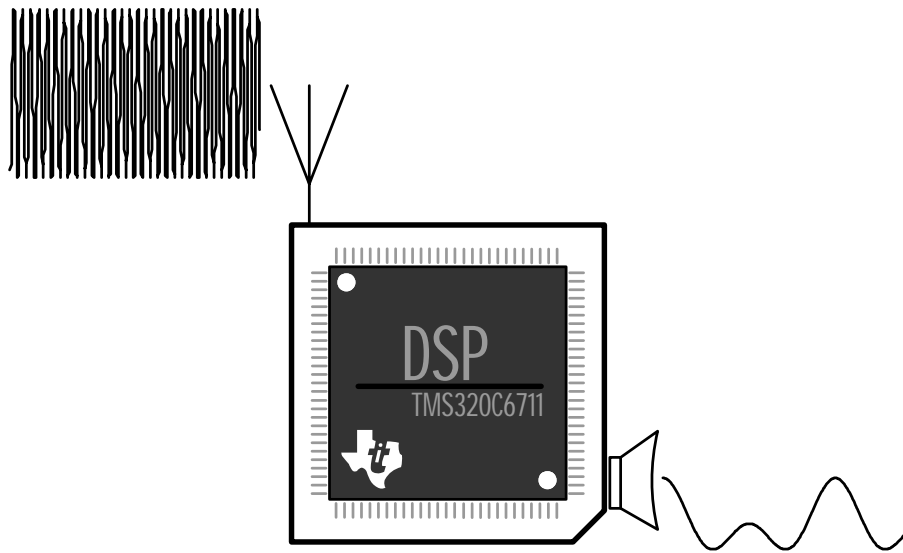


# Implementation of FM Demodulator Algorithms on a High Performance Digital Signal Processor



Franz Schnyder

Christoph Haller

11.1.2002

**Supervisors:** Dr. Chang Chip Hong, Andreas Ehrensperger

---

# Abstract

Analog modulations are more and more replaced by digital ones. To keep the compatibility the new device should be able to perform the analog modulation. A interesting solution is to solve this with digital signal processing. In an earlier semester project, four FM demodulation algorithms were developed and tested on their suitability. Two of them (mixed and PLL demodulator) are implemented on the DSP (TMS320C6711DSK).

Because both algorithms demand the same pre and after signal preparation, the demodulation is split into sub projects to reach a high compatibility between the algorithms. First they are realized in floating-point. That allows to concentrate on the algorithms themselves without any effects of a fixed-point implementation. After that, the algorithms are translated step by step from floating-point to fixed-point. Afterwards the algorithms are optimized in time and signal quality. As in the earlier work, the algorithms are tested out with measuring the SINAD and the signal noise ratio of the demodulated signal. Furthermore the needed computing time of the implementations is measured. The two algorithms are compared each other. The mixed demodulator shows a better performance than the PLL.

The Chapters Theory 3 and Simulation 4 were not developed in this project, but in the earlier semester project. Nevertheless they are included to get the whole context of developed digital FM demodulation in a single document.



# Contents

<b>Abstract</b>	<b>i</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Listings</b>	<b>xiv</b>
<b>Acknowledgement</b>	<b>xv</b>
<b>Subject</b>	<b>xvii</b>
Introduction . . . . .	xvii
Task . . . . .	xvii
Reporting . . . . .	xviii
Dates . . . . .	xviii
Organization . . . . .	xviii
<b>1. Introduction</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Objective . . . . .	1
1.3. Specifications . . . . .	1
1.3.1. FM Receiver Architecture . . . . .	1
1.3.2. The Signals . . . . .	2
<b>2. Project Management</b>	<b>3</b>
2.1. Milestones . . . . .	3
2.2. Responsibilities . . . . .	3
<b>3. Theory</b>	<b>5</b>
3.1. Frequency Modulation . . . . .	5
3.2. Algorithms for Signal Pretreatment . . . . .	6
3.2.1. Subsampling . . . . .	6
3.2.2. Quadrature-Mixer . . . . .	8
3.3. Algorithms for Digital FM Demodulation . . . . .	9
3.3.1. Baseband Delay Demodulator . . . . .	9
3.3.2. Phase-Adapter Demodulator . . . . .	11
3.3.3. Phase-Locked Loop . . . . .	13
3.3.4. Mixed Demodulator . . . . .	15

<b>4. Simulation</b>	<b>17</b>
4.1. General Knowledge from Simulations	17
4.1.1. Ideal Model	17
4.1.2. DSP Model	17
4.1.3. DSP Model with Gaussian Noise	19
4.2. Algorithms for Signal Pretreatment	19
4.2.1. FM Modulator	19
4.2.2. Subsampling	19
4.2.3. Quadrature-Mixer	21
4.3. Baseband Delay Demodulator	22
4.3.1. Ideal Model	22
4.3.2. DSP Model	24
4.3.3. DSP Model with Gaussian Noise	25
4.4. Phase-Adapter Demodulator	25
4.4.1. Ideal Model	25
4.5. Phase-Locked Loop	27
4.5.1. Ideal Model	28
4.5.2. DSP Model	29
4.5.3. DSP Model with Gaussian Noise	30
4.6. Mixed Demodulator	31
4.6.1. Ideal Model	31
4.6.2. DSP Model	33
4.6.3. DSP Model with Gaussian Noise	34
4.7. Comparison of the Algorithms	34
4.7.1. Signal Quality	34
4.7.2. Robustness	35
4.7.3. Computing Power and Storage Utilization	36
4.7.4. The Appropriate Algorithm	37
<b>5. Implementation</b>	<b>39</b>
5.1. System Architecture	39
5.1.1. Signal Path	39
5.1.2. The Sampling Rates	42
5.1.3. Analog-to-Digital Conversion THS1408	43
5.1.4. Digital-to-Analog Conversion	47
5.1.5. Programm Flow	47
5.1.6. Interrupt Routines and Scheduling	48
5.2. Floating-Point Algorithm	50
5.2.1. Quadrature Mixer	50
5.2.2. First Down Sampling	54
5.2.3. Mixed Demodulator	54
5.2.4. PLL Demodulator	55
5.2.5. Filter	56
5.2.6. Second Down Sampling	57
5.3. Fixed-Point Algorithm	57
5.3.1. Fractional Q Formats	58
5.3.2. Quadrature Mixer	60
5.3.3. Mixed Demodulator	62
5.3.4. PLL Demodulator	62
5.3.5. Out Filter	69
5.4. Optimization	70

5.4.1. Adaptive Seeking of Carrier Angular Frequency . . . . .	70
5.4.2. Time Optimization . . . . .	71
<b>6. Tests And Results</b>	<b>77</b>
6.1. Spectrum . . . . .	77
6.2. Signal Quality (SINAD) . . . . .	78
6.2.1. Mixed Demodulator . . . . .	79
6.2.2. PLL Demodulator . . . . .	81
6.2.3. Comparison of Mixed and PLL Demodulator . . . . .	82
6.3. Robustness (S/N) . . . . .	83
6.3.1. Mixed Demodulator . . . . .	84
6.3.2. PLL Demodulator . . . . .	84
6.3.3. Comparison of Mixed and PLL Demodulator . . . . .	85
6.4. Computing Time . . . . .	86
6.4.1. Demodulation Functions . . . . .	86
6.4.2. Interrupt Routines . . . . .	90
6.4.3. CPU Processing Load . . . . .	90
<b>7. Conclusions and Recommendations</b>	<b>93</b>
<b>A. Equipment</b>	<b>95</b>
A.1. Hardware . . . . .	95
A.1.1. DSP Board . . . . .	95
A.1.2. Analog-to-Digital Converter . . . . .	96
A.2. Software . . . . .	96
A.2.1. DSP IDE . . . . .	96
A.2.2. MATLAB . . . . .	96
A.3. Signal And Measure Instruments . . . . .	96
A.3.1. Signal Generator . . . . .	96
A.3.2. Oscilloscope . . . . .	97
A.3.3. Audio Measurement System . . . . .	97
<b>B. Abbreviations and Symbols</b>	<b>99</b>
B.1. Formula Symbols . . . . .	99
B.2. Abbreviations . . . . .	100
<b>C. Simulation Measure Algorithms</b>	<b>101</b>
C.1. Harmonic distortion k and SINAD . . . . .	101
C.2. Signal to noise ratio S/N . . . . .	102
<b>D. Bibliography</b>	<b>103</b>
D.1. English Books . . . . .	103
D.2. German Books . . . . .	103
D.3. Texas Instruments Documentation . . . . .	103
<b>E. Simulink models</b>	<b>105</b>
E.1. Baseband-Delaydemodulator . . . . .	105
E.2. Phase-Adapter-Demodulator . . . . .	108
E.3. Phasenregelschleife . . . . .	109
E.4. Mix Demodulator . . . . .	112

<b>F. Listings</b>	<b>115</b>
F.1. C Listings . . . . .	115
F.1.1. ADC and DAC . . . . .	115
F.1.2. Floating-Point . . . . .	119
F.1.3. Fixed-Point . . . . .	126
F.1.4. Fixed-Point Optimized . . . . .	136
F.1.5. Floating-Point Adaptive . . . . .	145
F.2. Linear Assembly Listings . . . . .	146
F.2.1. Fixed-Point Optimized . . . . .	146
F.3. MATLAB Listings . . . . .	149



# List of Figures

1.1. Block diagram FM receiver architecture . . . . .	1
1.2. Spectrum message signal . . . . .	2
1.3. Spectrum FM signal . . . . .	2
2.1. Project schedule . . . . .	4
3.1. Subdivisions of FM demodulation block . . . . .	7
3.2. Spectrum subsampling with even $\lambda$ . . . . .	7
3.3. Spectrum subsampling with odd $\lambda$ . . . . .	8
3.4. Quadrature-Mixer . . . . .	8
3.5. Real quadrature-mixer . . . . .	9
3.6. Complex baseband delay demodulator . . . . .	10
3.7. Real baseband delay demodulator . . . . .	10
3.8. Real amplitude normalization . . . . .	12
3.9. Phase-adapter demodulator . . . . .	12
3.10. Phase-Locked Loop . . . . .	13
3.11. Actual realization Phase-Locked Loop . . . . .	14
3.12. Mixed demodulator . . . . .	15
4.1. Quantizer and saturation blocks in Simulink . . . . .	18
4.2. Simulink block FM modulator . . . . .	19
4.3. Simulink model FM modulators . . . . .	19
4.4. Simulink model subsampling . . . . .	20
4.5. Plot 1 subsampling . . . . .	20
4.6. Plot 2 subsampling . . . . .	20
4.7. Simulink block quadrature-mixer . . . . .	21
4.8. Simulink model quadrature-mixer . . . . .	21
4.9. Plot quadrature-mixer . . . . .	22
4.10. Simulink model baseband delay demodulator . . . . .	22
4.11. Plot of delay demodulator ideal with $k_{FM} = 180$ . . . . .	23
4.12. Plot of delay demodulator ideal with $k_{FM} = 18000$ . . . . .	23
4.13. Spectrum of the chirp signal from a ideal delay demodulator . . . . .	23
4.14. Spectrum of chirp signal for delay demodulator DSP model . . . . .	24
4.15. Plot FM signal with noise . . . . .	26
4.16. Plot noise and output signal with noise . . . . .	26
4.17. Simulink model phase-adapter demodulator ideal . . . . .	26
4.18. Plot phase-adapter demodulator with $k_{FM} = 180$ . . . . .	27
4.19. Plot phase-adapter demodulator with $k_{FM} = 18000$ . . . . .	27
4.20. Plot PLL ideal with $k_{FM} = 1800$ . . . . .	29
4.21. Plot PLL ideal with $k_{FM} = 18000$ . . . . .	29
4.22. Spectrum of the chirp signal for the ideal PLL . . . . .	30

4.23. PLL DSP $k_{FM} = 1800$ , with bandpass filter . . . . .	30
4.24. PLL DSP $k_{FM} = 18000$ , with bandpass filter . . . . .	30
4.25. PLL DSP $k_{FM} = 1800$ , without bandpass filter . . . . .	31
4.26. PLL DSP $k_{FM} = 18000$ , without bandpass filter . . . . .	31
4.27. PLL DSP $k_{FM} = 18000$ , with the bandpass filter in the feedback loop . . . . .	31
4.28. Simulink model mixed demodulator ideal . . . . .	32
4.29. Plot of ideal mixed demodulator with $k_{FM} = 180$ . . . . .	33
4.30. Plot of ideal mixed demodulator with $k_{FM} = 18000$ . . . . .	33
4.31. Spectrum of the chirp signal for an ideal mixed demodulator . . . . .	33
4.32. Spectrum of the chirp signal for DSP mixed demodulator . . . . .	34
4.33. Plot of FM signal with noise . . . . .	35
4.34. Plot of the noise and the output signal . . . . .	35
4.35. Comparison of the harmonic distortion for ideal models of different algorithms with constant $k_{FM}$ of 18000 and variable frequency . . . . .	35
4.36. Comparison of harmonic distortion for ideal models of different algorithms with constant frequency of 2000Hz and variable $k_{FM}$ . . . . .	36
4.37. Comparison of SINAD for DSP models of different algorithms with constant $k_{FM}$ of 18000 and variable frequency . . . . .	36
4.38. Comparison of S/N ratio for different demodulation algorithmen, with a $k_{FM}$ of 18000 . . . . .	37
5.1. Overview of floating-point implementation . . . . .	40
5.2. Overview of fixed-point implementation . . . . .	41
5.3. Input and output samples . . . . .	42
5.4. Spectrum of FM signal after subsampling . . . . .	43
5.5. THS1408 . . . . .	43
5.6. Schema address bus wiring THS1408 . . . . .	44
5.7. Ping Pong Buffering . . . . .	46
5.8. EDMA setup for Ping Pong Buffering . . . . .	46
5.9. Ping Pong Buffering DAC . . . . .	47
5.10. Programm flow . . . . .	48
5.11. Layout main file . . . . .	48
5.12. Scheduling with software interrupt . . . . .	48
5.13. Overview interrupts . . . . .	49
5.14. DAC interrupt routine . . . . .	49
5.15. EDMA interrupt routine . . . . .	50
5.16. Demodulate software interrupt routine . . . . .	50
5.17. Quadrature mixer . . . . .	51
5.18. Frequency response of low-pass filter for quadrature mixing, floating point . . . . .	52
5.19. Digital block diagram of quadrature mixer . . . . .	53
5.20. Signal spectrum after quadrature mixing . . . . .	54
5.21. Signal spectrum after first down sampling . . . . .	54
5.22. Block diagram of mixed demodulator . . . . .	55
5.23. Block diagram of PLL demodulator . . . . .	56
5.24. Band-pass filter frequency response, floating point . . . . .	57
5.25. Signal spectrum before second down sampling . . . . .	57
5.26. Signal spectrum after second down sampling . . . . .	58
5.27. Q.15 Bit Fields . . . . .	58
5.28. Block diagram of cascaded low-pass filter . . . . .	60
5.29. Change in block diagram of . . . . .	60
5.30. Output signals and delay states of cascaded low-pass filter . . . . .	61

5.31. Output signals and delay states of cascaded low-pass filter . . . . .	61
5.32. Plot arc tangent lookup table . . . . .	63
5.33. Sine values with belonging indexes . . . . .	64
5.34. Index manipulation . . . . .	65
5.35. Linear interpolation . . . . .	66
5.36. Sign of correction changes . . . . .	67
5.37. Sign of correction does not change . . . . .	68
5.38. Block diagram transposed filter structure . . . . .	69
5.39. Adapted System . . . . .	70
5.40. Adaptive seeking of carrier angular frequency . . . . .	72
5.41. development flow . . . . .	73
6.1. FFT print of demodulated signal . . . . .	77
6.2. FFT print of DSP generated sine wave . . . . .	78
6.3. FFT print of demodulated signal for the range of 0 to 20kHz . . . . .	78
6.4. Principle of THD+N Measurement . . . . .	79
6.5. SINAD of fixed-point mixed demodulator with various table sizes for direct lookup . . . . .	80
6.6. SINAD of fixed-point mixed demodulator with various table sizes for interpolated lookup . . . . .	80
6.7. SINAD comparison of direct and interpolated lookup for fixed-point mixed demodulator and the floating-point implementation . . . . .	80
6.8. SINAD of fixed-point PLL demodulator with various table sizes for direct lookup . . . . .	81
6.9. SINAD of fixed-point PLL demodulator with various table sizes for interpolated lookup . . . . .	81
6.10. SINAD comparison of direct and interpolated lookup for fixed-point PLL demodulator and the floating-point implementation . . . . .	82
6.11. Error of linear interpolation . . . . .	82
6.12. SINAD comparison of different demodulator implementations . . . . .	83
6.13. Oscilloscope print FM signal normal . . . . .	83
6.14. Oscilloscope print FM signal with noise added . . . . .	83
6.15. Spectrum of an demodulated signal with an S/N ratio of 20dB by the FM signal . . . . .	84
6.16. S/N of the mixed demodulated signal with a S/N of 20dB at the input FM signal . . . . .	84
6.17. S/N of the mixed demodulated signal with a S/N of 15dB at the FM input signal . . . . .	85
6.18. S/N of the PLL demodulated signal with a S/N of 20dB at the input FM signal . . . . .	85
6.19. S/N of the PLL demodulated signal with a S/N of 15dB at the FM input signal . . . . .	85
6.20. S/N of the PLL demodulated signal with a S/N of 15dB at the FM input signal . . . . .	86
6.21. Time measurement results Quadrature mixer ( quad_mix ) . . . . .	87
6.22. Time measurement results mixed demodulator direct lookup ( mixed_demodulate ) . . . . .	88
6.23. Time measurement results mixed demodulator interpolated lookup ( mixed_demodulate ) . . . . .	88
6.24. Time measurement results PLL demodulator direct lookup ( pll_demodulate ) . . . . .	88
6.25. Time measurement results out filter ( out_filter ) . . . . .	89
6.26. Time comparison of the demodulation functions . . . . .	89
6.27. Executing time of interrupt routines . . . . .	90
6.28. CPU processing load . . . . .	91
E.1. Blockschaltbild Simulink Aufbau Idealer Baseband-Delaydemodulator . . . . .	105
E.2. Blockschaltbild Simulink Aufbau DSP Baseband-Delaydemodulator . . . . .	106
E.3. Blockschaltbild Simulink Aufbau DSP Baseband-Delaydemodulator mit gausschem Rauschen . . . . .	107
E.4. Blockschaltbild Simulink Aufbau Idealer Phase-Adapter-Demodulator . . . . .	108

## List of Figures

---

E.5. Blockschaltbild Simulink Aufbau Idealer PLL . . . . .	109
E.6. Blockschaltbild Simulink Aufbau DSP PLL . . . . .	110
E.7. Blockschaltbild Simulink Aufbau DSP PLL mit gausschem Rauschen . . . . .	111
E.8. Blockschaltbild Simulink Aufbau Idealer Mix Demodulator . . . . .	112
E.9. Blockschaltbild Simulink Aufbau DSP Mix Demodulator . . . . .	113
E.10. Blockschaltbild Simulink Aufbau DSP Mix Demodulator mit gausschem Rauschen .	114

# List of Tables

4.1. Parameter quantizer . . . . .	18
4.2. Parameter saturation . . . . .	18
4.3. Parameter FM modulator . . . . .	19
4.4. Parameter quadrature-mixer . . . . .	21
4.5. Harmonic distortion of the ideal delay demodulator . . . . .	23
4.6. SINAD delay demodulator DSP model . . . . .	24
4.7. S/N delay demodulator DSP model . . . . .	25
4.8. Harmonic distortion phase-adaptor demodulator . . . . .	26
4.9. Harmonic distortion PLL . . . . .	29
4.10. SINAD PLL . . . . .	30
4.11. S/N PLL . . . . .	31
4.12. Harmonic distortion factor mixed demodulator ideal . . . . .	32
4.13. SINAD DSP mixed demodulator . . . . .	33
4.14. S/N DSP Mix Demodulator . . . . .	34
5.1. Jumper settings THS1408 . . . . .	44
5.2. Addresses of registers THS1408 . . . . .	44
5.3. Truth table address decode THS1408 . . . . .	45
5.4. EMIF timing registers for THS1408 . . . . .	45
5.5. Filter specifications low-pass filter . . . . .	69
5.6. Filter specifications high-pass filter . . . . .	69



# List of Listings

5.1. DAC interrupt routine . . . . .	49
5.2. EDMA interrupt routine . . . . .	50
5.3. Demodulate softawre interrupt routine . . . . .	50
5.4. Argument calculation fixed-point mixed demodulator . . . . .	62
5.5. PLL demodulator . . . . .	65
5.6. Interpolated Lookup Table . . . . .	67
F.1. adc_THS1408.h . . . . .	115
F.2. adc_THS1408.c . . . . .	116
F.3. dac_codec.h . . . . .	118
F.4. dac_codec.c . . . . .	118
F.5. Floating-Point :: global_settings.h . . . . .	119
F.6. Floating-Point :: fm_dem_main.c . . . . .	119
F.7. Floating-Point :: quad_mix.h . . . . .	121
F.8. Floating-Point :: quad_mix.c . . . . .	122
F.9. Floating-Point :: downsample_one.h . . . . .	123
F.10. Floating-Point :: downsample_one.c . . . . .	123
F.11. Floating-Point :: mixed_demodulate.h . . . . .	123
F.12. Floating-Point :: mixed_demodulate.c . . . . .	123
F.13. Floating-Point :: pll_demodulate.h . . . . .	124
F.14. Floating-Point :: pll_demodulate.c . . . . .	124
F.15. Floating-Point :: out_filter.h . . . . .	125
F.16. Floating-Point :: out_filter.c . . . . .	125
F.17. Floating-Point :: downsample_two.h . . . . .	126
F.18. Floating-Point :: downsample_two.c . . . . .	126
F.19. Fixed-Point :: global_settings.h . . . . .	126
F.20. Fixed-Point :: fm_dem_main.c . . . . .	126
F.21. Fixed-Point :: quad_mix.h . . . . .	129
F.22. Fixed-Point :: quad_mix.c . . . . .	129
F.23. Fixed-Point :: mixed_demodulate.h . . . . .	130
F.24. Fixed-Point :: mixed_demodulate.c . . . . .	130
F.25. Fixed-Point Interpolated :: mixed_demodulate.h . . . . .	131
F.26. Fixed-Point Interpolated :: mixed_demodulate.c . . . . .	131
F.27. Fixed-Point :: pll_demodulate.h . . . . .	132
F.28. Fixed-Point :: pll_demodulate.c . . . . .	132
F.29. Fixed-Point Interpolated :: pll_demodulate.h . . . . .	133
F.30. Fixed-Point Interpolated :: pll_demodulate.c . . . . .	133
F.31. Fixed-Point :: out_filter.h . . . . .	135
F.32. Fixed-Point :: out_filter.c . . . . .	135
F.33. Fixed-Point Optimized :: global_settings.h . . . . .	136
F.34. Fixed-Point Optimized :: fm_dem_main.c . . . . .	137
F.35. Fixed-Point Optimized :: quad_mix.h . . . . .	139

F.36. Fixed-Point Optimized :: quad_mix.c . . . . .	139
F.37. Fixed-Point Optimized :: mixed_demodulate.h . . . . .	140
F.38. Fixed-Point Optimized :: mixed_demodulate.c . . . . .	140
F.39. Fixed-Point Optimized Interpolated :: mixed_demodulate.h . . . . .	141
F.40. Fixed-Point Optimized Interpolated :: mixed_demodulate.c . . . . .	141
F.41. Fixed-Point Optimized :: pll_demodulate.h . . . . .	141
F.42. Fixed-Point Optimized :: pll_demodulate.c . . . . .	142
F.43. Fixed-Point Optimized :: out_filter.h . . . . .	143
F.44. Fixed-Point Optimized :: out_filter.c . . . . .	143
F.45. Floating-Point Optimized Adaptive :: mixed_demodulate.h . . . . .	145
F.46. Floating-Point Optimized Adaptive :: mixed_demodulate.c . . . . .	145
F.47. Fixed-Point Optimized :: quad_mix_lin_ass.sa . . . . .	146
F.48. tanTabGen.m . . . . .	149
F.49. tanTabGenInterp.m . . . . .	150
F.50. SineTabGen.m . . . . .	150
F.51. SineTabGenInt.m . . . . .	151
F.52. mixer_design.m . . . . .	152
F.53. lowpass_design_noisefilter.m . . . . .	153
F.54. highpass_design_noisefilter.m . . . . .	153



# Acknowledgement

Living in Singapore and working on our diploma project for the last twelve weeks have been an enriching experience for us. Without the support of many people, this would not have been possible. At this point we would like to express our appreciation to all persons who were involved.

Thanks to Andreas Ehrensperger, our supervisor in Switzerland, for having the efforts of supporting us from overseas. Dr. Chang Chip Hong, our local supervisor, for his encouraging support here in Singapore. Dr. T. Srikanthan, director of Center for High Performance Embedded Systems. Dr. Ma Jian-Guo for reviewing the report. Much credit to those who made our stay even possible: Peter Schneider, Vice-President of the University of Applied Sciences (HSR), Rapperswil; Ms. Anita Riegler, International Relations Office, University of Applied Sciences (HSR), Rapperswil; Ms. Agnes Yap and Ms. Catherine Tai, International Relations Office, Nanyang Technological University (NTU), Singapore. We would like to thank the students and colleagues in the laboratory who supported us in any form.



# Subject

## Implementation of FM demodulator algorithms on a high performance digital signal processor

### Introduction

Mobile radio provider organizations are planning to replace their traditional analog FM-based radio system, by a new digital system like TETRA, TETRAPOL or GSM-R. To guarantee a seamless operation it could be very useful if the new system could operate in a special mode, which is compatible to the old FM system. From the commercial as well as from the technical point of view it would be very interesting if such an special mode could be implemented as a part of software of the mobile signal processors without need for any additional hardware.

### Task

At least two of the FM-demodulation algorithms, which have been designed and validated during the last HSR semester project, should be realized as real time.

The input signal is a FM-modulated intermediate frequency signal (IF) with the following specifications:

- carrier frequency: 10.7MHz
- maximum frequency deviation: 3kHz
- maximum signal bandwidth: 12.5kHz implementations on a digital signal processor.

This IF-signal should be sampled directly using bandpass-subsampling techniques.

Concerning the implementation of the algorithms, special attention should be paid to the following aspects:

- The implementations should be based on a fixed point architecture. (If a floating point signal processor is used, only the fixed point subset of the instruction set should be used)
- The implementation should be optimized with the highest priority to a low computing load, which lowers the power consumption of the target system.

With the implemented prototypes extensive tests under real world conditions should be made for validation.

## **Reporting**

The report has to be written in English and 6 copies are needed. Three complete reports are for the HSR including a CD-ROM containing the complete set of the elaborated data. Three complete reports also including copies of the CD-ROM for the NTU. The report shall also include a translation of the earlier semester project to have the whole context in an English document.

## **Dates**

Begin of Thesis	22 October 2001
End of Thesis	11 January 2002

## **Organization**

Supervisor NTU	Asst. Prof. Dr. Chang Chip Hong
Supervisor HSR	Andreas Ehrensperger

# 1. Introduction

## 1.1. Motivation

Frequency modulation (FM) is an analog modulation, which is for example used in VHF radio broadcasting. Another field of application for FM is private mobile radio (PMR), which is used by organizations like police, fire departments, railroad or power supply companies, for mobile communication. Those analog systems will be gradually replaced by a digital counterparts in the new generation. For a smooth transition of the two systems, it is essential for the new generation system to be able to communicate with the radio equipment of the old generation. Because all the new radio equipment are based on DSP-Technology, it is obvious and of commercial interest to perform the demodulation with the signal processor instead of adding additional analog hardware.

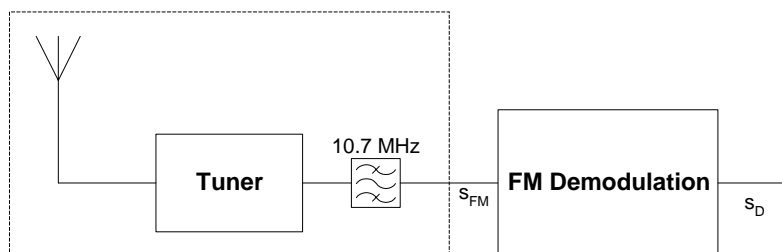
## 1.2. Objective

Various algorithms to perform the demodulation were developed and simulated in the previous semester project. This previous work is included in Chapters Theory 3 and Simulation 4. Two appropriate algorithms are now implemented on a DSP platform. These implementations are then tested for signal quality and robustness.

## 1.3. Specifications

### 1.3.1. FM Receiver Architecture

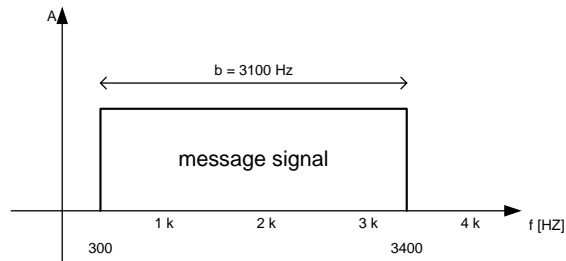
The modulated signal  $s_{FM}$  is frequency limited at an intermediate frequency of 10.7 MHz. The antenna, tuner, and the bandpass filter are given and do not fall in the scope of this work. (see Figure 1.1).



*Figure 1.1.: Block diagram FM receiver architecture*

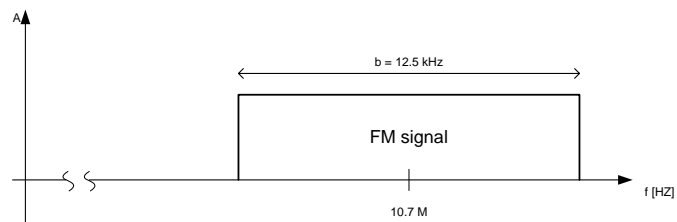
### 1.3.2. The Signals

The message signal  $s_N$  is a speech signal from 300 Hz to 3400 Hz. The amplitude is normalized to one (see Figure 1.2).



*Figure 1.2.: Spectrum message signal*

The FM-Signal  $s_{FM}$  has a bandwidth of 12.5 kHz and a carrier frequency of 10.7 MHz. The amplitude is also normalized to one (see Figure 1.3).



*Figure 1.3.: Spectrum FM signal*

## 2. Project Management

The project was divided into five main tasks, which were further divided into subtasks. For each subtask the required time was estimated shown in time schedule Figure 2.1. At the important stages of the project a milestone was set.

### 2.1. Milestones

**M1 First Floating-Point Demodulation** A first version of both demodulation algorithms is running, where time and quality are not of great interest and the source code is a draft version.

**M2 Floating-Point Demodulation** The two demodulation algorithms are running with a good signal quality and the source is well commented.

**M3 Fixed-Point Demodulation** Both algorithms are running stable on a fixed-point implementation in C, without great effort in speed optimization.

**M4 Optimized Fixed-Point Demodulation** Both algorithms are speed optimized and running stable. Various tests under real world conditions are done to characterize them.

**M4 Report and Presentation** The report is finished and the presentation is prepared.

### 2.2. Responsibilities

The project was carried out in a teamwork and therefore the tasks were distributed among the two persons. This is also shown in Figure 2.1.

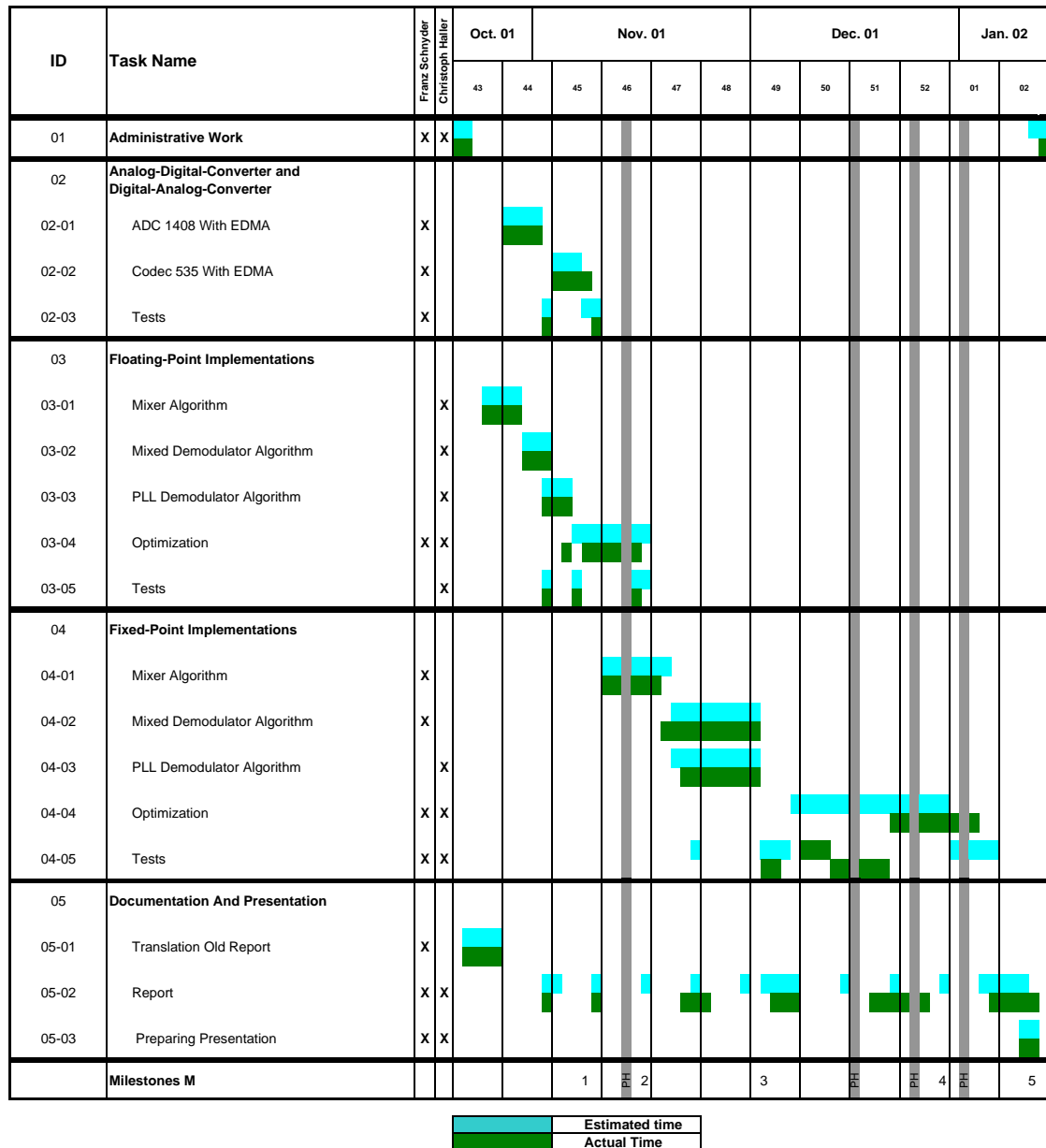


Figure 2.1.: Project schedule



## 3. Theory

### 3.1. Frequency Modulation

Frequency modulation (FM) is a type of angle-modulated signal. A conventional angle modulated signal is defined by the following equation.

$$s_{FM}(t) = A \cdot \cos(\omega_T \cdot t + \phi_{FM}(t)) \quad (3.1)$$

for FM, the relation of  $s_N$  to  $\phi_{FM}$  is given by

$$\phi_{FM}(t) = k_{FM} \cdot \int s_N(t) \cdot dt \quad (3.2)$$

where  $\omega_T$  is the carrier angular frequency expressed in rad/s and  $k_{FM}$  is the modulation index. There is no connection between the spectrum of the message  $s_N(t)$  and the spectrum of the modulated signal. If we use a harmonic message like

$$s_N(t) = K \cdot \cos(\omega_N \cdot t)$$

The modulated signal can be expressed as

$$s_{FM}(t) = A \cdot \cos(\omega_T \cdot t + \phi_{FM}(t))$$

with

$$\phi_{FM}(t) = k_{FM} \cdot \int K \cdot \cos(\omega_N \cdot t) \cdot dt$$

The integral does not convert so

$$\phi_{FM}(t) = k_{FM} \cdot K \cdot \lim_{T \rightarrow \infty} \int_{-T}^t \cos(\omega_N \cdot \tau) \cdot d\tau = \frac{k_{FM} \cdot K}{\omega_N} \cdot \left[ \sin(\omega_N \cdot t) + \lim_{T \rightarrow \infty} \sin(\omega_N \cdot T) \right]$$

$$\phi_{FM}(t) = \frac{k_{FM} \cdot K}{\omega_N} \cdot \sin(\omega_N \cdot t)$$

Let

$$\mu = \frac{k_{FM} \cdot K}{\omega_N}$$

and  $\omega_N = 2 \cdot \pi \cdot f_g$ , the frequency derivation is given by

$$\Delta F = \mu \cdot f_g = \frac{k_{FM} \cdot K}{2 \cdot \pi} \quad (3.3)$$

The modulated FM signal is now

$$s_{FM}(t) = A \cdot \cos[\omega_T \cdot t + \mu \cdot \sin(\omega_N \cdot t)]$$

Using the Bessel function Figure

$$J_n(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{i! \cdot (n+i)!} \cdot \left(\frac{x}{2}\right)^{n+2 \cdot i}$$

and the relation

$$\cos[\alpha + x \cdot \sin(\beta)] = \sum_{n=-\infty}^{\infty} J_n(x) \cdot \cos(\alpha + n \cdot \beta)$$

the FM signal can be written as

$$s_{FM}(t) = A \cdot \sum_{n=-\infty}^{\infty} J_n(\mu) \cdot \cos(\omega_T \cdot t + n \cdot \omega_N)$$

the frequency response can be determined with the Fourier transformation

$$S_{FM}(\omega) = \int_{-\infty}^{\infty} s_{FM}(t) \cdot e^{-j \cdot \omega \cdot t} \cdot dt$$

with

$$\int_{-\infty}^{\infty} \cos(\omega_x \cdot t) \cdot e^{-j \cdot \omega \cdot t} \cdot dt = \pi \cdot [\delta(\omega + \omega_x) + \delta(\omega - \omega_x)]$$

$S_{FM}$  can be expressed as

$$S_{FM}(\omega) = \sum_{n=-\infty}^{\infty} J_n(\mu) \cdot \pi \cdot [\delta(\omega + \omega_T + n \cdot \omega_N) + \delta(\omega - \omega_T - n \cdot \omega_N)]$$

In general the FM bandwidth is not limited, but it declines fast outside a bandwidth around the carrier frequency. This bandwidth can be found using the Carson's rule

$$b_{FM} = 2 \cdot (\Delta F + f_g) \quad (3.4)$$

where  $f_g$  is the highest frequency in the message. Hence the bandwidth rises with a rising  $\Delta F$  and a rising message frequency.

## 3.2. Algorithms for Signal Pretreatment

All the digital FM demodulation algorithms presented in the next section need the FM signal in the baseband. Therefore the FM demodulation unit from Figure 1.1 is further divided into three units: subsampling, quadrature mixing and baseband FM demodulator (see Figure 3.1).

### 3.2.1. Subsampling

The FM signal after the bandpass filter has a carrier frequency of 10.7 MHz and a bandwidth of 12.5 kHz. This results in a maximum frequency of over 10.7 MHz. Hence, a sampling rate of over 21 MHz is required. This data rate is too fast for today DSP's. However as the signal is frequency limited (Bandwidth  $b$ ), a subsampling is possible and the sampling rate can be calculated as follows [6]:

In the special case that

$$\begin{aligned} f_1 &= \lambda \cdot b \quad , \quad \lambda \in [\mathbb{N}] \\ f_2 &= (\lambda + 1) \cdot b \end{aligned}$$

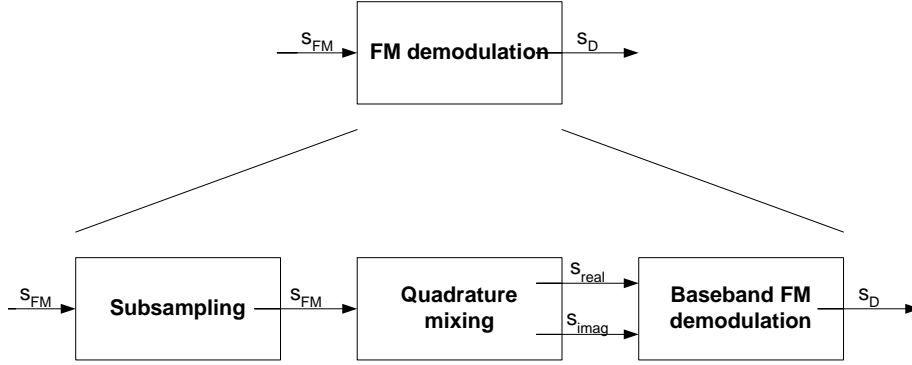
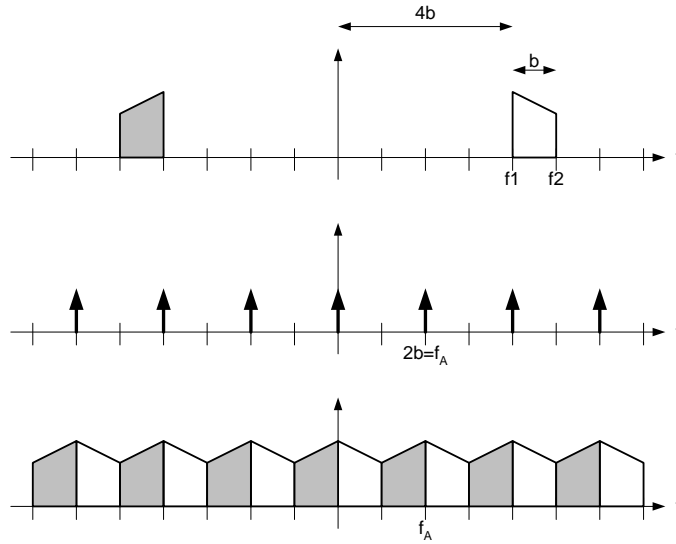


Figure 3.1.: Subdivisions of FM demodulation block

the sample rate is

$$f_A = 2 \cdot b \quad (3.5)$$

for a non aliasing periodic sequel of the spectrum. Figure 3.2 shows the spectrum subsampling for an even  $\lambda$ . Figure 3.3 shows the subsampling for an odd  $\lambda$ .


 Figure 3.2.: Spectrum subsampling with even  $\lambda$ 

If the subsampling is interpreted in terms of the carrier frequency,  $f_T$

$$f_T - \frac{b}{2} = \lambda \cdot b$$

$$f_T = b \frac{2\lambda + 1}{2}$$

For a general carrier frequency  $f_T$  and bandwidth  $b$ , the condition of an even  $\lambda$  is often not fulfilled. Thus, the bandwidth has to increase.

$$b' = b \cdot q \quad q > 1$$

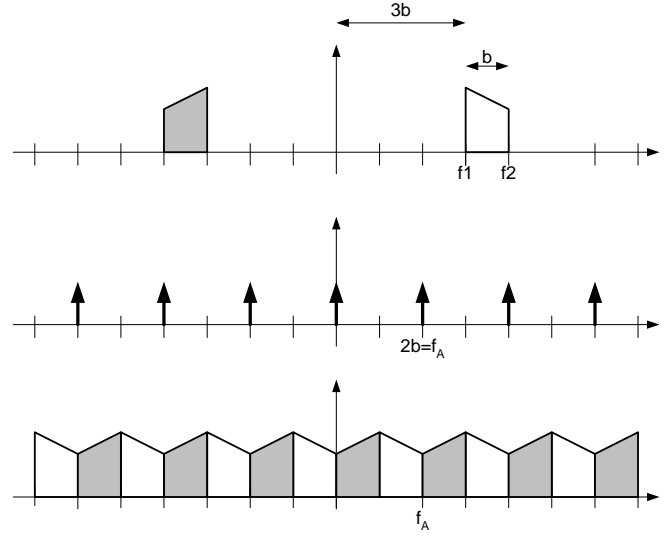


Figure 3.3.: Spectrum subsampling with odd  $\lambda$

the new bandwidth is

$$f_T = b' \frac{2\lambda + 1}{2}$$

$$b' = \frac{2f_T}{2\lambda + 1}$$

where  $\lambda$  is a largest integer number, but smaller than  $\frac{f_T - \frac{b}{2}}{b}$ . Therefore the sampling rate is

$$f_A = 2 \cdot b' = \frac{4f_T}{2\lambda + 1} \quad (3.6)$$

### 3.2.2. Quadrature-Mixer

The mixing to the baseband is carried out by the multiplication of the FM signal and a complex oscillator  $e^{j\omega_T n}$  and a low pass filter [6] (see Figure 3.4). The input signal is the modulated signal  $s_{FM}$

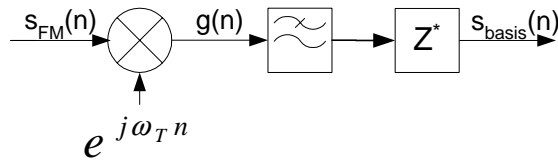


Figure 3.4.: Quadrature-Mixer

$$s_{FM}(n) = A \cdot \cos(\omega_T n + \phi_{FM}(n)) \quad (3.7)$$

The output signal of the mixer is:

$$g(n) = s_{FM}(n) \cdot e^{j\omega_T n} = A \cdot \cos(\omega_T n + \phi_{FM}(n)) \cdot e^{j\omega_T n} = A \cdot \frac{e^{j(\omega_T n + \phi_{FM}(n))} + e^{-j(\omega_T n + \phi_{FM}(n))}}{2} \cdot e^{j\omega_T n}$$

$$= \frac{A}{2} \left[ e^{j(\omega_T n + \phi_{FM}(n) + \omega_T n)} + e^{j(-\omega_T n - \phi_{FM}(n) + \omega_T n)} \right] = \frac{A}{2} \left[ e^{j(2\omega_T n + \phi_{FM}(n))} + e^{j(-\phi_{FM}(n))} \right]$$

$$s_{basis}(n) = (g(n) * TP)^* = \left( \frac{A}{2} e^{-j\phi_{FM}(n)} \right)^* = \frac{A}{2} e^{j\phi_{FM}(n)} = \frac{A}{2} \cos(\phi_{FM}(n)) + j \frac{A}{2} \sin(\phi_{FM}(n)) \quad (3.8)$$

The result is a complex signal  $s_{basis}$ . The mixer can also be realized with real signals by multiplying the FM signal with a sine and cosine oscillation signal (see Figure 3.5). The input signal is again

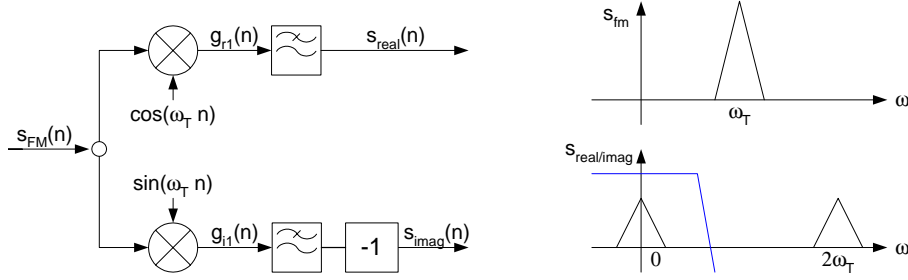


Figure 3.5.: Real quadrature-mixer

the FM signal  $s_{FM}$ .

$$s_{FM}(n) = A \cdot \cos(\omega_T n + \phi_{FM}(n))$$

Therefore, the output signals are

$$\begin{aligned} g_{r1}(n) &= s_{FM}(n) \cdot \cos(\omega_T n) = A \cdot \cos(\omega_T n + \phi_{FM}(n)) \cdot \cos(\omega_T n) \\ &= \frac{A}{2} [\cos(\omega_T n + \phi_{FM}(n) - \omega_T n) + \cos(\omega_T n + \phi_{FM}(n) + \omega_T n)] \\ &= \frac{A}{2} \cos(\phi_{FM}(n)) + \frac{A}{2} \cos(2\omega_T n + \phi_{FM}(n)) \\ s_{real}(n) &= g_{r1}(n) * g_{TP}(n) = \frac{A}{2} \cos(\phi_{FM}(n)) \end{aligned} \quad (3.9)$$

$$\begin{aligned} g_{i1}(n) &= s_{FM}(n) \cdot \sin(\omega_T n) = A \cdot \cos(\omega_T n + \phi_{FM}(n)) \cdot \sin(\omega_T n) \\ &= \frac{A}{2} [\sin(-\omega_T n - \phi_{FM}(n) + \omega_T n) + \sin(\omega_T n + \phi_{FM}(n) + \omega_T n)] \\ &= \frac{A}{2} \sin(-\phi_{FM}(n)) + \frac{A}{2} \sin(2\omega_T n + \phi_{FM}(n)) \\ s_{imag}(n) &= (-1) \cdot g_{i1}(n) * g_{TP}(n) = \frac{A}{2} \sin(\phi_{FM}(n)) \end{aligned} \quad (3.10)$$

It results in two signal, the real part  $s_{real}$  and the imaginary part  $s_{imag}$ , also known as the **I** (Inphase) and **Q** (Quadraturephase) signals.

### 3.3. Algorithms for Digital FM Demodulation

#### 3.3.1. Baseband Delay Demodulator

As the name implies, the baseband delay demodulator needs the FM-Signal in the baseband. For that reason a quadrature mixing (see Section 3.2.2) has to be done first. Figure 3.6 shows the block diagram of the complex baseband delay demodulator. The input signal is the complex FM signal

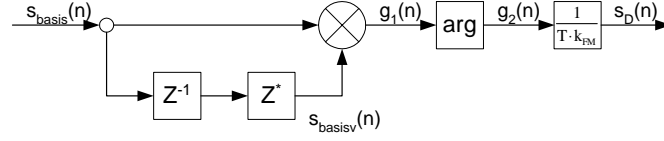


Figure 3.6.: Complex baseband delay demodulator

in the baseband  $s_{basis}$  given in Eq. 3.8. With Eq. 3.2 the system can be described as follow, where  $T$  is the sample time.

$$\begin{aligned}
 g_1(n) &= s_{basis} \cdot s_{basisv} = e^{j\phi_{FM}(n)} \cdot e^{-j\phi_{FM}(n-1)} = e^{j(\phi_{FM}(n) - \phi_{FM}(n-1))} \\
 g_2(n) &= \arg(g_1(n)) = \phi_{FM}(n) - \phi_{FM}(n-1) \\
 s_D(n) &= \frac{g_2(n)}{T \cdot k_{FM}} = \frac{\phi_{FM}(n) - \phi_{FM}(n-1)}{T \cdot k_{FM}} = \frac{\phi'_{FM}(n)}{k_{FM}} = \frac{k_{FM} \cdot s_N(n)}{k_{FM}} = s_N(n) \quad (3.11)
 \end{aligned}$$

The output signal  $s_D$  is equal to the original message  $s_N$ . Hence, the system demodulates a complex FM signal in the baseband,  $s_{basis}$ .

One complex multiplication needs four real multiplications and is therefore time-consuming. With the formula of Euler the calculation can be written as

$$\begin{aligned}
 g_1(n) &= e^{j\phi_{FM}(n)} \cdot e^{-j\phi_{FM}(n-1)} = [\cos(\phi_{FM}(n)) + j \sin(\phi_{FM}(n))] \cdot [\cos(\phi_{FM}(n-1)) - j \sin(\phi_{FM}(n-1))] \\
 &= \cos(\phi_{FM}(n)) \cos(\phi_{FM}(n-1)) + \sin(\phi_{FM}(n)) \sin(\phi_{FM}(n-1)) \\
 &\quad + j [\sin(\phi_{FM}(n)) \cos(\phi_{FM}(n-1)) - \cos(\phi_{FM}(n)) \sin(\phi_{FM}(n-1))] \\
 &= \cos(\phi_{FM}(n) - \phi_{FM}(n-1)) + j \sin(\phi_{FM}(n) - \phi_{FM}(n-1)) = e^{j(\phi_{FM}(n) - \phi_{FM}(n-1))}
 \end{aligned}$$

It shows that the required information appears in the real part as well as in the imaginary part. Thus the system can be reduced to the imaginary part. Using the two real signals in the baseband  $s_{real}$  from Eq. 3.9 and  $s_{imag}$  from Eq. 3.10 as input signals.

Figure 3.7 shows the block diagram of the real baseband delay demodulator. The output signal is

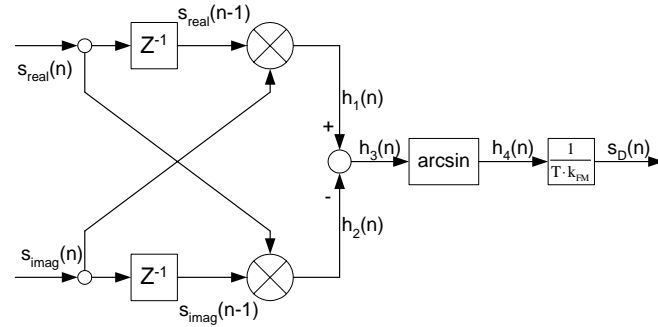


Figure 3.7.: Real baseband delay demodulator

$$\begin{aligned}
 h_1(n) &= s_{imag}(n) \cdot s_{real}(n-1) = \sin(\phi_{FM}(n)) \cdot \cos(\phi_{FM}(n-1)) \\
 h_2(n) &= s_{real}(n) \cdot s_{imag}(n-1) = \cos(\phi_{FM}(n)) \cdot \sin(\phi_{FM}(n-1)) \\
 h_3(n) &= h_1(n) - h_2(n) = \sin(\phi_{FM}(n)) \cos(\phi_{FM}(n-1)) - \cos(\phi_{FM}(n)) \sin(\phi_{FM}(n-1)) \\
 &= \sin(\phi_{FM}(n) - \phi_{FM}(n-1))
 \end{aligned}$$

$$s_D(n) = \arcsin h_3(n) \cdot \frac{1}{T \cdot k_{FM}} = \frac{\phi_{FM}(n) - \phi_{FM}(n-1)}{T \cdot k_{FM}} = \frac{\phi'_{FM}(n)}{k_{FM}} = \frac{k_{FM} \cdot s_N(n)}{k_{FM}} = s_N(n) \quad (3.12)$$

It is equal to the original message signal  $s_N$ . The system demodulates the two real FM signals in the baseband.

The signal after the arcsine

$$h_4(n) = \phi_{FM}(n) - \phi_{FM}(n-1) = \phi'_{FM} \cdot T = k_{FM} \cdot s_N(n) \cdot T = k_{FM} \cdot s_N(n) \cdot \frac{1}{f_A}$$

must be limited between  $-\frac{\pi}{2}$  and  $\frac{\pi}{2}$  to be clearly defined

$$\max \left( k_{FM} \cdot s_N(n) \cdot \frac{1}{f_A} \right) = \left| \frac{k_{FM} \cdot \hat{s}_N}{f_A} \right| < \frac{\pi}{2}$$

From Eq. 3.3 the maximal frequency deviation can be calculated for flawless demodulation.

$$\begin{aligned} k_{FM} \cdot \hat{s}_N &= \Delta F \cdot 2 \cdot \pi \\ \frac{\Delta F \cdot 2 \cdot \pi}{f_A} &< \frac{\pi}{2} \\ \Delta F &< \frac{\pi}{2} \cdot \frac{f_A}{2 \cdot \pi} < \frac{f_A}{4} \end{aligned} \quad (3.13)$$

It shows that the maximal  $\Delta F$  depends on the sampling rate. Therefore it is not limited, because with rise of  $\Delta F$  also the bandwidth of the FM-Signal rises. Thus the sample rate has to be increased.

The delay demodulator needs the modulated signal to have a constant amplitude, therefore an amplitude normalization has to be done prior to the delay demodulator.

### Amplitude normalization

The received FM signal due to distortion in the channel is not known. However the delay demodulator needs a constant amplitude which is achieved by normalizing the magnitude of the signal. This is done by dividing the complex signal (Eq. 3.8) by its absolute value.

$$s_{out} = \frac{s_{basis}}{|s_{basis}|} = \frac{a(n) \cdot e^{j\phi_{FM}(n)}}{|a(n) \cdot e^{j\phi_{FM}(n)}|} = \frac{a(n) \cdot e^{j\phi_{FM}(n)}}{a(n)} = e^{j\phi_{FM}(n)}$$

The same applies to the two real signals of Eq. 3.9 and Eq. 3.10 (see Figure 3.8).

The output signals are

$$out(n) = \frac{s_{real} + js_{imag}}{|s_{real} + js_{imag}|} = \frac{s_{real} + js_{imag}}{\sqrt{s_{real}^2 + s_{imag}^2}} = \frac{s_{real}}{\sqrt{s_{real}^2 + s_{imag}^2}} + j \frac{s_{imag}}{\sqrt{s_{real}^2 + s_{imag}^2}}$$

The result is a constant amplitude signal normalized to one.

### 3.3.2. Phase-Adapter Demodulator

The phase-adapter demodulator also needs a FM signal in the baseband as input signal (see Section 3.2.2). It works with real signal, so the input is  $s_{real}$  of Eq. 3.9 and  $s_{imag}$  of Eq. 3.10. Figure 3.9 shows the block diagram. The equation for the output signal is:

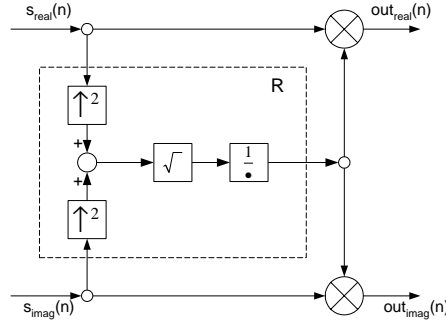


Figure 3.8.: Real amplitude normalization

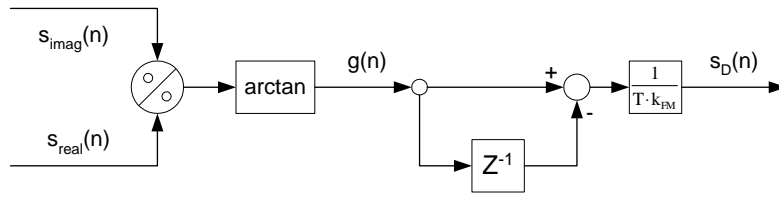


Figure 3.9.: Phase-adapter demodulator

$$g(n) = \arctan\left(\frac{s_{\text{imag}}(n)}{s_{\text{real}}(n)}\right) = \arctan\left(\frac{\sin(\phi_{FM}(n))}{\cos(\phi_{FM}(n))}\right) = \arctan(\tan(\phi_{FM}(n))) = \phi_{FM}(n)$$

$$s_D(n) = \frac{g(n) - g(n-1)}{T \cdot k_{FM}} = \frac{g'(n)}{k_{FM}} = s(n) \quad (3.14)$$

The output signal  $s_D$  is equal to the original message  $s_N$ . The system demodulates the FM signal in the baseband.

The signal after the arc tangent function  $g(n) = \phi_{FM}(n)$  must be limited between  $-\frac{\pi}{2}$  and  $\frac{\pi}{2}$  to be clearly defined. Assuming sinusoidal message signal, the following conditions can be derived:

$$|\phi_{FM}(n)| < \frac{\pi}{2}$$

$$|\phi_{FM}(n)| = \left| \frac{k_{FM} \cdot K}{\omega_N} \sin(\omega_N \cdot t) \right| = \frac{k_{FM} \cdot K}{\omega_N} < \frac{\pi}{2}$$

From Eq. 3.3 the maximal frequency derivation can be determined.

$$k_{FM} \cdot \hat{s}_N = \Delta F \cdot 2 \cdot \pi$$

$$\frac{\Delta F \cdot 2 \cdot \pi}{2 \cdot \pi \cdot f_N} = \frac{\Delta F}{f_N} < \frac{\pi}{2}$$

$$\Delta F < \frac{\pi \cdot f_N}{2} \quad (3.15)$$

The maximum derivation  $\Delta F$  depends on the frequency of the message. In the worst case, for very low message frequencies, the maximal derivation will be very low and not practicable for most applications. Therefore this demodulator is only useful for narrowband FM.

Because of the limitation of  $\phi_{FM}$  the problem of a division by zero is eliminated. This is because the signal  $s_{\text{real}} = \cos(\phi_{FM})$  can only be zero for  $\phi_{FM} = \pm \frac{\pi}{2} \cdot i$ , where  $i$  is an odd integer.



### 3.3.3. Phase-Locked Loop

The Phase-Locked Loop (PLL) is a feedback loop. Besides digital demodulation, it is also used for carrier and timing regeneration. Figure 3.10 shows the block diagram of a Phase-Locked Loop. FM

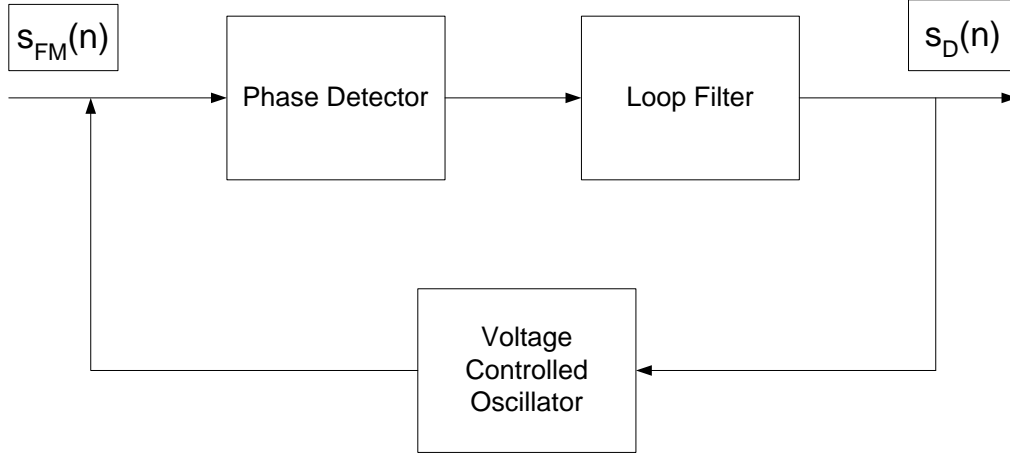


Figure 3.10.: Phase-Locked Loop

modulation stores the information in the variation of frequency. The idea of a PLL is the following: With the help of the feedback loop, the controlled frequency should closely follow the reference frequency. The controlled frequency in this case is the demodulated message  $s_D(n)$ , which should correspond to the original message  $s_N(n)$ . The reference is given by the modulated signal  $s_{FM}(n)$ , which indirectly represents the message  $s_N(n)$ . The Phase detector detects the phase difference between the modulated signal and the voltage controlled oscillator (VCO). This phase difference is filtered by the loop filter that the message signal results.

The actual realization of the PLL is shown in Figure 3.11. The function of the feedback loop will be discussed now. The goal is to describe mathematically the association between the reference signal  $s_{FM}(n)$  and the controlled signal  $s_{FM}(n)$ .  $s_{FM}(n)$  is:

$$s_{FM}(n) = A \cdot \cos(\omega_T \cdot n + k_{FM} \cdot \sum_{i=0}^{n-1} s_N(i))$$

The PLL as shown in Figure 3.11 is a baseband PLL, so the FM-Signal  $s_{FM}(n)$  will be mixed to the baseband prior to the demodulation (see Section 3.2.2). The upper signal path is:

$$\frac{A}{2} \cdot \cos(k_{FM} \cdot \sum_{i=0}^{n-1} s_N(i))$$

and the lower is:

$$\frac{A}{2} \cdot \sin(k_{FM} \cdot \sum_{i=0}^{n-1} s_N(i))$$

Now the demodulated signal  $s_D(n)$  can be written as:

$$s_D(n) = \left[ \frac{A}{2} \cdot \sin(k_{FM} \cdot \sum_{i=0}^{n-1} s_N(i)) \cdot \cos(k_{FM} \cdot \sum_{i=0}^{n-1} s_D(i)) - \frac{A}{2} \cdot \cos(k_{FM} \cdot \sum_{i=0}^{n-1} s_N(i)) \cdot \sin(k_{FM} \cdot \sum_{i=0}^{n-1} s_D(i)) \right] * g(n) \quad (3.16)$$

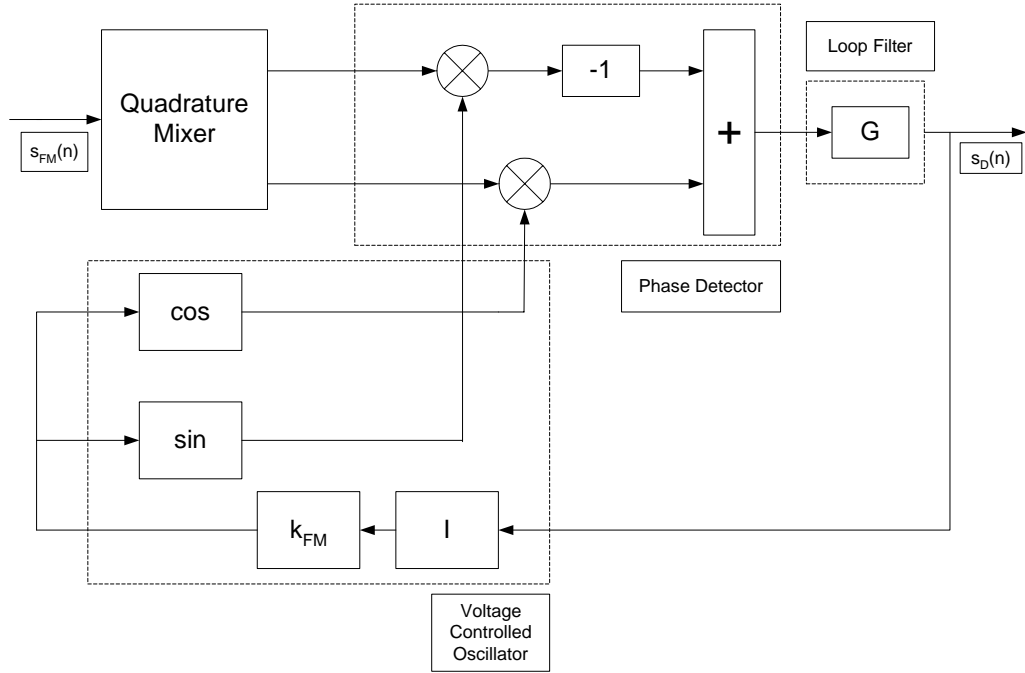


Figure 3.11.: Actual realization Phase-Locked Loop

It's difficult to discuss the equation above because of the convolution. Therefore a transformation to the Z-Domain would simplify the analysis, this is not possible because of the nonlinear loop. To get rid of the convolution, the transfer function of the filter  $G$  is reduced to a constant  $P$ . The convolution is now replaced by a multiplication. Further more the brackets can be simplified by the sine rule:

$$s_D(n) = \frac{A \cdot P}{2} \cdot \sin\left(k_{FM} \cdot \sum_{i=0}^{n-1} s_N(i) - k_{FM} \cdot \sum_{i=0}^{n-1} s_D(i)\right) = \frac{A \cdot P}{2} \cdot \sin\left(k_{FM} \cdot \sum_{i=0}^{n-1} s_N(i) - s_D(i)\right) \quad (3.17)$$

The equation can be written as:

$$\sum_{i=0}^{n-1} s_N(i) - s_D(i) = \frac{\arcsin\left(\frac{s_D(n) \cdot 2}{A \cdot P}\right)}{k_{FM}} \quad (3.18)$$

The aim of the feedback loop is to make:

$$s_N(n) = s_D(n)$$

This can be achieved if the right hand side of Eq. 3.18 is equal to zero. To do so, the constant  $P$  or  $k_{FM}$  needs to be big enough. The PLL can be described by the equation:

$$\sum_{i=0}^{n-1} k_{FM} \cdot s_N(i) - k_{FM} \cdot s_D(i) = 0$$

To fulfil the equation the PLL needs to control to:

$$s_D(n) = s_N(n)$$

Like described above the filter  $G$  was reduced to a constant  $P$  to demodulate the FM signal. The PLL is a nonlinear system. Nonlinear Systems have the characteristic to produce non harmonic frequencies. These frequencies are undesirable because they cause a distortion. One way to reduce this is by adding a filter to the loop based on the specification of the message signal. However the simulation showed that it makes the result even worse.

### 3.3.4. Mixed Demodulator

The mixed demodulator is a combination of the delay demodulator and the phase adapter Demodulator. That way, some of the disadvantages can be removed. Figure 3.12 shows the block diagram

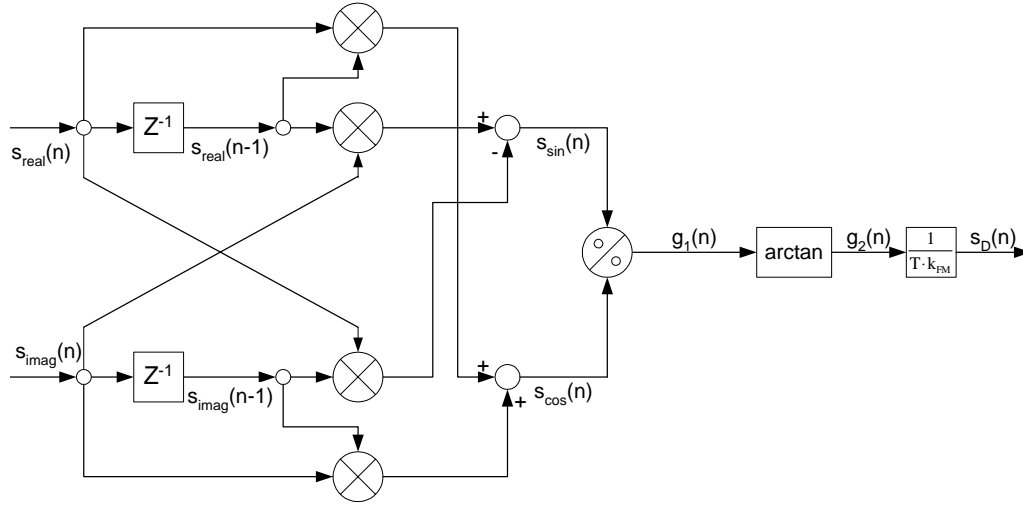


Figure 3.12.: Mixed demodulator

of the mixed demodulator. The input signals are the two real signals  $s_{real}$  and  $s_{imag}$  after the mixing Eq. 3.9 and Eq. 3.10. From Eq. 3.2 the output signal can be written as:

$$\begin{aligned} s_{\sin}(n) &= s_{imag}(n) \cdot s_{real}(n-1) - s_{real}(n) \cdot s_{imag}(n-1) \\ &= \sin(\phi_{FM}(n)) \cdot \cos(\phi_{FM}(n-1)) - \cos(\phi_{FM}(n)) \cdot \sin(\phi_{FM}(n-1)) \\ &= \sin(\phi_{FM}(n) - \phi_{FM}(n-1)) \end{aligned}$$

$$\begin{aligned} s_{\cos}(n) &= s_{real}(n) \cdot s_{real}(n-1) + s_{imag}(n) \cdot s_{imag}(n-1) \\ &= \cos(\phi_{FM}(n)) \cdot \cos(\phi_{FM}(n-1)) + \sin(\phi_{FM}(n)) \cdot \sin(\phi_{FM}(n-1)) \\ &= \cos(\phi_{FM}(n) - \phi_{FM}(n-1)) \end{aligned}$$

$$\begin{aligned} g_1(n) &= \frac{s_{\sin}(n)}{s_{\cos}(n)} = \frac{\sin(\phi_{FM}(n) - \phi_{FM}(n-1))}{\cos(\phi_{FM}(n) - \phi_{FM}(n-1))} = \tan(\phi_{FM}(n) - \phi_{FM}(n-1)) \\ g_2(n) &= \arctan(g_1(n)) = \phi_{FM}(n) - \phi_{FM}(n-1) \end{aligned}$$

$$s_D(n) = \frac{g_2(n)}{T \cdot k_{FM}} = \frac{\phi_{FM}(n) - \phi_{FM}(n-1)}{T \cdot k_{FM}} = \frac{\phi'_{FM}(n)}{k_{FM}} = s_N(n) \quad (3.19)$$

The mixed demodulator demodulates the FM signals in the baseband.

Similarly the signal after the arc tangent  $g_2(n)$  needs to be limited between  $-\frac{\pi}{2}$  and  $\frac{\pi}{2}$ . Therefore the same conclusions for the  $\Delta F$  as in the the delay demodulator can be made (refer to Eq. 3.13).  $\Delta F$  depends only on the sampling rate.

Because of the limitation of the signal  $g_2(n)$  the problem of a division by zero does not exist. The signal

$$s_{\cos}(n) = \cos(\phi_{FM}(n) - \phi_{FM}(n-1)) = \cos(g_2(n))$$

can be zero only when  $\phi_{FM} = \pm \frac{\pi}{2} \cdot i$ , where  $i$  is an odd integer. To avoid a division by zero, the output of the delay  $z^{-1}$  for the first sample needs to be initialized to a non-zero number.

## 4. Simulation

All the algorithms introduced in Section 3.3 will be simulated and evaluated by using SIMULINK<sup>®</sup> (Dynamic Systems for MATLAB<sup>®</sup>) models.

In this chapter, constants and parameters will be chosen. One parameter is  $k_{FM}$  which is determined by the signal specification. It can be calculated with Eq. 3.4 and the given bandwidth  $b_{FM}$ :

$$\begin{aligned} b_{FM} &= 2 \cdot (\Delta F + f_g) = 2 \cdot \left( \frac{\hat{s} \cdot k_{FM}}{2\pi} + f_g \right) \\ \hat{s} \cdot k_{FM} &= 2\pi \left( \frac{b_{FM}}{2} - f_g \right) \\ k_{FM} &= \frac{2\pi}{\hat{s}} \left( \frac{b_{FM}}{2} - f_g \right) = \frac{2\pi}{1} \left( \frac{12500}{2} - 3400 \right) = 2\pi \cdot 2850 \approx 18000 \end{aligned} \quad (4.1)$$

To test the correctness of the algorithms, an ideal model was created for each algorithm. A DSP model was also created for each one of them, to cover the real time performance of the implementation on hardware. To evaluate the robustness of the algorithms, Gaussian noise is added to the channel in the model.

The next section covers some of the knowledge, which apply to all algorithms and are mentioned here not to repeat them each time.

### 4.1. General Knowledge from Simulations

#### 4.1.1. Ideal Model

To make a statement about the quality of the signal, the harmonic distortion  $k$  was measured. In Appendix C.1 the procedure of measuring the harmonic distortion is explained. With the help of the Chirpsignal block of Simulink, a statement about the frequency response is possible. It shows that for a rising  $k_{FM}$  the signal quality sinks. This effect isn't caused by the algorithm, it appears as a result of the aliasing of the spectrum. The FM signal is in principle not bandlimited. The bandwidth based on Carson includes only 90 % of the energy of the signal. So the aliasing appears with a higher  $k_{FM}$  after the subsampling.

#### 4.1.2. DSP Model

There are fixed-point or floating-point DSP-Boards. The representable range of a 32bit fixed-point number is about

$$\left[ -\frac{2^{16}}{2} \dots + \frac{2^{16} - 1}{2} \right]$$

The range of a floating-point representation is

$$\pm [10^{-38} \dots 10^{38}]$$

Floating-point numbers are more complex and so they need more processing time than fixed-point numbers. The simulations are made for fixed-points. This has the advantage of speed but also

possesses some disadvantages. The number range is smaller and a quantization error is present. Also overflows can occur. For these reasons Quantizer and Saturation blocks were added to the models. Figure 4.1 shows these blocks and Table 4.1 and Table 4.2 show the parameters of these

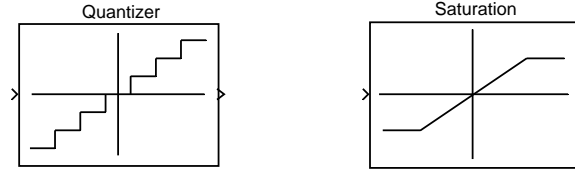


Figure 4.1.: Quantizer and saturation blocks in Simulink

blocks.

Table 4.1.: Parameter quantizer

Quantization interval	q
-----------------------	---

Table 4.2.: Parameter saturation

Upper limit	$G_O$
Lower limit	$G_U$

There are two common ways to define the quantization interval  $q$  and the limits  $G_O$ ,  $G_U$ . One possible set is :

$$q = 1$$

$$G_O = \frac{2^{16} - 1}{2}$$

$$G_U = -\frac{2^{16} - 1}{2}$$

the other is:

$$q = \frac{2}{2^{16} - 1}$$

$$G_O = 1$$

$$G_U = -1$$

For the simulation the second way, which is often called the Q format, was chosen.

The quantization can also be interpreted by adding Gaussian noise to the signal. For that reason it makes sense to filter the demodulated signal  $s_D(n)$  with a bandpass at the end. This bandpass needs a pass band from 100Hz to 4000Hz, this bandwidth covers the specification of the message signal with a little reserve. The filter was designed as butterworth filter as followed:

Type	Butterworth
Order	6
Lower $f_g$	100Hz
Upper $f_g$	4000Hz

To make a statement about the quality of the demodulated signal, the SINAD was measured by the DSP model. Appendix C.1 explains the exact procedure of the measurement.

### 4.1.3. DSP Model with Gaussian Noise

To simulate a interference channel a Gaussian noise was added to the modulated signal. Even this is a very simple model, some good statements about the robustness of the algorithm can be made. The Signal to Noise Ratio (S/N) was measured at the demodulated signal. The power of the noise was set, so the S/N ratio of the modulated signal is 10dB. The exact procedure of the measurement is explained in Appendix C.2. The S/N gets better with a rising  $k_{FM}$ , this is due to the bigger spectrum spreading. Therefore a compromise between a big  $k_{FM}$  for a good S/N and a small  $k_{FM}$  for a better signal quality has to be determined. A good  $k_{FM}$  value is provided by Eq. 3.4 which is needed for calculating the Carson bandwidth.

## 4.2. Algorithms for Signal Pretreatment

### 4.2.1. FM Modulator

To do the simulation, a FM Modulator block was created in Simulink. Figure 4.2 shows the block

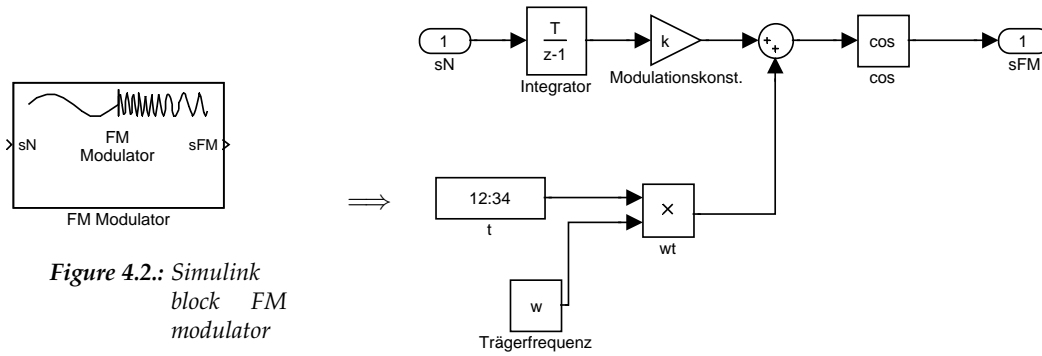


Figure 4.2.: Simulink block FM modulator

Figure 4.3.: Simulink model FM modulators

of the FM Modulator and Figure 4.3 shows the complete model. Table 4.3 shows the parameters of the block.

Table 4.3.: Parameter FM modulator

FM Constant		FM modulation constant $k_{FM}$
Sample Time	sec	Sample time of the simulation
Carrier Frequency	rad/sec	Carrier frequency of FM signals

### 4.2.2. Subsampling

Figure 4.4 shows the structure of the subsampling model. The simulation works with two different frequencies  $f_H$  und  $f_A$ .  $f_A$  is the sampling rate which can be calculated according to Eq. 3.6

$$\frac{f_T - \frac{b}{2}}{b} = \frac{10700000 - \frac{12500}{2}}{12500} = 855.5 \Rightarrow \lambda = 854$$

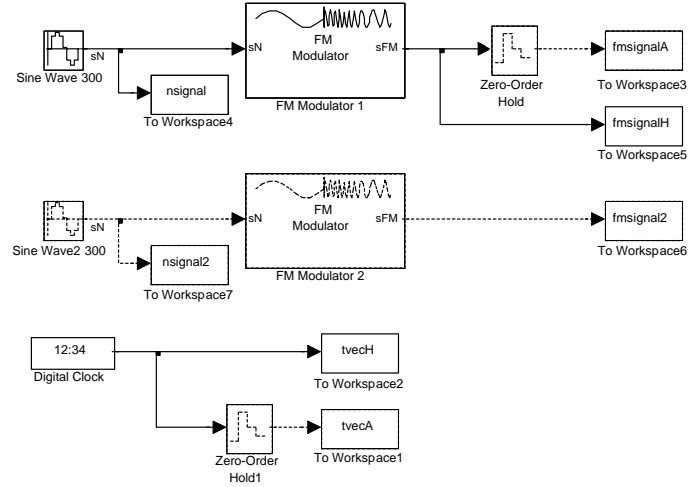


Figure 4.4.: Simulink model subsampling

$$f_A = \frac{4f_T}{2\lambda + 1} = \frac{4 \cdot 10700000}{2 \cdot 854 + 1} = 25043.89\text{Hz}$$

The second frequency  $f_H$  is used to simulate the continues, analog system which will be subsampled, and fulfils the sampling theorem for the modulated signal. It also has to be a multiple of  $f_A$  to run the simulation. All this results in

$$f_H = 855 \cdot f_A = 21412521.94\text{Hz}$$

The FM Modulator 1 modulates the message to a carrier frequency of 10.7Mhz. This signal is once connected straight to output fmsignalH and once subsampled with  $f_A$  and connected to output fmsignalA. The same message is modulated with FM Modulator 2 to a carrier frequency of  $\frac{f_A}{4}$  and connected to output fmsignal2.

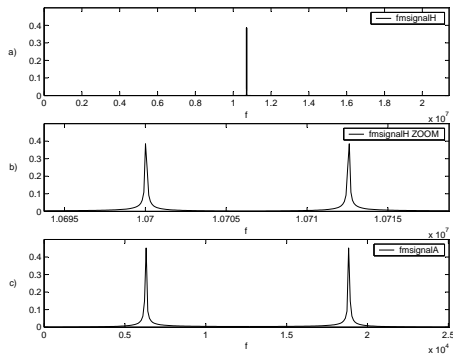


Figure 4.5.: Plot 1 subsampling

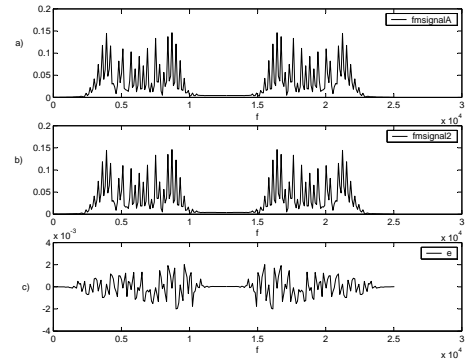


Figure 4.6.: Plot 2 subsampling

Figure 4.5 shows the plot of the simulation with  $k_{FM} = 10$ . A small spectrum for the FM-Signal results. The spectrum of the subsampled signal fmsignalA is compared to the original spectrum fmsignalH. Figure 4.5 (a) shows the original spectrum over the entire frequency band from DC to  $f_H$ , (b) shows the spectrum of the same signal just for the frequencies from  $854 \cdot f_A$  to  $855 \cdot f_A$ ,



this is the part that is shifted by subsampling. (c) shows the spectrum of the subsampled signal. It shows that the FM signal has a carrier of  $\frac{f_A}{4}$ . Furthermore it shows that b) is shifted to c) as described in Section 3.2.1.

Figure 4.6 shows the plot of the simulation with  $k_{FM} = 18000$ , which results in a wide spectrum. The spectrum of the subsampled signal `fmsignalA` is compared to the spectrum of the signal which was modulated direct to a carrier of  $\frac{f_A}{4}$ . Figure 4.6 (a) shows the spectrum of the subsampled signal, (b) the spectrum of the direct modulated signal and (c) the error signal  $e$  given by:

$$e = \text{fft}(\text{fmsignal2}) - \text{fft}(\text{fmsignalA})$$

The simulation result shows that the two spectrums are nearly identical. Hence, in subsequent simulations, the subsampling will be left out and the message will be modulated directly by a carrier of a quarter of the sampling rate.

### 4.2.3. Quadrature-Mixer

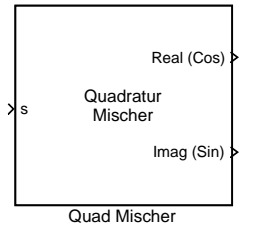


Figure 4.7.: Simulink block quadrature-mixer

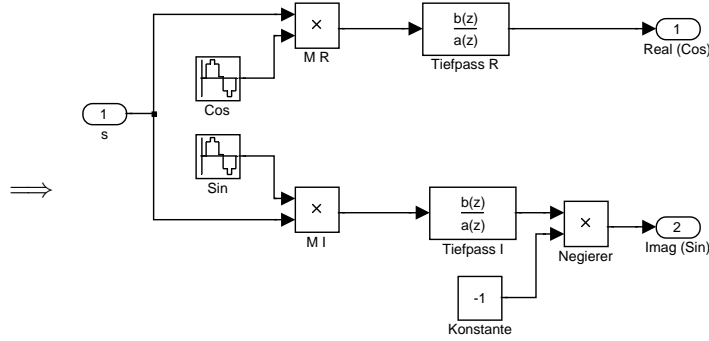


Figure 4.8.: Simulink model quadrature-mixer

Figure 4.7 shows the block of the Quadrature-Mixer and Figure 4.8 shows the complete subsystem model. Table 4.4 shows the parameters of the blocks. The two low-pass filters have butterworth

Table 4.4.: Parameter quadrature-mixer

Filter Order		Order of low-pass filter I low-pass filter R
Sample Time	sec	Sample time of the simulation
Edge Frequency	Hz	Edge Frequency of low-pass filter I low-pass filter R
Carrier Frequency	rad/sec	The carrier frequency of the FM signals

response low-passes and are calculated by MATLAB  $[b,a]=\text{butter}(\text{ordnung},2*f*T)$ .  $f$  is the cutoff frequency and  $T$  the sample time. Figure 4.9 shows the spectrum of the input signal  $s_{FM}$  and the two output signal  $s_{real}$  and  $s_{imag}$  with the following parameters.

Filter Order	10
Sample Time	$40\mu s$
Edge Frequency	$6250\text{Hz}$
Carrier Frequency	$2 \cdot \pi \cdot 6250\text{rad/sec}$

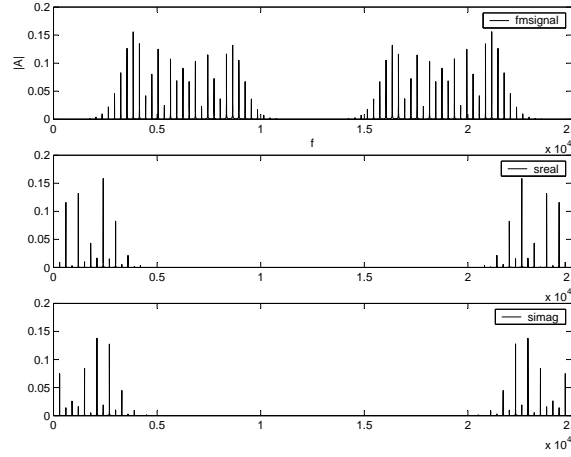


Figure 4.9.: Plot quadrature-mixer

### 4.3. Baseband Delay Demodulator

#### 4.3.1. Ideal Model

Figure 4.10 shows the Simulink model of the ideal baseband delay demodulator. Figure E.4 in the Appendix shows the layout for the simulation. Table 4.5 shows the harmonic distortion factor

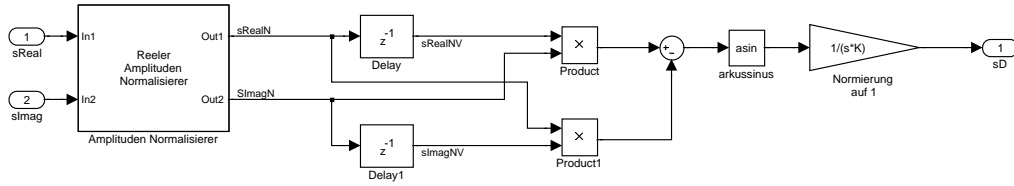


Figure 4.10.: Simulink model baseband delay demodulator

$k$  measured with various  $k_{FM}$  and various frequencies of a harmonic sinusoidal message signal. With Eq. 3.13 the maximal  $\Delta F$  and the maximal  $k_{FM}$  can be calculated.

$$k_{FM} = \frac{2 \cdot \pi \cdot f_A}{4\hat{s}} = \frac{\pi \cdot f_A}{2} = 39'270$$

This  $k_{FM}$  can never be reached due to the bandwidth. The increase in the harmonic distortion with rising  $k_{FM}$  and rising frequency is caused by the increasing bandwidth, which results in aliasing. Figures 4.11 and 4.12 show (a) the spectrum of the input signal `insignal`, (b) the spectrum of the output signal `outsignal` and (c) a time slot of the input and the output signal. The input signal is a sinusoidal with a frequency of 2000 Hz. In Figure 4.11,  $k_{FM}$  was set to 180 and in Figure 4.12 it was set to 18000. It shows that a low-pass filter with a 3dB cutoff frequency  $f_g \approx 4000\text{Hz}$  at the output would improve the signal quality for higher  $k_{FM}$ .

To make statement about the frequency response of the signal a chirp signal from 300 to 3400 Hz was applied. Figure 4.13 shows (a) the spectrum of the input signal and (b) to (e) the spectra of the output signal with various  $k_{FM}$ . Therefore, that with a higher  $k_{FM}$  and a high frequency the response degrades. This is again because of the aliasing.

Table 4.5.: Harmonic distortion of the ideal delay demodulator

$k_{FM}$	$f_N$	k
18	300	0.000
180	300	0.000
1800	300	0.000
18000	300	0.012
18	2000	0.000
180	2000	0.000
1800	2000	0.003
18000	2000	0.215
18	3400	0.000
180	3400	0.000
1800	3400	0.006
18000	3400	0.355

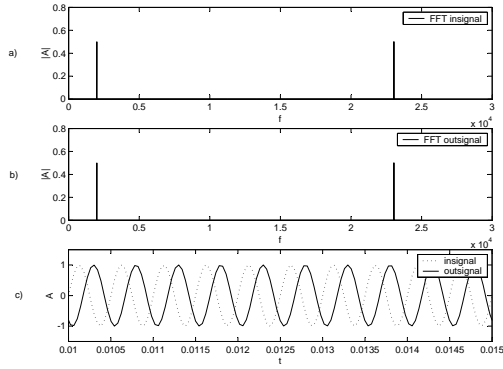
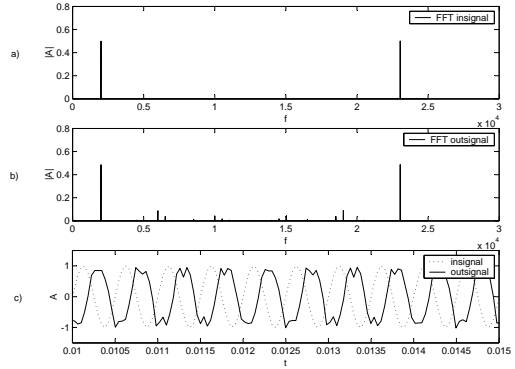
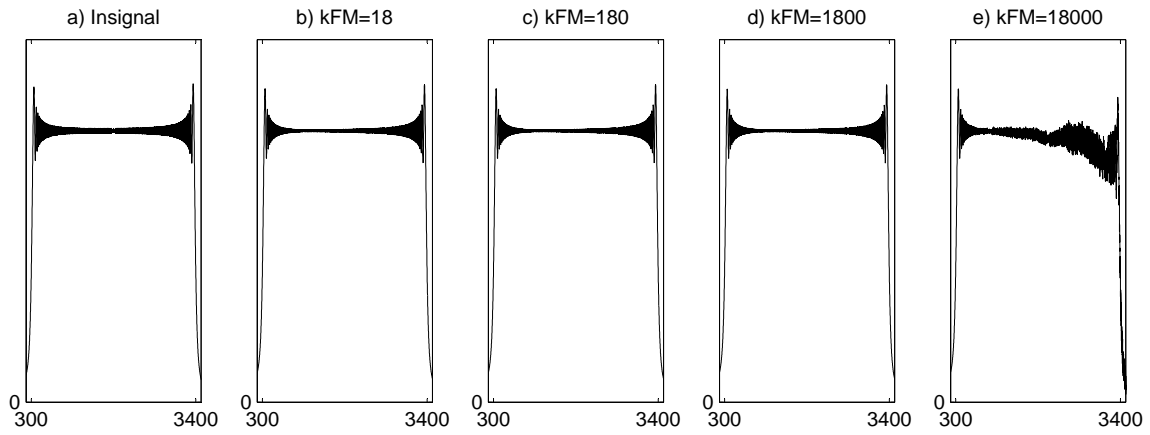

 Figure 4.11.: Plot of delay demodulator ideal with  $k_{FM} = 180$ 

 Figure 4.12.: Plot of delay demodulator ideal with  $k_{FM} = 18000$ 


Figure 4.13.: Spectrum of the chirp signal from a ideal delay demodulator

### 4.3.2. DSP Model

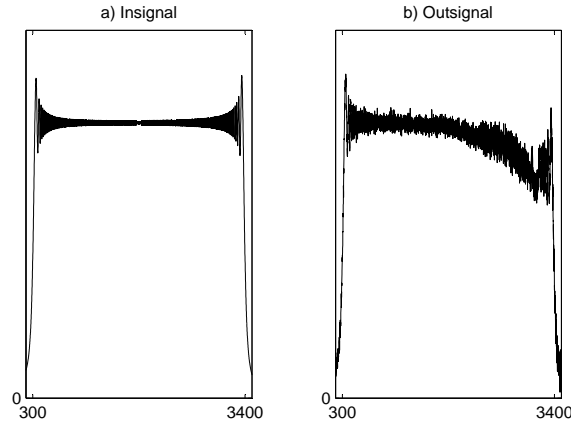
Figure E.2 in the Appendix shows the layout of the DSP simulation model. As described above, quantizer and saturation blocks were added. Also the cosine block was replaced by a lookup table with interpolation.

The SINAD was measured for various frequencies with a constant  $k_{FM} = 18000$ . Table 4.6 shows the results with and without a bandpass filter at the output.

**Table 4.6.:** SINAD delay demodulator DSP model

$f_N$	k (without filter)	k (with filter)
300	0.018	0.015
2000	0.218	0.011
3400	0.288	0.085

Again a chirp signal from 300 to 3400 Hz was applied to the input. Figure 4.14 shows (a) the spectrum of the input signal and (b) the spectrum of the output signal with a  $k_{FM}$  of 18000.



**Figure 4.14.:** Spectrum of chirp signal for delay demodulator DSP model

In the DSP model of the Delaydemodulator, a lower limit for  $\Delta F$  or rather  $k_{FM}$  has to be set. This is because of the subtraction in the derivation. If the slope of the signal is small, the result of the subtraction (*div*) is very small compared to the quantization interval  $q$ .

$$div = \phi_{FM}(n) - \phi_{FM}(n-1)$$

Hence, the signal quality degrades, since *div* can not be quantized sufficiently. With a bigger  $k_{FM}$ , the slope rises and so does *div*. The factor  $\frac{1}{T \cdot k_{FM}}$  can be interpreted as a metric for the degradation of the signal quality. A factor of unit would be the ideal. This means, with a given sample rate the  $k_{FM}$  should be equal to it. This is not possible because of the limitation in the bandwidth, but can be achieved approximately.

Another approach to enlarge *div* is to delay for  $k$  samples  $z^{-k}$  instead of just delay for one sample  $z^{-1}$ . This approach didn't lead to a usable solution, because the output signal was deformed by a  $\frac{\sin(x)}{x}$  function. This can be shown as followed. The transfer function of the integrator is

$$\frac{z^{-1} \cdot T}{1 - z^{-1}}$$

multiplied with the transfer function of the one sample delayed derivative is

$$\frac{z^{-1} \cdot T}{1 - z^{-1}} \cdot \frac{1 - z^{-1}}{T} = z^{-1}$$

The result is only a delay of one sample.

Now the integrator is multiplied with the k sample delayed derivative

$$\frac{z^{-1} \cdot T}{1 - z^{-1}} \cdot \frac{1 - z^{-k}}{T} = \frac{z^{-1} \cdot (1 - z^{-k})}{1 - z^{-1}}$$

The  $z^{-1}$  is only a delay. The rest is

$$\frac{1 - z^{-k}}{1 - z^{-1}}$$

This transfer function transformed to the time domain is a rectangle with the length k. Therefore a deformation of the spectrum with a  $\frac{\sin(x)}{x}$  results.

### 4.3.3. DSP Model with Gaussian Noise

Figure E.2 in the Appendix shows the layout of the simulation with gaussian noise.

Table 4.7 shows the signal to noise ratio (S/N) of the output signal. The power of the added noise was set, so the S/N ratio of the modulated signal is 10dB.

Table 4.7.: S/N delay demodulator DSP model

$f_N$	S/N [dB] $k_{FM} = 9000$	S/N [dB] $k_{FM} = 12000$	S/N [dB] $k_{FM} = 15000$	S/N [dB] $k_{FM} = 18000$
300	7.84	10.41	12.56	14.31
1000	7.69	10.36	12.31	13.94
2000	7.74	10.38	12.22	13.48
3400	6.99	8.99	10.13	10.68

It shows, that with a rising frequency the S/N degrades. This is because of the slight damping by the higher frequencies (see Figure 4.13), which lowers the power of the signal. Due to the smaller spreading of the FM spectrum, the sinking ratio with a smaller  $k_{FM}$  can be seen clearly.

Figure 4.15 shows the input signal of the demodulator with a sinusoidal signal of 300 Hz. Figure 4.16 shows a) the output signal in the same scale as the noise signal and b) the noise signal which was added. It shows that the noise was deformed and is rising with higher frequencies. This effect is due to the derivation, because higher frequencies are weighted more.

## 4.4. Phase-Adapter Demodulator

### 4.4.1. Ideal Model

Figure 4.17 shows the Simulink model of the ideal phase-adapter demodulator. Figure E.4 in the Appendix shows the layout for the simulation.

Table 4.8 shows the harmonic distortion factor  $k$  measured for various  $k_{FM}$  and frequencies of a sinusoidal message signal. Using Eq. 3.15 the maximal  $\Delta F$  and the maximal  $k_{FM}$  can be calculated.

$$k_{FM} = \frac{\pi^2 \cdot f_N}{\hat{s}} = \pi^2 \cdot f_N$$

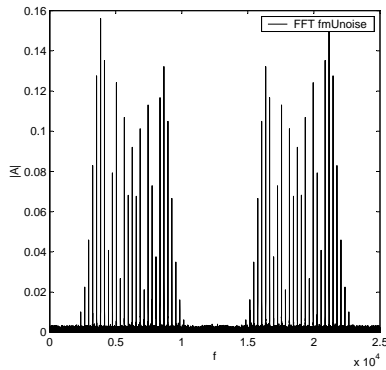


Figure 4.15.: Plot FM signal with noise

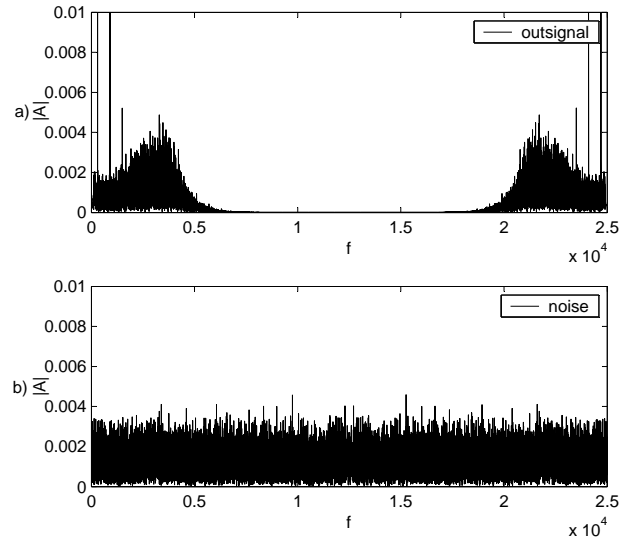


Figure 4.16.: Plot noise and output signal with noise

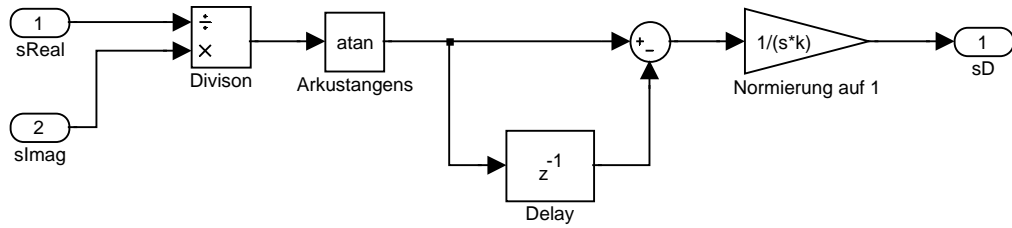


Figure 4.17.: Simulink model phase-adapter demodulator ideal

Table 4.8.: Harmonic distortion phase-adapter demodulator

$k_{FM}$	$\max(k_{FM})$	$f_N$	k
18	1480	300	0.000
180	1480	300	0.000
1800	1480	300	0.998
18000	1480	300	1.000
18	9870	2000	0.000
180	9870	2000	0.000
1800	9870	2000	0.003
18000	9870	2000	0.984
18	16780	3400	0.000
180	16780	3400	0.000
1800	16780	3400	0.006
18000	16780	3400	0.977

This equation is just valid for an ideal mathematical integrator, and so the sinusoidal signal would result in a cosine signal. The integrator of the FM Modulator has an initial condition of zero and so the integrated signal has an offset. Hence,  $k_{FM}$  has to be divided by two so that it is still limited

between  $-\frac{\pi}{2}$  to  $\frac{\pi}{2}$  after the arc tangent.

$$k_{FM} = \frac{\pi^2 \cdot f_N}{2}$$

As soon as this value is exceeded, the harmonic distortions raise above one and the demodulator can not be used anymore. Figures 4.18 and Figure 4.19 illustrate this. In both situations a sinusoidal signal of 2000 Hz was modulated and demodulated. One with  $k_{FM} = 180$  and another with  $k_{FM} = 18000$ . In both figures (a) shows the spectrum of the input signal and (b) shows the output signal, and (c) shows a time slot with input and output signals. This confirms the statement that

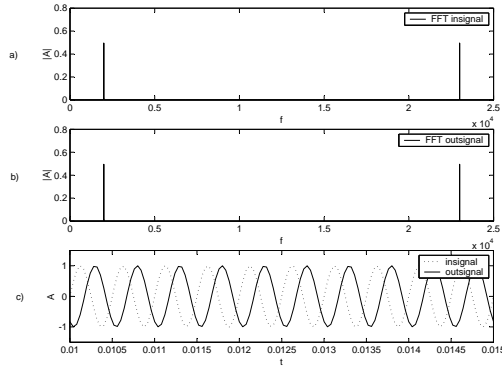


Figure 4.18.: Plot phase-adapter demodulator with  $k_{FM} = 180$

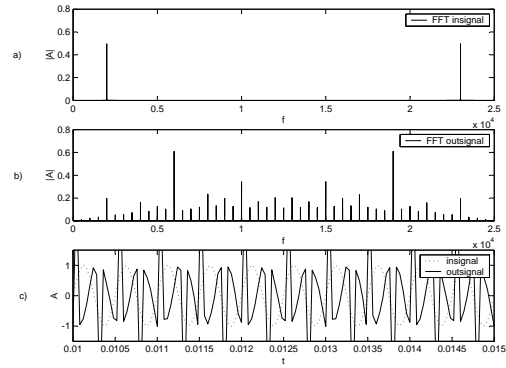


Figure 4.19.: Plot phase-adapter demodulator with  $k_{FM} = 18000$

the Phase-Adapter-Demodulator is only useful for narrowband FM. Therefore it does not meet the specifications and will not be discussed further.

## 4.5. Phase-Locked Loop

The loop gain  $P$  is the only unknown to be determined. In Section 3.3.3 it was described for small  $k_{FM}$  the loop gain  $P$  has to be set as big as possible. In this chapter  $k_{FM}$  was set to 18000 due to the Carson bandwidth. Criteria Eq. 3.18, which discusses the error  $e(n)$  between  $s_N(n)$  and  $s_D(n)$ , is reviewed here.

$$e(n) = s_N(n) - s_D(n)$$

$$\sum_{i=0}^{n-1} s_N(i) - s_D(i) = \frac{\arcsin\left(\frac{s_D(n) \cdot 2}{A \cdot P}\right)}{k_{FM}}$$

The best way to undo the summation on the left hand side is to change the expression to the Z-domain. In general:

$$\begin{aligned} a(n) &= \sum_{i=0}^{n-1} b(i) \leftrightarrow A(z) = \frac{z}{z-1} \cdot B(z) \\ B(z) &= \frac{z-1}{z} \cdot A(z) \leftrightarrow b(n) = a(n) - a(n-1) \end{aligned}$$

Substituted then into Eq. 3.18:

$$e(n) = s_N(n) - s_D(n) = \frac{\arcsin\left(\frac{s_D(n) \cdot 2}{A \cdot P}\right)}{k_{FM}} - \frac{\arcsin\left(\frac{s_D(n-1) \cdot 2}{A \cdot P}\right)}{k_{FM}}$$

The result of the arc tangent is between  $-\pi/2$  and  $\pi/2$ . So the maximal error  $e(n)$  is:

$$|e(n)_{\max}| = \frac{\pi}{k_{FM}} = 1.7453 \cdot 10^{-4}$$

Due to the normalization to one, the error is 0.017453 % of the signal amplitude. There is still the question of an appropriate loop gain  $P$ . It can be calculated with the help of Eq. 3.17:

$$s_D(n) = \frac{A \cdot P}{2} \cdot \sin(k_{FM} \cdot \sum_{i=0}^{n-1} s_N(i) - k_{FM} \cdot \sum_{i=0}^{n-1} s_D(i)) = \frac{A \cdot P}{2} \cdot \sin(\sum_{i=0}^{n-1} k_{FM} \cdot s_N(i) - k_{FM} \cdot s_D(i))$$

$$|s_D(n)_{\max}| = \frac{A \cdot P}{2} = 1$$

Because of the normalization,  $A$  and  $K$  (see Section 3.1) and the maximal value of  $s_D(n)$  are equal to one. Thus  $P$  is

$$P = 2$$

But there is a problem here. The amplitude of the sinusoidal signal in Eq. 3.17 is exactly one only if the following condition is true:

$$\Delta F = \frac{k_{FM}}{2 \cdot \pi} = \omega_N \quad (4.2)$$

This condition is analyzed in the time domain. The outputs of the mixer are:

$$\frac{A}{2} \cdot \cos(k_{FM} \cdot \int s_N(\omega_N \cdot t) \cdot dt) = \frac{A}{2} \cdot \cos(\frac{k_{FM}}{\omega_N} \cdot S_N(\omega_N \cdot t)) \quad (4.3)$$

and

$$\frac{A}{2} \cdot \sin(k_{FM} \cdot \int s_N(\omega_N \cdot t) \cdot dt) = \frac{A}{2} \cdot \sin(\frac{k_{FM}}{\omega_N} \cdot S_N(\omega_N \cdot t)) \quad (4.4)$$

$S_N(\omega_N \cdot t)$  is the antiderivative of  $s_N(\omega_N \cdot t)$  and  $\omega_N$  is the momentary frequency. So Eq. 4.2 can be written as:

$$k_{FM} = 2 \cdot \pi \cdot \omega_N$$

Substitute the result in the Eq. 4.3 and Eq. 4.4

$$\frac{A}{2} \cdot \cos(2 \cdot \pi \cdot S_N(\omega_N \cdot t))$$

and

$$\frac{A}{2} \cdot \sin(2 \cdot \pi \cdot S_N(\omega_N \cdot t))$$

It shows that in this case the argument of the cosine or the sine is between zero and  $2 \cdot \pi$ . If the range of the argument is smaller, the amplitude of the sine function in Eq. 3.17 is smaller than one. So the chosen loop gain of  $P = 2$  is not sufficient to amplify the amplitude of the demodulated signal to one. In this case the demodulated signal is just proportional but not equal to the original message.

#### 4.5.1. Ideal Model

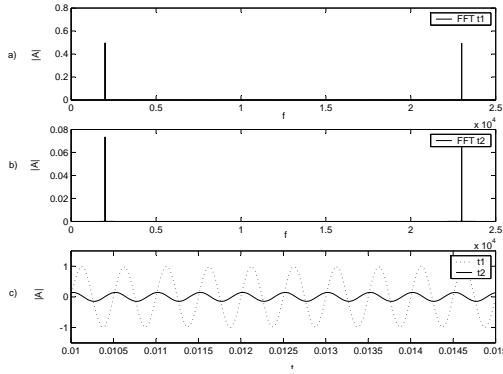
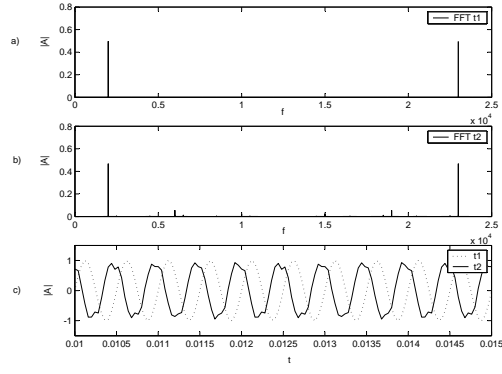
Figure E.5 in the Appendix shows the layout of the ideal simulation model in Simulink. Table 4.9 shows the harmonic distortion factor  $k$  of the demodulated signal for various frequencies and  $k_{FM}$  of a sinusoidal message.

Figure 4.20 and Figure 4.21 confirm the measurements of the harmonic distortion. a) shows the spectrum of the message  $s_N(n)$ . It is a sinusoidal signal with a frequency of  $f_N = 2000\text{Hz}$ . (b)



**Table 4.9.: Harmonic distortion PLL**

$k_{FM}$	$f_N$	k
18	300	0.167
180	300	0.002
1800	300	0.022
18000	300	0.046
18	2000	0.162
180	2000	0.002
1800	2000	0.000
18000	2000	0.144
18	3400	0.152
180	3400	0.002
1800	3400	0.003
18000	3400	0.254

**Figure 4.20.: Plot PLL ideal with  $k_{FM} = 1800$** **Figure 4.21.: Plot PLL ideal with  $k_{FM} = 18000$** 

shows the spectrum of the demodulated signal  $s_D(n)$ , and (c) shows a time slot of  $s_N(n)$  and  $s_D(n)$ . In Figure 4.21 (b) the harmonic distortion is worsen at frequencies over 5000 Hz. Figure 4.22 shows (a) the spectrum of the message signal and (b), (c), and (d) the spectra of the demodulated signals for various  $k_{FM}$ . The message signal is a chirp signal from  $f_N = 300\text{Hz}$  to  $f_N = 3400\text{Hz}$ .

#### 4.5.2. DSP Model

Figure E.6 in the Appendix shows the layout of the DSP simulation. As mentioned earlier quantizer and saturation blocks were added. Table 4.10 shows the results of the SINAD measurements. It was measured with and without a bandpass filter at the output. The filter improves the quality of the demodulated signal.

Figures 4.23 to 4.27 illustrate the measurements of the SINAD. The same problem with additional high frequencies generated by a higher  $k_{FM}$  appears. This problem is reduced by the filter. Figure 4.27 shows the demodulated signal  $s_D(n)$  with a bandpass in the feedback loop. Figure 4.27 (b) shows that a lot of additional frequencies appear. Hence a filter in the loop doesn't bring any benefits. On contrary it worsens the quality.

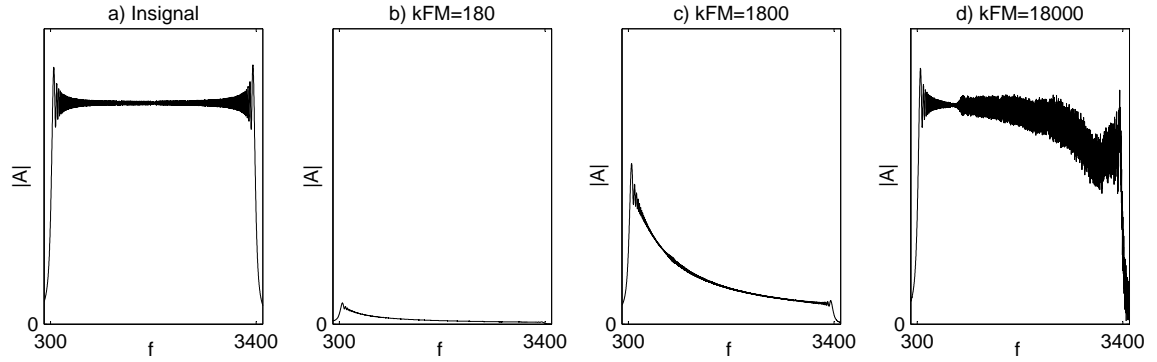
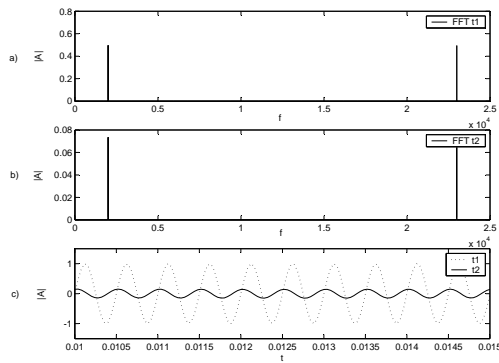
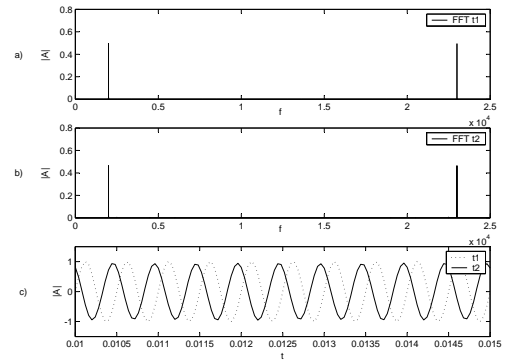


Figure 4.22.: Spectrum of the chirp signal for the ideal PLL

Table 4.10.: SINAD PLL

$k_{FM}$	$f_N$	k with filter	k without filter
18	300	0.018	0.168
180	300	0.018	0.021
1800	300	0.025	0.036
18000	300	0.088	0.198
18	2000	0.041	0.195
180	2000	0.043	0.114
1800	2000	0.042	0.115
18000	2000	0.080	0.247
18	3400	0.076	0.218
180	3400	0.078	0.186
1800	3400	0.078	0.186
18000	3400	0.080	0.396

Figure 4.23.: PLL DSP  $k_{FM} = 1800$ , with bandpass filterFigure 4.24.: PLL DSP  $k_{FM} = 18000$ , with bandpass filter

### 4.5.3. DSP Model with Gaussian Noise

Table 4.11 shows the S/N ratios of the PLL with various  $k_{FM}$ . It was stated above that a better signal quality is achieved with a smaller  $k_{FM}$ , the degradation wasn't serious though. This model

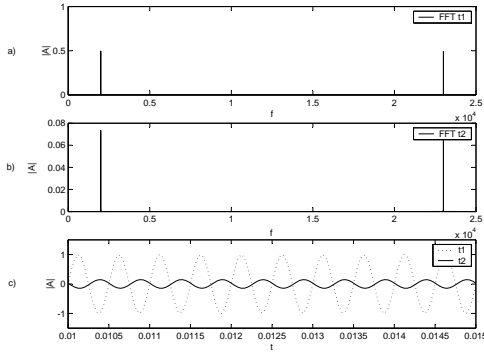


Figure 4.25.: PLL DSP  $k_{FM} = 1800$ , without bandpass filter

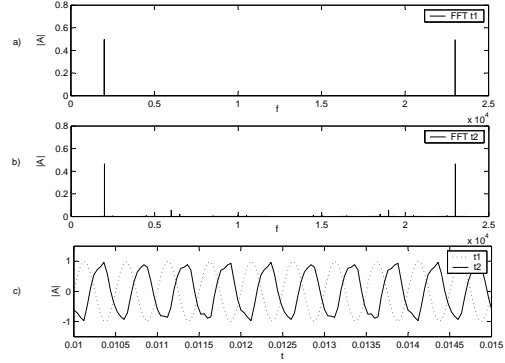


Figure 4.26.: PLL DSP  $k_{FM} = 18000$ , without bandpass filter

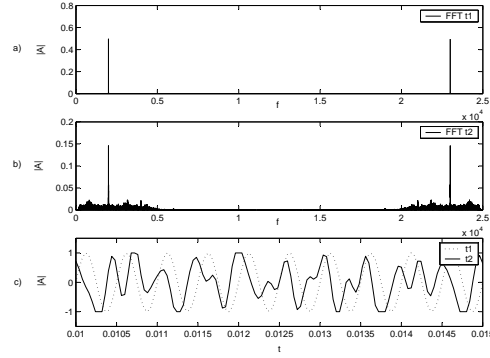


Figure 4.27.: PLL DSP  $k_{FM} = 18000$ , with the bandpass filter in the feedback loop

Table 4.11.: S/N PLL

$f_N$	S/N [dB] $k_{FM} = 9000$	S/N [dB] $k_{FM} = 12000$	S/N [dB] $k_{FM} = 15000$	S/N [dB] $k_{FM} = 18000$
300	11.90	10.39	8.754	7.45
1000	10.71	11.73	12.29	12.02
2000	8.22	10.24	11.66	12.62
3400	5.23	7.70	9.44	10.59

shows that with a bigger  $k_{FM}$ , an improvement in the S/N ratio can be achieved. The specified  $k_{FM} = 18000$  will achieve the best results.

## 4.6. Mixed Demodulator

### 4.6.1. Ideal Model

Figure 4.28 shows the Simulink model of the Mix Demodulator. Figure E.4 in the Appendix shows the layout of the Simulink simulation.

Table 4.12 shows the harmonic distortion factor  $k$ , measured for various  $k_{FM}$  and various frequen-

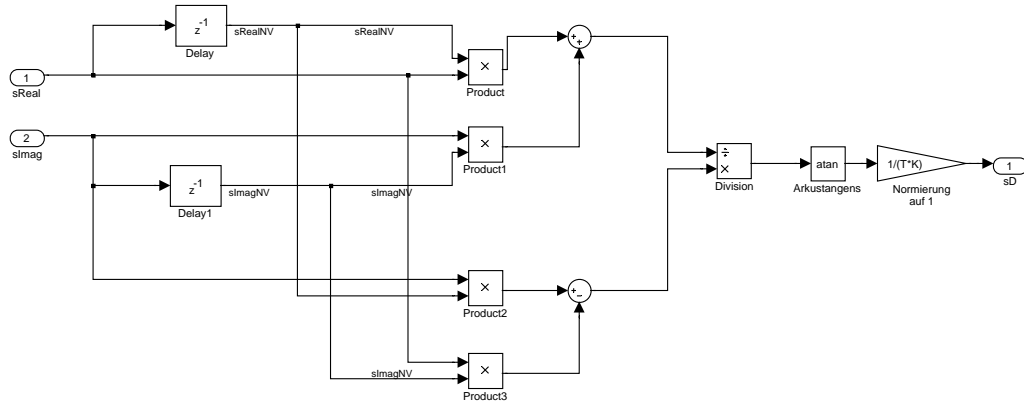


Figure 4.28.: Simulink model mixed demodulator ideal

cies of a sinusoidal message signal.

Table 4.12.: Harmonic distortion factor mixed demodulator ideal

$k_{FM}$	$f_N$	$k$
18	300	0.000
180	300	0.000
1800	300	0.000
18000	300	0.012
18	2000	0.000
180	2000	0.000
1800	2000	0.003
18000	2000	0.217
18	3400	0.000
180	3400	0.000
1800	3400	0.006
18000	3400	0.353

Like the delay demodulator the maximal  $\Delta F$  can be calculated by Eq. 3.13. From which the maximal  $k_{FM}$  can be determined.

$$k_{FM} = \frac{2 \cdot \pi \cdot f_A}{4\hat{s}} = \frac{\pi \cdot f_A}{2} = 39'270$$

Due to the bandwidth limitation, this  $k_{FM}$  is not reachable. The degradation of the harmonic distortion with rising  $k_{FM}$  and higher frequencies is again the result of the starting aliasing.

Figure 4.29 and Figure 4.30 show (a) the spectrum of the input signal `insignal`, (b) the spectrum of the output signal `outsignal` and (c) a time slot of the input and output signals. The input signal is a sinusoidal signal with a frequency of 2000 Hz. In Figure 4.29, a  $k_{FM}$  of 180 was applied. In Figure 4.30 the  $k_{FM}$  is 18000. It shows again that a low-pass filter with  $f_g \approx 4000\text{ Hz}$  at the output could improve the signal quality under a high  $k_{FM}$ .

Again a chirp signal from 300 to 3400 Hz was applied to the input. Figure 4.31 shows (a) the spectrum of the input signal and (b) to (e) the spectra of the output signals with various  $k_{FM}$ . It can be seen that the signal transmission worsens with higher frequencies, again because of the aliasing.

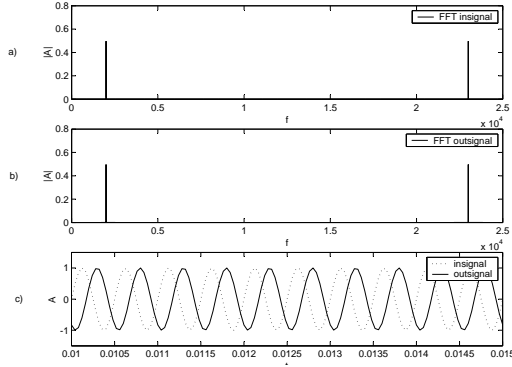


Figure 4.29.: Plot of ideal mixed demodulator with  $k_{FM} = 180$

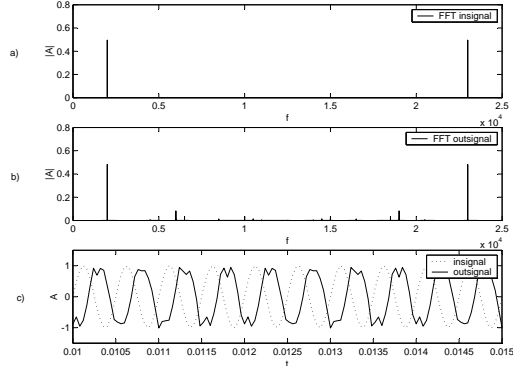


Figure 4.30.: Plot of ideal mixed demodulator with  $k_{FM} = 18000$

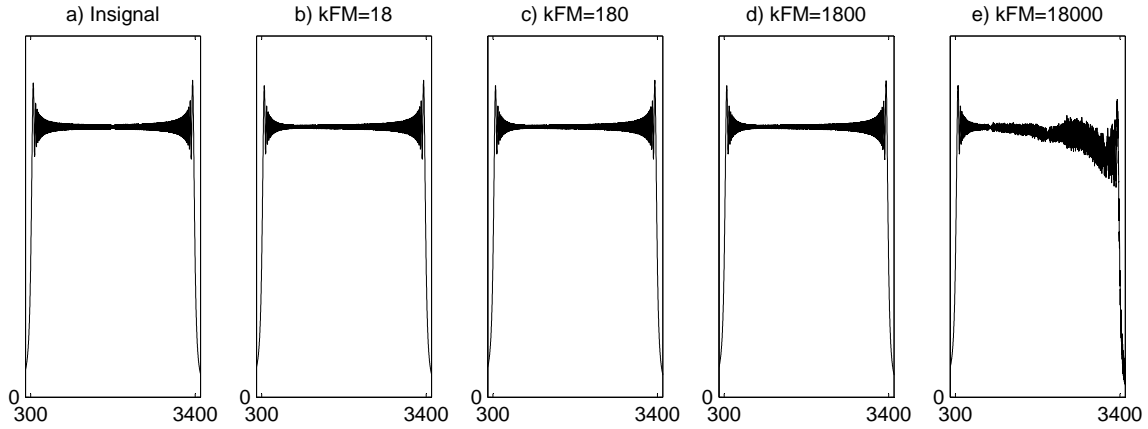


Figure 4.31.: Spectrum of the chirp signal for an ideal mixed demodulator

#### 4.6.2. DSP Model

Figure E.9 in the Appendix shows the layout of the Simulink simulation. As mentioned before, quantizer and saturation blocks were added. In addition the arc tangent block was replaced by a lookup table with interpolation.

The SINAD was measured again for various frequencies and a constant  $k_{FM} = 18000$ . Table 4.13 shows the results with and without a bandpass filter at the output.

Table 4.13.: SINAD DSP mixed demodulator

$f_N$	k (without filter)	k (with filter)
300	0.013	0.012
2000	0.216	0.008
3400	0.286	0.087

Again a chirp signal from 300 to 3400 Hz was applied to input. Figure 4.32 shows (a) the spectrum of the input signal and (b) the spectrum of the output signal with a  $k_{FM}$  of 18000.

For the mixed demodulator, the lower limit for  $k_{FM}$  is given as by the delay demodulator described in Section 4.3.2.

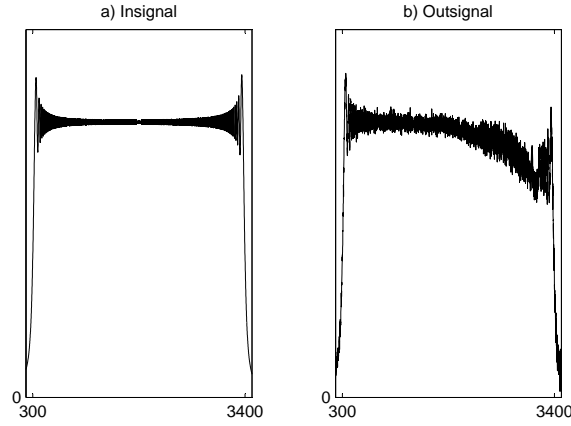


Figure 4.32.: Spectrum of the chirp signal for DSP mixed demodulator

### 4.6.3. DSP Model with Gaussian Noise

Figure E.9 in the Appendix shows the layout for the Simulink simulation of the DSP model with gaussian noise.

Table 4.14 shows the S/N ratio at the output of the demodulator. The power of the added noise was set, so the S/N ratio of the modulated signal is 10dB.

Table 4.14.: S/N DSP Mix Demodulator

$f_N$	S/N [dB] $k_{FM} = 9000$	S/N [dB] $k_{FM} = 12000$	S/N [dB] $k_{FM} = 15000$	S/N [dB] $k_{FM} = 18000$
300	7.83	10.41	12.44	13.71
1000	7.70	10.33	12.24	13.43
2000	7.77	10.39	12.10	12.64
3400	6.97	8.81	9.41	8.40

Figure 4.33 shows the noisy FM signal, which is applied to the demodulator. The message is a sinusoidal signal with a frequency of  $f_N = 300\text{Hz}$ . Figure 4.34 shows (a) the noise signal which was added, (b) the output signal with the same scale as the noise. The deformation of the noise due to the derivation was observed evidently.

## 4.7. Comparison of the Algorithms

In this section the algorithms will be compared to each other concerning signal quality, robustness, computing power and storage utilization. The phase-adapter demodulator will not be included in the comparison, because it does not meet the specifications.

### 4.7.1. Signal Quality

The comparison regarding the signal quality is done on the basis of the harmonic distortion and SINAD. The delay demodulator and the mixed demodulator show nearly identical values for the ideal and the DSP models, due to their similarity. The harmonic distortion rises with rising frequencies or rising  $k_{FM}$ , which is, due to rising bandwidth and aliasing. The PLL also shows a

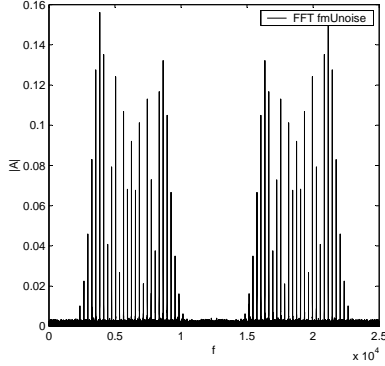


Figure 4.33.: Plot of FM signal with noise

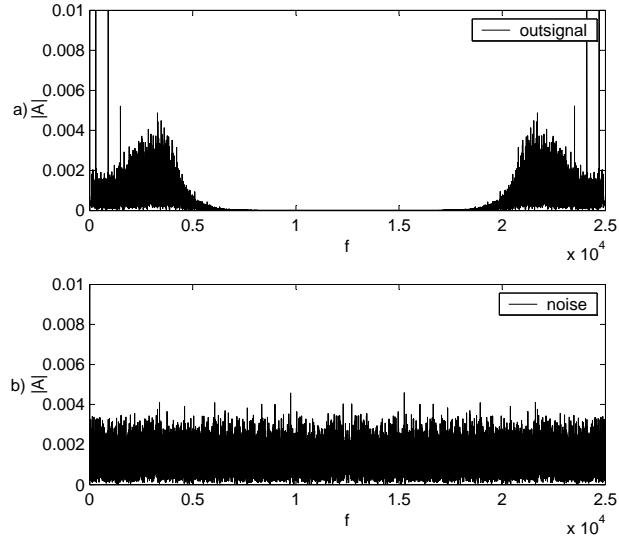


Figure 4.34.: Plot of the noise and the output signal

similar behavior, with a small difference that the harmonic distortion is rising for a very small  $k_{FM}$ . In general, the values of  $k$  for all the algorithms are fairly close.

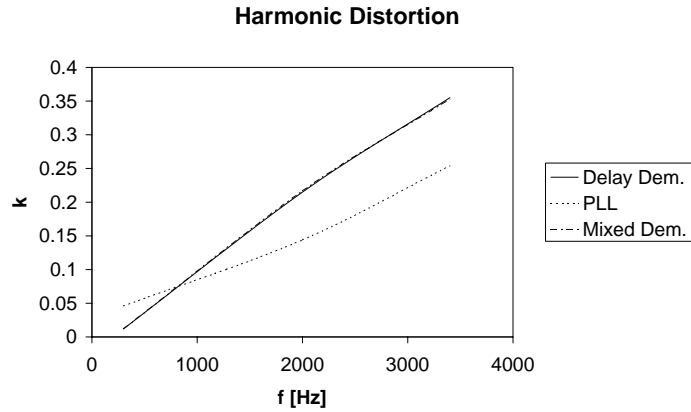


Figure 4.35.: Comparison of the harmonic distortion for ideal models of different algorithms with constant  $k_{FM}$  of 18000 and variable frequency

Another performance characteristic can be compared by applying a chirp signal. Again the delay demodulator and the mixed demodulator show nearly the same behavior. The PLL has a slightly worse frequency response.

#### 4.7.2. Robustness

The robustness measure gives the information on how the demodulator acts with noise on the channel. The signal to noise ratio (S/N) of the demodulated signal provides an insight into the robustness. Therefore, a Gaussian noise was added to the FM-Signal before applying to the demodulator and the ratio was measured at the output. In this measurement the delay demodulator

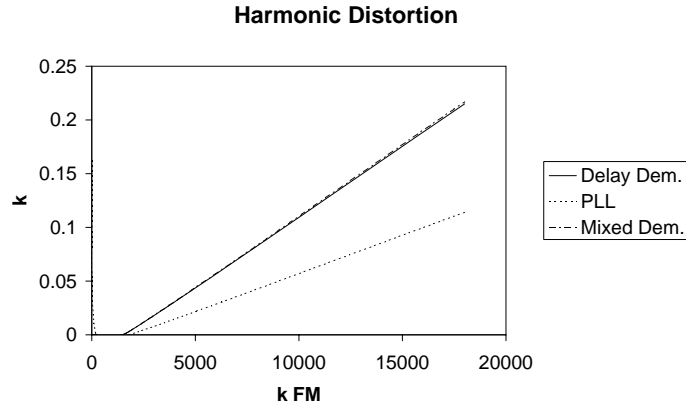


Figure 4.36.: Comparison of harmonic distortion for ideal models of different algorithms with constant frequency of 2000Hz and variable  $k_{FM}$

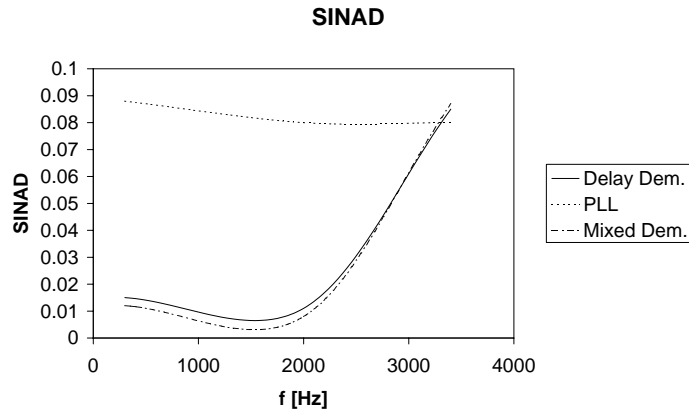


Figure 4.37.: Comparison of SINAD for DSP models of different algorithms with constant  $k_{FM}$  of 18000 and variable frequency

and the mixed demodulator show a similar behavior, with a slightly better result obtained by the delay demodulator. The S/N drops in both demodulators with an sinking  $k_{FM}$  owing to sinking frequency spreading. With higher frequencies, the ratio sinks a little, because of the sinking frequency response at high frequencies. The PLL shows a different behavior for low frequencies. The ratio sinks not just for high frequencies but also for the low ones.

### 4.7.3. Computing Power and Storage Utilization

In this section no absolute values will be determined. Just a cross comparison is done.

All the algorithms need a mixer so this block will be left out for the comparison. First the delay demodulator and the mixed demodulator will be compared. They both use about the same amount of additions and multiplications. Both of them need a division and one arc table function. Also the delay register is of the same size. However, the delay demodulator needs an additional table for the root function in the amplitude normalization. Hence it needs a little bit more storage and computing power. The PLL needs the least computing power as no division is required. Besides, the number of additions and multiplications is smaller. The trigonometric functions sine and cosine can be realized with just one table. Hence the storage requirement is about the same as that of the



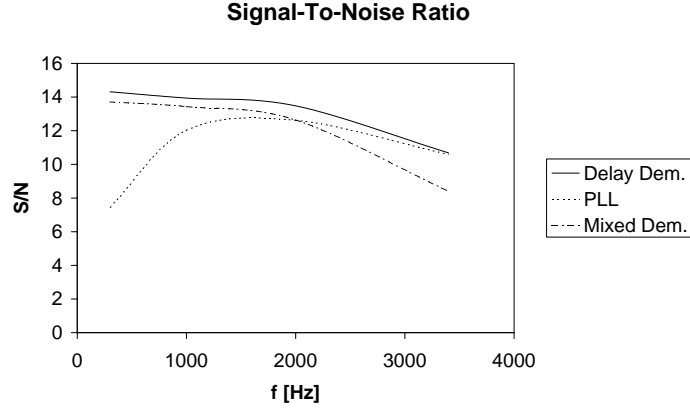


Figure 4.38.: Comparison of S/N ratio for different demodulation algorithmen, with a  $k_{FM}$  of 18000

mixed demodulator. So the following ranking can be obtained.

Computing power:

$$PLL < MixedDemodulator \approx DelayDemodulator$$

Storage utilization:

$$PLL \approx MixedDemodulator < DelayDemodulator$$

#### 4.7.4. The Appropriate Algorithm

Out of this knowledge the mixed demodulator seems to be the most suitable algorithm. Its realization needs less storage than that of the delay demodulator to achieve nearly the same signal quality and robustness. The PLL requires less resources, but its robustness is worse for low frequencies. Hence, the mixed demodulator is the first choice for the implementation. The PLL is the second because of low computing power.



## 5. Implementation

The mixed demodulator and the PLL demodulator are implemented on a Texas Instrument TMS320C6711 digital signal processor. A DSP starter kit is used which includes a TMS320C6711DSK board and the Code Composer Studio (CCS) IDE. The CCS is a complete IDE featuring code building, debugging and real time analysis. It is especially designed for Texas Instrument DSP's. Also included are DSP/BIOS a Chip Support Library (CSL) and a Board Support Library (BSL). It supports implementation in C++ , C and Assembler.

The algorithms are implemented in C. The first approach is implemented in floating-point to avoid problems with overflows and quantization. The code sequences are kept simple without any optimization or time effective programming. The transformation of the algorithm to C code at this stage aims at only verifying its functionality on the targeted DSP.

In the next step the floating-point implementation is transformed into a fixed-point with still no great effort in speed optimization. The problems of overflows and quantization are discussed. Thereafter, the C implementation is optimized in speed. Finally some of time intensive parts are written in linear assembler to decrease the computing load.

### 5.1. System Architecture

#### 5.1.1. Signal Path

Figure 5.1 outlines the floating-point implementation of the demodulator. It shows the sampling rates and the buffer lengths for the different stages of the demodulation. The entry end exit points of the functions are marked and the used buffers are shown. For specific buffers a sketch of the signal's spectrum is shown.

Figure 5.2 shows the same for the fixed-point implementation.

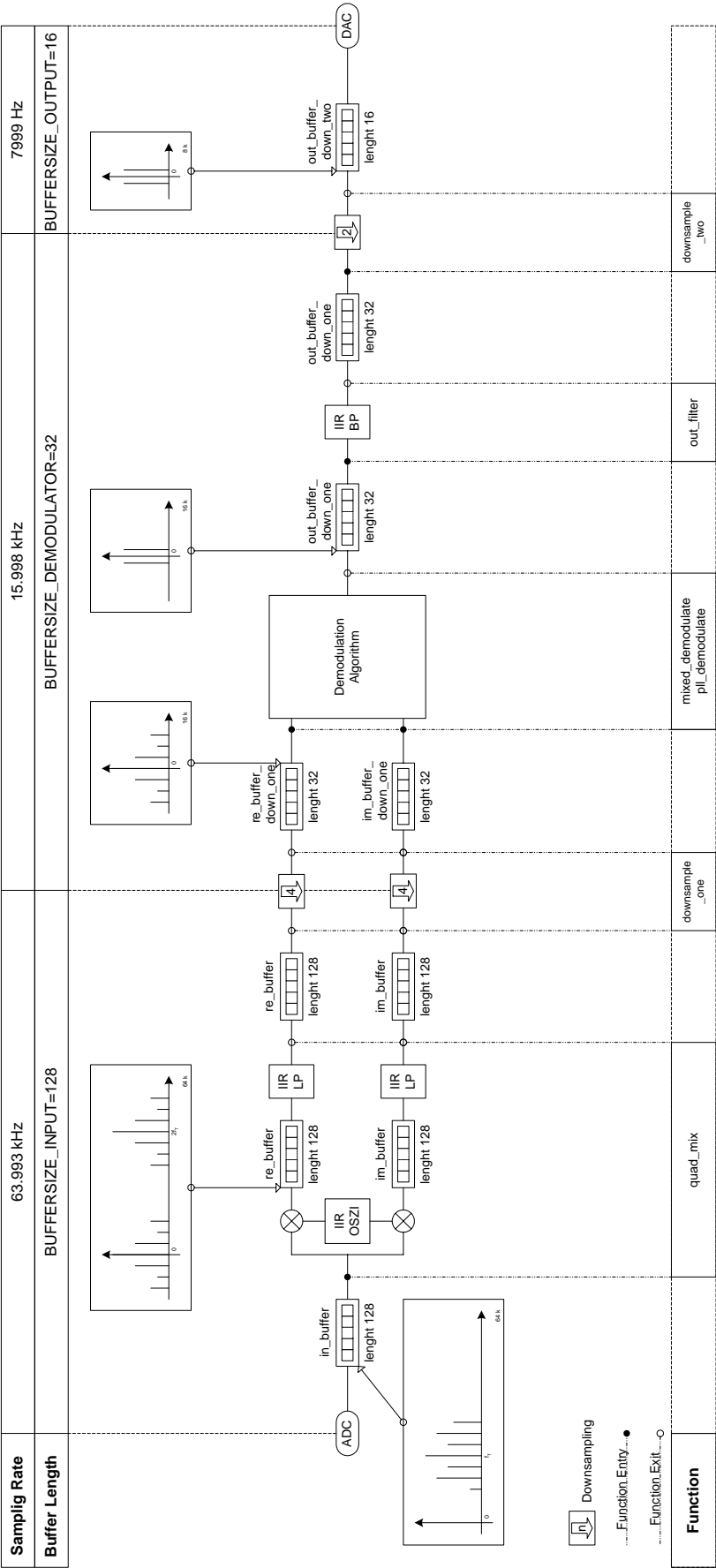


Figure 5.1.: Overview of floating-point implementation

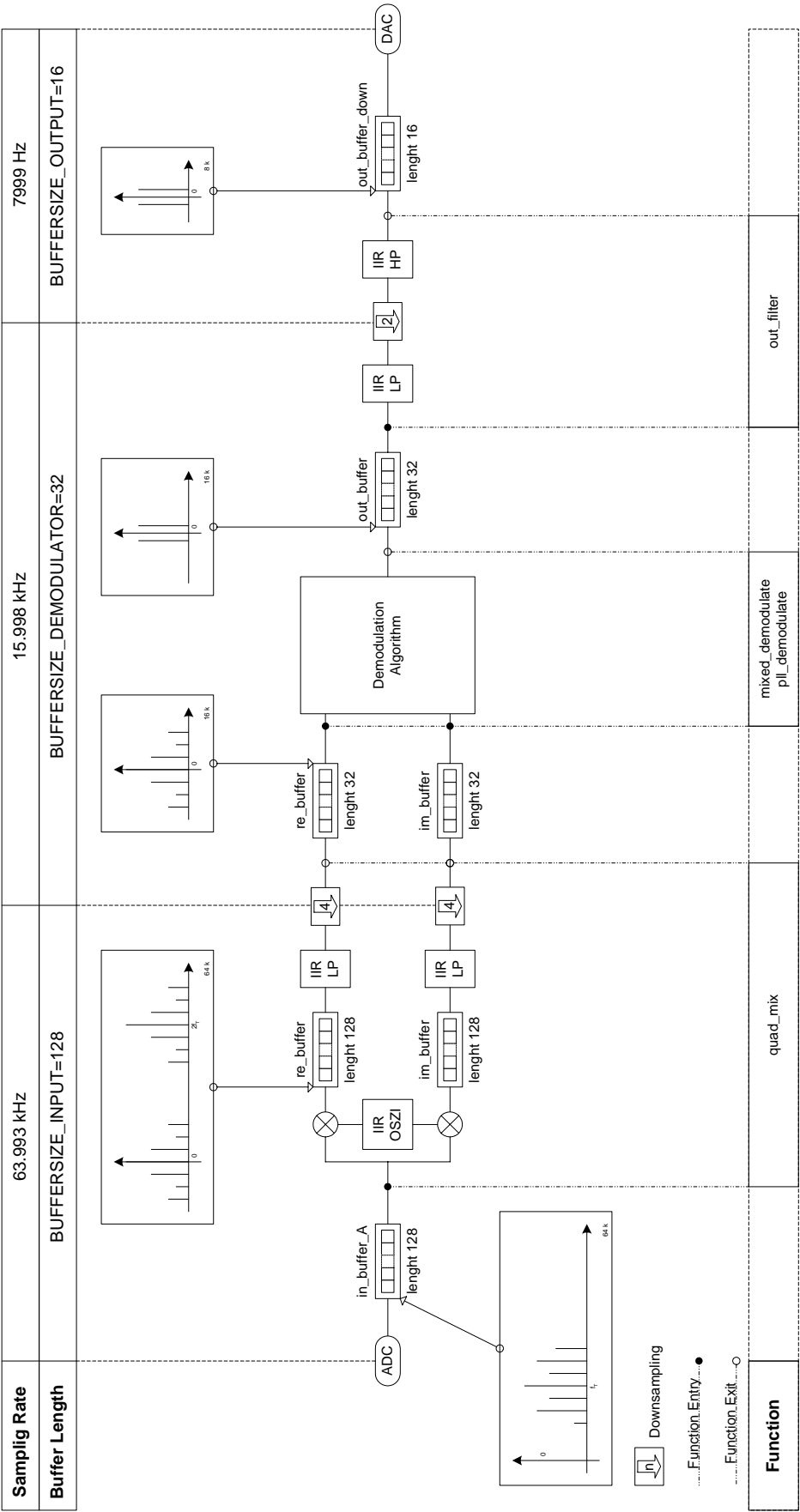


Figure 5.2.: Overview of fixed-point implementation

### 5.1.2. The Sampling Rates

In Section 4.4 a sampling rate of 25043.89Hz was calculated according to Eq. 3.6. This would be the ideal sampling frequency for the subsampling, but the hardware sets the following limitations:

**Analog Digital (Converter THS1408)** The clock of the ADC is generated by the Timer inside the DSP. Therefore only sampling rates of  $f_A = \frac{150MHz}{8 \cdot i}$  are possible, where  $i$  is an integer.

**Digital Analog (Converter On Board Codec)** The on board codec has an unchangeable sampling rate of 8000Hz. Thus the sampling rate of the ADC has to be a multiple of 8000Hz for the downsampling.

In addition, the conditions of the subsampling need to be fulfilled by the sample rate of the ADC. With the hardware setup of the DSP DSK and the ADC EVM it is not possible to meet all these conditions because it is not possible to run the ADC with a multiple frequency of the DAC frequency. The sample rate which was chosen is

$$f_A = \frac{150MHz}{8 \cdot 293} = 63.993kHz$$

it meets all conditions except the one for the downsampling. During the processing the signal will be totally downsampled by 8 and therefore the sampling frequency of the output signal is:

$$f_A = \frac{150000000}{8 \cdot 293 \cdot 8} = \frac{63993.17}{8} = 7999.15Hz$$

Therefore the output samples arrive too early (Figure 5.3), and the output signal appears to be

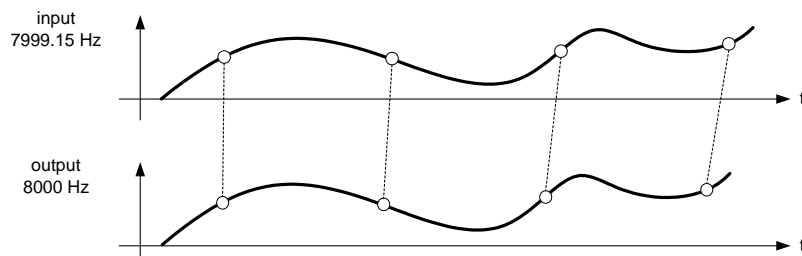


Figure 5.3.: Input and output samples

slightly faster. This is not a problem because the difference is smaller than 1 Hz. The problem is that about every second there is a missing sample. Therefore a value between the last output and the next to come is interpolated for the missing sample. Due to the phase shift of about one sampling period introduced by the additional sample, a low short rustle in the output signal can be heard every time the additional sample is issued.

This problem is introduced by the hardware and could be solved by using one master clock from which both the ADC clock and the DAC clock are derived so that both ADC and DAC operations are synchronized. An attempt was made to realize this hardware solution with the DSK by wiring the codec clock (4.096 MHz) to the input pin of the DSP timer. Unfortunately the clock was not able to drive both the codec and the timer input. The effort of adding a driver was not made.

Because the noise introduced by this error is very small and as audible for just pure sinusoidal signals and not for speech, no further efforts were done to reduce it. In a commercial implementation the solution with a master clock would eliminate this problem completely.

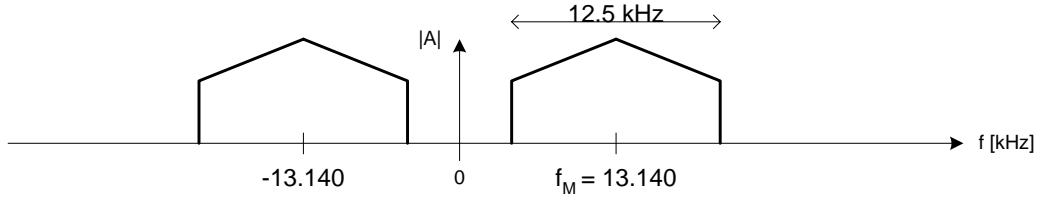


Figure 5.4.: Spectrum of FM signal after subsampling

The spectrum of the FM signal after the downsampling is shown in Figure 5.4. The new carrier frequency  $f_M$  can be calculated with equation

$$f_M = f_T - f_A \cdot \text{floor}\left(\frac{f_T}{f_A}\right) = 10.7 \cdot 10^6 - 63993 \cdot \text{floor}\left(\frac{10.7 \cdot 10^6}{63993}\right) = 13.140 \text{ Hz} \quad (5.1)$$

where  $f_T$  is the carrier frequency of the analog FM signal.

### 5.1.3. Analog-to-Digital Conversation THS1408

The on board codec of the DSP DSK has a fix sample rate of 8000 Hz. Therefore an additional ADC (THS1408 EVM) is connected to the daughtercard connectors of the DSP. The data from the ADC is transferred by the Enhanced Direct Memory Access unit (EDMA) to reduce the workload of the DSP.

The implementation to setup the ADC and the EDMA is done in the two files *adc\_THS1408.h* (Listing F.1) and *adc\_THS1408.c* (Listing F.2). It uses the EMIF, EDMA, TIMER, and IRQ modules of the Chip Support Library (CSL) to provide an interface to control and configure the on chip peripherals of the DSP.

#### THS1408

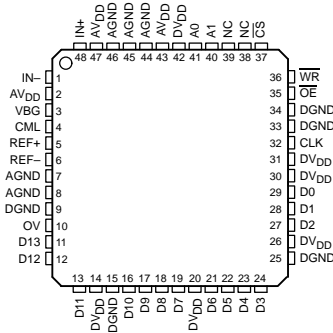


Figure 5.5.: THS1408

The THS1408 is a 14-bit, 8MSPS single supply analog-to-digital converter with an internal reference, differential inputs, programmable input gain and on-chip sample and hold amplifier. For more detailed information refer to [11] and [12]. The jumper settings of the EVM are summarized in Table 5.1. The Power to the EVM is supplied from the DSP DSK (jumper H14). The input is

Table 5.1.: Jumper settings THS1408

Jumper	Setting	Description	Jumper	Setting	Description
H1	-	unused	H9	2-3	input
H2	-	unused	H10	-	unused
H3	-	clock	H11	-	unused
H4	-	unused	H12	-	hardware loopback
H5	-	clock	H13	-	hardware loopback
H6	1-2	clock	H14	1-2	power
H7	2-3	input	H15	2-3	board
H8	1-2	input			

applied as a single ended input (jumpers H7 to H9). The other jumper settings are explained in the following sections.

### Address Mapping

The ADC EVM is connected to the DSP DSK via the daughtercard connectors. The device contains several registers. A register is selected by the values of the A1 and A0 in Table 5.2. Figure 5.6

Table 5.2.: Addresses of registers THS1408

A1	A0	Register	Abbreviation	HEX Address
0	0	Conversation result	RES	0xA000'0040
0	1	PGA (Gain)	PGA	0xA000'0044
1	0	Offset	OFF	0xA000'0048
1	1	Control	CTL	0xA000'004C

shows how the address bus is wired to the EVM and the ADC, with the jumper H15 set to 2-3. The

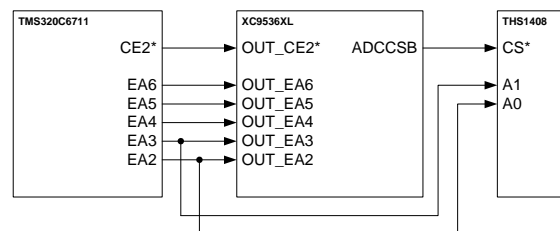


Figure 5.6.: Schema address bus wiring THS1408

EVM uses a Xilinx XC9536 CPLD to perform the address decode and control. Table 5.3 is the truth table of the address decode logic.

The CE2 signal is a EMIF memory space enable signal. On the C6711 it defines the memory block from address 0xA000'0000 with the length of 256MB [14]. With this information the addresses of the THS1408 registers can be determined (Table 5.2).

The off chip memory is accessed through the EMIF, therefore the values for the CE2CTL register of the EMIF have to be set according to the specifications of the THS1408 ADC (Tabel 5.4 [12] [16]).



Table 5.3.: Truth table address decode THS1408

OUT_EA6	OUT_EA5	OUT_EA4	OUT_EA3	OUT_EA2	OUT_CE*	DACCSB
1	0	0	0	0	0	0
1	0	0	0	1	0	0
1	0	0	1	0	0	0
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	0	1	0	1	0	1
1	1	0	0	0	0	0
All other combinations						1

Table 5.4.: EMIF timing registers for THS1408

Field	Abbreviation	Value
Read Setup	RDSETUP	1
Read strobe	RDSTRB	3
Read hold	RDHLD	1
Write setup	WRSETUP	4
Write strobe	WRHLD	6
Write hold	WRSTRB	3
Memory type	MTYPE	32-bit-wide asynchronous interface

### Conversion Rate

The clock for the ADC is generated with the timer 0 of the DSP. Jumpers H3, H5, and H6 of the EVM have to be set according to Table 5.1, to connect the timer output to the clock input of the ADC. The timer 0 is clocked by the internal source of  $CPU\_Rate/4 = \frac{f_{CPU}}{4}$ .

$$f(clock\_source) = \frac{f_{CPU}}{4}$$

The output is set as clock output, the frequency is

$$f_{clk} = \frac{f(clock\_source)}{2 \cdot timer\_period\_register}$$

hence the frequency is:

$$f_{clk} = \frac{f_{CPU}}{4 \cdot 2 \cdot i} \quad (5.2)$$

Where  $i$  is the value set in the timer period register and  $f_{CPU}$  is the DSP Speed. The timer period register is set to  $i = 293$  which results in a frequency of  $f_{clk} = f_A = 63993Hz$ . The selection of the sampling rate is further explained in Section 5.1.2.

### Ping Pong Buffering

To reduce the overhead of interrupt routines vector processing is applied. The CPU will not be interrupted by every new value from the ADC to fetch it. Instead, the EDMA will take care of this. The EDMA will transfer the values from the ADC to the memory of the DSP and interrupt the CPU after one hole vector (in this case 128 values) has been transferred. Then the hole vector can be processed by the CPU, while the EDMA will continue to fetch the ADC values. To avoid

the EDMA overwriting the values on which the CPU is working, the vector is transferred to two different buffers. One buffer will be filled by the EDMA while the CPU is working with the other buffer (Figure 5.7). This is widely known as Ping Pong Buffering.

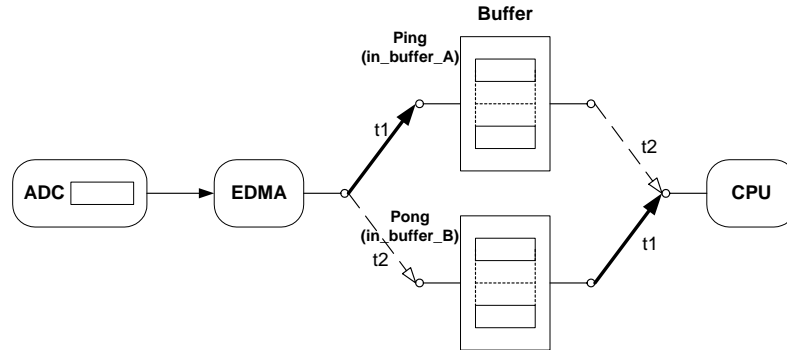


Figure 5.7.: Ping Pong Buffering

To do this, one EDMA Channel and two EDMA Reload Tables are needed. Two configurations are needed, one for the ping and one for the pong buffer. In both of them the source address is the Conversion Result Register (0xA000'0040) of the ADC and the source increment is disabled. The destination address in one of them is the start address of the ping buffer and the start address of the pong buffer is the destination of the other. The destination increment is enabled and the counter is set to the length of the vector (in this case 128). The link in the configuration for ping points to the configuration of the pong buffer and visa versa to result in a circular system. This is illustrated in Figure 5.8.

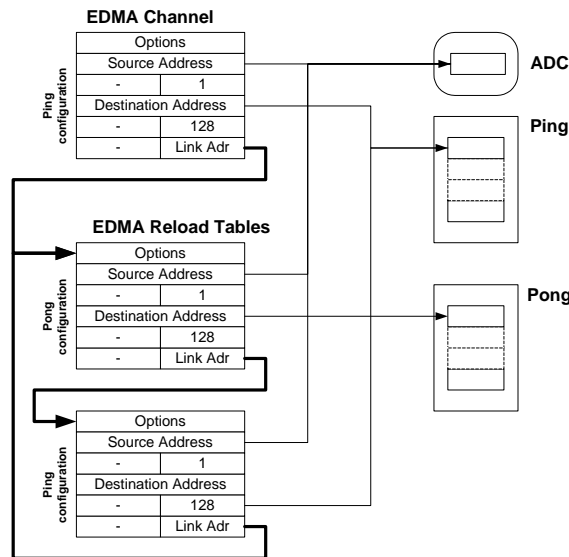


Figure 5.8.: EDMA setup for Ping Pong Buffering

Then the EDMA is configured to interrupt the CPU after a hole vector is transferred and the EDMA interrupt is enabled on the DSP. To determine which buffer the EDMA has just finished filling each configuration sets a different field in the Channel Interrupt Pending Register (CIPR). This is

specified by the Transfer Complete Code (TCC) field in the option part of the configuration. For more information on EDMA setup and registers, refer to [16].

Because the data from the ADC to the buffers are transferred by the EDMA, the L2 cache of the DSP does not recognize it and the cache valid flags are not changed. Therefore the DSP would read the old values from the cache. There are two ways to avoid this. The first way is to clear the valid flags of the cache for the buffer area every time the EDMA has finished a transfer. This can be done by the `CACHE_clean` function. The second way is to configure a part of the on-chip memory as mapped memory instead of cache. In this case the number of cache ways is reduced, but the data from the ADC are directly stored in on chip memory which gains performance. Therefore the second resolution is applied. The change of the internal memory usage can be done in the DSP/BIOS configuration file.

#### 5.1.4. Digital-to-Analog Conversation

To perform the digital-to-analog conversation the on board codec is used. It is a 16-bit dual channel voice or data codec and has a fix sample rate of 8000 Hz ([13]). It is produced by a 4.096 MHz clock on the DSK and the internal divider (512 times) of the codec. The codec is accessed through the multichannel buffered serial port (McBSP).

The two files `dac_codec.h` (Listing F.3) and `dac_codec.c` (Listing F.4) include the implementation for the setup and the handling of the codec. It uses the AD535 module from the Board Support Library (BSL) which is a set of application programming interfaces used to configure and control on-board devices.

As the sampling rate of the DAC is low compared to the ADC frequency and the the downsampling rate of the ADC can not be matched exactly (see Section 5.1.2), the data is directly written to the DAC by the DSP without using the EDMA. Nevertheless, the output also uses two Buffers because vector processing is applied. Therefore Figure 5.7 can be extended to Figure 5.9.

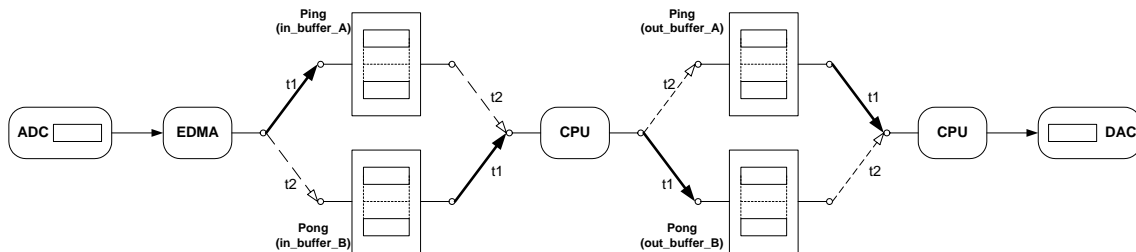


Figure 5.9.: Ping Pong Buffering DAC

The scheduling of the two CPU blocks is explained in Section 5.1.6.

#### 5.1.5. Programm Flow

Like in every C programm the entry point is the main function. In this case the main function is used to initialize the used hardware, the software libraries, and to set up the interrupts. After the initialization, the application goes to an *idle* loop to be driven by interrupts (see Figure 5.10). The main function and the interrupt routines are implemented in the `fm_dem_main.c` file. Figure 5.11 shows a simplified layout.

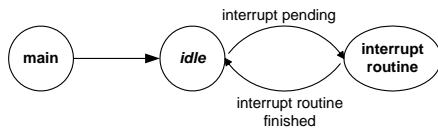


Figure 5.10.: Program flow

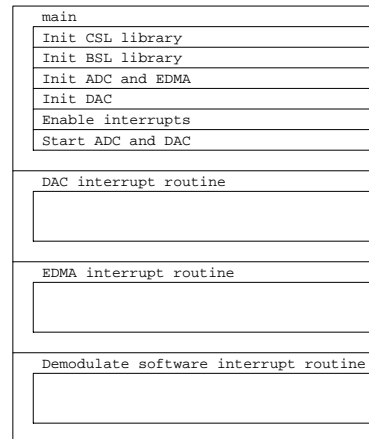


Figure 5.11.: Layout main file

### 5.1.6. Interrupt Routines and Scheduling

The system is driven by two different hardware interrupts. One from the EDMA of the ADC and the other from the codec DAC. Therefore the CPU is invoked from two different sources and necessitates proper scheduling. The two hardware interrupts have the same priority level and can not interrupt each other. To avoid missing an interrupt the time intensive calculation of the demodulation algorithm is done in a software interrupt (SWI) which is posted from the EDMA interrupt routine and can be interrupted by any hardware interrupt as shown in Figure 5.12. Software inter-

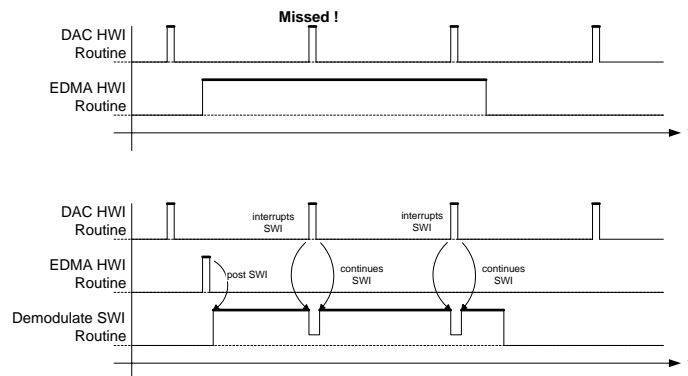


Figure 5.12.: Scheduling with software interrupt

rupts can be configured in the DSP/BIOS configuration and they are posted with the `SWI_post` function. The hardware interrupts are kept as short as possible.

Figure 5.13 shows an outline of the interrupts, interrupt routines and what is responsible for setting the pointers.

#### DAC Interrupt Routine

The DAC interrupt routine just writes the current value to the codec and increments the output counter, which determines current index of the buffer. In case of the missing sample it also performs the interpolation as in Listing 5.1 and Figure 5.14.

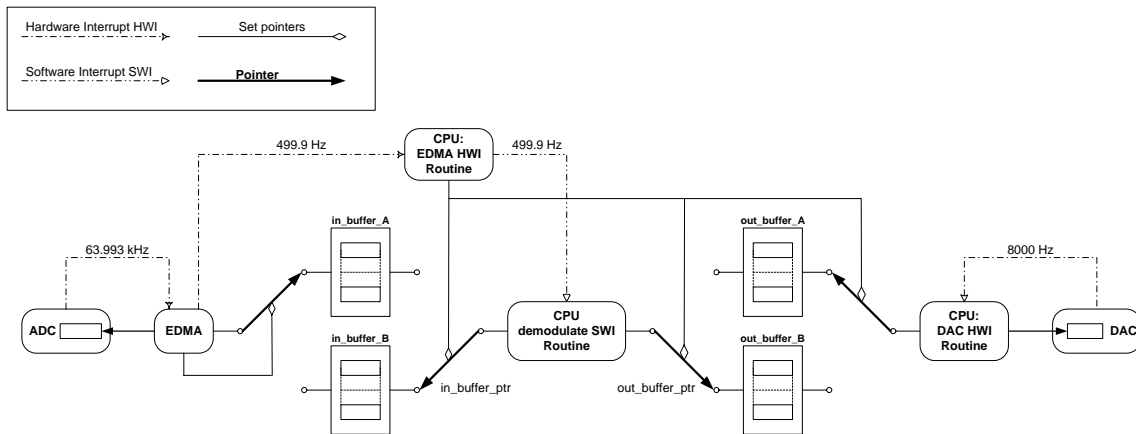


Figure 5.13.: Overview interrupts

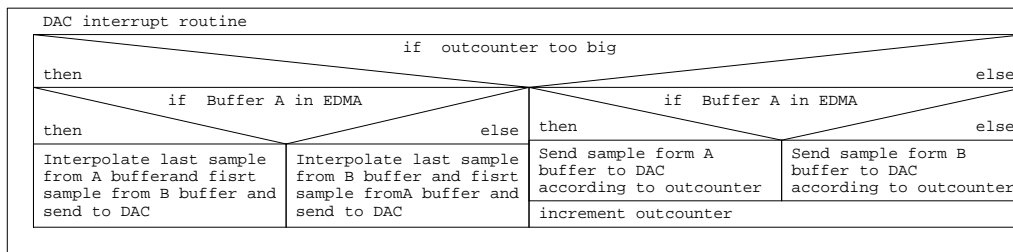


Figure 5.14.: DAC interrupt routine

Listing 5.1: DAC interrupt routine

```

1 void dacInt(void)
2 {
3     /* interpolate a missing sample */
4     if (outcounter>15)
5     {
6         /* select the buffer */
7         if(a_inEDMA)
8         {
9             AD535_HWI_write(outAD535h, (( out_buffer_A[15]+
10                out_buffer_B[0])/2));
11         }else
12         {
13             AD535_HWI_write(outAD535h, ((out_buffer_B[15]+
14                out_buffer_A[0])/2));
15         }
16     }else /* output a value */
17     {
18         /* select the buffer */
19         if(a_inEDMA)
20         {
21             AD535_HWI_write(outAD535h,out_buffer_A[outcounter]);
22         }else
23         {
24             AD535_HWI_write(outAD535h,out_buffer_B[outcounter]);
25         }
26         /* update outcounter */
27         outcounter++;
28     }
29 }

```

## EDMA Interrupt Routine

The EDMA interrupt routine resets the output counter, clears the transfer completion interrupt flag, sets the pointers and flags for the new buffers, and then posts the software interrupt (Listing 5.2 and Figure 5.2).

## Demodulate Software Interrupt Routine

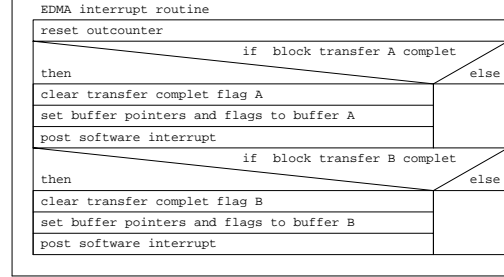
In this routine the actual demodulation is carried out. The functions of the demodulation algorithms are called as in Listing 5.3 and Figure 5.16.

**Listing 5.2: EDMA interrupt routine**

```

1 void edmaInt(void)
2 {
3     /* reset outcounter */
4     outcounter=0;
5
6     /* select EDMA channel */
7     if (EDMA_intTest(THS1408_BUFFER_A_TCC))
8     {
9         /* Clear transfer completion interrupt flag */
10        EDMA_intClear(THS1408_BUFFER_A_TCC);
11
12        /* set buffer pointers */
13        a_inEDMA=0;
14        in_buffer_ptr=in_buffer_A;
15        out_buffer_ptr=out_buffer_A;
16
17        /* post software interrupt -> start demodulation
18        */
19        SWI_post(&demodulate_swi);
20    }
21    if (EDMA_intTest(THS1408_BUFFER_B_TCC))
22    {
23        /* Clear transfer completion interrupt flag */
24        EDMA_intClear(THS1408_BUFFER_B_TCC);
25
26        /* set buffer pointers */
27        a_inEDMA=1;
28        in_buffer_ptr=in_buffer_B;
29        out_buffer_ptr=out_buffer_B;
30
31        /* post software interrupt -> start demodulation
32        */
33        SWI_post(&demodulate_swi);
34    }
35 }

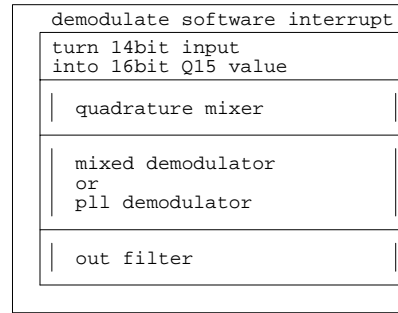
```

**Figure 5.15.: EDMA interrupt routine****Listing 5.3: Demodulate software interrupt routine**

```

1 void demodulateSwiFunc(void)
2 {
3     int i;
4     /* turn 14bit input into 16bit Q15 value */
5     for (i=0 ; i<BUFFERSIZE_INPUT;i++)
6     {
7         in_buffer_ptr[i]=in_buffer_ptr[i]<<2;
8     }
9
10    quad_mix(in_buffer_ptr,re_buffer,im_buffer);
11
12    mixed_demodulate(re_buffer,im_buffer,out_buffer);
13
14    out_filter(out_buffer,out_buffer_ptr);
15 }

```

**Figure 5.16.: Demodulate software interrupt routine**

## 5.2. Floating-Point Algorithm

In this section the mixed demodulator and PLL demodulator are specially discussed in regards to their implementations in C.

### 5.2.1. Quadrature Mixer

Figure 5.17 shows again the block diagram of the Quadrature with the buffer names as they are used in the code sequences. There are two important tasks. A sine and a cosine oscillators have to be generated first. Then a low-pass filter will be specified.

To generate a sine or a cosine signal it is possible to use an IIR filter. The impulse responses of the IIR filters have to take the following forms:

$$h_S(n) = \sin(\omega_M \cdot n \cdot T)$$

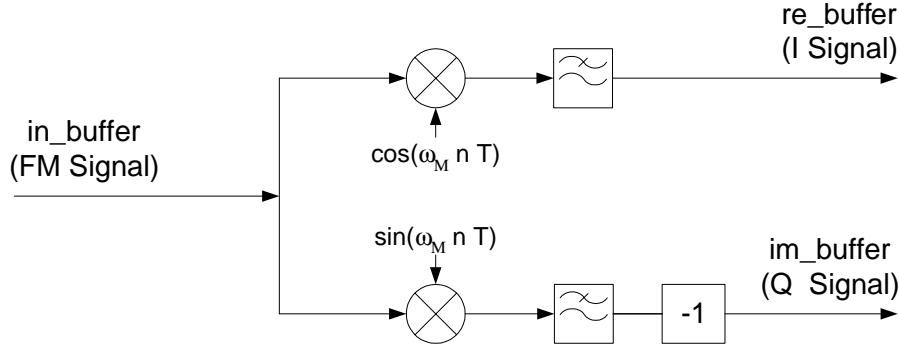


Figure 5.17.: Quadrature mixer

and:

$$h_C(n) = \cos(\omega_M \cdot n \cdot T)$$

$T$  represents the sample time, which is used to run the filters.  $\omega_M$  is the angular frequency of the demanded oscillation. The exact angular frequency depends on the spectrum situation before quadrature mixing (see Chapter 5.1).

The filter characteristics of an IIR filter are completely specified by their impulse responses. The Z transformation of the impulse response delivers all coefficients of the filter. In this case it is about designing a filter on the limit of stability (non decaying impulse response). A difficulty of such filters, specially in a fixed point implementation, is to keep the limit of stability all the time. The required Z transfer function can be found in a simple Z transformation table:

$$h_S(n) = \sin(\omega_M \cdot n \cdot T) \xleftrightarrow{Z} H_S(z) = \frac{\sin(\omega_M \cdot T) \cdot z^{-1}}{1 - 2 \cdot \cos(\omega_M \cdot T) \cdot z^{-1} + z^{-2}} \quad (5.3)$$

and:

$$h_C(n) = \cos(\omega_M \cdot n \cdot T) \xleftrightarrow{Z} H_C(z) = \frac{1 - \cos(\omega_M \cdot T) \cdot z^{-1}}{1 - 2 \cdot \cos(\omega_M \cdot T) \cdot z^{-1} + z^{-2}} \quad (5.4)$$

The denominators of the two Z transfer functions are equal, this brings along the common use of the delay stores. Hence the two oscillations can be created with a filter with one feedback path and two forward paths (see Figure 5.19). The filter structure can be directly derived from the Z transfer function [5]. If the filter is actuated with a delta function, the output would be a sine and a cosine signal, respectively (because their impulse response specification). These signals are then multiplied with the FM signal  $s_{FM}(n)$ .

The next step is to filter them with a low pass filter. For fast applications, an IIR filter is preferred to a FIR filter due to the smaller filter order for the same specifications. High filter orders are the main cause of undesirable time delay.

To get the IIR filter specifications, it is necessary to look at the spectrum situation. Figure 5.4 shows the spectrum of the FM signal after subsampling. The pass frequency is defined by the half bandwidth of the original FM signal ( $f_p = 6300\text{Hz}$ ). That is the highest frequency which has to be handled. The stop frequency is chosen as high as possible, but low enough to prevent aliasing ( $f_s = 19000\text{Hz}$ ). The pass band loss and the stop band attenuation were declared, to obtain the filter order  $n$  as small as possible, but larger enough attenuation in the stop band.

A useful tool to calculate IIR filter coefficients is MATLAB. The following code sequence shows a matlab M-File which returns the coefficients of a butterworth IIR filter with the given specifications:

```

1 %IIR low-pass filter for quadrature mixing
2
3 %specifications:
4 %sample frequency in Hz
5 fa=15000000/(8*293);
6 %pass frequency in Hz
7 fp=6300;
8 %stop frequency in Hz
9 fs=19000;
10 %loss in pass band in dB
11 dp=2;
12 %loss in stop band in dB
13 ds=60;
14
15 %computing:
16 [n, fn]=buttord(fp*2/fa, fs*2/fa, dp, ds);
17 %return:
18 %order
19 n
20 %cutoff frequency
21 fn
22
23 %computing:
24 [B,A]=butter(n,fn);
25 %return:
26 %denominator of transfer function
27 B
28 %numerator of transfer function
29 A
30
31 %computing:
32 [H,W]=freqz(B,A,n);
33 %return:
34 freqzplot(H,W);

```

The filter order is computed to  $n = 5$  and the 3dB cutoff frequency to  $f_n = 6653\text{Hz}$ . So the transfer function of the low pass filter is:

$$H_{LP}(z) = 10^{-3} \cdot \frac{1.513433 + 7.567165 \cdot z^{-1} + 15.134330 \cdot z^{-2} + 15.134330 \cdot z^{-3} + 7.567165 \cdot z^{-4} + 1.513433 \cdot z^{-5}}{1 - 2.895729 \cdot z^{-1} + 3.634448 \cdot z^{-2} - 2.392826 \cdot z^{-3} + 0.817619 \cdot z^{-4} - 0.115082 \cdot z^{-5}}$$

Figure 5.18 shows the frequency response of the low pass filter.

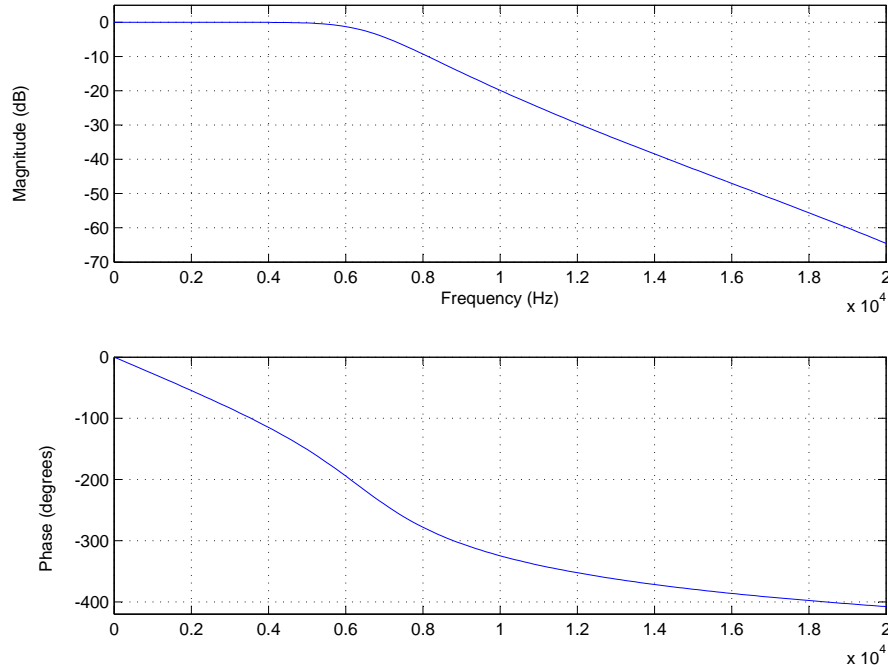


Figure 5.18.: Frequency response of low-pass filter for quadrature mixing, floating point

Now the parts of the quadrature mixer are discussed in the view of its digital realization. Figure 5.19 shows the whole digital block diagram of the quadrature mixer with the buffer and variable names which are used in the code sequences. The following C function represents the translation from digital block structure to C code.



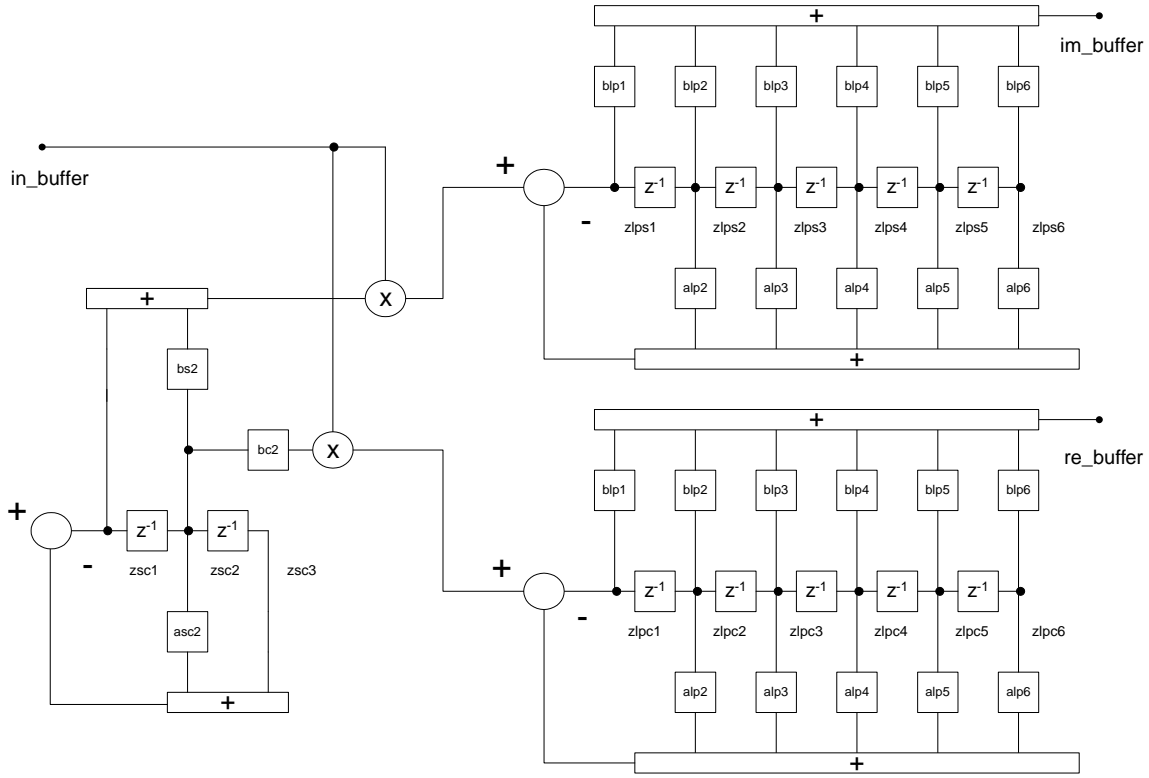


Figure 5.19.: Digital block diagram of quadrature mixer

```

1 void quad_mix(float in_buffer[], float re_buffer[], float
  im_buffer[])
2 {
3   int i;
4   /*__cosine/sine generation__*/
5   for(i=0;i<BUFFERSIZE_INPUT;i++)
6   {
7     /*feedback*/
8     zsc1=-(zsc2*asc2+zsc3);
9     /*foreward*/
10    re_buffer[i]=zsc1+zsc2*bc2/*cosine*/
11    im_buffer[i]=zsc2*bs2/*sine*/
12    /*shift delays*/
13    zsc3=zsc2;
14    zsc2=zsc1;
15  }
16  /*__mixing__*/
17  for(i=0;i<BUFFERSIZE_INPUT;i++)
18  {
19    /*mixing*/
20    re_buffer[i]=re_buffer[i]*in_buffer[i]/*I-path*/
21    im_buffer[i]=im_buffer[i]*in_buffer[i]/*Q-path*/
22  }
23  /*__low pass__*/
24
25  31
26  32 for(i=0;i<BUFFERSIZE_INPUT;i++)
27  33 {
28  34   /*feedback*/
29  35   zlpc1=re_buffer[i]-(alp2*zlpc2+alp3*zlpc3+alp4*zlpc4+
30  36   alp5*zlpc5+alp6*zlpc6); /*I-path*/
31  37   zpls1=im_buffer[i]-(alp2*zpls2+alp3*zpls3+alp4*zpls4+
32  38   alp5*zpls5+alp6*zpls6); /*Q-path*/
33  39   /*foreward*/
34  40   re_buffer[i]=blp1*zlpc1+blp2*zlpc2+blp3*zlpc3+blp4*zlpc4
35  41   +blp5*zlpc5+blp6*zlpc6; /*I-path*/
36  42   im_buffer[i]=-(blp1*zpls1+blp2*zpls2+blp3*zpls3+blp4*
37  43   zpls4+blp5*zpls5+blp6*zpls6); /*Q-path*/
38  44   /*shift delays*/
39  45   /*I-path*/
40  46   zlpc6=zlpc5;
41  47   zlpc5=zlpc4;
42  48   zlpc4=zlpc3;
43  49   zlpc3=zlpc2;
44  50   zlpc2=zlpc1;
45  51   /*Q-path*/
46  52   zpls6=zpls5;
47  53   zpls5=zpls4;
48  54   zpls4=zpls3;
49  55   zpls3=zpls2;
50  56   zpls2=zpls1;
51  }
52 }

```

The function is called with three parameters. The first parameter is a pointer to an array containing values of the subsampled FM signal. The other arguments are the pointers to the arrays used for the output of the quadrature mixer: re\_buffer and im\_buffer contain values of the I and Q signals, respectively.

First the cosine and sine signals are produced and saved in the re\_buffer (cosine) and the im\_buffer

(sine). The IIR filter, which is used to generate these signals, must be actuated with a delta function. Instead of using a delta function as input signal, the delay variables can be declared to achieve the same result.

The next step is to multiply the cosine and sine signal with the subsampled FM signal and save them back to the `re_buffer` and `im_buffer`, respectively.

At last the signals in the `re_buffer` and `im_buffer` are low-pass filtered and saved back to `re_buffer` and `im_buffer`.

### 5.2.2. First Down Sampling

In Figure 5.20 the spectrum after quadrature mixing (of one path) is shown. It shows that the

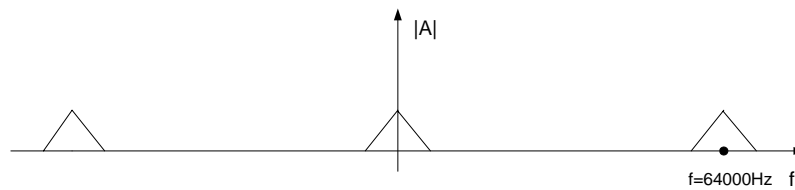


Figure 5.20.: Signal spectrum after quadrature mixing

sampling frequency  $f = 64000\text{Hz}$  can be reduced with a factor of four to  $f = 16000\text{Hz}$ . Figure 5.21 represents the frequency response after down sampling (of one path). The reduction can

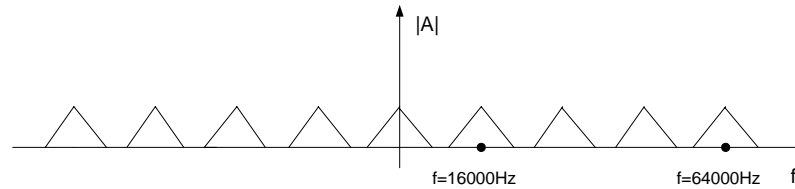


Figure 5.21.: Signal spectrum after first down sampling

be realized by picking every fourth value of the `re_buffer` and `im_buffer` and store them in the `re_buffer_down_one` and `im_buffer_down_one` (reduced buffersize), respectively.

The code sequence below explains the down sampling in C:

```

1 void downsample_one(float re_buffer[], float im_buffer[], float re_buffer_down_one[], float im_buffer_down_one[])
2 {
3     int i;
4
5     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
6     {
7         re_buffer_down_one[i]=re_buffer[i*DOWNSAMPLE_ONE];
8         im_buffer_down_one[i]=im_buffer[i*DOWNSAMPLE_ONE];
9     }
10 }
```

### 5.2.3. Mixed Demodulator

Figure 5.22 shows again the architecture of the mixed demodulator. The signals are annotated by their names in the code sequence. The associated C code is given as follows:

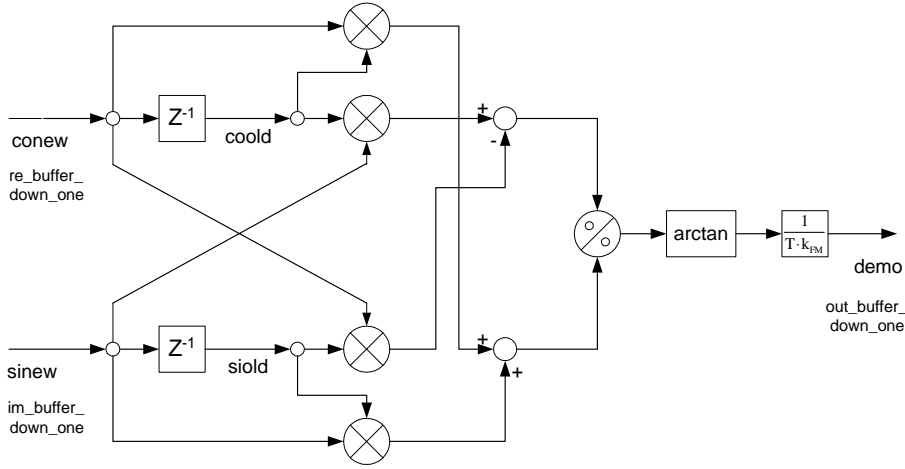


Figure 5.22.: Block diagram of mixed demodulator

```

1 void mixed_demodulate(float re_buffer_down_one[], float im_buffer_down_one[], float out_buffer_down_one[])
2 {
3     int i;
4     for(i=0; i<BUFFERSIZE_DEMODULATOR; i++)
5     {
6         /*__shifting values__*/
7         coold=conew;
8         siold=sinew;
9         conew=re_buffer_down_one[i];
10        sinew=im_buffer_down_one[i];
11
12        /*__calculating demodulated value__*/
13        demo=(coold*sinew-siold*conew)/(coold*conew+siold*sinew);/*atan argument*/
14        demo=atanf(demo);/*atan value*/
15        demo=demo*decon;/*demodulated value*/
16        out_buffer_down_one[i]=demo;
17    }
18 }
19

```

The parameters are the pointers to the arrays which contain the down sampled I signal and Q signal (re\_buffer\_down\_one, im\_buffer\_down\_one) and a pointer to an array containing the demodulated signal (out\_buffer\_down\_one).

First the delay has to be realized. The value conew and sinew must be declared with a nonzero value to prevent a division by zero by the first call. Then the demodulated value can be calculated as explained in the block diagram.

#### 5.2.4. PLL Demodulator

Figure 5.23 shows the block diagram with the buffer and variable names as they are used in C. The constant P\_PLL is chosen  $P_{PLL} = 2$  as declared in Section 3.3.3. Another constant K\_PLL was declared to be  $K_{PLL} = k_{FM} \cdot T \approx 1.2$ . With these values, there are some distortions around  $f = 1000\text{Hz}$ . The best result is achieved with  $K_{PLL} = 2.8$ .

```

1 void pll_demodulate(float re_buffer_down_one[], float
2     im_buffer_down_one[], float out_buffer_down_one[])
3 {
4     int i;
5     float demo; /*value of demodulation*/
6     for(i=0; i<BUFFERSIZE_DEMODULATOR; i++)
7     {
8         /*__calculation of demodulation__*/
9         i_path=re_buffer_down_one[i]*sinf(arg);
10        q_path=im_buffer_down_one[i]*cosf(arg);
11
12        demo=q_path-i_path;
13        demo=demo*P_PLL;
14
15        out_buffer_down_one[i]=demo*GAIN_PLL;
16
17        /*__summation of argument__*/
18        arg=arg+K_PLL*demo;
19
20        /*__limitation of argument__*/
21        if (arg>2*PI)
22        {
23            arg=arg-2*PI;
24        }
25    }
26 }

```

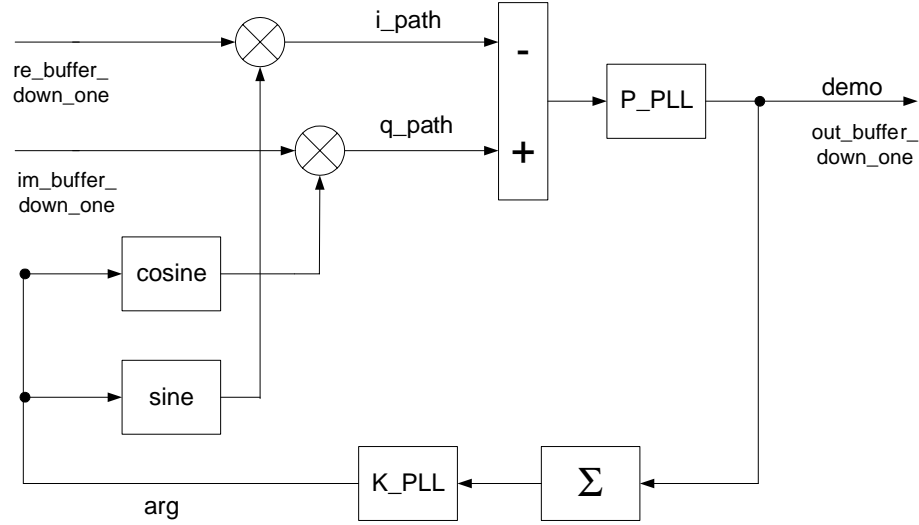


Figure 5.23.: Block diagram of PLL demodulator

```

24     }
25
26     else if (arg<-2*PI)
27     {
28         arg=arg+2*PI;
29     }
30     } /* for */
31 } /* pll_demodulate */

```

The argument (arg) of sine and cosine must be limited because a small DC offset is enough to rise the argument beyond the valid range due to the summation.

### 5.2.5. Filter

After demodulation the demodulated signal should be filtered with a band-pass filter to suppress the noise or DC offset.

The coefficients of the band-pass filter are also calculated with a MATLAB M-file:

```

1 %IIR band-pass filter
2
3 %specifications:
4 %sample frequency in Hz
5 fa=15000000/(8*293*4);
6 %first pass-band frequency in Hz
7 fp1=200;
8 %second pass-band frequency in Hz
9 fp2=3600;
10 %IIR band-pass filter order
11 n=5;
12
13 %computing
14 [b,a]=butter(n,[fp1*2/fa fp2*2/fa]); [H,W]=freqz(b,a);
15 freqzplot(H,W);

```

The band-pass filter frequencies are specified as the speech message signal (see Section 1.3). The stop-band frequency is chosen, that the band-pass filter order is  $n = 5$ . Figure 5.24 shows the frequency response of the band-pass filter. The kind of implementation in C is the same as the low-pass filter of the quadrature mixer:

```

1 void out_filter(float out_buffer_down_one[])
2 {
3     int i;
4
5     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
6     {
7         /*feedback*/
8         zbp1=out_buffer_down_one[i]-(abp2*zbp2+abp3*zbp3+abp4*
9             zbp4+abp5*zbp5+abp6*zbp6+abp7*zbp7+abp8*zbp8+abp9
10             *zbp9+abp10*zbp10+abp11*zbp11);
11
12         /*foreward*/
13         out_buffer_down_one[i]=bbp1*zbp1+bbp2*zbp2+bbp3*zbp3+
14             bbp4*zbp4+bbp5*zbp5+bbp6*zbp6+bbp7*zbp7+bbp8*zbp8
15             +bbp9*zbp9+bbp10*zbp10+bbp11*zbp11;
16
17         /*shift delays*/
18         zbp11=zbp10;
19         zbp10=zbp9;
20         zbp9=zbp8;
21         zbp8=zbp7;
22         zbp7=zbp6;

```

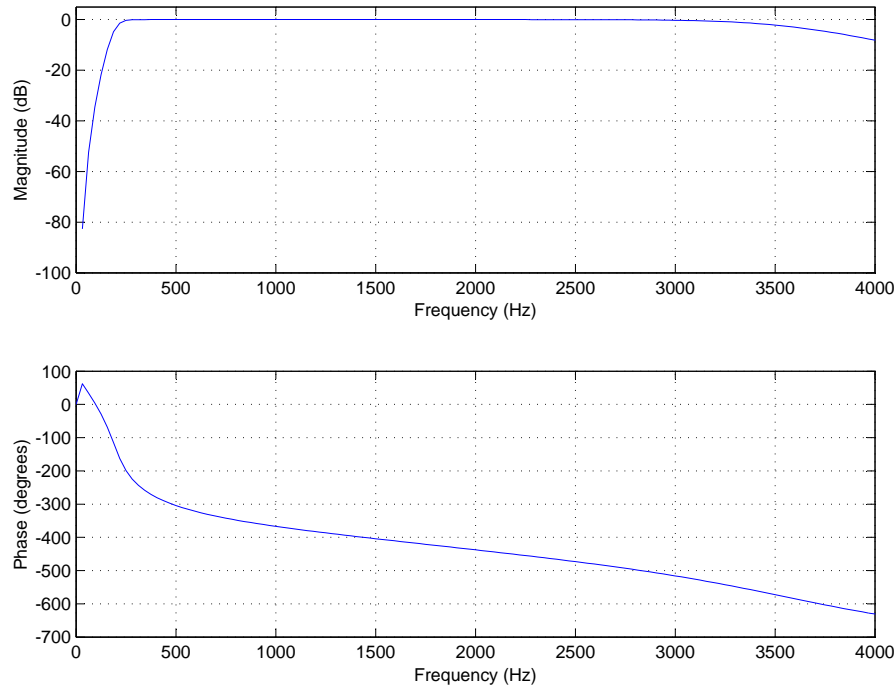


Figure 5.24.: Band-pass filter frequency response, floating point

```

19      zbp6=zbp5;
20      zbp5=zbp4;
21      zbp4=zbp3;
22      zbp3=zbp2;

23      zbp2=zbp1;
24      }
25  }
```

### 5.2.6. Second Down Sampling

After filtering, a second down sampling is necessary because the digital analog converter runs with a fix frequency of  $f = 8000\text{Hz}$ . Figure 5.25 represents the signal spectrum after demodulation (speech signal). After the second down sampling the spectrum of the signal is showed in Figure 5.26.

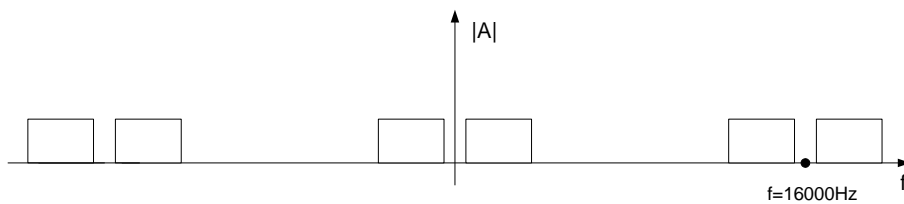


Figure 5.25.: Signal spectrum before second down sampling

## 5.3. Fixed-Point Algorithm

In this section the fixed-point implementations are explained. Before the algorithms are discussed a short introduction to fractional fixed-point numbers is given.

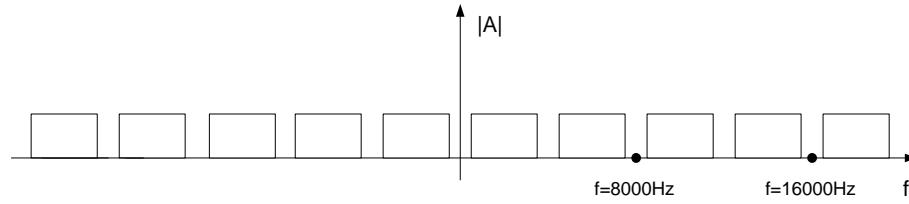


Figure 5.26.: Signal spectrum after second down sampling

### 5.3.1. Fractional Q Formats

In computing arithmetic, fractional quantities can be approximated by using a pair of integers  $(n, e)$  the mantissa and the exponent. The pair represents the fraction:

$$n \cdot 2^{-e}$$

The exponent  $e$  can be considered as the number of digits you have to move into  $n$  before placing the binary point. If  $e$  is a variable quantity, held in a register and unknown at compile time,  $(n, e)$  is said to be a floating-point number. If  $e$  is known in advance, at compile time,  $(n, e)$  is said to be a fixed-point number. Fixed-point numbers can be stored in standard integer variables by storing the mantissa.

In a  $Q_m.n$  format, there are  $m$  bits used to represent the two's complement integer portion of the number, and  $n$  bits used to represent the two's complement fractional portion.  $m + n + 1$  bits are needed to store a general  $Q_m.n$  number. The extra bit is needed to store the sign of the number in the most-significant bit position. The representable integer range is specified by  $(-2^m, 2^m)$  and the finest fractional resolution is  $2^{-n}$ . The most commonly used format is Q.15. Q.15 means that

Bit	15	14	13	12	11	10	9	...	0
Value	S	Q14	Q13	Q12	Q11	Q10	Q9	...	Q0

Figure 5.27.: Q.15 Bit Fields

a 16-bit word is used to express a signed number between positive and negative one. The most-significant binary digit is interpreted as the sign bit in any Q format number. Thus, in Q.15 format, the decimal point is placed immediately to the right of the sign bit. The fractional portion to the right of the sign bit is stored in regular two's complement format (see Figure 5.27). The approximate allowable range of numbers in Q.15 representation is  $(-1, 1)$  and the finest fractional resolution is  $2^{-15} = 3.05 \cdot 10^{-5}$ . The integer value of an Q.15 value can be computed using equation:

$$i = \text{round}(f \cdot 2^{15}) \quad (5.5)$$

Where  $i$  is the integer value and  $f$  is the fractional value. For example:

$$i = \text{round}(0.5 \cdot 32768) = 16384$$

The following subsections show how to perform the basic arithmetic operations on two fixed point numbers,  $a = n2^{-p}$  and  $b = m2^{-q}$ , expressing the answer in the form  $c = k2^{-r}$ , where  $p$ ,  $q$  and  $r$  are fixed constant exponents.

### Change of Exponent

The simplest operation to be performed on a fixed-point number is to change the exponent. To change the exponent from  $p$  to  $r$  the mantissa  $k$  can be calculated from  $n$  by a simple shift.

Since:

$$n2^{-p} = n2^{r-p} \cdot 2^{-r}$$

the implementation is

$$\begin{aligned} k &= n \ll (r-p) \text{ if } (r \geq p) \\ k &= n \gg (p-r) \text{ if } (p > r) \end{aligned}$$

### Addition and Subtraction

To perform the operation  $c = a + b$ , first convert  $a$  and  $b$  to have the same exponent  $r$  as  $c$  and then add the mantissas. This method is proved by the equation:

$$n2^{-r} + m2^{-r} = (n + m)2^{-r}$$

Therefore two Q.15 numbers can be simply added:

$$k = n + m$$

Subtraction is similar.

### Multiplication

The product  $c = a \cdot b$  can be performed using a single integer multiplication. From the equation:

$$a \cdot b = n2^{-p} \cdot m2^{-q} = (n \cdot m)2^{-(p+q)}$$

it follows that the product  $n \cdot m$  is the mantissa of the answer with exponent  $p + q$ . To convert the answer to have exponent  $r$ , perform shifts as described above.

For example, two Q.15 numbers:

$$k = (n * m) \gg 15$$

### Division

Division  $c = a/b$  can also be performed using a single integer division. The equation is:

$$\frac{a}{b} = \frac{n2^{-p}}{m2^{-q}} = \left(\frac{n}{m}\right)2^{q-p} = \left(\frac{n}{m}\right)2^{(r+q-p)}2^{-r}$$

In order not to lose precision, the multiplication must be performed before the division by  $m$ .

Q.15 example:

$$k = (n \ll 15) / m$$

### 5.3.2. Quadrature Mixer

Like in the floating-point implementation, the sine and the cosine signal of the mixer are generated with an second order IIR filter. The coefficients remain the same, except that they have to be transformed to a Q.15 value according to Eq. 5.5. To avoid overflows in the delay states and the output value, the delay states are initialized to  $\{0 \ 0 \ 0.8\}$  or rather  $\{0 \ 0 \ 26214\}$  in Q.15 instead of  $\{0 \ 0 \ 1\}$  (delta function).

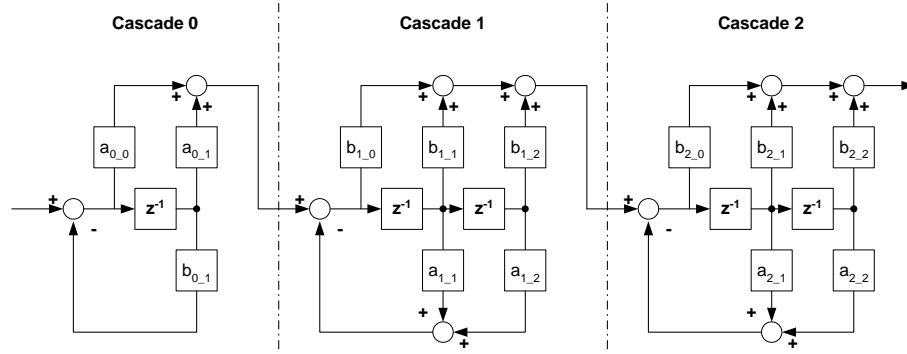


Figure 5.28.: Block diagram of cascaded low-pass filter

The low-pass filter designed in the floating-point implementation was fifth order. IIR filters is very sensitive to quantization errors. The filter transfer function of a filter higher than second order will change so much due to quantization that the filter is no longer working properly. The solution is to realize the filter with cascades of first and second order filters. Hence the quadrature mixer low-pass filter is slitted into one first order and two second order filters (Figure 5.28).

MATLAB provides functions to split the filter into cascaded second-order sections. First  $[z, p, k] = \text{butter}(n, Wn)$  obtains zero-pole-gain form, then  $[sos, g] = \text{zp2sos}(z, p, k, 'order', 'scale')$  converts the zero-pole-gain parameters to second-order sections. Using infinity-norm scaling (scale) in conjunction with up-ordering (order) minimizes the probability of overflow in the realization.

The  $a_{x\_1}$  coefficient of each cascade is greater than one and can therefore not be expressed as a Q.15 number. The solution is to split the coefficient into two coefficients

$$a_{x\_1a} + a_{x\_1b} = a_{x\_1}$$

which are now less than one and can be expressed in the Q.15 format. The implementation will change as shown in Figure 5.29.

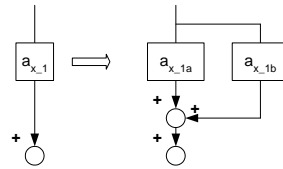


Figure 5.29.: Change in block diagram of

To reduce the computing load, the first downsampling is included in the last cascade of the filter. Hence the forward path of the last cascade is only executed every fourth time as in Figure 5.30. The output values are stored in the first quarter of the re\_buffer and the im\_buffer and thus the



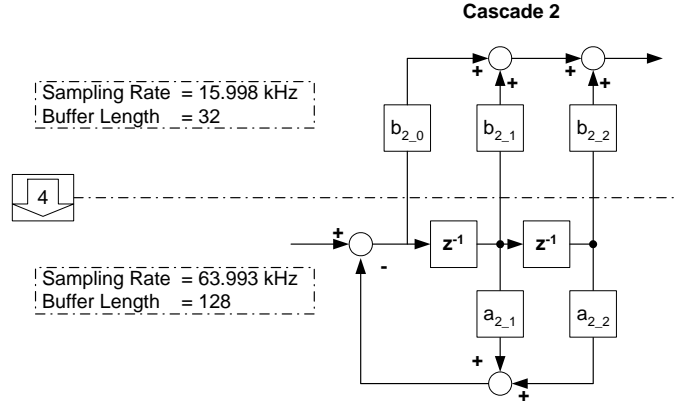


Figure 5.30.: Output signals and delay states of cascaded low-pass filter

buffer length is reduced from 128 to 32.

The design of the low-pass filter and the oscillator for the fixed-point implementation are done with the MATLAB file *mixer\_design.m*. It calculates the coefficients for the filter and the oscillator, scales them to the Q.15 value and plots them in a format to the screen that can be copied into the source code. Afterwards it tests the filter for overflows by monitoring the delay states and outputs of every cascade. To represent a worst case scenario, a square waveform is chosen as the input signal. Figure 5.31 shows the plots of all the monitored signals. It shows that due to scaling, an

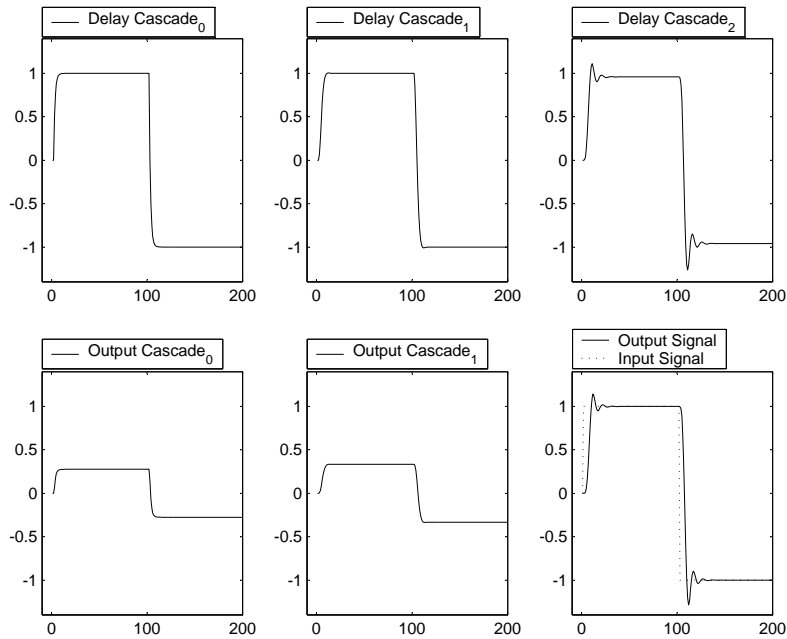


Figure 5.31.: Output signals and delay states of cascaded low-pass filter

overflow could still occur during the settling time. To preclude this the down scaling of 0.8 introduced through the oscillator is not undone before the filter and therefore the input signal to the

filter is always  $\leq 0.8$ .

The implementation of the quadrature mixer is done in the files *quad\_mix.c* and *quad\_mix.h*.

### 5.3.3. Mixed Demodulator

The implementation of the fixed-point mixed demodulator is very close to the floating-point implementation. The main differences are highlighted below.

The arctangent table is created with the files *atan.c* and *antan.h*. They define a constant table with the values of the arctangent function. The MATLAB file *tanTabGen.m* is used to calculate the values of the table and write the h and c files. To reduce the computing load and avoid scaling and overflow problems, the factor  $\frac{1}{T \cdot k_{FM}}$  (see Eq.5.22) is included in the table. Therefore the lookup table fulfils the equation:

$$out = \frac{1}{T \cdot k_{FM}} \cdot \arctan(in) \quad (5.6)$$

The minimum range of the table can be calculated by using the maximum derivation of 3000 Hz:

$$in_{\max} = \tan(k_{FM} \cdot T) = \tan\left(\frac{2 \cdot \pi \cdot \Delta F}{f_A}\right) = \tan\left(\frac{2 \cdot \pi \cdot 3000}{15998}\right) = 2.415$$

To allow easy determination of the index, the range of the table should be a power of two. Therefore the input range of the arctangent is set to  $2^2 = 4$ . Only the positive part of the arctangent function is stored in the table, because the function is symmetrical and the negative part can be derived from the positive part.

One approach is to use the value from the lookup table directly as output. Therefore the table has to store enough values to be sufficient accurate. Another approach is to interpolate the output. This involves additional computing time, but the table size can be reduced. To further reduce the additional computing time a second table with the gradients of the function is used for the interpolation. The MATLAB file *tanTabGenInterpol.m* is used to generate these two tables. Figure 5.32 shows the window plot of the direct lookup table with a table size of 128, the interpolated lookup with two tables of 16 values, and the actual arctangent function. It shows that the interpolated lookup is more accurate.

In Section 6.2.1 various measurements with different table sizes are done and it showed that direct lookup with table of 128 values and interpolated lookup with two tables of 16 values achieve a good effort and profit ratio.

Listing 5.4 shows the calculation of the argument. The signals before the division are stored in the Q.31 format to avoid underflows and the division is scaled to a Q17.14 format. Therefore the three local variables *re\_path*, *im\_path* and *dem\_arg* are Integers (32bit).

**Listing 5.4:** Argument calculation fixed-point mixed demodulator

```
1 re_path = ( im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
2 im_path = ( re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */
3
4 /* shift delays */
5 dem_z_re=re_buffer[i];
6 dem_z_im=im_buffer[i];
7
8 /* calculate the argument */
9 dem_arg=re_path/(im_path>>14); /* Q17.14 */
```

### 5.3.4. PLL Demodulator

The fixed-point implementation of the PLL demodulator differs from the floating-point implementation, specially in the calculation of the sine and cosine values of the summed demodulated signal

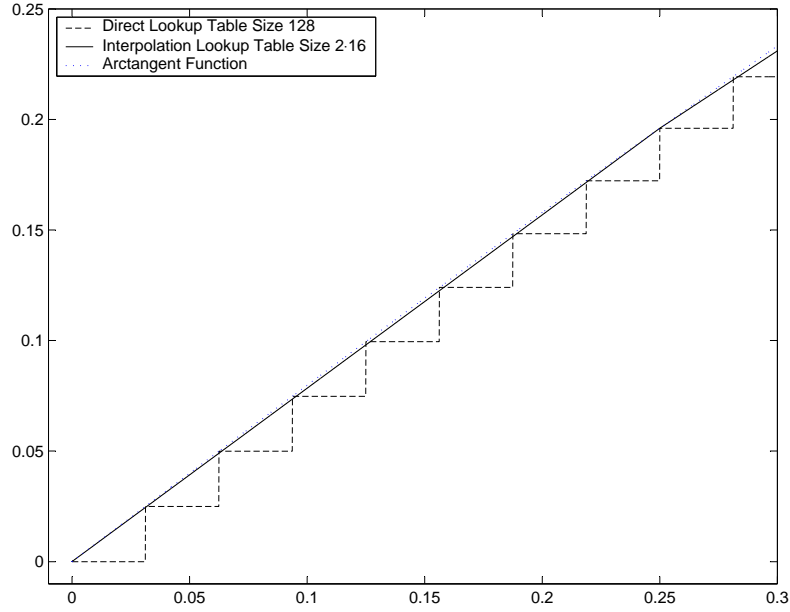


Figure 5.32.: Plot arc tangent lookup table

(arg). The rest of the algorithm involves normal change from floating-point to a convenient Q format.

The calculation of the demodulated value is done in the Q15 format (Listing 5.5 line 12 to 20). The following summation of the demodulated signal is limited to  $2 \cdot \pi$ . That means that the Q format has to change from Q15 to Q3.12 (Listing 5.5 line 23 to 39):

$$Q3.12_{\max} = (2^{15} - 1) \cdot 2^{-12} \approx 8 > 2 \cdot \pi$$

The next step is to compute the sine and cosine values of the summed demodulated signal (Listing 5.5 line 42 to 162). The idea is to access a buffer, which contains the sine values. To calculate sine and cosine values of an argument between  $-2\pi$  and  $2\pi$ , it is enough to save a fourth of a sine wave. This is first done in a M-file which is later integrated to the C code. The M-file delivers an array, called `sine_buffer` with the array size `BUFFER_SIZE_SINE`. `sine_buffer` contains sine values of sine arguments in a range from 0 to  $\pi/2$ .

To access the `sine_buffer` it is necessary to calculate an index to the argument (Listing 5.5 line 44 to 60). In a floating-point application the index could be calculated as:

$$index_{short} = \frac{arg_{float}}{step_{float}}$$

$step_{float}$  is the step between the sine arguments, declared in the M-file:

$$step_{float} = \frac{\pi}{2 \cdot (BUFFER\_SIZE\_SINE - 1)}$$

$arg_{float}$  is the summed demodulated signal. Because  $arg_{float}$  is limited to  $\pm 2 \cdot \pi$ , the range of the reachable index is:

$$index_{short} = \left[ -\frac{2 \cdot \pi}{step_{float}}, +\frac{2 \cdot \pi}{step_{float}} \right]$$

or expressed in terms of the constant `BUFFER_SIZE_SINE`:

$$index_{short} = [-4 \cdot (BUFFER\_SIZE\_SINE - 1), +4 \cdot (BUFFER\_SIZE\_SINE - 1)]$$

The index range remains the same in the fixed point realization. To calculate the index, it is better to use a multiplication instead of a division:

$$index_{short} = (\arg_{Q\_format} \cdot STEP\_INVERSE_{Q\_format}) >> (2 \cdot Q\_FORMAT)$$

The needed Q format (`Q_format`) is also calculated in the M-file. It depends on the index range (`BUFFER_SIZE_SINE`). The M-file delivers the shifting number of the Q format (`Q_FORMAT`). The shifting always rounded down to the next smaller integer value. In the case that the index is negative, the rounding is erroneous. That is why a negative index must be negated to become positive before shifting. After that the sign of the index can be changed to negative again.

Now the calculated index has to be prepared to access to the `sine_buffer` because the `sine_buffer` has only an index range from zero to `BUFFER_SIZE_SINE-1` (Listing 5.5 line 62 to 162). The index preparation will now be explained with an example. Figure 5.33 shows a positive sine wave with belonging arguments and indexes respectively. The fat printed part of sine wave represents the available sine values in `sine_buffer`. In Figure 5.34 the index preparation is explained pictorially. First an inquiry must be issued to find out in which area the index lies. The first picture shows the found index area with associated sine values. The following pictures show the new index areas with the expected values after the prior index manipulation. As described, there are two basic index manipulations. If the index is in the index area 1 the `sine_buffer` access can be done directly

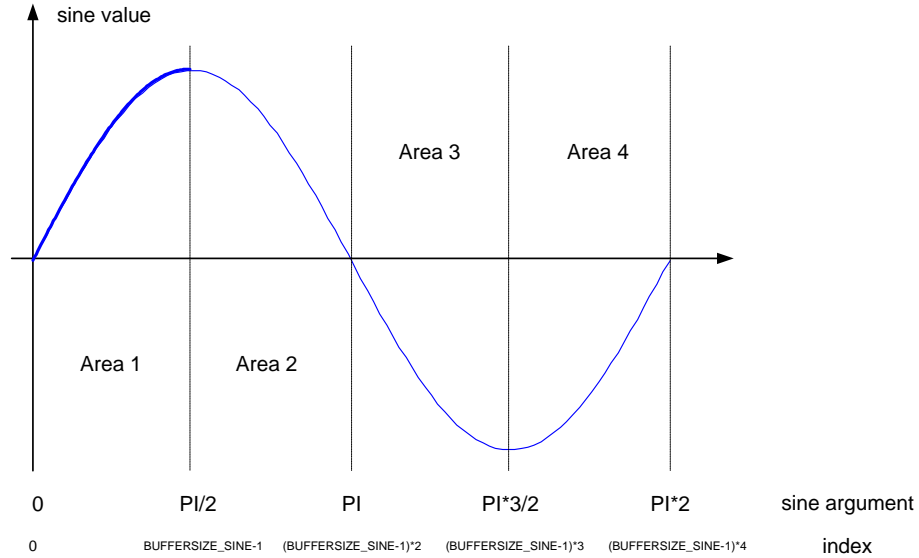


Figure 5.33.: Sine values with belonging indexes

with the calculated index. An index in area 2 needs both manipulations. For indexes in area 3, only the first index manipulation is necessary but the found value after `sine_buffer` access has to be multiplied with minus one. The same goes for indexes in area 4 with the exception that both index manipulations are also needed.

The same principle is applied to the negative sine wave as well as whole cosine wave.

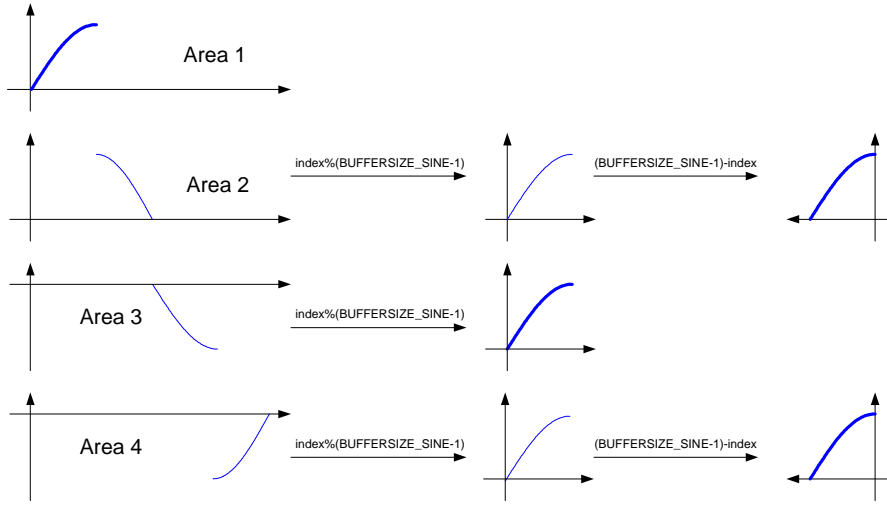


Figure 5.34.: Index manipulation

Listing 5.5: PLL demodulator

```

1 void pll_demodulate(short re_buffer[], short im_buffer[],
2   short out_buffer[])
3 {
4   int i;
5   int flag=0;
6   short demo; /*index for sine_buffer access*/
7   short index; /*value of demodulation*/
8
9   for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
10  {
11    /*__calculation of demodulated value, using Q15__*/
12    i_path=(re_buffer[i]*i_path)>>15;
13    q_path=(im_buffer[i]*q_path)>>15;
14    demo=q_path-i_path;
15    demo=(demo*P_PLL)>>15;
16    out_buffer[i]=(demo*GAIN_PLL)>>15;
17
18    /*__summation of demodulated signal, using Q3.12__*/
19    demo=demo>>3;
20    arg=arg+((demo*K_PLL)>>12);
21
22    /*__limitation of argument to 2*PI, using Q3.12__*/
23    if (arg>TWO_PI)
24    {
25      arg=arg-TWO_PI;
26    }
27    else if (arg<-TWO_PI)
28    {
29      arg=arg+TWO_PI;
30    }
31
32    /*__calculation of sine and cosine value of argument__*/
33    /*calculating of index for sine_buffer access, using
34      new Q format*/
35    index=arg>>REDUCTION;
36    if (index<0)
37    {
38      index=-index;
39      flag=1;
40    }
41
42    index=(index*STEP_INVERSE)>>(2*Q_FORMAT);
43    if (flag==1)
44    {
45      index=-index;
46      flag=0;
47    }
48    /*access on sine_buffer with index*/
49    if(index>=0)
50    {
51      if(index<(BUFFERSIZE_SINE-1))
52      {
53        /*sine value*/
54        i_path=sine_buffer[index];
55        /*cosine value*/
56        index=(BUFFERSIZE_SINE-1)-index;
57        q_path=sine_buffer[index];
58      }
59      else if (index<2*(BUFFERSIZE_SINE-1))
60      {
61        index=index % (BUFFERSIZE_SINE-1);
62        /*sine value*/
63        index=(BUFFERSIZE_SINE-1)-index;
64        i_path=sine_buffer[index];
65        /*cosine value*/
66        index=(BUFFERSIZE_SINE-1)-index;
67        q_path=-sine_buffer[index];
68      }
69      else if (index<3*(BUFFERSIZE_SINE-1))
70      {
71        index=index % (BUFFERSIZE_SINE-1);
72        /*sine value*/
73        i_path=-sine_buffer[index];
74        /*cosine value*/
75        index=(BUFFERSIZE_SINE-1)-index;
76        q_path=-sine_buffer[index];
77      }
78      else if (index<4*(BUFFERSIZE_SINE-1))
79      {
80        index=index % (BUFFERSIZE_SINE-1);
81        /*sine value*/
82        index=(BUFFERSIZE_SINE-1)-index;
83        i_path=sine_buffer[index];
84        /*cosine value*/
85        index=(BUFFERSIZE_SINE-1)-index;
86        q_path=sine_buffer[index];
87      }
88    }
89    else
90    {
91      index=index % (BUFFERSIZE_SINE-1);
92      /*sine value*/
93      i_path=sine_buffer[index];
94      /*cosine value*/
95      index=(BUFFERSIZE_SINE-1)-index;
96      q_path=sine_buffer[index];
97    }
98    if(index>-(BUFFERSIZE_SINE-1))
99    {
100     index=-index;
101   }
102 }

```

```

118         /*sine value*/
119         i_path=-sine_buffer[index];
120         /*cosine value*/
121         index=(BUFFERSIZE_SINE-1)-index;
122         q_path=sine_buffer[index];
123     }
124     else if(index>-2*(BUFFERSIZE_SINE-1))
125     {
126         index=-(index % (BUFFERSIZE_SINE-1));
127         /*sine value*/
128         index=(BUFFERSIZE_SINE-1)-index;
129         i_path=-sine_buffer[index];
130         /*cosine value*/
131         index=(BUFFERSIZE_SINE-1)-index;
132         q_path=-sine_buffer[index];
133     }
134     else if (index>-3*(BUFFERSIZE_SINE-1))
135     {
136         index=-(index % (BUFFERSIZE_SINE-1));
137         /*sine value*/
138         i_path=sine_buffer[index];
139         /*cosine value*/
140         index=(BUFFERSIZE_SINE-1)-index;
141         q_path=-sine_buffer[index];
142     }
143     else if (index>-4*(BUFFERSIZE_SINE-1))
144     {
145         index=-(index % (BUFFERSIZE_SINE-1));
146         /*sine value*/
147         index=(BUFFERSIZE_SINE-1)-index;
148         i_path=sine_buffer[index];
149         /*cosine value*/
150         index=(BUFFERSIZE_SINE-1)-index;
151         q_path=sine_buffer[index];
152     }
153     else
154     {
155         index=-(index % (BUFFERSIZE_SINE-1));
156         /*sine value*/
157         i_path=-sine_buffer[index];
158         /*cosine value*/
159         index=(BUFFERSIZE_SINE-1)-index;
160         q_path=sine_buffer[index];
161     }
162 }
163 }
164 }

```

## Interpolated Lookup Table

To get more precise sine values, an interpolation can be carried out between two saved sine values. As mentioned the Q format of the index depends on the chosen `BUFFERSIZE_SINE`. An index can be interpreted as a floating-point in the used Q format. Without interpolation, only the integer parts are considered. With interpolation the fraction is also included to calculate the sine value. The fraction is stored in the variable `rest`. How many digits of the fraction are known, depends on `Q_FORMAT`. A linear interpolation can be done with the fraction. For that, an extra buffer has to be generated. This buffer is also prepared in a M-file and later integrated to the C code. It is known as the `sine_grad_buffer` and it contains the differences of two adjacent values of the `sine_buffer`. Figure 5.35 explains how the linear interpolation is implemented. The smaller the step between

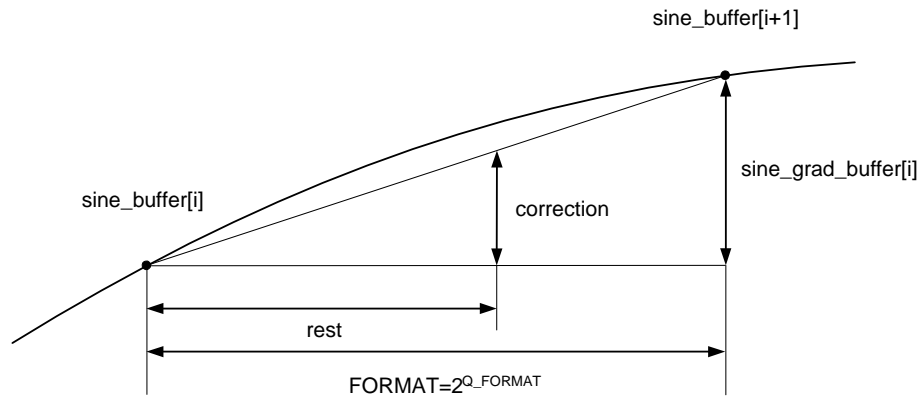


Figure 5.35.: Linear interpolation

the saved sine values, the more accurate the sine value is approximated.

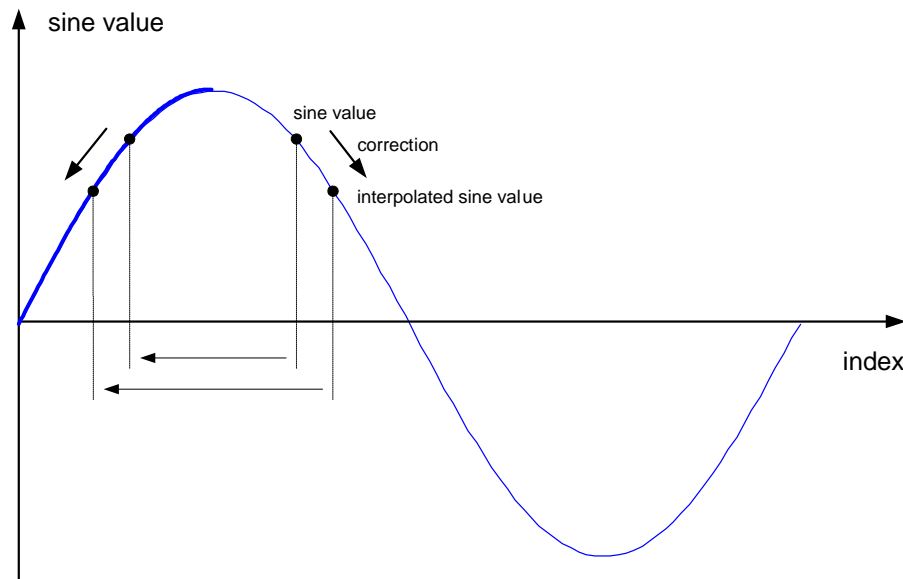
$$\frac{\text{correction}}{\text{rest}} = \frac{\text{sine\_grad\_buffer}[i]}{\text{FORMAT}}$$

$$\text{correction} = (\text{sine\_grad\_buffer}[i] \cdot \text{rest}) \gg Q\_FORMAT$$

The interpolated sine value is:

$$\text{interpolated\_sine\_value} = \text{sine\_buffer}[i] + \text{correction}$$

Listing 5.6 shows the part of `sine_buffer_access` with the necessary new instructions. In Listing 5.6 line 14 the fraction of the index is saved in the variable `rest` before shifting. In Listing 5.6 the needed correction is computed due to `rest` and `sine_grad_buffer` access (for example line 30). Attention has been paid to which sign the correction is added to the `sine_buffer` value. The sign of `rest` and the values of `sine_grad_buffer` are always positive. Thus only the sign of the correction must be manipulated. Figure 5.36 and 5.37 show two examples with different signs of the correction.



*Figure 5.36.: Sign of correction changes*

*Listing 5.6: Interpolated Lookup Table*

```

1 /*__calculation of sine and cosine value of argument__*/
2
3     /*calculating of index for sine_buffer access, using new Q
4         format*/
5
6     index=arg>>REDUCTION;
7
8     if (index<0)
9     {
10         index=-index;
11         flag=1;
12     }
13
14     index=(index*STEP_INVERSE)>>Q_FORMAT;
15     rest=index%FORMAT;
16     index=index>>Q_FORMAT;
17
18     if (flag==1)
19     {
20         index=-index;
21         flag=0;
22     }
23
24     /*access on sine_buffer with index*/
25
26     if(index>=0)
27     {
28         if(index<(BUFFERSIZE_SINE-1))
29         {
30             /*sine value*/
31             correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
32             /*positive*/
33             i_path=sine_buffer[index]+correction;
34             /*cosine value*/

```

```

33     index=(BUFFERSIZE_SINE-1)-index;
34     correction=- (sine_grad_buffer[index]*rest)>>Q_FORMAT;
35     /*negative*/
36     q_path=sine_buffer[index]+correction;
37 }
38 else if (index<2*(BUFFERSIZE_SINE-1))
39 {
40     index=index % (BUFFERSIZE_SINE-1);
41     /*sine value*/
42     index=(BUFFERSIZE_SINE-1)-index;
43     correction=- (sine_grad_buffer[index]*rest)>>Q_FORMAT;
44     /*negative*/
45     i_path=sine_buffer[index]+correction;
46     /*cosine value*/
47     index=(BUFFERSIZE_SINE-1)-index;
48     correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
49     /*positive*/
50     q_path=- (sine_buffer[index]-correction);
51 }
52 else if (index<3*(BUFFERSIZE_SINE-1))
53 {
54     index=index % (BUFFERSIZE_SINE-1);
55     /*sine value*/
56     correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
57     /*positive*/
58     i_path=- (sine_buffer[index]-correction);
59     /*cosine value*/
60     index=(BUFFERSIZE_SINE-1)-index;
61     correction=- (sine_grad_buffer[index]*rest)>>Q_FORMAT;
62     /*negative*/
63     q_path=- (sine_buffer[index]+correction);
64 }
65 else if (index<4*(BUFFERSIZE_SINE-1))
66 {
67     index=index % (BUFFERSIZE_SINE-1);

```

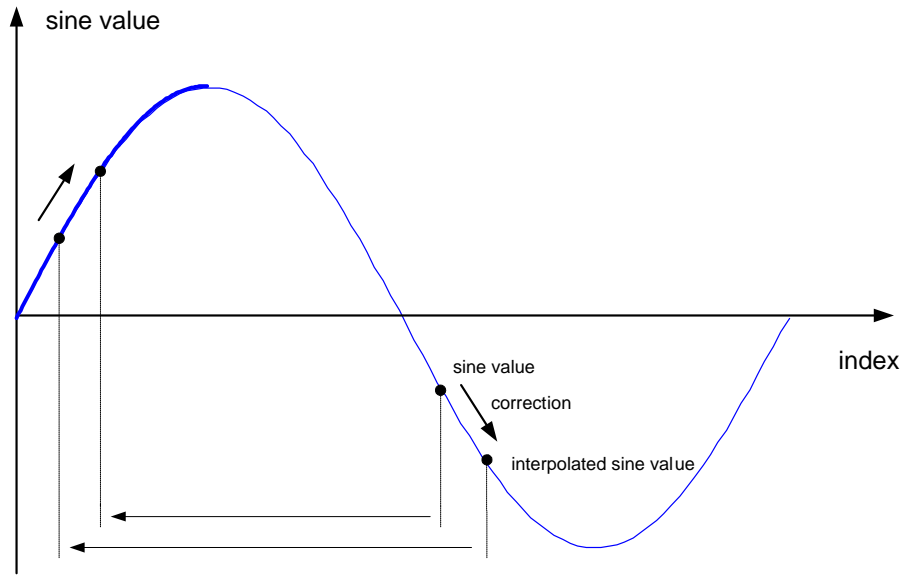


Figure 5.37.: Sign of correction does not change

```

63      /*sine value*/
64      index=(BUFFERSIZE_SINE-1)-index;
65      correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
66      /*negative*/
67      i_path=-(sine_buffer[index]+correction);
68      /*cosine value*/
69      index=(BUFFERSIZE_SINE-1)-index;
70      correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
71      /*positive*/
72      q_path=sine_buffer[index]+correction;
73  }
74  else
75  {
76      index=index % (BUFFERSIZE_SINE-1);
77      /*sine value*/
78      correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
79      /*positive*/
80      i_path=sine_buffer[index]+correction;
81      /*cosine value*/
82      index=(BUFFERSIZE_SINE-1)-index;
83      correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
84      /*negative*/
85      q_path=sine_buffer[index]+correction;
86  }
87  }
88  else
89  {
90      if(index>-(BUFFERSIZE_SINE-1))
91      {
92          index=-index;
93          /*sine value*/
94          correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
95          /*positive*/
96          i_path=-(sine_buffer[index]+correction);
97          /*cosine value*/
98          index=(BUFFERSIZE_SINE-1)-index;
99          correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
100         /*negative*/
101         q_path=sine_buffer[index]+correction;
102     }
103     else if(index>-2*(BUFFERSIZE_SINE-1))
104     {
105         index=-(index % (BUFFERSIZE_SINE-1));
106         /*sine value*/
107         correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
108         /*positive*/
109         i_path=sine_buffer[index]+correction;
110         /*cosine value*/
111         index=(BUFFERSIZE_SINE-1)-index;
112         correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
113         /*negative*/
114         q_path=-(sine_buffer[index]+correction);
115     }
116     else if(index>-4*(BUFFERSIZE_SINE-1))
117     {
118         index=-(index % (BUFFERSIZE_SINE-1));
119         /*sine value*/
120         correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
121         /*positive*/
122         i_path=sine_buffer[index]+correction;
123         /*cosine value*/
124         index=(BUFFERSIZE_SINE-1)-index;
125         correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
126         /*negative*/
127         q_path=-(sine_buffer[index]+correction);
128     }
129     else
130     {
131         index=-(index % (BUFFERSIZE_SINE-1));
132         /*sine value*/
133         correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
134         /*positive*/
135         i_path=-(sine_buffer[index]+correction);
136         /*cosine value*/
137         index=(BUFFERSIZE_SINE-1)-index;
138         correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
139         /*negative*/
140         q_path=sine_buffer[index]+correction;
141     }
142 }
143 }
144 }

```



### 5.3.5. Out Filter

It was not possible to design a stable bandpass filter in fixed-point as in the floating-point implementation. Therefore a low-pass filter was designed first. It includes the downsampling in the last cascade. Then a high-pass filtering is applied at the lower frequency.

#### Low-Pass

The MATLAB file *lowpass\_design\_noisefilter.m* is used to design the low-pass filter. The specifications are given in Table 5.5. This results in a 13th order filter, which is again splitted into one first

**Table 5.5.:** Filter specifications low-pass filter

pass-band edge frequency	3500 Hz
stop-band edge frequency	4000 Hz
pass-band ripple	2dB
stop-band ripple	20 dB

and six second order cascades. To reduce the computing load, the second downsampling is included in the last cascade of the filter as described in Section 5.3.2. The signal is again additionally down scaled by 0.8 to avoid overflows in the filter.

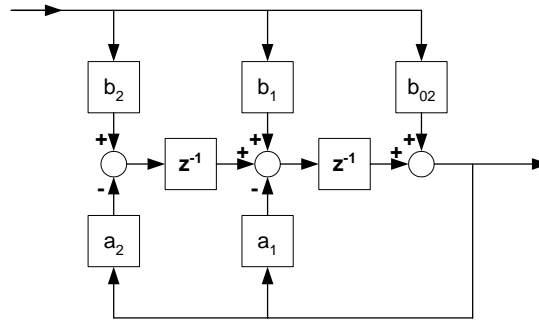
#### High-Pass

The MATLAB file *highpass\_design\_noisefilter.m* is used to design the high-pass filter. The specifications are given in Table 5.6. This filter implemented in the direct from II would have to include too

**Table 5.6.:** Filter specifications high-pass filter

pass-band edge frequency	300 Hz
stop-band edge frequency	50 Hz
pass-band ripple	0.1 dB
stop-band ripple	40 dB

much down scaling to avoid overflows. Therefore the implementation structure was changed to the transposed form. This structure has the advantage, that the delay states range is quite close to the range of the output signal. The splitting can be done by using  $[sos, g] = zp2sos(z, p, k, )$



**Figure 5.38.:** Block diagram transposed filter structure

without the specific scaling argument. Because the transposed form is more sensitive to overflows

an additional down scaling of 0.5 is applied prior to the high-pass. The total down scaling is still much smaller than the one which would be needed by the direct from II. After the high-pass the signal is multiplied by 2 to undo the additional down scaling.

## 5.4. Optimization

### 5.4.1. Adaptive Seeking of Carrier Angular Frequency

A general problem of FM demodulation is a DC offset due to inexact angular frequency of the quadrature mixer  $\omega_M$ .  $\omega_M$  depends on the FM carrier angular frequency  $\omega_T$  and the sample frequency  $\omega_A$  (see Eq.5.1). Because the FM modulation is done with analog components  $\omega_T$  is not a constant, it fluctuates. Thus  $\omega_M$  is also fluctuating. This drift causes the described DC offset in the demodulated message signal. In the analog FM demodulation this error can be suppressed with a high-pass filter after demodulation. That can also be done in the digital FM demodulation. A smarter solution is the adaptive seeking of carrier angular frequency. To implement adaptive control algorithms it is necessary to describe mathematically the error which should be corrected. For that reason the Mixed demodulator is preferred to the PLL demodulator. The PLL demodulator is more difficult to describe mathematically, due to its non linear closed loop.

Figure 5.39 shows a system with an adaptive behavior. The following equations explains how

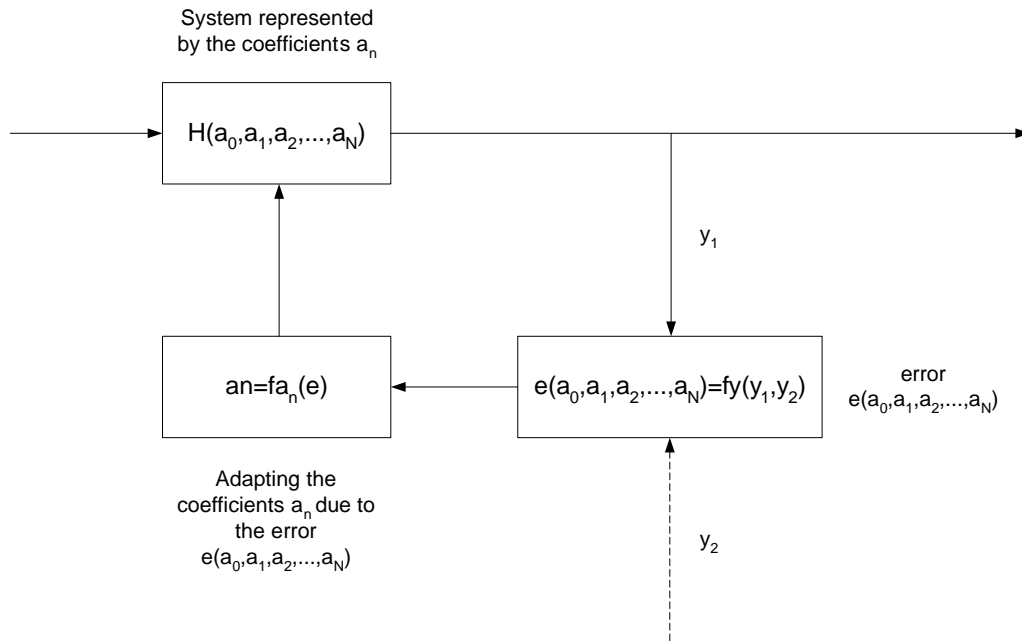


Figure 5.39.: Adapted System

the DC offset error results because of the inexact  $\omega_M$ .  $\omega_{Me}$  is the actual angular frequency and  $\omega_{Ms}$  the supposed angular frequency of the quadrature mixer, the I and the Q signal of the quadrature mixer are:

$$\frac{A}{2} \cos((\omega_{Me} - \omega_{Ms}) \cdot T \cdot n + \Delta F \cdot 2 \cdot \pi \cdot T \cdot \sum_{i=0}^n s_N(n)) = \frac{A}{2} \cos(\eta(n))$$

and:

$$\frac{A}{2} \sin((\omega_{Me} - \omega_{Ms}) \cdot T \cdot n + \Delta F \cdot 2 \cdot \pi \cdot T \cdot \sum_{i=0}^n s_N(n)) = \frac{A}{2} \sin(\eta(n))$$

Compare with Section 3.2.2.

After the demodulation with the mixed demodulator the demodulated signal is:

$$s_D(n) = \frac{\eta(n) - \eta(n-1)}{T \cdot \Delta F \cdot 2 \cdot \pi} = \frac{\omega_{Me} - \omega_{Ms}}{\Delta F \cdot 2 \cdot \pi} + s_N(n)$$

Compare with Section 3.3.4, especially with Eq.3.19.

Now the DC offset error is described due to  $\omega_{Ms}$ :

$$e(\omega_{Ms}) = \frac{\omega_{Me} - \omega_{Ms}}{\Delta F \cdot 2 \cdot \pi}$$

Therefore  $\omega_{Me}$  must be:

$$\omega_{Me} = \omega_{Ms} + e(\omega_{Ms}) \cdot \Delta F \cdot 2 \cdot \pi$$

For that,  $e(\omega_{Ms})$  must be known. A DC offset of a number sequence is calculated as:

$$e(\omega_{Ms}) = \frac{1}{N} \sum_{i=0}^{N-1} s_D(i)$$

The larger N is, as exacter the error will be computed and as less computing time is needed. N also determines the period of the adaptive algorithm. Thus the upper limit of N is given by the speed of the carrier angular frequency change.

$\omega_{Me}$  is used to calculate the new coefficients of the IIR filter, which is used for sine and cosine generation:

$$\begin{aligned} \text{bc2} &= -\cos(\omega_{Me} \cdot T) \\ \text{bs2} &= \sin(\omega_{Me} \cdot T) \\ \text{asc2} &= -2 \cdot \cos(\omega_{Me} \cdot T) \end{aligned}$$

Compare with Section 5.2.1.

If these coefficients are regularly computed the offset error of the demodulated signal  $s_D(n)$  will disappear, due to the adaptive seeking of the carrier angular frequency.

Figure 5.40 gives an overview of the structure of the adaptive filter. The effort to implement the calculation of the IIR Filter coefficients in fixed-point would be very big. Therefore the implementation is only applied to the floating-point implementation.

The adaptive seeking of the carrier angular frequency brings some advantages: The output filter can be reduced to a low-pass, because the DC offset error is corrected. It will improve the signal quality, if the carrier frequencies of the transmitter and receiver do not match exactly.

### 5.4.2. Time Optimization

Texas Instruments recommends a three phase code development flow for the C6000 shown in Figure 5.41 citeProgGuide.

Phase one is to translate algorithms into C Code, compile and profile the implementation to verify the correct performance. This phase has been done in the previous section of this chapter. Phase two and three are matter of this section. Phase two is further divided into two steps, reducing computing load and using compiler specific optimizations.

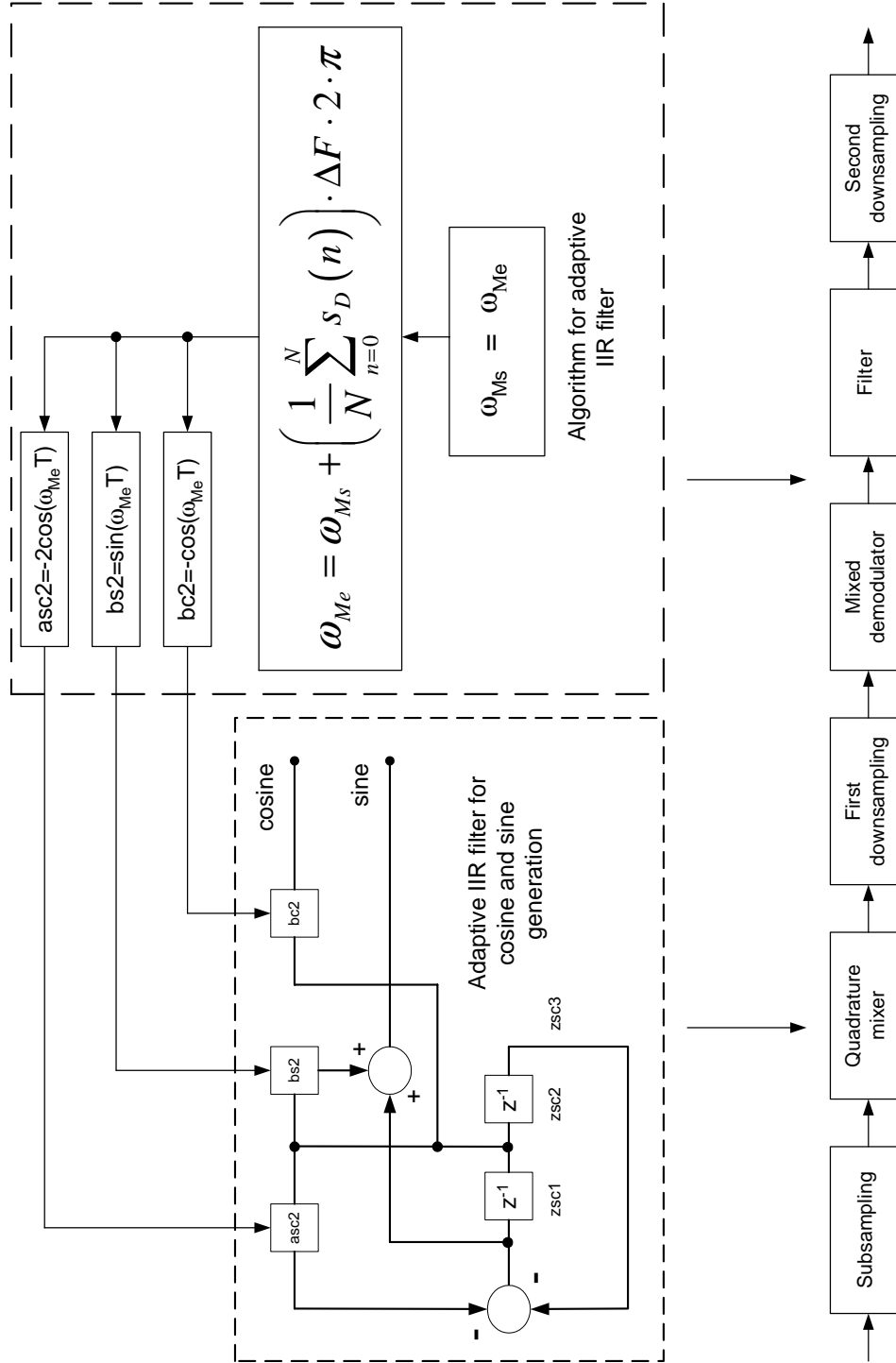


Figure 5.40.: Adaptive seeking of carrier angular frequency

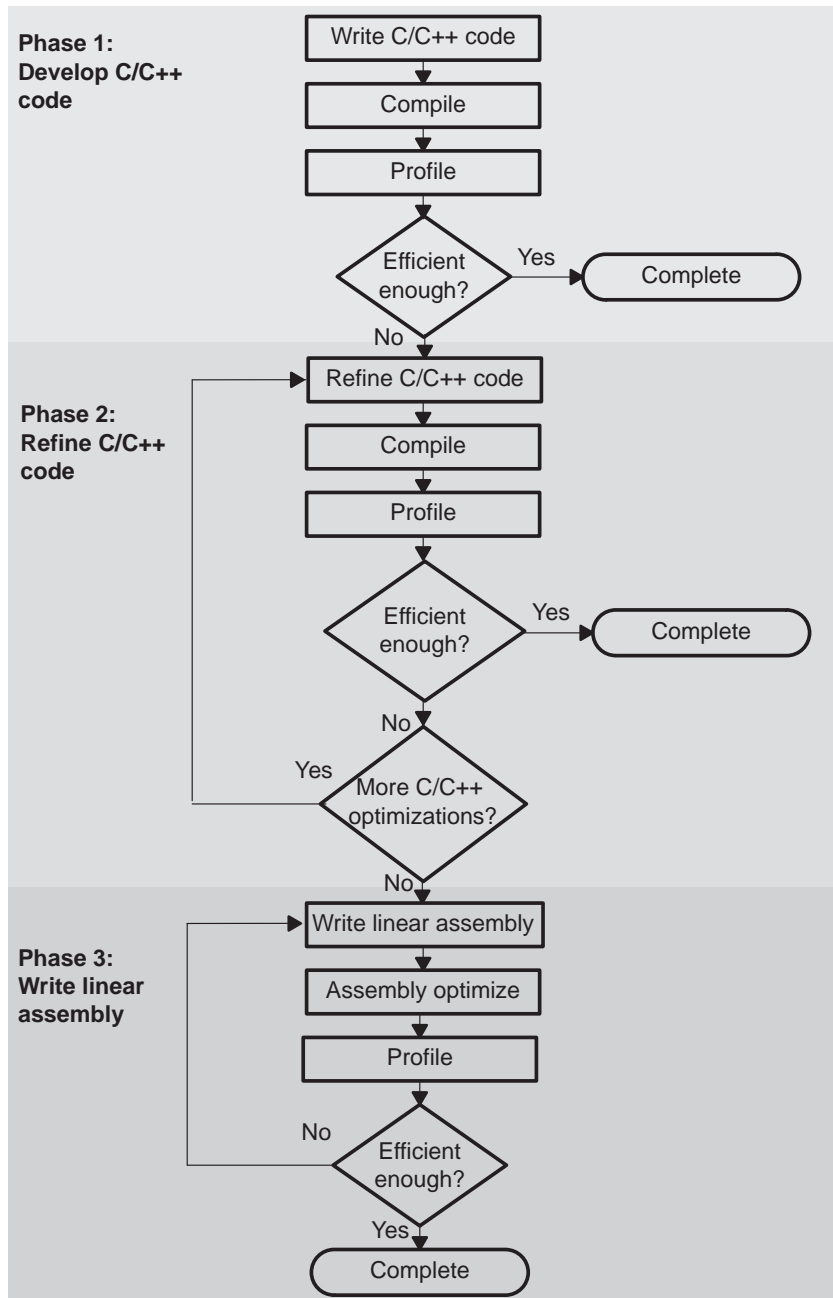


Figure 5.41.: development flow

### Reducing Computing Load (Phase 2 - Step 1)

The implementation is reviewed to reduce the computing load of the implemented algorithms. The following changes are applied:

- The scaling of the low-pass filter in the quadrature mixer is included in the output of oscillators of the quadrature mixer. Thus the two numerators of the oscillator IIR filter (see Eq. 5.3

and Eq. 5.4) are multiplied by the scale factor of the low-pass filter.

- The scaling of the low-pass filter in the out filter is included in the arctangent lookup table. All the values in the arctangent lookup table are multiplied with the scaling factor. Thus the lookup table of the optimized implementation fulfils the equation:

$$out = lp\_scale \cdot \frac{1}{T \cdot k_{FM}} \cdot \arctan(in)$$

- The scaling of high-pass filter in the out filter is included last cascade's forward feed of the low-pass filter in the out filter . Hence the numerator of the last cascade is multiplied with the scale factor.

This changes reduce the used multiplications and therefore the computing load the gain in processing speed is measured and commented in Chapter 6.4.

### Compiler Specific Optimizations (Phase 2 - Step 2)

The C6000 Compiler provides several keywords, pragma directives , and intrinsics to pass information to the compiler and optimizer.

**Pragma** directives tell the compile how to treat a certain function, object, or section of code.

**Intrinsics** are special functions that map directly to inlined C67x instructions or tell the optimizer that a expression is true to give hints to the optimizer, to optimize the C code quickly.

**The Restrict Keyword** To help the compiler determine memory dependencies, pointers, references, or arrays can be qualified with the *restrict* keyword. The restrict keyword is a type qualifier. Its use represents a guarantee by the programmer that within the scope of the pointer declaration the object pointed to can be accessed only by that pointer. This practice helps the compiler optimize certain sections of the code because aliasing information can be more easily determined. The *restrict* keyword is applied to all the function definitions of the implementation, because all the buffers passed to the functions are not aliasing.

```
void quad_mix(short in_buffer[restrict] , short re_buffer[restrict] , short im_buffer[restrict]
    )

void mixed_demodulate(short re_buffer[restrict] , short im_buffer[restrict] , short out_buffer[
    restrict])

void pll_demodulate(short re_buffer[restrict] , short im_buffer[restrict] , short out_buffer[
    restrict])

void out_filter(short out_buffer[restrict] , short out_buffer_down[restrict])
```

**The MUST\_ITERATE Pragma** The *MUST\_ITERATE* pragma specifies to the compiler certain properties of a loop. The maximum and the minimum trip count can be specified for a loop. In the implementation all loop counts are constants and therefore the compiler can retrieve the needed information without the explicit declaration of the *MUST\_ITERATE* pragma.

**The DATA\_MEM\_BANK Pragma** The `DATA_MEM_BANK` pragma aligns a symbol or variable to a specified C6000 internal data memory bank boundary. All the buffers used are aligned that no memory stall occurs (two access to the same memory bank).

```
#pragma DATA_MEM_BANK (in_buffer_A, 0);
short in_buffer_A[BUFFER_SIZE_INPUT];
#pragma DATA_MEM_BANK (in_buffer_B, 0);
short in_buffer_B[BUFFER_SIZE_INPUT];

#pragma DATA_MEM_BANK (re_buffer, 2);
short re_buffer[BUFFER_SIZE_INPUT];
#pragma DATA_MEM_BANK (im_buffer, 4);
short im_buffer[BUFFER_SIZE_INPUT];

#pragma DATA_MEM_BANK (out_buffer, 0);
short out_buffer[BUFFER_SIZE_DEMODULATOR];
#pragma DATA_MEM_BANK (out_buffer_A, 6);
short out_buffer_A[BUFFER_SIZE_OUTPUT];
#pragma DATA_MEM_BANK (out_buffer_B, 6);
short out_buffer_B[BUFFER_SIZE_OUTPUT];
```

This also assures that the buffers are word-aligned.

**The \_nassert Intrinsic** The `_nassert` intrinsic generates no code. It tells the optimizer that the expression declared with the assert function is true. This gives a hint to the compiler as to what optimizations might be valid.

In the implementation we use the `_nassert` intrinsic to tell the compiler that the buffers passed to the functions are word-aligned. The `WORD_ALIGNED` macro is defined as followed

```
#define WORD_ALIGNED(x) (_nassert(((int)(x) & 0x3) == 0))
```

and is used in the functions.

```
WORD_ALIGNED(XX_buffer);
```

All this information helps the compiler and optimizer to perform better loop unrolling and software pipelining to parallelize loop iterations, Section 6.4 outlines the time improvements due to these optimizations.

**Software pipelining** is a technique used to schedule instructions from a loop so that multiple iterations execute in parallel. The parallel resources on the DSP make it possible to initiate a new loop iteration before previous iterations finish. The goal of software pipelining is to start a new loop iteration as soon as possible.

### Writing Linear Assembly (Phase 3)

In this phase, the time-critical areas are extracted from the C code and rewritten in linear assembly. The assembly optimizer is used to optimize this code. The code is written without being concerned with the pipeline structure or with assigning registers, with the optimizer will handle.

This phase is quite time intensive and therefore only the quadrature mixer is rewritten in linear assembly. The mixer algorithm runs at the highest frequency and uses the most computing time. The rewriting should show the improvement against the optimized C implementations.

The complete `quad_mix` function is transformed to linear assembly. Linear assembly provides directives to write a C callable assembly function. The function has to start with the directive `.cproc` and ends with `.endproc`. Hence the structure of the quadrature mixer function is

```
_quad_mix: .cproc in_s , re_b_s , im_b_s
...
.endproc
```

The time improvement against the C optimized code is rather small compared to the time invested for the rewriting. The exact timings are measured in Chapter 6.4. Also the clarity of the code sinks and makes it harder for reuse.



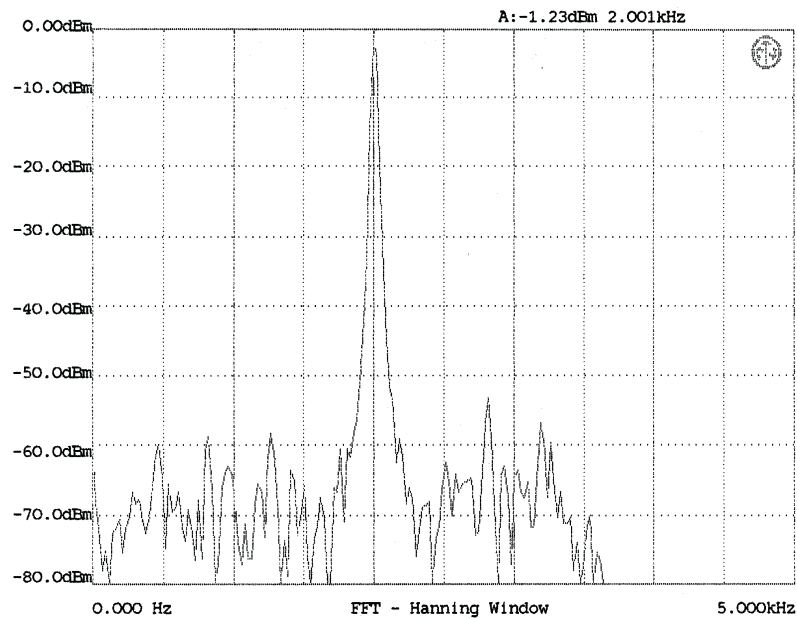


## 6. Tests And Results

To test and verify the implementations various measurements are carried out and described below. The instruments used are described in Appendix A.3

### 6.1. Spectrum

The spectrum of a signal gives a general statement about the signal. The Neutrik A2-D can calculate the Fast Fourier Transformation (FFT) of the measured signal.



*Figure 6.1.: FFT print of demodulated signal*

Figure 6.1 shows the FFT print of the demodulated signal with fixed-point mixed demodulator, direct lookup. The spectrum of the other implemented algorithms are quite similar. Compared to Figure 6.2, which shows the spectrum of a pure sine signal generated with the DSP, it can be seen that the noise floor is mainly introduced by the demodulation algorithm and not by the used devices. Figure 6.3 shows the spectrum of the same signal used in Figure 6.1 over the maximal range the A2-D can handle. The Noise reduction between 3200 Hz and 4000 Hz is introduced by the DAC interpolation filter. The noise above 4000 Hz is the noise level introduced by the devices. This noise level is more than 20dB lower than the noise level introduced by the demodulation algorithms and will therefore not distort the following measurements.

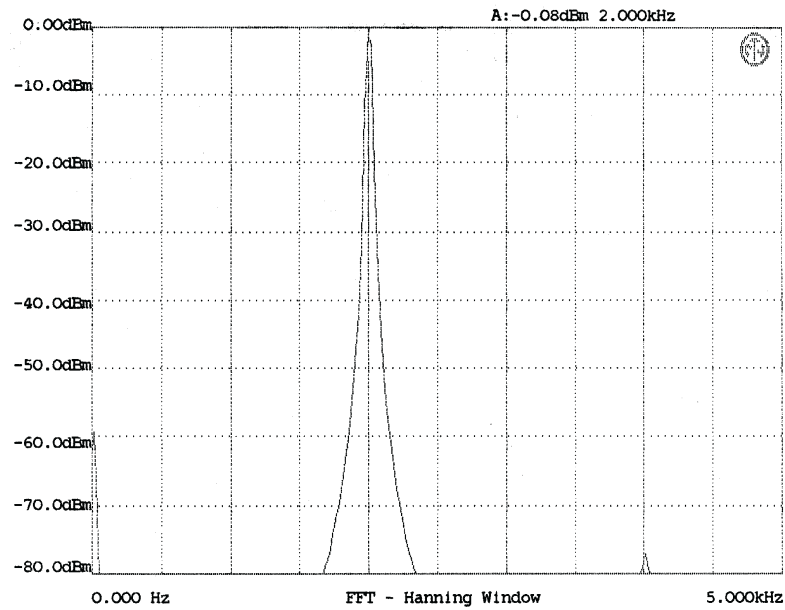


Figure 6.2.: FFT print of DSP generated sine wave

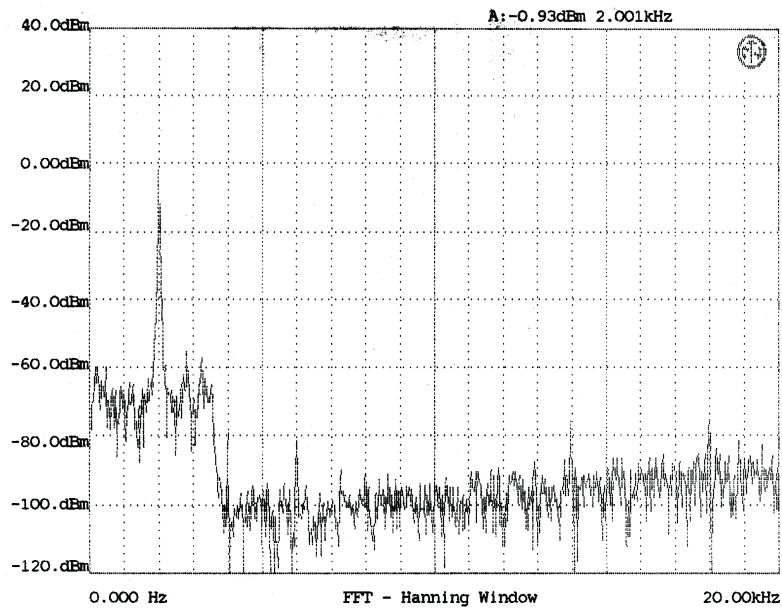


Figure 6.3.: FFT print of demodulated signal for the range of 0 to 20kHz

## 6.2. Signal Quality (SINAD)

In the simulations (Chapter 4) the SINAD was measured to make a statement about the signal quality. Therefore the SINAD is also measured for the DSP implementation.

The Neutrik A2-D has a THD+N Function, which measures the total Harmonic Distortion and noise by inserting a band-reject (notch) filter into the signal path (Figure 6.4). The THD+N value is

calculated according to Eq. 6.1.

$$THD + N = \frac{(Distortion + Noise)}{Signal + (Distortion + Noise)} \quad (6.1)$$

This definition is equal to the definition of the SINAD (see Eq. C.2 in the appendix) used in the

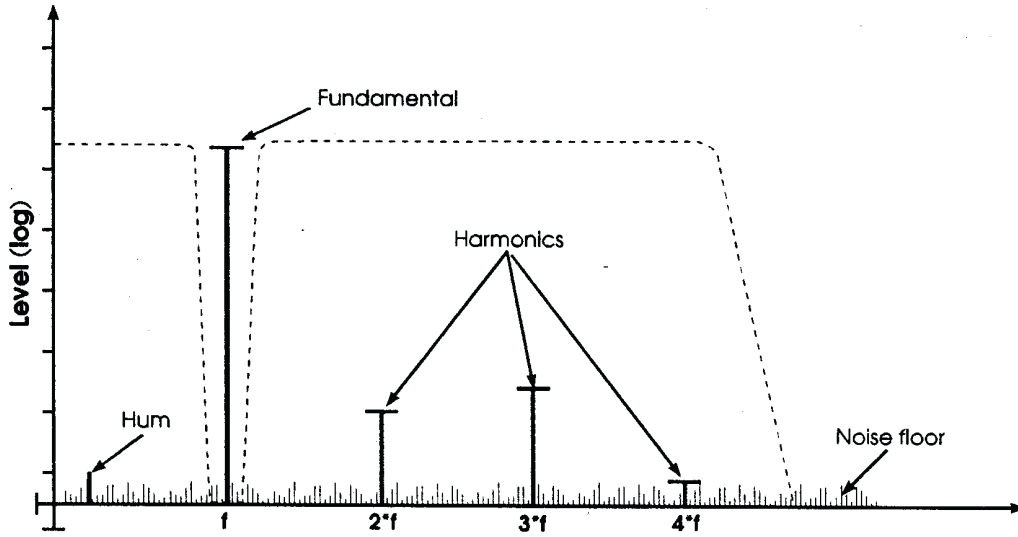


Figure 6.4.: Principle of THD+N Measurement

simulations. Hence the results of the THD+N Function are further referenced as SINAD to keep the same reference to the simulation.

### 6.2.1. Mixed Demodulator

The quality of the output signal depends a lot on the accuracy of the implementation of the arctangent function. First the implementation with the direct table lookup for the arctangent function is measured with different table sizes. The input FM signal is generated with the HP function generator (Appendix A.3.1), with the following parameters:

Carrier frequency $f_T$	10.7 MHz
Frequency deviation $\Delta F$	3000 Hz
Message signal frequency	100-3700 Hz

Figure 6.5 shows the plot of the SINAD measured for tables' sizes of 32, 64, 128, 256, and 512. It shows that a table size greater than 128 does not increase the signal quality much. Thus the table size for the direct lookup table is set to 128.

Second the implementation with the interpolation is measured with different table sizes and the same input signal parameters. Figure 6.6 shows the plot of the SINAD measured for tables' sizes of 8, 16, and 32. It shows that a table size 16 is the most optimal.

Figure 6.7 shows the comparison of the two implemented versions of the arctangent function and the floating-point implementation. The floating-point implementation uses the arctangent

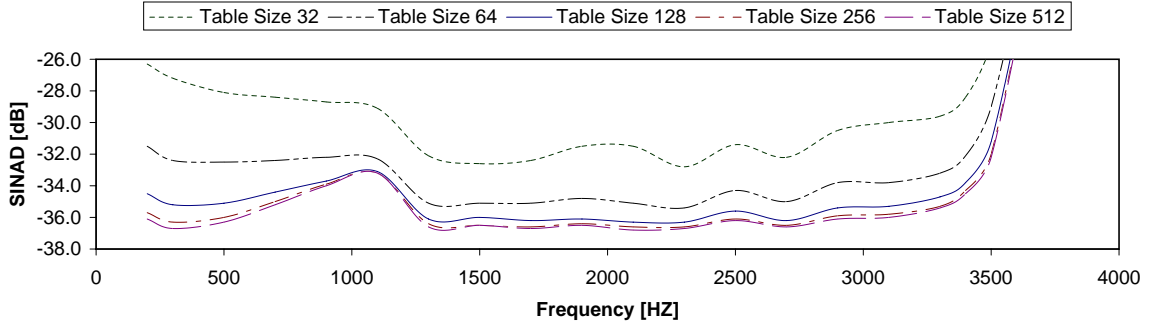


Figure 6.5.: SINAD of fixed-point mixed demodulator with various table sizes for direct lookup

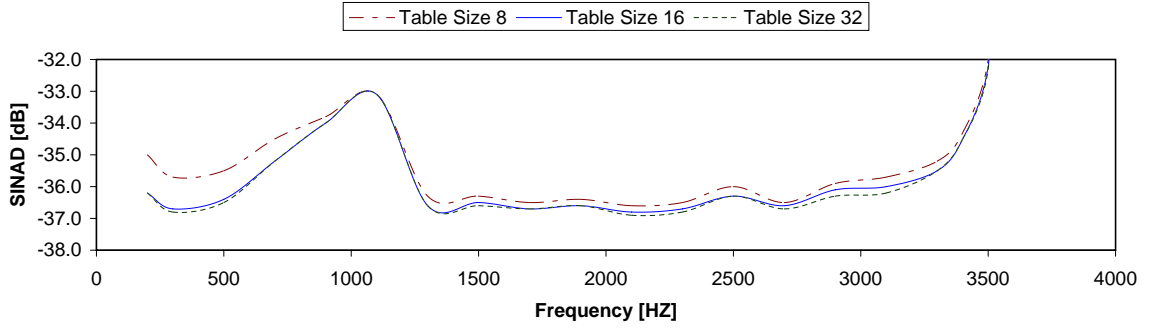


Figure 6.6.: SINAD of fixed-point mixed demodulator with various table sizes for interpolated lookup

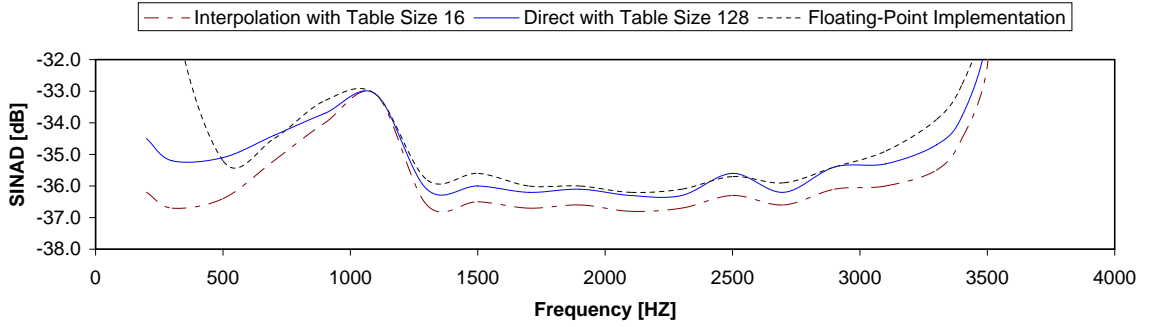


Figure 6.7.: SINAD comparison of direct and interpolated lookup for fixed-point mixed demodulator and the floating-point implementation

function of the standard math library, which provides several trigonometric, exponential, and hyperbolic math functions. The direct lookup uses a lookup table with 128 values and the interpolated lookup uses two tables with 16 values in each of them. It shows that the interpolated version achieves better results although it needs less storage. This advantage is paid by the additional computing time used to perform the interpolation.

### 6.2.2. PLL Demodulator

The signal quality of the PLL demodulator depends on the implementation of the sine and cosine function used in the loop and the constants  $P_{PLL}$  and  $K_{PLL}$ . First the implementation with the direct lookup is measured for different table sizes. The input FM signal is generated with the HP function generator (Appendix A.3.1), with the following parameters:

Carrier frequency $f_T$	10.7 MHz
Frequency deviation $\Delta F$	3000 Hz
Message signal frequency	100-3700 Hz

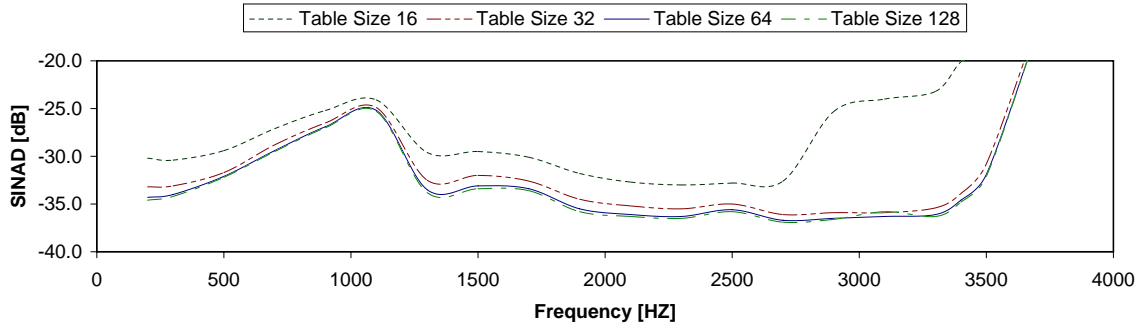


Figure 6.8.: SINAD of fixed-point PLL demodulator with various table sizes for direct lookup

Figure 6.8 shows the plot of the SINAD measured for tables' sizes of 16, 32, 64, 128. It shows that a table size of 64 is the most appropriate for the direct lookup implementation.

Second the implementation with the interpolation is measured with different table sizes and the same input signal parameters. Figure 6.9 shows the plot of the SINAD measured for table sizes of

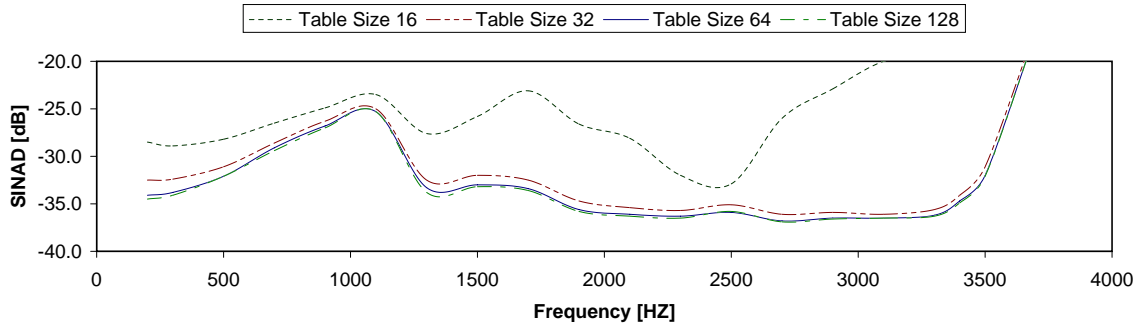


Figure 6.9.: SINAD of fixed-point PLL demodulator with various table sizes for interpolated lookup

16, 32, 64, and 128.

Figure 6.10 shows the comparison of the direct lookup, the interpolated lookup and the floating-point implementation which uses the math library functions. The direct lookup uses a lookup table with 64 values and the interpolated lookup uses two tables with 32 values in each of them. It shows that the interpolated version is not achieving a better results although it uses the same amount of storage and even more computing time. A general problem of a linear sine interpolation is described pictorially in Figure 6.11. The bigger the step between two saved sine values is chosen,

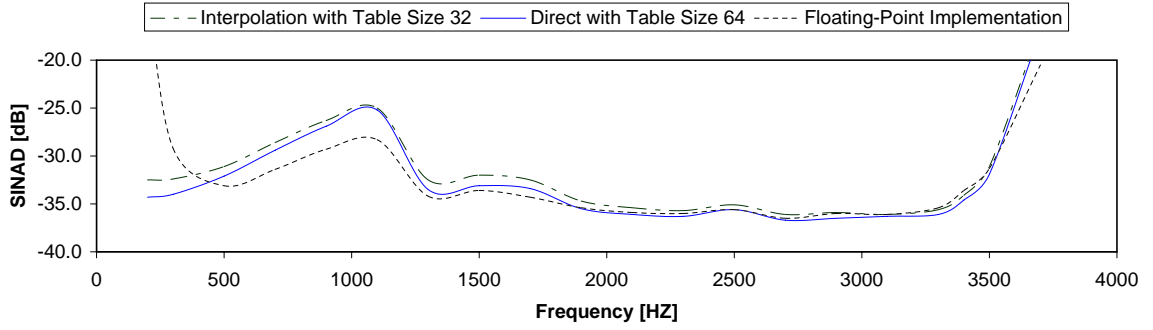


Figure 6.10.: SINAD comparison of direct and interpolated lookup for fixed-point PLL demodulator and the floating-point implementation

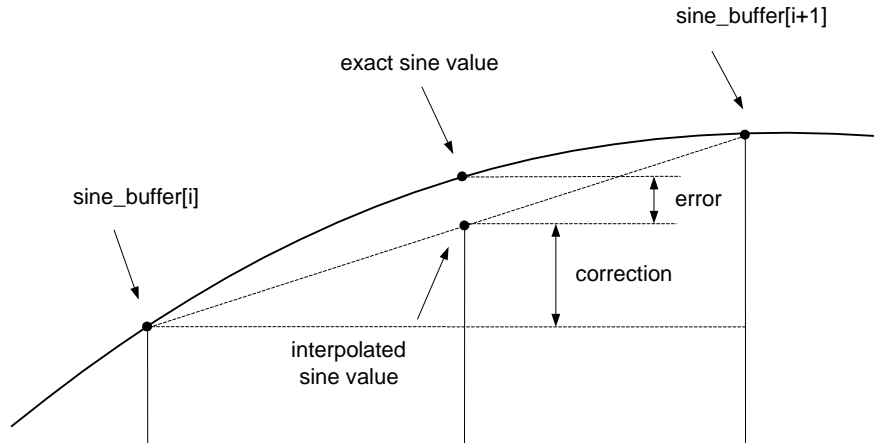


Figure 6.11.: Error of linear interpolation

the bigger is the error of interpolation. After the careful investigation, it is found that this fact is not responsible for the less than expected improvement. The main reason is that the PLL output is not directly dependent on the sine and cosine values. Thus an exact sine and cosine calculation with an interpolation does not bring along the expected improvement. Therefore the direct lookup implementation is the best choice and will be used.

### 6.2.3. Comparison of Mixed and PLL Demodulator

Figure 6.12 compares the different demodulation implementations. The floating- and the fixed-point implementation of the mixed demodulator show nearly the same behavior. The fixed-point implementation of the PLL has shown a worse signal quality than that of the floating-point implementation for lower frequencies. In general the PLL shows a worse behavior for low frequencies than that of the mixed demodulator, especially around 1000 Hz. All the implementations show a slight decrease of signal quality around the frequency of 1000 Hz.

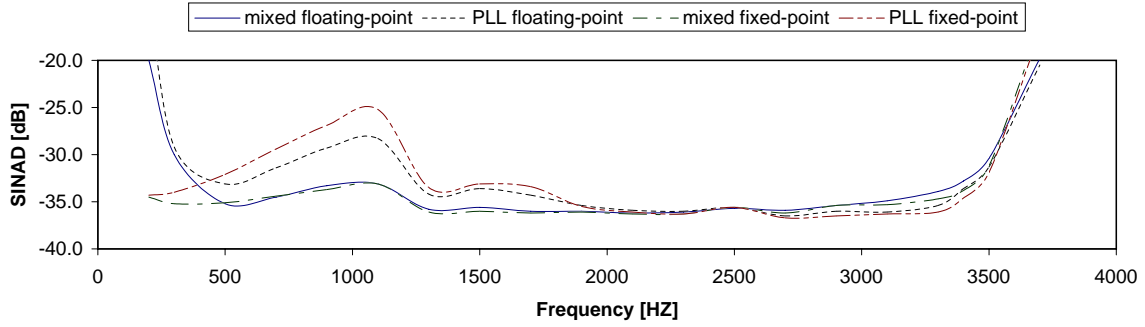


Figure 6.12.: SINAD comparison of different demodulator implementations

### 6.3. Robustness (S/N)

To test the robustness of the implementations noise is added to the FM signal before it is applied to the demodulator and the signal-to-noise ratio (S/N) is measured at the demodulated signal. The noise is generated with a second function generator.

With Neutrik A2-D the S/N can not be measured directly, but it can be determined from the THD+N measurement (see Section 6.2), when everything other than the signal (S) is considered as noise ( $N_T$ ), including the distortions (D).

$$\frac{1 - (THD + N)}{(THD + N)} = \frac{1 - \frac{D+N}{S+D+N}}{\frac{D+N}{S+D+N}} = \frac{S + D + N - (D + N)}{D + N} = \frac{S}{D + N} = \frac{S}{N_T}$$

This calculation is equal to the one used in the simulations.

The implementations are tested with two different noise levels. For the first the S/N of the FM signal is set to 20dB and for the second it is set to 15dB. Figure 6.13 shows the oscilloscope view of the FM signal without noise as it is generated by the function generator. Figure 6.14 shows the FM signal with added noise which is applied to the input for the robustness measurements.

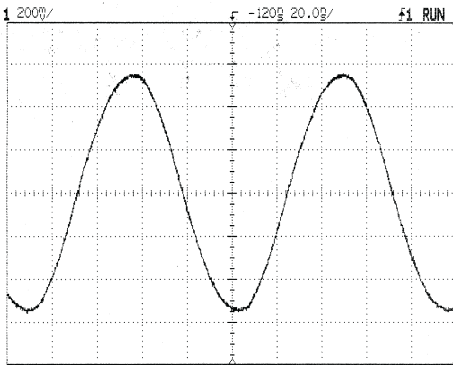


Figure 6.13.: Oscilloscope print FM signal normal

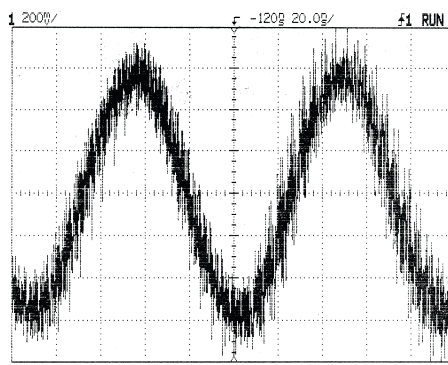


Figure 6.14.: Oscilloscope print FM signal with noise added

Figure 6.15 shows the spectrum of the demodulated signal with an FM input signal which has an S/N ratio of 20dB. The fixed-point mixed implementation with the direct table lookup is used. Compared to Figure 6.1 which shows the same spectrum for an FM signal without noise, it shows clearly that the noise naturally grows. A slight deformation of the noise due to the derivation can also be observed.

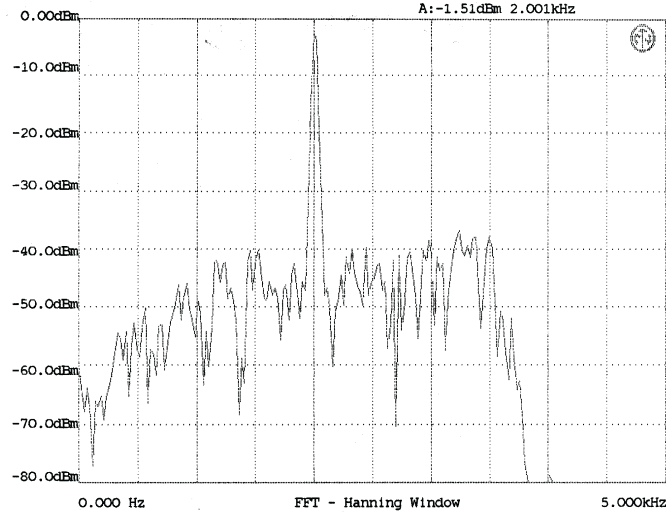


Figure 6.15.: Spectrum of an demodulated signal with an S/N ratio of 20dB by the FM signal

### 6.3.1. Mixed Demodulator

For the mixed demodulator algorithm the robustness of the fixed-point implementation with the direct lookup and the one with the interpolated lookup is measured. Further also the floating-point implementation is tested. Figure 6.16 shows the results, when the input FM signal has a S/N

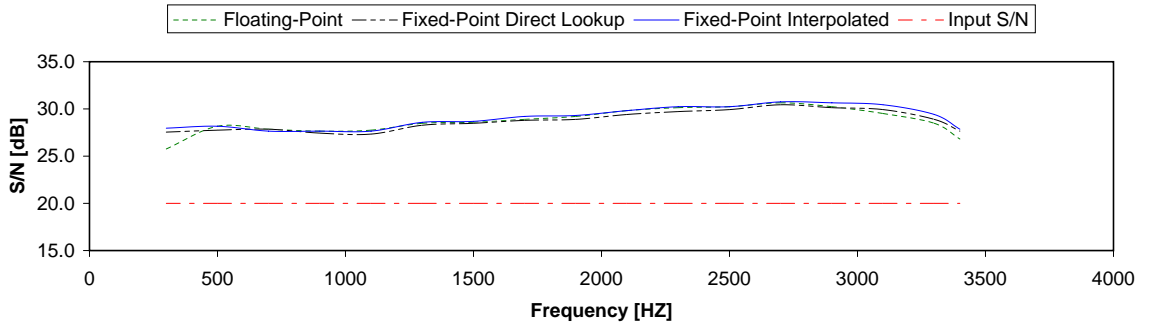


Figure 6.16.: S/N of the mixed demodulated signal with a S/N of 20dB at the input FM signal

of 20dB. Figure 6.16 shows the results, when the input FM signal has a S/N of 15dB. Both figures also show the S/N of the input FM signal as a reference.

It can be seen that all the implementations have an almost identical behavior. The S/N ratio is improved compared to the input ratio for all frequencies. It is slightly better for higher frequencies than for lower ones.

### 6.3.2. PLL Demodulator

For the PLL demodulator algorithm the fixed-point with the direct lookup and the floating-point implementation are tested with noise added to the input signal. Figure 6.16 shows the results, when the input FM signal has a S/N of 20dB. Figure 6.16 shows the results, when the input FM



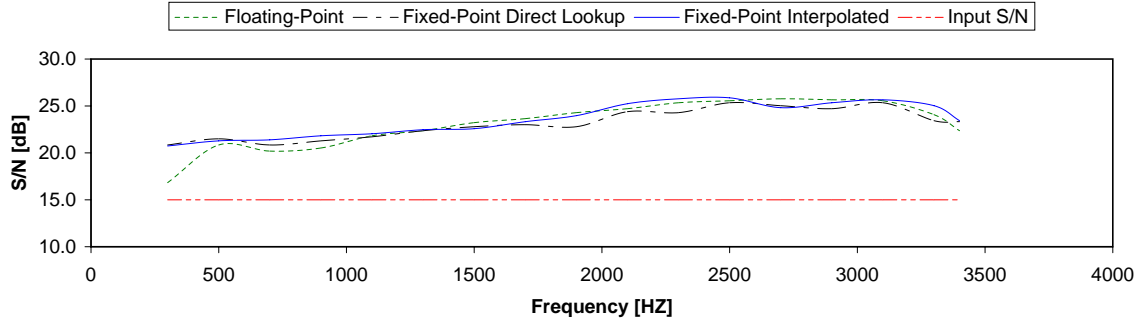


Figure 6.17.: S/N of the mixed demodulated signal with a S/N of 15dB at the FM input signal

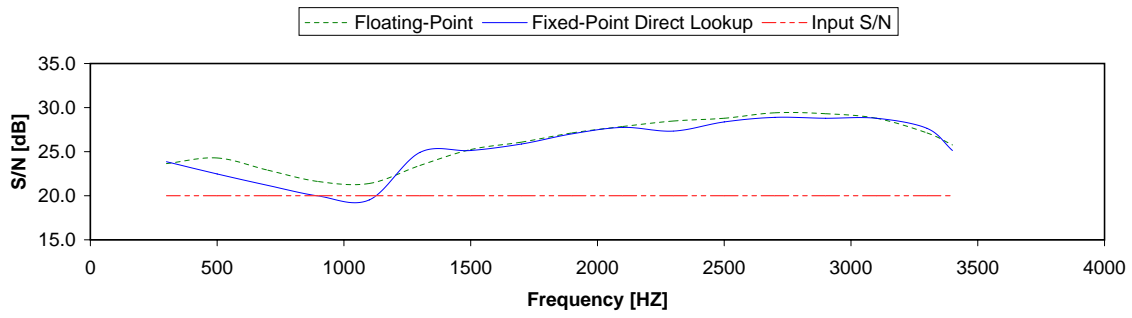


Figure 6.18.: S/N of the PLL demodulated signal with a S/N of 20dB at the input FM signal

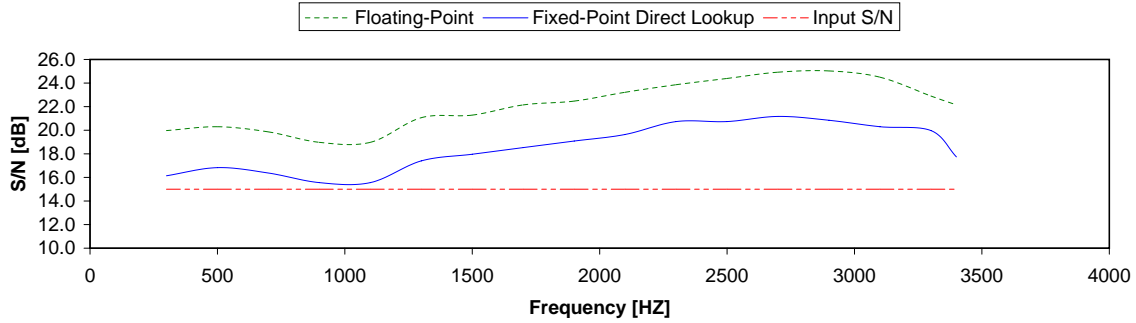


Figure 6.19.: S/N of the PLL demodulated signal with a S/N of 15dB at the FM input signal

signal has a S/N of 15dB. Again both figures also show the S/N of the input FM signal for reference.

### 6.3.3. Comparison of Mixed and PLL Demodulator

Figure 6.20 compares the different fixed-point demodulation implementations. The mixed demodulator algorithm is more robust than the PLL demodulator algorithm. It can be seen that the PLL shows a poor behavior at around 1000 Hz, which also could be observed by the signal quality measurements in Section 6.2.

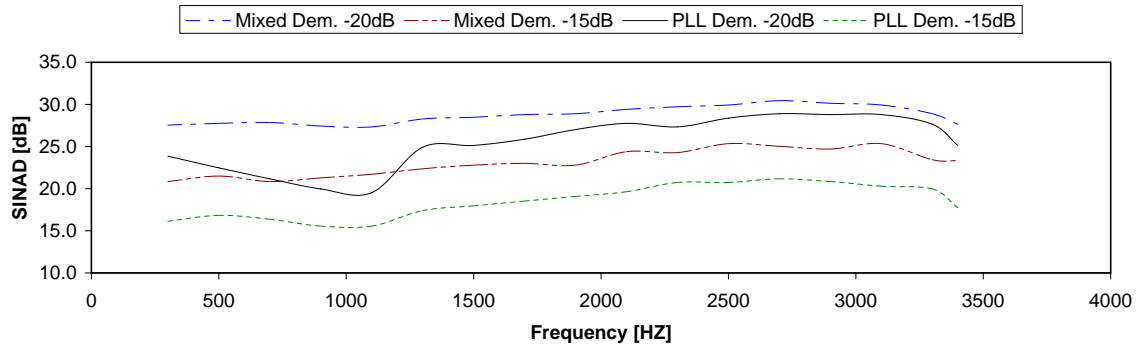


Figure 6.20.: S/N of the PLL demodulated signal with a S/N of 15dB at the FM input signal

## 6.4. Computing Time

### 6.4.1. Demodulation Functions

To benchmark the implementation each of the called functions is timed. This is made with the DSP/BIOS STS module (statistics objects manager) and the `CLK_gettime` function of the DSP/BIOS. `CLK_gettime` returns the number of high resolution clock cycles that have occurred. The time measurement are included between preprocessor directives to easily turn it on and off. Therefore function calls look like:

```
#if FM_DEM_CF_FUNC_TIME_STS
    STS_set(&mix_sts,CLK_gettime());
#endif
quad_mix(in_buffer_ptr,re_buffer,im_buffer);
#if FM_DEM_CF_FUNC_TIME_STS
    STS_delta(&mix_sts,CLK_gettime());
#endif

#if FM_DEM_CF_FUNC_TIME_STS
    STS_set(&dem_sts,CLK_gettime());
#endif
#if (FM_DEMODULATOR==FM_PLL_DEM)
    pll_demodulate(re_buffer,im_buffer,out_buffer);
#elif (FM_DEMODULATOR==FM_MIXED_DEM)
    mixed_demodulate(re_buffer,im_buffer,out_buffer);
#else
    #error No demodulator algorithm defined
#endif
#if FM_DEM_CF_FUNC_TIME_STS
    STS_delta(&dem_sts,CLK_gettime());
#endif

#if FM_DEM_CF_FUNC_TIME_STS
    STS_set(&filter_sts,CLK_gettime());
#endif
out_filter(out_buffer,out_buffer_ptr);
#if FM_DEM_CF_FUNC_TIME_STS
    STS_delta(&filter_sts,CLK_gettime());
#endif
```

`STS_set` can be used in conjunction with `STS_delta` to benchmark program performance. `STS_set` saves a value as the previous value in an STS object. `STS_delta` subtracts this saved value from the value it is passed.

The functions of the different implementations of the phases explained in Section 5.4.2 are compiled with the different optimization levels of the compiler. Additionally the debug information is turned off combined with the highest optimization level (`-o3` & no Debug).

<b>-o0</b>	<ul style="list-style-type: none"> <li>• Performs control-flow-graph simplification</li> <li>• Allocates variables to registers</li> <li>• Performs loop rotation</li> <li>• Eliminates unused code</li> <li>• Simplifies expressions and statements</li> <li>• Expands calls to functions declared inline</li> </ul>		<ul style="list-style-type: none"> <li>• Eliminates global unused assignments</li> <li>• Converts array references in loops to incremented pointer form</li> <li>• Performs loop unrolling</li> </ul>
<b>-o1</b>	Performs all -o0 optimizations, and: <ul style="list-style-type: none"> <li>• Performs local copy /constant propagation</li> <li>• Removes unused assignments</li> <li>• Eliminates local common expressions</li> </ul>	<b>-o3</b>	Performs all -o2 optimizations, and: <ul style="list-style-type: none"> <li>• Removes all functions that are never called</li> <li>• Simplifies functions with return values that are never used</li> <li>• Inlines calls to small functions</li> <li>• Reorders function declarations so that the attributes of called functions are known when the caller is optimized</li> <li>• Propagates arguments into function bodies when all calls pass the same value in the same argument position</li> <li>• Identifies file-level variable characteristics</li> </ul>
<b>-o2</b>	Performs all -o1 optimizations, and: <ul style="list-style-type: none"> <li>• Performs software pipelining</li> <li>• Performs loop optimizations</li> <li>• Eliminates global common subexpressions</li> </ul>		

The results are shown in Figures 6.21 to 6.25. It shows that the demodulation functions did not improve as much as the quadrature mixer and out filter functions. In the mixed demodulator this is caused by the division which is as a function call. Therefore a branch occurs in the loop and the optimizer is not able to parallelize the loop with software pipelining. In the PLL demodulator implementation the if-then-else structure to determine the sine and cosine also makes it impossible to perform those optimization.

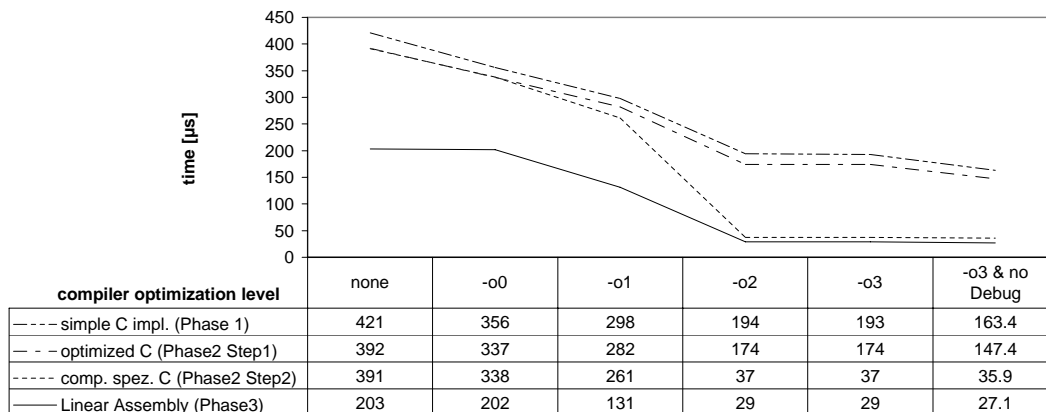


Figure 6.21.: Time measurement results Quadrature mixer ( *quad\_mix* )

Figure 6.26 compares the performance of mixed demodulation function with direct lookup, the mixed demodulation function with interpolated lookup and the PLL demodulation function. The comparison is carried out with the most time optimized implementation. The difference between the direct lookup and the interpolated lookup implementation of the mixed demodulator is very small, less than 9% when compiled with the highest optimization level. Hence the additional computing load due to the interpolation is rather small compared to the benefit of less storage and better signal quality (see Section 6.2.1).

The PLL demodulator uses clearly more computing time then the two mixed demodulators although it does not use a division. The computing load is introduced by the complex if-then-else structure to perform the sine and cosine functions.

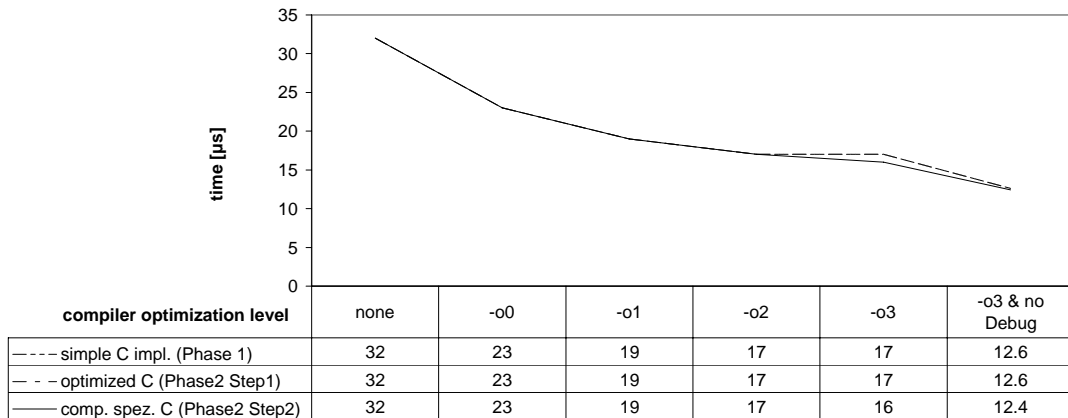


Figure 6.22.: Time measurement results mixed demodulator direct lookup ( *mixed\_demodulate* )

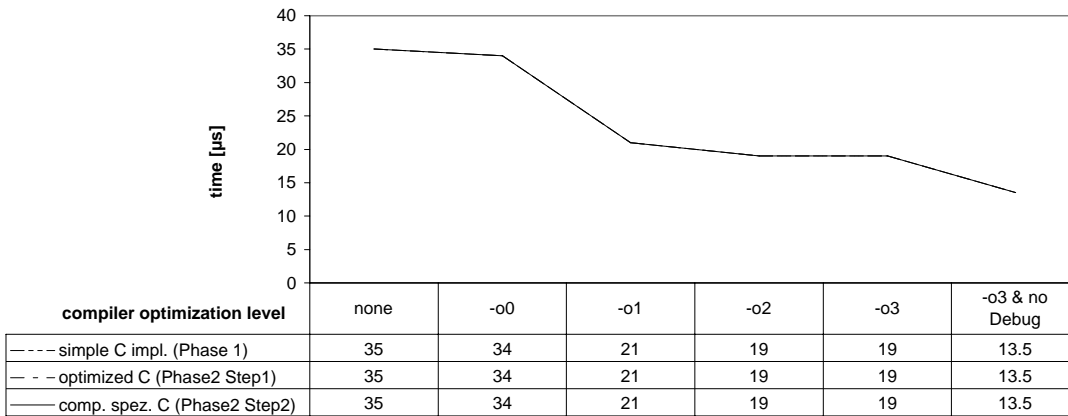


Figure 6.23.: Time measurement results mixed demodulator interpolated lookup ( *mixed\_demodulate* )

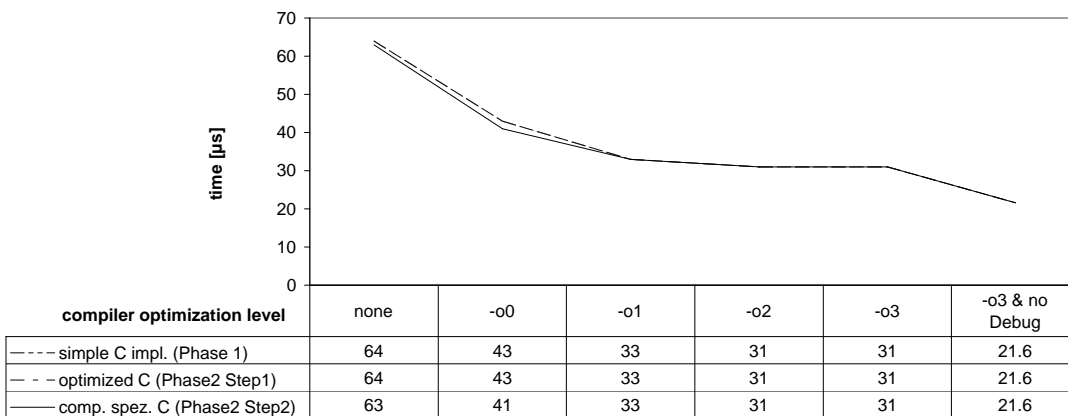


Figure 6.24.: Time measurement results PLL demodulator direct lookup ( *pll\_demodulate* )

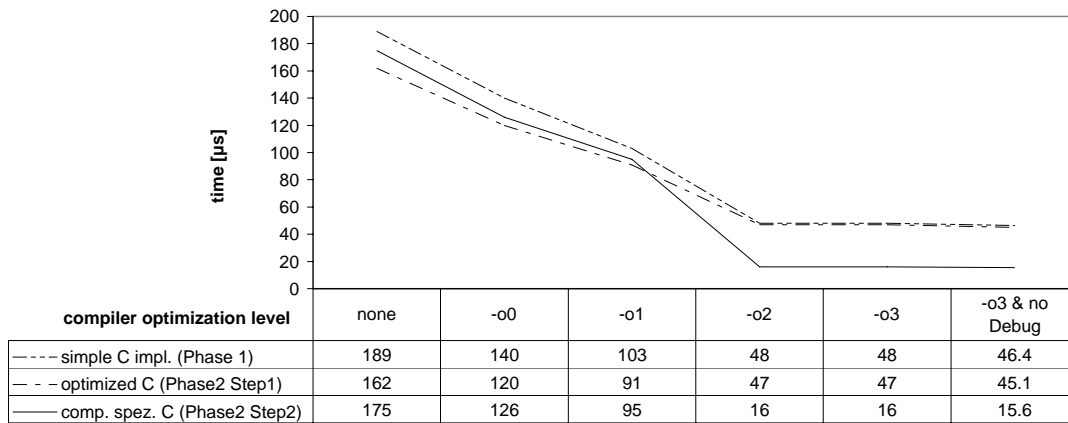


Figure 6.25.: Time measurement results out filter ( out\_filter )

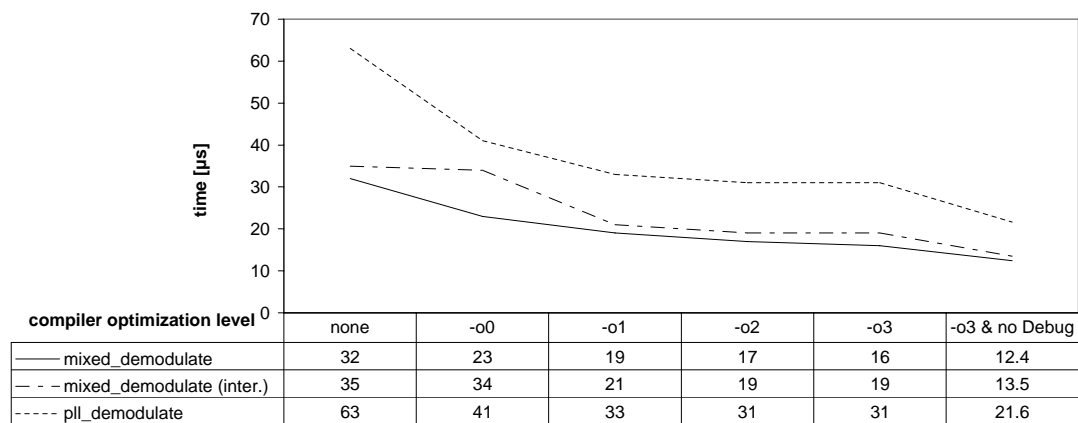


Figure 6.26.: Time comparison of the demodulation functions

### 6.4.2. Interrupt Routines

To measure the executing time of the interrupt routines (EDMA interrupt ,DAC interrupt and demodulate software interrupt (SWI)) the same measurement proceed as in Section 6.4.1 is applied. At the beginning of the routine a `STS_set` call is added. The `STS_delta` call is included at the end of the routine.

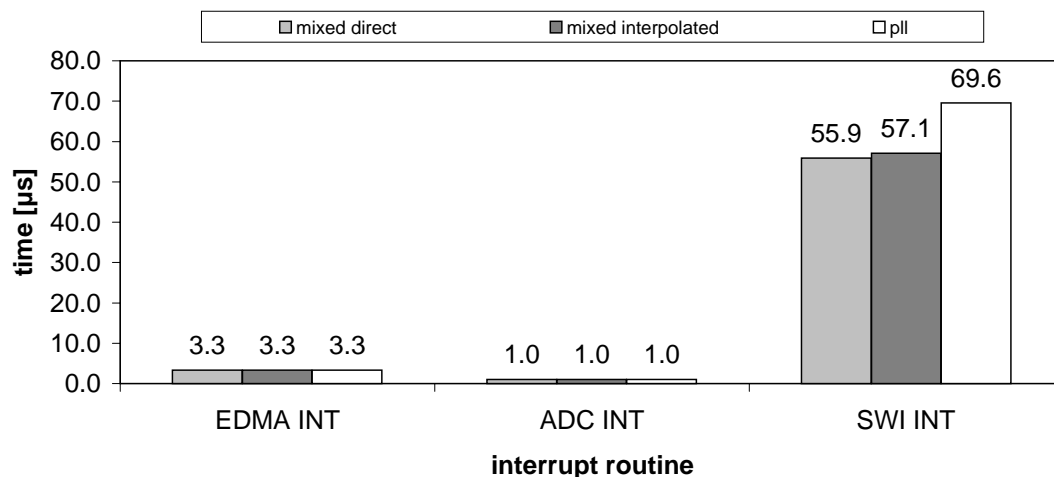


Figure 6.27.: Executing time of interrupt routines

The results are shown in Figure 6.27. The executing time for the hardware interrupt routines (EDMA INT , DAC INT) is for all implementations the same, because they are implemented equally, the difference is in the software interrupt where the actual demodulation is performed. The execution time of the hardware interrupts is very short compared to the time used by the software interrupt. This is as mentioned in Section 5.1.6 because the hardware interrupts are kept as short as possible and all the computing is executed in the software interrupt.

### 6.4.3. CPU Processing Load

The DSP/BOIS tool CPU Load Graph displays a graph of the target CPU processing load. The CPU load is defined as the amount of time not spent performing the low-priority task that runs when no other thread needs to run (idle loop). Thus all the other instrumenting is turned off and the CPU load graph shows the CPU load introduced by the FM demodulation.

Figure 6.28 shows the CPU processing load for the time optimized implementations compiled with the highest optimization level. All the implemented algorithms use less than 6% of the CPU processing. The difference between the different implementations is less than 0.1%.

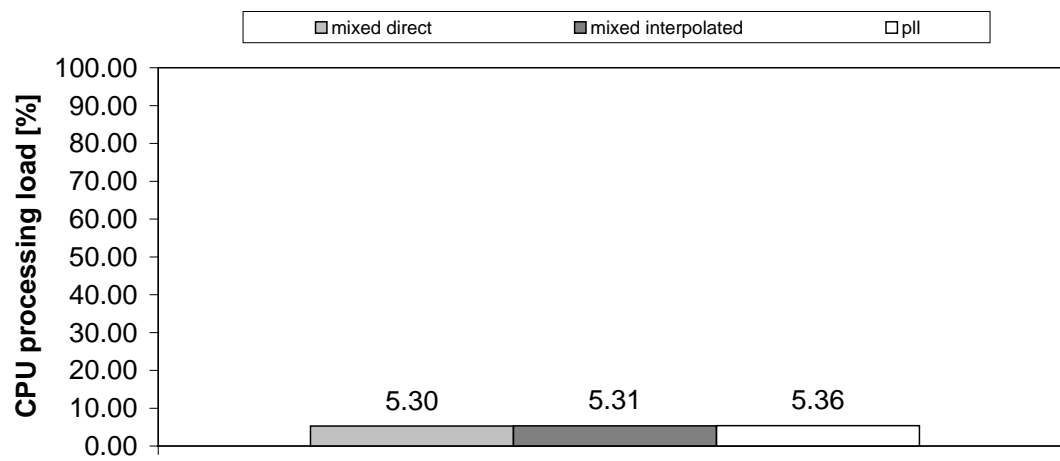


Figure 6.28.: CPU processing load





## 7. Conclusions and Recommendations

The given and planned objectives have been achieved. The mixed and the PLL demodulator algorithms are implemented on a DSP. The implementations are optimized in computing load and have been tested for signal quality and robustness.

The PLL demodulator function needs nearly twice as much computing time as the mixed demodulator function. Hence the total computing time of the PLL implementation is about 20% higher than the one of the mixed demodulator. Also the signal quality of the PLL is worse. There are some distortions around 1000Hz. The distortion are due to the non-linear closed loop.

For the given specifications, the mixed demodulator is clear a better choice. The difference in computing time between the implementation with the direct lookup and the interpolated lookup table is smaller than 3%, but the used storage space is lower and the signal quality is better. Therefore the implementation with the interpolated lookup table is preferred.

The PLL would be useful for applications with a lower frequency range. That would reduce the signal size in the PLL and so avoid the distortions. A smaller derivation brings along the same effect.

The CPU processing load is lower than 6% for all the optimized implementations. This could still be reduced by also rewriting the demodulation and the filter functions in linear assembly or even hand optimized assembly.

An implementation of the adaptive seeking of the carrier angular frequency in fixed-point would improve the signal quality. The output filter could then be reduced to a low-pass filter because there is no DC offset yet. That will also decrease the computing time.

Some improvements could be done in the PLL implementation. In the lookup table includes only the values of a fourth of sine wave. That brings along more queries and calculations to calculate the sine values. If more sine values are stored in the lookup table less queries and calculations are needed. Hence the computing time will be reduced but the needed storage space increases.

The floating-point implementations are not as slow as expected. In this work the floating-point implementations are not time optimized. Because the latest DSP generations support both the floating-point and fixed-point, it is worth optimizing them. The effort of a floating-point implementation is much lesser than a fixed-point implementation.



# A. Equipment

## A.1. Hardware

### A.1.1. DSP Board

**Name No.** TMS320C6711DSK

**Manufacturer** Texas Instruments

**Description** DSP Starter Kit : The DSK is a parallel port interfaced platform that allows TI, its customers, and third-parties, to efficiently develop and test applications for the C6711. The DSK consists of a C6711-based printed circuit board that will serve as a hardware reference design for TI's customers' products. With extensive host PC and target DSP software support, including bundled TI tools, the DSK provides ease-of-use and capabilities that are attractive to DSP engineers.

#### Key Features

- 150-MHz C6711DSP capable of executing 900 million floating-point operations per second (MFLOPS)
- Dual clock support; CPU at 150MHz and external memory interface (EMIF) at 100MHz
- Parallel port controller (PPC) interface to standard parallel port on a host PC (EEP or bi-directional SPP support)
- 16M Bytes of 100 MHz synchronous dynamic random access memory (SDRAM)
- 128K Bytes of flash programmable and erasable read only memory (ROM)
- 8-bit memory-mapped I/O port
- Embedded JTAG emulation via the parallel port and external XDS510 support
- Host port interface (HPI) access to all DSP memory via the parallel port
- 16-bit audio codec
- Onboard switching voltage regulators for 1.8 volts direct current (VDC) and 3.3 VDC
- Six light emitting diode (LED) indicators (one power-on indicator, one TBC-in-use indicator, one reset-active indicator, and three user-defined indicators)
- External desktop operation utilizing an external power supply and IEEE1284 parallel cable or an XDS510 emulator. Power supply and IEEE1284 parallel cable are provided with the DSK.
- Expansion memory and peripheral connectors for daughterboard support

### A.1.2. Analog-to-Digital Converter

**Name No.** THS1408EVM

**Manufacturer** Texas Instruments

**Description** Analog Digital Converter: The THS1408 evaluation module (EVM) provides a platform for evaluating the THS1408 analog-to-digital converter (ADC) under various signal, reference, and supply conditions.

**Key Features**

- 14-Bit Resolution
- 8 MSPS
- Internal Reference
- Timing Compatible with TMS320C6000 DSP

## A.2. Software

### A.2.1. DSP IDE

**Name No.** Code Composer Studio

**Version** 2.00.00

**Manufacturer** Texas Instruments

**Description** Integrated Development Environment (IDE) : Code Composer Studio (CCS) software is a fully IDE supporting Texas Instruments industry leading DSP platforms. Code Composer Studio integrates all host and target tools in a unified environment to simplify DSP system configuration and application design. This easy to use development environment allows DSP designers of all experience levels full access to all phases of the code development process.

### A.2.2. MATLAB

**Name No.** MATLAB & SIMULINK

**Version** 6.0.0.88 Release 12

**Manufacturer** The MathWorks

**Description** MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation.

## A.3. Signal And Measure Instruments

### A.3.1. Signal Generator

**Name No.** 33120A

**Manufacturer** Hewlett Packard

**Description** 15 MHz Function / Arbitrary Waveform Generator

### **A.3.2. Oscilloscope**

**Name No.** 54645A

**Manufacturer** Hewlett Packard

**Description** 100 MHz Oscilloscope (200 MSa/s)

### **A.3.3. Audio Measurement System**

**Name No.** A2-D Audio Measurement System

**Manufacturer** Neutrik Cortex Instruments

**Description** High performance two-channel audio test set. With a high performance analyzer providing a wide variety of measurement functions.



## B. Abbreviations and Symbols

### B.1. Formula Symbols

$b$	bandwidth	Hz
$f_A$	sample rate	Hz
$f_N$	message frequency	Hz
$f_T$	carrier frequency	Hz
$J_n(x)$	bessel function	
$k$	harmonic distortion	
$k_{FM}$	FM Constant	
$q$	quantization interval	
$s_{basis}$	complex baseband signal	
$s_D$	demodulated message signal	
$s_{FM}$	FM modulated signal	
$s_N$	message signal	
$s_{imag}$	Imaginary baseband signal (cos)	
$s_{real}$	Real baseband signal (cos)	
$s_T$	carrier	
$S/N$	signal to noise ratio	dB
$T$	sample time	sec
$\Delta F$	frequency derivation	
$\phi_{FM}$	FM argument function	
$\omega_T$	carrier frequency	rad/sec
$\omega_N$	message frequency	rad/sec
$\mu$	Modulation index	

## B.2. Abbreviations

ADC	Analog to Digital Converter
BSL	Board Support Library
CCS	Code Composer Studio
clk	Clock
CPU	Center Processing Unit
CSL	Chip Support Library
DAC	Digital to Analog Converter
DSK	DSP Starter Kit
DSP	Digital Signal Processor
EDMA	Enhanced Direct Memory Access
EMIF	External Memory Interface
Eq	Equation
EVM	Evaluation Module
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FM	Frequency Modulation
GSM-R	Global System for Mobile Communication - Railway
HSR	Fachhochschule Rapperswil
HW	Hardware
HWI	Hardware Interrupt
I	Inphase
IDE	Integrated Development Environment
IIR	Infinite Impulse Response
INT	Interrupt
IRQ	Interrupt Request
McBSP	Multichannel Buffered Serial Port
NTU	Nanyang Technological University
PLL	Phase-Locked Loop
PMR	Private (Professional) Mobile Radio
Q	Quadraturephase
S/N	Signal to Noise Ratio
SINAD	Signal, Noise and Distortion
SWI	Software Interrupt
TCC	Transfer Complete Code
TERRA	Terrestrial Trunked Radio
TERRAPOL	Digital PMR technology
THD+N	Total Harmonic Distortion and Noise
Ti	Texas Instruments
VCO	Voltage Controlled Oscillator
VHF	Very High Frequency



## C. Simulation Measure Algorithms

### C.1. Harmonic distortion k and SINAD

A statement about the signal quality can be done by the harmonic distortion factor  $k$ . It shows the rate of nonlinear distortions in a signal. It is defined as:

$$k = \sqrt{\frac{A_{2f}^2 + A_{3f}^2 + A_{4f}^2 \dots}{A_f^2 + A_{2f}^2 + A_{3f}^2 + A_{4f}^2 \dots}} \quad (\text{C.1})$$

It is rather difficult to measure all the single amplitudes of the oscillation. Therefore the following is used:

$$k = \sqrt{\frac{P_{tot} - P_f}{P_{tot}}} \quad (\text{C.2})$$

The power is calculated over a time period.  $P_{tot}$  is the power of the hole signal. To calculate the power  $P_{tot} - P_f$  the signal is first filtered with a band-stop and calculated afterwards. The so calculated factor corresponds to the harmonic distortion if there are just the harmonics and no noise. If this is not the case, not only the harmonics are included, but also the noise. The so calculated factor is called SINAD (*signal, noise and distortion*).

For the simulation the formula can be written as:

$$k = \sqrt{\frac{\frac{T}{T_{sim}} \cdot \sum (s(n) * BSP(z))^2}{\frac{T}{T_{sim}} \cdot s^2(n)}} = \sqrt{\frac{\sum (s(n) * h_{BSP}(n))^2}{s^2(n)}}$$

A MATLAB function was *kfaktor* created.

**kfaktor.m**

```
function k=kfaktor(signal,f0,fa)

% k=kfaktor(signal,f0,fa)
%
% Berechnet den Klirrfaktor k eines Signals mit frequenz f0 welches
% mit einer Frequenz von fa abgetastet wurde mit der Hilfe einer
% Bandsperre bei f0
% Wichtig signal muss mindestens 2000 Werte beinhalten

buffer=size(signal);
buffer=buffer(1); %Signallaenge
fN=fa/2;          %Bezugsfrequenz
N=1000;           %FIR Filterl"ange

if (buffer < (2*N))
error('Signal Vektor muss mindestens 2000 Werte enthalten');
end

%FIR Bandsperre
B=firl(N,[(f0-100)/fN (f0+100)/fN], 'stop',kaiser(N+1,8));
```

```
signal2=filter(B,1,signal);

%Abschneiden des Einschwingen des Filters
signal = signal(N:buffer);
signal2 = signal2(N:buffer);

k=sqrt((sum(signal2.^2))/(sum(signal.^2)));
```

## C.2. Signal to noise ratio S/N

Also to measure the signal to noise ratio a MATLAB function *snr* was programmed. It calculates the S/N ration in dB. With a bandpass the message signal is filtered out of the hole signal. This is subtracted from the hole signal. The result is the noise.

### snr.m

```
function sn=snr(signal,f0,fa)
% sn=snr(signal,f0,fa) [in dB]
%
% snr berechnet den Signal Noise abstand in einem signal
% das mit fa abgetastet ist f0 ist die Frequenz des Signal oder
% die Start- und Endfrequenz des [fstart fstop] Signals.
% sn ist eine Angabe in dB.

buffer=size(signal);
buffer=buffer(1); %Signallaenge
fN=fa/2; %Bezugsfrequenz
N=1000; %FIR Filterl"ange

if (buffer < (2*N))
error('Signal Vektor muss mindestens 2000 Werte enthalten');
end

%Filter berechnung
if (size(f0)==[ 1 2])
B=firl(N,[(f0(1))/fN (f0(2))/fN],kaiser(N+1,8));
else
B=firl(N,[(f0-100)/fN (f0+100)/fN],kaiser(N+1,8));
end
reinsignal = filter(B,1,signal); noise = signal-reinsignal;

%Abschneiden des Einschwingen des Filters
reinsignal = reinsignal(N:buffer); noise = noise(N:buffer); signal = signal(N:
buffer);

psig=sum(reinsignal.^2); pnoi=sum(noise.^2);
sn=psig/pnoi; sn=10*log10(sn);
```

## D. Bibliography

### D.1. English Books

- [1] N. Kehtarnavaz and B. Simsek, *C6x-Based Digital Signal Processing*, Prentice Hall, New Jersey 2000.
- [2] N. Dahnoun, *DSP implementation using the TMS320C6000 DSP platform*, Prentice Hall, Harlow England 2000.
- [3] MathWorks, *Simulink<sup>®</sup> Dynamic System Simulation for Matlab<sup>®</sup> (Version 4)*, MathWorks, Natick November 2000.

### D.2. German Books

- [4] P. Gerdson and P. Kröger, *Digitale Signalverarbeitung in der Nachrichtenübertragung*, Springer-Verlag, Berlin-Heidelberg 1993.
- [5] Dr. A. Schüeli, *Digitale Signalverarbeitung*, Rapperswil, Version 2001.
- [6] K. D. Kammeyer and K. Kroschel, *Digitale Signalverarbeitung*, Teuber Studienbücher, Stuttgart 1996.
- [7] E. Pehl, *Digitale und analoge Nachrichtenübertragung*, Hüntig, Heidelberg 1998.
- [8] E. Stadler, *Modulationsverfahren*, Vogel-Verlag ,Würzburg 1976.
- [9] H.D. Lüke, *Signalübertragung*, Springer-Verlag, Berlin-Heidelberg 1985.

### D.3. Texas Instruments Documentation

- [10] SPRA509 Aaron Kofi Aboagye, *Overflow Avoidance Techniques in Cascaded IIR Filter Implementations on the TMS320 DSP's* Texas Instruments , May 1999.
- [11] SLAU045A , *THS14xx/5691 EVM for the THS14xx ADC and THS56xx DAC Families User's Guide* Texas Instruments , September 2000.
- [12] SLAS248 , *Data Sheet THS1408 Analog-to-Digital Converter* Texas Instruments , December 1999.
- [13] SLAS202B , *Data Manual TLC320AD535C/I Dual Channel Voice/Data Codec* Texas Instruments , 2000.
- [14] SPRS088B , *Data Sheet TMS320C6711, TMS320C6711B floating-point digital signal processors* Texas Instruments , February 1999.

- [15] SPRU198F , *TMS320C6000 Programmer's Guide* Texas Instruments , February 2001.
- [16] SPRU190D , *TMS320C6000 PeripheralsReference Guide* Texas Instruments , February 2001.
- [17] SPRU401B , *TMS320C6000 Chip Support Library API User's Guide* Texas Instruments , April 2001.
- [18] SPRU432 , *TMS320C6000 DSK Board Support Library API User's Guide* Texas Instruments , October 2000.

## E. Simulink models

### E.1. Baseband-Delaydemodulator

#### Idealer Basisband-Verzögerungsdemodulator

##### Callback Routines:

InitFcn -> init\_verzoegerungideal.m  
StopFcn -> stop\_verzoegerungideal.m

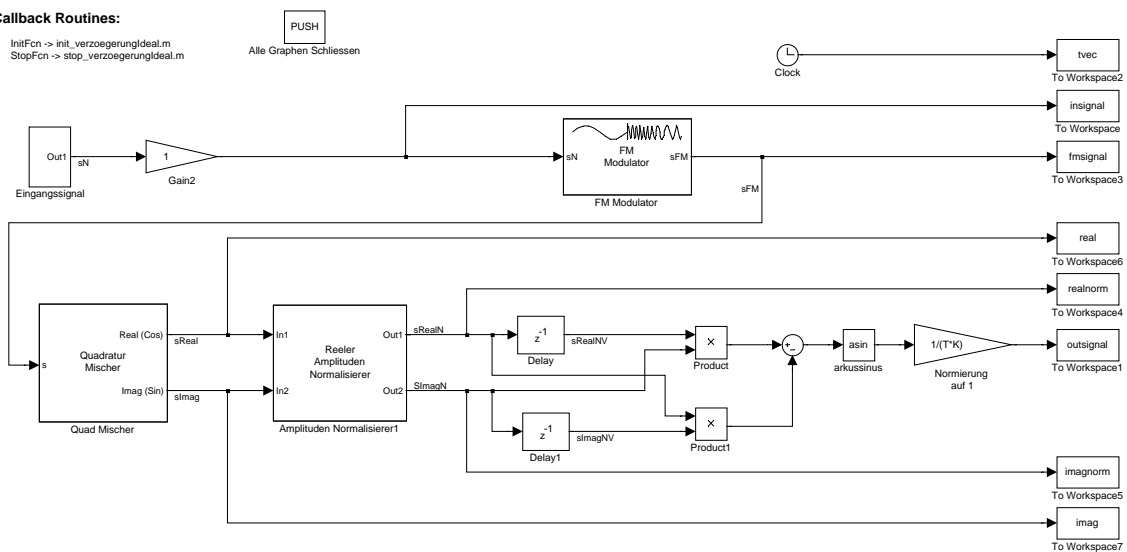
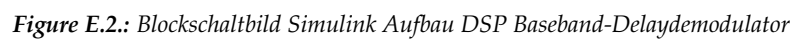
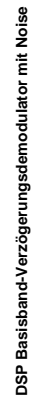


Figure E.1.: Blockschaltbild Simulink Aufbau Idealer Baseband-Delaydemodulator





## E.2. Phase-Adapter-Demodulator

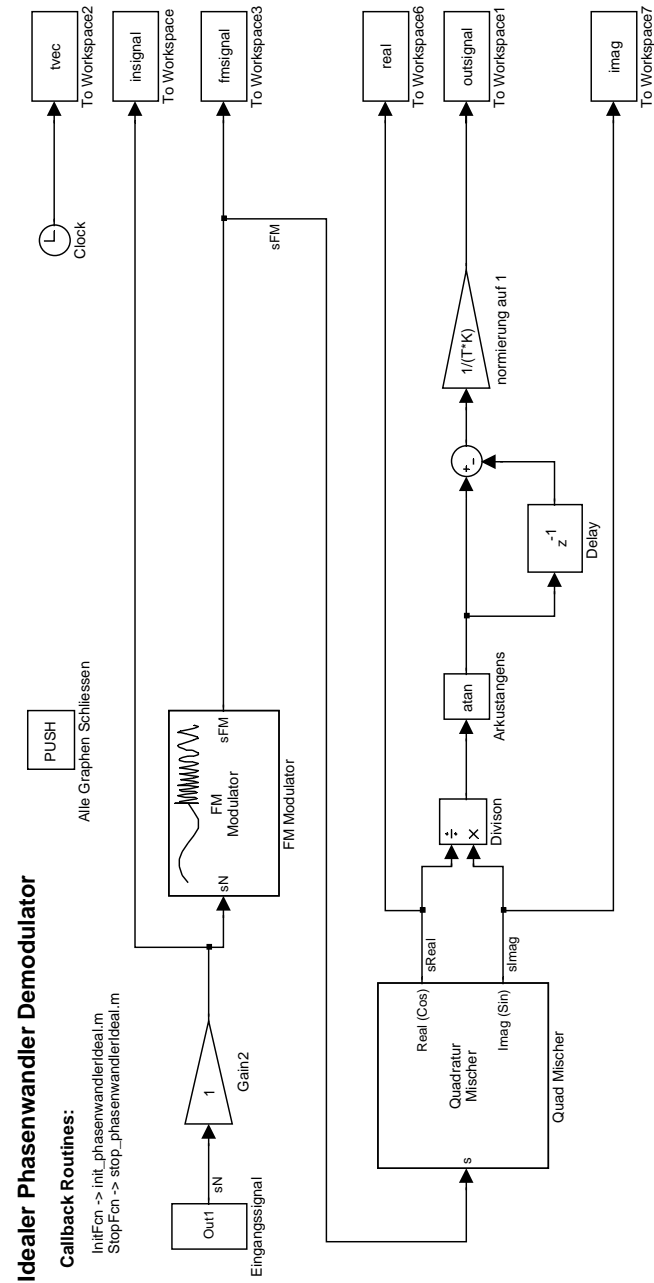


Figure E.4.: Blockschaltbild Simulink Aufbau Idealer Phase-Adapter-Demodulator



## E.3. Phasenregelschleife

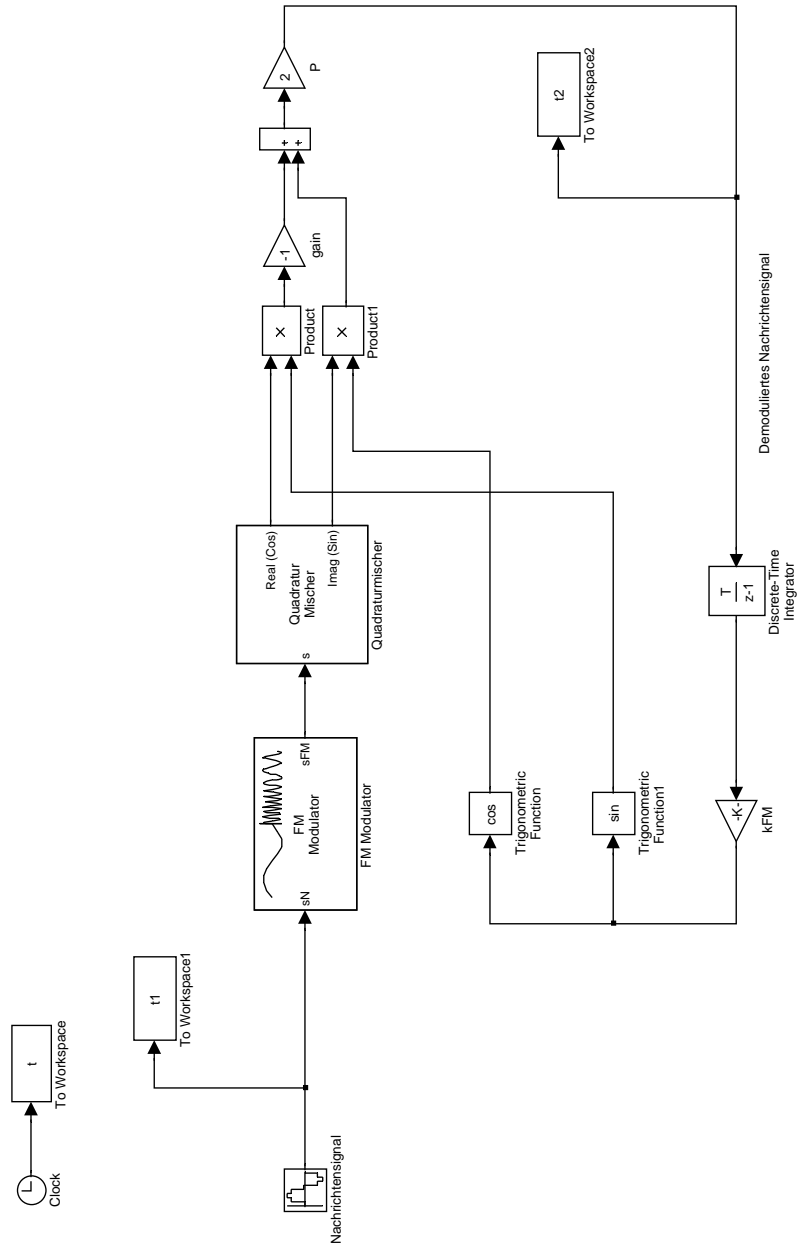


Figure E.5.: Blockschaltbild Simulink Aufbau Idealer PLL

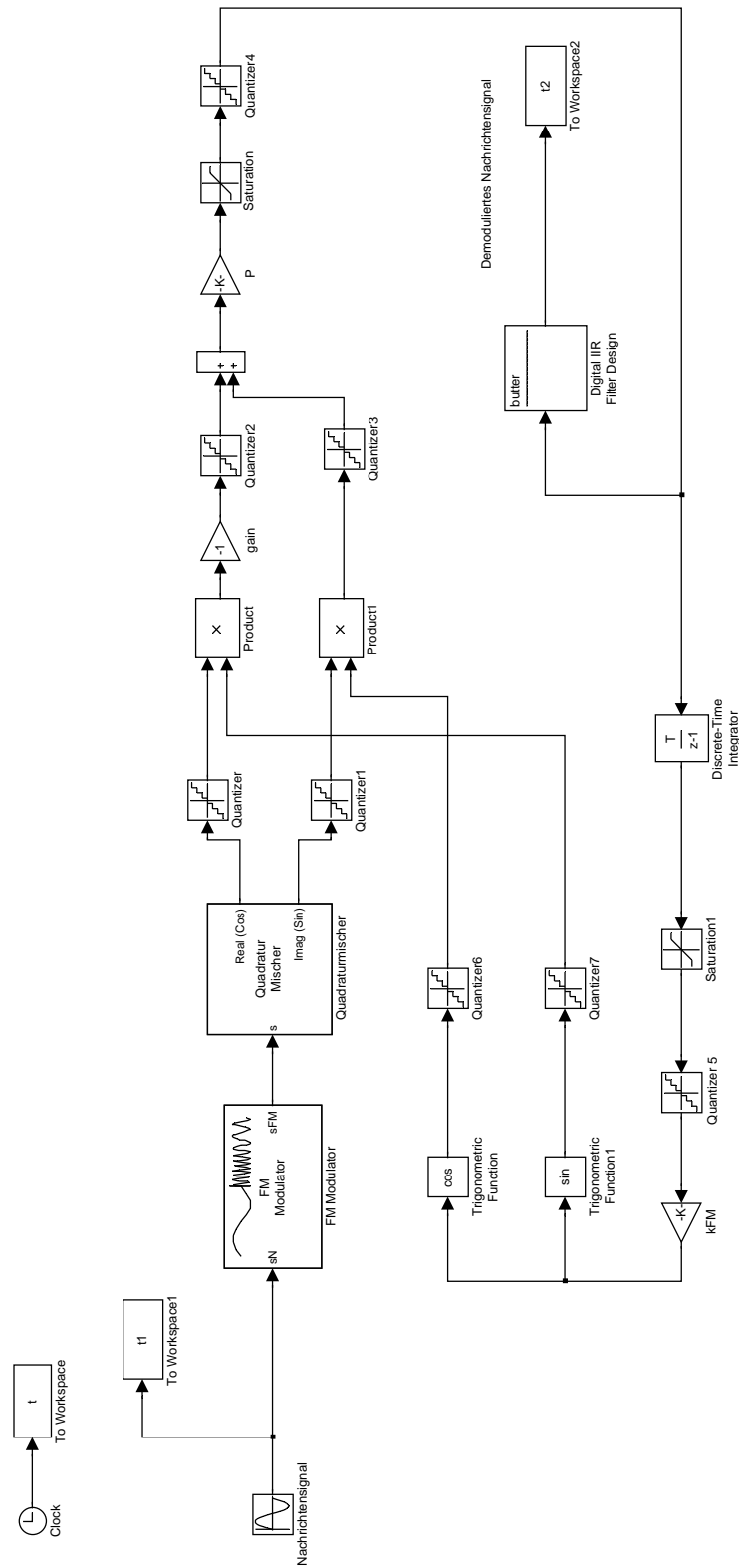


Figure E.6.: Blockschaltbild Simulink Aufbau DSP PLL

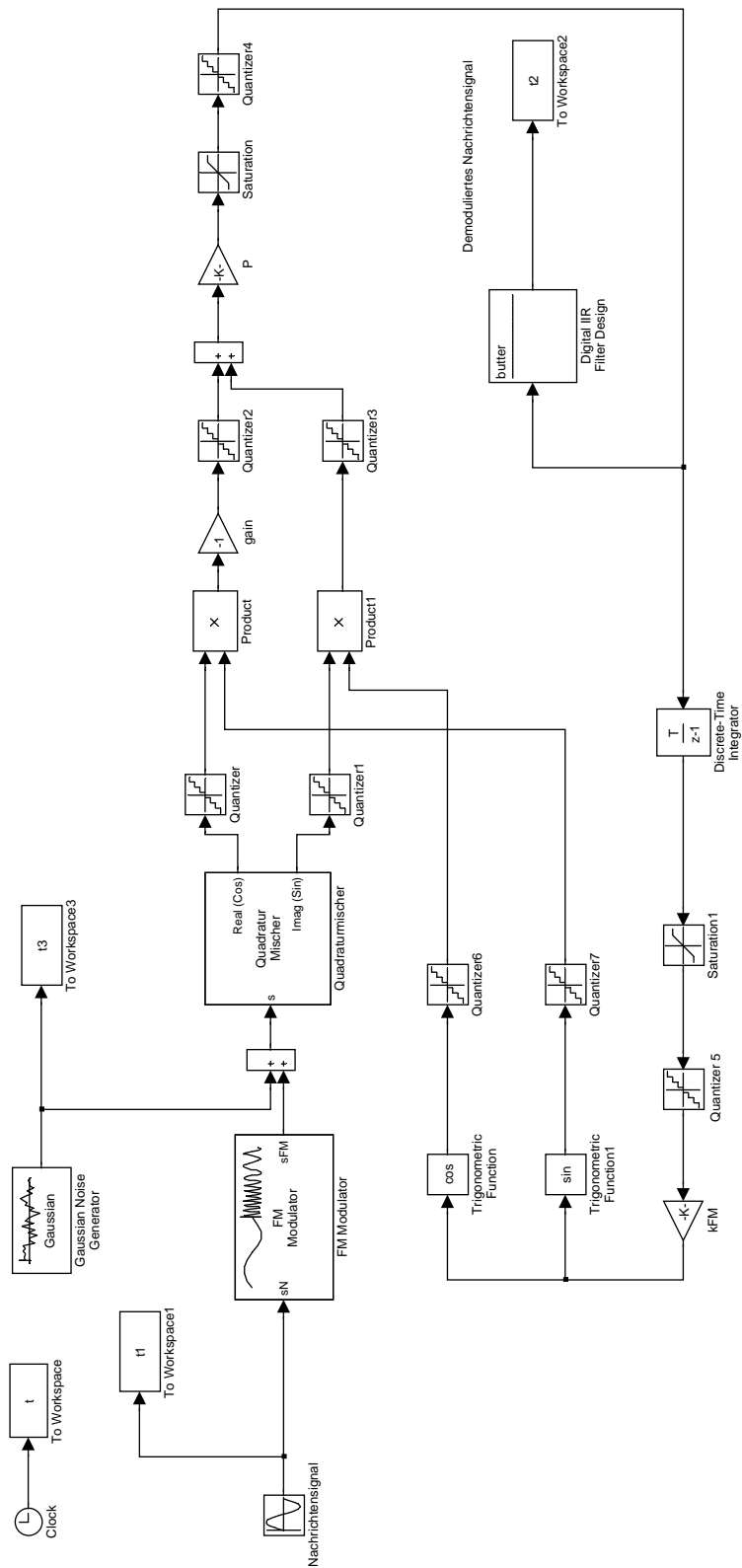
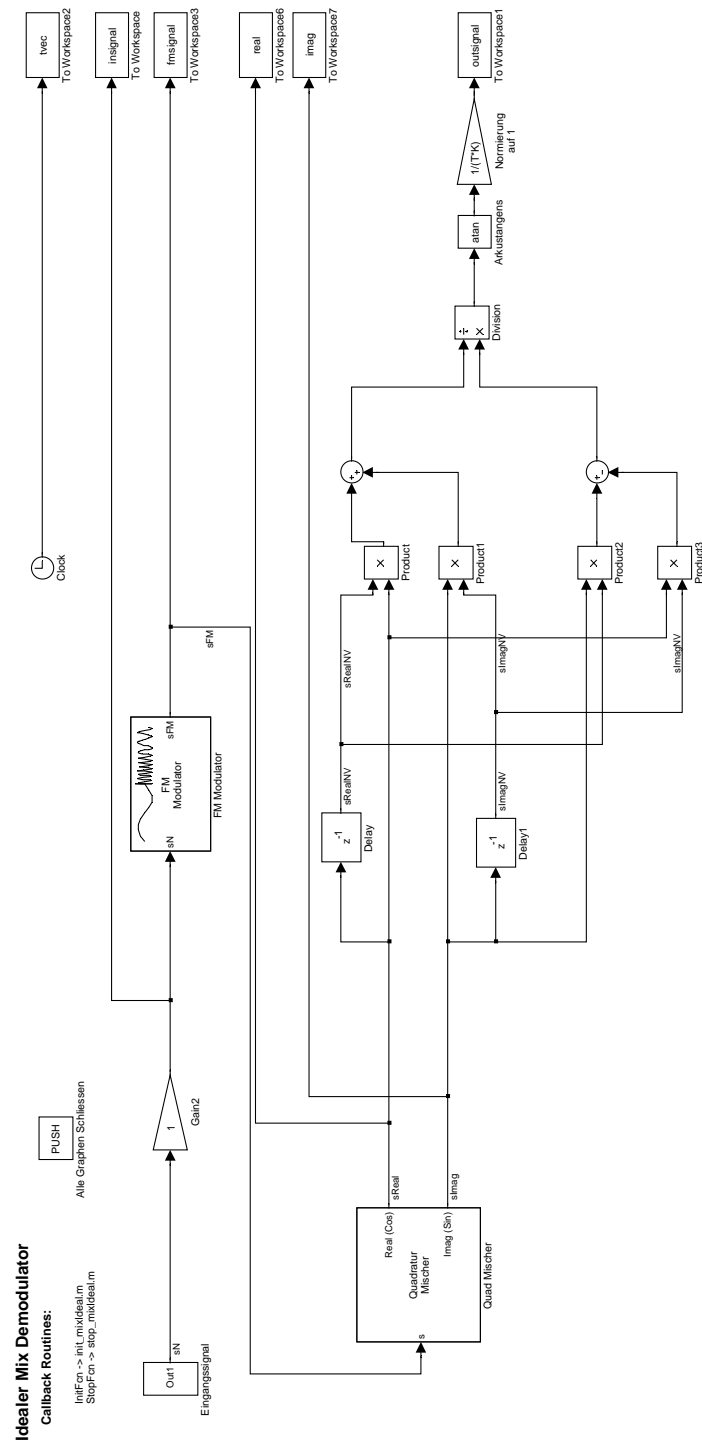
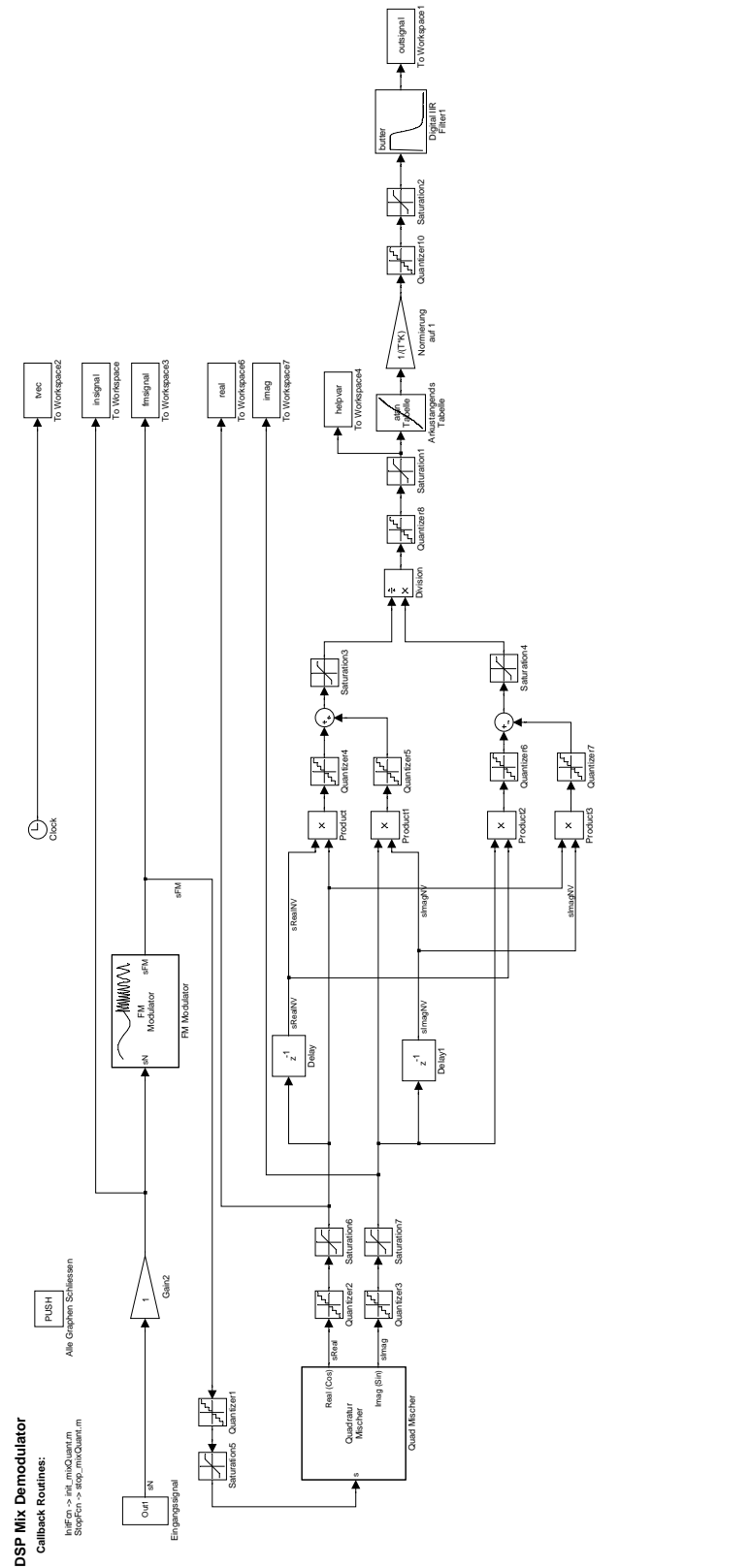


Figure E.7.: Blockschaltbild Simulink Aufbau DSP PLL mit gausschem Rauschen





**Figure E.9.: Blockschaltbild Simulink Aufbau DSP Mix Demodulator**

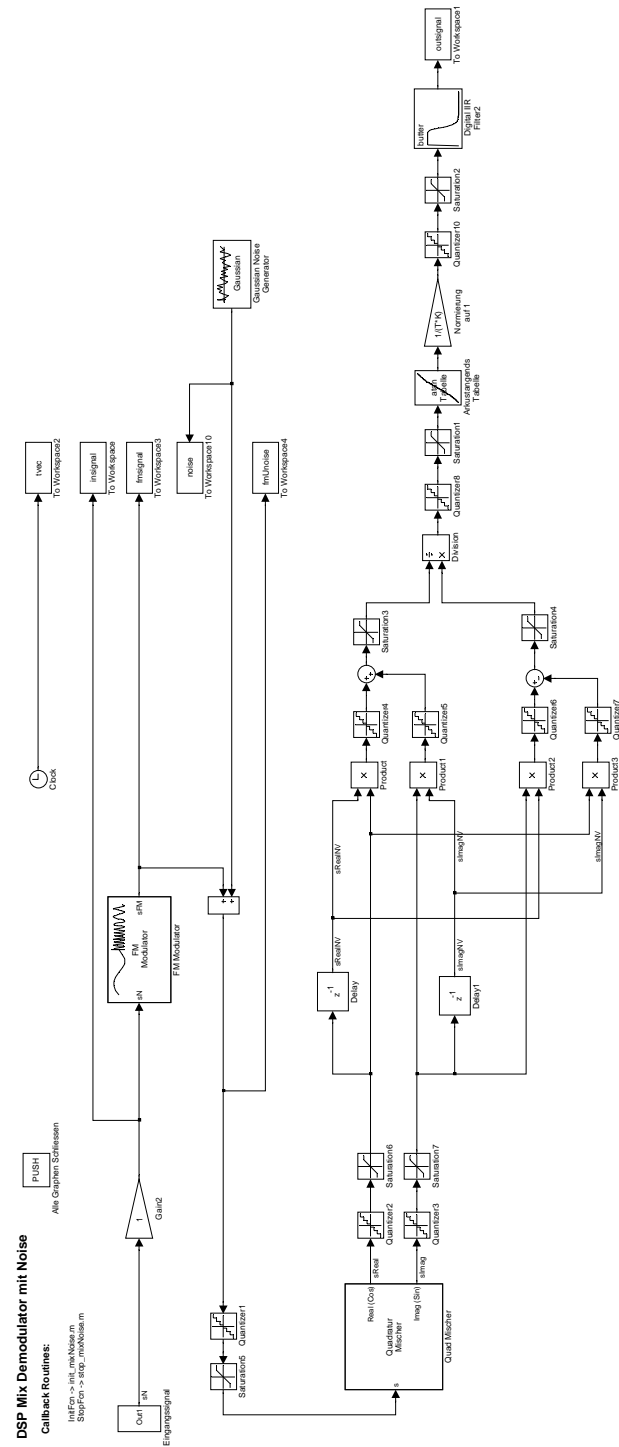


Figure E.10.: Blockschaltbild Simulink Aufbau DSP Mix Demodulator mit gausschem Rauschen

# F. Listings

In the following sections the C, assembler, and MATLAB listings are included.

## F.1. C Listings

### F.1.1. ADC and DAC

adc\_THS1408

Listing F.1: adc\_THS1408.h

```

1  /*
2  2  ANALOG DIGITAL CONVERTER THS1408 - VERSION 1
3
4  file : adc_THS1408.h
5  date : oct 2001 - Jan 2002
6  by : c. haller
7  f. schnyder
8  */
9
10 #ifndef _ADC_THS1408_
11 #define _ADC_THS1408_
12
13
14 /*-----
15 includes
16 -----
17
18 #include <cs1_timer.h>
19 #include <cs1_emif.h>
20 #include <cs1_irq.h>
21 #include <cs1_edma.h>
22
23 #include "global_settings.h"
24
25 /*-----
26 debug defines
27 -----
28 #define THS1408_ENABLE_SET_FREQ 0
29 /* enable or disable the function ths1408_setFreq */
30
31 /*-----
32 defines EDMA TCC for the two buffers
33 -----
34 #define THS1408_BUFFER_A_TCC (4) /* */
35 #define THS1408_BUFFER_B_TCC (5) /* */
36
37 /*-----
38 defines for Control Register of the THS1408 (refer SLAS248)
39 -----
40 /* Power Down D13 */
41 #define THS1408_PWD_NORMAL (0) /* normal operation */

```

```

42 #define THS1408_PWD_DOWN (1) /* power down */
43 /* Reference Select D12 */
44 #define THS1408_REF_INTERNAL (0) /* internal reference */
45 #define THS1408_REF_EXTERNAL (1) /* external reference */
46 /* Output Format D11 */
47 #define THS1408_FOR_STRAIGHT (0) /* straight binary */
48 #define THS1408_FOR_2SCOMPL (1) /* 2s complement */
49 /* Test Mode D8-D10 */
50 #define THS1408_TM_NORMAL (0x0) /* normal operation */
51 #define THS1408_TM_REFN_RFFN (0x1) /* both inputs=REF- */
52 #define THS1408_TM_VREF_RFFN (0x2) /* IN+=Vref/2, IN-=REF- */
53 #define THS1408_TM_REFP_RFFN (0x3) /* IN+=REF+, IN-=REF- */
54 #define THS1408_TM_NORMAL2 (0x4) /* normal operation */
55 #define THS1408_TM_REFP_RFFP (0x5) /* both inputs=REF+ */
56 #define THS1408_TM_REFN_VREF (0x6) /* IN+=REF-, IN-=Vref/2 */
57 #define THS1408_TM_REFN_RFFP (0x7) /* IN+=REF-, IN-=REF+ */
58 /* Offset correction D7 */
59 #define THS1408_OF_ENABLE (0) /* enable */
60 #define THS1408_OF_DISABLE (1) /* disable */
61 /* Reserved Fields D0-D6 */
62 #define THS1408_ISB_RESERVED (0x00) /* zeroes for RES Field */
63 /* Reserved Fields D14-D15 */
64 #define THS1408_MSB_RESERVED (0x0) /* zeroes for RES Field */
65
66 /*-----
67 Parameters for the THS1408EVM Analog Digital Converter
68 to TMS320C6711
69 -----
70 /*-- Handle parameters --*/
71 #define THS1408_RES_ADDR (0xA0000040) /* address of RES register */
72 #define THS1408_PGA_ADDR (0xA0000044) /* address of PGA register */
73 #define THS1408_OFF_ADDR (0xA0000048) /* address of OFF register */
74 #define THS1408_CTL_ADDR (0xA000004C) /* address of CTL register */
75 /*-- EMIF Interface Parameters --*/ /*Org*/
76 #define THS1408_RDSRTP (1) /* read */
77 #define THS1408_RDSRTRB (3)
78 #define THS1408_RDHLD (1)
79 #define THS1408_WRSRTP (4) /* write */
80 #define THS1408_WRHLD (6)
81 #define THS1408_WRSRTRB (3)
82 /*-- Global Parameters */
83 #define THS1408_DSP_TYPE TMS320C6711 /* DSP Type */
84 #define THS1408_DSP_FREQ (150) /* in MHz */
85
86 /*-----
87 Default Initial Parameters
88 -----
89 /* Sample Frequency */
90 #define THS1408_SAMPLE_FREQ (64) /* in kHz */
91
92 /* DSP FREQ */
93 #define THS1408_TIM_PERIOD (0x125) /* =----- */
94 /* 8*SAMPLE_FREQ */
95
96 /* Register Parameters */
97 /* CTL */
98 #define THS1408_POW THS1408_PWD_NORMAL /* normal operation */
99 #define THS1408_REF THS1408_REF_INTERNAL /* internal reference */
100 #define THS1408_FORMAT THS1408_FOR_2SCOMPL /* 2s complement */
101 #define THS1408_TM THS1408_TM_NORMAL /* normal operation */
102 /* PGA */
103 #define THS1408_PGA (0x0) /* gain of 7dB */
104 /*-- OFF --*/
105 #define THS1408_OFFSET (0) /* no offset */
106
107 /*-----
108 structs and unions
109 -----
110 typedef union
111 {
112     unsigned short value;
113     struct
114     {
115         unsigned int lab_reserved :7;
116         unsigned int off :1;
117         unsigned int tm :3;
118         unsigned int format :1;
119         unsigned int ref :1;
120         unsigned int pwd :1;
121         unsigned int msb_reserved :2;
122     } control_bit;

```

```

122 } THS1408_CTRL; /* the Control Register of ADC (refer SLAS248 P.15) */
123
124 typedef struct
125 {
126     unsigned short pba;
127     unsigned short offset;
128     THS1408_CTRL ctrl;
129     Uint32 tim_period;
130 } THS1408_CONFIG; /* Configuration Struct */
131
132 typedef struct
133 {
134     volatile short *const res_addr;
135     volatile short *const psa_addr;
136     volatile short *const off_addr;
137     volatile short *const ctl_addr;
138 } THS1408_HANDLE; /* Handle to the ADC */
139
140 /*-----
141 global variables ( extern defined in .c )
142 -----*/
143 extern THS1408_HANDLE hths1408; /* handle to the AD Converter */
144 extern TIMER_Handle htimer0; /* handle to the Timer 0 */
145 extern EDMA_Handle hEdmaIn; /* handle to the Input EDMA */
146
147 /*-----
148 global functions
149 -----*/
150 /* Notice:
151 =====
152 - The function 'CSL_init()' must have been called before any
153   of the following functions can be called.
154 - First call 'ths1408_init' before using the other functions
155 */
156
157 /*
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201

```

## Listing F.2: adc\_THS1408.c

```

202 double ths1408_setFreq(double freq);
203 #endif
204
205 #endif /* _ADC_THS1408_ */
206
207 /*-----
208 END OF THE FILE adc_THS1408.h
209 -----*/
210
211 1 /*
212 2 ANALOG DIGITAL CONVERTER THS1408 - VERSION 1
213 3
214 4 file : adc_THS1408.c
215 5 date : oct 2001 - jan 2002
216 6 by : c. haller
217 7 f. schneider
218 8
219 9 */
220
221 10 /*-----
222 11 debug defines
223 12 -----*/
224 13 #define THS1408_CF_ERROR_DISP 1
225 14 /* Enable Error Log
226 15 If enabled a 'error_log' LOG needs to be defined in the
227 16 DSP/BIOS Config -> Log Manager
228 17 */
229 18
230 19 #if THS1408_CF_ERROR_DISP
231 20 #include <std.h>
232 21 #include <log.h>
233 22 extern far LOG_Obj error_log;
234 23 #endif
235 24
236 25 /*-----
237 26 includes
238 27 -----*/
239 28 #include "adc_THS1408.h"
240 29
241 30 /*-----
242 31 local functions
243 32 -----*/
244 33 void emif_init_THS1408();
245 34 void timer_init_THS1408();
246 35 void adc_edma_init(short inBufferA_ptr[], short inBufferB_ptr[]);
247 36 void adc_irq_init();
248 37
249 38 /*-----
250 39 global variables
251 40 -----*/
252 41 THS1408_HANDLE hths1408 = {
253 42     (volatile short *)THS1408_RES_ADDR,
254 43     (volatile short *)THS1408_PGA_ADDR,
255 44     (volatile short *)THS1408_OFF_ADDR,
256 45     (volatile short *)THS1408_CTL_ADDR,
257 46     /* handle to the AD Converter */
258 47 };
259 48
260 49 TIMER_Handle hTimer0; /* handle to the Timer 0 */
261 50 EDMA_Handle hEdmaIn; /* handle to the Input EDMA */
262 51 EDMA_Handle hEdmaOut; /* EDMA link for Buffer A */
263 52 EDMA_Handle hEdmaMainLinkB; /* EDMA link for Buffer B */
264 53
265 54 /*-----
266 55 implementation of functions
267 56 -----*/
268 57 void ths1408_init(short inBufferA[], short inBufferB[])
269 58 {
270 59     THS1408_CONFIG defconfig; /* default config */
271 60
272 61     /* Setup the emif parameters */
273 62     emif_init_THS1408();
274 63
275 64     /* Open and init the Timer0 */
276 65     timer_init_THS1408();
277 66
278 67     /* Setup the EDMA */
279 68     adc_edma_init(inBufferA, inBufferB);

```



```

69  /* Configure the THS1408 */
70  defconfig_ctrl.control_bit.pwd=THS1408_POW;
71  defconfig_ctrl.control_bit.ref=THS1408_REF;
72  defconfig_ctrl.control_bit.format=THS1408_FORMAT;
73  defconfig_ctrl.control_bit.twt=THS1408_TM;
74  defconfig_ctrl.control_bit.off=THS1408_OFF;
75  defconfig_ctrl.control_bit.msb_reserved=THS1408_MSB_RESERVED;
76  defconfig_ctrl.control_bit.lsb_reserved=THS1408_LSB_RESERVED;
77  defconfig_ctrl.control_bit.isb_reserved=THS1408_ISB_RESERVED;
78  defconfig_offset=THS1408_OFFSET;
79  defconfig_pga=THS1408_PGA;
80  defconfig_tim_period=THS1408_TIM_PERIOD;
81  ths1408_config(defconfig);
82
83  /* IRQ setup */
84  adc_irq_init();
85
86  void ths1408_config(THS1408_CONFIG config)
87  {
88      *ths1408_ctl_addr=config.ctlr.value;
89      *ths1408_off_addr=config.offset;
90      *ths1408_pga_addr=config.pga;
91
92      TIMER_setPeriod(htimer0,config.tim_period);
93      /* set Count to 0 because it could be bigger than Period
94       -> would need one overflow*/
95      TIMER_setCount(htimer0,0x0000);
96  }
97
98  void emif_init_THS1408()
99  {
100     /* set the EMIF parameters for the CE2 (THS1408) */
101     EMIF_FSET(CBCTL2, WRSTURB , THS1408_WRSTURB);
102     EMIF_FSET(CBCTL2, WRSTRB , THS1408_WRSTRB);
103     EMIF_FSET(CBCTL2, WRHLD , THS1408_WRHLD);
104     EMIF_FSET(CBCTL2, RDSTRB , THS1408_RDSTRB);
105     EMIF_FSET(CBCTL2, RDSTRB , THS1408_RDSTRB);
106     EMIF_FSET(CBCTL2, RDHLD , THS1408_RDHLD);
107     EMIF_FSET(CBCTL2, MTYPE , EMIF_CBCTL_MTYPE_ASNVC32);
108     EMIF_FSET(CBCTL2, TA , EMIF_CBCTL_TA_OF(0x00000000));
109     EMIF_FSET(CBCTL2, TA , EMIF_CBCTL_TA_OF(0x00000000));
110  }
111
112  void timer_init_THS1408()
113  {
114      TIMER_Config timerConfig;
115      Uint32 opt;
116
117      /* Open up timer 0 device */
118      htimer0 = TIMER_open(TIMER_DEV0, TIMER_OPEN_RESET);
119      TIMER_reset(htimer0);
120
121      #if THS1408_CF_ERROR_DISP
122      if (htimer0==INV)
123      {
124          LOG_printf(&error_log,"Error THS1408 : Could Not Open Timer 0 !");
125      }
126      #endif
127
128      /* Configure the timer device */
129      opt = TIMER_FMK(CTL, FUNC , TIMER_CTL_FUNC_TOUT)
130      / TIMER_FMK(CTL, INVOUT , TIMER_CTL_INVOUT_NO)
131      / TIMER_FMK(CTL, DATOUT , TIMER_CTL_DATOUT_0)
132      / TIMER_FMK(CTL, PMID , TIMER_CTL_PMIID_ONE)
133      / TIMER_FMK(CTL, GO , TIMER_CTL_GO_NO)
134      / TIMER_FMK(CTL, HLD , TIMER_CTL_HLD_YES)
135      / TIMER_FMK(CTL, CP , TIMER_CTL_CP_CLOCK)
136      / TIMER_FMK(CTL, CLKSRC , TIMER_CTL_CLKSRC_CPU0VR4)
137      / TIMER_FMK(CTL, INVINP , TIMER_CTL_INVINP_NO);
138
139      timerConfig.ctl = opt;
140      timerConfig.prd = THS1408_TIM_PERIOD;
141      timerConfig.cnt = 0;
142      TIMER_config(htimer0,&timerConfig);
143  }
144
145  void adc_edma_init(short inBufferA_ptr[],short inBufferB_ptr[])
146  {
147      EDMA_Config edmaConfigA;
148      EDMA_Config edmaConfigB;

```

```

149  /* open one EDMA Channel */
150  hEdmain = EDMA_Open(EDMA_CHA_TINT0_EDMA_OPEN_RESET);
151
152  #if THS1408_CF_ERROR_DISP
153  if (hEdmain==EDMA_HINV)
154  {
155      LOG_printf(&error_log,"Error THS1408 : Could Not Open EDMA Channel !");
156  }
157  #endif
158
159  /* Get a link table for A */
160  hEdmaInLinkA = EDMA_allocTable(-1);
161
162  #if THS1408_CF_ERROR_DISP
163  if (hEdmaInLinkA==EDMA_HINV)
164  {
165      LOG_printf(&error_log,"Error THS1408 : Could Not Allocate EDMA LINK A !");
166  }
167  #endif
168
169  /* Get a link tabel for B */
170  hEdmaInLinkB = EDMA_allocTable(-1);
171
172  #if THS1408_CF_ERROR_DISP
173  if (hEdmaInLinkB==EDMA_HINV)
174  {
175      LOG_printf(&error_log,"Error THS1408 : Could Not Allocate EDMA LINK B !");
176  }
177  #endif
178
179  /* configuration for in_buffers */
180  edmaConfigA.opt = EDMA_FMK(OPT, PRI , EDMA_OPT_PRI_HIGH)
181  / EDMA_FMK(OPT, ESIZE , EDMA_OPT_ESIZE_16BIT)
182  / EDMA_FMK(OPT, 2DS , EDMA_OPT_2DS_NO)
183  / EDMA_FMK(OPT, SUM , EDMA_OPT_SUM_NONE)
184  / EDMA_FMK(OPT, 2DD , EDMA_OPT_2DD_NO)
185  / EDMA_FMK(OPT, DUM , EDMA_OPT_DUM_INC)
186  / EDMA_FMK(OPT, TCINT , EDMA_OPT_TCINT_YES)
187  / EDMA_FMK(OPT, TCC , EDMA_OPT_TCC_OF(THS1408_BUFFER_A_TCC))
188  / EDMA_FMK(OPT, LINK , EDMA_OPT_LINK_YES)
189  / EDMA_FMK(OPT, FS , EDMA_OPT_FS_NO);
190
191  edmaConfigA.src = (Uint32) hths1408_res_addr;
192  edmaConfigA.cnt = BUFFERSIZE_INPUT;
193  edmaConfigA.dat = (Uint32) inBufferA_ptr;
194  edmaConfigA.idx = EDMA_FMK(IDX, FRMIDX , EDMA_IDX_FRMIDX_DEFAULT)
195  / EDMA_FMK(IDX, ELEIDX , EDMA_IDX_ELEIDX_DEFAULT);
196  edmaConfigA.rld = EDMA_FMK(RLD, ELERLD , EDMA_RLD_ELERLD_DEFAULT)
197  / EDMA_FMK(RLD, LINK , EDMA_RLD_LINK_OF(hEdmaInLinkB));
198
199  /* configuration for in_buffers */
200  edmaConfigB.opt = EDMA_FMK(OPT, PRI , EDMA_OPT_PRI_HIGH)
201  / EDMA_FMK(OPT, ESIZE , EDMA_OPT_ESIZE_16BIT)
202  / EDMA_FMK(OPT, 2DS , EDMA_OPT_2DS_NO)
203  / EDMA_FMK(OPT, SUM , EDMA_OPT_SUM_NONE)
204  / EDMA_FMK(OPT, 2DD , EDMA_OPT_2DD_NO)
205  / EDMA_FMK(OPT, DUM , EDMA_OPT_DUM_INC)
206  / EDMA_FMK(OPT, TCINT , EDMA_OPT_TCINT_YES)
207  / EDMA_FMK(OPT, TCC , EDMA_OPT_TCC_OF(THS1408_BUFFER_B_TCC))
208  / EDMA_FMK(OPT, LINK , EDMA_OPT_LINK_YES)
209  / EDMA_FMK(OPT, FS , EDMA_OPT_FS_NO);
210
211  edmaConfigB.src = (Uint32) hths1408_res_addr;
212  edmaConfigB.cnt = BUFFERSIZE_INPUT;
213  edmaConfigB.dat = (Uint32) inBufferB_ptr;
214  edmaConfigB.idx = EDMA_FMK(IDX, FRMIDX , EDMA_IDX_FRMIDX_DEFAULT)
215  / EDMA_FMK(IDX, ELEIDX , EDMA_IDX_ELEIDX_DEFAULT);
216  edmaConfigB.rld = EDMA_FMK(RLD, ELERLD , EDMA_RLD_ELERLD_DEFAULT)
217  / EDMA_FMK(RLD, LINK , EDMA_RLD_LINK_OF(hEdmaInLinkA));
218
219  /* configure channel and links */
220  EDMA_config(hEdmain,&edmaConfigA);
221  EDMA_config(hEdmaInLinkA,&edmaConfigA);
222  EDMA_config(hEdmaInLinkB,&edmaConfigB);
223
224  /* enable transfer completion interrupt for A and B */
225  EDMA_intEnable(THS1408_BUFFER_A_TCC);
226  EDMA_intEnable(THS1408_BUFFER_B_TCC);
227  }
228
229  void adc_irq_init()
230  {

```

```

229 /* Map and enable EDMA interrupt */
230 IRQ_map(IRQ_EVT_EDMAINT_8);
231 IRQ_enable(IRQ_EVT_EDMAINT);
232 )
233
234 void tsh1408_start()
235 {
236     TIMER_start(timer0);
237     EDMA_enableChannel(hEDmain);
238 }
239
240 #if TSH1408_ENABLE_SET_FREQ
241     double tsh1408_setFreq(double freq)
242     {
243         Uint32 period=0;
244         double ret=0;
245         period=(1000.0*TSH1408_DSP_FREQ)/(8*freq);
246         ret=(1000.0*TSH1408_DSP_FREQ)/(8*period);
247     }
248
249     TIMER_setPeriod(timer0,period);
250     /* set Count to 0 because it could be bigger than Period
251        -> would need one overflow*/
252     TIMER_setCount(timer0,0x0000);
253     return ret;
254 }
255 #endif
256
257 /*
258 END OF THE FILE adc_TSH1408.c
259 */

```

```

40 /*
41 void codec_start();
42 */
43
44 /*
45 Writes value to the input of the DAC. Unlike
46 the AD535 write API, it does not use polling to establish
47 that the McBSP is ready to write another sample. Rather, it
48 requires the McBSP to already be ready. In other words,
49 the AD535 writeHwi is for use within an Interrupt Service
50 Routine. The fact that you arrived at an McBSP transmit ISR
51 signifies that the McBSP is ready with another sample.
52
53 Parameters :
54     handle    handle to codec channel
55     out_data  value to be written to DAC.
56
57 void AD535_HWI_write(AD535_Handle handle, short out_data);
58 */
59
60 /* Sets 1 to the FREE field of the SPCR register of the
61    given McBSP port. When FREE is set to 1, the serial
62    clocks continue to run during an emulation halt.
63
64 Parameters :
65     port      McBSP port ( 0 or 1 )
66
67 void McBSP_setfree(int port);
68 */
69
70 #endif /* _DAC_CODEC_ */
71
72 /*
73 END OF THE FILE
74 */

```

## dac\_codec

Listing F3: dac\_codec.h

```

1 /*
2 DIGITAL ANALOG CONVERTER ON BOARD CODEC - VERSION 1
3
4 file : dac_codec.h
5 date : oct 2001 - jan 2002
6 by : c. haller
7 f. schnyder
8
9 */
10
11 #ifndef _DAC_CODEC_
12 #define _DAC_CODEC_
13
14 /*
15 includes
16
17 #include <csl_irq.h>
18 #include <csl_mcbasp.h>
19 #include <bsl_ad535.h>
20
21 */
22
23 global variables ( extern defined in .c )
24
25 extern AD535_Handle outAD535h ;
26
27 /*
28 global functions
29
30
31 */
32
33 Initialize the codec;
34 Sets up the config and maps and enables the interrupt
35
36 void codec_init();
37
38 /*
39 Starts the codec to produce interrupts
40 */

```

Listing F4: dac\_codec.c

```

1 /*
2 DIGITAL ANALOG CONVERTER ON BOARD CODEC - VERSION 1
3
4 file : dac_codec.c
5 date : oct 2001 - jan 2002
6 by : c. haller
7 f. schnyder
8
9 */
10
11 /*
12 includes
13
14 #include "dac_codec.h"
15
16
17
18 global variables
19
20 AD535_Handle outAD535h ;
21
22 */
23
24 implementation of functions
25
26 void codec_init()
27 {
28     AD535_Config outAD535c = {
29         AD535_LOOPBACK_DISABLE, /*Loopback mode */
30         AD535_MICGAIN_OFF, /*Microphone gain */
31         AD535_GAIN_0DB, /*ADC input gain */
32         AD535_GAIN_0DB, /*DAC input gain */
33     };
34
35     outAD535h = AD535_open(AD535_localId); /* open the AD535 for codec use */
36     AD535_reset(outAD535h); /* reset it */
37     AD535_config(outAD535h,& outAD535c ); /* setup according to outAD535c */
38     McBSP_setfree(MCBSP_DEV0);
39
40     IRQ_map(IRQ_EVT_XINT0_9); /* map the codec interrupt to the dsp */
41     IRQ_enable(IRQ_EVT_XINT0); /* enable the codec interrupt */
42 }

```

```

43 }
44
45 void codec_start()
46 {
47     AD535_write(outAD535h,0); /* codec starts to after the first value is
48                                written to it */
49 }
50
51 void AD535_HWI_write(AD535_Handle handle, short out_data)
52 {
53     NCBSF_Handle h;
54     h = AD535_getNcbgHandle(handle);
55     out_data &= 0xFFFE;
56     NCBSF_write(h, (int)out_data);
57 }
58
59 void NCBSF_setfree(int port)
60 {
61     switch (port)
62     {
63     case 0:
64         NCBSF_FSET(SPCR0_FREE,1);
65     case 1:
66         break;
67     case 2:
68         NCBSF_FSET(SPCR0_FREE,1);
69         break;
70     }
71 }
72
73 /*
74  * END OF THE FILE
75  */

```

## F1.2. Floating-Point

global\_settings

Listing F5: Floating-Point :: global\_settings.h

```

1  /*
2  * GLOBAL_SETTINGS - VERSION FLOATING POINT 1.0
3  */
4  file : global_settings.h
5  date : oct 2001 - jan 2002
6  by : c. haller
7       f. schnyder
8  */
9
10 #define BUFSIZE 128
11 #define DOWNSAMPLE_ONE 4
12 #define DOWNSAMPLE_TWO 2
13
14 #define BUFSIZE INPUT (BUFSIZE)
15 #define BUFSIZE DEMODULATOR (BUFSIZE/DOWNSAMPLE_ONE)
16 #define BUFSIZE OUTPUT (BUFSIZE/DOWNSAMPLE_TWO)
17
18 #define PI 3.14159265359

```

fm\_dem\_main

Listing F6: Floating-Point :: fm\_dem\_main.c

```

1  /*
2  * FM DEMODULATION MAIN - VERSION FLOATING POINT 1.0
3  */

```

```

5  date : oct 2001 - jan 2002
6  by : c. haller
7       f. schnyder
8  */
9
10 /*
11  * demodulator define
12  */
13 #define FM_MIXED_DDEM 10
14 #define FM_PLL_DDEM 20
15
16 #define FM_DEMODULATOR FM_MIXED_DDEM
17
18 /*
19  * includes
20  */
21
22 #include <std.h>
23 #include <swi.h>
24 #include <tbl.h>
25 #include <bsl.h>
26
27 #include "global_settings.h"
28 #include "adc_PHS1408.h"
29 #include "dac_codec.h"
30 #include "quad_mix.h"
31 #include "dovsample_one.h"
32 #if (FM_DEMODULATOR==FM_PLL_DDEM)
33 #include "pll_demodulate.h"
34 #elif (FM_DEMODULATOR==FM_MIXED_DDEM)
35 #include "mixed_demodulate.h"
36 #else
37 #error No demodulator algorithm defined
38 #endif
39 #include "out_filter.h"
40 #include "dovsample_two.h"
41
42 /*
43  * debug defines
44  */
45 #define FM_DEM_CF_DEBUG 0
46 /* to print comments to debug_log */
47
48 #define FM_DEM_CF_INT_OCCUR_LED 0
49 /* Led toggles when an interrupt occurs */
50 /* LED 1 codec interrupt (output) */
51 /* LED 2 edma interrupt (input) */
52 /* LED 3 timing error */
53
54 #define FM_DEM_CF_INT_TIME_LED 0
55 /* Led ON by interrupt start OFF at the end */
56 /* LED 1 codec interrupt (output) */
57 /* LED 2 edma interrupt (input) */
58 /* LED 3 software interrupt */
59
60 #define FM_DEM_CF_INT_TIME_STS 0
61 /* Enable STS Objects: */
62 /* -dac_sts */
63 /* -demodulate_sts */
64 /* -edma_sts */
65 /* to measure the duration time of the interrupts */
66
67 #define FM_DEM_CF_FUNC_TIME_STS 1
68 /* Enable STS Objects: */
69 /* -mix_sts */
70 /* -dem_sts */
71 /* -filter_sts */
72 /* to measure the duration time of the functions */
73
74 #if FM_DEM_CF_DEBUG
75 #include <log.h>
76 #extern far LOG_Obj debug_log;
77 #endif
78
79 #if FM_DEM_CF_INT_OCCUR_LED
80 #include <bsl_led.h>
81 #endif
82
83 #if FM_DEM_CF_INT_TIME_LED

```

```

85 #include <bsl_led.h>
86 #endif
87
88 #if FM_DEM_CF_INT_TIME_STS
89 #include <clk.h>
90 #include <sts.h>
91 extern far STS_Obj dac_sts;
92 extern far STS_Obj demodulate_sts;
93 extern far STS_Obj edma_sts;
94 #endif
95
96 #if FM_DEM_CF_FUNC_TIME_STS
97 #include <clk.h>
98 #include <sts.h>
99 extern far STS_Obj mix_sts;
100 extern far STS_Obj dem_sts;
101 extern far STS_Obj filter_sts;
102 extern far STS_Obj down_one_sts;
103 extern far STS_Obj down_two_sts;
104 #endif
105
106 /*-----
107 DSP/BIOS extern variables
108 -----*/
109
110 extern far SWI_Obj demodulate_swi;
111 extern HMI_enable();
112 extern HMI_disable();
113
114 /*-----
115 global variables
116 -----*/
117
118 short in_buffer_A[BUFFERSIZE_INPUT];
119 short in_buffer_B[BUFFERSIZE_INPUT];
120
121 float in_buffer[BUFFERSIZE_INPUT];
122
123 float re_buffer[BUFFERSIZE_INPUT];
124 float im_buffer[BUFFERSIZE_INPUT];
125
126 float re_buffer_down_one[BUFFERSIZE_DEMODULATOR];
127 float im_buffer_down_one[BUFFERSIZE_DEMODULATOR];
128
129 float out_buffer_down_one[BUFFERSIZE_DEMODULATOR];
130 float out_buffer_down_two[BUFFERSIZE_OUTPUT];
131
132 short out_buffer_A[BUFFERSIZE_OUTPUT];
133 short out_buffer_B[BUFFERSIZE_OUTPUT];
134
135
136 short* in_buffer_ptr;
137 short* out_buffer_ptr;
138 Bool a_inEDMA=TRUE;
139 short outcounter=0;
140
141 /*-----
142 the main function
143 -----*/
144 void main()
145 {
146     HMI_disable();
147
148     #if FM_DEM_CF_DEBUG
149     LOG_printf(debug_log,"Info : Begin of main ");
150     #endif
151
152     CSL_init(); /* on chip init */
153     BSL_init(); /* on board init */
154
155     ths1408_init(in_buffer_A,in_buffer_B);
156     codec_init();
157
158     HMI_enable();
159
160     ths1408_start();
161     codec_start();
162
163     #if FM_DEM_CF_INT_TIME_LED
164     LED_on(LED_1);
165
166     LED_off(LED_ALL);
167     #endif
168
169     #if FM_DEM_CF_DEBUG
170     LOG_printf(debug_log,"Info : End Of main ");
171     #endif
172 }
173
174 /*-----
175 hardware interrupt functions
176 -----*/
177
178 void edmaInt(void)
179 {
180     #if FM_DEM_CF_INT_OCCUR_LED
181     LED_toggle(LED_2);
182     #endif
183
184     #if FM_DEM_CF_INT_TIME_LED
185     LED_on(LED_2);
186     #endif
187
188     #if FM_DEM_CF_INT_TIME_STS
189     STS_set(&edma_sts,CLK_gettime());
190     #endif
191
192     /* reset outcounter */
193     outcounter=0;
194
195     /* select EDMA channel */
196     if (EDMA_intTest(THS1408_BUFFER_A_TCC))
197     {
198         #if FM_DEM_CF_DEBUG
199         LOG_printf(debug_log,"In_A");
200         #endif
201         /* Clear transfer completion interrupt flag */
202         EDMA_intClear(THS1408_BUFFER_A_TCC);
203
204         /* set buffer pointers */
205         a_inEDMA=FALSE;
206         in_buffer_ptr=in_buffer_A;
207         out_buffer_ptr=out_buffer_A;
208
209         /* post software interrupt --> start demodulation */
210         SWI_post(&demodulate_swi);
211     }
212     if (EDMA_intTest(THS1408_BUFFER_B_TCC))
213     {
214         #if FM_DEM_CF_DEBUG
215         LOG_printf(debug_log,"In_B");
216         #endif
217         /* Clear transfer completion interrupt flag */
218         EDMA_intClear(THS1408_BUFFER_B_TCC);
219
220         /* set buffer pointers */
221         a_inEDMA=TRUE;
222         in_buffer_ptr=in_buffer_B;
223         out_buffer_ptr=out_buffer_B;
224
225         /* post software interrupt --> start demodulation */
226         SWI_post(&demodulate_swi);
227     }
228
229     #if FM_DEM_CF_INT_TIME_STS
230     STS_delta(&edma_sts,CLK_gettime());
231     #endif
232
233     #if FM_DEM_CF_INT_TIME_LED
234     LED_off(LED_2);
235     #endif
236
237     void dacInt(void)
238     {
239         #if FM_DEM_CF_INT_OCCUR_LED
240         LED_toggle(LED_1);
241         #endif
242
243         #if FM_DEM_CF_INT_TIME_LED
244         LED_on(LED_1);
245     }

```

```

245 #endif
246
247 #if FM_DEM_CF_INT_TIME_STS
248 STS_set(&dac_sts,CLK_gettime());
249 #endif
250
251 /* interpolate a missing sample */
252 if (outcounter>15)
253 {
254     #if FM_DEM_CF_INT_OCCUR_LED
255     LED_toggle(LED_3);
256     #endif
257     /* select the buffer */
258     if(a.inEDMA)
259     {
260         AD535_HMI_write(outAD535h,(( out_buffer_A[15]+out_buffer_B[0])/2));
261         #if FM_DEM_CF_DEBUG
262         LOG_printf(&debug_log,"Out A error :%d",outcounter);
263         #endif
264     }
265     #else
266     {
267         AD535_HMI_write(outAD535h,((out_buffer_B[15]+out_buffer_A[0])/2));
268         #if FM_DEM_CF_DEBUG
269         LOG_printf(&debug_log,"Out B error :%d",outcounter);
270         #endif
271     }
272     #else /* output a value */
273     {
274         /* select the buffer */
275         if(a.inEDMA)
276         {
277             AD535_HMI_write(outAD535h,out_buffer_A[outcounter]);
278             #if FM_DEM_CF_DEBUG
279             LOG_printf(&debug_log,"Out A :%d",outcounter);
280             #endif
281         }
282         #else
283         {
284             AD535_HMI_write(outAD535h,out_buffer_B[outcounter]);
285             #if FM_DEM_CF_DEBUG
286             LOG_printf(&debug_log,"Out B :%d",outcounter);
287             #endif
288         }
289         /* update outcounter */
290         outcounter++;
291     }
292     #if FM_DEM_CF_INT_TIME_STS
293     STS_delta(&dac_sts,CLK_gettime());
294     #endif
295     #if FM_DEM_CF_INT_TIME_LED
296     LED_off(LED_1);
297     #endif
298 }
299
300 /*-----
301 software interrupt functions
302 -----*/
303
304 void demodulateSwiFunc(void)
305 {
306     int i;
307
308     #if FM_DEM_CF_INT_TIME_LED
309     LED_on(LED_3);
310     #endif
311
312     #if FM_DEM_CF_INT_TIME_STS
313     STS_set(&demodulate_sts,CLK_gettime());
314     #endif
315
316     for (i=0;i<BUFFERSIZE_INPUT;i++)
317     {
318         in_buffer_ptr[i]=in_buffer_ptr[i]<<2;
319     }
320
321     /*convert short to float*/
322     for (i=0;i<BUFFERSIZE_INPUT;i++)
323     {
324         in_buffer[i]=(float)in_buffer_ptr[i]/32767;

```

## quad\_mix

## Listing F.7: Floating-Point :: quad\_mix.h

```

1 /*
2  QUADRATURE MIXER - VERSION FLOATING POINT 1.0
3
4  file : quad_mix.h
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8  */

```

```

9 */
10
11 #ifndef _QUAD_MIX_
12 #define _QUAD_MIX_
13 /*
14 -----
15 includes
16 -----
17 #include "global_settings.h"
18
19 -----
20 global functions
21 -----
22 */
23
24 The in_buffer is quadrature mixed to the two
25 buffers re_buffer and im_buffer.
26
27 Parameters :
28 in_buffer buffer of input signal
29 re_buffer buffer of I signal
30 im_buffer buffer of Q signal
31 */
32 void quad_mix(float in_buffer[], float re_buffer[], float im_buffer[])
33 #endif /* _QUAD_MIX_ */
34
35 /*
36 -----
37 END OF THE FILE
38 -----
39 */

```

```

47
48 float zlp1=0.0;
49 float zlp2=0.0;
50 float zlp3=0.0;
51 float zlp4=0.0;
52 float zlp5=0.0;
53 float zlp6=0.0;
54
55 /*delay variables of iir filter for low pass filtering the Q-path*/
56
57 float zlp1=0.0;
58 float zlp2=0.0;
59 float zlp3=0.0;
60 float zlp4=0.0;
61 float zlp5=0.0;
62 float zlp6=0.0;
63
64 -----
65 implementation of function
66 -----
67 void quad_mix(float in_buffer[], float re_buffer[], float im_buffer[])
68 {
69     int i;
70
71     /*_cosine/sine generation_*/
72     for(i=0;i<BUFFERSIZE_INPUT;i++)
73     {
74         /*feedback*/
75         zsc1=-(zsc2*asc2+zsc3);
76
77         /*forward*/
78         re_buffer[i]=zsc1+zsc2*bc3;/*cosine*/
79         im_buffer[i]=zsc2*bs2;/*sine*/
80
81         /*shift delays*/
82         zsc3=zsc2;
83         zsc2=zsc1;
84
85     }
86
87     /*_mixing_*/
88     for(i=0;i<BUFFERSIZE_INPUT;i++)
89     {
90         /*mixing*/
91         re_buffer[i]=re_buffer[i]*in_buffer[i];/*I-path*/
92         im_buffer[i]=im_buffer[i]*in_buffer[i];/*Q-path*/
93     }
94
95     /*_low pass_*/
96     for(i=0;i<BUFFERSIZE_INPUT;i++)
97     {
98         /*feedback*/
99         zlp1=re_buffer[i]-(alp2*zlp2+alp3*zlp3+alp4*zlp4+alp5*zlp5+alp6*zlp6);
100         /*r-path*/
101         zlp1=im_buffer[i]-(alp2*zlp2+alp3*zlp3+alp4*zlp4+alp5*zlp5+alp6*zlp6);
102         /*Q-path*/
103
104         /*forward*/
105         re_buffer[i]=zlp1+zlp2+alp3*zlp3+alp4*zlp4+alp5*zlp5+alp6*zlp6*
106         im_buffer[i]-(b1p1+zlp1+alp2*zlp2+alp3*zlp3+alp4*zlp4+alp5*zlp5+alp6*
107         zlp6);/*Q-path*/
108
109         /*shift delays*/
110         zlp6=zlp5;
111         zlp5=zlp4;
112         zlp4=zlp3;
113         zlp3=zlp2;
114         zlp2=zlp1;
115         /*Q-path*/
116         zlp6=zlp5;
117         zlp5=zlp4;
118         zlp4=zlp3;
119         zlp3=zlp2;
120         zlp2=zlp1;
121     }
122

```

```

1 */
2 QUADRATURE MIXER - VERSION FLOATING POINT 1.0
3
4 file : quad_mix.c
5 date : oct 2001 - Jan 2002
6 by : c. haller
7 f. schnyder
8
9 */
10
11 -----
12 includes
13 -----
14 #include "quad_mix.h"
15
16
17 -----
18 global variables
19 -----
20 /*coefficients of iir filter for cosine/sine generation*/
21
22 float bc2=-0.27697926505;/*cosine*/
23 float bs2=0.960875895593;/*sine*/
24 float asc2=-0.5539585301;/*sine/cosine*/
25
26 /*delay variables of iir filter for cosine/sine generation*/
27
28 float zsc1=0.0;
29 float zsc2=0.0;
30 float zsc3=1.0; /*for first value of delta function*/
31
32 /*coefficients of iir filter for low pass filtering*/
33
34 float b1p1=0.0015;
35 float b1p2=0.0076;
36 float b1p3=0.0151;
37 float b1p4=0.0151;
38 float b1p5=0.0076;
39 float b1p6=0.0015;
40 float alp2=-2.8957;
41 float alp3=3.6344;
42 float alp4=-2.3928;
43 float alp5=0.8176;
44 float alp6=-0.1151;
45
46 /*delay variables of iir filter for low pass filtering the I-path*/

```

## Listing E8: Floating-Point :: quad\_mix.c

```

123
124
125 /*
126
127 END OF THE FILE
128 */

```

## downsample\_one

Listing F.9: Floating-Point :: downsample\_one.h

```

1 /*
2  * DOWNSAMPLE ONE - VERSION FLOATING POINT 1.0
3  */
4 file : downsample_one.h
5 date : oct 2001 - jan 2002
6 by : c. haller
7 f. schnyder
8
9 */
10 #ifndef DOWNSAMPLE_ONE_
11 #define DOWNSAMPLE_ONE_
12
13 /*
14  *
15  * includes
16  *
17  * #include "global_settings.h"
18  *
19  *
20  * global functions
21  *
22  *
23  * The re_buffer and im_buffer are downsampled
24  * to the re_buffer_down_one and im_buffer_down_one.
25  *
26  * Parameter :
27  *
28  * re_buffer      buffer of I signal
29  * re_buffer_down_one buffer of Q signal
30  * im_buffer_down_one buffer of downsampled I signal
31  * im_buffer_down_one buffer of downsampled Q signal
32  */
33 void downsample_one(float re_buffer[], float im_buffer[], float
34 re_buffer_down_one[], float im_buffer_down_one[]);
35
36 #endif /* DOWNSAMPLE_ONE_ */
37
38 /*
39 END OF THE FILE
40 */

```

Listing F.10: Floating-Point :: downsample\_one.c

```

1 /*
2  * DOWNSAMPLE ONE - VERSION FLOATING POINT 1.0
3  */
4 file : downsample_one.c
5 date : oct 2001 - jan 2002
6 by : c. haller
7 f. schnyder
8
9 */
10
11 /*
12  *
13  * includes
14  *
15  * #include "downsample_one.h"
16  *
17  * implementation of function
18  *
19  *
20  *
21  *
22  *
23  *
24  *
25  *
26  *
27  *
28  *
29  *
30  *
31  *
32  *
33  *
34  *
35  *
36  *
37  *
38  *
39  *
40  *
41  *
42  *
43  *
44  *
45  *
46  *
47  *
48  *
49  *
50  *
51  *
52  *
53  *
54  *
55  *
56  *
57  *
58  *
59  *
60  *
61  *
62  *
63  *
64  *
65  *
66  *
67  *
68  *
69  *
70  *
71  *
72  *
73  *
74  *
75  *
76  *
77  *
78  *
79  *
80  *
81  *
82  *
83  *
84  *
85  *
86  *
87  *
88  *
89  *
90  *
91  *
92  *
93  *
94  *
95  *
96  *
97  *
98  *
99  *
100  *
101  *
102  *
103  *
104  *
105  *
106  *
107  *
108  *
109  *
110  *
111  *
112  *
113  *
114  *
115  *
116  *
117  *
118  *
119  *
120  *
121  *
122  *
123  *
124  *
125  *
126  *
127  *
128  *
129  *
130  *
131  *
132  *
133  *
134  *
135  *
136  *
137  *
138  *
139  *
140  *
141  *
142  *
143  *
144  *
145  *
146  *
147  *
148  *
149  *
150  *
151  *
152  *
153  *
154  *
155  *
156  *
157  *
158  *
159  *
160  *
161  *
162  *
163  *
164  *
165  *
166  *
167  *
168  *
169  *
170  *
171  *
172  *
173  *
174  *
175  *
176  *
177  *
178  *
179  *
180  *
181  *
182  *
183  *
184  *
185  *
186  *
187  *
188  *
189  *
190  *
191  *
192  *
193  *
194  *
195  *
196  *
197  *
198  *
199  *
200  *
201  *
202  *
203  *
204  *
205  *
206  *
207  *
208  *
209  *
210  *
211  *
212  *
213  *
214  *
215  *
216  *
217  *
218  *
219  *
220  *
221  *
222  *
223  *
224  *
225  *
226  *
227  *
228  *
229  *
230  *
231  *
232  *
233  *
234  *
235  *
236  *
237  *
238  *
239  *
240  *
241  *
242  *
243  *
244  *
245  *
246  *
247  *
248  *
249  *
250  *
251  *
252  *
253  *
254  *
255  *
256  *
257  *
258  *
259  *
260  *
261  *
262  *
263  *
264  *
265  *
266  *
267  *
268  *
269  *
270  *
271  *
272  *
273  *
274  *
275  *
276  *
277  *
278  *
279  *
280  *
281  *
282  *
283  *
284  *
285  *
286  *
287  *
288  *
289  *
290  *
291  *
292  *
293  *
294  *
295  *
296  *
297  *
298  *
299  *
300  *
301  *
302  *
303  *
304  *
305  *
306  *
307  *
308  *
309  *
310  *
311  *
312  *
313  *
314  *
315  *
316  *
317  *
318  *
319  *
320  *
321  *
322  *
323  *
324  *
325  *
326  *
327  *
328  *
329  *
330  *
331  *
332  *
333  *
334  *
335  *
336  *
337  *
338  *
339  *
340  *
341  *
342  *
343  *
344  *
345  *
346  *
347  *
348  *
349  *
350  *
351  *
352  *
353  *
354  *
355  *
356  *
357  *
358  *
359  *
360  *
361  *
362  *
363  *
364  *
365  *
366  *
367  *
368  *
369  *
370  *
371  *
372  *
373  *
374  *
375  *
376  *
377  *
378  *
379  *
380  *
381  *
382  *
383  *
384  *
385  *
386  *
387  *
388  *
389  *
390  *
391  *
392  *
393  *
394  *
395  *
396  *
397  *
398  *
399  *
400  *
401  *
402  *
403  *
404  *
405  *
406  *
407  *
408  *
409  *
410  *
411  *
412  *
413  *
414  *
415  *
416  *
417  *
418  *
419  *
420  *
421  *
422  *
423  *
424  *
425  *
426  *
427  *
428  *
429  *
430  *
431  *
432  *
433  *
434  *
435  *
436  *
437  *
438  *
439  *
440  *
441  *
442  *
443  *
444  *
445  *
446  *
447  *
448  *
449  *
450  *
451  *
452  *
453  *
454  *
455  *
456  *
457  *
458  *
459  *
460  *
461  *
462  *
463  *
464  *
465  *
466  *
467  *
468  *
469  *
470  *
471  *
472  *
473  *
474  *
475  *
476  *
477  *
478  *
479  *
480  *
481  *
482  *
483  *
484  *
485  *
486  *
487  *
488  *
489  *
490  *
491  *
492  *
493  *
494  *
495  *
496  *
497  *
498  *
499  *
500  *
501  *
502  *
503  *
504  *
505  *
506  *
507  *
508  *
509  *
510  *
511  *
512  *
513  *
514  *
515  *
516  *
517  *
518  *
519  *
520  *
521  *
522  *
523  *
524  *
525  *
526  *
527  *
528  *
529  *
530  *
531  *
532  *
533  *
534  *
535  *
536  *
537  *
538  *
539  *
540  *
541  *
542  *
543  *
544  *
545  *
546  *
547  *
548  *
549  *
550  *
551  *
552  *
553  *
554  *
555  *
556  *
557  *
558  *
559  *
560  *
561  *
562  *
563  *
564  *
565  *
566  *
567  *
568  *
569  *
570  *
571  *
572  *
573  *
574  *
575  *
576  *
577  *
578  *
579  *
580  *
581  *
582  *
583  *
584  *
585  *
586  *
587  *
588  *
589  *
590  *
591  *
592  *
593  *
594  *
595  *
596  *
597  *
598  *
599  *
600  *
601  *
602  *
603  *
604  *
605  *
606  *
607  *
608  *
609  *
610  *
611  *
612  *
613  *
614  *
615  *
616  *
617  *
618  *
619  *
620  *
621  *
622  *
623  *
624  *
625  *
626  *
627  *
628  *
629  *
630  *
631  *
632  *
633  *
634  *
635  *
636  *
637  *
638  *
639  *
640  *
641  *
642  *
643  *
644  *
645  *
646  *
647  *
648  *
649  *
650  *
651  *
652  *
653  *
654  *
655  *
656  *
657  *
658  *
659  *
660  *
661  *
662  *
663  *
664  *
665  *
666  *
667  *
668  *
669  *
670  *
671  *
672  *
673  *
674  *
675  *
676  *
677  *
678  *
679  *
680  *
681  *
682  *
683  *
684  *
685  *
686  *
687  *
688  *
689  *
690  *
691  *
692  *
693  *
694  *
695  *
696  *
697  *
698  *
699  *
700  *
701  *
702  *
703  *
704  *
705  *
706  *
707  *
708  *
709  *
710  *
711  *
712  *
713  *
714  *
715  *
716  *
717  *
718  *
719  *
720  *
721  *
722  *
723  *
724  *
725  *
726  *
727  *
728  *
729  *
730  *
731  *
732  *
733  *
734  *
735  *
736  *
737  *
738  *
739  *
740  *
741  *
742  *
743  *
744  *
745  *
746  *
747  *
748  *
749  *
750  *
751  *
752  *
753  *
754  *
755  *
756  *
757  *
758  *
759  *
760  *
761  *
762  *
763  *
764  *
765  *
766  *
767  *
768  *
769  *
770  *
771  *
772  *
773  *
774  *
775  *
776  *
777  *
778  *
779  *
780  *
781  *
782  *
783  *
784  *
785  *
786  *
787  *
788  *
789  *
790  *
791  *
792  *
793  *
794  *
795  *
796  *
797  *
798  *
799  *
800  *
801  *
802  *
803  *
804  *
805  *
806  *
807  *
808  *
809  *
810  *
811  *
812  *
813  *
814  *
815  *
816  *
817  *
818  *
819  *
820  *
821  *
822  *
823  *
824  *
825  *
826  *
827  *
828  *
829  *
830  *
831  *
832  *
833  *
834  *
835  *
836  *
837  *
838  *
839  *
840  *
841  *
842  *
843  *
844  *
845  *
846  *
847  *
848  *
849  *
850  *
851  *
852  *
853  *
854  *
855  *
856  *
857  *
858  *
859  *
860  *
861  *
862  *
863  *
864  *
865  *
866  *
867  *
868  *
869  *
870  *
871  *
872  *
873  *
874  *
875  *
876  *
877  *
878  *
879  *
880  *
881  *
882  *
883  *
884  *
885  *
886  *
887  *
888  *
889  *
890  *
891  *
892  *
893  *
894  *
895  *
896  *
897  *
898  *
899  *
900  *
901  *
902  *
903  *
904  *
905  *
906  *
907  *
908  *
909  *
910  *
911  *
912  *
913  *
914  *
915  *
916  *
917  *
918  *
919  *
920  *
921  *
922  *
923  *
924  *
925  *
926  *
927  *
928  *
929  *
930  *
931  *
932  *
933  *
934  *
935  *
936  *
937  *
938  *
939  *
940  *
941  *
942  *
943  *
944  *
945  *
946  *
947  *
948  *
949  *
950  *
951  *
952  *
953  *
954  *
955  *
956  *
957  *
958  *
959  *
960  *
961  *
962  *
963  *
964  *
965  *
966  *
967  *
968  *
969  *
970  *
971  *
972  *
973  *
974  *
975  *
976  *
977  *
978  *
979  *
980  *
981  *
982  *
983  *
984  *
985  *
986  *
987  *
988  *
989  *
990  *
991  *
992  *
993  *
994  *
995  *
996  *
997  *
998  *
999  *
1000  */

```

```

19 void downsample_one(float re_buffer[], float im_buffer[], float
20 re_buffer_down_one[], float im_buffer_down_one[])
21 {
22     int i;
23     for(i=0;i<BUFFERSIZE_DRMODULATOR;i++)
24     {
25         re_buffer_down_one[i]=re_buffer[i]*DOWNSAMPLE_ONE;
26         im_buffer_down_one[i]=im_buffer[i]*DOWNSAMPLE_ONE;
27     }
28 }
29
30 /*
31 END OF THE FILE
32 */

```

## mixed\_demodulate

Listing F.11: Floating-Point :: mixed\_demodulate.h

```

1 /*
2  * MIXED DEMODULATE - VERSION FLOATING POINT 1.0
3  */
4 file : mixed_demodulate.h
5 date : oct 2001 - jan 2002
6 by : c. haller
7 f. schnyder
8
9 */
10
11 #ifndef MIXED_DEMODULATE_
12 #define MIXED_DEMODULATE_
13
14 /*
15  *
16  * includes
17  *
18  * #include "global_settings.h"
19  *
20  * #include <math.h>
21  *
22  *
23  * global functions
24  *
25  *
26  * The re_buffer_down_one and im_buffer_down_one are demodulated
27  * with the mixed demodulator algorithm to out_buffer_down_one.
28  *
29  * Parameters :
30  *
31  * re_buffer_down_one buffer of downsampled I signal
32  * im_buffer_down_one buffer of downsampled Q signal
33  * out_buffer_down_one buffer of demodulated signal
34  *
35  * void mixed_demodulate(float re_buffer_down_one[], float im_buffer_down_one[],
36  * float out_buffer_down_one[])
37  *
38  * #endif /* MIXED_DEMODULATE_ */
39
40 /*
41 END OF THE FILE
42 */

```

Listing F.12: Floating-Point :: mixed\_demodulate.c

```

1 /*
2  * MIXED DEMODULATOR - VERSION FLOATING POINT 1.0
3  */
4 file : mixed_demodulate.c
5 date : oct 2001 - jan 2002
6 by : c. haller
7 f. schnyder
8
9 */
10
11 /*
12  *
13  *
14  *
15  *
16  *
17  *
18  *
19  *
20  *
21  *
22  *
23  *
24  *
25  *
26  *
27  *
28  *
29  *
30  *
31  *
32  *
33  *
34  *
35  *
36  *
37  *
38  *
39  *
40  *
41  *
42  *
43  *
44  *
45  *
46  *
47  *
48  *
49  *
50  *
51  *
52  *
53  *
54  *
55  *
56  *
57  *
58  *
59  *
60  *
61  *
62  *
63  *
64  *
65  *
66  *
67  *
68  *
69  *
70  *
71  *
72  *
73  *
74  *
75  *
76  *
77  *
78  *
79  *
80  *
81  *
82  *
83  *
84  *
85  *
86  *
87  *
88  *
89  *
90  *
91  *
92  *
93  *
94  *
95  *
96  *
97  *
98  *
99  *
100  *
101  *
102  *
103  *
104  *
105  *
106  *
107  *
108  *
109  *
110  *
111  *
112  *
113  *
114  *
115  *
116  *
117  *
118  *
119  *
120  *
121  *
122  *
123  *
124  *
125  *
126  *
127  *
128  *
129  *
130  *
131  *
132  *
133  *
134  *
135  *
136  *
137  *
138  *
139  *
140  *
141  *
142  *
143  *
144  *
145  *
146  *
147  *
148  *
149  *
150  *
151  *
152  *
153  *
154  *
155  *
156  *
157  *
158  *
159  *
160  *
161  *
162  *
163  *
164  *
165  *
166  *
167  *
168  *
169  *
170  *
171  *
172  *
173  *
174  *
175  *
176  *
177  *
178  *
179  *
180  *
181  *
182  *
183  *
184  *
185  *
186  *
187  *
188  *
189  *
190  *
191  *
192  *
193  *
194  *
195  *
196  *
197  *
198  *
199  *
200  *
201  *
202  *
203  *
204  *
205  *
206  *
207  *
208  *
209  *
210  *
211  *
212  *
213  *
214  *
215  *
216  *
217  *
218  *
219  *
220  *
221  *
222  *
223  *
224  *
225  *
226  *
227  *
228  *
229  *
230  *
231  *
232  *
233  *
234  *
235  *
236  *
237  *
238  *
239  *
240  *
241  *
242  *
243  *
244  *
245  *
246  *
247  *
248  *
249  *
250  *
251  *
252  *
253  *
254  *
255  *
256  *
257  *
258  *
259  *
260  *
261  *
262  *
263  *
264  *
265  *
266  *
267  *
268  *
269  *
270  *
271  *
272  *
273  *
274  *
275  *
276  *
277  *
278  *
279  *
280  *
281  *
282  *
283  *
284  *
285  *
286  *
287  *
288  *
289  *
290  *
291  *
292  *
293  *
294  *
295  *
296  *
297  *
298  *
299  *
300  *
301  *
302  *
303  *
304  *
305  *
306  *
307  *
308  *
309  *
310  *
311  *
312  *
313  *
314  *
315  *
316  *
317  *
318  *
319  *
320  *
321  *
322  *
323  *
324  *
325  *
326  *
327  *
328  *
329  *
330  *
331  *
332  *
333  *
334  *
335  *
336  *
337  *
338  *
339  *
340  *
341  *
342  *
343  *
344  *
345  *
346  *
347  *
348  *
349  *
350  *
351  *
352  *
353  *
354  *
355  *
356  *
357  *
358  *
359  *
360  *
361  *
362  *
363  *
364  *
365  *
366  *
367  *
368  *
369  *
370  *
371  *
372  *
373  *
374  *
375  *
376  *
377  *
378  *
379  *
380  *
381  *
382  *
383  *
384  *
385  *
386  *
387  *
388  *
389  *
390  *
391  *
392  *
393  *
394  *
395  *
396  *
397  *
398  *
399  *
400  *
401  *
402  *
403  *
404  *
405  *
406  *
407  *
408  *
409  *
410  *
411  *
412  *
413  *
414  *
415  *
416  *
417  *
418  *
419  *
420  *
421  *
422  *
423  *
424  *
425  *
426  *
427  *
428  *
429  *
430  *
431  *
432  *
433  *
434  *
435  *
436  *
437  *
438  *
439  *
440  *
441  *
442  *
443  *
444  *
445  *
446  *
447  *
448  *
449  *
450  *
451  *
452  *
453  *
454  *
455  *
456  *
457  *
458  *
459  *
460  *
461  *
462  *
463  *
464  *
465  *
466  *
467  *
468  *
469  *
470  *
471  *
472  *
473  *
474  *
475  *
476  *
477  *
478  *
479  *
480  *
481  *
482  *
483  *
484  *
485  *
486  *
487  *
488  *
489  *
490  *
491  *
492  *
493  *
494  *
495  *
496  *
497  *
498  *
499  *
500  *
501  *
502  *
503  *
504  *
505  *
506  *
507  *
508  *
509  *
510  *
511  *
512  *
513  *
514  *
515  *
516  *
517  *
518  *
519  *
520  *
521  *
522  *
523  *
524  *
525  *
526  *
527  *
528  *
529  *
530  *
531  *
532  *
533  *
534  *
535  *
536  *
537  *
538  *
539  *
540  *
541  *
542  *
543  *
544  *
545  *
546  *
547  *
548  *
549  *
550  *
551  *
552  *
553  *
554  *
555  *
556  *
557  *
558  *
559  *
560  *
561  *
562  *
563  *
564  *
565  *
566  *
567  *
568  *
569  *
570  *
571  *
572  *
573  *
574  *
575  *
576  *
577  *
578  *
579  *
580  *
581  *
582  *
583  *
584  *
585  *
586  *
587  *
588  *
589  *
590  *
591  *
592  *
593  *
594  *
595  *
596  *
597  *
598  *
599  *
600  *
601  *
602  *
603  *
604  *
605  *
606  *
607  *
608  *
609  *
610  *
611  *
612  *
613  *
614  *
615  *
616  *
617  *
618  *
619  *
620  *
621  *
622  *
623  *
624  *
625  *
626  *
627  *
628  *
629  *
630  *
631  *
632  *
633  *
634  *
635  *
636  *
637  *
638  *
639  *
640  *
641  *
642  *
643  *
644  *
645  *
646  *
647  *
648  *
649  *
650  *
651  *
652  *
653  *
654  *
655  *
656  *
657  *
658  *
659  *
660  *
661  *
662  *
663  *
664  *
665  *
666  *
667  *
668  *
669  *
670  *
671  *
672  *
673  *
674  *
675  *
676  *
677  *
678  *
679  *
680  *
681  *
682  *
683  *
684  *
685  *
686  *
687  *
688  *
689  *
690  *
691  *
692  *
693  *
694  *
695  *
696  *
697  *
698  *
699  *
700  *
701  *
702  *
703  *
704  *
705  *
706  *
707  *
708  *
709  *
710  *
711  *
712  *
713  *
714  *
715  *
716  *
717  *
718  *
719  *
720  *
721  *
722  *
723  *
724  *
725  *
726  *
727  *
728  *
729  *
730  *
731  *
732  *
733  *
734  *
735  *
736  *
737  *
738  *
739  *
740  *
741  *
742  *
743  *
744  *
745  *
746  *
747  *
748  *
749  *
750  *
751  *
752  *
753  *
754  *
755  *
756  *
757  *
758  *
759  *
760  *
761  *
762  *
763  *
764  *
765  *
766  *
767  *
768  *
769  *
770  *
771  *
772  *
773  *
774  *
775  *
776  *
777  *
778  *
779  *
780  *
781  *
782  *
783  *
784  *
785  *
786  *
787  *
788  *
789  *
790  *
791  *
792  *
793  *
794  *
795  *
796  *
797  *
798  *
799  *
800  *
801  *
802  *
803  *
804  *
805  *
806  *
807  *
808  *
809  *
810  *
811  *
812  *
813  *
814  *
815  *
816  *
817  *
818  *
819  *
820  *
821  *
822  *
823  *
824  *
825  *
826  *
827  *
828  *
829  *
830  *
831  *
832  *
833  *
834  *
835  *
836  *
837  *
838  *
839  *
840  *
841  *
842  *
843  *
844  *
845  *
846  *
847  *
848  *
849  *
850  *
851  *
852  *
853  *
854  *
855  *
856
```

```

12  includes
13  -----*/
14  #include "mixed_demodulate.h"
15
16  /*
17  -----
18  global variables
19  -----*/
20  /*variables of demodulating (attention division by zero!)*/*
21  float coold=0.0; /*old value of I-path*/
22  float conew=1.0; /*new value of I-path*/
23  float siold=0.0; /*old value of Q-path*/
24  float sinew=1.0; /*new value of Q-path*/
25  float demo=0.0; /*value demodulating*/
26
27  /*constant of demodulating*/
28
29  float decon=0.7;
30
31  /*
32  -----
33  implementation of function
34  -----*/
35  void mixed_demodulate(float re_buffer_down_one[], float im_buffer_down_one[],
36                        float out_buffer_down_one[])
37  {
38      int i;
39      for(i=0; i<BUFFERSIZE_DEMODULATOR; i++)
40      {
41          /*shifting values_*/
42          coold=conew;
43          siold=sinew;
44          conew=re_buffer_down_one[i];
45          sinew=im_buffer_down_one[i];
46
47          /*calculating demodulated value_*/
48          demo=(coold*siold+sinew*conew)/(coold*conew+siold*sinew); /*atan argument*/
49          democatf(demo); /*atan value*/
50          demodemo*decon; /*demodulated value*/
51          out_buffer_down_one[i]=demo;
52      }
53  }
54
55  /*
56  -----
57  END OF THE FILE
58  -----*/

```

## pll\_demodulate

### Listing F.13: Floating-Point :: pll\_demodulate.h

```

1  /*
2  PLL DEMODULATOR - VERSION FLOATING POINT 1.0
3
4  file : pll_demodulate.h
5  date : oct 2001 - jan 2002
6  by : c. haller
7       f. schnyder
8
9  */
10
11  #ifndef _PLL_DEMODULATE_
12  #define _PLL_DEMODULATE_
13
14  /*
15  -----
16  includes
17  -----*/
18  #include "global_settings.h"
19  #include <math.h>
20
21  /*
22  -----
23  global functions
24  -----*/

```

```

23  /*
24  The re_buffer_down_one and im_buffer_down_one are demodulated
25  with the pll algorithm to the out_buffer_down_one.
26
27  Parameters :
28
29  re_buffer_down_one buffer of downsampled I signal
30  im_buffer_down_one buffer of downsampled Q signal
31  out_buffer_down_one buffer of demodulated signal
32  */
33  void pll_demodulate(float re_buffer_down_one[], float im_buffer_down_one[],
34                    float out_buffer_down_one[]);
35
36  #endif /* _PLL_DEMODULATE_ */
37
38  /*
39  -----
40  END OF THE FILE
41  -----*/

```

### Listing F.14: Floating-Point :: pll\_demodulate.c

```

1  /*
2  PLL DEMODULATOR - VERSION FLOATING POINT 1.0
3
4  file : pll_demodulate.c
5  date : oct 2001 - jan 2002
6  by : c. haller
7       f. schnyder
8
9  */
10
11  /*
12  -----
13  includes
14  -----*/
15  #include "pll_demodulate.h"
16
17  /*
18  -----
19  global variables
20  -----*/
21  /*constants for demodulate*/
22  float p_pll=2.0;
23  float k_pll=2.5;
24  float gain_pll=1.5;
25
26  /*variables for demodulate*/
27  float arg=0.0; /*argument of sine and cosine*/
28  float i_path=0.0; /*I path*/
29  float q_path=0.0; /*Q path*/
30
31  /*
32  -----
33  implementation of function
34  -----*/
35  void pll_demodulate(float re_buffer_down_one[], float im_buffer_down_one[],
36                    float out_buffer_down_one[])
37  {
38      int i;
39
40      float demo; /*value of demodulation*/
41      for(i=0; i<BUFFERSIZE_DEMODULATOR; i++)
42      {
43          /*calculation of demodulation_*/
44          i_path=re_buffer_down_one[i]*sinf(arg);
45          q_path=im_buffer_down_one[i]*cosf(arg);
46          demo=q_path-i_path;
47          out_buffer_down_one[i]=demo*p_pll;
48
49          /*summation of argument_*/
50          arg=arg+k_pll*demo;
51
52          /*limitation of argument_*/
53          if (arg>2*PI)
54          {
55              (

```



```

57     arg=arg-2*PI;
58 }
59
60 else if (arg<-2*PI)
61 {
62     arg=arg+2*PI;
63 }
64 }
65 }
66
67 /*
68  END OF THE FILE
69 */
70

```

**out\_filter**

```

1  /*
2  OUTPUT_FILTER - VERSION FLOATING POINT 1.0
3
4  file : out_filter.h
5  date : oct 2001 - Jan 2002
6  by   : c. haller
7       f. schnyder
8  */
9
10 #ifndef _OUT_FILTER_
11 #define _OUT_FILTER_
12
13 /*
14  includes
15
16  #include "global_settings.h"
17
18  */
19
20 global functions
21
22 /*
23  The out_buffer_down_one is band pass filtered.
24
25  Parameters :
26
27  out_buffer_down_one  buffer of demodulated signal
28
29  */
30 void out_filter(float out_buffer_down_one[]);
31
32 #endif /* _OUT_FILTER_ */
33
34 /*
35  END OF THE FILE
36 */

```

**Listing F15: Floating-Point :: out\_filter.h**

```

1  /*
2  OUTPUT_FILTER - VERSION FLOATING POINT 1.0
3
4  file : out_filter.h
5  date : oct 2001 - Jan 2002
6  by   : c. haller
7       f. schnyder
8  */
9
10 #ifndef _OUT_FILTER_
11 #define _OUT_FILTER_
12
13 /*
14  includes
15
16  #include "global_settings.h"
17
18  */
19
20 global functions
21
22 /*
23  The out_buffer_down_one is band pass filtered.
24
25  Parameters :
26
27  out_buffer_down_one  buffer of demodulated signal
28
29  */
30 void out_filter(float out_buffer_down_one[]);
31
32 #endif /* _OUT_FILTER_ */
33
34 /*
35  END OF THE FILE
36 */

```

**Listing F16: Floating-Point :: out\_filter.c**

```

1  /*
2  OUTPUT_FILTER - VERSION FLOATING POINT 1.0
3
4  file : out_filter.c
5  date : oct 2001 - Jan 2002
6  by   : c. haller
7       f. schnyder
8  */
9
10 /*
11  includes
12
13  #include "out_filter.h"
14
15

```

```

16  */
17 /*
18  global variables
19
20  */
21 /*
22  coefficients of output filter (band pass) after demodulation
23
24  */
25 /*band pass order : 10 */
26 /*cutoff frequency 1 : 200 Hz */
27 /*cutoff frequency 2 : 3600 Hz */
28
29 /*coefficients of iir filtering for band pass filtering*/
30 float abp2=-5.36553964549091;
31 float abp3=-12.78395381803902;
32 float abp4=-18.34338736253170;
33 float abp5=-18.18669891324587;
34 float abp6=-13.24200959389255;
35 float abp7=7.05286413754873;
36 float abp8=-2.65500194444943;
37 float abp9=0.69066231776403;
38 float abp10=-0.11583317016500;
39 float abp11=0.00759591472846;
40 float bbp1=0.02788206889508;
41 float bbp2=0.0;
42 float bbp3=-0.13941034447538;
43 float bbp4=0.0;
44 float bbp5=0.27882068895075;
45 float bbp6=0.0;
46 float bbp7=-0.27882068895075;
47 float bbp8=0.0;
48 float bbp9=0.13941034447538;
49 float bbp10=0.0;
50 float bbp11=-0.02788206889508;
51
52 /*delay variables of iir filtering for band pass filtering*/
53 float zbp1=0.0;
54 float zbp2=0.0;
55 float zbp3=0.0;
56 float zbp4=0.0;
57 float zbp5=0.0;
58 float zbp6=0.0;
59 float zbp7=0.0;
60 float zbp8=0.0;
61 float zbp9=0.0;
62 float zbp10=0.0;
63 float zbp11=0.0;
64
65 /*
66  implementation of function
67
68  */
69 void out_filter(float out_buffer_down_one[])
70 {
71     int i;
72
73     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
74     {
75         /*feedback*/
76         zbp1=out_buffer_down_one[i]- (abp2*zbp2+abp3*zbp3+abp4*zbp4+abp5*zbp5+abp6*
77             zbp6+abp7*zbp7+abp8*zbp8+abp9*zbp9+abp10*zbp10+abp11*zbp11);
78
79         /*forward*/
80         out_buffer_down_one[i]=bbp1*zbp1+bbp2*zbp2+bbp3*zbp3+bbp4*zbp4+bbp5*zbp5+
81             bbp6*zbp6+bbp7*zbp7+bbp8*zbp8+bbp9*zbp9+bbp10*zbp10+bbp11*zbp11;
82
83         /*shift delays*/
84         zbp11=zbp10;
85         zbp10=zbp9;
86         zbp9=zbp8;
87         zbp8=zbp7;
88         zbp7=zbp6;
89         zbp6=zbp5;
90         zbp5=zbp4;
91         zbp4=zbp3;
92         zbp3=zbp2;
93         zbp2=zbp1;
94     }
95 }

```

```

94     }
95
96 /*
97  END OF THE FILE
98 */

```

## downsample\_two

*Listing F.17: Floating-Point :: downsample\_two.h*

```

1  /*
2  DOWNSAMPLE_TWO - VERSION FLOATING POINT 1.0
3
4  file : downsample_two.h
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8
9  */
10
11 #ifndef DOWNSAMPLE_TWO_
12 #define DOWNSAMPLE_TWO_
13
14 /*-----
15 includes
16 -----*/
17 #include "global_settings.h"
18
19 /*-----
20 global functions
21 -----*/
22
23 /* The out_buffer_down_one is downsampled to the
24    out_buffer_down_two.
25
26 Parameters :
27
28 out_buffer_down_one buffer of demodulated signal.
29 out_buffer_down_two buffer of downsampled demodulated signal
30
31 */
32 void downsample_two(float out_buffer_down_one[], float out_buffer_down_two[]);
33 #endif /* DOWNSAMPLE_TWO_ */
34
35 /*
36  END OF THE FILE
37
38 */

```

*Listing F.18: Floating-Point :: downsample\_two.c*

```

1  /*
2  DOWNSAMPLE_TWO - VERSION FLOATING POINT 1.0
3
4  file : downsample_two.c
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8
9  */
10
11 /*-----
12 includes
13 -----*/
14 #include "downsample_two.h"
15
16 /*-----
17 implementation of function
18 -----*/
19 void downsample_two(float out_buffer_down_one[], float out_buffer_down_two[])
20 {
21     int i;
22

```

```

23     for(i=0;i<BUFFERSIZE_OUTPUT;i++)
24     {
25         out_buffer_down_two[i]=out_buffer_down_one[i*DOWNSAMPLE_TWO];
26     }
27
28
29 /*
30  END OF THE FILE
31 */

```

## F.1.3. Fixed-Point

### global\_settings

*Listing F.19: Fixed-Point :: global\_settings.h*

```

1  /*
2  GLOBAL_SETTINGS - - VERSION FIXED POINT 0.5
3
4  file : global_settings.h
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8
9  */
10
11 #define BUFFERSIZE 128
12 #define DOWNSAMPLE_ONE 4
13 #define DOWNSAMPLE_TWO 2
14
15 #define BUFFERSIZE_INPUT (BUFFERSIZE)
16 #define BUFFERSIZE_DEMODULATOR (BUFFERSIZE/DOWNSAMPLE_ONE)
17 #define BUFFERSIZE_OUTPUT (BUFFERSIZE/(DOWNSAMPLE_ONE*DOWNSAMPLE_TWO))
18
19 #define PI 3.14159265359

```

### fm\_dem\_main

*Listing F.20: Fixed-Point :: fm\_dem\_main.c*

```

1  /*
2  FM_DEMODULATION_MAIN - VERSION FIXED POINT 0.5
3
4  file : fm_dem_main.c
5  date : nov 2001
6  by   : c. haller
7        f. schnyder
8
9  */
10
11 /*-----
12 demodulator define
13 -----*/
14 #define FM_MIXED_DEM 10
15 #define FM_PLL_DEM 20
16
17 #define FM_DEMODULATOR FM_MIXED_DEM
18
19 /*-----
20 includes
21 -----*/
22
23 #include <std.h>
24 #include <swi.h>
25 #include <cs1.h>
26 #include <bs1.h>
27
28 #include "global_settings.h"
29 #include "adc_TMS1408.h"

```

```

30 #include "dac_codec.h"
31 #include "quad_mix.h"
32 #include "out_filter.h"
33 #if (FM_DEMODULATOR==FM_PLL_DEM)
34 #include "pll_demodulate.h"
35 #elif (FM_DEMODULATOR==FM_MIXED_DEM)
36 #include "mixed_demodulate.h"
37 #else
38 #error No demodulator algorithm defined
39 #endif
40
41 /*-----
42 debug defines
43 -----*/
44 #define FM_DEM_CF_DEBUG 0
45 /* to print comments to debug_log */
46
47 #define FM_DEM_CF_INT_OCCUR_LED 0
48 /* LED toggles when an interrupt occurs */
49 /* LED 1 codec interrupt (output) */
50 /* LED 2 edma interrupt (input) */
51 /* LED 3 timing error */
52
53 #define FM_DEM_CF_INT_TIME_LED 0
54 /* LED ON by interrupt start OFF at the end */
55 /* LED 1 codec interrupt (output) */
56 /* LED 2 edma interrupt (input) */
57 /* LED 3 software interrupt */
58
59 #define FM_DEM_CF_INT_TIME_STS 0
60 /* Enable STS Objects: */
61 /* -dac_sts */
62 /* -demodulate_sts */
63 /* -edma_sts */
64 /* to measure the duration time of the interrupts */
65
66 #define FM_DEM_CF_FUNC_TIME_STS 1
67 /* Enable STS Objects: */
68 /* -mix_sts */
69 /* -dem_sts */
70 /* -filter_sts */
71 /* to measure the duration time of the functions */
72
73 #if FM_DEM_CF_DEBUG
74 #include <log.h>
75 extern far LOG_Obj debug_log;
76 #endif
77
78 #if FFM_DEM_CF_INT_OCCUR_LED
79 #include <bal_led.h>
80 #endif
81
82 #if FFM_DEM_CF_INT_TIME_LED
83 #include <bal_led.h>
84 #endif
85
86 #if FM_DEM_CF_INT_TIME_STS
87 #include <clk.h>
88 #include <sts.h>
89 extern far STS_Obj dac_sts;
90 extern far STS_Obj demodulate_sts;
91 extern far STS_Obj edma_sts;
92 #endif
93
94 #if FM_DEM_CF_FUNC_TIME_STS
95 #include <clk.h>
96 #include <sts.h>
97 extern far STS_Obj mix_sts;
98 extern far STS_Obj dem_sts;
99 extern far STS_Obj filter_sts;
100 #endif
101
102 /*-----
103 DSP/BIOS extern variables
104 -----*/
105
106 extern far SWI_Obj demodulate_sw;
107 extern HMI_enable();
108 extern HMI_disable();
109
110 /*-----
111 global variables
112 -----*/
113
114 short in_buffer_A[BUFFERSIZE_INPUT];
115 short in_buffer_B[BUFFERSIZE_INPUT];
116
117 short re_buffer[BUFFERSIZE_INPUT];
118 short im_buffer[BUFFERSIZE_INPUT];
119
120 short out_buffer[BUFFERSIZE_DEMODULATOR];
121 short out_buffer_A[BUFFERSIZE_OUTPUT];
122 short out_buffer_B[BUFFERSIZE_OUTPUT];
123
124
125 short* in_buffer_ptr;
126 short* out_buffer_ptr;
127 Bool a_inEDMA=TRUE;
128 short outcounter=0;
129
130 /*-----
131 the main function
132 -----*/
133
134 void main()
135 {
136     HMI_disable();
137
138     #if FM_DEM_CF_DEBUG
139         LOG_printf(&debug_log,"Info : Begin Of main ");
140     #endif
141
142     CSI_init(); /* on chip init */
143     BSI_init(); /* on board init */
144
145     ths1408_init(in_buffer_A,in_buffer_B);
146     codec_init();
147
148     HMI_enable();
149     ths1408_start();
150     codec_start();
151
152     #if FM_DEM_CF_INT_TIME_LED
153         LED_off(LED_ALL);
154     #endif
155
156     #if FM_DEM_CF_DEBUG
157         LOG_printf(&debug_log,"Info : End Of main ");
158     #endif
159 }
160
161 /*-----
162 hardware interrupt functions
163 -----*/
164
165 void edmaInt(void)
166 {
167     #if FM_DEM_CF_INT_OCCUR_LED
168         LED_toggle(LED_2);
169     #endif
170
171     #if FM_DEM_CF_INT_TIME_LED
172         LED_on(LED_2);
173     #endif
174
175     #if FM_DEM_CF_INT_TIME_STS
176         STS_set(&edma_sts,CLK_gettime());
177     #endif
178
179     /* reset outcounter */
180     outcounter=0;
181
182     /* select EDMA channel */
183     #if (EDMA_intTest(THS1408_BUFFER_A_TCC))
184     {
185         #if FM_DEM_CF_DEBUG
186             LOG_printf(&debug_log,"In_A");
187         #endif
188         /* Clear transfer completion interrupt flag */
189     }
190 }

```

```

190 EDMA_intClear(THS1408_BUFFER_A_TCC);
191 /* set buffer pointers */
192 a_inEDMA=FALSE;
193 in_buffer_ptr=in_buffer_A;
194 out_buffer_ptr=out_buffer_A;
195 /* post software interrupt -> start demodulation*/
196 SWI_post(&demodulate_swi);
197 }
198 }
199
200 if (EDMA_intTest(THS1408_BUFFER_B_TCC))
201 {
202     #if PM_DEM_CF_DEBUG
203     LOG_printf(&debug_log," In B");
204     #endif
205     /* Clear transfer completion interrupt flag */
206     EDMA_intClear(THS1408_BUFFER_B_TCC);
207
208     /* set buffer pointers */
209     a_inEDMA=TRUE;
210     in_buffer_ptr=in_buffer_B;
211     out_buffer_ptr=out_buffer_B;
212
213     /* post software interrupt -> start demodulation*/
214     SWI_post(&demodulate_swi);
215 }
216
217 #if PM_DEM_CF_INT_TIME_STS
218 STS_delta(&edma_sts,CLK_gettime());
219 #endif
220
221 #if PM_DEM_CF_INT_TIME_LED
222 LED_off(LED_2);
223 #endif
224 }
225
226 void dacInt(void)
227 {
228     #if PM_DEM_CF_INT_OCCUR_LED
229     LED_toggle(LED_1);
230     #endif
231
232     #if PM_DEM_CF_INT_TIME_LED
233     LED_on(LED_1);
234     #endif
235
236     #if PM_DEM_CF_INT_TIME_STS
237     STS_set(&dac_sts,CLK_gettime());
238     #endif
239
240     /* interpolate a missing sample */
241     if (outcounter>15)
242     {
243         #if PM_DEM_CF_INT_OCCUR_LED
244         LED_toggle(LED_3);
245         #endif
246         /* select the buffer */
247         if(a_inEDMA)
248         {
249             AD535_HWI_write(outAD535h,(( out_buffer_A[15]+out_buffer_B[0])/2));
250             #if PM_DEM_CF_DEBUG
251             LOG_printf(&debug_log,"Out A error :%d",outcounter);
252             #endif
253         }
254         else
255         {
256             AD535_HWI_write(outAD535h,((out_buffer_B[15]+out_buffer_A[0])/2));
257             #if PM_DEM_CF_DEBUG
258             LOG_printf(&debug_log,"Out B error :%d",outcounter);
259             #endif
260         }
261         /* output a value */
262         /* select the buffer */
263         if(a_inEDMA)
264         {
265             AD535_HWI_write(outAD535h,out_buffer_A[outcounter]);
266             #if PM_DEM_CF_DEBUG
267             LOG_printf(&debug_log,"Out A :%d",outcounter);
268             #endif
269         }
270         else
271         {
272             AD535_HWI_write(outAD535h,out_buffer_B[outcounter]);
273             #if PM_DEM_CF_DEBUG
274             LOG_printf(&debug_log,"Out B :%d",outcounter);
275             #endif
276         }
277         outcounter++;
278     }
279 }
280
281 #if PM_DEM_CF_INT_TIME_STS
282 STS_delta(&dem_sts,CLK_gettime());
283 #endif
284
285 #if PM_DEM_CF_INT_TIME_LED
286 LED_off(LED_1);
287 #endif
288 }
289
290 -----
291 software interrupt functions
292 -----
293
294 void demodulateSwiFunc(void)
295 {
296     int i;
297
298     #if PM_DEM_CF_INT_TIME_LED
299     LED_on(LED_3);
300     #endif
301
302     #if PM_DEM_CF_INT_TIME_STS
303     STS_set(&demodulate_sts,CLK_gettime());
304     #endif
305
306     for (i=0 ;i<BUFFERSIZE_INPUT;i++)
307     {
308         in_buffer_ptr[i]=in_buffer_ptr[i]<<2;
309     }
310
311     #if PM_DEM_CF_FUNC_TIME_STS
312     STS_set(&mix_sts,CLK_gettime());
313     #endif
314     quad_mix(in_buffer_ptr,re_buffer,im_buffer);
315     #if PM_DEM_CF_FUNC_TIME_STS
316     STS_delta(&mix_sts,CLK_gettime());
317     #endif
318
319     #if PM_DEM_CF_FUNC_TIME_STS
320     STS_set(&dem_sts,CLK_gettime());
321     #endif
322     #if (PM_DEMODULATOR==PM_PLI_DEM)
323     pli_demodulate(re_buffer,im_buffer,out_buffer);
324     #elif (PM_DEMODULATOR==PM_MIXED_DEM)
325     mixed_demodulate(re_buffer,im_buffer,out_buffer);
326     #else
327     /*error No demodulator algorithm defined
328     #endif
329
330     #if PM_DEM_CF_FUNC_TIME_STS
331     STS_delta(&dem_sts,CLK_gettime());
332     #endif
333
334     #if PM_DEM_CF_FUNC_TIME_STS
335     STS_set(&filter_sts,CLK_gettime());
336     #endif
337     out_filter(out_buffer,out_buffer_ptr);
338     #if PM_DEM_CF_FUNC_TIME_STS
339     STS_delta(&filter_sts,CLK_gettime());
340     #endif
341
342     #if PM_DEM_CF_INT_TIME_STS
343     STS_delta(&demodulate_sts,CLK_gettime());
344     #endif
345
346     #if PM_DEM_CF_INT_TIME_LED
347     LED_off(LED_3);
348     #endif
349 }

```

## quad\_mix

Listing F.21: Fixed-Point :: quad\_mix.h

```

1 /*
2  QUADRATURE MIXER - VERSION FIXED POINT 0.5
3
4  file : quad_mix.h
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8 */
9
10
11 #ifndef _QUAD_MIX_
12 #define _QUAD_MIX_
13
14 /*
15  includes
16  -----
17  #include "global_settings.h"
18
19  -----
20  global functions
21  -----
22  */
23  the signal in the in_buffer is quadrature mixed to the two
24  buffers re_buffer and im_buffer and is also downsampled by
25  DOWNSAMPLE_ONE
26
27  Parameters :
28  in_buffer buffer with input signal
29  re_buffer buffer of signal mixed with cosine
30  out_buffer buffer of signal mixed with sine
31 */
32 void quad_mix(short in_buffer[], short re_buffer[], short im_buffer[]);
33
34 #endif /* _QUAD_MIX_ */
35
36 /*
37  END OF THE FILE
38 */

```

Listing F.22: Fixed-Point :: quad\_mix.c

```

1 /*
2  QUADRATURE MIXER - VERSION FIXED POINT 0.5
3
4  file : quad_mix.c
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8 */
9
10
11 /*
12  includes
13  -----
14  #include "quad_mix.h"
15
16  -----
17  global variables
18  -----
19
20  -----
21  coefficients of sine and cosine oscillator
22
23  -----
24  Frequency : 13139.93 Hz*/
25
26 short gen_bl_cos = -9076; /*cos*/
27 short gen_bl_sin = 31486; /*sin*/
28 short gen_al = -18152; /*sin/cos*/
29
30 /* delay variables */
31 short gen_z [3] = {0.0, 26213}; /* first value (delta function) */
32

```

```

33 /* 0.8 to avoid overflows */
34
35 -----
36 coefficients of mixer low-pass (mix_lp)
37 -----
38 /* Order : 5 */
39 /* Pass-band : 2 dB @ 6300.00 Hz */
40 /* Stop-band : 60 dB @ 19000.00 Hz */
41
42 #define MIX_LP_CASCADES 3
43 short const mix_lp_a[][3]={ /* a_i_la a_i_lb a_i_2 */
44     { -8092 , -8092 , 0 },
45     { -17444 , -17444 , 11166 },
46     { -21907 , -21907 , 22406 },
47 };
48
49 short const mix_lp_b[][3]={ /* b_i_lo b_i_l b_i_2 */
50     { 4523 , 4523 , 0 },
51     { 2725 , 5450 , 2725 },
52     { 8537 , 17074 , 8537 },
53 };
54
55 short const mix_lp_scale=16583; /*16583 to avoid overflows */
56
57 /* delay variables */
58 short mix_lp_z_re[][3]={
59     { 0.0,0 },
60     { 0.0,0 },
61     { 0.0,0 },
62 };
63 short mix_lp_z_im[][3]={
64     { 0.0,0 },
65     { 0.0,0 },
66     { 0.0,0 },
67 };
68
69 -----
70 implementation of function
71 -----
72 void quad_mix(short in_buffer[], short re_buffer[], short im_buffer[])
73 {
74     int i,k;
75
76     /*__ sine and cosine generation __*/
77
78     for(i=0;i<BUFFERSIZE_INPUT;i++)
79     {
80         /* feedback */
81         gen_z[0]=-((gen_al*gen_z[1])>>15) + gen_z[2] );
82
83         /* forward */
84         re_buffer[i]=gen_z[0] +(gen_bl_cos*gen_z[1])>>15;
85         im_buffer[i]=(gen_bl_sin*gen_z[1])>>15;
86
87         /* shift delays */
88         gen_z[2]=gen_z[1];
89         gen_z[1]=gen_z[0];
90     }
91
92     /* __ mixing and scaling __ */
93     for(i=0;i<BUFFERSIZE_INPUT;i++)
94     {
95         /* mixing */
96         re_buffer[i]=(re_buffer[i]*in_buffer[i])>>15;
97         im_buffer[i]=(im_buffer[i]*in_buffer[i])>>15;
98
99         /* scaling and neg of im path */
100         re_buffer[i]=(re_buffer[i]*mix_lp_scale)>>15;
101         im_buffer[i]=-((im_buffer[i]*mix_lp_scale)>>15);
102     }
103
104     /* __ low-pass filtering __ */
105
106     /* -- Cascade 0 (first Order) -- */
107     for(i=0;i<BUFFERSIZE_INPUT;i++)
108     {
109         /* feedback */
110         mix_lp_z_re[0][0] = re_buffer[i]-( ( mix_lp_a[0][0]*mix_lp_z_re[0][1] + /*
111             a_la*zi*/

```

```

112         mix_lp_a[0][1]*mix_lp_z_re[0][1] /*a_lb*21*/
113     )>>15;
114
115     mix_lp_z_im[0][0] = im_buffer[i] - ( mix_lp_a[0][0]*mix_lp_z_im[0][1] + /*
116         a_la*21*/
117         mix_lp_a[0][1]*mix_lp_z_im[0][1] /*a_lb*21*/
118     )>>15;
119
120     /* forward */
121     re_buffer[i] = ( mix_lp_b[0][0]*mix_lp_z_re[0][0] + /* b_0 * z0 */
122         mix_lp_b[0][1]*mix_lp_z_re[0][1] /* b_1 * z1 */
123     )>>15;
124
125     im_buffer[i] = ( mix_lp_b[0][0]*mix_lp_z_im[0][0] + /* b_0 * z0 */
126         mix_lp_b[0][1]*mix_lp_z_im[0][1] /* b_1 * z1 */
127     )>>15;
128
129     /* shift delays */
130     mix_lp_z_re[0][1]=mix_lp_z_re[0][0];
131     mix_lp_z_im[0][1]=mix_lp_z_im[0][0];
132 }
133
134 /* -- Cascade 1 (Second Order)--*/
135 for(i=0;i<BUFFERSIZE_INPUT;i++)
136 {
137     /* feedback */
138     mix_lp_z_re[1][0] = re_buffer[i] - ( mix_lp_a[1][0]*mix_lp_z_re[1][1] + /*
139         a_la*21*/
140         mix_lp_a[1][1]*mix_lp_z_re[1][1] + /*a_lb*21*/
141         mix_lp_a[1][2]*mix_lp_z_re[1][2] /*a_2 *z2*/
142     )>>15;
143
144     mix_lp_z_im[1][0] = im_buffer[i] - ( mix_lp_a[1][0]*mix_lp_z_im[1][1] + /*
145         a_la*21*/
146         mix_lp_a[1][1]*mix_lp_z_im[1][1] + /*a_lb*21*/
147         mix_lp_a[1][2]*mix_lp_z_im[1][2] /*a_2 *z2*/
148     )>>15;
149
150     /* forward */
151     re_buffer[i] = ( mix_lp_b[1][0]*mix_lp_z_re[1][0] + /* b_0 * z0 */
152         mix_lp_b[1][1]*mix_lp_z_re[1][1] + /* b_1 * z1 */
153         mix_lp_b[1][2]*mix_lp_z_re[1][2] /* b_2 * z2 */
154     )>>15;
155
156     im_buffer[i] = ( mix_lp_b[1][0]*mix_lp_z_im[1][0] + /* b_0 * z0 */
157         mix_lp_b[1][1]*mix_lp_z_im[1][1] + /* b_1 * z1 */
158         mix_lp_b[1][2]*mix_lp_z_im[1][2] /* b_2 * z2 */
159     )>>15;
160
161     /* shift delays */
162     mix_lp_z_re[1][2]=mix_lp_z_re[1][1];
163     mix_lp_z_re[1][1]=mix_lp_z_re[1][0];
164     mix_lp_z_im[1][2]=mix_lp_z_im[1][1];
165     mix_lp_z_im[1][1]=mix_lp_z_im[1][0];
166 }
167
168 /* -- Cascade 2 (second Order)--*/ /*including downsampling*/
169 for(i=0;k=0;i<BUFFERSIZE_INPUT;i++)
170 {
171     /* feedback */
172     mix_lp_z_re[2][0] = re_buffer[i] - ( mix_lp_a[2][0]*mix_lp_z_re[2][1] + /*
173         a_la*21*/
174         mix_lp_a[2][1]*mix_lp_z_re[2][1] + /*a_lb*21*/
175         mix_lp_a[2][2]*mix_lp_z_re[2][2] /*a_2 *z2*/
176     )>>15;
177
178     im_buffer[i] = im_buffer[i] - ( mix_lp_a[2][0]*mix_lp_z_im[2][1] + /*
179         a_la*21*/
180         mix_lp_a[2][1]*mix_lp_z_im[2][1] + /*a_lb*21*/
181         mix_lp_a[2][2]*mix_lp_z_im[2][2] /*a_2 *z2*/
182     )>>15;
183
184     /* forward including downsampling -> take every 4th value*/
185     if(i%4==0)
186     {
187         re_buffer[k] = ( mix_lp_b[2][0]*mix_lp_z_re[2][0] + /* b_0 * z0 */
188             mix_lp_b[2][1]*mix_lp_z_re[2][1] + /* b_1 * z1 */
189             mix_lp_b[2][2]*mix_lp_z_re[2][2] /* b_2 * z2 */

```

```

187     )>>15;
188
189     im_buffer[k] = ( mix_lp_b[2][0]*mix_lp_z_im[2][0] + /* b_0 * z0 */
190         mix_lp_b[2][1]*mix_lp_z_im[2][1] + /* b_1 * z1 */
191         mix_lp_b[2][2]*mix_lp_z_im[2][2] /* b_2 * z2 */
192     )>>15;
193
194     k++;
195 }
196
197 /* shift delays */
198 mix_lp_z_re[2][1]=mix_lp_z_re[2][0];
199 mix_lp_z_re[2][0]=mix_lp_z_re[2][0];
200
201 mix_lp_z_im[2][2]=mix_lp_z_im[2][1];
202 mix_lp_z_im[2][1]=mix_lp_z_im[2][0];
203 }
204
205
206
207 /*
208 END OF THE FILE
209 */

```

## mixed\_demodulate

Listing F.23: Fixed-Point :: mixed\_demodulate.h

```

1  /*
2  MIXED DEMODULATOR - VERSION FIXED POINT 0.5
3
4  file : mixed_demodulate.h
5  date : oct 2001 - jan 2002
6  by   : c. hailer
7       : f. schnyder
8  */
9
10 #ifndef MIXED_DEMODULATE_
11 #define MIXED_DEMODULATE_
12
13 /*
14 -----
15 includes
16 -----
17 #include "global_settings.h"
18 #include "atan.h"
19
20 -----
21 Global functions
22 -----
23 */
24
25 /*
26 FM demodulates the two baseband signals re_buffer and im_buffer
27 into the out_buffer with the mixed algorithm
28
29 Parameters :
30 re_buffer reel FM path in baseband
31 im_buffer imag FM path in baseband
32 out_buffer demodulated signal
33 */
34 void mixed_demodulate(short re_buffer[], short im_buffer[], short out_buffer[]);
35
36 #endif /* MIXED_DEMODULATE_ */
37
38 /*
39 END OF THE FILE
40 */

```

Listing F.24: Fixed-Point :: mixed\_demodulate.c

```

1  /*
2  MIXED DEMODULATOR - VERSION FIXED POINT 0.5
3
4  file : mixed_demodulate.c
5  date : oct 2001 - jan 2002

```

```

6 by : c.haller
7 f. schnyder
8
9 */
10
11 /*-----
12 includes
13 -----*/
14 #include "mixed_demodulate.h"
15
16 /*-----
17 global variables
18 -----*/
19 /* delay variables of the mixed demodulator */
20 short dem_z_re=16384; /* !=0 to avoid division with 0 */
21 short dem_z_im=16384; /* !=0 to avoid division with 0 */
22
23 /*-----
24 implementation of function
25 -----*/
26 void mixed_demodulate(short re_buffer[], short im_buffer[], short out_buffer[])
27 {
28     int i;
29     int re_path=0;
30     int im_path=0;
31     int dem_arg=0;
32
33     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
34     {
35         re_path = ( im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
36         im_path = ( re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */
37
38         /* shift delays */
39         dem_z_re=re_buffer[i];
40         dem_z_im=im_buffer[i];
41
42         /* calculate the argument */
43         dem_arg=re_path/(im_path>>14); /* Q17.14 */
44
45         /* atan lookup */
46         if(dem_arg>0) /*positive*/
47         {
48             dem_arg=dem_arg>>(ATAN_SHIFT);
49             if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
50             out_buffer[i]=atan_buffer(dem_arg);
51         }
52         else
53         {
54             dem_arg=-dem_arg;
55             dem_arg=-dem_arg>>(ATAN_SHIFT);
56             if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
57             out_buffer[i]=-atan_buffer(dem_arg);
58         }
59     }
60 }
61
62 /*-----
63 END OF THE FILE
64 -----*/

```

Listing F25: Fixed-Point Interpolated :: mixed\_demodulate.h

```

1 /*-----
2 MIXED DEMODULATOR WITH INTERPOLATION - VERSION FIXED POINT 0.5
3 -----*/
4 file : mixed_demodulate.h
5 date : oct 2001 - jan 2002
6 by : c.haller
7 f. schnyder
8
9 */
10
11 #ifndef _MIXED_DEMODULATE_
12 #define _MIXED_DEMODULATE_
13
14 /*-----
15 includes
16 -----*/
17 #include "global_settings.h"

```

Listing F26: Fixed-Point Interpolated :: mixed\_demodulate.c

```

1 /*-----
2 MIXED DEMODULATOR WITH INTERPOLATION - VERSION FIXED POINT 0.5
3 -----*/
4 file : mixed_demodulate.c
5 date : oct 2001 - jan 2002
6 by : c.haller
7 f. schnyder
8
9 */
10
11 /*-----
12 includes
13 -----*/
14 #include "mixed_demodulate.h"
15
16 /*-----
17 Global Variables
18 -----*/
19 /* delay variables of the mixed demodulator */
20 short dem_z_re=16384; /* !=0 to avoid division with 0 */
21 short dem_z_im=16384; /* !=0 to avoid division with 0 */
22
23 /*-----
24 implementation of function
25 -----*/
26 void mixed_demodulate(short re_buffer[], short im_buffer[], short out_buffer[])
27 {
28     int i;
29     int re_path=0;
30     int im_path=0;
31     int dem_arg=0;
32     short cutoff=0;
33
34     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
35     {
36         re_path = ( im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
37         im_path = ( re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */
38
39         /* shift delays */
40         dem_z_re=re_buffer[i];
41         dem_z_im=im_buffer[i];
42
43         /* calculate the argument */
44         dem_arg=re_path/(im_path>>14); /* Q17.14 */
45
46         /* atan lookup */
47         if(dem_arg>0) /*positive*/
48         {
49             cutoff=dem_arg>>(ATAN_SHIFT);
50             dem_arg=dem_arg>>(ATAN_SHIFT);
51             if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
52             out_buffer[i]=atan_buffer(dem_arg)+((cutoff*atan_grad_buffer(dem_arg))>>
53                 ATAN_SHIFT);
54         }
55     }
56 }

```

```

55     }else                /*negative*/
56     {
57
58         dem_arg=-dem_arg;
59         cutoff=dem_arg%ATAN_INTERPOL;
60         if(dem_arg>=BUFFERSIZE_ATAN)
61             out_buffer[i]=-(atan_buffer(dem_arg)+((cutoff*atan_grad_buffer(dem_arg))
62                                     >>ATAN_SHIFT));
63     }
64 }
65 }
66
67 /*
68  END OF THE FILE
69 */

```

## pll\_demodulate

Listing E27: Fixed-Point :: pll\_demodulate.h

```

1  /*
2  PLL DEMODULATOR - VERSION FIXED POINT 0.5
3
4  file : pll_demodulate.h
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8  */
9
10 #ifndef _PLL_DEMODULATE_
11 #define _PLL_DEMODULATE_
12
13 /*
14  includes
15
16  #include "global_settings.h"
17  #include "sine.h"
18
19  */
20
21 global functions
22
23 /*
24  FM demodulates the two baseband signals re_buffer and im_buffer
25  into the out_buffer with the pll algorithm
26
27  Parameters :
28  re_buffer reel FM path in baseband
29  im_buffer imag FM path in baseband
30  out_buffer demodulated signal
31  */
32 void pll_demodulate(short re_buffer[], short im_buffer[], short out_buffer[]);
33
34 #endif /* _PLL_DEMODULATE_ */
35
36 /*
37  END OF THE FILE
38 */

```

Listing E28: Fixed-Point :: pll\_demodulate.c

```

1  /*
2  PLL DEMODULATOR - VERSION FIXED POINT 0.5
3
4  file : pll_demodulate.c
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8  */
9
10 /*
11

```

```

12 includes
13
14 #include "pll_demodulate.h"
15
16 /*
17  global variables
18
19  short arg=0; /*argument of sine and cosine*/
20  short i_path=1000; /*I path*/
21  short q_path=0; /*Q path*/
22
23 /*
24  global defines
25
26  #define P_PLL 65536 /*P_PLL=2.0, using q15*/
27  #define K_PLL 10240 /*K_PLL=2.5, using q3.12*/
28  #define GAIN_PIL 55721 /*GAIN_PIL=1.7, using q15*/
29  #define TWO_PI 25735 /*2*PI, using q3.12*/
30
31 /*
32  implementation of function
33
34  void pll_demodulate(short re_buffer[], short im_buffer[], short out_buffer[])
35  {
36      int i;
37      int flag=0;
38
39      short demo; /*index for sine_buffer access*/
40      short index; /*value of demodulation*/
41
42      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
43      { /*_calculation of demodulated value, using Q15_*/
44          i_path=(re_buffer[i]*i_path)>>15;
45          q_path=(im_buffer[i]*q_path)>>15;
46          demo=q_path-i_path;
47          demo=(demo*P_PLL)>>15;
48          out_buffer[i]=(demo*GAIN_PIL)>>15;
49
50          /*_summation of demodulated signal, using Q3.12_*/
51          demo=demo>>3;
52          arg=arg+(demo*K_PLL)>>12;
53
54          /*_limitation of argument to 2*PI, using Q3.12_*/
55          if (arg>TWO_PI)
56          {
57              arg=arg-TWO_PI;
58          }
59          else if (arg<-TWO_PI)
60          {
61              arg=arg+TWO_PI;
62          }
63
64          /*_calculation of sine and cosine value of argument_*/
65          /*_calculating of index for sine_buffer access, using new Q format*/
66          index=arg>>REDUCTION;
67
68          if (index<0)
69          {
70              index=-index;
71              flag=1;
72          }
73
74          index=(index*STEP_INVERSE)>>(2*Q_FORMAT);
75
76          if (flag==1)
77          {
78
79
80
81
82
83
84
85
86
87
88
89
90
91

```



```

172 index=-index;
173 flag=0;
174 }
175 /*access on sine_buffer with index*/
176 if(index>0)
177 {
178     i_path=sine_buffer[index];
179     /*cosine value*/
180     index=(BUFFERSIZE_SINE-1)-index;
181     q_path=-sine_buffer[index];
182 }
183 else if (index<-4*(BUFFERSIZE_SINE-1))
184 {
185     index=-(index % (BUFFERSIZE_SINE-1));
186     /*sine value*/
187     index=(BUFFERSIZE_SINE-1)-index;
188     i_path=sine_buffer[index];
189     /*cosine value*/
190     index=(BUFFERSIZE_SINE-1)-index;
191     q_path=-sine_buffer[index];
192 }
193 else if (index<-3*(BUFFERSIZE_SINE-1))
194 {
195     index=-(index % (BUFFERSIZE_SINE-1));
196     /*sine value*/
197     index=(BUFFERSIZE_SINE-1)-index;
198     i_path=sine_buffer[index];
199     /*cosine value*/
200     index=(BUFFERSIZE_SINE-1)-index;
201     q_path=-sine_buffer[index];
202 }
203 END OF THE FILE
204 */

```

Listing F29: Fixed-Point Interpolated :: pll\_demodulate.h

```

1 /*
2  * PLL DEMODULATOR WITH INTERPOLATION - VERSION FIXED POINT 0.5
3  */
4 #include "pll_demodulate.h"
5 #include "oct_2001_jan_2002"
6 #include "c_halter"
7 #include "f_schneider"
8
9 /*
10  *
11  * #ifndef PLL_DEMODULATE_
12  * #define PLL_DEMODULATE_
13  *
14  * -----
15  * includes
16  * -----
17  * #include "global_settings.h"
18  * #include "sine.h"
19  *
20  * -----
21  * global functions
22  * -----
23  */
24 /*
25  * PLL demodulates the two baseband signals re_buffer and im_buffer
26  * into the out_buffer with the pll algorithm
27  *
28  * Parameters :
29  * re_buffer reel FM path in baseband
30  * im_buffer imag FM path in baseband
31  * out_buffer demodulated signal
32  */
33 void pll_demodulate(short re_buffer[], short im_buffer[], short out_buffer[]);
34 #endif /* _PLL_DEMODULATE_ */
35
36 /*
37  * END OF THE FILE
38  */

```

Listing F30: Fixed-Point Interpolated :: pll\_demodulate.c

```

1 /*
2  * PLL DEMODULATOR WITH INTERPOLATION - VERSION FIXED POINT 0.5
3  */

```

```

92 index=-index;
93 flag=0;
94 }
95 /*access on sine_buffer with index*/
96 if(index>0)
97 {
98     i_path=sine_buffer[index];
99     /*cosine value*/
100     index=(BUFFERSIZE_SINE-1)-index;
101     q_path=-sine_buffer[index];
102 }
103 else if (index<-4*(BUFFERSIZE_SINE-1))
104 {
105     index=-(index % (BUFFERSIZE_SINE-1));
106     /*sine value*/
107     index=(BUFFERSIZE_SINE-1)-index;
108     i_path=sine_buffer[index];
109     /*cosine value*/
110     index=(BUFFERSIZE_SINE-1)-index;
111     q_path=-sine_buffer[index];
112 }
113 else if (index<-3*(BUFFERSIZE_SINE-1))
114 {
115     index=-(index % (BUFFERSIZE_SINE-1));
116     /*sine value*/
117     index=(BUFFERSIZE_SINE-1)-index;
118     i_path=sine_buffer[index];
119     /*cosine value*/
120     index=(BUFFERSIZE_SINE-1)-index;
121     q_path=-sine_buffer[index];
122 }
123 else if (index<-2*(BUFFERSIZE_SINE-1))
124 {
125     index=-(index % (BUFFERSIZE_SINE-1));
126     /*sine value*/
127     index=(BUFFERSIZE_SINE-1)-index;
128     i_path=sine_buffer[index];
129     /*cosine value*/
130     index=(BUFFERSIZE_SINE-1)-index;
131     q_path=-sine_buffer[index];
132 }
133 else if (index<-1*(BUFFERSIZE_SINE-1))
134 {
135     index=-(index % (BUFFERSIZE_SINE-1));
136     /*sine value*/
137     index=(BUFFERSIZE_SINE-1)-index;
138     i_path=sine_buffer[index];
139     /*cosine value*/
140     index=(BUFFERSIZE_SINE-1)-index;
141     q_path=-sine_buffer[index];
142 }
143 else if (index<0)
144 {
145     index=-(index % (BUFFERSIZE_SINE-1));
146     /*sine value*/
147     index=(BUFFERSIZE_SINE-1)-index;
148     i_path=sine_buffer[index];
149     /*cosine value*/
150     index=(BUFFERSIZE_SINE-1)-index;
151     q_path=-sine_buffer[index];
152 }
153 else if (index<-1*(BUFFERSIZE_SINE-1))
154 {
155     index=-(index % (BUFFERSIZE_SINE-1));
156     /*sine value*/
157     index=(BUFFERSIZE_SINE-1)-index;
158     i_path=sine_buffer[index];
159     /*cosine value*/
160     index=(BUFFERSIZE_SINE-1)-index;
161     q_path=-sine_buffer[index];
162 }
163 else if (index<-2*(BUFFERSIZE_SINE-1))
164 {
165     index=-(index % (BUFFERSIZE_SINE-1));
166     /*sine value*/
167     index=(BUFFERSIZE_SINE-1)-index;
168     i_path=sine_buffer[index];
169     /*cosine value*/
170     index=(BUFFERSIZE_SINE-1)-index;
171     q_path=-sine_buffer[index];
172 }
173 else if (index<-3*(BUFFERSIZE_SINE-1))
174 {
175     index=-(index % (BUFFERSIZE_SINE-1));
176     /*sine value*/
177     index=(BUFFERSIZE_SINE-1)-index;
178     i_path=sine_buffer[index];
179     /*cosine value*/
180     index=(BUFFERSIZE_SINE-1)-index;
181     q_path=-sine_buffer[index];
182 }
183 else if (index<-4*(BUFFERSIZE_SINE-1))
184 {
185     index=-(index % (BUFFERSIZE_SINE-1));
186     /*sine value*/
187     index=(BUFFERSIZE_SINE-1)-index;
188     i_path=sine_buffer[index];
189     /*cosine value*/
190     index=(BUFFERSIZE_SINE-1)-index;
191     q_path=-sine_buffer[index];
192 }
193 }
194 END OF THE FILE
195 */

```

```

4  file : pll_demodulate.c
5  date : oct 24/11 - jan 2002
6  by : C. Haller
7       F. schnyder
8
9  -----
10
11  /*
12  -----
13  -----
14  #include "pll_demodulate.h"
15
16  /*
17  -----
18  -----
19  global variables
20
21  short arg0; /*argument of sine and cosine*/
22  short i_path=0; /*I path*/
23  short q_path=0; /*Q path*/
24
25  /*
26  -----
27  -----
28  global defines
29
30  #define P_PLL 65536 /*P_PLL=2.0, using q15*/
31  #define K_PLL 10240 /*K_PLL=2.5, using q12*/
32  #define GAIN_PLL 55721 /*GAIN_PLL=1.7, using q15*/
33  #define TWO_PI 25735 /*2*PI, using q12*/
34
35  /*
36  -----
37  -----
38  Implementation of function
39
40  void pll_demodulate(short re_buffer[], short im_buffer[], short out_buffer[])
41  {
42      int i;
43      int flag=0; /*flag to check sign of index*/
44
45      short index; /*index for sine_buffer access*/
46      short correction; /*correction used for interpolation*/
47      short rest; /*rest used for interpolation*/
48      short demo; /*value of demodulation*/
49
50      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
51      {
52          /*__calculation of demodulated value using Q15__*/
53
54          i_path=re_buffer[i]*i_path>>15;
55          q_path=im_buffer[i]*q_path>>15;
56
57          demo=q_path-i_path;
58          demo=(demo>>P_PLL)>>15;
59
60          out_buffer[i]=(demo*GAIN_PLL)>>15;
61
62          /*__summation of demodulated signal, using Q3.12__*/
63
64          demo=demo>>3;
65          arg=arg+((demo*K_PLL)>>12);
66
67          /*__limitation of argument to 2*PI, using Q3.12__*/
68
69          if (arg>TWO_PI)
70          {
71              arg=arg-TWO_PI;
72          }
73
74          else if (arg<-TWO_PI)
75          {
76              arg=arg+TWO_PI;
77          }
78
79          /*__calculation of sine and cosine value of argument__*/
80
81          /*calculating of index for sine_buffer access, using new Q format*/
82
83          index=arg>>REDUCTION;

```

```

164 if(index>-(BUFFERSIZE_SINE-1))
165 {
166     index=index;
167     /*sine value*/
168     corrections=(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*positive*/
169     i_path=-(sine_buffer(index)+correction);
170     /*cosine value*/
171     index=(BUFFERSIZE_SINE-1)-index;
172     corrections=-(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*negative*/
173     q_path=sine_buffer(index)+correction;
174 }
175 else if(index>-3*(BUFFERSIZE_SINE-1))
176 {
177     index=-(index % (BUFFERSIZE_SINE-1));
178     /*sine value*/
179     index=(BUFFERSIZE_SINE-1)-index;
180     corrections=(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*negative*/
181     i_path=-(sine_buffer(index)+correction);
182     /*cosine value*/
183     index=(BUFFERSIZE_SINE-1)-index;
184     corrections=(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*positive*/
185     q_path=-(sine_buffer(index)+correction);
186 }
187 else if(index>-3*(BUFFERSIZE_SINE-1))
188 {
189     index=-(index % (BUFFERSIZE_SINE-1));
190     /*sine value*/
191     corrections=(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*positive*/
192     i_path=sine_buffer(index)+correction;
193     /*cosine value*/
194     index=(BUFFERSIZE_SINE-1)-index;
195     corrections=-(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*negative*/
196     q_path=-(sine_buffer(index)+correction);
197 }
198 else if(index>-4*(BUFFERSIZE_SINE-1))
199 {
200     index=-(index % (BUFFERSIZE_SINE-1));
201     /*sine value*/
202     index=(BUFFERSIZE_SINE-1)-index;
203     corrections=(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*negative*/
204     i_path=sine_buffer(index)+correction;
205     /*cosine value*/
206     index=(BUFFERSIZE_SINE-1)-index;
207     corrections=(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*positive*/
208     q_path=sine_buffer(index)+correction;
209 }
210 else
211 {
212     index=-(index % (BUFFERSIZE_SINE-1));
213     /*sine value*/
214     corrections=(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*positive*/
215     i_path=-(sine_buffer(index)+correction);
216     /*cosine value*/
217     index=(BUFFERSIZE_SINE-1)-index;
218     corrections=-(sine_grad_buffer(index)*rest)>>Q_FORMAT;/*negative*/
219     q_path=sine_buffer(index)+correction;
220 }
221 }
222 }
223 }
224 }
225 /*
226 END OF THE FILE
227 */

```

out\_filter

Listing E32: Fixed-Point :: out\_filter.c

```

1 /*
2 OUTPUT AND NOISE FILTER - VERSION FIXED POINT 0.5
3
4 file : out_filter.c
5 date : oct 2001 - jan 2002
6 by : c. haller
7 f. schnyder
8 */
9 */
10
11 /*-----
12 includes
13 -----*/
14 #include "out_filter.h"
15
16 /*-----
17 Global variables
18 -----*/
19
20 /*-----
21 coefficients of noise low-pass (no_lp) after demodulation
22 -----*/
23 /* Order : 13
24 /* Pass-band : 2 dB @ 3500.00 Hz */
25 /* Stop-band : 20 dB @ 3999.57 Hz */
26
27 #define NO_LP_CASCADES 7
28 short no_lp_a[12]={ /* a_i-1 a_i-2 */
29     { -2888 , 0 },
30     { -5861 , 738 },
31     { -6124 , 2244 },
32     { -6600 , 4962 },
33     { -7352 , 9261 },
34     { -8497 , 15808 },
35     { -10247 , 25815 },
36     };
37
38 short no_lp_b[13]={ /* b_i-0 b_i-1 b_i-2 */
39     { 13822 , 13822 , 0 },
40     { 7222 , 14444 , 7222 },
41     { 7783 , 15565 , 7783 },
42     { 8669 , 17339 , 8669 },
43     { 10020 , 20040 , 10020 },
44     { 7947 , 15893 , 7947 },
45     };

```

Listing E31: Fixed-Point :: out\_filter.h

```

1 /*
2 OUTPUT AND NOISE FILTER - VERSION FIXED POINT 0.5
3
4 file : out_filter.h
5 date : oct 2001 - jan 2002
6 by : c. haller

```

```

125 /* shift delays */
126 no_lp_z[j][2]=no_lp_z[j][1];
127 no_lp_z[j][1]=no_lp_z[j][0];
128 }
129 }
130 }
131 /* last cascade including downsampling */
132 for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
133 {
134 /* feedback */
135 no_lp_z[j][0] = out_buffer[i]-( ( no_lp_a[j][0]*no_lp_z[j][1] + /*a_1*z1*/
136 /*a_2*z2*/
137 )>>15);
138 }
139 }
140 /* forward including downsampling -> take every 2nd value*/
141 if(!&DOWNSAMPLE_TWO==0)
142 {
143 out_buffer_down[k]= ( no_lp_b[j][0]*no_lp_z[j][0]+ /* b_0 * z0 */
144 /* b_1 * z1 */
145 no_lp_b[j][1]*no_lp_z[j][1]+ /* b_1 * z1 */
146 no_lp_b[j][2]*no_lp_z[j][2] /* b_2 * z2 */
147 )>>15;
148 k++;
149 }
150 /* shift delays */
151 no_lp_z[j][2]=no_lp_z[j][1];
152 no_lp_z[j][1]=no_lp_z[j][0];
153 }
154 }
155 /*__ the high-pass __*/
156 }
157 }
158 /* scaling */
159 for(i=0;i<BUFFERSIZE_OUTPUT;i++)
160 {
161 out_buffer_down[i]=(out_buffer_down[i]*en_bp_scale)>>15;
162 }
163 }
164 /* high-pass in transposed form */
165 for(j=0;j<EN_HP_CASCADES;j++)
166 {
167 for(i=0;i<BUFFERSIZE_OUTPUT;i++)
168 {
169 en_bp_z[j][2]= en_bp_z[j][1]+out_buffer_down[i];
170 en_bp_z[j][1]= -out_buffer_down[i]-out_buffer_down[i]-
171 ((en_bp_a[j][0]*en_bp_z[j][2]+
172 en_bp_a[j][1]*en_bp_z[j][2])>>15)+
173 en_bp_z[j][0];
174 }
175 }
176 en_bp_z[j][0]= out_buffer_down[i]-
177 ((en_bp_a[j][2]*en_bp_z[j][2])>>15);
178 out_buffer_down[i]=en_bp_z[j][2];
179 }
180 }
181 for(i=0;i<BUFFERSIZE_OUTPUT;i++)
182 {
183 out_buffer_down[i]=out_buffer_down[i]*EN_HP_UPSCALE;
184 }
185 }
186 }
187 }
188 }
189 /*
190 END OF THE FILE
*/

```

## F.1.4. Fixed-Point Optimized

global\_settings

Listing F33: Fixed-Point Optimized :: global\_settings.h

```

46 { 12457 , 24914 , 12457 },
47 };
48
49 short no_lp_scale=23904; /* 29880*0.8 0.911862 to avoid overflows */
50
51 /* delay variables */
52 short no_lp_z[j][3]=(
53 { 0,0,0},
54 { 0,0,0},
55 { 0,0,0},
56 { 0,0,0},
57 { 0,0,0},
58 { 0,0,0},
59 { 0,0,0},
60 );
61
62 /*----- coefficients of end high-pass (en_hp) before dac -----*/
63
64 /* Order : 4 */
65 /* Pass-band : 0.1 dB @ 300.00 Hz */
66 /* Stop-band : 40.0 dB @ 50.00 Hz */
67
68 #define EN_HP_CASCADES 2
69 short en_bp_a[j][3]=( /* aa_i1a a_i1b a_i2 */
70 { -29180 , -29180 , 26045 },
71 { -31046 , -31046 , 29805 },
72 );
73
74 /*
75 short en_bp_b[j][3]=( b_i0 b_i1 b_i2
76 { 1 , -2 , 1 },
77 { 1 , -2 , 1 },
78 );
79 */
80
81 short en_bp_scale=13931; /* 0.5*0.850269 to avoid overflows */
82
83 #define EN_HP_UPSCALE 2
84
85 /* delay variables */
86 short en_bp_z[j][3]=(
87 { 0,0,0},
88 { 0,0,0},
89 );
90
91 /*----- implementation of function -----*/
92
93 void out_filter(short out_buffer[], short out_buffer_down[])
94 {
95 int i,j,k;
96 k=0;
97
98 /*__ the low-pass __*/
99
100 /* scaling */
101 for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
102 {
103 out_buffer[i]=(out_buffer[i]*no_lp_scale)>>15;
104 }
105
106 /* all cascades except the last one */
107 for(j=0;j<NO_LP_CASCADES-1;j++)
108 {
109 for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
110 {
111 /* feedback */
112 no_lp_z[j][0] = out_buffer[i]-( ( no_lp_a[j][0]*no_lp_z[j][1] + /*a_1*z1*/
113 /*a_2*z2*/
114 )>>15);
115 }
116 }
117
118 /* forward */
119 out_buffer[i]= ( no_lp_b[j][0]*no_lp_z[j][0]+ /* b_0 * z0 */
120 /* b_1 * z1 */
121 no_lp_b[j][1]*no_lp_z[j][1]+ /* b_1 * z1 */
122 no_lp_b[j][2]*no_lp_z[j][2] /* b_2 * z2 */
123 )>>15;
124 }

```

```

2 GLOBAL SETTINGS - VERSION FIXED POINT 1
3
4 file : global_settings.h
5 date : oct 2001 - Jan 2002
6 by : c. haller
7      f. schnyder
8 */
9
10 #define BUFFERSIZE 128
11 #define DOWNSAMPLE_ONE 4
12 #define DOWNSAMPLE_TWO 2
13
14 #define BUFFERSIZE_INPUT (BUFFERSIZE)
15 #define BUFFERSIZE_DEMODULATOR (BUFFERSIZE/DOWNSAMPLE_ONE)
16 #define BUFFERSIZE_OUTPUT (BUFFERSIZE/(DOWNSAMPLE_ONE*DOWNSAMPLE_TWO))
17
18 #define PI 3.14159265359
19
20 #define WORD_ALIGNED(x) (_nassert(((int)(x) & 0x3) == 0))
21
31 fm_dem_main
32
33 Listing F.34: Fixed-Point Optimized :: fm_dem_main.c
34
35 1 /*
36 2 FM DEMODULATION MAIN - VERSION FIXED POINT 1
37 3
38 4 file : fm_dem_main.c
39 5 date : nov 2001
40 6 by : c. haller
41 7      f. schnyder
42 8 */
43 9
44 10 /*----- demodulator define -----*/
45 11 #define FM_MIXED_DEM 10
46 12 #define FM_PLL_DEM 20
47 13 #define FM_DEMODULATOR FM_MIXED_DEM
48 14
49 15 /*----- includes -----*/
50 16 #include <std.h>
51 17 #include <swi.h>
52 18 #include <csl.h>
53 19 #include <bsl.h>
54 20
55 21 #include "global_settings.h"
56 22 #include "adc_THS1408.h"
57 23 #include "dac_codec.h"
58 24 #include "quad_mix.h"
59 25 #include "out_filter.h"
60 26 #if (FM_DEMODULATOR==FM_PLL_DEM)
61 27 #include "pll_demodulate.h"
62 28 #elif (FM_DEMODULATOR==FM_MIXED_DEM)
63 29 #include "mixed_demodulate.h"
64 30 #else
65 31 #error No demodulator algorithm defined
66 32 #endif
67 33
68 34 #pragma DATA MEM_BANK (in_buffer_A, 0);
69 35 #pragma DATA MEM_BANK (in_buffer_B, 0);
70 36 #pragma DATA MEM_BANK (re_buffer, 2);
71 37 #pragma DATA MEM_BANK (im_buffer, 4);
72 38 #pragma DATA MEM_BANK (out_buffer, 0);
73 39 #pragma DATA MEM_BANK (out_buffer_A, 6);
74 40 #pragma DATA MEM_BANK (out_buffer_B, 6);
75 41
76 42 #define FM_DEM_CF_FUNC_TIME_STS 0
77 43 /* Enable STS Objects: */
78 44 /* -mix_sts */
79 45 /* -dem_sts */
80 46 /* -filter_sts */
81 47 /* to measure the duration time of the interrupts */
82 48
83 49 #define FM_DEM_CF_INT_TIME_STS 0
84 50 /* Enable STS Objects: */
85 51 /* -mix_sts */
86 52 /* -dem_sts */
87 53 /* -filter_sts */
88 54 /* to measure the duration time of the interrupts */
89 55
90 56 #define FM_DEM_CF_INT_TIME_LED 0
91 57 /* Led ON by interrupt start OFF at the end */
92 58
93 59 #define FM_DEM_CF_INT_TIME_STS 0
94 60 /* Enable STS Objects: */
95 61 /* -mix_sts */
96 62 /* -dem_sts */
97 63 /* -filter_sts */
98 64 /* to measure the duration time of the interrupts */
99 65
100 66 #define FM_DEM_CF_FUNC_TIME_STS 0
101 67 /* Enable STS Objects: */
102 68 /* -mix_sts */
103 69 /* -dem_sts */
104 70 /* -filter_sts */
105 71 /* to measure the duration time of the interrupts */
106 72
107 73 #define FM_DEM_CF_DEBUG
108 74 #include <log.h>
109 75 #extern far LOG_Obj debug_log;
110 76 #endif
111
112 77 #if FM_DEM_CF_INT_OCCUR_LED
113 78 #include <bsl_led.h>
114 79 #endif
115
116 80 #if FM_DEM_CF_INT_TIME_LED
117 81 #include <bsl_led.h>
118 82 #endif
119
120 83 #if FM_DEM_CF_INT_TIME_STS
121 84 #include <clk.h>
122 85 #include <sts.h>
123 86 #extern far STS_Obj dac_sts;
124 87 #extern far STS_Obj demodulate_sts;
125 88 #extern far STS_Obj edma_sts;
126 89 #endif
127
128 89 #if FM_DEM_CF_FUNC_TIME_STS
129 90 #include <clk.h>
130 91 #include <sts.h>
131 92 #extern far STS_Obj mix_sts;
132 93 #extern far STS_Obj dem_sts;
133 94 #extern far STS_Obj filter_sts;
134 95 #endif
135
136 96 /*----- DSP/BIOS extern variables -----*/
137 97
138 98 #extern far SWI_Obj demodulate_swi;
139 99 #extern HWI_enable();
140 100 #extern HWI_disable();
141 101
142 102 /*----- Global variables -----*/
143 103
144 104 #pragma DATA MEM_BANK (in_buffer_A, 0);
145 105 #pragma DATA MEM_BANK (in_buffer_B, 0);
146 106 #pragma DATA MEM_BANK (re_buffer, 2);
147 107 #pragma DATA MEM_BANK (im_buffer, 4);
148 108 #pragma DATA MEM_BANK (out_buffer, 0);
149 109 #pragma DATA MEM_BANK (out_buffer_A, 6);
150 110 #pragma DATA MEM_BANK (out_buffer_B, 6);
151 111
152 112 #define FM_DEM_CF_FUNC_TIME_STS 0
153 113 /* Enable STS Objects: */
154 114 /* -mix_sts */
155 115 /* -dem_sts */
156 116 /* -filter_sts */
157 117 /* to measure the duration time of the interrupts */
158 118
159 119 #define FM_DEM_CF_INT_TIME_STS 0
160 120 /* Enable STS Objects: */
161 121 /* -mix_sts */
162 122 /* -dem_sts */
163 123 /* -filter_sts */
164 124 /* to measure the duration time of the interrupts */
165 125
166 126 #define FM_DEM_CF_INT_TIME_LED 0
167 127 /* Led ON by interrupt start OFF at the end */
168 128
169 129 #define FM_DEM_CF_INT_TIME_STS 0
170 130 /* Enable STS Objects: */
171 131 /* -mix_sts */
172 132 /* -dem_sts */
173 133 /* -filter_sts */
174 134 /* to measure the duration time of the interrupts */
175 135
176 136 #define FM_DEM_CF_FUNC_TIME_STS 0
177 137 /* Enable STS Objects: */
178 138 /* -mix_sts */
179 139 /* -dem_sts */
180 140 /* -filter_sts */
181 141 /* to measure the duration time of the interrupts */
182 142
183 143 #define FM_DEM_CF_INT_TIME_STS 0
184 144 /* Enable STS Objects: */
185 145 /* -mix_sts */
186 146 /* -dem_sts */
187 147 /* -filter_sts */
188 148 /* to measure the duration time of the interrupts */
189 149
190 150 #define FM_DEM_CF_INT_TIME_LED 0
191 151 /* Led ON by interrupt start OFF at the end */
192 152
193 153 #define FM_DEM_CF_INT_TIME_STS 0
194 154 /* Enable STS Objects: */
195 155 /* -mix_sts */
196 156 /* -dem_sts */
197 157 /* -filter_sts */
198 158 /* to measure the duration time of the interrupts */
199 159
200 160 #define FM_DEM_CF_FUNC_TIME_STS 0
201 161 /* Enable STS Objects: */
202 162 /* -mix_sts */
203 163 /* -dem_sts */
204 164 /* -filter_sts */
205 165 /* to measure the duration time of the interrupts */
206 166
207 167 #define FM_DEM_CF_INT_TIME_STS 0
208 168 /* Enable STS Objects: */
209 169 /* -mix_sts */
210 170 /* -dem_sts */
211 171 /* -filter_sts */
212 172 /* to measure the duration time of the interrupts */
213 173
214 174 #define FM_DEM_CF_INT_TIME_LED 0
215 175 /* Led ON by interrupt start OFF at the end */
216 176
217 177 #define FM_DEM_CF_INT_TIME_STS 0
218 178 /* Enable STS Objects: */
219 179 /* -mix_sts */
220 180 /* -dem_sts */
221 181 /* -filter_sts */
222 182 /* to measure the duration time of the interrupts */
223 183
224 184 #define FM_DEM_CF_FUNC_TIME_STS 0
225 185 /* Enable STS Objects: */
226 186 /* -mix_sts */
227 187 /* -dem_sts */
228 188 /* -filter_sts */
229 189 /* to measure the duration time of the interrupts */
230 190
231 191 #define FM_DEM_CF_INT_TIME_STS 0
232 192 /* Enable STS Objects: */
233 193 /* -mix_sts */
234 194 /* -dem_sts */
235 195 /* -filter_sts */
236 196 /* to measure the duration time of the interrupts */
237 197
238 198 #define FM_DEM_CF_INT_TIME_LED 0
239 199 /* Led ON by interrupt start OFF at the end */
240 200
241 201 #define FM_DEM_CF_INT_TIME_STS 0
242 202 /* Enable STS Objects: */
243 203 /* -mix_sts */
244 204 /* -dem_sts */
245 205 /* -filter_sts */
246 206 /* to measure the duration time of the interrupts */
247 207
248 208 #define FM_DEM_CF_FUNC_TIME_STS 0
249 209 /* Enable STS Objects: */
250 210 /* -mix_sts */
251 211 /* -dem_sts */
252 212 /* -filter_sts */
253 213 /* to measure the duration time of the interrupts */
254 214
255 215 #define FM_DEM_CF_INT_TIME_STS 0
256 216 /* Enable STS Objects: */
257 217 /* -mix_sts */
258 218 /* -dem_sts */
259 219 /* -filter_sts */
260 220 /* to measure the duration time of the interrupts */
261 221
262 222 #define FM_DEM_CF_INT_TIME_LED 0
263 223 /* Led ON by interrupt start OFF at the end */
264 224
265 225 #define FM_DEM_CF_INT_TIME_STS 0
266 226 /* Enable STS Objects: */
267 227 /* -mix_sts */
268 228 /* -dem_sts */
269 229 /* -filter_sts */
270 230 /* to measure the duration time of the interrupts */
271 231
272 232 #define FM_DEM_CF_FUNC_TIME_STS 0
273 233 /* Enable STS Objects: */
274 234 /* -mix_sts */
275 235 /* -dem_sts */
276 236 /* -filter_sts */
277 237 /* to measure the duration time of the interrupts */
278 238
279 239 #define FM_DEM_CF_INT_TIME_STS 0
280 240 /* Enable STS Objects: */
281 241 /* -mix_sts */
282 242 /* -dem_sts */
283 243 /* -filter_sts */
284 244 /* to measure the duration time of the interrupts */
285 245
286 246 #define FM_DEM_CF_INT_TIME_LED 0
287 247 /* Led ON by interrupt start OFF at the end */
288 248
289 249 #define FM_DEM_CF_INT_TIME_STS 0
290 250 /* Enable STS Objects: */
291 251 /* -mix_sts */
292 252 /* -dem_sts */
293 253 /* -filter_sts */
294 254 /* to measure the duration time of the interrupts */
295 255
296 256 #define FM_DEM_CF_FUNC_TIME_STS 0
297 257 /* Enable STS Objects: */
298 258 /* -mix_sts */
299 259 /* -dem_sts */
300 260 /* -filter_sts */
301 261 /* to measure the duration time of the interrupts */
302 262
303 263 #define FM_DEM_CF_INT_TIME_STS 0
304 264 /* Enable STS Objects: */
305 265 /* -mix_sts */
306 266 /* -dem_sts */
307 267 /* -filter_sts */
308 268 /* to measure the duration time of the interrupts */
309 269
310 270 #define FM_DEM_CF_INT_TIME_LED 0
311 271 /* Led ON by interrupt start OFF at the end */
312 272
313 273 #define FM_DEM_CF_INT_TIME_STS 0
314 274 /* Enable STS Objects: */
315 275 /* -mix_sts */
316 276 /* -dem_sts */
317 277 /* -filter_sts */
318 278 /* to measure the duration time of the interrupts */
319 279
320 280 #define FM_DEM_CF_FUNC_TIME_STS 0
321 281 /* Enable STS Objects: */
322 282 /* -mix_sts */
323 283 /* -dem_sts */
324 284 /* -filter_sts */
325 285 /* to measure the duration time of the interrupts */
326 286
327 287 #define FM_DEM_CF_INT_TIME_STS 0
328 288 /* Enable STS Objects: */
329 289 /* -mix_sts */
330 290 /* -dem_sts */
331 291 /* -filter_sts */
332 292 /* to measure the duration time of the interrupts */
333 293
334 294 #define FM_DEM_CF_INT_TIME_LED 0
335 295 /* Led ON by interrupt start OFF at the end */
336 296
337 297 #define FM_DEM_CF_INT_TIME_STS 0
338 298 /* Enable STS Objects: */
339 299 /* -mix_sts */
340 300 /* -dem_sts */
341 301 /* -filter_sts */
342 302 /* to measure the duration time of the interrupts */
343 303
344 304 #define FM_DEM_CF_FUNC_TIME_STS 0
345 305 /* Enable STS Objects: */
346 306 /* -mix_sts */
347 307 /* -dem_sts */
348 308 /* -filter_sts */
349 309 /* to measure the duration time of the interrupts */
350 310
351 311 #define FM_DEM_CF_INT_TIME_STS 0
352 312 /* Enable STS Objects: */
353 313 /* -mix_sts */
354 314 /* -dem_sts */
355 315 /* -filter_sts */
356 316 /* to measure the duration time of the interrupts */
357 317
358 318 #define FM_DEM_CF_INT_TIME_LED 0
359 319 /* Led ON by interrupt start OFF at the end */
360 320
361 321 #define FM_DEM_CF_INT_TIME_STS 0
362 322 /* Enable STS Objects: */
363 323 /* -mix_sts */
364 324 /* -dem_sts */
365 325 /* -filter_sts */
366 326 /* to measure the duration time of the interrupts */
367 327
368 328 #define FM_DEM_CF_FUNC_TIME_STS 0
369 329 /* Enable STS Objects: */
370 330 /* -mix_sts */
371 331 /* -dem_sts */
372 332 /* -filter_sts */
373 333 /* to measure the duration time of the interrupts */
374 334
375 335 #define FM_DEM_CF_INT_TIME_STS 0
376 336 /* Enable STS Objects: */
377 337 /* -mix_sts */
378 338 /* -dem_sts */
379 339 /* -filter_sts */
380 340 /* to measure the duration time of the interrupts */
381 341
382 342 #define FM_DEM_CF_INT_TIME_LED 0
383 343 /* Led ON by interrupt start OFF at the end */
384 344
385 345 #define FM_DEM_CF_INT_TIME_STS 0
386 346 /* Enable STS Objects: */
387 347 /* -mix_sts */
388 348 /* -dem_sts */
389 349 /* -filter_sts */
390 350 /* to measure the duration time of the interrupts */
391 351
392 352 #define FM_DEM_CF_FUNC_TIME_STS 0
393 353 /* Enable STS Objects: */
394 354 /* -mix_sts */
395 355 /* -dem_sts */
396 356 /* -filter_sts */
397 357 /* to measure the duration time of the interrupts */
398 358
399 359 #define FM_DEM_CF_INT_TIME_STS 0
400 360 /* Enable STS Objects: */
401 361 /* -mix_sts */
402 362 /* -dem_sts */
403 363 /* -filter_sts */
404 364 /* to measure the duration time of the interrupts */
405 365
406 366 #define FM_DEM_CF_INT_TIME_LED 0
407 367 /* Led ON by interrupt start OFF at the end */
408 368
409 369 #define FM_DEM_CF_INT_TIME_STS 0
410 370 /* Enable STS Objects: */
411 371 /* -mix_sts */
412 372 /* -dem_sts */
413 373 /* -filter_sts */
414 374 /* to measure the duration time of the interrupts */
415 375
416 376 #define FM_DEM_CF_FUNC_TIME_STS 0
417 377 /* Enable STS Objects: */
418 378 /* -mix_sts */
419 379 /* -dem_sts */
420 380 /* -filter_sts */
421 381 /* to measure the duration time of the interrupts */
422 382
423 383 #define FM_DEM_CF_INT_TIME_STS 0
424 384 /* Enable STS Objects: */
425 385 /* -mix_sts */
426 386 /* -dem_sts */
427 387 /* -filter_sts */
428 388 /* to measure the duration time of the interrupts */
429 389
430 390 #define FM_DEM_CF_INT_TIME_LED 0
431 391 /* Led ON by interrupt start OFF at the end */
432 392
433 393 #define FM_DEM_CF_INT_TIME_STS 0
434 394 /* Enable STS Objects: */
435 395 /* -mix_sts */
436 396 /* -dem_sts */
437 397 /* -filter_sts */
438 398 /* to measure the duration time of the interrupts */
439 399
440 400 #define FM_DEM_CF_FUNC_TIME_STS 0
441 401 /* Enable STS Objects: */
442 402 /* -mix_sts */
443 403 /* -dem_sts */
444 404 /* -filter_sts */
445 405 /* to measure the duration time of the interrupts */
446 406
447 407 #define FM_DEM_CF_INT_TIME_STS 0
448 408 /* Enable STS Objects: */
449 409 /* -mix_sts */
450 410 /* -dem_sts */
451 411 /* -filter_sts */
452 412 /* to measure the duration time of the interrupts */
453 413
454 414 #define FM_DEM_CF_INT_TIME_LED 0
455 415 /* Led ON by interrupt start OFF at the end */
456 416
457 417 #define FM_DEM_CF_INT_TIME_STS 0
458 418 /* Enable STS Objects: */
459 419 /* -mix_sts */
460 420 /* -dem_sts */
461 421 /* -filter_sts */
462 422 /* to measure the duration time of the interrupts */
463 423
464 424 #define FM_DEM_CF_FUNC_TIME_STS 0
465 425 /* Enable STS Objects: */
466 426 /* -mix_sts */
467 427 /* -dem_sts */
468 428 /* -filter_sts */
469 429 /* to measure the duration time of the interrupts */
470 430
471 431 #define FM_DEM_CF_INT_TIME_STS 0
472 432 /* Enable STS Objects: */
473 433 /* -mix_sts */
474 434 /* -dem_sts */
475 435 /* -filter_sts */
476 436 /* to measure the duration time of the interrupts */
477 437
478 438 #define FM_DEM_CF_INT_TIME_LED 0
479 439 /* Led ON by interrupt start OFF at the end */
480 440
481 441 #define FM_DEM_CF_INT_TIME_STS 0
482 442 /* Enable STS Objects: */
483 443 /* -mix_sts */
484 444 /* -dem_sts */
485 445 /* -filter_sts */
486 446 /* to measure the duration time of the interrupts */
487 447
488 448 #define FM_DEM_CF_FUNC_TIME_STS 0
489 449 /* Enable STS Objects: */
490 450 /* -mix_sts */
491 451 /* -dem_sts */
492 452 /* -filter_sts */
493 453 /* to measure the duration time of the interrupts */
494 454
495 455 #define FM_DEM_CF_INT_TIME_STS 0
496 456 /* Enable STS Objects: */
497 457 /* -mix_sts */
498 458 /* -dem_sts */
499 459 /* -filter_sts */
500 460 /* to measure the duration time of the interrupts */
501 461
502 462 #define FM_DEM_CF_INT_TIME_LED 0
503 463 /* Led ON by interrupt start OFF at the end */
504 464
505 465 #define FM_DEM_CF_INT_TIME_STS 0
506 466 /* Enable STS Objects: */
507 467 /* -mix_sts */
508 468 /* -dem_sts */
509 469 /* -filter_sts */
510 470 /* to measure the duration time of the interrupts */
511 471
512 472 #define FM_DEM_CF_FUNC_TIME_STS 0
513 473 /* Enable STS Objects: */
514 474 /* -mix_sts */
515 475 /* -dem_sts */
516 476 /* -filter_sts */
517 477 /* to measure the duration time of the interrupts */
518 478
519 479 #define FM_DEM_CF_INT_TIME_STS 0
520 480 /* Enable STS Objects: */
521 481 /* -mix_sts */
522 482 /* -dem_sts */
523 483 /* -filter_sts */
524 484 /* to measure the duration time of the interrupts */
525 485
526 486 #define FM_DEM_CF_INT_TIME_LED 0
527 487 /* Led ON by interrupt start OFF at the end */
528 488
529 489 #define FM_DEM_CF_INT_TIME_STS 0
530 490 /* Enable STS Objects: */
531 491 /* -mix_sts */
532 492 /* -dem_sts */
533 493 /* -filter_sts */
534 494 /* to measure the duration time of the interrupts */
535 495
536 496 #define FM_DEM_CF_FUNC_TIME_STS 0
537 497 /* Enable STS Objects: */
538 498 /* -mix_sts */
539 499 /* -dem_sts */
540 500 /* -filter_sts */
541 501 /* to measure the duration time of the interrupts */
542 502
543 503 #define FM_DEM_CF_INT_TIME_STS 0
544 504 /* Enable STS Objects: */
545 505 /* -mix_sts */
546 506 /* -dem_sts */
547 507 /* -filter_sts */
548 508 /* to measure the duration time of the interrupts */
549 509
550 510 #define FM_DEM_CF_INT_TIME_LED 0
551 511 /* Led ON by interrupt start OFF at the end */
552 512
553 513 #define FM_DEM_CF_INT_TIME_STS 0
554 514 /* Enable STS Objects: */
555 515 /* -mix_sts */
556 516 /* -dem_sts */
557 517 /* -filter_sts */
558 518 /* to measure the duration time of the interrupts */
559 519
560 520 #define FM_DEM_CF_FUNC_TIME_STS 0
561 521 /* Enable STS Objects: */
562 522 /* -mix_sts */
563 523 /* -dem_sts */
564 524 /* -filter_sts */
565 525 /* to measure the duration time of the interrupts */
566 526
567 527 #define FM_DEM_CF_INT_TIME_STS 0
568 528 /* Enable STS Objects: */
569 529 /* -mix_sts */
570 530 /* -dem_sts */
571 531 /* -filter_sts */
572 532 /* to measure the duration time of the interrupts */
573 533
574 534 #define FM_DEM_CF_INT_TIME_LED 0
575 535 /* Led ON by interrupt start OFF at the end */
576 536
577 537 #define FM_DEM_CF_INT_TIME_STS 0
578 538 /* Enable STS Objects: */
579 539 /* -mix_sts */
580 540 /* -dem_sts */
581 541 /* -filter_sts */
582 542 /* to measure the duration time of the interrupts */
583 543
584 544 #define FM_DEM_CF_FUNC_TIME_STS 0
585 545 /* Enable STS Objects: */
586 546 /* -mix_sts */
587 547 /* -dem_sts */
588 548 /* -filter_sts */
589 549 /* to measure the duration time of the interrupts */
590 550
591 551 #define FM_DEM_CF_INT_TIME_STS 0
592 552 /* Enable STS Objects: */
593 553 /* -mix_sts */
594 554 /* -dem_sts */
595 555 /* -filter_sts */
596 556 /* to measure the duration time of the interrupts */
597 557
598 558 #define FM_DEM_CF_INT_TIME_LED 0
599 559 /* Led ON by interrupt start OFF at the end */
600 560
601 561 #define FM_DEM_CF_INT_TIME_STS 0
602 562 /* Enable STS Objects: */
603 563 /* -mix_sts */
604 564 /* -dem_sts */
605 565 /* -filter_sts */
606 566 /* to measure the duration time of the interrupts */
607 567
608 568 #define FM_DEM_CF_FUNC_TIME_STS 0
609 569 /* Enable STS Objects: */
610 570 /* -mix_sts */
611 571 /* -dem_sts */
612 572 /* -filter_sts */
613 573 /* to measure the duration time of the interrupts */
614 574
615 575 #define FM_DEM_CF_INT_TIME_STS 0
616 576 /* Enable STS Objects: */
617 577 /* -mix_sts */
618 578 /* -dem_sts */
619 579 /* -filter_sts */
620 580 /* to measure the duration time of the interrupts */
621 581
622 582 #define FM_DEM_CF_INT_TIME_LED 0
623 583 /* Led ON by interrupt start OFF at the end */
624 584
625 585 #define FM_DEM_CF_INT_TIME_STS 0
626 586 /* Enable STS Objects: */
627 587 /* -mix_sts */
628 588 /* -dem_sts */
629 589 /* -filter_sts */
630 590 /* to measure the duration time of the interrupts */
631 591
632 592 #define FM_DEM_CF_FUNC_TIME_STS 0
633 593 /* Enable STS Objects: */
634 594 /* -mix_sts */
635 595 /* -dem_sts */
636 596 /* -filter_sts */
637 597 /* to measure the duration time of the interrupts */
638 598
639 599 #define FM_DEM_CF_INT_TIME_STS 0
640 600 /* Enable STS Objects: */
641 601 /* -mix_sts */
642 602 /* -dem_sts */
643 603 /* -filter_sts */
644 604 /* to measure the duration time of the interrupts */
645 605
646 606 #define FM_DEM_CF_INT_TIME_LED 0
647 607 /* Led ON by interrupt start OFF at the end */
648 608
649 609 #define FM_DEM_CF_INT_TIME_STS 0
650 610 /* Enable STS Objects: */
651 611 /* -mix_sts */
652 612 /* -dem_sts */
653 613 /* -filter_sts */
654 614 /* to measure the duration time of the interrupts */
655 615
656 616 #define FM_DEM_CF_FUNC_TIME_STS 0
657 617 /* Enable STS Objects: */
658 618 /* -mix_sts */
659 619 /* -dem_sts */
660 620 /* -filter_sts */
661 621 /* to measure the duration time of the interrupts */
662 622
663 623 #define FM_DEM_CF_INT_TIME_STS 0
664 624 /* Enable STS Objects: */
665 625 /* -mix_sts */
666 626 /* -dem_sts */
667 627 /* -filter_sts */
668 628 /* to measure the duration time of the interrupts */
669 629
670 630 #define FM_DEM_CF_INT_TIME_LED 0
671 631 /* Led ON by interrupt start OFF at the end */
672 632
673 633 #define FM_DEM_CF_INT_TIME_STS 0
674 634 /* Enable STS Objects: */
675 635 /* -mix_sts */
676 636 /* -dem_sts */
677 637 /* -filter_sts */
678 638 /* to measure the duration time of the interrupts */
679 639
680 640 #define FM_DEM_CF_FUNC_TIME_STS 0
681 641 /* Enable STS Objects: */
682 642 /* -mix_sts */
683 643 /* -dem_sts */
684 644 /* -filter_sts */
685 645 /* to measure the duration time of the interrupts */
686 646
687 647 #define FM_DEM_CF_INT_TIME_STS 0
688 648 /* Enable STS Objects: */
689 649 /* -mix_sts */
690 650 /* -dem_sts */
691 651 /* -filter_sts */
692 652 /* to measure the duration time of the interrupts */
693 653
694 654 #define FM_DEM_CF_INT_TIME_LED 0
695 655 /* Led ON by interrupt start OFF at the end */
696 656
697 657 #define FM_DEM_CF_INT_TIME_STS 0
698 658 /* Enable STS Objects: */
699 659 /* -mix_sts */
700 660 /* -dem_sts */
701 661 /* -filter_sts */
702 662 /* to measure the duration time of the interrupts */
703 663
704 664 #define FM_DEM_CF_FUNC_TIME_STS 0
705 665 /* Enable STS Objects: */
706 666 /* -mix_sts */
707 667 /* -dem_sts */
708 668 /* -filter_sts */
709 669 /* to measure the duration time of the interrupts */
710 670
711 671 #define FM_DEM_CF_INT_TIME_STS 0
712 672 /* Enable STS Objects: */
713 673 /* -mix_sts */
714 674 /* -dem_sts */
715 675 /* -filter_sts */
716 676 /* to measure the duration time of the interrupts */
717 677
718 678 #define FM_DEM_CF_INT_TIME_LED 0
719 679 /* Led ON by interrupt start OFF at the end */
720 680
721 681 #define FM_DEM_CF_INT_TIME_STS 0
722 682 /* Enable STS Objects: */
723 683 /* -mix_sts */
724 684 /* -dem_sts */
725 685 /* -filter_sts */
726 686 /* to measure the duration time of the interrupts */
727 687
728 688 #define FM_DEM_CF_FUNC_TIME_STS 0
729 689 /* Enable STS Objects: */
730 690 /* -mix_sts */
731 691 /* -dem_sts */
732 692 /* -filter_sts */
733 693 /* to measure the duration time of the interrupts */
734 694
735 695 #define FM_DEM_CF_INT_TIME_STS 0
736 696 /* Enable STS Objects: */
737 697 /* -mix_sts */
738 698 /* -dem_sts */
739 699 /* -filter_sts */
740 700 /* to measure the duration time of the interrupts */
741 701
742 702 #define FM_DEM_CF_INT_TIME_LED 0
743 703 /* Led ON by interrupt start OFF at the end */
744 704
745 705 #define FM_DEM_CF_INT_TIME_STS 0
746 706 /* Enable STS Objects: */
747 707 /* -mix_sts */
748 708 /* -dem_sts */
749 709 /* -filter_sts */
750 710 /* to measure the duration time of the interrupts */
751 711
752 712 #define FM_DEM_CF_FUNC_TIME_STS 0
753 713 /* Enable STS Objects: */
754 714 /* -mix_sts */
755 715 /* -dem_sts */
756 716 /* -filter_sts */
757 717 /* to measure the duration time of the interrupts */
758 718
759 719 #define FM_DEM_CF_INT_TIME_STS 0
760 720 /* Enable STS Objects: */
761 721 /* -mix_sts */
762 722 /* -dem_sts */
763 723 /* -filter_sts */
764 724 /* to measure the duration time of the interrupts */
765 725
766 726 #define FM_DEM_CF_INT_TIME_LED 0
767 727 /* Led ON by interrupt start OFF at the end */
768 728
769 729 #define FM_DEM_CF_INT_TIME_STS 0
770 730 /* Enable STS Objects: */
771 731 /* -mix_sts */
772 732 /* -dem_sts */
773 733 /* -filter_sts */
774 734 /* to measure the duration time of the interrupts */
775 735
776 736 #define FM_DEM_CF_FUNC_TIME_STS 0
777 737 /* Enable STS Objects: */
778 738 /* -mix_sts */
779 739 /* -dem_sts */
780 740 /* -filter_sts */
781 741 /* to measure the duration time of the interrupts */
782 742
783 743 #define FM_DEM_CF_INT_TIME_STS 0
784 744 /* Enable STS Objects: */
785 745 /* -mix_sts */
786 746 /* -dem_sts */
787 747 /* -filter_sts */
788 748 /* to measure the duration time of the interrupts */
789 749
790 750 #define FM_DEM_CF_INT_TIME_LED 0
791 751 /* Led ON by interrupt start OFF at the end */
792 752
793 753 #define FM_DEM_CF_INT_TIME_STS 0
794 754 /* Enable STS Objects: */
795 755 /* -mix_sts */
796 756 /* -dem_sts */
797 757 /* -filter_sts */
798 758 /* to measure the duration time of the interrupts */
799 759
800 760 #define FM_DEM_CF_FUNC_TIME_STS 0
801 761 /* Enable STS Objects: */
802 762 /* -mix_sts */
803 763 /* -dem_sts */
804 764 /* -filter_sts */
805 765 /* to measure the duration time of the interrupts */
806 766
807 767 #define FM_DEM_CF_INT_TIME_STS 0
808 768 /* Enable STS Objects: */
809 769 /* -mix_sts */
810 770 /* -dem_sts */
811 771 /* -filter_sts */
812 772 /* to measure the duration time of the interrupts */
813 773
814 774 #define FM_DEM_CF_INT_TIME_LED 0
815 775 /* Led ON by interrupt start OFF at the end */
816 776
817 777 #define FM_DEM_CF_INT_TIME_STS 0
818 778 /* Enable STS Objects: */
819 779 /* -mix_sts */
820 780 /* -dem_sts */
821 781 /* -filter_sts */
822 782 /* to measure the duration time of the interrupts */
823 783
824 784 #define FM_DEM_CF_FUNC_TIME_STS 0
825 785 /* Enable STS Objects: */
826 786 /* -mix_sts */
827 787 /* -dem_sts */
828 788 /* -filter_sts */
829 789 /* to measure the duration time of the interrupts */
830 790
831 791 #define FM_DEM_CF_INT_TIME_STS 0
832 792 /* Enable STS Objects: */
833 793 /* -mix_sts */
834 794 /* -dem_sts */
835 795 /* -filter_sts */
836 796 /* to measure the duration time of the interrupts */
837 797
838 798 #define FM_DEM_CF_INT_TIME_LED 0
839 799 /* Led ON by interrupt start OFF at the end */
840 800
841 801 #define FM_DEM_CF_INT_TIME_STS 0
842 802 /* Enable STS Objects: */
843 803 /* -mix_sts */
844 804 /* -dem_sts */
845 805 /* -filter_sts */
846 806 /* to measure the duration time of the interrupts */
847 807
848 808 #define FM_DEM_CF_FUNC_TIME_STS 0
849 809 /* Enable STS Objects: */
850 810 /* -mix_sts */
851 811 /* -dem_sts */
852 812 /* -filter_sts */
853 813 /* to measure the duration time of the interrupts */
854 814
855 815 #define FM_DEM_CF_INT_TIME_STS 0
856 816 /* Enable STS Objects: */
857 817 /* -mix_sts */
858 818 /* -dem_sts */
859 819 /* -filter_sts */
860 820 /* to measure the duration time of the interrupts */
861 821
862 822 #define FM_DEM_CF_INT_TIME_LED 0
863 823 /* Led ON by interrupt start OFF at the end */
864 824
865 825 #define FM_DEM_CF_INT_TIME_STS 0
866 826 /* Enable STS Objects: */
867 827
```

```

131     short out_buffer_B(BUFFERSIZE_OUTPUT);
132
133
134     short* in_buffer_ptr;
135     short* out_buffer_ptr;
136     Bool a_inDMA=TRUE;
137     Bool a_inDMA=TRUE;
138     short outcounter=0;
139
140     /*-----
141     the main function
142     void main()
143     {
144         HWI_disable();
145
146         #if FM DEM CF DEBUG
147             LOG_printf(debug_log,"Info : Begin of main ");
148         #endif
149
150         CSI_init(); /* on chip init */
151         BSL_init(); /* on board init */
152
153         tsh1408_init(in_buffer_A,in_buffer_B);
154         codec_init();
155
156         in_buffer_A(0)=10;
157
158         HWI_enable();
159
160         tsh1408_start();
161         codec_start();
162
163
164         #if FM DEM CF INT_TIME_LED
165             LED_off(LED_ALL);
166         #endif
167
168         #if FM DEM CF DEBUG
169             LOG_printf(debug_log,"Info : End of main ");
170         #endif
171     }
172
173     /*-----
174     hardware interrupt functions
175
176     void edmaInt(void)
177     {
178         #if FM DEM CF INT_OCCUR_LED
179             LED_toggle(LED_2);
180         #endif
181
182         #if FM DEM CF INT_TIME_LED
183             LED_on(LED_2);
184         #endif
185
186         #if FM DEM CF INT_TIME_STS
187             STS_set(edma_sts,CLK_gettime());
188         #endif
189
190         /* reset outcounter */
191         outcounter=0;
192
193         /* select EDMA channel */
194         #if EDMA_intTest(TSH1408_BUFFER_A_TCC)
195         {
196             #if FM DEM CF DEBUG
197                 LOG_printf(debug_log," In A");
198             #endif
199             /* Clear transfer completion interrupt flag */
200             EDMA_intClear(TSH1408_BUFFER_A_TCC);
201
202             /* set buffer pointers */
203             a_inDMA=FALSE;
204             in_buffer_ptr=in_buffer_A;
205             out_buffer_ptr=out_buffer_A;
206
207             /* post software interrupt -> start demodulation*/
208             SWI_post(edmodulate_swi);
209
210
211     }
212     if (EDMA_intTest(TSH1408_BUFFER_B_TCC))
213     {
214         #if FM DEM CF DEBUG
215             LOG_printf(debug_log," In B");
216         #endif
217         /* Clear transfer completion interrupt flag */
218         EDMA_intClear(TSH1408_BUFFER_B_TCC);
219
220         /* set buffer pointers */
221         a_inDMA=TRUE;
222         in_buffer_ptr=in_buffer_B;
223         out_buffer_ptr=out_buffer_B;
224
225         /* post software interrupt -> start demodulation*/
226         SWI_post(edmodulate_swi);
227     }
228     #if FM DEM CF INT_TIME_STS
229         STS_delta(edma_sts,CLK_gettime());
230     #endif
231     #if FM DEM CF INT_TIME_LED
232         LED_off(LED_2);
233     #endif
234
235     void dacInt(void)
236     {
237         #if FM DEM CF INT_OCCUR_LED
238             LED_toggle(LED_1);
239         #endif
240
241         #if FM DEM CF INT_TIME_LED
242             LED_on(LED_1);
243         #endif
244
245         #if FM DEM CF INT_TIME_STS
246             STS_set(edac_sts,CLK_gettime());
247         #endif
248
249         /* interpolate a missing sample */
250         if (outcounter>15)
251         {
252             #if FM DEM CF INT_OCCUR_LED
253                 LED_toggle(LED_3);
254             #endif
255             /* select the buffer */
256             if(a_inDMA)
257             {
258                 AD535_HWI_write(outAD535h,(( out_buffer_A[15]+out_buffer_B(0))/2));
259                 #if FM DEM CF DEBUG
260                     LOG_printf(debug_log,"Out A error :%d",outcounter);
261                 #endif
262             }
263             else
264             {
265                 AD535_HWI_write(outAD535h,((out_buffer_B[15]+out_buffer_A(0))/2));
266                 #if FM DEM CF DEBUG
267                     LOG_printf(debug_log,"Out B error :%d",outcounter);
268                 #endif
269             }
270
271             /* output a value */
272             /* select the buffer */
273             if(a_inDMA)
274             {
275                 AD535_HWI_write(outAD535h,out_buffer_A[outcounter]);
276                 #if FM DEM CF DEBUG
277                     LOG_printf(debug_log,"Out A :%d",outcounter);
278                 #endif
279             }
280             else
281             {
282                 AD535_HWI_write(outAD535h,out_buffer_B[outcounter]);
283                 #if FM DEM CF DEBUG
284                     LOG_printf(debug_log,"Out B :%d",outcounter);
285                 #endif
286             }
287
288             /* update outcounter */
289             outcounter++;
290

```

```

291 #if FM_DEM_CF_INT_TIME_STS
292 STS_delta(&dem_sts,CLK_gettime());
293 #endif
294
295 #if FM_DEM_CF_INT_TIME_LED
296 LED_off(LED_1);
297 #endif
298
299 }
300
301 /*-----
302 software interrupt functions
303 -----*/
304
305 void demodulateSwiFunc(void)
306 {
307     int i;
308
309     #if FM_DEM_CF_INT_TIME_LED
310     LED_on(LED_3);
311     #endif
312
313     #if FM_DEM_CF_INT_TIME_STS
314     STS_set(&demodulate_sts,CLK_gettime());
315     #endif
316
317     for (i=0 ;i<BUFFERSIZE_INPUT;i++)
318     {
319         in_buffer_ptr[i]=in_buffer_ptr[i]<<2;
320     }
321
322     #if FM_DEM_CF_FUNC_TIME_STS
323     STS_set(&mix_sts,CLK_gettime());
324     #endif
325
326     quad_mix(in_buffer_ptr,re_buffer,im_buffer);
327     #if FM_DEM_CF_FUNC_TIME_STS
328     STS_delta(&mix_sts,CLK_gettime());
329     #endif
330
331     #if FM_DEM_CF_FUNC_TIME_STS
332     STS_set(&dem_sts,CLK_gettime());
333     #endif
334
335     #if (FM_DEMODULATOR==FM_PIL_DEM)
336     pll_demodulate(re_buffer,im_buffer,out_buffer);
337     #elif (FM_DEMODULATOR==FM_MIXED_DEM)
338     mixed_demodulate(re_buffer,im_buffer,out_buffer);
339     #else
340     #error No demodulator algorithm defined
341     #endif
342
343     #if FM_DEM_CF_FUNC_TIME_STS
344     STS_delta(&dem_sts,CLK_gettime());
345     #endif
346
347     #if FM_DEM_CF_FUNC_TIME_STS
348     STS_set(&filter_sts,CLK_gettime());
349     #endif
350
351     out_filter(out_buffer,out_buffer_ptr);
352     #if FM_DEM_CF_FUNC_TIME_STS
353     STS_delta(&filter_sts,CLK_gettime());
354     #endif
355
356     #if FM_DEM_CF_INT_TIME_LED
357     LED_off(LED_3);
358     #endif
359 }

```

## quad\_mix

Listing F.36: Fixed-Point Optimized :: quad\_mix.c

```

1 /*
2 QUADRATURE MIXER - VERSION FIXED POINT 1
3
4 file : quad_mix.c
5 date : oct 2001 - jan 2002
6 by : c. haller
7 f. schnyder
8
9 */
10
11 /*
12 includes
13 -----
14 #include "quad_mix.h"
15 -----
16
17 Global variables
18 -----
19
20 /*
21 coefficients of sine and cosine oscillator
22 -----
23 /* Frequency : 13139.93 Hz*/
24
25 short gen_b0_cos = 16583; /*cos*/
26 short gen_b1_cos = -4593; /*cos*/
27 short gen_b1_sin = -15934; /*sin*/
28 short gen_a1 = -18152; /*sin/cos*/
29
30 /* delay variables */
31 short gen_z[3] = {0.0,26213}; /* 0.8 for first value of delta function*/
32
33 /*
34 coefficients of mixer low-pass (mix_lp)
35 -----
36 /* Order : 5
37 /* Pass-band : 2 dB @ 6300.00 Hz */
38 /* Stop-band : 60 dB @ 19000.00 Hz */
39

```

Listing F.35: Fixed-Point Optimized :: quad\_mix.h

```

22  /* ----- */
23  /*
24  FW demodulates the the two baseband signals re_buffer and im_buffer
25  into the out_buffer with the mixed algorithm
26
27  Parameters :
28  re_buffer reel FW path in baseband
29  im_buffer inag FW path in baseband
30  out_buffer demodulated signal
31  */
32  void mixed_demodulate(short re_buffer[], short im_buffer[], short out_buffer[]);
33
34  #endif /* _MIXED_DEMODULATE_ */
35
36  /* ----- */
37  END OF THE FILE
38  */

```

Listing F38: Fixed-Point Optimized :: mixed\_demodulate.c

```

1  /*
2  MIXED DEMODULATOR - VERSION FIXED POINT 1
3
4  file : mixed_demodulate.c
5  date : oct 2001 - jan 2002
6  by : c. haller
7  f. schnyder
8  */
9
10 /* ----- */
11 includes
12 ----- */
13 #include "mixed_demodulate.h"
14
15 /* ----- */
16 Global variables
17 ----- */
18 /* delay variables of the mixed demodulator */
19 short dem_z_re=16384; /* !=0 to avoid division with 0 */
20 short dem_z_im=16384; /* !=0 to avoid division with 0 */
21
22 /* ----- */
23 implementation of function
24 ----- */
25 void mixed_demodulate(short re_buffer[restrict], short im_buffer[restrict],
26 short out_buffer[restrict])
27 {
28     int i;
29     int re_path=0;
30     int im_path=0;
31     int dem_arg=0;
32
33     WORD_ALIGNED(re_buffer);
34     WORD_ALIGNED(im_buffer);
35     WORD_ALIGNED(out_buffer);
36
37     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
38     {
39         re_path = (im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
40         im_path = (re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */
41
42         /* shift delays */
43         dem_z_re=re_buffer[i];
44         dem_z_im=im_buffer[i];
45
46         /* calculate the argument */
47         dem_arg=re_path/(im_path>>14); /* Q17.14 */
48
49         /* atan lookup */
50         if(dem_arg>0) /* positiv */
51         {
52             dem_arg=dem_arg>>(ATAN_SHIFT);
53             if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
54             out_buffer[i]=atan_buffer[dem_arg];
55         }
56         /* negativ */
57         {
58             dem_arg=-dem_arg;
59             dem_arg=dem_arg>>(ATAN_SHIFT);
60         }
61     }
62 }

```

```

40 #define MIX_LP_CASCADES 3
41 short mix_lp_a1[3]={ /* a_i1a a_i1b a_i2 */
42     { -8092 , -8092 , 0 },
43     { -17444 , -17444 , 11166 },
44     { -21907 , -21907 , 22406 },
45 };
46
47 short mix_lp_b1[3]={ /* b_i0 b_i1 b_i2 */
48     { 4523 , 4523 , 0 },
49     { 2725 , 5450 , 2725 },
50     { 8537 , 17074 , 8537 },
51 };
52
53 /* Scaling included in output of oszis */
54
55 /* delay variables */
56 short mix_lp_z_re[3]={
57     { 0,0,0 },
58     { 0,0,0 },
59     { 0,0,0 },
60 };
61 short mix_lp_z_im[3]={
62     { 0,0,0 },
63     { 0,0,0 },
64     { 0,0,0 },
65 };
66
67 /* delay variables */
68 short as_mix_lp_z_re[3]={
69     { 0,0,0 },
70     { 0,0,0 },
71     { 0,0,0 },
72 };
73 short as_mix_lp_z_im[3]={
74     { 0,0,0 },
75     { 0,0,0 },
76     { 0,0,0 },
77 };
78
79 /* ----- */
80 implementation of function
81 ----- */
82
83 /* implementation in linear assembly in
84 file quad_mix_lin_asm !!! */
85
86 /* ----- */
87 END OF THE FILE
88 */

```

## mixed\_demodulate

Listing F37: Fixed-Point Optimized :: mixed\_demodulate.h

```

1  /*
2  MIXED DEMODULATOR - VERSION FIXED POINT 1
3
4  file : mixed_demodulate.h
5  date : oct 2001 - jan 2002
6  by : c. haller
7  f. schnyder
8  */
9
10
11 #ifndef _MIXED_DEMODULATE_
12 #define _MIXED_DEMODULATE_
13
14 /* ----- */
15 includes
16 ----- */
17 #include "global_settings.h"
18 #include "atan.h"
19
20 /* ----- */
21 global functions

```



```

59         if(dem_arg>BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
60         out_buffer[i]=atan_buffer[dem_arg];
61     }
62 }
63 }
64 }
65 /*
66 END OF THE FILE
67
68 */

```

### Listing F39: Fixed-Point Optimized Interpolated mixed\_demodulate.h

```

1  /*
2  MIXED DEMODULATOR - VERSION FIXED POINT 1
3
4  file : mixed_demodulate.h
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8
9  */
10
11 #ifndef _MIXED_DEMODULATE_
12 #define _MIXED_DEMODULATE_
13
14 /*
15 includes
16
17 #include "global_settings.h"
18 #include "atan.h"
19
20 */
21 global functions
22
23 /*
24 FM demodulates the two baseband signals re_buffer and im_buffer
25 into the out_buffer with the mixed algorithm
26
27 Parameters :
28 re_buffer real FM path in baseband
29 im_buffer imag FM path in baseband
30 out_buffer demodulated signal
31 */
32 void mixed_demodulate(short re_buffer[], short im_buffer[], short out_buffer[]);
33
34 #endif /* _MIXED_DEMODULATE_ */
35
36 /*
37 END OF THE FILE
38
39 */

```

### Listing F40: Fixed-Point Optimized Interpolated mixed\_demodulate.c

```

1  /*
2  MIXED DEMODULATOR - VERSION FIXED POINT 1
3
4  file : mixed_demodulate.c
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8
9  */
10
11 /*
12 includes
13
14 #include "mixed_demodulate.h"
15
16 */
17 global variables
18
19 /* delay variables of the mixed demodulator */

```

```

20 short dem_z_re=16384; /* !=0 to avoid division with 0 */
21 short dem_z_im=16384; /* !=0 to avoid division with 0 */
22
23 /*
24 implementation of function
25
26 void mixed_demodulate(short re_buffer[restrict], short im_buffer[restrict],
27 short out_buffer[restrict])
28 {
29     int i;
30     int re_path=0;
31     int im_path=0;
32     int dem_arg=0;
33     short cutoff=0;
34     WORD_ALIGNED(re_buffer);
35     WORD_ALIGNED(im_buffer);
36     WORD_ALIGNED(out_buffer);
37
38     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
39     {
40         re_path = ( im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
41         im_path = ( re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */
42
43         /* shift delays */
44         dem_z_re=re_buffer[i];
45         dem_z_im=im_buffer[i];
46
47         /* calculate the argument */
48         dem_arg=re_path/(im_path>>14); /* Q17.14 */
49
50         /* atan lookup */
51         if(dem_arg>0) /*positive*/
52         {
53             cutoff=dem_arg*ATAN_INTERPOL;
54             dem_arg=dem_arg>>(ATAN_SHIFT);
55             if(dem_arg>BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
56             out_buffer[i]=atan_buffer[dem_arg]+((cutoff*atan_grad_buffer[dem_arg])>>
57             ATAN_SHIFT);
58         }
59         else /*negative*/
60         {
61             dem_arg=-dem_arg;
62             cutoff=dem_arg*ATAN_INTERPOL;
63             dem_arg=dem_arg>>(ATAN_SHIFT);
64             if(dem_arg>BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
65             out_buffer[i]=-(atan_buffer[dem_arg]+((cutoff*atan_grad_buffer[dem_arg])>>
66             >>ATAN_SHIFT));
67         }
68     }
69 }
70
71 /*
72 END OF THE FILE
73
74 */

```

### pll\_demodulate

### Listing F41: Fixed-Point Optimized :: pll\_demodulate.h

```

1  /*
2  PLL DEMODULATOR - VERSION FIXED POINT 1
3
4  file : pll_demodulate.h
5  date : oct 2001 - jan 2002
6  by   : c. haller
7        f. schnyder
8
9  */
10
11 #ifndef _PLL_DEMODULATE_
12 #define _PLL_DEMODULATE_
13
14 /*

```

```

15  includes
16  -----*/
17  #include "global_settings.h"
18  #include "sine.h"
19
20  /*-----
21  global functions
22  -----*/
23
24  /*
25  FM demodulates the two baseband signals re_buffer and im_buffer
26  into the out_buffer with the pll algorithm
27
28  Parameters :
29  re_buffer real FM path in baseband
30  im_buffer imag FM path in baseband
31  out_buffer demodulated signal
32  */
33  void pll_demodulate(short re_buffer[], short im_buffer[], short out_buffer[]);
34  #endif /* _PLL_DEMODULATE_ */
35
36  /*
37  END OF THE FILE
38  */

```

Listing F.42: Fixed-Point Optimized :: pll\_demodulate.c

```

52  i_path=(re_buffer[i]*i_path)>>15;
53  q_path=(im_buffer[i]*q_path)>>15;
54
55  demo=q_path-i_path;
56  demo=(demo*P_PLL)>>15;
57
58  out_buffer[i]=(demo*GAIN_PLL)>>15;
59
60  /*
61  summation of demodulated signal, using Q3.12_*/
62
63  demo=demo>>3;
64  arg=arg+((demo*K_PLL)>>12);
65
66  /*
67  limitation of argument to 2*PI, using Q3.12_*/
68
69  if (arg>TWO_PI)
70  {
71      arg=arg-TWO_PI;
72  }
73
74  else if (arg<-TWO_PI)
75  {
76      arg=arg+TWO_PI;
77  }
78
79  /*
80  calculation of sine and cosine value of argument_*/
81
82  /*calculating of index for sine_buffer access, using new Q format*/
83
84  index=arg>>REDUCTION;
85
86  if (index<0)
87  {
88      index=-index;
89      flag=1;
90  }
91
92  index=(index*STEP_INVERSE)>>(2*Q_FORMAT);
93
94  if (flag==1)
95  {
96      index=-index;
97      flag=0;
98  }
99
100  /*access on sine_buffer with index*/
101
102  if (index>=0)
103  {
104      if (index<(BUFFERSIZE_SINE-1))
105      {
106          /*sine value*/
107          i_path=sine_buffer[index];
108          /*cosine value*/
109          index=(BUFFERSIZE_SINE-1)-index;
110          q_path=sine_buffer[index];
111      }
112      else if (index<2*(BUFFERSIZE_SINE-1))
113      {
114          index=index % (BUFFERSIZE_SINE-1);
115          /*sine value*/
116          index=(BUFFERSIZE_SINE-1)-index;
117          i_path=sine_buffer[index];
118          /*cosine value*/
119          index=(BUFFERSIZE_SINE-1)-index;
120          q_path=sine_buffer[index];
121      }
122      else if (index<3*(BUFFERSIZE_SINE-1))
123      {
124          index=index % (BUFFERSIZE_SINE-1);
125          /*sine value*/
126          i_path=sine_buffer[index];
127          /*cosine value*/
128          index=(BUFFERSIZE_SINE-1)-index;
129          q_path=sine_buffer[index];
130      }
131      else if (index<4*(BUFFERSIZE_SINE-1))

```

## out\_filter

Listing F.43: Fixed-Point Optimized :: out\_filter.h

```

1 /*
2  * OUTPUT AND NOISE FILTER - VERSION FIXED POINT 1
3  */
4 #include "out_filter.h"
5 #define DATE "oct 2001 - jan 2002"
6 #define BY "c. haller"
7 #define F. schwyder
8
9 /*
10  *
11  * #ifndef _OUT_FILTER_
12  * #define _OUT_FILTER_
13  *
14  *
15  * includes
16  *
17  * #include "global_settings.h"
18  *
19  *
20  * global functions
21  *
22  *
23  * low-pass filters the out_buffer_down_one signal downsamples it
24  * and then filters it with an high-pass and writes it to
25  * out_buffer_down_two
26  *
27  * Parameters :
28  * out_buffer_down_one input buffer
29  * out_buffer_down_two output buffer
30  *
31  * void out_filter(short out_buffer[]) . short out_buffer_down[];
32  *
33  * #endif /* _OUT_FILTER_ */
34  *
35  *
36  * END OF THE FILE
37  */

```

Listing F.44: Fixed-Point Optimized :: out\_filter.c

```

1 /*
2  * OUTPUT AND NOISE FILTER - VERSION FIXED POINT 1
3  */
4 #include "out_filter.h"
5 #define DATE "oct 2001 - jan 2002"
6 #define BY "c. haller"
7 #define F. schwyder
8
9 /*
10  *
11  *
12  * includes
13  *
14  * #include "out_filter.h"
15  *
16  *
17  *
18  * global variables
19  *
20  *
21  * coefficients of noise low-pass (no_lp) after demodulation
22  *
23  *
24  * Order : 13
25  * Pass-band : 2 dB @ 3500.00 Hz */
26  * Stop-band : 20 dB @ 3999.57 Hz */
27
28 #define NO_LP_CASCADES 7
29 #define NO_LP_A1_1 0.0
30 #define NO_LP_A1_2 0.0
31 #define NO_LP_A1_3 0.0
32 #define NO_LP_A1_4 0.0
33 #define NO_LP_A1_5 0.0
34 #define NO_LP_A1_6 0.0

```

```

132 {
133     index=index % (BUFFERSIZE_SINE-1);
134     /*sine value*/
135     index=(BUFFERSIZE_SINE-1)-index;
136     i_path=sine_buffer[index];
137     /*cosine value*/
138     index=(BUFFERSIZE_SINE-1)-index;
139     q_path=sine_buffer[index];
140 }
141
142 else
143 {
144     index=index % (BUFFERSIZE_SINE-1);
145     /*sine value*/
146     i_path=sine_buffer[index];
147     /*cosine value*/
148     index=(BUFFERSIZE_SINE-1)-index;
149     q_path=sine_buffer[index];
150 }
151
152 else
153 {
154     if(index>=(BUFFERSIZE_SINE-1))
155     {
156         index=index;
157         /*sine value*/
158         i_path=sine_buffer[index];
159         /*cosine value*/
160         index=(BUFFERSIZE_SINE-1)-index;
161         q_path=sine_buffer[index];
162     }
163     else if(index>2*(BUFFERSIZE_SINE-1))
164     {
165         index=-(index % (BUFFERSIZE_SINE-1));
166         /*sine value*/
167         index=(BUFFERSIZE_SINE-1)-index;
168         i_path=sine_buffer[index];
169         /*cosine value*/
170         index=(BUFFERSIZE_SINE-1)-index;
171         q_path=sine_buffer[index];
172     }
173     else if (index>3*(BUFFERSIZE_SINE-1))
174     {
175         index=-(index % (BUFFERSIZE_SINE-1));
176         /*sine value*/
177         i_path=sine_buffer[index];
178         /*cosine value*/
179         index=(BUFFERSIZE_SINE-1)-index;
180         q_path=sine_buffer[index];
181     }
182     else if (index>4*(BUFFERSIZE_SINE-1))
183     {
184         index=-(index % (BUFFERSIZE_SINE-1));
185         /*sine value*/
186         index=(BUFFERSIZE_SINE-1)-index;
187         i_path=sine_buffer[index];
188         /*cosine value*/
189         index=(BUFFERSIZE_SINE-1)-index;
190         q_path=sine_buffer[index];
191     }
192     else
193     {
194         index=-(index % (BUFFERSIZE_SINE-1));
195         /*sine value*/
196         i_path=sine_buffer[index];
197         /*cosine value*/
198         index=(BUFFERSIZE_SINE-1)-index;
199         q_path=sine_buffer[index];
200     }
201 }
202 }
203 }
204
205 /*
206  * END OF THE FILE
207  */

```

```

35     { -8497 , 15808 },
36     { -10247 , 25815 },
37 };
38
39 short no_lp_bf[3]={ /* b_i_0 b_i_1 b_i_2 */
40     { 13822 , 13822 , 0 },
41     { 7222 , 14444 , 7222 },
42     { 7783 , 15565 , 7783 },
43     { 8669 , 17339 , 8669 },
44     { 10020 , 20040 , 10020 },
45     { 7947 , 15893 , 7947 },
46     { 5296 , 10592 , 5296 },
47 };
48
49 /*short no_lp_scale= 23904;*/ /* 0.911862*0.8 to avoid overflows */
50
51 /* delay variables */
52 short no_lp_z[3]={
53     { 0,0,0},
54     { 0,0,0},
55     { 0,0,0},
56     { 0,0,0},
57     { 0,0,0},
58     { 0,0,0},
59     { 0,0,0},
60 };
61
62 /*-----
63 coefficients of end high-pass (en_hp) before dac
64 -----*/
65 /* Order : 4 */
66 /* Pass-band : 0.1 dB @ 300.00 Hz */
67 /* Stop-band : 40.0 dB @ 50.00 Hz */
68
69 #define EN_HP_CASCADES 2
70 short en_hp_a[3]={ /* aa_i_1a a_i_1b a_i_2 */
71     { -29180 , -29180 , 26045 },
72     { -31046 , -31046 , 29805 },
73 };
74
75 /*
76 short en_hp_b[3]={ b_i_0 b_i_1 b_i_2
77     { 1 , -2 , 1 },
78     { 1 , -2 , 1 },
79 };
80 */
81
82 short en_hp_scale=13931; /* 0.5*0.850269 to avoid overflows */
83 #define EN_HP_UPSCALE 2
84 /* delay variables */
85 short en_hp_z[3]={
86     { 0,0,0},
87     { 0,0,0},
88     { 0,0,0},
89 };
90
91 /*-----
92 implementation of function
93 -----*/
94 void out_filter(short out_buffer[restrict], short out_buffer_down[restrict])
95 {
96     int i,j,k;
97
98     WORD_ALIGNED(out_buffer);
99     WORD_ALIGNED(out_buffer_down);
100
101     k=0;
102
103     /*_ the low-pass _*/
104
105     /* all cascades except fiert the last one */
106     for(j=0;j<(NO_LP_CASCADES-1);j++)
107     {
108         for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
109         {
110             /* feedback */
111             no_lp_z[j][0] = out_buffer[i]-( ( no_lp_a[j][0]*no_lp_z[j][1] + /*a_1*z1
112
113
114
115             no_lp_a[j][1]*no_lp_z[j][2] /*a_2 *z2*/
116             )>>15);
117
118             /* forward */
119             out_buffer[i]= ( no_lp_b[j][0]*no_lp_z[j][0]+ /* b_0 * z0 */
120                 no_lp_b[j][1]*no_lp_z[j][1]+ /* b_1 * z1 */
121                 no_lp_b[j][2]*no_lp_z[j][2] /* b_2 * z2 */
122                 )>>15;
123
124             /* shift delays */
125             no_lp_z[j][2]=no_lp_z[j][1];
126             no_lp_z[j][1]=no_lp_z[j][0];
127         }
128     }
129
130     /* last cascade including downsampling*/
131     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
132     {
133         /* feedback */
134         no_lp_z[NO_LP_CASCADES-1][0] = out_buffer[i]-( ( no_lp_a[NO_LP_CASCADES
135             no_lp_a[NO_LP_CASCADES-1][1] + /*a_1*z1*/
136             no_lp_a[NO_LP_CASCADES-1][2] /*a_2 *z2*/
137             )>>15);
138
139         /* forward including downsampling -> take every 2nd value*/
140         if(i%DOWNSAMPLE_TWO==0)
141         {
142             out_buffer_down[k]= ( no_lp_b[NO_LP_CASCADES-1][0]*no_lp_z[NO_LP_CASCADES
143                 no_lp_b[NO_LP_CASCADES-1][1]*no_lp_z[NO_LP_CASCADES
144                 no_lp_b[NO_LP_CASCADES-1][2]*no_lp_z[NO_LP_CASCADES
145                 )>>15;
146             k++;
147         }
148
149         /* shift delays */
150         no_lp_z[NO_LP_CASCADES-1][2]=no_lp_z[NO_LP_CASCADES-1][1];
151         no_lp_z[NO_LP_CASCADES-1][1]=no_lp_z[NO_LP_CASCADES-1][0];
152     }
153
154     /*_ the high-pass _*/
155
156     /* high-pass in transposed form */
157     for(j=0;j<(EN_HP_CASCADES);j++)
158     {
159         for(i=0;i<BUFFERSIZE_OUTPUT;i++)
160         {
161             en_hp_z[j][2]= en_hp_z[j][1]+out_buffer_down[i];
162             en_hp_z[j][1]= -out_buffer_down[i]-out_buffer_down[i]-
163                 ((en_hp_a[j][0]*en_hp_z[j][2]+
164                 en_hp_a[j][1]*en_hp_z[j][2])>>15)+
165                 en_hp_z[j][0];
166             en_hp_z[j][0]= out_buffer_down[i]-
167                 ((en_hp_a[j][2]*en_hp_z[j][2])>>15);
168             out_buffer_down[i]=en_hp_z[j][2];
169         }
170     }
171
172     for(i=0;i<BUFFERSIZE_OUTPUT;i++)
173     {
174         out_buffer_down[i]=out_buffer_down[i]*EN_HP_UPSCALE;
175     }
176
177 }
178
179 /*
180 END OF THE FILE
181 */
182

```

## F.1.5. Floating-Point Adaptive mixed\_demodulate

Listing F45: Floating-Point Adaptive mixed\_demodulate.h ::

```

1  /*
2  MIXED DEMODULATOR WITH CARRIER CORRECTION
3  - VERSION FLOATING POINT 1.1
4
5  file : mixed_demodulate.h
6  date : oct 2001 - Jan 2002
7  by   : c. haller
8       f. schnyder
9
10 */
11
12 #ifndef _MIXED_DEMODULATE_
13 #define _MIXED_DEMODULATE_
14
15 /*-----*/
16 #include "global_settings.h"
17 #include <math.h>
18
19 /*-----*/
20
21 /*-----*/
22 global functions
23
24 /* The re_buffer_down_one and im_buffer_down_one are demodulated
25 with the mixed demodulator algorithm to out_buffer_down_one.
26
27 Parameters :
28 re_buffer_down_one buffer of downsampled I signal
29 im_buffer_down_one buffer of downsampled Q signal
30 out_buffer_down_one buffer of demodulated signal
31 */
32 void mixed_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] ,
33                      float out_buffer_down_one[]);
34
35 #endif /* _MIXED_DEMODULATE_ */
36
37 /*-----*/
38
39 END OF THE FILE
40 */

```

Listing F46: Floating-Point Adaptive mixed\_demodulate.c ::

```

1  /*
2  MIXED DEMODULATOR WITH CARRIER CORRECTION
3  - VERSION FLOATING POINT 1.1
4
5  file : mixed_demodulate.c
6  date : oct 2001 - Jan 2002
7  by   : c. haller
8       f. schnyder
9
10 */
11
12 /*-----*/
13 #include
14 #include "mixed_demodulate.h"
15
16 /*-----*/
17 global variables
18
19 /*variables of demodulating (attention division by zero!)*/*
20
21 float coold=0.0; /*old value of I-path*/
22

```

```

23 float conew=1.0; /*new value of I-path*/
24 float siold=0.0; /*old value of Q-path*/
25 float sinew=1.0; /*new value of Q-path*/
26 float demo=0.0; /*value demodulating*/
27
28 /*constant of demodulating*/
29
30 float decon=0.7;
31
32 /*variables of adaptive filtering*/
33
34 int counter=0;
35
36 float sum=0;
37 float wMe=82561; /*exstant mixer angular frequency*/
38 float wMs=0; /*supposed mixer angular frequency*/
39
40 extern float bc2;
41 extern float bs2;
42 extern float asc2;
43
44 /*constants of adaptive filtering*/
45 #define N 32000 /**BUFFERSIZE_DEMODULATOR*/
46 #define deriv 3000
47
48 float T=0.000015625; /*sample time*/
49 float adcon=2*Pi*derivation/N;
50
51
52
53
54 /*-----*/
55 implementation of function
56
57 void mixed_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] ,
58                      float out_buffer_down_one[])
59 {
60     int i;
61     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
62     {
63         /* shifting values */
64         coold=conew;
65         siold=sinew;
66         conew=re_buffer_down_one[i];
67         sinew=im_buffer_down_one[i];
68
69         /* calculating demodulated value */
70         demo=(coold*sinew-siold*conew)/(coold*conew+siold*sinew); /*atan argument*/
71         demo=atanf(demo); /*atan value*/
72         demo=demo*decon; /*demodulated value*/
73         out_buffer_down_one[i]=demo;
74
75         sum=sum+demo;
76         counter++;
77     }
78
79     if(counter==N)
80     {
81         wMs=wMe;
82         wMe=wMs+sum*adcon;
83
84         bc2=-cosf(wMe*T);
85         bs2=sinf(wMe*T);
86         asc2=-2*cosf(wMe*T);
87
88         sum=0;
89         counter=0;
90     }
91
92
93 }
94
95
96 /*-----*/
97 END OF THE FILE
98 */

```

## F2. Linear Assembly Listings

### F2.1. Fixed-Point Optimized

#### quad\_mix

*Listing F47: Fixed-Point Optimized :: quad\_mix\_lin\_ass.sa*

```

1 ;
2 ; QUADRATURE MIXER - VERSION FIXED POINT 1
3 ;
4 ;
5 ; file : quad_mix.c
6 ; date : oct 2001 - jan 2002
7 ; by : c. haller
8 ; f. schnyder
9 ;
10
11
12 .global _quad_mix
13 .global _mix_lp_a
14 .global _mix_lp_b
15 .global _mix_lp_z_re
16 .global _mix_lp_z_im
17 .global _gen_b0_cos
18 .global _gen_b1_cos
19 .global _gen_b1_sin
20 .global _gen_a1
21 .global _gen_z
22
23 .sect *.text*
24
25 _quad_mix: .cproc ins, re_b_s, im_b_s
26
27 .reg re_b, im_b
28 .reg cnt
29 .reg ptr_re, ptr_im
30
31 .reg re_in, re_prodl, re_prod2
32 .reg im_in, re_prodl, im_prod2
33
34 .reg re_a1a, re_a1b, re_a2, re_b0, re_b1, re_b2
35 .reg im_a1a, im_a1b, im_a2, im_b0, im_b1, im_b2
36
37 .reg re_z0, re_z1, re_z2
38 .reg im_z0, im_z1, im_z2
39
40 .no_mdep
41
42
43 ;=====
44 ; BEGIN mixing
45 ;=====
46 ; copy address of input buffers
47 .reg re_b_s, re_b
48 .reg im_b_s, im_b
49
50 ; load filter coefficients
51 .MVKL _gen_a1, ptr_re
52 .MVKL _gen_b1_sin, ptr_im
53 .MVKH _gen_a1, ptr_re
54 .MVKH _gen_b1_sin, ptr_im
55
56 .LDH *ptr_re, re_a1a
57 .LDH *ptr_im, im_b1
58
59 .MVKL _gen_b0_cos, ptr_re
60 .MVKL _gen_b1_cos, ptr_im
61 .MVKH _gen_b0_cos, ptr_re
62 .MVKH _gen_b1_cos, ptr_im
63
64 .LDH *ptr_re, re_b0
65 .LDH *ptr_im, re_b1
66
67 ; load delay states
68
69 .MVKL _gen_z, ptr_re
70 .MVKH _gen_z, ptr_re
71
72 .LDH **ptr_re, re_z1
73 .LDH **ptr_re, re_z2
74
75 ; setup loop
76 .MVK 128, cnt
77
78 MIX_LOOP_GEN: .trip 128, 128 ; loop with min 128 iter and max 128 iter
79
80 ; load input values
81 .LDH *in_s++, re_in
82
83 ; feedback of oszi
84 .MPY re_a1a, re_z1, re_z0, z1*a1
85 .SHR re_z0, 15, re_z0 ; back to Q15
86 .ADD re_z0, re_z2, re_z0, (z1*a1)*z2
87 .NEG re_z0, re_z0, / -(z1*a1)*z2
88
89 ; forward of oszi
90 .MPY re_z0, re_b0, re_prodl
91
92 .MPY re_z1, re_b1, re_prod2
93 .MPY re_z1, im_b1, im_prod2
94
95 .ADD re_prodl, re_prod2, re_prod2
96
97 .SHR re_prod2, 15, re_prod2
98 .SHR im_prod2, 15, im_prod2
99
100 ; shift delays
101 .MV re_z1, re_z2
102 .MV re_z0, re_z1
103
104 ; mixing
105 .MPY im_prod2, re_in, im_in
106 .MPY re_prod2, re_in, re_in
107 .SHR im_in, 15, im_in
108 .SHR re_in, 15, re_in
109
110 ; store output values and set Pointer to next value
111 .STH re_in, *re_b++
112 .STH im_in, *im_b++
113
114 ; increment counter and branch till loop cnt=0
115 [cnt] SUB cnt, 1, cnt
116 [cnt] B MIX_LOOP_GEN
117
118 ; store delay states
119 .MVKL _gen_z, ptr_re
120 .MVKH _gen_z, ptr_re
121
122 .STH re_z1, **ptr_re
123 .STH re_z2, **ptr_re
124
125 ;=====
126 ; END mixing
127 ;=====
128 ;=====
129 ; BEGIN low-pass filter
130 ;=====
131 ;=====
132 ;=====
133 ; cascade 0 (first order)
134 ;=====
135 ;=====
136
137 ; copy address of input buffers
138 .reg re_b_s, re_b
139 .reg im_b_s, im_b
140
141 ; load filter coefficients
142 .MVKL _mix_lp_a, ptr_re
143 .MVKL _mix_lp_b, ptr_im
144 .MVKH _mix_lp_a, ptr_re
145 .MVKH _mix_lp_b, ptr_im
146
147 .LDH *ptr_re++, re_a1a

```

```

148 LDH *ptr_im++, re_b_0
149 LDH *ptr_re++, re_a_1b
150 LDH *ptr_im++, re_b_1
151
152 ; copy to have them for both paths
153 MV re_a_1a , im_a_1a
154 MV re_b_0 , im_b_0
155 MV re_a_1b , im_a_1b
156 MV re_b_1 , im_b_1
157
158 ; load delay states
159 MVKL _mix_lp_z_re , ptr_re
160 MVKL _mix_lp_z_im , ptr_im
161 MVKH _mix_lp_z_re , ptr_re
162 MVKH _mix_lp_z_im , ptr_im
163
164 LDH **ptr_re, re_z_1
165 LDH **ptr_im, im_z_1
166
167 ; setup the loop
168 MVK 128 , cnt
169
170 MIX_LOOP_C0: .trip 128 , 128 ; loop with min 128 iter and max 128 iter
171
172 ; load input values
173 LDH *re_b , re_in
174 LDH *im_b , im_in
175
176 ; feedback of filter
177 MPY re_z_1 , re_a_1a , re_prod1 ; z1 * a1a
178 MPY im_z_1 , im_a_1a , im_prod1
179
180 MPY re_z_1 , re_a_1b , re_z_0 ; z1 * a1b
181 MPY im_z_1 , im_a_1b , im_z_0
182
183 ADD re_z_0 , re_prod1 , re_z_0 ; z1*a1a+z1*a1b
184 ADD im_z_0 , im_prod1 , im_z_0
185
186 SHR re_z_0 , 15 , re_z_0 ; back to Q15
187 SHR im_z_0 , 15 , im_z_0
188
189 SUB re_in , re_z_0 , re_z_0 ; in-(z1*a1a+z1*a1b)
190 SUB im_in , im_z_0 , im_z_0
191
192 ; forward of filter
193 MPY re_z_0 , re_b_0 , re_prod1 ; z0*b0
194 MPY im_z_0 , im_b_0 , im_prod1
195
196 MPY re_z_1 , re_b_1 , re_in ; z1*b1
197 MPY im_z_1 , im_b_1 , im_in
198
199 ADD re_in , re_prod1 , re_in ; z0*b0+z1*b1
200 ADD im_in , im_prod1 , im_in
201
202 SHR re_in , 15 , re_in ; back to Q15
203 SHR im_in , 15 , im_in
204
205 ; shift delays
206 MV re_z_0 , re_z_1
207 MV im_z_0 , im_z_1
208
209 ; store output values and set Pointer to next value
210 STH re_in , *re_b++
211 STH im_in , *im_b++
212
213 ; increment counter and branch till loop cnt=0
214 {cnt} SUB cnt, 1, cnt
215 {cnt} B MIX_LOOP_C0
216
217 ; store delay states
218 MVKL _mix_lp_z_re , ptr_re
219 MVKL _mix_lp_z_im , ptr_im
220 MVKH _mix_lp_z_re , ptr_re
221 MVKH _mix_lp_z_im , ptr_im
222
223 STH re_z_1 , **ptr_re
224 STH im_z_1 , **ptr_im
225
226 ;-----
227 ; cascade 1 (second order)

```

```

228 -----
229 ; copy address of input buffers
230 MV re_b_s , re_b
231 MV im_b_s , im_b
232
233 ; load coefficients
234 MVKL _mix_lp_a , ptr_re
235 MVKL _mix_lp_b , ptr_im
236 MVKH _mix_lp_a , ptr_re
237 MVKH _mix_lp_b , ptr_im
238
239 ADD 6 , ptr_re , ptr_re
240 ADD 6 , ptr_im , ptr_im
241
242 LDH *ptr_re++, re_a_1a
243 LDH *ptr_im++, re_b_0
244 LDH *ptr_re++, re_a_1b
245 LDH *ptr_im++, re_b_1
246 LDH *ptr_re++, re_a_2
247 LDH *ptr_im++, re_b_2
248 LDH *ptr_re++, re_b_2
249
250 MV re_a_1a , im_a_1a
251 MV re_b_0 , im_b_0
252 MV re_a_1b , im_a_1b
253 MV re_b_1 , im_b_1
254 MV re_a_2 , im_a_2
255 MV re_b_2 , im_b_2
256
257 ; load delay states
258 MVKL _mix_lp_z_re , ptr_re
259 MVKL _mix_lp_z_im , ptr_im
260 MVKH _mix_lp_z_re , ptr_re
261 MVKH _mix_lp_z_im , ptr_im
262
263 ADD 6 , ptr_re , ptr_re ; redirect pointer to coeff
264 ADD 6 , ptr_im , ptr_im
265
266 LDH **ptr_re, re_z_1
267 LDH **ptr_im, im_z_1
268
269 LDH **ptr_re, re_z_2
270 LDH **ptr_im, im_z_2
271
272 ; setup loop
273 MVK 128 , cnt
274
275
276 MIX_LOOP_C1: .trip 128 , 128 ; loop with min 128 iter and max 128 iter
277
278 ; load input values
279 LDH *re_b , re_in
280 LDH *im_b , im_in
281
282 ; feedback of filter
283 MPY re_z_1 , re_a_1a , re_prod1 ; z1 * a1a
284 MPY im_z_1 , im_a_1a , im_prod1
285
286 MPY re_z_1 , re_a_1b , re_prod2 ; z1 * a1b
287 MPY im_z_1 , im_a_1b , im_prod2
288
289 MPY re_z_2 , re_a_2 , re_z_0 ; z2 * a2
290 MPY im_z_2 , im_a_2 , im_z_0
291
292 ADD re_z_0 , re_prod1 , re_z_0 ; z1*a1a + z2*a2
293 ADD im_z_0 , im_prod1 , im_z_0
294
295 ADD re_z_0 , re_prod2 , re_z_0 ; z1*a1a + z2*a1b + z2*a2
296 ADD im_z_0 , im_prod2 , im_z_0
297
298 SHR re_z_0 , 15 , re_z_0 ; back to Q15
299 SHR im_z_0 , 15 , im_z_0
300
301 SUB re_in , re_z_0 , re_z_0 ; in-(z1*a1a + z1*a1b + z2*a2)
302 SUB im_in , im_z_0 , im_z_0
303
304 ; forward of filter
305 MPY re_z_0 , re_b_0 , re_prod1 ; z0*b0
306 MPY im_z_0 , im_b_0 , im_prod1
307

```

```

308      MPY      re_z_1, re_b_1, re_prod2 ; z1*b1
309      MPY      im_z_1, im_b_1, im_prod2
310
311      MPY      re_z_2, re_b_2, re_in ; z2*b2
312      MPY      im_z_2, im_b_2, im_in
313
314      ADD      re_in, re_prod1, re_in ; z0*b0+z2*b2
315      ADD      im_in, im_prod1, im_in
316
317      ADD      re_in, re_prod2, re_in ; z0*b0+z1*b1+z2*b2
318      ADD      im_in, im_prod2, im_in
319
320      SHR      re_in, 15, re_in ; back to Q15
321      SHR      im_in, 15, im_in
322
323      ; shift delays
324      MV      re_z_1, re_z_2
325      MV      im_z_1, im_z_2
326
327      MV      re_z_0, re_z_1
328      MV      im_z_0, im_z_1
329
330      ; store output values and set Pointer to next value
331      STH      re_in, *re_b++
332      STH      im_in, *im_b++
333
334      ; increment counter and branch till loop cnt=0
335      [cnt] SUB      cnt, 1, cnt
336      [cnt] B      MIX_LOOP_C1
337
338      ; store delay states
339      MVKL     mix_lp_z_re, ptr_re
340      MVKL     mix_lp_z_im, ptr_im
341
342      MVKH     mix_lp_z_re, ptr_re
343      MVKH     mix_lp_z_im, ptr_im
344
345      ADD      6, ptr_re, ptr_re
346      ADD      6, ptr_im, ptr_im
347
348      STH      re_z_1, **ptr_re
349      STH      re_z_2, **ptr_re
350
351      STH      im_z_1, **ptr_im
352      STH      im_z_2, **ptr_im
353
354      ; cascade 2 (second order)
355      -----
356      -----
357      ; copy address of input buffers
358      MV      re_b_s, re_b
359      MV      im_b_s, im_b
360
361      ; load coefficients
362      MVKL     mix_lp_a, ptr_re
363      MVKL     mix_lp_b, ptr_im
364
365      MVKH     mix_lp_a, ptr_re
366      MVKH     mix_lp_b, ptr_im
367
368      ADD      6, ptr_re, ptr_re
369      ADD      6, ptr_im, ptr_im
370
371      ADD      6, ptr_re, ptr_re
372      ADD      6, ptr_im, ptr_im
373
374      LDH      *ptr_re++, re_a_1_a
375      LDH      *ptr_im++, re_b_0
376      LDH      *ptr_re++, re_a_1_b
377      LDH      *ptr_im++, re_b_1
378      LDH      *ptr_re++, re_a_2
379      LDH      *ptr_im++, re_b_2
380
381      MV      re_a_1_a, im_a_1_a
382      MV      re_b_0, im_b_0
383      MV      re_a_1_b, im_a_1_b
384      MV      re_b_1, im_b_1
385      MV      re_a_2, im_a_2
386      MV      re_b_2, im_b_2
387
388      ; load delay states
389      MVKL     mix_lp_z_re, ptr_re
390      MVKL     mix_lp_z_im, ptr_im
391
392      MVKH     mix_lp_z_re, ptr_re
393      MVKH     mix_lp_z_im, ptr_im
394
395      ADD      6, ptr_re, ptr_re
396      ADD      6, ptr_im, ptr_im ; add 12
397
398      ADD      6, ptr_re, ptr_re
399      ADD      6, ptr_im, ptr_im ; add 12
400
401      LDH      **ptr_re, re_z_1
402      LDH      **ptr_im, im_z_1
403
404      LDH      **ptr_re, re_z_2
405      LDH      **ptr_im, im_z_2
406
407      ; setup loop
408      MV      128, cnt
409
410      MIX_LOOP_C2: .trip 128, 128 ; loop with min 128 iter and max 128 iter
411
412      ; load input values
413      LDH      *re_b, re_in
414      LDH      *im_b++, im_in
415
416      ; feedback of filter
417      MPY      re_z_1, re_a_1_a, re_prod1, z1 * a1a
418      MPY      im_z_1, im_a_1_a, im_prod1
419
420      MPY      re_z_1, re_a_1_b, re_prod2, z1 * alb
421      MPY      im_z_1, im_a_1_b, im_prod2
422
423      MPY      re_z_2, re_a_2, re_z_0, z2 * a2
424      MPY      im_z_2, im_a_2, im_z_0
425
426      ADD      re_z_0, re_prod1, re_z_0, z1*a1a + z2*a2
427      ADD      im_z_0, im_prod1, im_z_0
428
429      ADD      re_z_0, re_prod2, re_z_0, z1*a1a + z1*alb + z2*a2
430      ADD      im_z_0, im_prod2, im_z_0
431
432      SHR      re_z_0, 15, re_z_0 ; back to Q15
433      SHR      im_z_0, 15, im_z_0
434
435      SUB      re_in, re_z_0, re_z_0, in-(z1*a1a + z1*alb + z2*a2)
436      SUB      im_in, im_z_0, im_z_0
437
438      ; forward of filter
439      AND      0x7, re_b, ptr_re ; do forward only every 4th time
440      AND      0x2, re_b, re_b, redirect pointer
441
442      [ptr_re] MPY      re_z_0, re_b_0, re_prod1 ; z0*b0
443      [ptr_re] MPY      im_z_0, im_b_0, im_prod1
444
445      [ptr_re] MPY      re_z_1, re_b_1, re_prod2 ; z1*b1
446      [ptr_re] MPY      im_z_1, im_b_1, im_prod2
447
448      [ptr_re] MPY      re_z_2, re_b_2, re_in ; z2*b2
449      [ptr_re] MPY      im_z_2, im_b_2, im_in
450
451      [ptr_re] ADD      re_in, re_prod1, re_in ; z0*b0+z2*b2
452      [ptr_re] ADD      im_in, im_prod1, im_in
453
454      [ptr_re] ADD      re_in, re_prod2, re_in ; z0*b0+z1*b1+z2*b2
455      [ptr_re] ADD      im_in, im_prod2, im_in
456
457      [ptr_re] SHR      re_in, 15, re_in ; back to Q15
458      [ptr_re] SHR      im_in, 15, im_in
459
460      ; store output
461      [ptr_re] STH      re_in, *re_b_s++
462      [ptr_re] STH      im_in, *im_b_s++
463
464      ; shift delays
465      MV      re_z_1, re_z_2
466      MV      im_z_1, im_z_2
467

```



```

468 MW re_z_0 , re_z_1
469 MW im_z_0 , im_z_1
470
471 ; increment counter and branch till loop cnt=0
472 [cnt] SUB cnt , 1, cnt
473 [cnt] B MIX_LOOP_C2
474
475 ; store delay states
476 MYKL mix_lp_z_re , ptr_re
477 MYKL mix_lp_z_im , ptr_im
478 MYKH mix_lp_z_re , ptr_re
479 MYKH mix_lp_z_im , ptr_im
480
481 ADD 6 , ptr_re , ptr_re
482 ADD 6 , ptr_im , ptr_im
483
484 ADD 6 , ptr_re , ptr_re
485 ADD 6 , ptr_im , ptr_im
486
487 STH re_z_1 , ++ptr_re
488 STH re_z_2 , ++ptr_re
489
490 STH im_z_1 , ++ptr_im
491 STH im_z_2 , ++ptr_im
492
493 ; =====
494 ; END low-pass filter
495 ; =====
496
497 .endproc ;quad_mix

```

## F.3. MATLAB Listings

### Table Generators

Listing F.48: *tanTabGen.m*

```

1 % Arc Tangent Table Generator - Version 0.5
2 %
3 % file : tanTabGen.m
4 % date : oct 2001 - Jan 2002
5 % by : c. haller
6 % f. schnyder
7 %
8
9 %input parameters
10 name='atan'; % name of the output files
11 Nbit=7; % lenght of table = 2^Nbit
12 scale=0.8; % scale factor included in table (0.8)
13 Mbit=2; % bits of max input -> table from 0 to 2^bit
14 inbits=14; % Q format of input
15
16 % calculate rest of parameters
17 N=2^Nbit;
18 max=2^Mbit;
19 shift=inbits-(Nbit-Mbit);
20 shiftval=2^shift;
21
22 % calculation of table values
23 v=inspace(0,max,N+1);
24 values=atan(v)/scale;
25
26 % scaling to Q15
27 values=round(values*(2^15));
28 values=values(1:N);
29
30 % avoid overflows (saturation)
31 for i=1:N
32 if values(i)>(2^15-1)
33 values(i)=2^15-1;
34 end
35 end

```

```

36 % generate .c and .h file
37
38
39 % .c file
40 outfile=fopen(strcat(name,'.c'),'w');
41 fprintf(outfile,'%n');
42 fprintf(outfile,' ARCTANGENT FUNCTION TABLE - VERSION FIXED POINT 0.5 \n');
43 fprintf(outfile,' generated with tanTabGen.m\n');
44 fprintf(outfile,' N = %i\n',N);
45 fprintf(outfile,' Range = %i\n',max);
46 fprintf(outfile,' Scale = %f\n',scale);
47 fprintf(outfile,' \n');
48 fprintf(outfile,' file : %s\n',strcat(name,'.c'));
49 fprintf(outfile,' date : oct 2001 - Jan 2002\n');
50 fprintf(outfile,' by : c. haller\n');
51 fprintf(outfile,' f. schnyder\n');
52 fprintf(outfile,' \n');
53 fprintf(outfile,'*/\n');
54 fprintf(outfile,' \n');
55 fprintf(outfile,' #include "%s"\n',strcat(name,'.h'));
56 fprintf(outfile,' \n');
57 fprintf(outfile,' short atan_buffer[]={\n');
58 fprintf(outfile,' %i,\n',values);
59 fprintf(outfile,' \n');
60 fprintf(outfile,' \n');
61 fclose(outfile);
62
63 % .h file
64 outfile=fopen(strcat(name,'.h'),'w');
65 fprintf(outfile,'%n');
66 fprintf(outfile,' ARCTANGENT FUNCTION TABLE - VERSION FIXED POINT 0.5 \n');
67 fprintf(outfile,' generated with tanTabGen.m\n');
68 fprintf(outfile,' N = %i\n',N);
69 fprintf(outfile,' Range = %i\n',max);
70 fprintf(outfile,' Scale = %f\n',scale);
71 fprintf(outfile,' \n');
72 fprintf(outfile,' file : %s\n',strcat(name,'.h'));
73 fprintf(outfile,' date : oct 2001 - Jan 2002\n');
74 fprintf(outfile,' by : c. haller\n');
75 fprintf(outfile,' f. schnyder\n');
76 fprintf(outfile,' \n');
77 fprintf(outfile,'*/\n');
78 fprintf(outfile,' \n');
79 fprintf(outfile,' #ifndef _%s_\n',upper(name));
80 fprintf(outfile,' #define _%s_\n',upper(name));
81 fprintf(outfile,' \n');
82 fprintf(outfile,' #define BUFFERSIZE ATAN %i\n',N);
83 fprintf(outfile,' #define ATAN_SHIFT %i\n',shift);
84 fprintf(outfile,' #define ATAN_INTERPOL %i\n',shiftval);
85 fprintf(outfile,' \n');
86 fprintf(outfile,' extern short atan_buffer[];\n');
87 fprintf(outfile,' \n');
88 fprintf(outfile,' #endif /* _%s_ */\n',upper(name));
89 fprintf(outfile,' \n');
90
91 fclose(outfile);
92
93 % compare polt
94 testv=0:(2*(inbits-Mbit));
95 testv=testv./length(testv)-1;
96 indexv=floor(testv./(2^shift));
97 curv=testv-((2^shift)*indexv); % modula
98 direct=testv;
99 for i=length(testv)
100 direct(i)=values(indexv(i)+1);
101 end
102 interpol=testv;
103 for i=length(testv)
104 if indexv(i)<2^Nbit-1
105 interpol(i)=(values(indexv(i)+2)*curv(i)+values(indexv(i)+1))*((2^shift)-curv(i)
106 ))/(2^shift);
107 else
108 interpol(i)=values(indexv(i)+1);
109 end
110 testv=(testv./(2^inbits));

```

```

111 plot(testv,direct./2^15,testv,interpol./2^15,testv,atan(testv)*scale);
112 legend('Direct Lookup','Interpolation Lookup','Atan')

```

### Listing F49: tanTabGenInterp.m

```

1 % Arc Tangent Table With Interpolation Generator - Version 0.5
2 %
3 % file : tanTabGenInterp.m
4 % date : oct 2001 - jan 2002
5 % by : c. haller
6 %
7 % f. schnyder
8 %

```

```

9 %input parameters
10 name='atan'; % name of the output files
11 Nbit=4; % length of table = 2^Nbit
12 scale=0.8; % scale factor included in table
13 Mbit=2; % bits of max input -> table from 0 to 2^bit
14 inbits=14; % Q format of input
15
16 % calculate rest of parameters
17 N=2^Nbit;
18 max=2^Mbit;
19 shift=inbits-(Nbit-Mbit);
20 shiftval=2^shift;
21
22 % calculation of values
23 v=linspace(0,max,N+1);
24 values=atan(v)*scale;
25
26 % scaling to Q15
27 values=round(values*(2^15));
28 values=values(1:N);
29
30 % avoid overflows (saturation)
31 for i=1:N
32 if values(i)>(2^15-1)
33 values(i)=2^15-1;
34 end
35
36 gradient=values(2:N)-values(1:N-1);
37 gradient(N)=0;
38
39 % generate .c and .h file
40
41 % .c file
42 outfile=fopen(strcat(name,'.c'),'w');
43 fprintf(outfile,'%*
\n');
44 fprintf(outfile,' ARCTANGENT FUNCTION TABLE WITH INTERPOLATION \n');
45 fprintf(outfile,' - VERSION FIXED POINT 0.5 \n');
46 fprintf(outfile,' generated with tanTabGen.m\n');
47 fprintf(outfile,' N = %i\n',N);
48 fprintf(outfile,' Range = %i\n',max);
49 fprintf(outfile,' Scale = %f\n',scale);
50 fprintf(outfile,' \n');
51 fprintf(outfile,' file : %s\n',strcat(name,'.c'));
52 fprintf(outfile,' date : oct 2001 - jan 2002\n');
53 fprintf(outfile,' by : c. haller\n');
54 fprintf(outfile,' f. schnyder\n');
55 fprintf(outfile,'
\n');
56 fprintf(outfile,'%*\n');
57 fprintf(outfile,' \n');
58 fprintf(outfile,' #include "<math>\pi</math>"\n',strcat(name,'.h'));
59 fprintf(outfile,' \n');
60 fprintf(outfile,' short atan_buffer[%i]=\n',N);
61 fprintf(outfile,' %i ,\n',values);
62 fprintf(outfile,' );\n');
63 fprintf(outfile,' \n');
64 fprintf(outfile,' short atan_grad_buffer[%i]=\n',N);
65 fprintf(outfile,' %i ,\n',gradient);
66 fprintf(outfile,' );\n');
67 fprintf(outfile,' \n');
68 fclose(outfile);
69
70 % .h file
71 outfile=fopen(strcat(name,'.h'),'w');

```

```

72 fprintf(outfile,'%*
\n');
73 fprintf(outfile,' ARCTANGENT FUNCTION TABLE WITH INTERPOLATION \n');
74 fprintf(outfile,' - VERSION FIXED POINT 0.5 \n');
75 fprintf(outfile,' generated with tanTabGen.m\n');
76 fprintf(outfile,' N = %i\n',N);
77 fprintf(outfile,' Range = %i\n',max);
78 fprintf(outfile,' Scale = %f\n',scale);
79 fprintf(outfile,' \n');
80 fprintf(outfile,' file : %s\n',strcat(name,'.h'));
81 fprintf(outfile,' date : oct 2001 - jan 2002\n');
82 fprintf(outfile,' by : c. haller\n');
83 fprintf(outfile,' f. schnyder\n');
84 fprintf(outfile,'
\n');
85 fprintf(outfile,'%*\n');
86 fprintf(outfile,' \n');
87 fprintf(outfile,' #ifndef _%s_upper\n',upper(name));
88 fprintf(outfile,' #define _%s_upper\n',upper(name));
89 fprintf(outfile,' \n');
90 fprintf(outfile,' #define BUFFERSIZE ATAN %i\n',N);
91 fprintf(outfile,' #define ATAN_SHIFT %i\n',shift);
92 fprintf(outfile,' #define ATAN_INTERPOL %i\n',shiftval);
93 fprintf(outfile,' \n');
94 fprintf(outfile,' extern short atan_buffer[%i]\n',N);
95 fprintf(outfile,' \n');
96 fprintf(outfile,' extern short atan_grad_buffer[%i]\n',N);
97 fprintf(outfile,' \n');
98 fprintf(outfile,' #endif /* _%s_upper\n',upper(name));
99 fprintf(outfile,' \n');
100 fclose(outfile);
101
102 % compaire polt
103 testv=0:(2^(inbits-Mbit));
104 testv=testv(1:length(testv)-1);
105 indexv=floor(testv./(2^shift));
106 curv=testv-((2^shift)*indexv); % modula
107 direct=testv;
108 for i=1:length(testv)
109 direct(i)=values(indexv(i)+1);
110 end
111 interpol=testv;
112 for i=1:length(testv)
113 if indexv(i)<2^Nbit-1
114 interpol(i)=values(indexv(i)+1)+gradient(indexv(i)+1)*curv(i)/(2^shift);
115 else
116 interpol(i)=values(indexv(i)+1);
117 end
118 end
119 testv=testv./(2^inbits);
120 plot(testv,direct./2^15,testv,interpol./2^15,testv,atan(testv)*scale);
121 legend('Direct Lookup','Interpolation Lookup','Atan')

```

### Listing F50: SineTabGen.m

```

1 % Sine Table Generator - Version 1
2 %
3 % file : SineTabGen.m
4 % date : oct 2001 - jan 2002
5 % by : c. haller
6 % f. schnyder
7 %
8
9 % name of the output files
10 name='sine';
11
12 % file input: bitnumber of buffer length (4, 5, 6, 7)
13 n=7;
14
15 % calculation of constants
16 len=2^n; %table length
17 qformat=15-log2(4*len); %size of needed Q format
18 format=2^qformat; %format to calculate step_inverse
19 reduction=12-qformat; %change from Q3.12 to new Q format
20 step=(pi/2)/(len-1); %step between the arguments
21 step_inverse=round((1/step)*format); %step_inverse
22
23 % calculation of values
24 arg=linspace(0,pi/2,len);

```

```

25 values=sin(arg);
26
27 % scaling to Q15
28 values=round(values*((2^15)-1)); %conversion from float to q15
29
30
31 % generate .c and .h file
32 outfile=fopen(strcat(name,'.c'),'w');
33 fprintf(outfile,'%s\n',
34     '\n');
35 fprintf(outfile,' SINE FUNCTION TABLE \n');
36 fprintf(outfile,' generated with SineTabGen.m\n');
37 fprintf(outfile,' \n');
38 fprintf(outfile,' file : %s\n',strcat(name,'.c'));
39 fprintf(outfile,' date : oct 2001 - jan 2002\n');
40 fprintf(outfile,' by : c. haller\n');
41 fprintf(outfile,' f. schnyder\n');
42 fprintf(outfile,' \n');
43 fprintf(outfile,'**/\n');
44 fprintf(outfile,' \n');
45 fprintf(outfile,' #include "%s\n',strcat(name,'.h'));
46 fprintf(outfile,' \n');
47 fprintf(outfile,' short sine_buffer[%i\n',values);
48 fprintf(outfile,' %i\n',values);
49 fprintf(outfile,' \n');
50 fprintf(outfile,' \n');
51
52 fclose(outfile);
53
54
55 outfile=fopen(strcat(name,'.h'),'w');
56 fprintf(outfile,'%s\n',
57     '\n');
58 fprintf(outfile,' SINE FUNCTION TABLE \n');
59 fprintf(outfile,' generated with SineTabGen.m\n');
60 fprintf(outfile,' \n');
61 fprintf(outfile,' file : %s\n',strcat(name,'.h'));
62 fprintf(outfile,' date : oct 2001 - jan 2002\n');
63 fprintf(outfile,' by : c. haller\n');
64 fprintf(outfile,' f. schnyder\n');
65 fprintf(outfile,' \n');
66 fprintf(outfile,'**/\n');
67 fprintf(outfile,' \n');
68 fprintf(outfile,' #ifndef %s\n',upper(name));
69 fprintf(outfile,' #define %s\n',upper(name));
70 fprintf(outfile,' \n');
71 fprintf(outfile,' #define BUFFERSIZE SINE %i\n',len);
72 fprintf(outfile,' #define STEP_INVERSE %i\n',step_inverse);
73 fprintf(outfile,' #define Q_FORMAT %i\n',q_format);
74 fprintf(outfile,' #define FORMAT %i\n',format);
75 fprintf(outfile,' #define REDUCTION %i\n',reduction);
76 fprintf(outfile,' \n');
77 fprintf(outfile,' extern short sine_buffer[%i\n');
78 fprintf(outfile,' \n');
79 fprintf(outfile,' #endif /* %s */\n',upper(name));
80 fprintf(outfile,' \n');
81 fclose(outfile);
82
83
84 plot(values./2^15)

```

Listing F51: SineTabGenInt.m

```

1 % Sine Table With Interpolation Generator - Version 1
2 %
3 % file : SineTabGenInt.m
4 % date : oct 2001 - jan 2002
5 % by : c. haller
6 % f. schnyder
7 %
8
9 % name of the output files
10 name='sine';
11
12 % file input: bitnumber of buffer lenght (4, 5, 6, 7)

```

```

13 n=4;
14
15 % calculation of constants
16 len=2^n;
17 q_format=15-log2(4*len);
18 format=2^q_format;
19 reduction=12-q_format;
20 step=(pi/2)/(len-1);
21 step_inverse=round((1/step)*format); %step_inverse
22
23 % calculation of sine table
24 arg=linspace(0,(pi/2),len);
25 values=sin(arg);
26
27 % scaling to Q15
28 values=round(values*((2^15)-1)); %conversion from float to q15
29
30 % calculation of sine gradient table
31 gradient=values(2:len)-values(1:len-1);
32 gradient(len)=0;
33
34 % generate .c and .h file
35 outfile=fopen(strcat(name,'.c'),'w');
36 fprintf(outfile,'%s\n',
37     '\n');
38 fprintf(outfile,' SINE FUNCTION TABLE \n');
39 fprintf(outfile,' generated with SineTabGenInt.m\n');
40 fprintf(outfile,' \n');
41 fprintf(outfile,' file : %s\n',strcat(name,'.c'));
42 fprintf(outfile,' date : oct 2001 - jan 2002\n');
43 fprintf(outfile,' by : c. haller\n');
44 fprintf(outfile,' f. schnyder\n');
45 fprintf(outfile,' \n');
46 fprintf(outfile,'**/\n');
47 fprintf(outfile,' \n');
48 fprintf(outfile,' #include "%s\n',strcat(name,'.h'));
49 fprintf(outfile,' \n');
50 fprintf(outfile,' short sine_buffer[%i\n',values);
51 fprintf(outfile,' %i\n',values);
52 fprintf(outfile,' \n');
53 fprintf(outfile,' \n');
54 fprintf(outfile,' short sine_grad_buffer[%i\n',values);
55 fprintf(outfile,' %i\n',values);
56 fprintf(outfile,' \n');
57 fprintf(outfile,' \n');
58
59 fclose(outfile);
60
61
62 outfile=fopen(strcat(name,'.h'),'w');
63 fprintf(outfile,'%s\n',
64     '\n');
65 fprintf(outfile,' SINE FUNCTION TABLE \n');
66 fprintf(outfile,' generated with SineTabGenInt.m\n');
67 fprintf(outfile,' \n');
68 fprintf(outfile,' file : %s\n',strcat(name,'.h'));
69 fprintf(outfile,' date : oct 2001 - jan 2002\n');
70 fprintf(outfile,' by : c. haller\n');
71 fprintf(outfile,' f. schnyder\n');
72 fprintf(outfile,' \n');
73 fprintf(outfile,'**/\n');
74 fprintf(outfile,' \n');
75 fprintf(outfile,' #ifndef %s\n',upper(name));
76 fprintf(outfile,' #define %s\n',upper(name));
77 fprintf(outfile,' \n');
78 fprintf(outfile,' #define BUFFERSIZE SINE %i\n',len);
79 fprintf(outfile,' #define STEP_INVERSE %i\n',step_inverse);
80 fprintf(outfile,' #define Q_FORMAT %i\n',q_format);
81 fprintf(outfile,' #define FORMAT %i\n',format);
82 fprintf(outfile,' #define REDUCTION %i\n',reduction);
83 fprintf(outfile,' \n');
84 fprintf(outfile,' extern short sine_buffer[%i\n');
85 fprintf(outfile,' \n');
86 fprintf(outfile,' extern short sine_grad_buffer[%i\n');
87 fprintf(outfile,' \n');
88 fprintf(outfile,' #endif /* %s */\n',upper(name));

```

```

89 fprintf(outfile,'\n');
90 fclose(outfile);
91
92
93 plot(values./2^15)

```

## Filter Coefficients

Listing F.52: mixer\_design.m

```

1 % Mixer Design - Version 0.5
2 %
3 % file : mixer_design.m
4 % date : oct 2001 - Jan 2002
5 % by : c. haller
6 % f. schnyder
7 %
8
9 format long
10
11 fa=150e6/(8*293); % pass-band edge frequency
12 fp=6300; % pass-band edge frequency
13 fs=19000; % stop-band edge frequency
14 dp=2; % stop-band ripple
15 ds=60; % stop-band ripple
16 fc=10.7e6; % carrier frequency
17
18
19 % the sine and cosine oscillator
20 sub=floor(fc/fa);
21 foszi=fc-sub*fa;
22
23 oszi_al=-2*cos(2*pi*foszi*(1/fa));
24 oszi_bl_cos=-cos(2*pi*foszi*(1/fa));
25 oszi_bl_sin=sin(2*pi*foszi*(1/fa));
26 % to Q15
27 ioszi_al=round(oszi_al*2^15);
28 ioszi_bl_cos=round(oszi_bl_cos*2^15);
29 ioszi_bl_sin=round(oszi_bl_sin*2^15);
30
31 % the filter
32 [n,Wn]=buttord(fp*2/fa, fs*2/fa, dp, ds); % determine the order
33
34 [b,a]=butter(n,Wn); % determine direct coefficients
35 freqz(b,a,256,fa) % and plot the filter
36
37 [z,p,k]=butter(n,Wn); % determine second order coefficients
38 [coeff,g]=zp2sos(z,p,k,'up','inf');
39
40 % to Q15
41 icoeff=coeff;
42 Ns=size(coeff,1);
43 for i=1:Ns
44     icoeff(i,4)=coeff(i,5)/2;
45     icoeff(i,5)=coeff(i,5)/2;
46     icoeff(i,:)=round(icoeff(i,:)*2^15);
47 end
48
49 % write data output
50
51 fprintf('/*-----\n');
52 fprintf(' coefficients of sine and cosine oscillator \n');
53 fprintf('-----\n');
54 fprintf('/* Frequency : %10.2f Hz*/ \n',foszi);
55 fprintf('\n');
56 fprintf(' short gen_bl_cos = %i; /*cos*/ \n',ioszi_bl_cos);
57 fprintf(' short gen_bl_sin = %i; /*sin*/ \n',ioszi_bl_sin);
58 fprintf(' short gen_al = %i; /*sin/cos*/ \n',ioszi_al);
59 fprintf('\n');
60 fprintf('/* delay variables */ \n');
61 fprintf(' short gen_z [3] = {0,0,26213}; /* 0.8 for first value of delta function*/ \n');
62 fprintf('\n');
63
64

```

```

65 fprintf('/*-----\n');
66 fprintf(' coefficients of mixer low-pass (mix_lp) \n');
67 fprintf('-----\n');
68 fprintf('/* Order : %2i */ \n',n);
69 fprintf('/* Pass-band : %21 dB @ %6.2f Hz */ \n',dp,fp);
70 fprintf('/* Stop-band : %21 dB @ %6.2f Hz */ \n',ds,fs);
71 fprintf('\n');
72 fprintf('define MIX_LP_CASCADES %i, Ns);
73 fprintf('\n');
74 fprintf(' short mix_lp_a[](3)={ /* a_i_1a_i_1b_i_2 */ \n';
75 fprintf('      { %6i , %6i , %6i }, \n', icoeff(:,4:6));
76 fprintf('      }; \n';
77 fprintf('\n');
78 fprintf(' short mix_lp_b[](3)={ /* b_i_0 b_i_1 b_i_2 */ \n';
79 fprintf('      { %6i , %6i , %6i }, \n', icoeff(:,1:3));
80 fprintf('      }; \n';
81 fprintf('\n');
82 fprintf(' short mix_lp_scale= %i; \n/* %f to avoid overflows */ \n',round(g*2^15),g);
83
84 fprintf('/* delay variables */ \n';
85 fprintf(' short mix_lp_z_re[](3)={ \n';
86 for i=1:Ns
87     fprintf('      { 0,0,0}, \n', icoeff(:,5:6));
88 end
89 fprintf('      }; \n';
90 fprintf(' short mix_lp_z_im[](3)={ \n';
91 for i=1:Ns
92     fprintf('      { 0,0,0}, \n', icoeff(:,5:6));
93 end
94 fprintf('      }; \n';
95 fprintf('\n');
96
97 % Filter Test for overflows
98 Ns=200;
99 y=zeros(Ns,Ns+2);
100
101 % create input signal
102 t=linspace(0,1/fa*(Ns+1),Ns+2);
103 %x=sin(2*pi*3000*t);
104 %x=chirp(t,100,t(Ns+2),10000);
105 %x=[0 0 (rand(1,Ns)-0.5)*2];
106 %x=[0 1 ones(1,Ns/2) (ones(1,Ns/2)*-1)];
107
108 xsave=x;
109
110 % scaling from the mixer
112 x=x/g*0.8;
113 d=zeros(Ns,Ns+2);
114
115 % do the filtering
116 for n=3:Ns+2
117     for i=1:Ns
118         d(i,n)=(x(n)-coeff(i,5)*d(i,n-1)-coeff(i,6)*d(i,n-2));
119         y(i,n)=(coeff(i,1)*d(i,n)+coeff(i,2)*d(i,n-1)+coeff(i,3)*d(i,n-2));
120         x(n) = y(i,n);
121     end
122 end
123
124 %plot delay states and output values
125
126 figure
127 for i=1:Ns
128     subplot(2,Ns,i);
129     plot(d(i,:), 'k');
130     axis([-10 Ns -1.4 1.4]);
131     legend('Delay Cascade_',int2str(i-1),4);
132
133     subplot(2,Ns,i+Ns);
134     plot(y(i,:), 'k');
135     axis([-10 Ns -1.4 1.4]);
136     legend('Output Cascade_',int2str(i-1),4);
137 end
138
139 %plot(y/Ns,:);
140 hold on
141 plot(xsave,'k');
142 legend('Output Signal' , 'Input Signal',4);
143 hold off

```

Listing F.53: lowpass\_design\_noisefilter.m

```

1 % Low-Pass Filter Design - Version 0.5
2 %
3 % filter : low-pass before dac
4 % file : lowpass_design_noisefilter.m
5 % date : oct 2001 - jan 2002
6 % by : c. haller
7 %
8 %
9
10 fa=15000000/(8*293*4); % sample rate
11 fp=3500; % pass-band edge frequency
12 fs=fa/4; % stop-band edge frequency
13 dp=2; % pass-band ripple
14 ds=20; % stop-band ripple
15
16 [n,Wh]=buttord(fp*2/fa, fs*2/fa, dp, ds); % determine the order
17
18 [b,a]=butter(n,Wh); % determine direct coefficients
19 freqz(b,a,256,fa)
20
21 [z,p,k]=butter(n,Wh); % determine second order coefficients
22 [coeff,g]=zp2sos(z,p,k,'up','int');
23
24 coeffaave=coeff;
25 icoeff=coeff;
26
27 Ns=size(coeff,1);
28
29 for i=1:Ns
30 icoeff(i,:)=round(icoeff(i,:)*2^15);
31 end
32
33 fprintf('/*-----\n');
34 fprintf(' coefficients of noise low-pass (no_lp) after demodulation \n');
35 fprintf(' /* Order : %2i */ \n',n);
36 fprintf(' /* Pass-band : %2i dB @ %6.2f Hz */ \n',dp,fp);
37 fprintf(' /* Stop-band : %2i dB @ %6.2f Hz */ \n',ds,fs);
38 fprintf(' \n');
39 fprintf(' #define NO_LP_CASCADES %i',Ns);
40
41 fprintf(' short no_lp_a[12]={ /* a_i-1 a_i-2 */ \n';
42 fprintf(' { %6i , %6i }, \n', icoeff(:,5:6));
43 fprintf(' } \n';
44
45 fprintf(' short no_lp_b[13]={ /* b_i-0 b_i-1 b_i-2 */ \n';
46 fprintf(' { %6i , %6i , %6i }, \n', icoeff(:,1:3));
47 fprintf(' } \n';
48
49 fprintf(' short no_lp_scale= %i; \t/* %f*0.8 to avoid overflows */ \n',(round((g*0.8)*2^15))/g);
50
51 fprintf(' \n');
52 fprintf(' /* delay variables */ \n';
53 fprintf(' short no_lp_z[13]={ \n';
54 for i=1:Ns
55 fprintf(' ( 0,0,0), \n', icoeff(:,5:6));
56 end
57 fprintf(' } \n';
58
59 % Filter Test
60 Nx=300;
61 y=zeros(Ns,Nx*2);
62
63 %x=[0 0 (rand(1,Nx)-0.5)*2];
64
65 t=linspace(0,(1/fa*(Nx+1)),Nx*2);
66 %x=sin(2*pi*3000*t);
67 %x=chirp(t,100,t/(Nx*2),10000);
68
69 x=[0 1 ones(1,Nx)];
70
71 xave=x;
72
73 x=x*g*0.8;
74
75 d=zeros(Ns,Nx*2);
76
77 for n=3:Nx*2
78 for i=1:Ns

```

```

79 d(i,n)=(x(n)-coeff(i,5)*d(i,n-1)-coeff(i,6)*d(i,n-2));
80 y(i,n)=(coeff(i,1)*d(i,n)+coeff(i,2)*d(i,n-1)+coeff(i,3)*d(i,n-2));
81 x(n) = y(i,n);
82 end
83 end
84
85 figure
86 for i=1:Ns
87 subplot(2,Ns,i);
88 plot(d(i,:));
89 legend('D',int2str(i));
90
91 subplot(2,Ns,i+Ns);
92 plot(y(i,:));
93 legend('Y');
94 end
95
96 figure
97 plot(y(Ns,:));
98 hold on
99 plot(xave,'r');
100 hold off

```

Listing F.54: highpass\_design\_noisefilter.m

```

1 % High-Pass Filter Design - Version 0.5
2 %
3 % filter : high-pass before dac
4 % file : highpass_design_noisefilter.m
5 % date : oct 2001 - jan 2002
6 % by : c. haller
7 %
8 %
9
10 fa=15000000/(8*293*4*2); % sample rate
11 fp=300; % pass-band edge frequency
12 fs=50; % stop-band edge frequency
13 ds=0.1; % pass-band ripple
14 ds=40; % stop-band ripple
15
16 [n,Wh]=buttord(fp*2/fa, fs*2/fa, dp, ds); % determine the order
17
18 [b,a]=butter(n,Wh,'high'); % determine direct coefficients
19
20 freqz(b,a,256,fa)
21
22 [z,p,k]=butter(n,Wh,'high'); % determine second order coefficients
23 [sos,g]=sp2sos(z,p,k);
24 nc=size(sos,1);
25 A=sos(:,4:6);
26 B=sos(:,1:3);
27 iA=A;
28 iB=B;
29
30 for i=1:nc
31 iA(i,1)=iA(i,2)/2;
32 iA(i,2)=iA(i,2)/2;
33 iA(i,:)=round(iA(i,:)*2^15);
34 iB(i,:)=round(iB(i,:)*2^15);
35 end
36
37 fprintf('/*-----\n');
38 fprintf(' coefficients of end high-pass (en_hp) before dac \n');
39 fprintf(' /* Order : %2i */ \n',n);
40 fprintf(' /* Pass-band : %2.1f dB @ %6.2f Hz */ \n',dp,fp);
41 fprintf(' /* Stop-band : %2.1f dB @ %6.2f Hz */ \n',ds,fs);
42 fprintf(' \n');
43 fprintf(' #define EN_HP_CASCADES %i',nc);
44
45 fprintf(' short en_hp_a[12]={ /* aa_i-1a_i-1b a_i-2 */ \n';
46 fprintf(' { %6i , %6i , %6i }, \n', iA(i,:));
47 fprintf(' } \n';
48
49 fprintf(' short en_hp_b[13]={ /* b_i-0 b_i-1 b_i-2 */ \n';
50 fprintf(' { %6i , %6i , %6i }, \n', iB(i,:));
51 fprintf(' } \n';
52
53 fprintf(' short en_hp_c[13]={ /* c_i-0 c_i-1 c_i-2 */ \n';
54 fprintf(' { %6i , %6i , %6i }, \n';

```

```

55 fprintf(' \n');
56 fprintf(' short en_hp_scale= %i; \t/* 0.5*f to avoid overflows */\n', (round(g
    *2*15/2))/g);
57 fprintf(' \n');
58 fprintf(' #define EN_HP_UPSCALE 2\n');
59 fprintf(' \n');
60 fprintf(' /* delay variables */\n');
61 fprintf(' short en_hp_z[3]={ \n';
62 for i=1:nc
63     { 0,0,0 }\n';
64 end
65 fprintf('
66     }\n');
67
68
69 nx=100
70 t=linspace(0,1/1e8*(nx-1),nx);
71 x=sin(2*pi*300*t(1:50));
72 x=x./cos(2*pi*300*t(1:51:nx));
73 x=[0 1 ones(1,nx-2)];
74 %x=[0 1 ones(1,nx-2)];
75
76
77 xsave=x;
78 x=x*g;
79
80 zeta=zeros(nc,nx);
81 kappa=zeros(nc,nx);
82 y=zeros(nc,nx);
83
84 for i=1:length(t)
85     y(c,i)=zeta(c,i)+(B(c,1)*x(i));
86     zeta(c,i+1)=B(c,2)*x(i)-A(c,2)*y(c,i)+kappa(c,i);
87     kappa(c,i+1)=B(c,3)*x(i)-A(c,3)*y(c,i);
88     x(i)=y(c,i);
89 end
90
91 alpha=zeros(nc,nx);
92 beta=zeros(nc,nx);
93
94

```

```

95 gamma=zeros(nc,nx);
96 y2=zeros(nc,nx);
97
98 xsave;
99
100 for i=1:length(t)
101     for c=1:nc
102         alpha(c,i)=-A(c,3)*gamma(c,i)+(-A(c,2)*beta(c,i)+x(i));
103         y2(c,i)=B(c,1)*alpha(c,i)+B(c,2)*beta(c,i)+(B(c,3)*gamma(c,i));
104         gamma(c,i+1)=beta(c,i);
105         beta(c,i+1)=alpha(c,i);
106         x(i)=y(c,i);
107     end
108 end
109
110 labels=int2str(1:nc);
111
112 plot(t,xsave,t,y);
113 legend('in','out1','out2');
114 figure;
115 subplot(2,1,1)
116 plot(t,zeta(:,1:nx));
117 legend('zeta1','zeta2');
118 subplot(2,1,2)
119 plot(t,kappa(:,1:nx));
120 legend('kappa1','kappa2');
121
122 figure
123 plot(t,xsave,t,y);
124 legend('in n','out1','out2');
125 figure;
126 subplot(3,1,1)
127 plot(t,alpha(:,1:nx));
128 legend('alpha1','alpha2');
129 subplot(3,1,2)
130 plot(t,beta(:,1:nx));
131 legend('beta1','beta2');
132 subplot(3,1,3)
133 plot(t,gamma(:,1:nx));
134 legend('gamma1','gamma2');

```