersey, USA, 1983
- [24] Byoung-Ki MIN *Architecture VLSI pour le decodeur de viterbi*, PhD thesis, Ecole Nationale Supérieure des Télécommunications, June 1991
- [25] Eric Moulines, Joseph Boutros *Egalisation Numérique*, Ecole Nationale Supérieure des Télécommunications de Paris, 1998
- [26] Gregory J. Pottie, M. Vedat Eyuboglu *Combined Coding and Precoding for PAM and QAM HDSL Systems*, IEEE journal on selected areas in communications, vol. 9, no. 6, 1991
- [27] John G. Proakis *Digital Communications*, McGRAW-HILL international editions, 3rd edition, 1995
- [28] Kimmo K. Saarela *ADSL*, Tampere University of Technology, Finland, 1995
- [29] Kamran Sistanizadeh, Peter S. Chow, John M. Cioffi *Multi-Tone Transmission for Asymmetric Digital Subscriber Lines (ADSL)*, IEEE, 1993
- [30] P. P. Vaidyanathan *Multirate Systems and Filter Banks*, Prentice Hall, Englewood Cliffs, New Jersey, USA, 1993

- [31] Robert Vallet *Etude des canaux sélectifs, Notions de diversité*, Ecole Nationale Supérieure des Télécommunications de Paris, 1995
- [32] L. Vandendorpe *MMSE equalizers for multitone systems without guard time*, 1996
- [33] L. Vandendorpe *Fractionally Spaced Linear and Decision-Feedback Detectors for Transmultiplexers*, IEEE transactions on signal processing, vol. 46, no. 4, April 1998
- [34] B. Widrow, J. M. McCool, M. G. Larimore, C. R. Johnson, Jr. *Stationary and non-stationary learning characteristics of the LMS adaptive filter*, Proc. IEEE, vol. 64, pp. 1151-1162, 1976
- [35] J. Yang, S. Roy *Data-Driven Echo Cancellation for a Multitone Modulation System*, IEEE transactions on communications, vol. 42, no. 5, May 1994
- [36] Gavin Young *Asymmetric Digital Subscriber Line (ADSL) Technology: Introduction and Overview*, IIR conference, September 1994

List of Figures

2.1	<i>An OFDM/DMT symbol in the frequency domain.</i>	24
2.2	<i>Discrete channel model.</i>	25
2.3	<i>Influence of the previous DMT symbol.</i>	26
2.4	<i>General OFDM/DMT transmission system.</i>	28
3.1	<i>The frequency spectrum of ADSL.</i>	30
3.2	<i>The ADSL system reference model.</i>	31
3.3	<i>A complete ADSL transceiver, remote terminal end.</i>	33
4.1	<i>The main function of the ADSL simulator.</i>	39
4.2	<i>The function preparing one superframe for transmission.</i>	40
4.3	<i>The function preparing one DMT symbol for transmission.</i>	41
4.4	<i>The function receiving one DMT symbol.</i>	41
4.5	<i>The function treating one superframe at the reception site.</i>	42
5.1	<i>A guard interval larger than the memory of the channel.</i>	43
5.2	<i>Equalization using a Target Impulse Response (TIR) filter.</i>	46
6.1	<i>The TEQ and TIR filters.</i>	49
6.2	<i>Shortening the Channel Impulse Response (CIR) by a Time Domain Equalizer (TEQ).</i>	51
6.3	<i>The optimal structure for the tap update.</i>	58
6.4	<i>A radix-2 butterfly and an efficient solution for a FFT of 512 points.</i>	60
7.1	<i>The channel impulse response (CIR) corresponding to CSA-Loop #6 in the time and frequency domain.</i>	69
7.2	<i>The initialization of the TEQ filter in the frequency domain.</i>	70
7.3	<i>The TEQ filter after one iteration for 128 and 16 filter taps using the WSAF algorithm.</i>	71

7.4	<i>The TEQ filter after one iteration for 128 and 16 filter taps without a weighted criterion.</i>	71
7.5	<i>The convergence properties without noise, non-weighted (16 filter taps) and WSAF (64 filter taps).</i>	72
7.6	<i>The convergence properties without noise, 64 filter taps for the non-weighted case and the WSAF (zoomed and over 1000 iterations).</i>	72
7.7	<i>The convergence properties with SNR=30dB.</i>	73
7.8	<i>The convergence properties with SNR=40dB.</i>	73
7.9	<i>The convergence properties with SNR=50dB.</i>	73
7.10	<i>The convergence properties with SNR=30dB (zoomed).</i>	74
7.11	<i>The convergence properties with SNR=40dB (zoomed).</i>	74
7.12	<i>The convergence properties with SNR=50dB (zoomed).</i>	74
7.13	<i>The true error during the optimization with SNR=30dB.</i>	75
7.14	<i>The true error during the optimization with SNR=40dB.</i>	75
7.15	<i>The true error during the optimization with SNR=50dB.</i>	76
7.16	<i>The resulting impulse response $w_n * c_n$ and the one of the TEQ filter (no noise) using a weighted criterion.</i>	76
D.1	<i>Top level diagram of the various modules of ADSL-SIMULATOR.</i>	89

List of Tables

3.1	<i>DMT downstream parameters.</i>	32
3.2	<i>DMT upstream parameters.</i>	32
6.1	<i>The complexity of basic operations.</i>	59
6.2	<i>The complexity of FFT operations.</i>	59
6.3	<i>The complexity of FFT operations.</i>	60
6.4	<i>Operations to be performed for the convolution/correlation.</i>	61
6.5	<i>Operations to be performed for the convolution/correlation (with $N = 512$ carriers and hermitian symmetry in the frequency domain).</i>	61
6.6	<i>The operations for the block LMS (BLMS) algorithm using fast convolutions/correlations.</i>	62
6.7	<i>The operations for the block LMS (BLMS) algorithm using linear convolutions/correlations in the time domain.</i>	63
6.8	<i>The operations for the WSAF algorithm using fast convolutions/correlations.</i>	64
6.9	<i>The complexity of the different filter update algorithms.</i>	65
6.10	<i>The complexity of the different filter update algorithms normalized by the number of multiplications/additions of the LMS algorithm (performing convolutions/correlations in time domain, $L = 16$ filter taps).</i>	66
6.11	<i>The complexity of the different convolution algorithms.</i>	68
6.12	<i>The complexity of the different convolution algorithms (ADSL).</i>	68
7.1	<i>Initialization of the TEQ filter taps.</i>	70
7.2	<i>Initialization of the TEQ filter taps.</i>	72
7.3	<i>The portion of the remaining energy of $w_n * c_n$ after the guard interval.</i>	77
7.4	<i>Minimum number of iterations for a remaining energy of $w_n * c_n$ after the guard interval.</i>	78