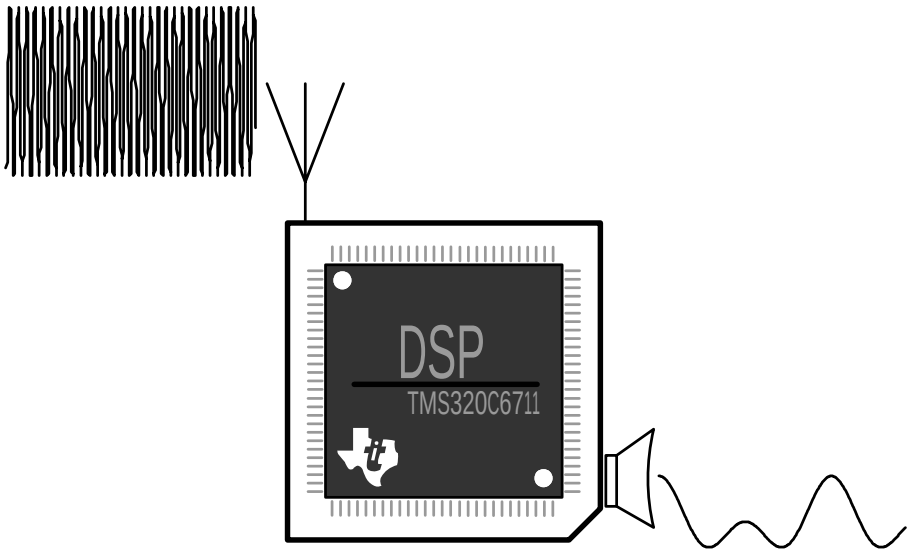
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
ELEKTROTECHNIK

Diploma Thesis

Nanyang
Technological University

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 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 themeselves 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

List of Figures

List of Tables

List of Listings

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).

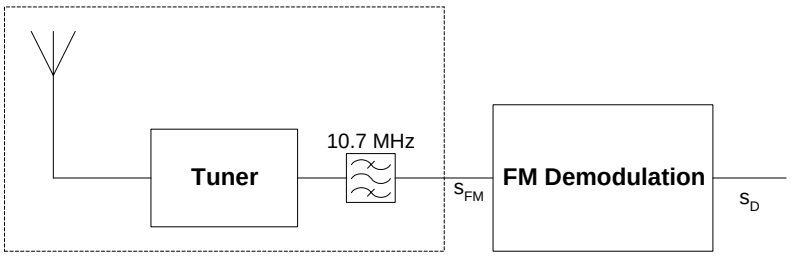


Figure 1.1.: *Block diagram FM receiver architecture*

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).

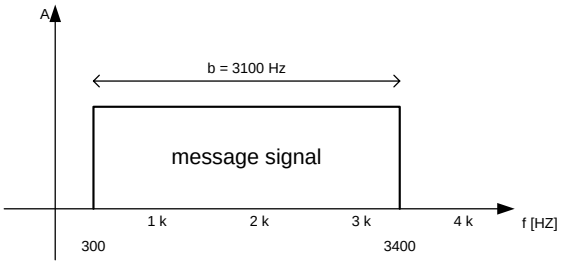


Figure 1.2.: *Spectrum message signal*

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).

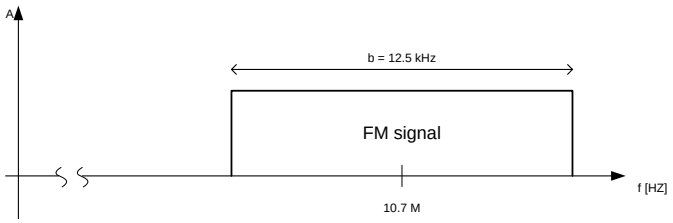


Figure 1.3.: *Spectrum FM signal*

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.

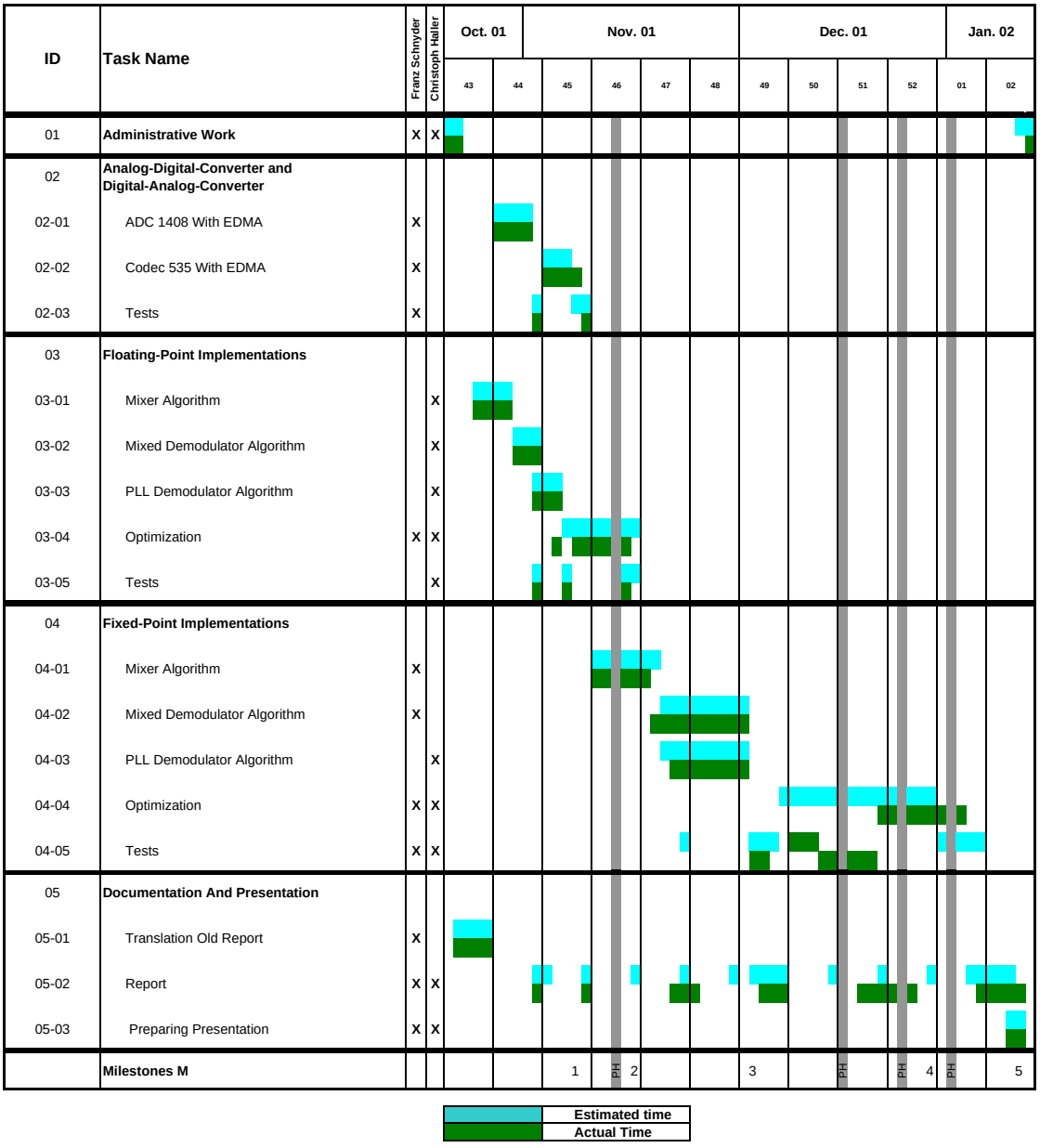


Figure 2.1.: *Project schedule*

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.

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

for FM, the relation of s_N to ϕ_{FM} is given by

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

where ω_T is the carrier angular frequency expressed ind 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

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

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

Let

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

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

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

The modulated FM signal is now

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

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

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

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

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

where f_g is the highest frequency in the message. Hence the bandwidth rises with a rising ΔF 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

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

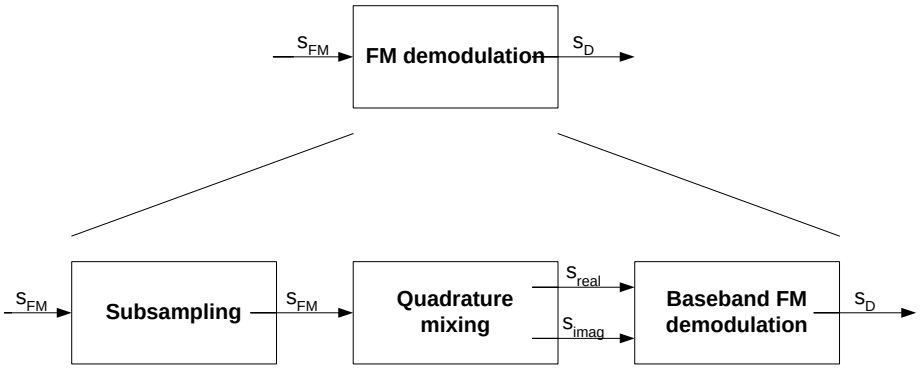


Figure 3.1.: *Subdivisions of FM demodulation block*

the sample rate is

$$f_A = 2 \cdot b \tag{3.5}$$

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

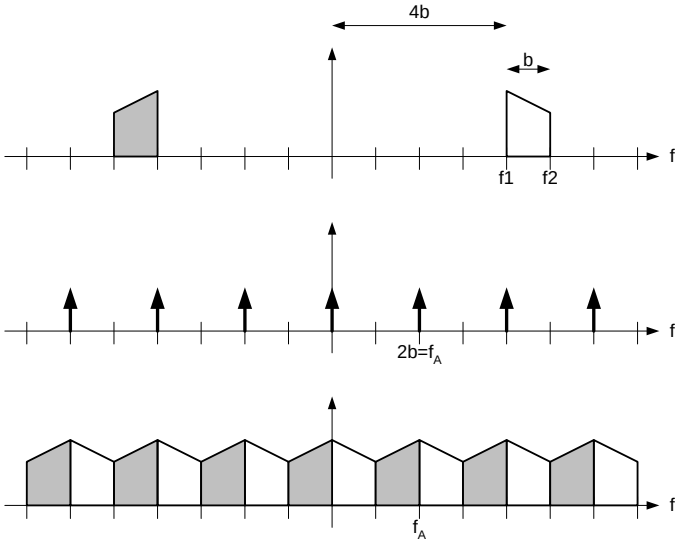


Figure 3.2.: *Spectrum subsampling with even λ*

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 λ is often not fulfilled. Thus, the bandwidth has to increase.

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

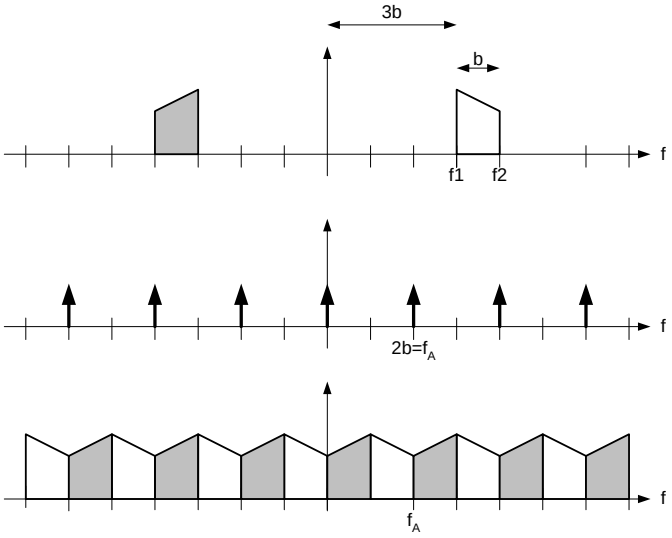


Figure 3.3.: *Spectrum subsampling with odd λ*

the new bandwidth is

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

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

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

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}

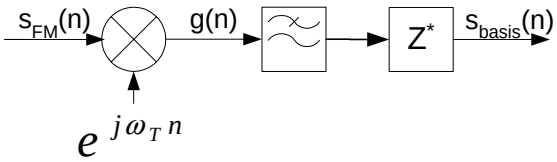


Figure 3.4.: *Quadrature-Mixer*

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

The output signal of the mixer is:

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

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

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

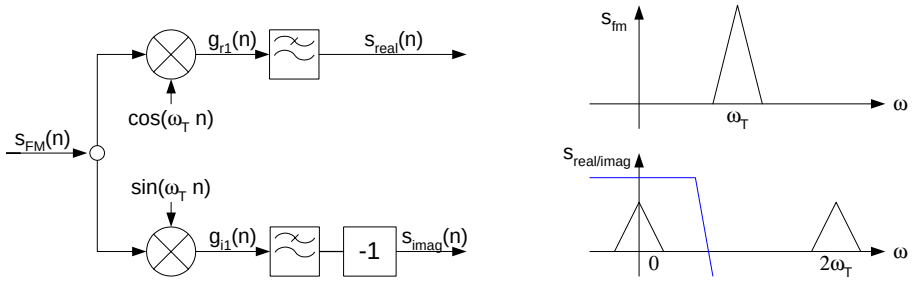


Figure 3.5.: *Real quadrature-mixer*

the FM signal s_{FM}.

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

Therefore, the output signals are

$$\begin{aligned} g_{r1}(n) &= s_{FM}(n) \cdot \underline{\cos(\omega_T n)} = A \cdot \cos(\omega_T n + \phi_{FM}(n)) \cdot \cos(\omega_T n) \\ &= \frac{A}{2}\left[\cos(\omega_T n + \phi_{FM}(n) - \omega_T n) + \cos(\omega_T n + \phi_{FM}(n) + \omega_T n)\right] \\ &= \underline{\frac{A}{2}\cos(\phi_{FM}(n))} + \frac{A}{2}\cos(2\omega_T n + \phi_{FM}(n)) \end{aligned}$$

$$s_{real}(n) = g_{r1}(n) * g_{TP}(n) = \frac{A}{2}\cos(\phi_{FM}(n)) \tag{3.9}$$

$$\begin{aligned} g_{i1}(n) &= s_{FM}(n) \cdot \underline{\sin(\omega_T n)} = A \cdot \cos(\omega_T n + \phi_{FM}(n)) \cdot \sin(\omega_T n) \\ &= \frac{A}{2}\left[\sin(-\omega_T n - \phi_{FM}(n) + \omega_T n) + \sin(\omega_T n + \phi_{FM}(n) + \omega_T n)\right] \\ &= \underline{\frac{A}{2}\sin(-\phi_{FM}(n))} + \frac{A}{2}\sin(2\omega_T n + \phi_{FM}(n)) \end{aligned}$$

$$s_{imag}(n) = (-1) \cdot g_{i1}(n) * g_{TP}(n) = \frac{A}{2}\sin(\phi_{FM}(n)) \tag{3.10}$$

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

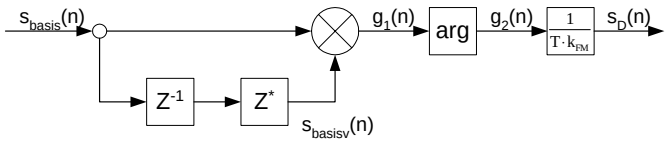


Figure 3.6.: *Complex baseband delay demodulator*

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.

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

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

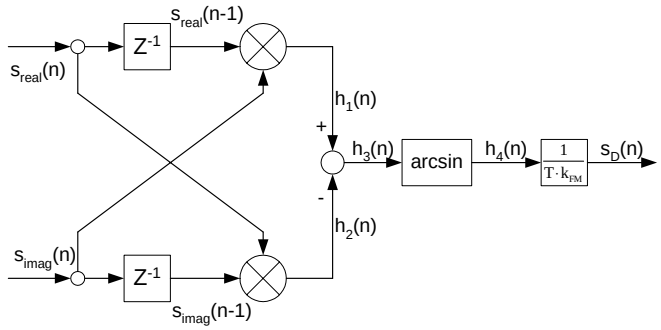


Figure 3.7.: *Real baseband delay demodulator*

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

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

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.

$$k_{FM} \cdot \hat{s}_N = \Delta F \cdot 2 \cdot \pi$$
$$\frac{\Delta F \cdot 2 \cdot \pi}{f_A} < \frac{\pi}{2}$$
$$\Delta F < \frac{\pi}{2} \cdot \frac{f_A}{2 \cdot \pi} < \frac{f_A}{4} \tag{3.13}$$

It shows that the maximal ΔF depends on the sampling rate. Therefore it is not limited, because with rise of ΔF 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.

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

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:

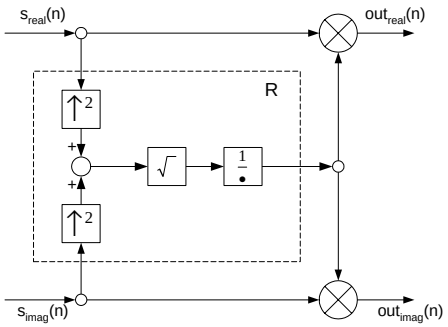


Figure 3.8.: *Real amplitude normalization*

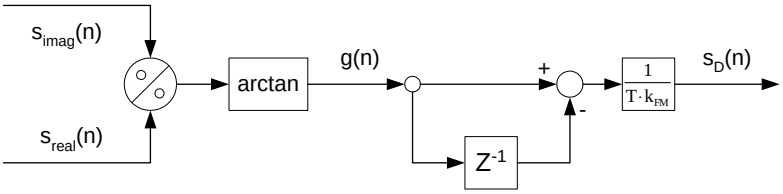


Figure 3.9.: *Phase-adapter demodulator*

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

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

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. 3.3the maximal frequency derivation can be determined.

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

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

$$\Delta F < \frac{\pi \cdot f_N}{2} \tag{3.15}$$

The maximum derivation ΔF 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 ϕ_{FM} the problem of a division by zero is eliminated. This is because the signal $s_{real} = \cos(\phi_{FM})$ 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

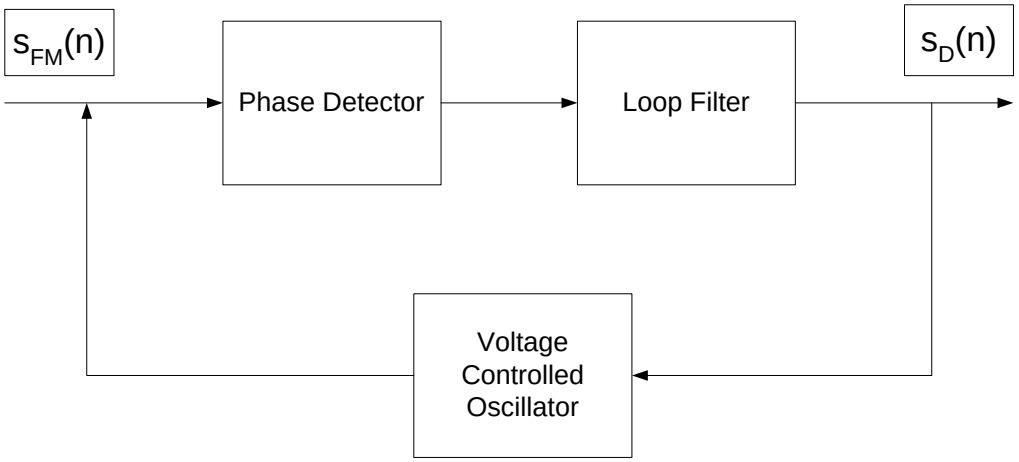


Figure 3.10.: *Phase-Locked Loop*

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 assosiation 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-Siganl $s_{FM}(n)$ 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:

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

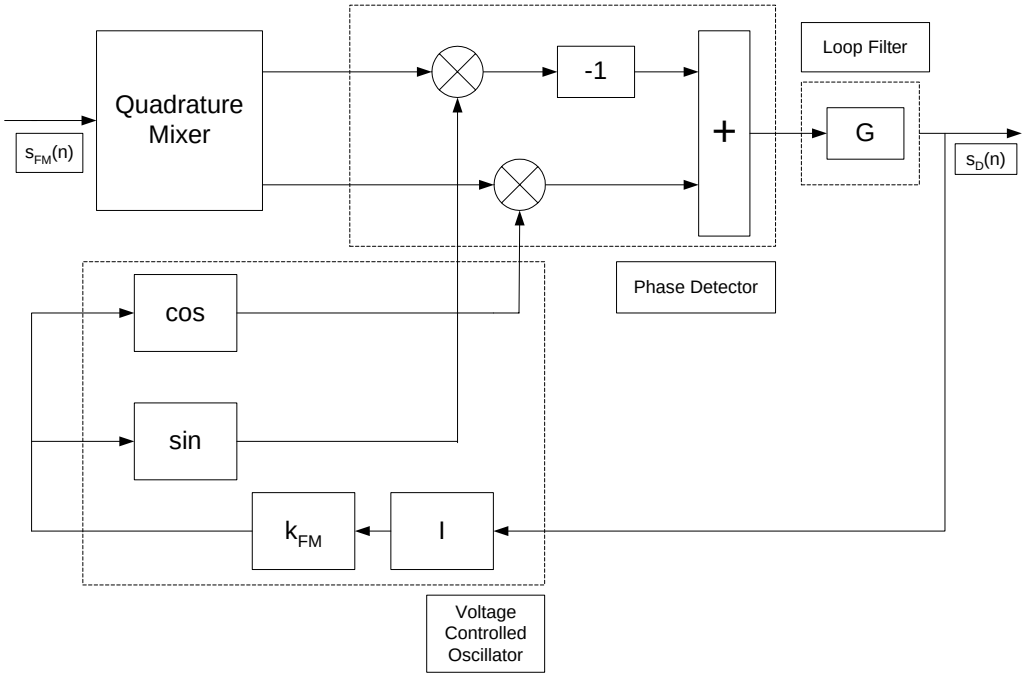


Figure 3.11.: *Actual realization Phase-Locked Loop*

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:

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

The equation can be written as:

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

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

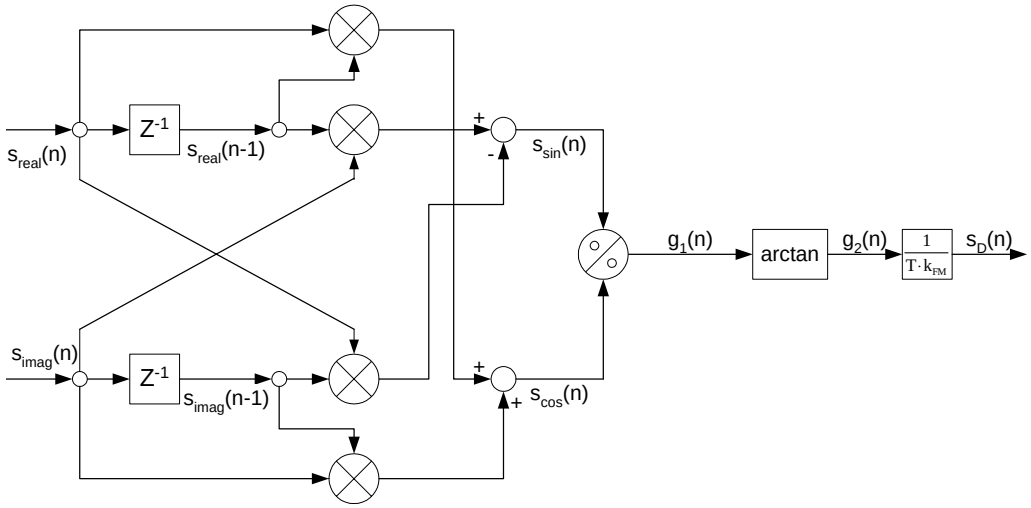


Figure 3.12.: *Mixed demodulator*

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:

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

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

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

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

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 ΔF as in the the delay demodulator can be made (refer to Eq. 3.13). ΔF 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 ® (Dynamic Systems for MATLAB ®) 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}:

$$b_{FM} = 2 \cdot (\Delta F + f_g) = 2 \cdot \left(\frac{\hat{s} \cdot k_{FM}}{2\pi} + f_g \right)$$
$$\hat{s} \cdot k_{FM} = 2\pi \left(\frac{b_{FM}}{2} - f_g \right)$$

$$k_{FM} = \frac{2\pi}{\hat{s}} \left(\frac{b_{FM}}{2} - f_g \right) = \frac{2\pi}{1} \left(\frac{12500}{2} - 3400 \right) = 2\pi \cdot 2850 \approx 18000 \tag{4.1}$$

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

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

The range of a floating-point representation is

$$\pm \left[10^{-38} ... 10^{38} \right]$$

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

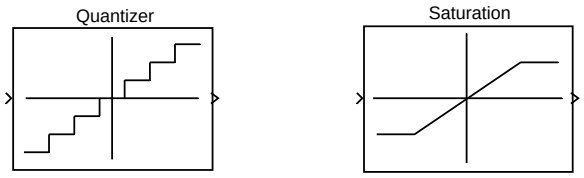


Figure 4.1.: *Quantizer and saturation blocks in Simulink*

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

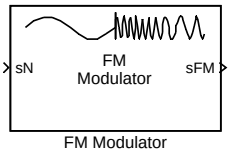


Figure 4.2.: *Simulink block FM modulator*

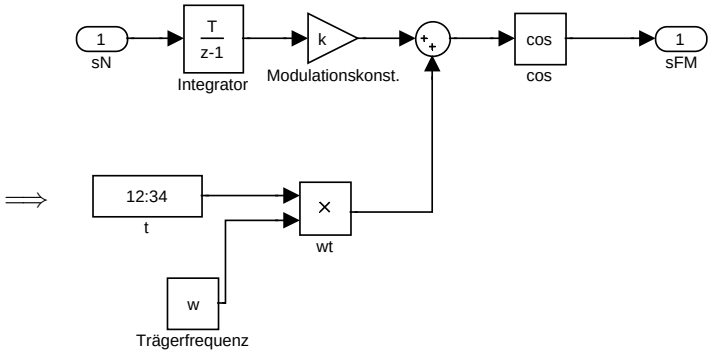


Figure 4.3.: *Simulink model FM modulators*

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

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

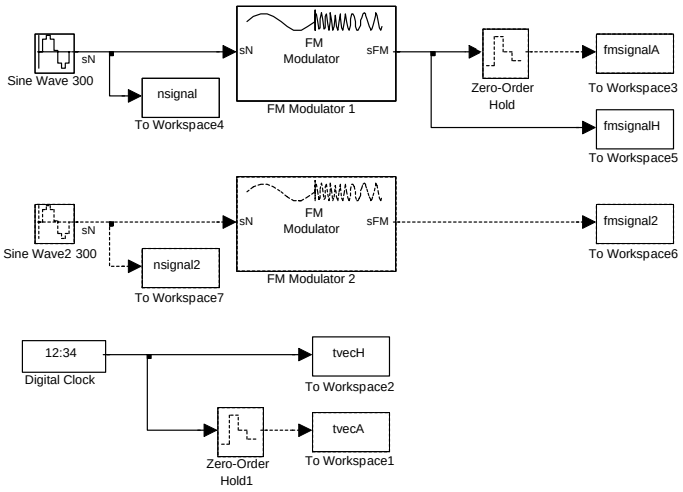


Figure 4.4.: *Simulink model subsampling*

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

The second frequency f_H is used to simulate the continues, analog system which will be subsampeled, 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

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

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`.

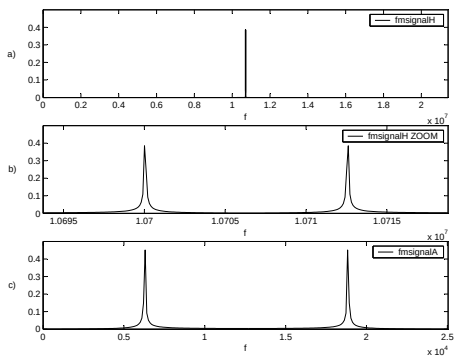


Figure 4.5.: *Plot 1 subsampling*

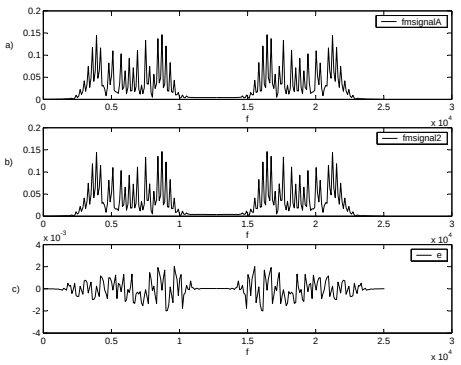


Figure 4.6.: *Plot 2 subsampling*

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:

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

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

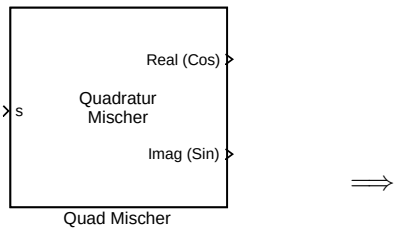


Figure 4.7.: *Simulink block quadrature-mixer*

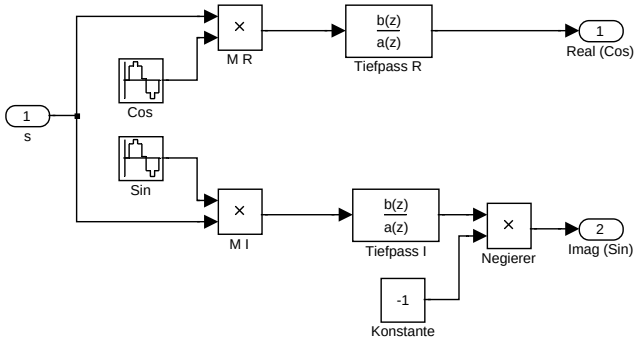


Figure 4.8.: *Simulink model quadrature-mixer*

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]=butter(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	$6250Hz$
Carrier Frequency	$2 \cdot \pi \cdot 6250rad/sec$

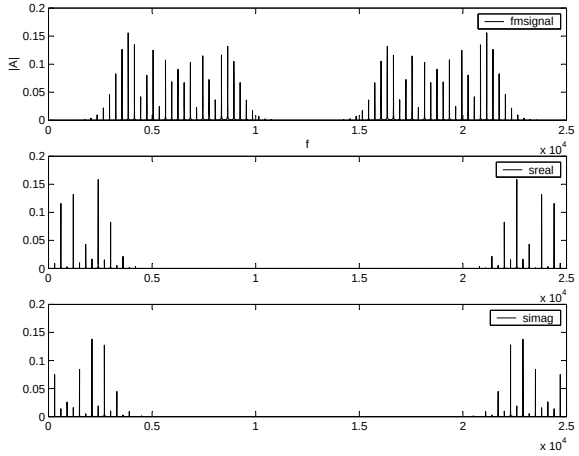


Figure 4.9.: *Plot quadrature-mixer*

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

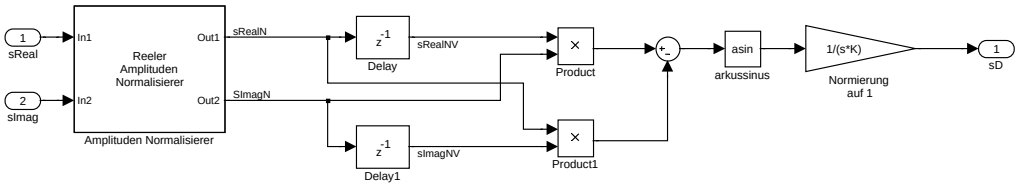


Figure 4.10.: *Simulink model baseband delay demodulator*

k measured with various k_{FM} and various frequencies of a harmonic sinusoidal message signal. With Eq. 3.13 the maximal ΔF 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 $f_g \approx 4000Hz$ 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

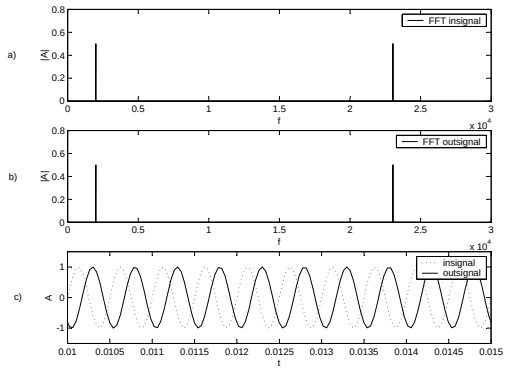


Figure 4.11.: *Plot of delay demodulator ideal with* $k_{FM} = 180$

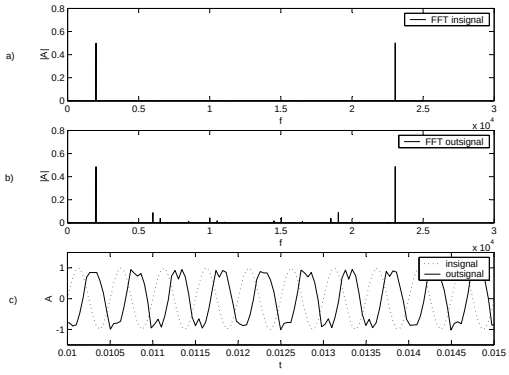


Figure 4.12.: *Plot of delay demodulator ideal with* $k_{FM} = 18000$

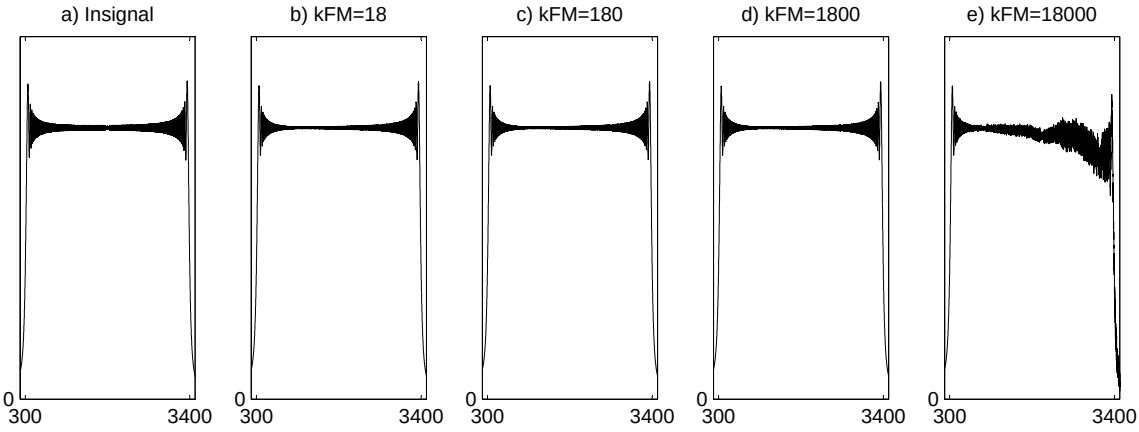


Figure 4.13.: *Spectrum of the chirp signal from a ideal delay demodulator*

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.

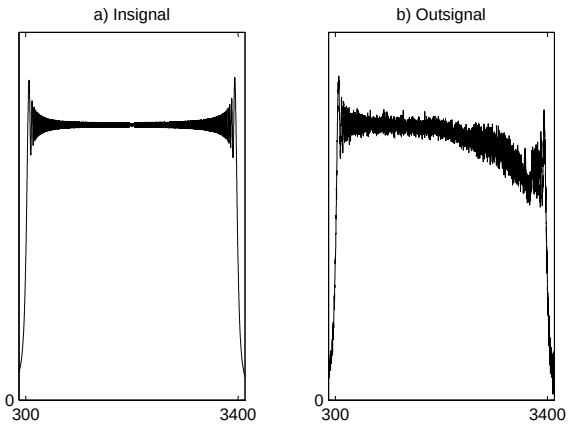


Figure 4.14.: *Spectrum of chirp signal for delay demodulator DSP model*

In the DSP model of the Delaydemodulator, a lower limit for ΔF 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 substraction (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

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

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 ΔF and the maximal k_{FM} can be calculated.

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

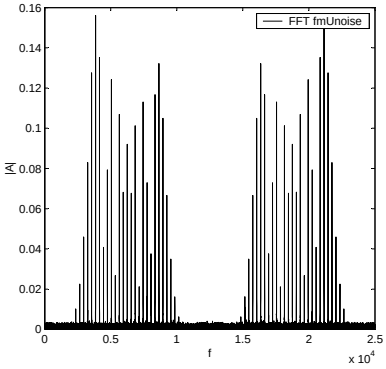


Figure 4.15.: *Plot FM signal with noise*

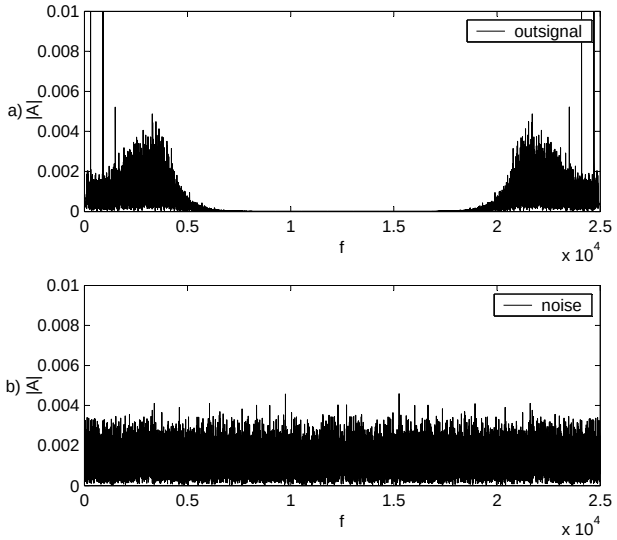


Figure 4.16.: *Plot noise and output signal with noise*

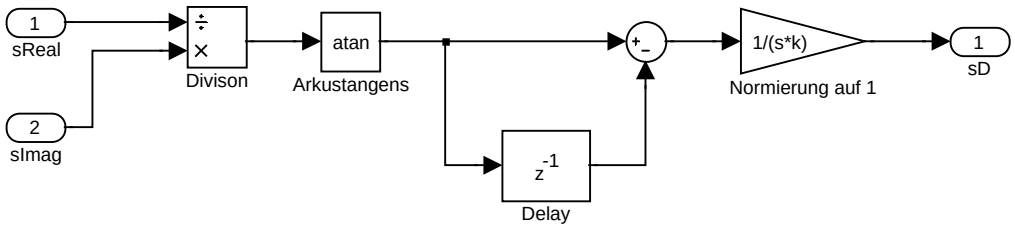


Figure 4.17.: *Simulink model phase-adapter demodulator ideal*

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

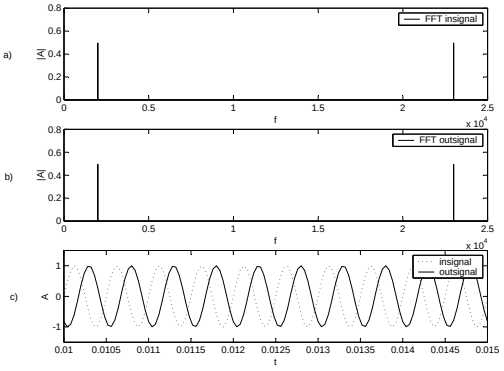


Figure 4.18.: *Plot phasea-adapter demodulator with* $k_{FM} = 180$

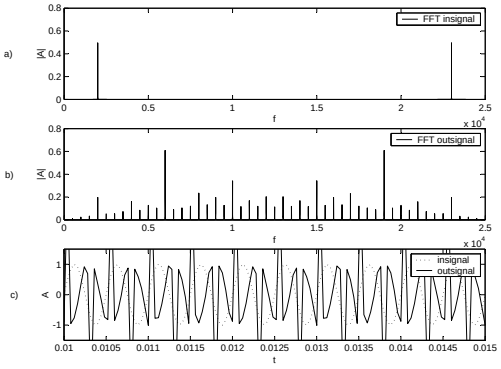


Figure 4.19.: *Plot phase-adapter demodulator with* $k_{FM} = 18000$

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.

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

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

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

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

Substituted then into Eq. 3.18:

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

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:

$$\Delta F = \frac{k_{FM}}{2 \cdot \pi} = \omega_N \tag{4.2}$$

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

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

and

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

$S_N(\omega_N \cdot t)$ is the antiderivative of $s_N(\omega_N \cdot t)$ and ω_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 $f_N = 2000Hz$. (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

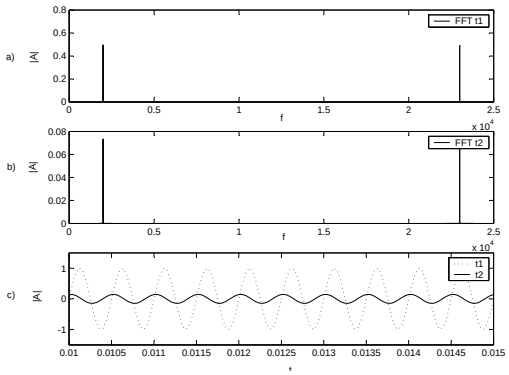


Figure 4.20.: *Plot PLL ideal with* $k_{FM} =$ 1800

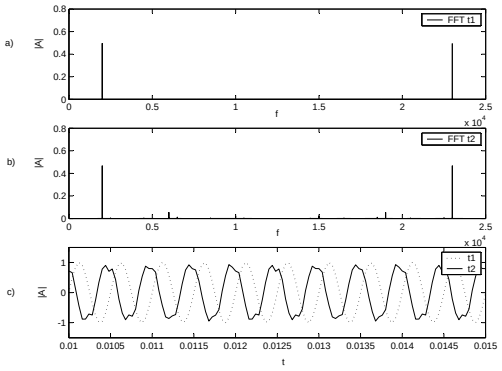


Figure 4.21.: *Plot PLL ideal with* $k_{FM} =$ 18000

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 $f_N = 300Hz$ to $f_N = 3400Hz$.

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.

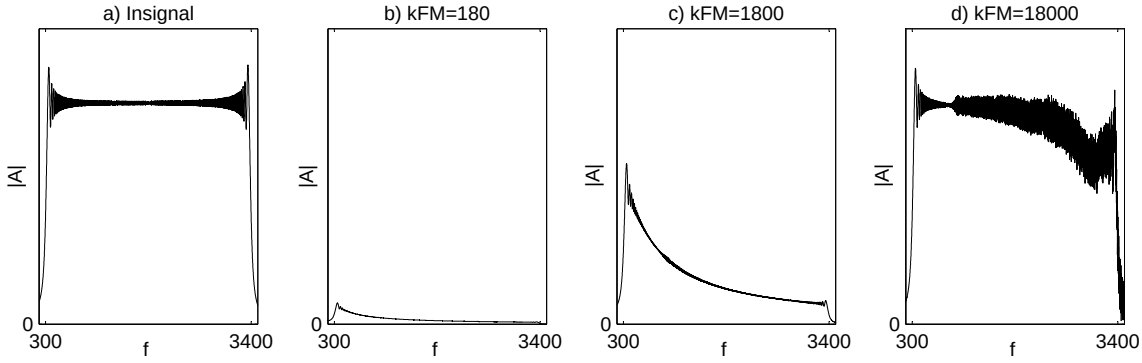


Figure 4.22.: *Spectrum of the chirp signal for the ideal PLL*

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

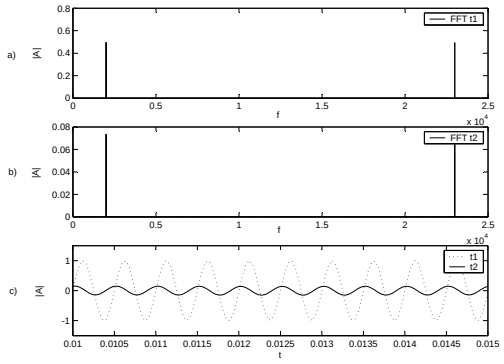


Figure 4.23.: *PLL DSP $k_{FM} = 1800$, with bandpass filter*

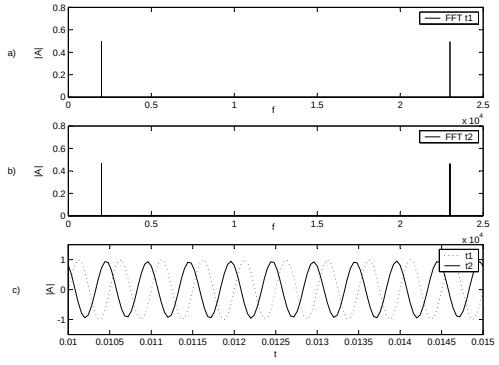


Figure 4.24.: *PLL DSP $k_{FM} = 18000$, with bandpass filter*

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

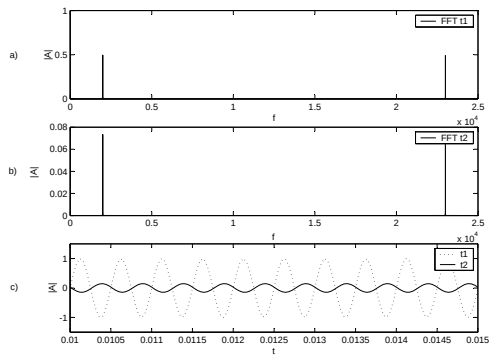


Figure 4.25.: *PLL DSP $k_{FM} = 1800$, without bandpass filter*

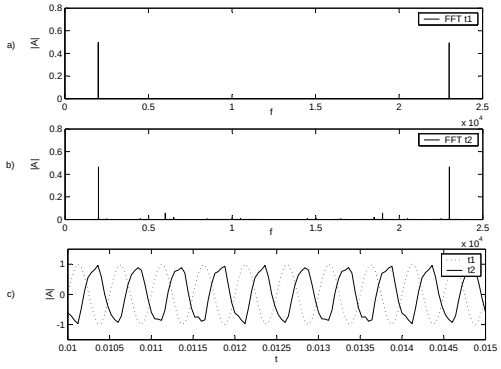


Figure 4.26.: *PLL DSP $k_{FM} = 18000$, without bandpass filter*

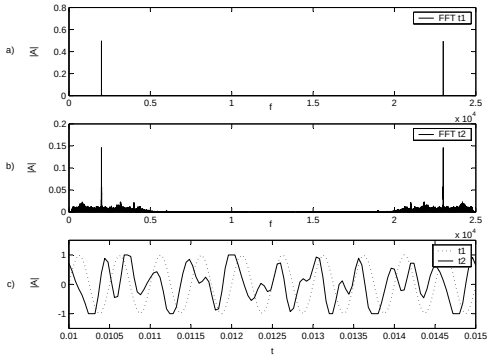


Figure 4.27.: *PLL DSP $k_{FM} = 18000$, with the bandpass filter in the feedback loop*

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-

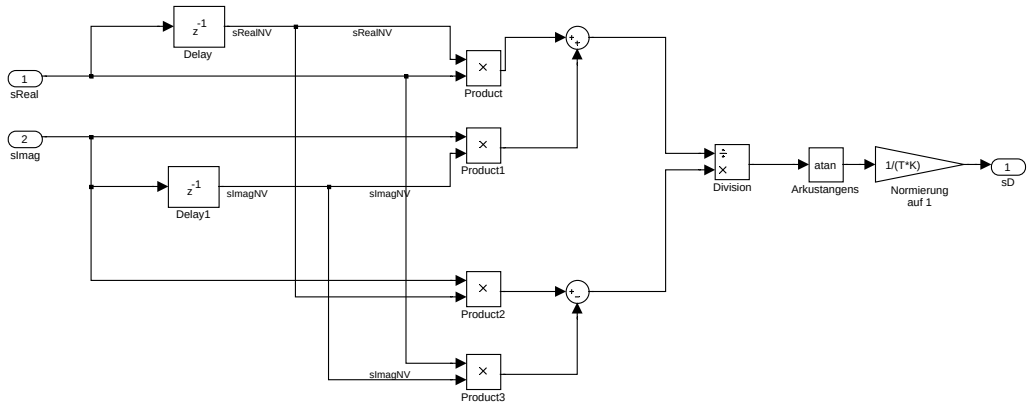


Figure 4.28.: *Simulink model mixed demodulator ideal*

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 ΔF 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 $f_g \approx 4000Hz$ 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.

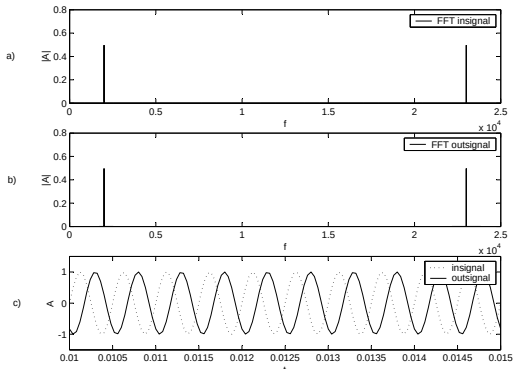


Figure 4.29.: *Plot of ideal mixed demodulator with $k_{FM} = 180$*

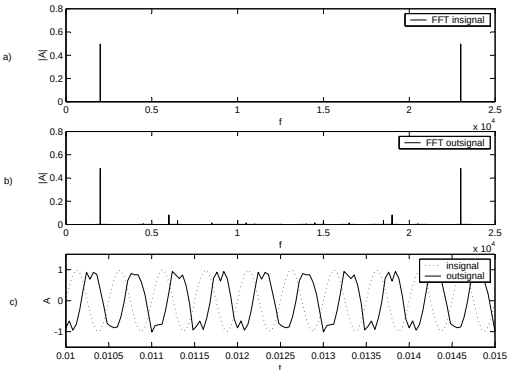


Figure 4.30.: *Plot of ideal mixed demodulator with $k_{FM} = 18000$*

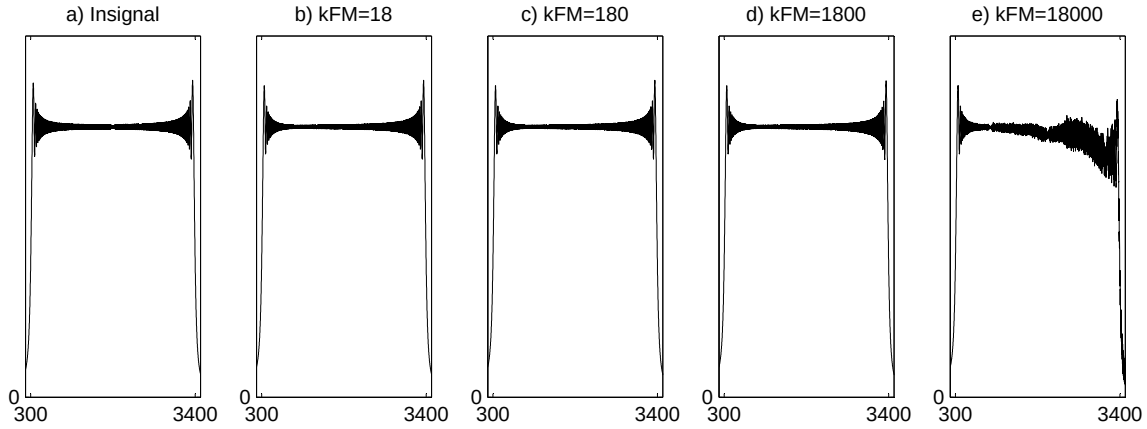


Figure 4.31.: *Spectrum of the chirp signal for an ideal mixed demodulator*

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.

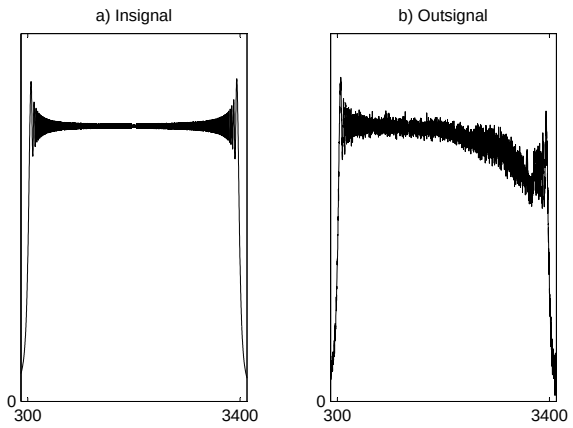


Figure 4.32.: *Spectrum of the chirp signal for DSP mixed demodulator*

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 $f_N = 300Hz$. 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

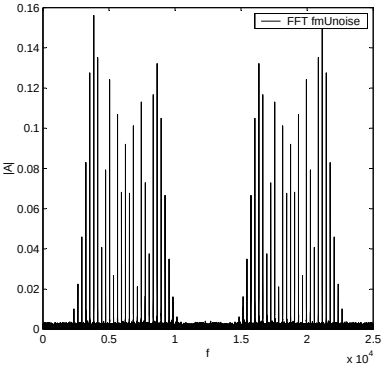


Figure 4.33.: *Plot of FM signal with noise*

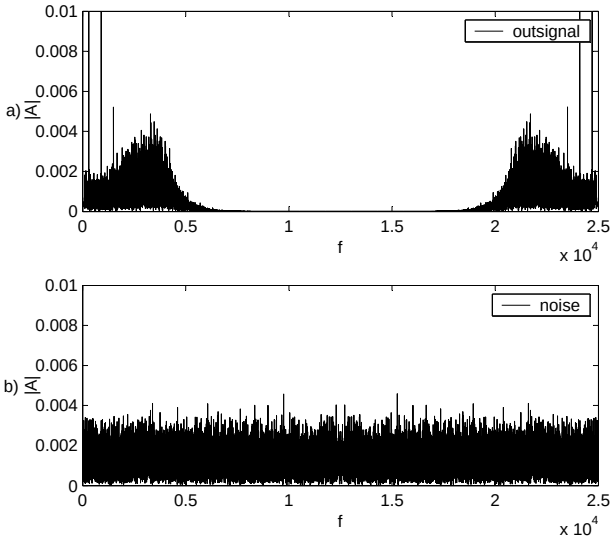


Figure 4.34.: *Plot of the noise and the output signal*

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.

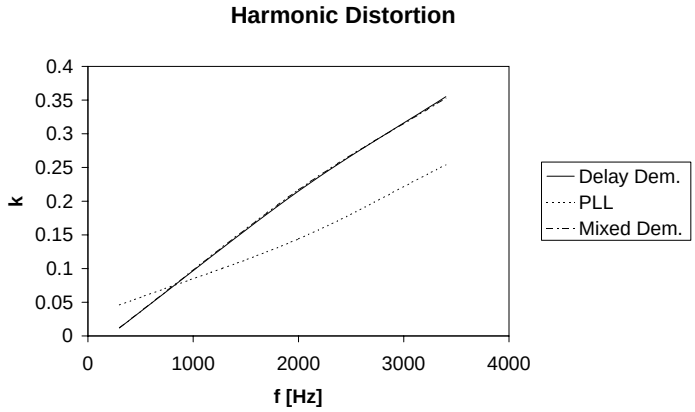


Figure 4.35.: *Comparison of the harmonic distortion for ideal models of different algorithms with constant k_{FM} of 18000 and variable frequency*

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

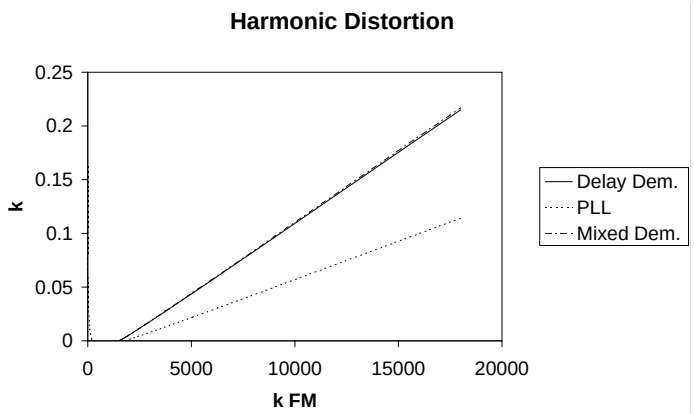


Figure 4.36.: *Comparison of harmonic distortion for ideal models of different algorithms with constant frequency of 2000Hz and variable k_{FM}*

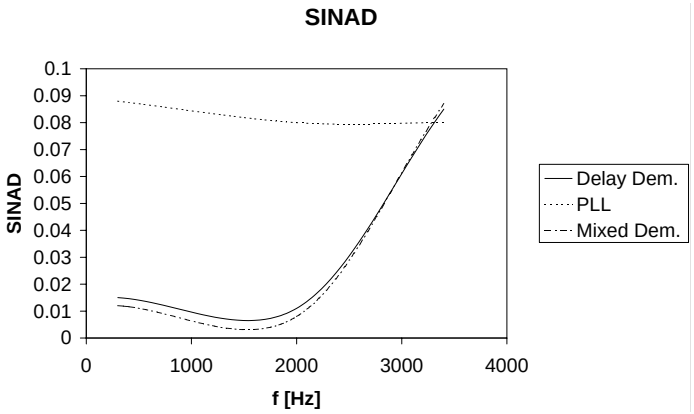


Figure 4.37.: *Comparison of SINAD for DSP models of different algorithms with constant k_{FM} of 18000 and variable frequency*

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

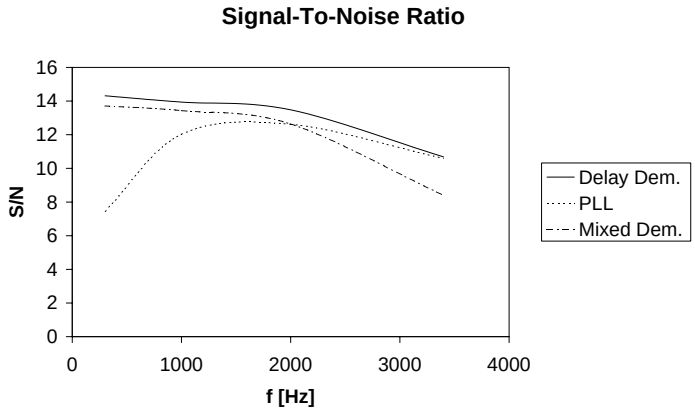


Figure 4.38.: *Comparison of S/N ratio for different demodulation algorithmen, with a k_{FM} of 18000*

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 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.

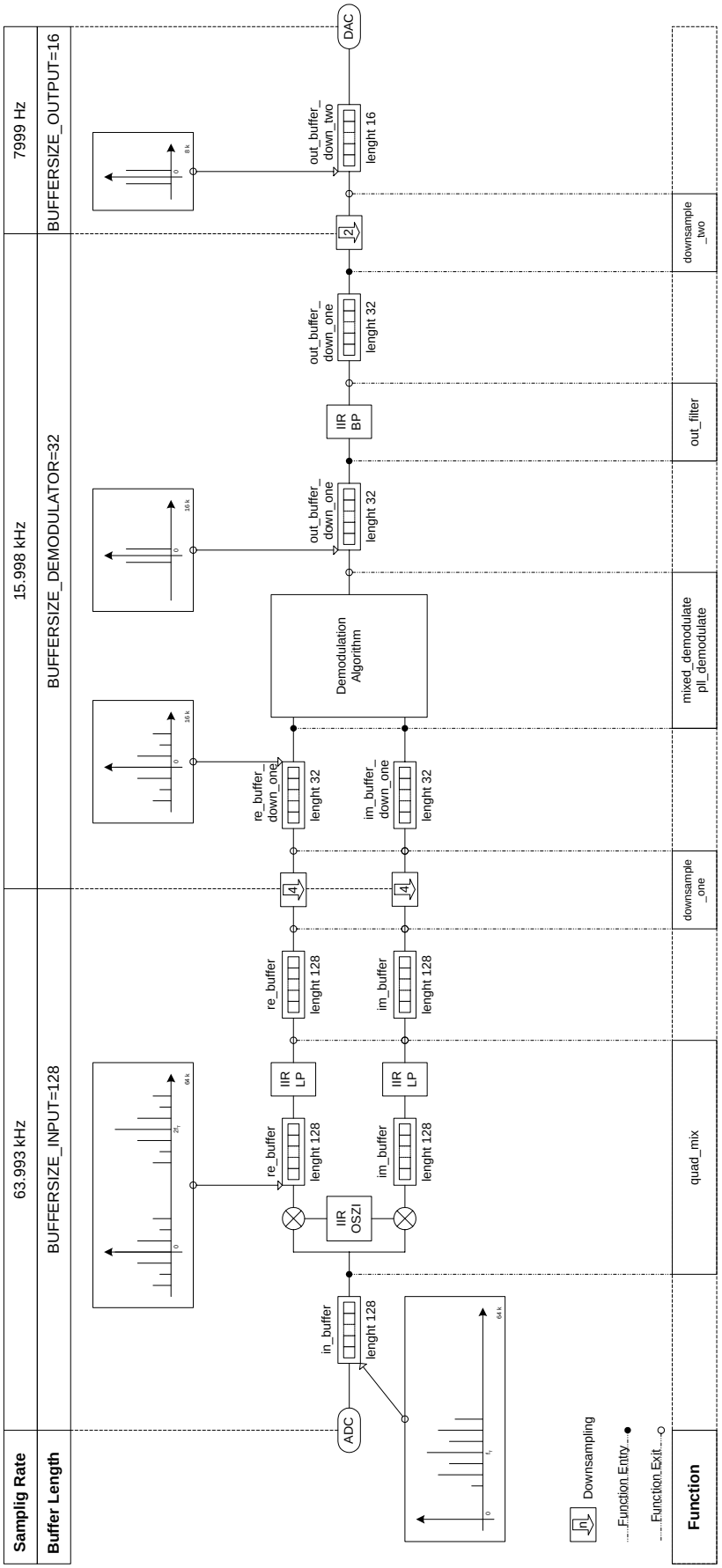


Figure 5.1.: *Overview of floating-point implementation*

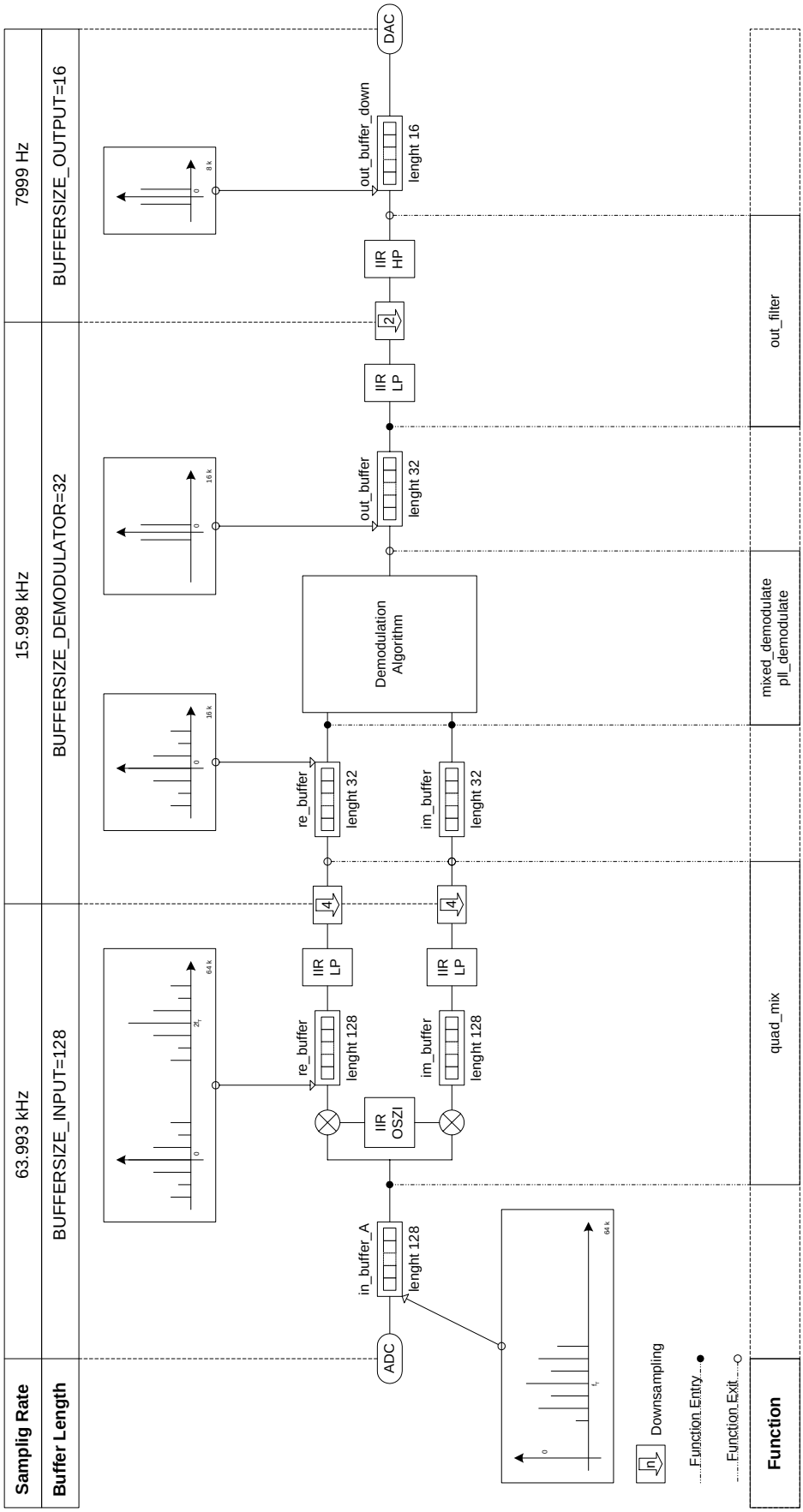


Figure 5.2.: *Overview of fixed-point implementation*

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

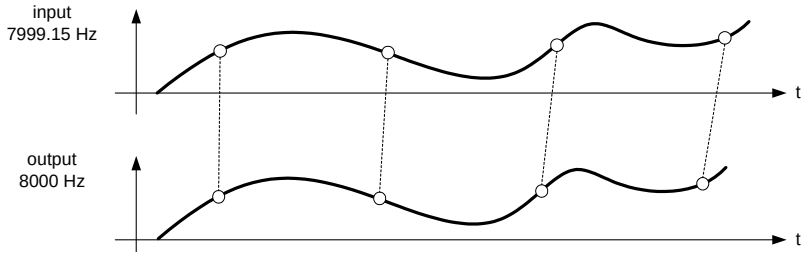


Figure 5.3.: *Input and output samples*

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.

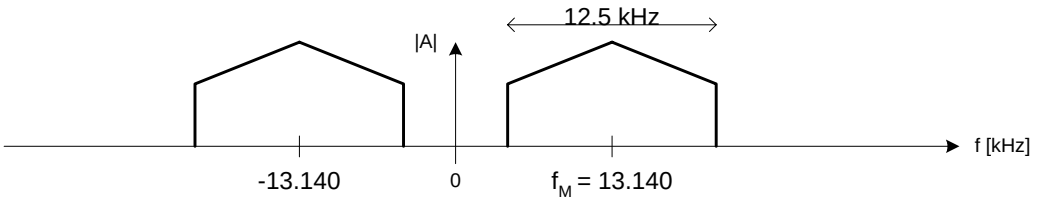


Figure 5.4.: *Spectrum of FM signal after subsampling*

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

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

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

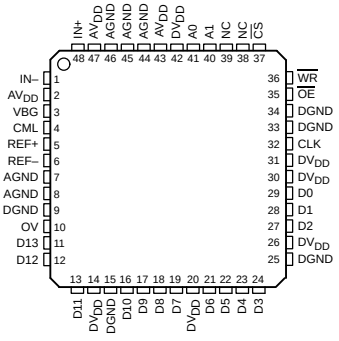


Figure 5.5.: *THS1408*

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

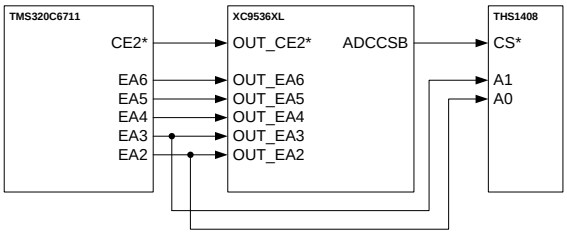


Figure 5.6.: *Schema address bus wiring THS1408*

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:

$$f_{clk} = \frac{f_{CPU}}{4 \cdot 2 \cdot i} \tag{5.2}$$

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.

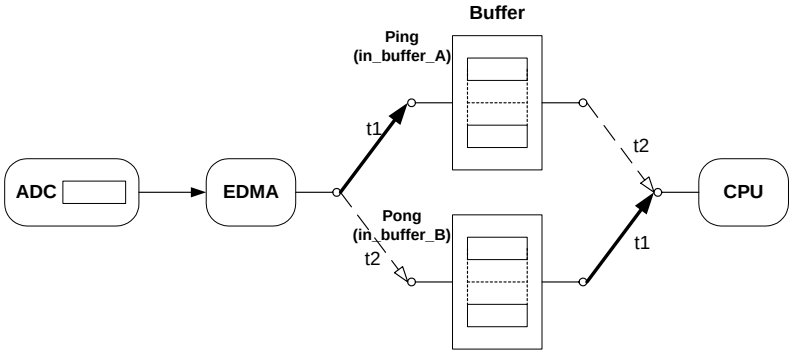


Figure 5.7.: *Ping Pong Buffering*

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.

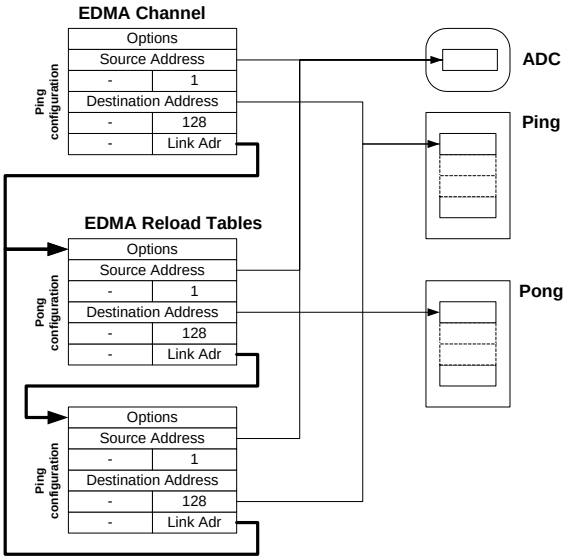


Figure 5.8.: *EDMA setup for Ping Pong Buffering*

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.

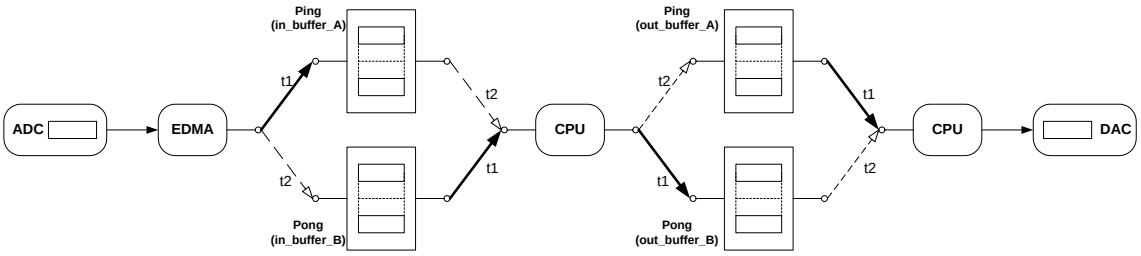


Figure 5.9.: *Ping Pong Buffering DAC*

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.

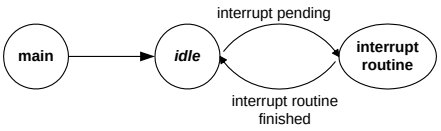


Figure 5.10.: *Programm flow*

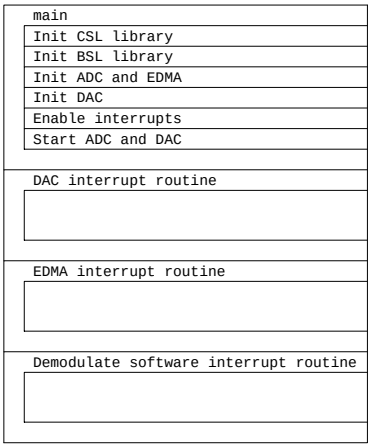


Figure 5.11.: *Layout main file*

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-

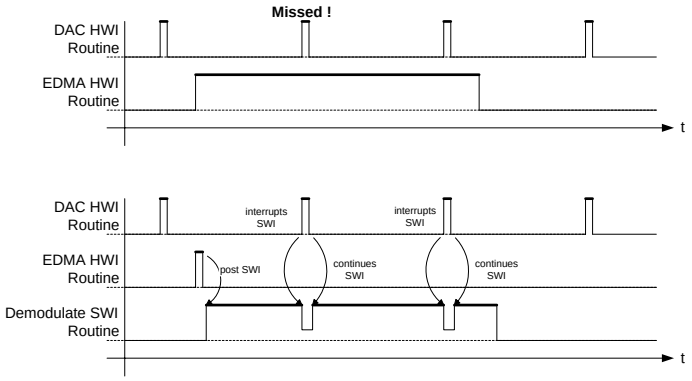


Figure 5.12.: *Scheduling with software interrupt*

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.

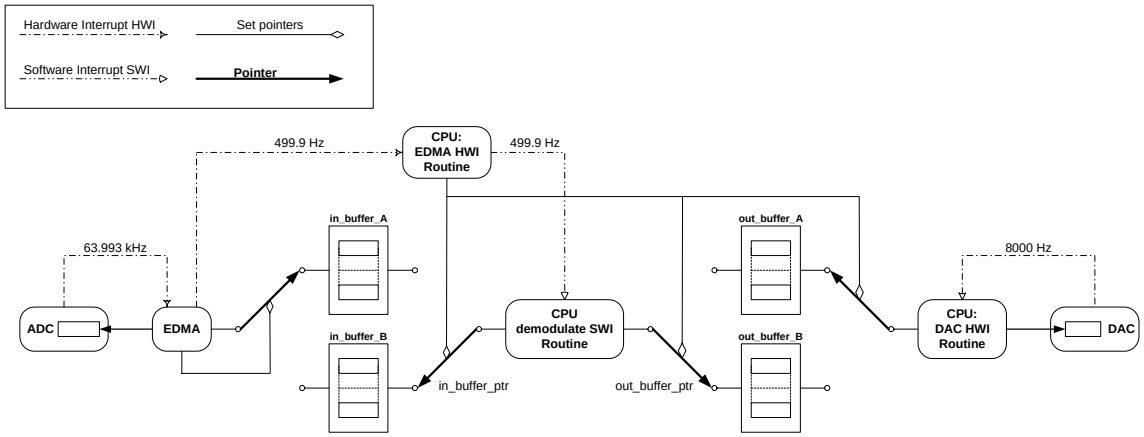


Figure 5.13.: *Overview interrupts*

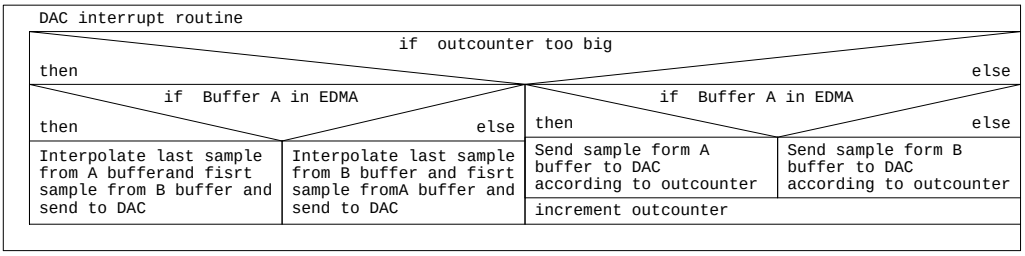


Figure 5.14.: *DAC interrupt routine*

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]+
                 out_buffer_B[0])/2));
10       }else
11       {
12          AD535_HWI_write(outAD535h,((out_buffer_B[15]+
                 out_buffer_A[0])/2));
13       }
14    }else /* output a value */
15    {
16       /* select the buffer */
17       if(a_inEDMA)
18       {
19          AD535_HWI_write(outAD535h,out_buffer_A[outcounter]);
20       }else
21       {
22          AD535_HWI_write(outAD535h,out_buffer_B[outcounter]);
23       }
24       /* update outcounter */
25       outcounter++;
26    }
27 }
```

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       SWI_post(&demodulate_swi);
19    }
20    if (EDMA_intTest(THS1408_BUFFER_B_TCC))
21    {
22       /* Clear transfer completion interrupt flag */
23       EDMA_intClear(THS1408_BUFFER_B_TCC);
24
25       /* set buffer pointers */
26       a_inEDMA=1;
27       in_buffer_ptr=in_buffer_B;
28       out_buffer_ptr=out_buffer_B;
29
30       /* post software interrupt -> start demodulation
               */
31       SWI_post(&demodulate_swi);
32    }
33 }
```

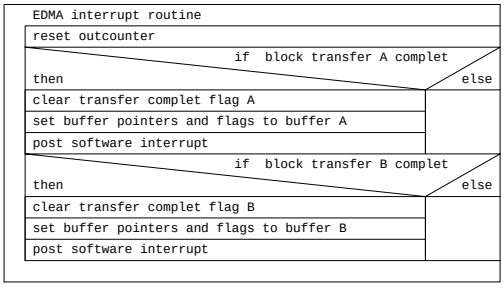


Figure 5.15.: *EDMA interrupt routine*

Listing 5.3: *Demodulate softawre 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 }
```

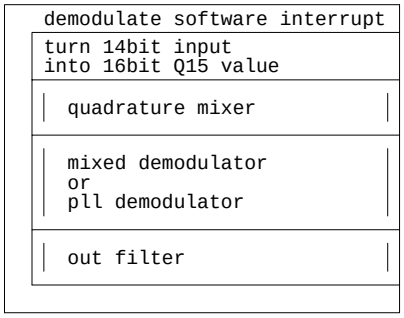


Figure 5.16.: *Demodulate software interrupt routine*

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:

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

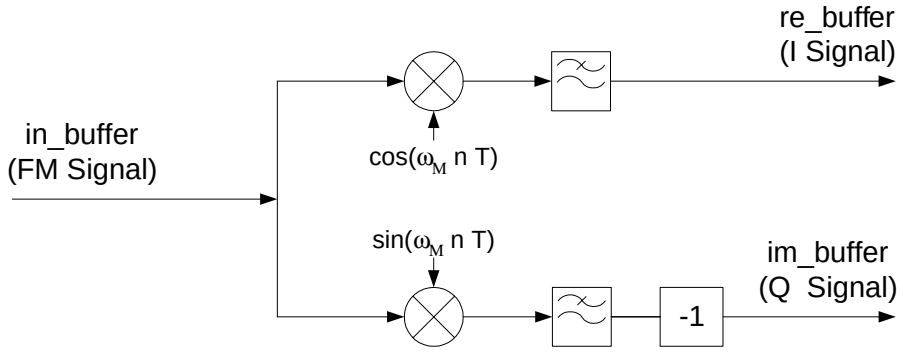


Figure 5.17.: *Quadrature mixer*

and:

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

T represents the sample time, which is used to run the filters. ω_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:

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

and:

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

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 ($fp = 6300Hz$). 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 ($fs = 19000Hz$). The pass band loss and the stop band attenuation were declared, to optain 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:

```
%IIR low-pass filter for quadrature mixing

%specifications:
%sample frequency in Hz
fa=150000000/(8*293);
%pass frequency in Hz
fp=6300;
%stop frequency in Hz
fs=19000;
%loss in pass band in dB
dp=2;
%loss in stop band in dB
ds=60;

%computing:
[n, fn]=buttord(fp*2/fa, fs*2/fa, dp, ds);
%return:
%order
n
%cutoff frequency
fn

%computing:
[B,A]=butter(n,fn);
%return:
%denominator of transfer function
B
%nominator of transfer function
A

%computing:
[H,W]=freqz(B,A,n);
%return:
freqzplot(H,W);
```

The filter order is computed to $n = 5$ and the 3dB cutoff frequency to $f_n = 6653Hz$. 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.

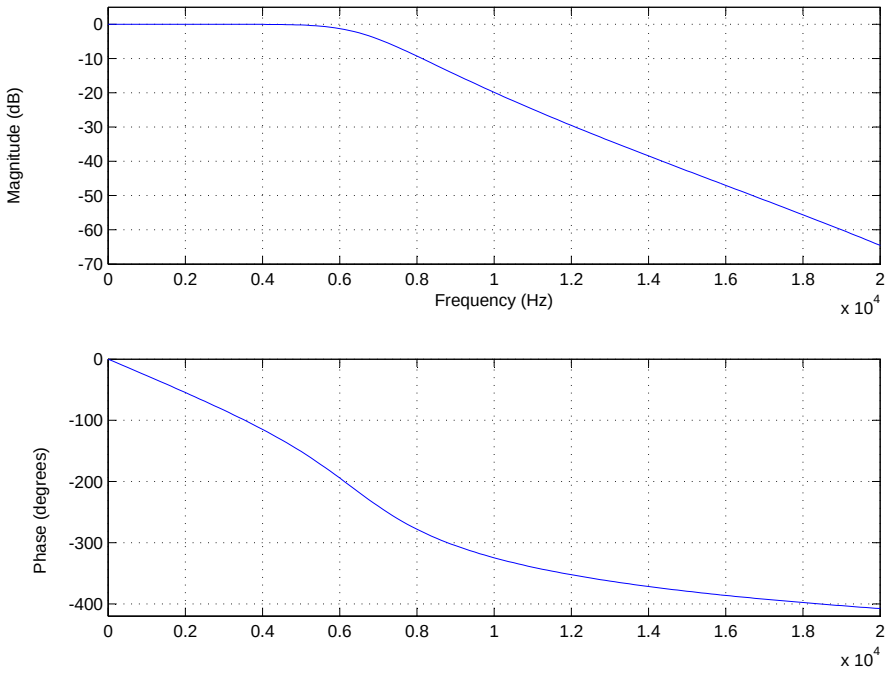


Figure 5.18.: *Frequency response of low-pass filter for quadrature mixing, floating point*

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.

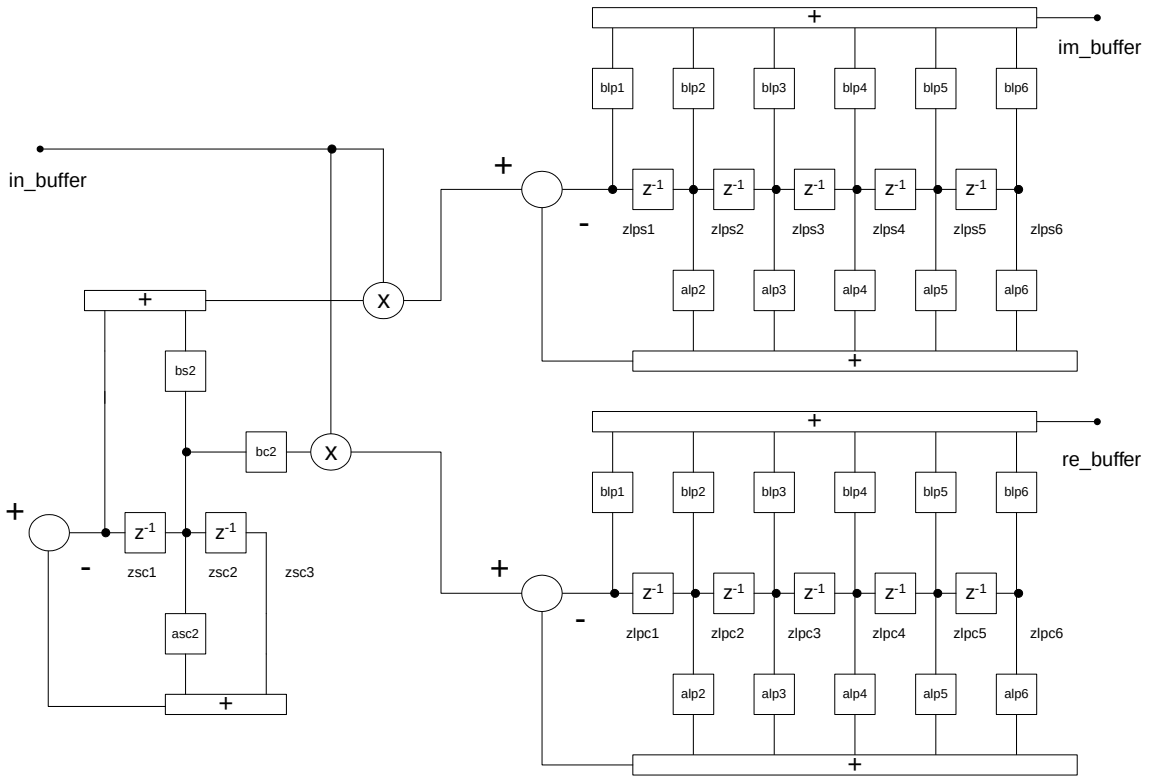


Figure 5.19.: *Digital block diagram of quadrature mixer*

```
1  void quad_mix(float in_buffer[] , float re_buffer[] , float
       im_buffer[])
2  {
3     int i;
4
5     /*__cosine/sine generation__*/
6
7     for(i=0;i<BUFFERSIZE_INPUT;i++)
8     {
9        /*feedback*/
10       zsc1=-(zsc2*asc2+zsc3);
11
12       /*foreward*/
13       re_buffer[i]=zsc1+zsc2*bc2;/*cosine*/
14       im_buffer[i]=zsc2*bs2;/*sine*/
15
16       /*shift delays*/
17       zsc3=zsc2;
18       zsc2=zsc1;
19    }
20
21    /*__mixing__*/
22
23    for(i=0;i<BUFFERSIZE_INPUT;i++)
24    {
25       /*mixing*/
26       re_buffer[i]=re_buffer[i]*in_buffer[i];/*I-path*/
27       im_buffer[i]=im_buffer[i]*in_buffer[i];/*Q-path*/
28    }
29
30    /*__low pass__*/
31
32    for(i=0;i<BUFFERSIZE_INPUT;i++)
33    {
34       /*feedback*/
35       zlpc1=re_buffer[i]-(alp2*zlpc2+alp3*zlpc3+alp4*zlpc4+
             alp5*zlpc5+alp6*zlpc6); /*I-path*/
36       zlps1=im_buffer[i]-(alp2*zlps2+alp3*zlps3+alp4*zlps4+
             alp5*zlps5+alp6*zlps6); /*Q-path*/
37
38       /*foreward*/
39       re_buffer[i]=blp1*zlpc1+blp2*zlpc2+blp3*zlpc3+blp4*zlpc4
             +blp5*zlpc5+blp6*zlpc6; /*I-path*/
40       im_buffer[i]=-(blp1*zlps1+blp2*zlps2+blp3*zlps3+blp4*
             zlps4+blp5*zlps5+blp6*zlps6); /*Q-path*/
41
42       /*shift delays*/
43       /*I-path*/
44       zlpc6=zlpc5;
45       zlpc5=zlpc4;
46       zlpc4=zlpc3;
47       zlpc3=zlpc2;
48       zlpc2=zlpc1;
49       /*Q-path*/
50       zlps6=zlps5;
51       zlps5=zlps4;
52       zlps4=zlps3;
53       zlps3=zlps2;
54       zlps2=zlps1;
55    }
56 }
```

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

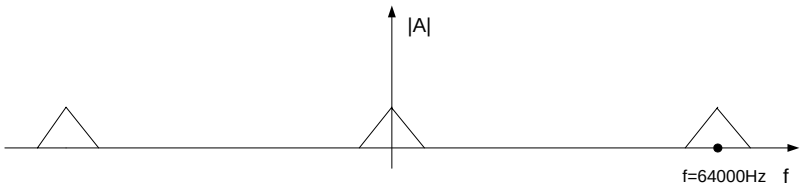


Figure 5.20.: *Signal spectrum after quadrature mixing*

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

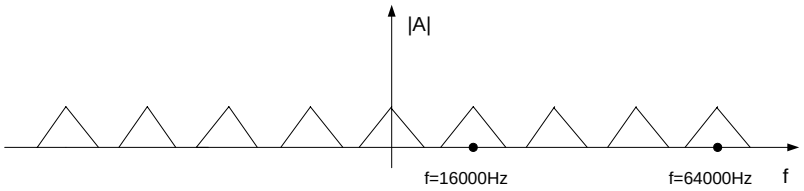


Figure 5.21.: *Signal spectrum after first down sampling*

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:

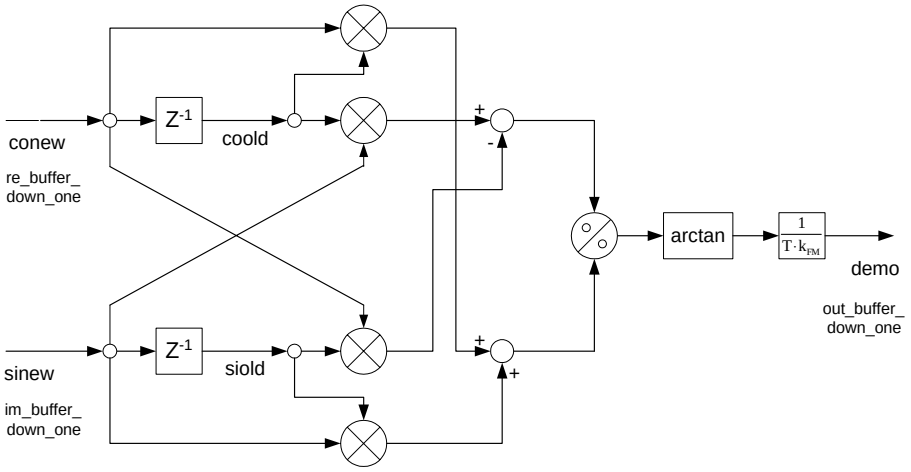


Figure 5.22.: *Block diagram of mixed demodulator*

```
1  void mixed_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] , float out_buffer_down_one[])
2  {
3     int i;
4
5     for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
6     {
7        /*__shifting values__*/
8        coold=conew;
9        siold=sinew;
10       conew=re_buffer_down_one[i];
11       sinew=im_buffer_down_one[i];
12
13       /*__calculating demodulated value__*/
14       demo=(coold*sinew-siold*conew)/(coold*conew+siold*sinew);/*atan argument*/
15       demo=atanf(demo);/*atan value*/
16       demo=demo*decon;/*demodulated value*/
17       out_buffer_down_one[i]=demo;
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 $f = 1000Hz$. The best result is achieved with $K_PLL = 2.8$.

```
1  void pll_demodulate(float re_buffer_down_one[] , float
       im_buffer_down_one[] , float out_buffer_down_one[])
2  {
3     int i;
4     float demo; /*value of demodulation*/
5
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;
```

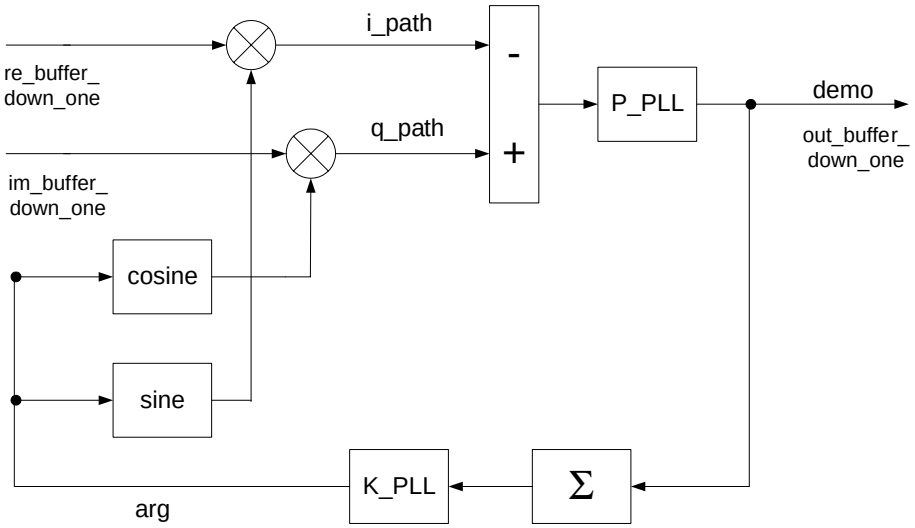


Figure 5.23.: *Block diagram of PLL demodulator*

```
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=150000000/(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*
               zbp4+abp5*zbp5+abp6*zbp6+abp7*zbp7+abp8*zbp8+abp9
               *zbp9+abp10*zbp10+abp11*zbp11);
9
10       /*foreward*/
11       out_buffer_down_one[i]=bbp1*zbp1+bbp2*zbp2+bbp3*zbp3+
               bbp4*zbp4+bbp5*zbp5+bbp6*zbp6+bbp7*zbp7+bbp8*zbp8
               +bbp9*zbp9+bbp10*zbp10+bbp11*zbp11;
12
13       /*shift delays*/
14       zbp11=zbp10;
15       zbp10=zbp9;
16       zbp9=zbp8;
17       zbp8=zbp7;
18       zbp7=zbp6;
```

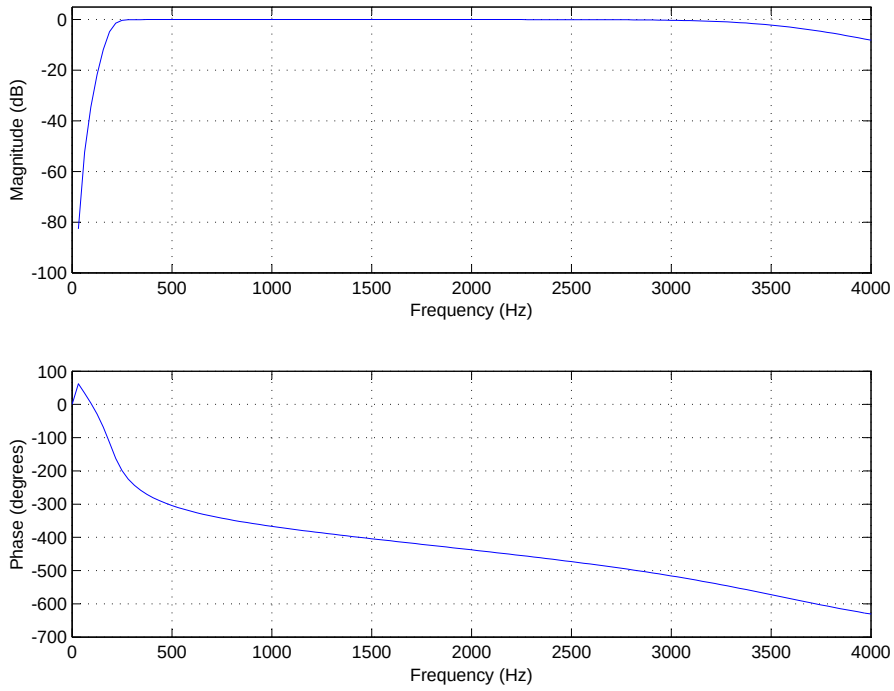


Figure 5.24.: *Band-pass filter frequency response, floating point*

```
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 $f = 8000Hz$. 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.

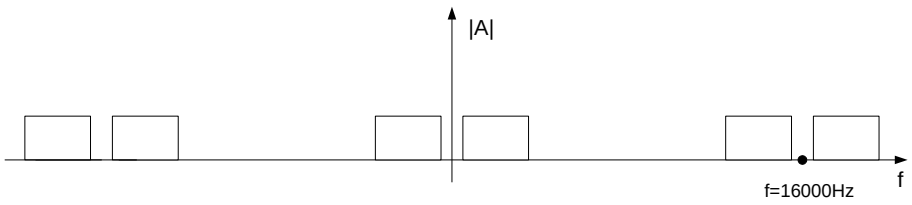


Figure 5.25.: *Signal spectrum before second down sampling*

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.

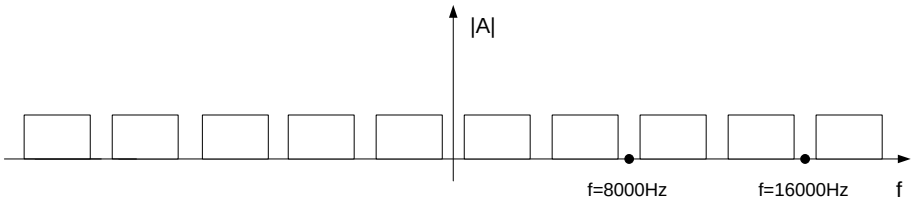


Figure 5.26.: *Signal spectrum after second down sampling*

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 a fixed-point number. Fixed-point numbers can be stored in standard integer variables by storing the mantissa.

In a Qm.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 Qm.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:

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

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

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

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

```
k = n << (r-p) if (r>=p)
k = n >> (p-r) if (p>r)
```

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:

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

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)>>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<<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).

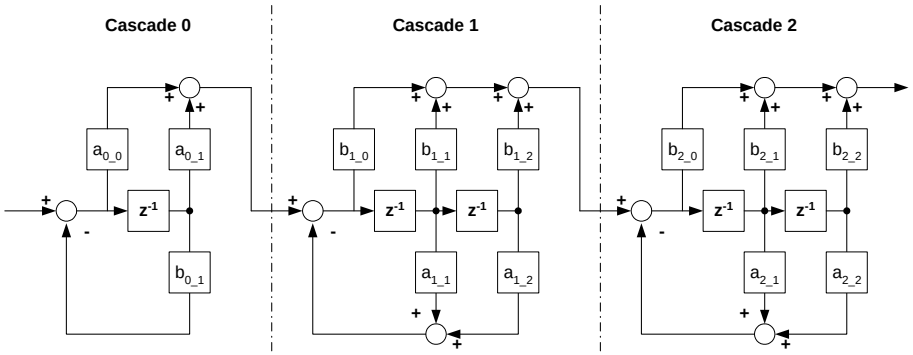


Figure 5.28.: *Block diagram of cascaded low-pass filter*

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] = butter(n,Wn)` obtains zero-pole-gain form, then `[sos,g] = 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.

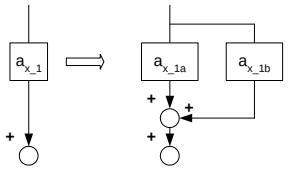


Figure 5.29.: *Change in block diagram of*

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

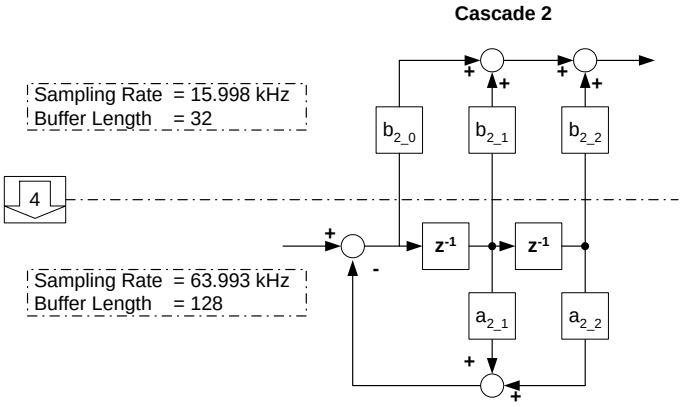


Figure 5.30.: *Output signals and delay states of cascaded low-pass filter*

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

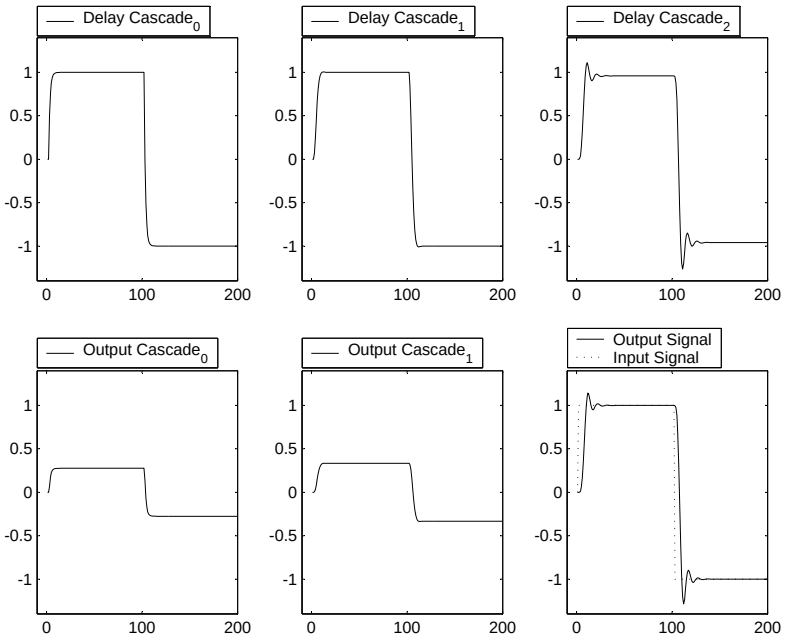


Figure 5.31.: *Output signals and delay states of cascaded low-pass filter*

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 ≤ 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:

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

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

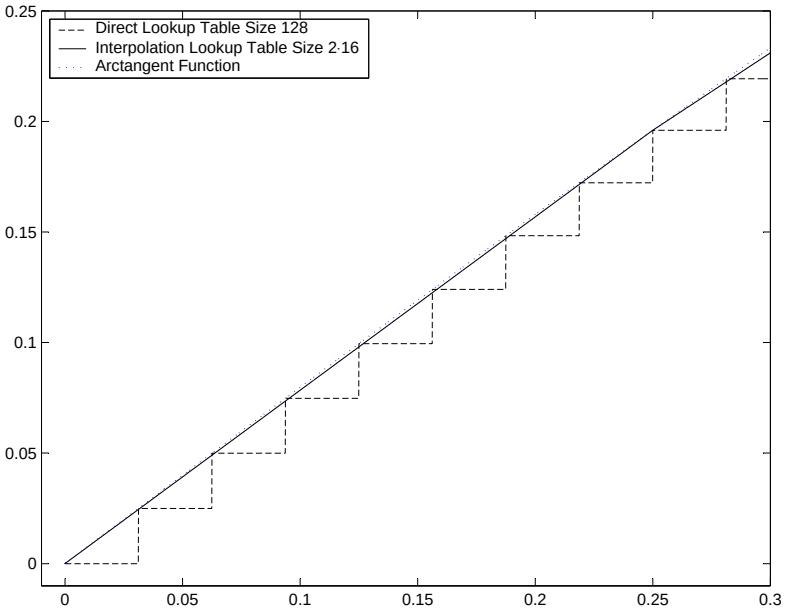


Figure 5.32.: *Plot arc tangent lookup table*

(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 -2π and -2π, 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 BUFFERSIZE_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:

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

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 BUFFERSIZE_SINE:

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

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 (BUFFERSIZE_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 BUFFERSIZE_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

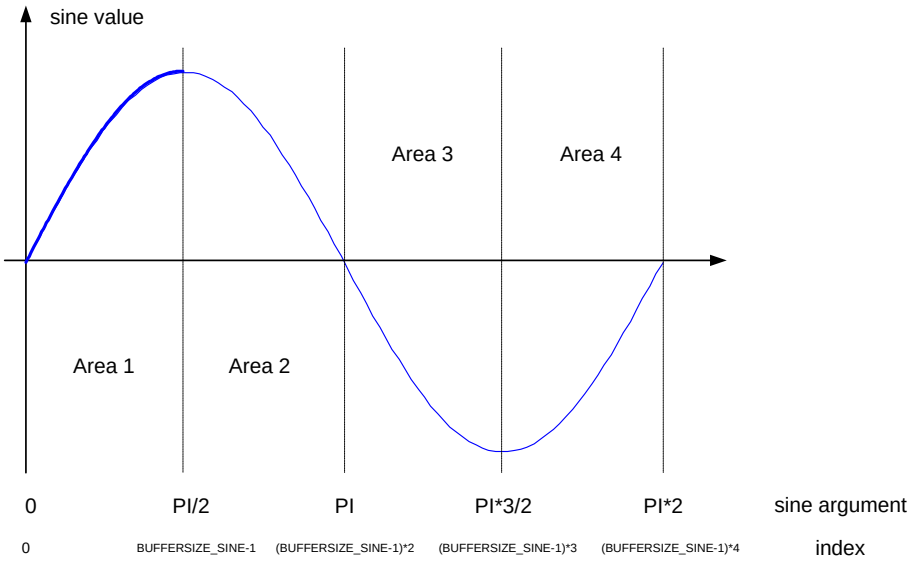


Figure 5.33.: *Sine values with belonging indexes*

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 principe is applied to the negative sine wave as well as whole cosine wave.

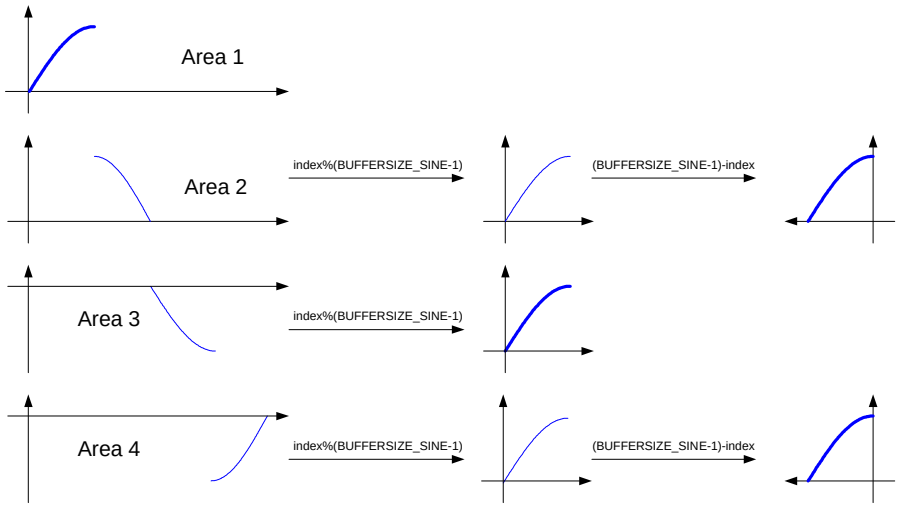


Figure 5.34.: *Index manipulation*

Listing 5.5: *PLL demodulator*

```
1  void pll_demodulate(short re_buffer[] , short im_buffer[] ,
          short out_buffer[])
2  {
3     int i;
4     int flag=0;
5
6     short demo;    /*index for sine_buffer access*/
7     short index;   /*value of demodulation*/
8
9
10    for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
11    {
12       /*__calculation of demodulated value, using Q15__*/
13
14       i_path=(re_buffer[i]*i_path)>>15;
15       q_path=(im_buffer[i]*q_path)>>15;
16
17       demo=q_path-i_path;
18       demo=(demo*P_PLL)>>15;
19
20       out_buffer[i]=(demo*GAIN_PLL)>>15;
21
22
23       /*__summation of demodulated signal, using Q3.12__*/
24
25       demo=demo>>3;
26       arg=arg+((demo*K_PLL)>>12);
27
28
29       /*__limitation of argument to 2*PI, using Q3.12__*/
30
31       if (arg>TWO_PI)
32       {
33          arg=arg-TWO_PI;
34       }
35
36       else if (arg<-TWO_PI)
37       {
38          arg=arg+TWO_PI;
39       }
40
41
42       /*__calculation of sine and cosine value of argument__*/
43
44       /*calculating of index for sine_buffer access, using
             new Q format*/
45
46       index=arg>>REDUCTION;
47
48       if (index<0)
49       {
50          index=-index;
51          flag=1;
52       }
53
54       index=(index*STEP_INVERSE)>>(2*Q_FORMAT);
55
56       if (flag==1)
57       {
58          index=-index;
59          flag=0;
60       }
61
62       /*access on sine_buffer with index*/
63
64       if(index>=0)
65       {
66          if(index<(BUFFERSIZE_SINE-1))
67          {
68             /*sine value*/
69             i_path=sine_buffer[index];
70             /*cosine value*/
71             index=(BUFFERSIZE_SINE-1)-index;
72             q_path=sine_buffer[index];
73          }
74          else if(index<2*(BUFFERSIZE_SINE-1))
75          {
76             index=index % (BUFFERSIZE_SINE-1);
77             /*sine value*/
78             index=(BUFFERSIZE_SINE-1)-index;
79             i_path=sine_buffer[index];
80             /*cosine value*/
81             index=(BUFFERSIZE_SINE-1)-index;
82             q_path=-sine_buffer[index];
83          }
84          else if (index<3*(BUFFERSIZE_SINE-1))
85          {
86             index=index % (BUFFERSIZE_SINE-1);
87             /*sine value*/
88             i_path=-sine_buffer[index];
89             /*cosine value*/
90             index=(BUFFERSIZE_SINE-1)-index;
91             q_path=-sine_buffer[index];
92          }
93          else if (index<4*(BUFFERSIZE_SINE-1))
94          {
95             index=index % (BUFFERSIZE_SINE-1);
96             /*sine value*/
97             index=(BUFFERSIZE_SINE-1)-index;
98             i_path=-sine_buffer[index];
99             /*cosine value*/
100            index=(BUFFERSIZE_SINE-1)-index;
101            q_path=sine_buffer[index];
102         }
103         else
104         {
105            index=index % (BUFFERSIZE_SINE-1);
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      }
113      else
114      {
115         if(index>-(BUFFERSIZE_SINE-1))
116         {
117            index=-index;
```

```
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 chose 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 digit 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

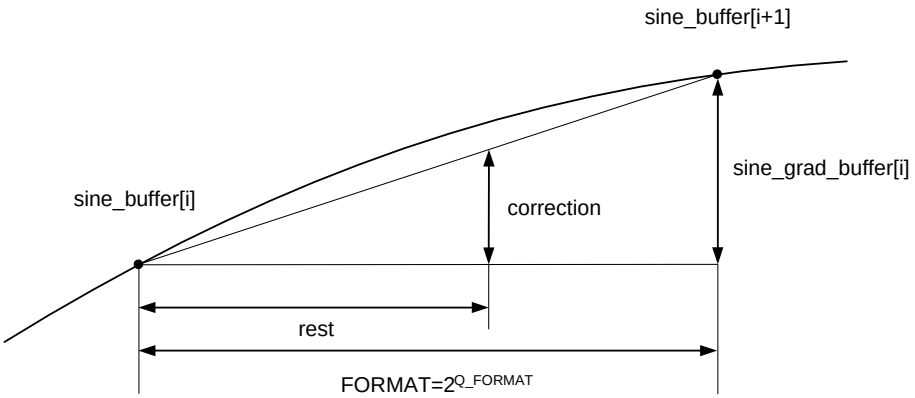


Figure 5.35.: *Linear interpolation*

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

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

$$correction = (sine\_grad\_buffer[i] \cdot rest) >> Q\_FORMAT$$

The interpolated sine value is:

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

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.

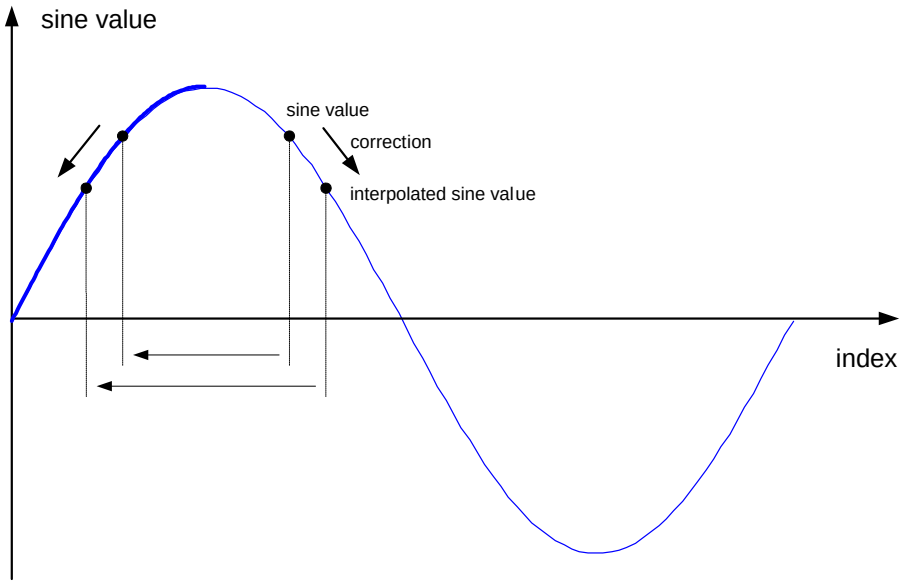


Figure 5.36.: *Sign of correction changes*

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
            format*/
4
5     index=arg>>REDUCTION;
6
7     if (index<0)
8     {
9        index=-index;
10       flag=1;
11    }
12
13    index=(index*STEP_INVERSE)>>Q_FORMAT;
14    rest=index%FORMAT;
15    index=index>>Q_FORMAT;
16
17    if (flag==1)
18    {
19       index=-index;
20       flag=0;
21    }
22
23    /*access on sine_buffer with index*/
24
25    if(index>=0)
26    {
27       if(index<(BUFFERSIZE_SINE-1))
28       {
29          /*sine value*/
30          correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                /*positive*/
31          i_path=sine_buffer[index]+correction;
32          /*cosine value*/
33          index=(BUFFERSIZE_SINE-1)-index;
34          correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                /*negative*/
35          q_path=sine_buffer[index]+correction;
36       }
37       else if(index<2*(BUFFERSIZE_SINE-1))
38       {
39          index=index % (BUFFERSIZE_SINE-1);
40          /*sine value*/
41          index=(BUFFERSIZE_SINE-1)-index;
42          correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                /*negative*/
43          i_path=sine_buffer[index]+correction;
44          /*cosine value*/
45          index=(BUFFERSIZE_SINE-1)-index;
46          correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                /*positive*/
47          q_path=-(sine_buffer[index]+correction);
48       }
49       else if (index<3*(BUFFERSIZE_SINE-1))
50       {
51          index=index % (BUFFERSIZE_SINE-1);
52          /*sine value*/
53          correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                /*positive*/
54          i_path=-(sine_buffer[index]+correction);
55          /*cosine value*/
56          index=(BUFFERSIZE_SINE-1)-index;
57          correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                /*negative*/
58          q_path=-(sine_buffer[index]+correction);
59       }
60       else if (index<4*(BUFFERSIZE_SINE-1))
61       {
62          index=index % (BUFFERSIZE_SINE-1);
```

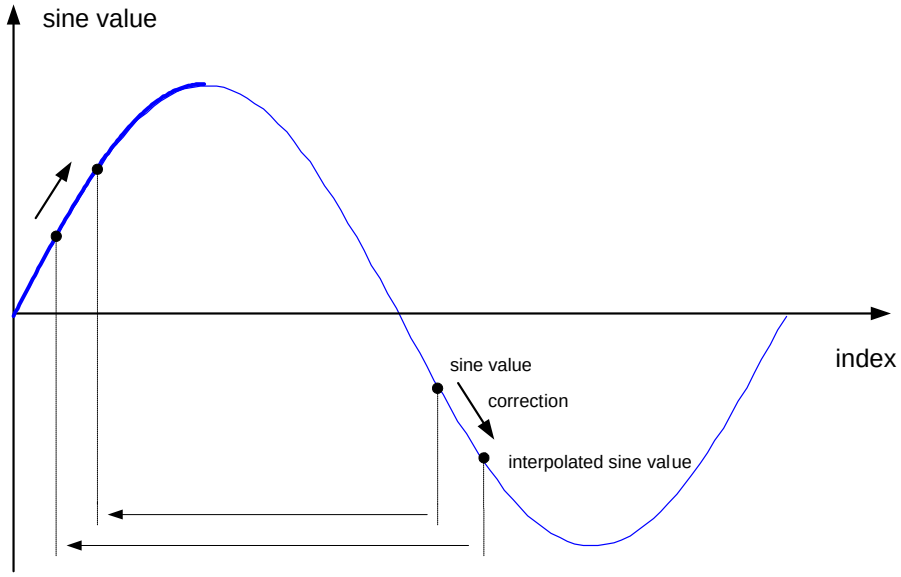


Figure 5.37.: *Sign of correction does not change*

```
63          /*sine value*/
64          index=(BUFFERSIZE_SINE-1)-index;
65          correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*negative*/
66          i_path=-(sine_buffer[index]+correction);
67          /*cosine value*/
68          index=(BUFFERSIZE_SINE-1)-index;
69          correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*positive*/
70          q_path=sine_buffer[index]+correction;
71       }
72       else
73       {
74          index=index % (BUFFERSIZE_SINE-1);
75          /*sine value*/
76          correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*positive*/
77          i_path=sine_buffer[index]+correction;
78          /*cosine value*/
79          index=(BUFFERSIZE_SINE-1)-index;
80          correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*negative*/
81          q_path=sine_buffer[index]+correction;
82       }
83    }
84    else
85    {
86       if(index>-(BUFFERSIZE_SINE-1))
87       {
88          index=-index;
89          /*sine value*/
90          correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*positive*/
91          i_path=-(sine_buffer[index]+correction);
92          /*cosine value*/
93          index=(BUFFERSIZE_SINE-1)-index;
94          correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*negative*/
95          q_path=sine_buffer[index]+correction;
96       }
97       else if(index>-2*(BUFFERSIZE_SINE-1))
98       {
99          index=-(index % (BUFFERSIZE_SINE-1));
100         /*sine value*/
101         index=(BUFFERSIZE_SINE-1)-index;
102         correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*negative*/
103         i_path=-(sine_buffer[index]+correction);
104         /*cosine value*/
105         index=(BUFFERSIZE_SINE-1)-index;
106         correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*positive*/
107         q_path=-(sine_buffer[index]+correction);
108      }
109      else if (index>-3*(BUFFERSIZE_SINE-1))
110      {
111         index=-(index % (BUFFERSIZE_SINE-1));
112         /*sine value*/
113         correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*positive*/
114         i_path=sine_buffer[index]+correction;
115         /*cosine value*/
116         index=(BUFFERSIZE_SINE-1)-index;
117         correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*negative*/
118         q_path=-(sine_buffer[index]+correction);
119      }
120      else if (index>-4*(BUFFERSIZE_SINE-1))
121      {
122         index=-(index % (BUFFERSIZE_SINE-1));
123         /*sine value*/
124         index=(BUFFERSIZE_SINE-1)-index;
125         correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*negative*/
126         i_path=sine_buffer[index]+correction;
127         /*cosine value*/
128         index=(BUFFERSIZE_SINE-1)-index;
129         correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*positive*/
130         q_path=sine_buffer[index]+correction;
131      }
132      else
133      {
134         index=-(index % (BUFFERSIZE_SINE-1));
135         /*sine value*/
136         correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*positive*/
137         i_path=-(sine_buffer[index]+correction);
138         /*cosine value*/
139         index=(BUFFERSIZE_SINE-1)-index;
140         correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;
                  /*negative*/
141         q_path=sine_buffer[index]+correction;
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 MATHLAB 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 splited 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,)`

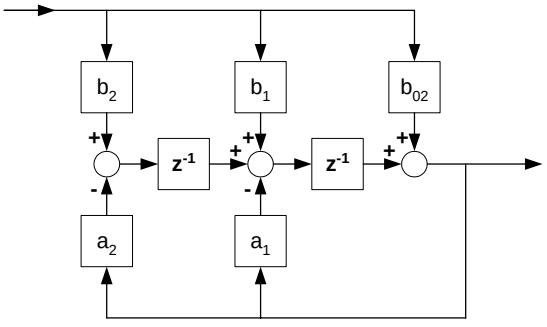


Figure 5.38.: *Block diagram transposed filter structure*

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 ω_M. ω_M depends on the FM carrier angular frequency ω_T and the sample frequency ω_A (see Eq.5.1). Because the FM modulation is done with analog components ω_T is not a constant, it fluctuates. Thus ω_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 prefered 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

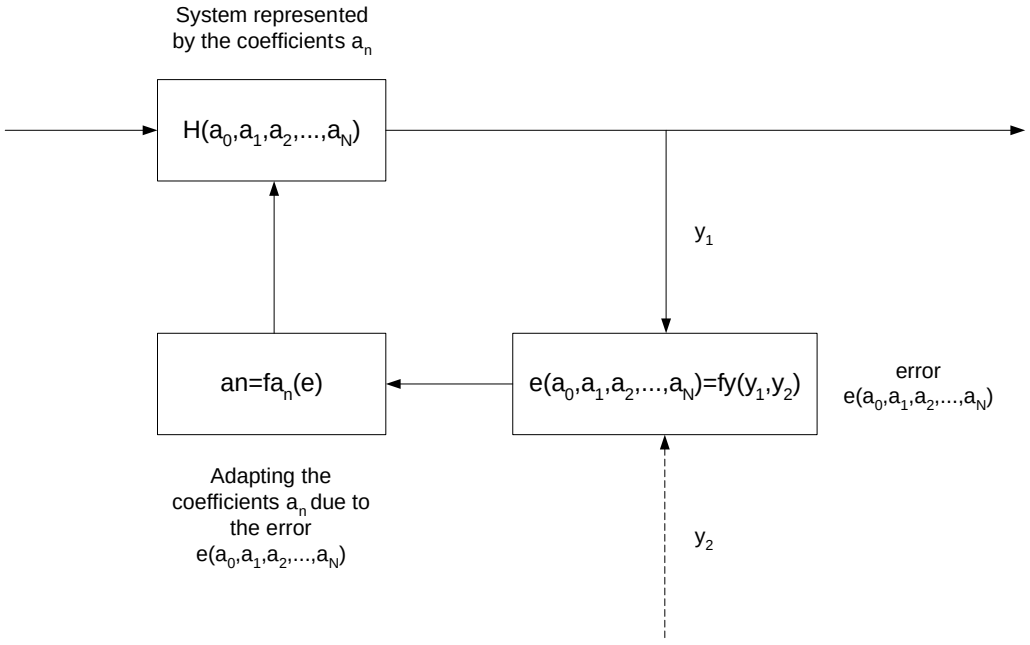


Figure 5.39.: *Adapted System*

the DC offset error results because of the inexact ω_M. ω_{Me} is the actual angular frequency and ω_{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 ω_{Ms}:

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

Therefore ω_{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 lager 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.

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

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

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.

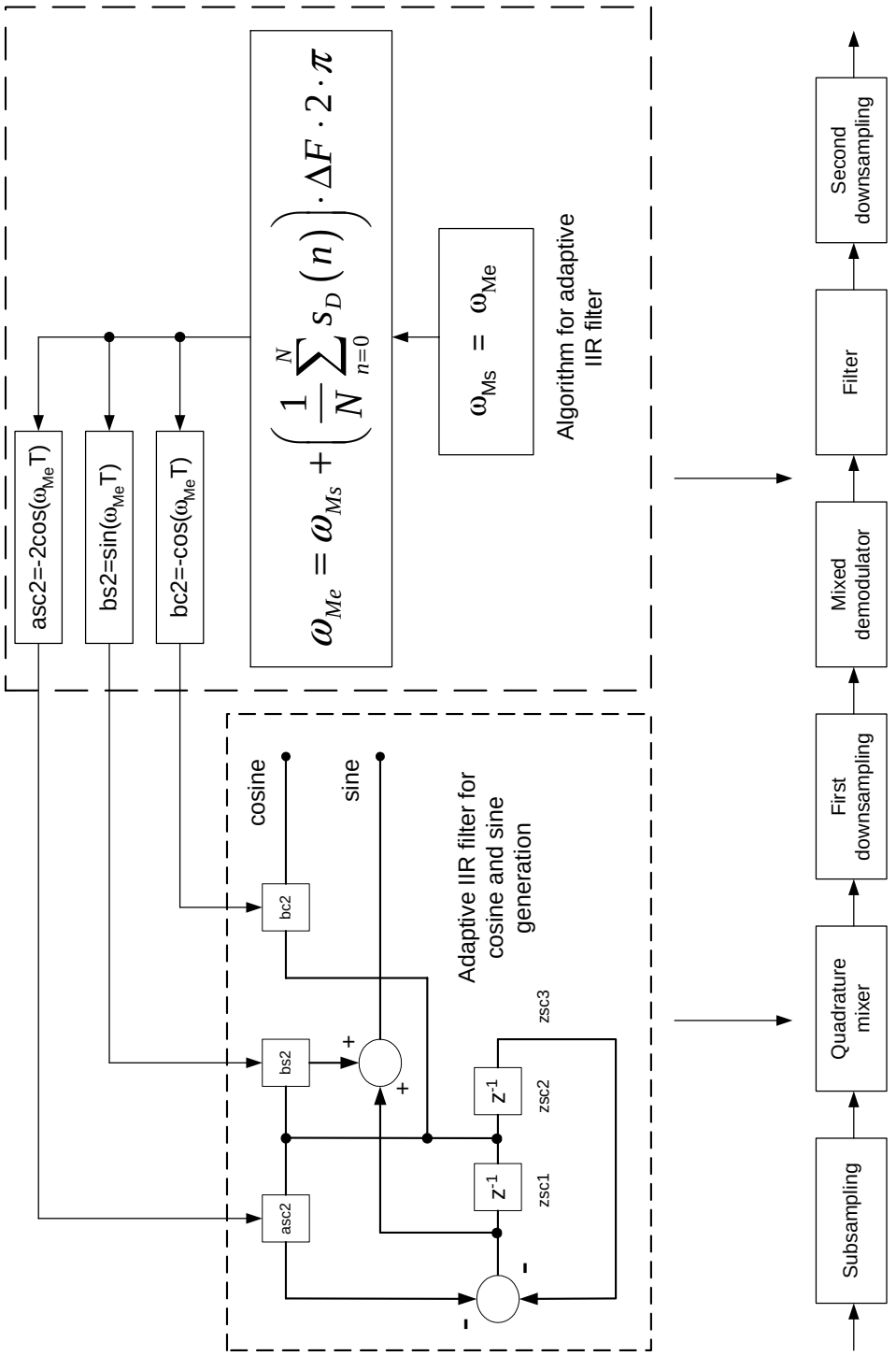


Figure 5.40.: *Adaptive seeking of carrier angular frequency*

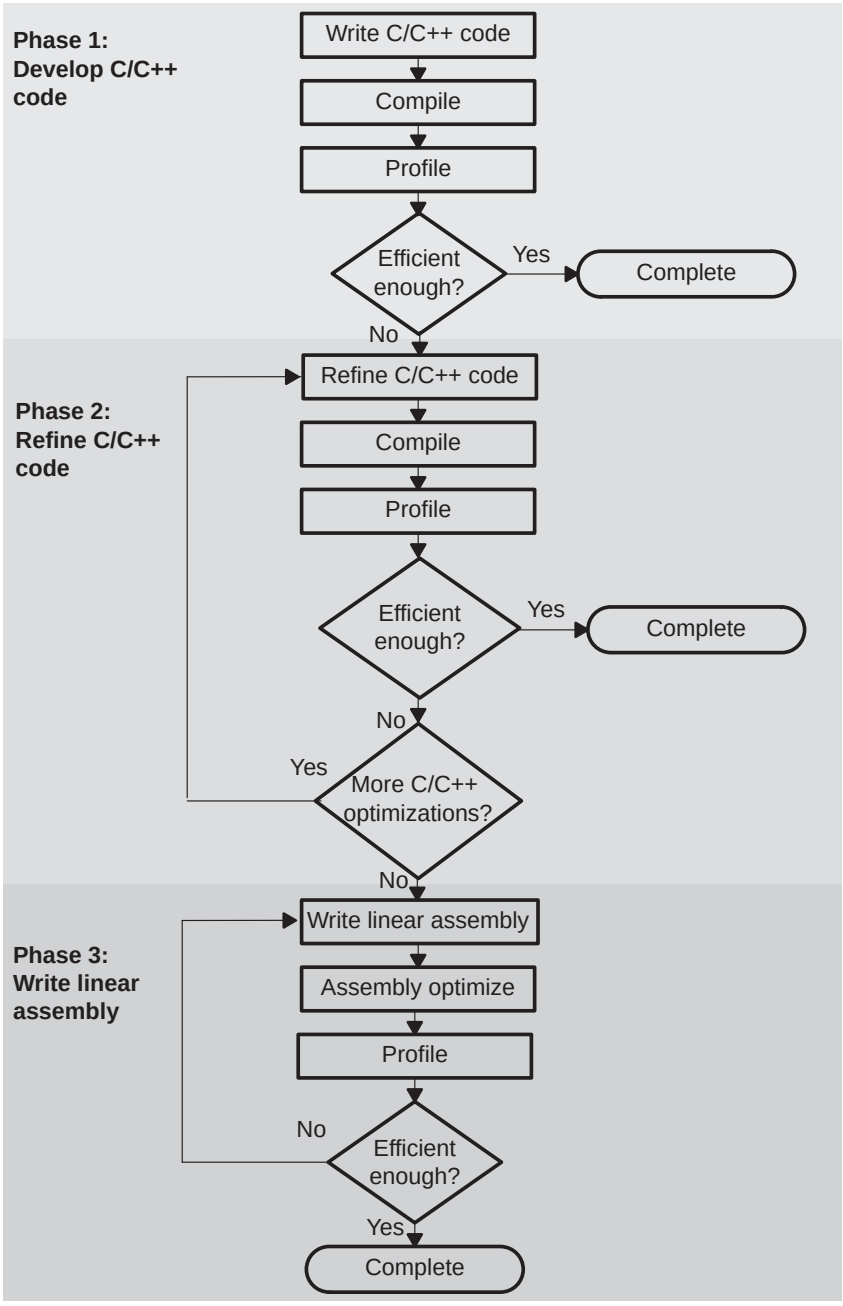


Figure 5.41.: *development flow*

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[BUFFERSIZE_INPUT];
#pragma DATA_MEM_BANK (in_buffer_B, 0);
short in_buffer_B[BUFFERSIZE_INPUT];

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

#pragma DATA_MEM_BANK (out_buffer, 0);
short out_buffer[BUFFERSIZE_DEMODULATOR];
#pragma DATA_MEM_BANK (out_buffer_A, 6);
short out_buffer_A[BUFFERSIZE_OUTPUT];
#pragma DATA_MEM_BANK (out_buffer_B, 6);
short out_buffer_B[BUFFERSIZE_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, Section6.4 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.

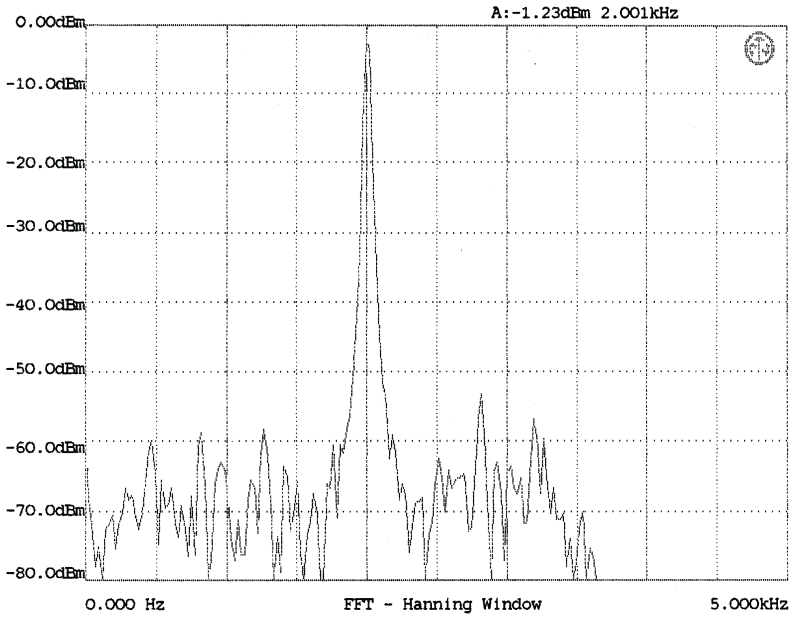


Figure 6.1.: *FFT print of demodulated signal*

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.

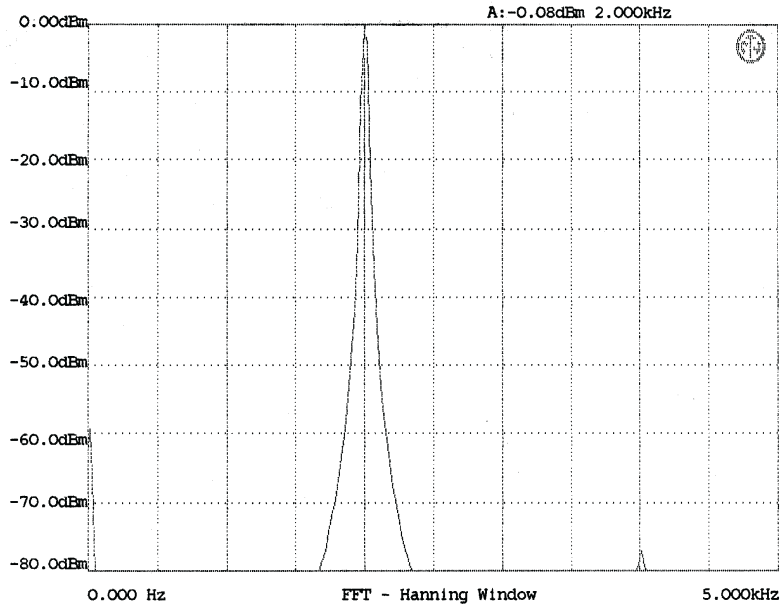


Figure 6.2.: *FFT print of DSP generated sine wave*

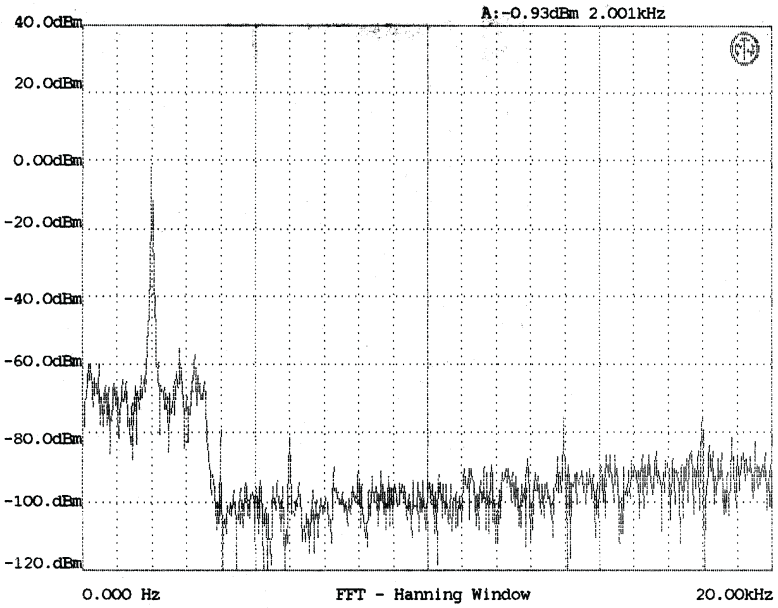


Figure 6.3.: *FFT print of demodulated signal for the range of 0 to 20kHz*

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.

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

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

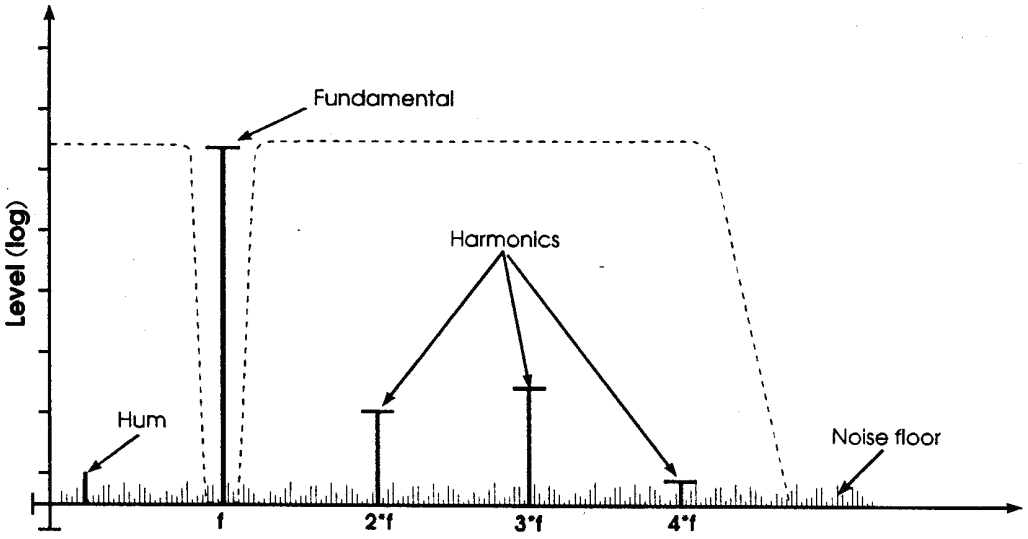


Figure 6.4.: *Principle of THD+N Measurement*

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 deviationΔF	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

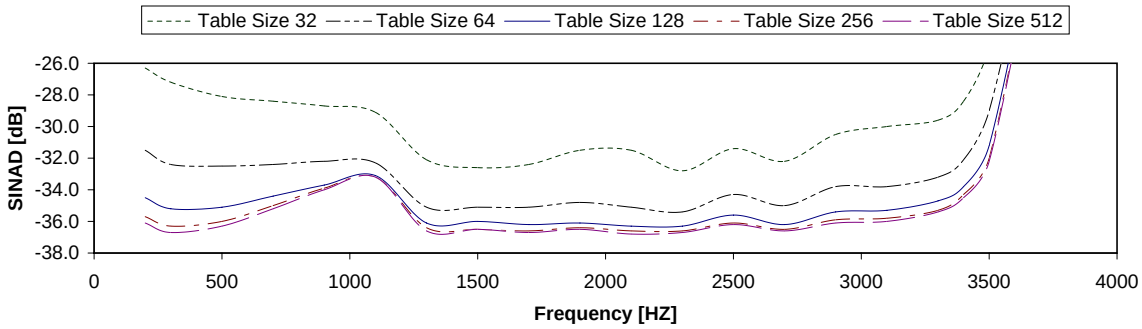


Figure 6.5.: *SINAD of fixed-point mixed demodulator with various table sizes for direct lookup*

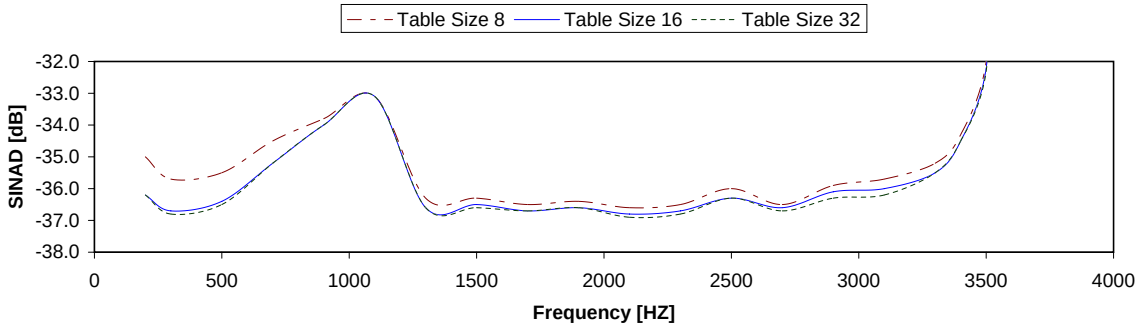


Figure 6.6.: *SINAD of fixed-point mixed demodulator with various table sizes for interpolated lookup*

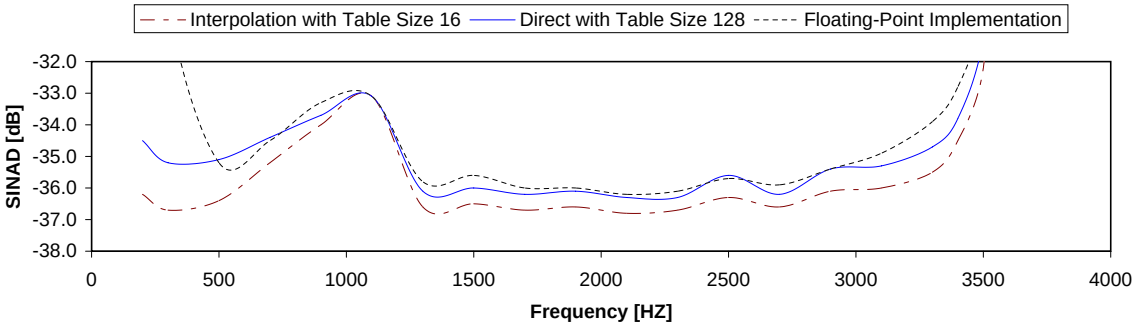


Figure 6.7.: *SINAD comparison of direct and interpolated lookup for fixed-point mixed demodulator and the floating-point implementation*

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 deviationΔF	3000 Hz
Message signal frequency	100-3700 Hz

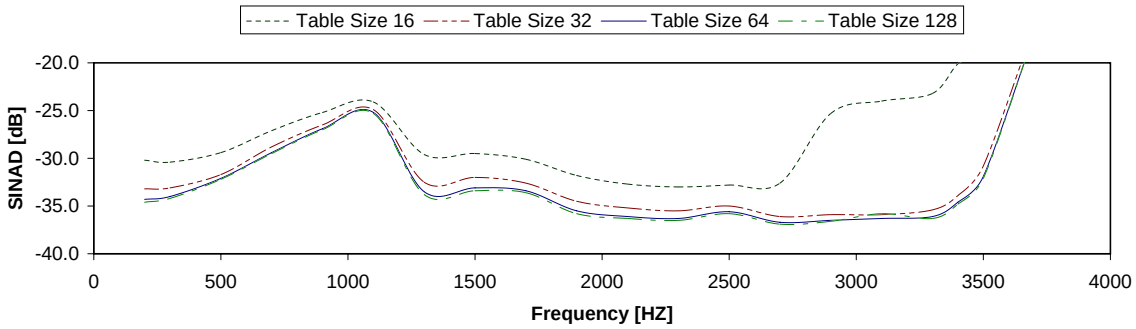


Figure 6.8.: *SINAD of fixed-point PLL demodulator with various table sizes for direct lookup*

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

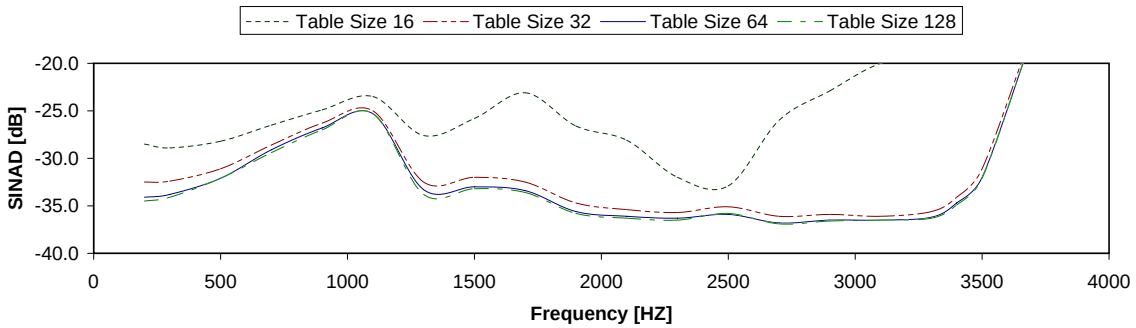


Figure 6.9.: *SINAD of fixed-point PLL demodulator with various table sizes for interpolated lookup*

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,

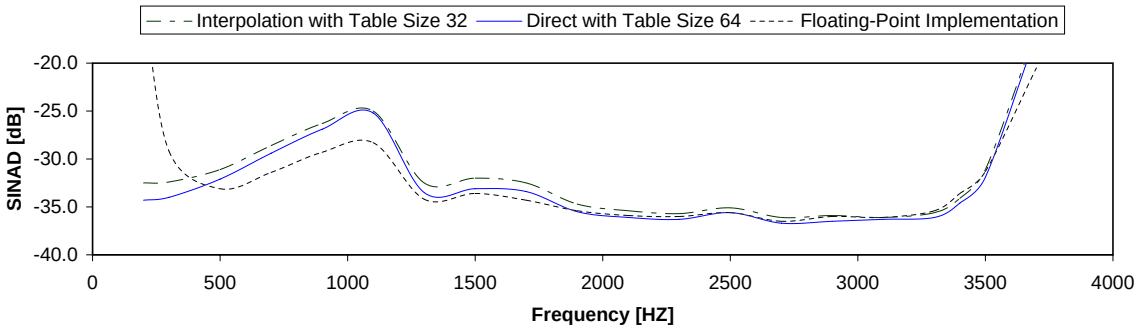


Figure 6.10.: *SINAD comparison of direct and interpolated lookup for fixed-point PLL demodulator and the floating-point implementation*

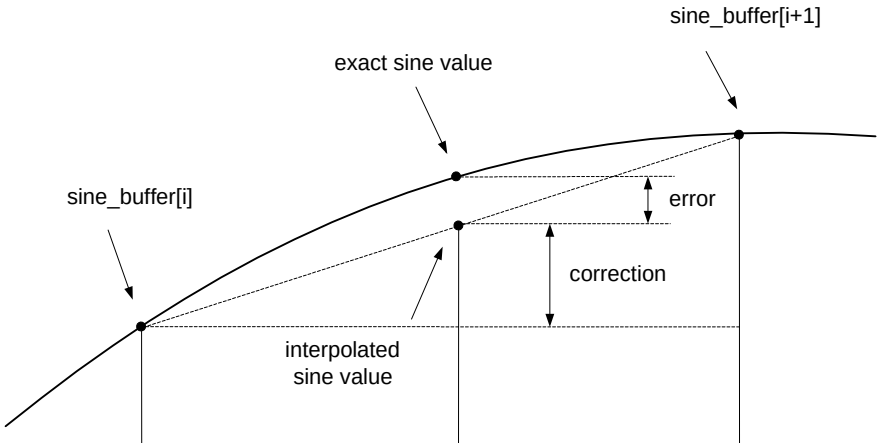


Figure 6.11.: *Error of linear interpolation*

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.

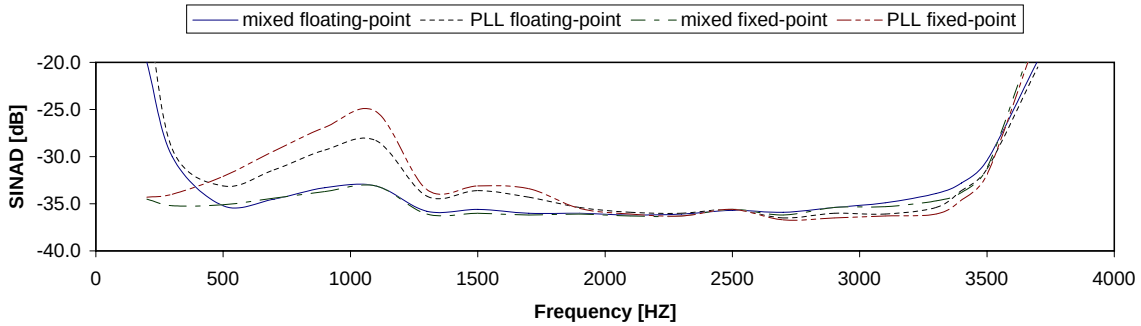


Figure 6.12.: *SINAD comparison of different demodulator implementations*

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 generater.
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.

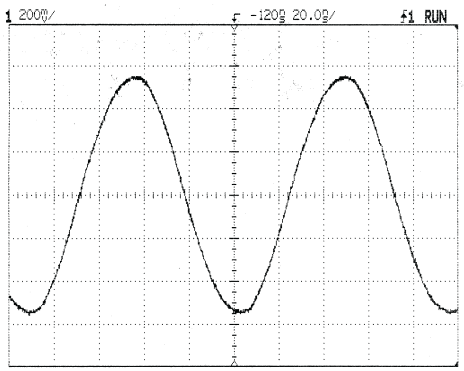


Figure 6.13.: *Oscilloscope print FM signal normal*

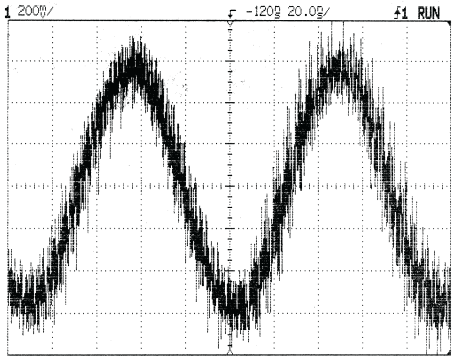


Figure 6.14.: *Oscilloscope print FM signal with noise added*

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.

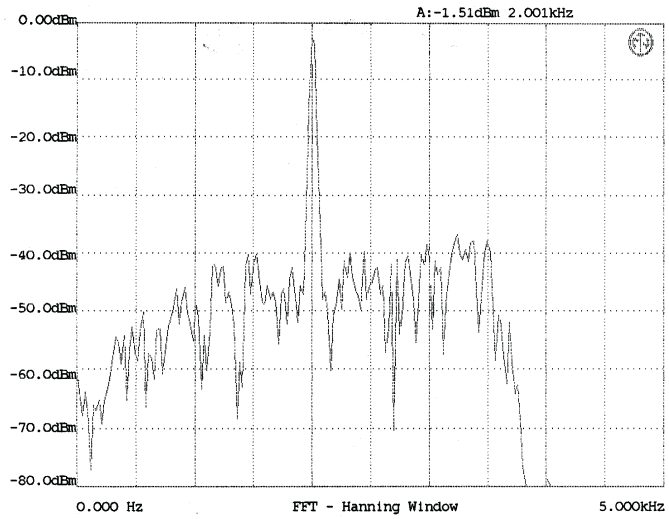


Figure 6.15.: *Spectrum of an demodulated signal with an S/N ratio of 20dB by the FM signal*

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

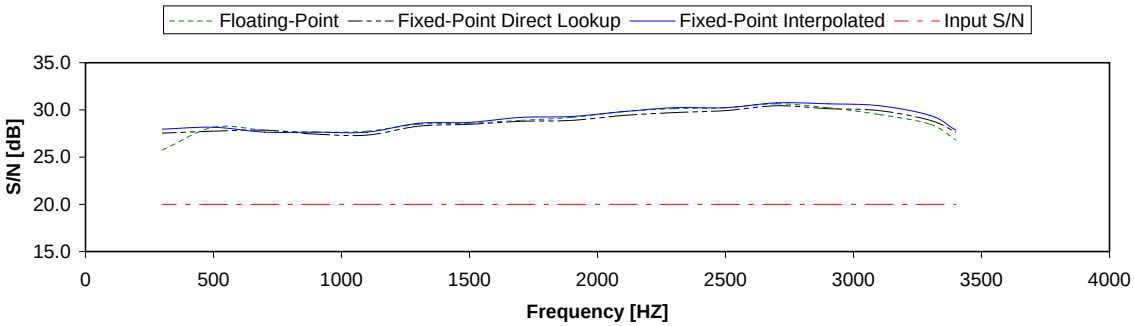


Figure 6.16.: *S/N of the mixed demodulated signal with a S/N of 20dB at the input FM signal*

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

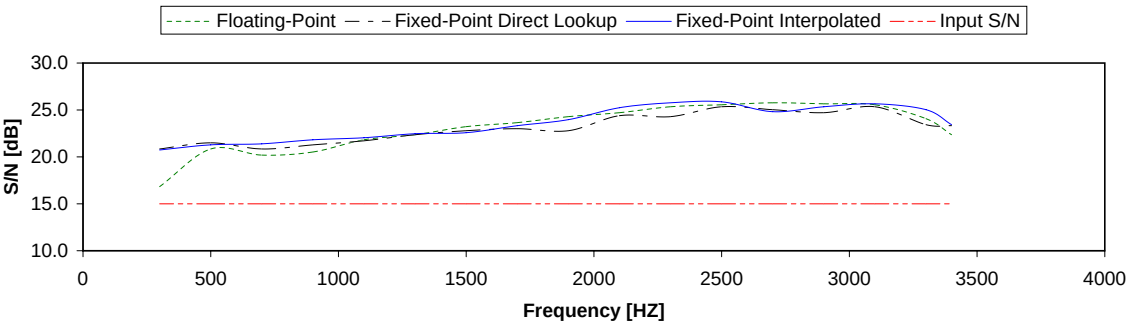


Figure 6.17.: *S/N of the mixed demodulated signal with a S/N of 15dB at the FM input signal*

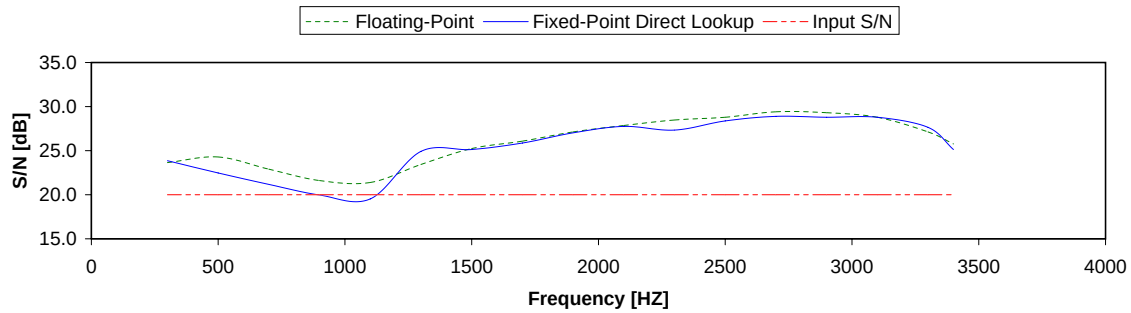


Figure 6.18.: *S/N of the PLL demodulated signal with a S/N of 20dB at the input FM signal*

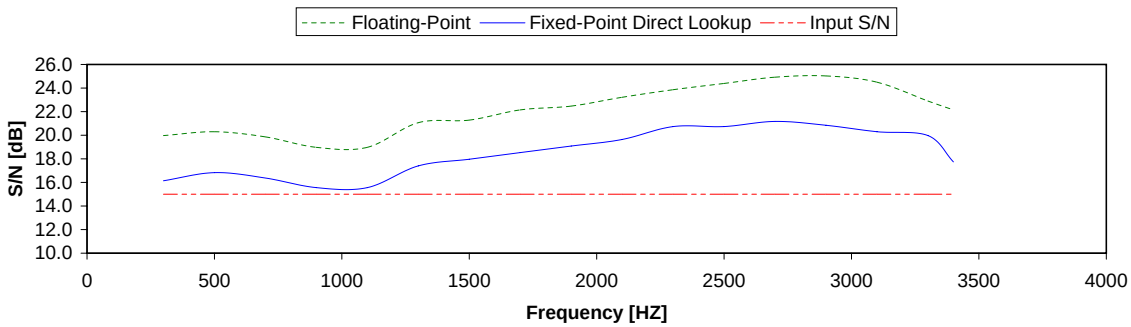


Figure 6.19.: *S/N of the PLL demodulated signal with a S/N of 15dB at the FM input signal*

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.

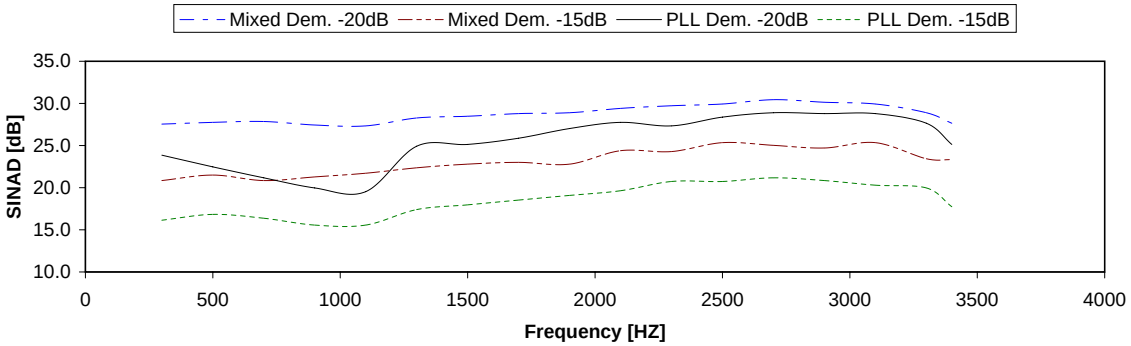


Figure 6.20.: *S/N of the PLL demodulated signal with a S/N of 15dB at the FM input signal*

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_gethtime` function of the DSP/BIOS. `CLK_gethtime` 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_gethtime());
#endif
quad_mix(in_buffer_ptr,re_buffer,im_buffer);
#if FM_DEM_CF_FUNC_TIME_STS
   STS_delta(&mix_sts,CLK_gethtime());
#endif

#if FM_DEM_CF_FUNC_TIME_STS
   STS_set(&dem_sts,CLK_gethtime());
#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_gethtime());
#endif

#if FM_DEM_CF_FUNC_TIME_STS
   STS_set(&filter_sts,CLK_gethtime());
#endif
out_filter(out_buffer,out_buffer_ptr);
#if FM_DEM_CF_FUNC_TIME_STS
   STS_delta(&filter_sts,CLK_gethtime());
#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).

-o0

- Performs control-flow-graph simplification
- Allocates variables to registers
- Performs loop rotation
- Eliminates unused code
- Simplifies expressions and statements
- Expands calls to functions declared inline

-o1 Performs all -o0 optimizations, and:

- Performs local copy/constant propagation
- Removes unused assignments
- Eliminates local common expressions

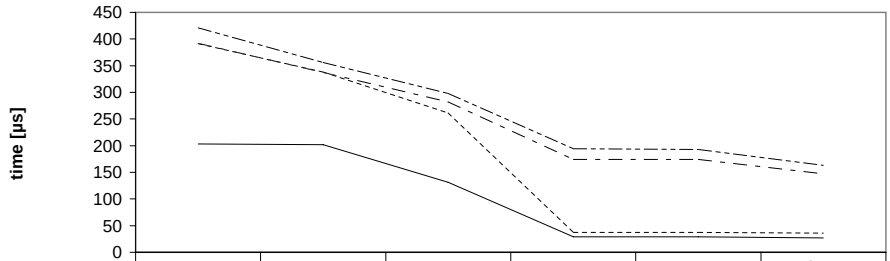
-o2 Performs all -o1 optimizations, and:

- Performs software pipelining
- Performs loop optimizations
- Eliminates global common subexpressions
- Eliminates global unused assignments
- Converts array references in loops to incremented pointer form
- Performs loop unrolling

-o3 Performs all -o2 optimizations, and:

- Removes all functions that are never called
- Simplifies functions with return values that are never used
- Inlines calls to small functions
- Reorders function declarations so that the attributes of called functions are known when the caller is optimized
- Propagates arguments into function bodies when all calls pass the same value in the same argument position
- Identifies file-level variable characteristics

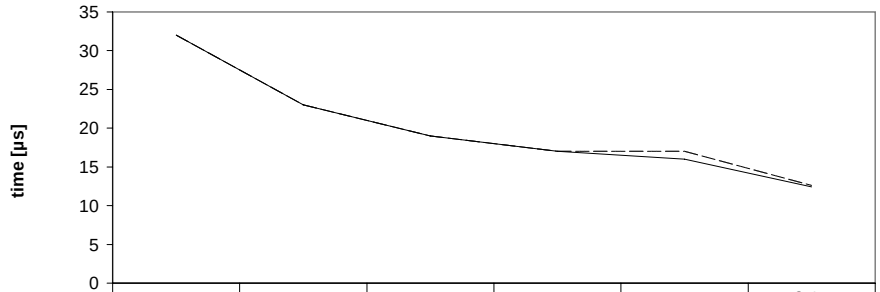
The results are shown in Figures 6.21 to 6.25. It shows that the demodulation functions did not improve as much as the quadrature mixer and out filter functions. In the mixed demodulator this is caused by the division which is as a function call. Therefore a branch occurs in the loop and the optimizer 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.



compiler optimization level	none	-o0	-o1	-o2	-o3	-o3 & no Debug
simple C impl. (Phase 1)	421	356	298	194	193	163.4
optimized C (Phase2 Step1)	392	337	282	174	174	147.4
comp. spez. C (Phase2 Step2)	391	338	261	37	37	35.9
Linear Assembly (Phase3)	203	202	131	29	29	27.1

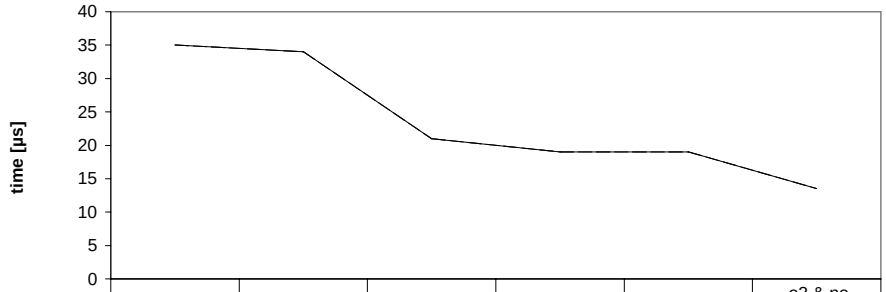
Figure 6.21.: *Time measurement results Quadrature mixer (quad_mix)*

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.



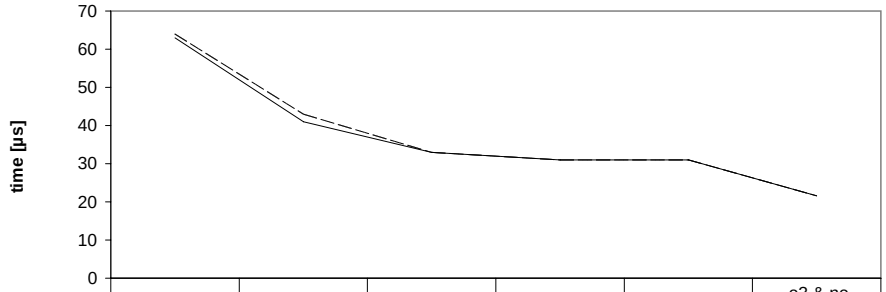
compiler optimization level	none	-o0	-o1	-o2	-o3	-o3 & no Debug
---- simple C impl. (Phase 1)	32	23	19	17	17	12.6
- - - optimized C (Phase2 Step1)	32	23	19	17	17	12.6
—— comp. spez. C (Phase2 Step2)	32	23	19	17	16	12.4

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



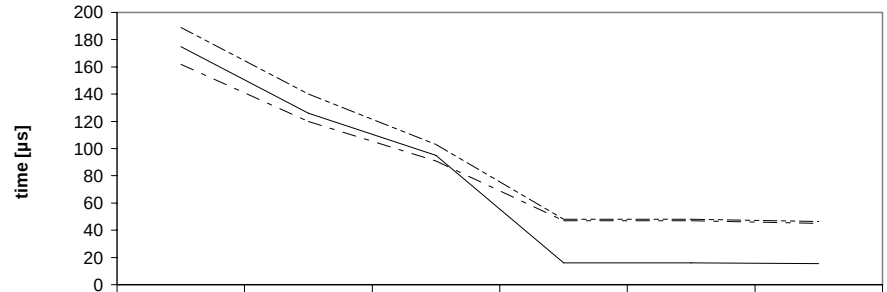
compiler optimization level	none	-o0	-o1	-o2	-o3	-o3 & no Debug
---- simple C impl. (Phase 1)	35	34	21	19	19	13.5
- - - optimized C (Phase2 Step1)	35	34	21	19	19	13.5
—— comp. spez. C (Phase2 Step2)	35	34	21	19	19	13.5

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



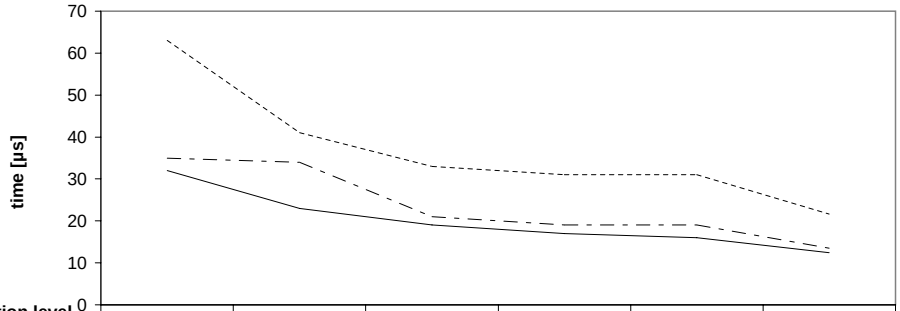
compiler optimization level	none	-o0	-o1	-o2	-o3	-o3 & no Debug
---- simple C impl. (Phase 1)	64	43	33	31	31	21.6
- - - optimized C (Phase2 Step1)	64	43	33	31	31	21.6
—— comp. spez. C (Phase2 Step2)	63	41	33	31	31	21.6

Figure 6.24.: *Time measurement results PLL demodulator direct lookup (pll_demodulate)*



compiler optimization level	none	-o0	-o1	-o2	-o3	-o3 & no Debug
simple C impl. (Phase 1)	189	140	103	48	48	46.4
optimized C (Phase2 Step1)	162	120	91	47	47	45.1
comp. spez. C (Phase2 Step2)	175	126	95	16	16	15.6

Figure 6.25.: *Time measurement results out filter (out_filter)*



compiler optimization level	none	-o0	-o1	-o2	-o3	-o3 & no Debug
mixed_demodulate	32	23	19	17	16	12.4
mixed_demodulate (inter.)	35	34	21	19	19	13.5
pll_demodulate	63	41	33	31	31	21.6

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.

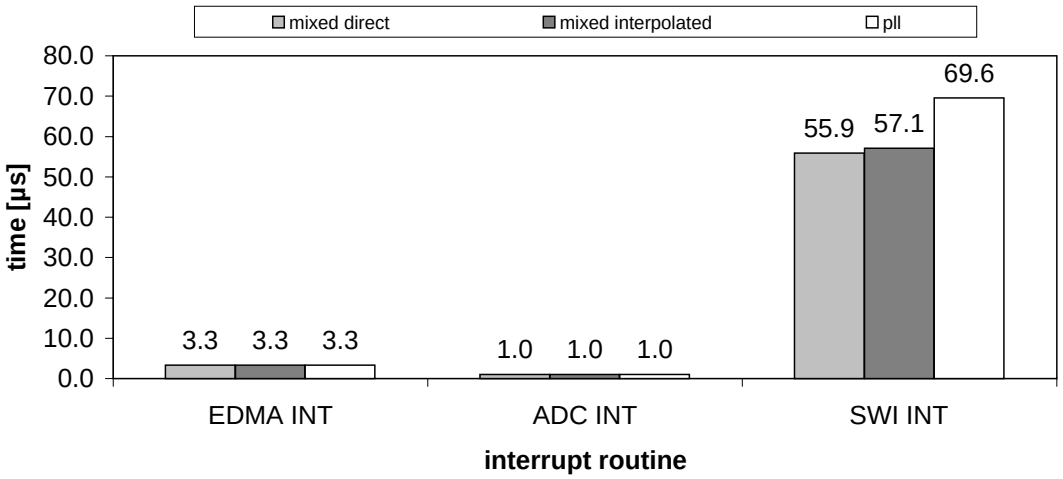


Figure 6.27.: *Executing time of interrupt routines*

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%.

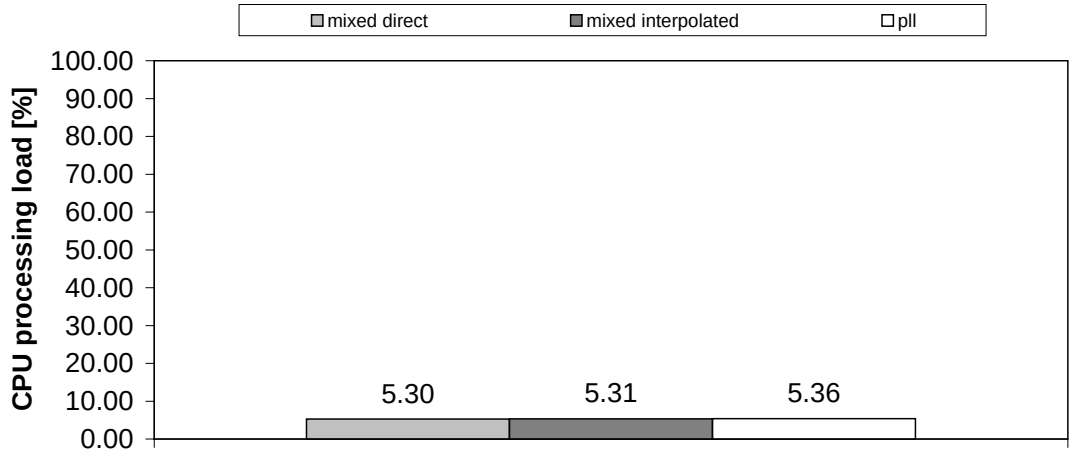


Figure 6.28.: *CPU processing load*

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 excepted. 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.

`Singapore 11.1.2002` Christoph Haller Franz Schnyder

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
ΔF	frequency derivation	
ϕ_{FM}	FM argument function	
ω_T	carrier frequency	rad/sec
ω_N	message frequency	rad/sec
μ	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 Inpulse 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 Inpulse 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:

$$k = \sqrt{\frac{A_{2\cdot f}^2 + A_{3\cdot f}^2 + A_{4\cdot f}^2 \cdots}{A_f^2 + A_{2\cdot f}^2 + A_{3\cdot f}^2 + A_{4\cdot f}^2 \cdots}} \tag{C.1}$$

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

$$k = \sqrt{\frac{P_{tot} - P_f}{P_{tot}}} \tag{C.2}$$

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:

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

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=fir1(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 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=fir1(N,[(f0(1))/fN (f0(2))/fN],kaiser(N+1,8));
else
   B=fir1(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 ® Dynamic System Simulation for Matlab ® (Version 4)*, MathWorks, Natick November 2000.

D.2. German Books

[4] P. Gerdsen 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 UserŠs Guide* Texas Instruments , April 2001.

[18] SPRU432 , *TMS320C6000 DSK Board Support Library API UserŠs Guide* Texas Instruments , October 2000.

E. Simulink models

E.1. Baseband-Delaydemodulator

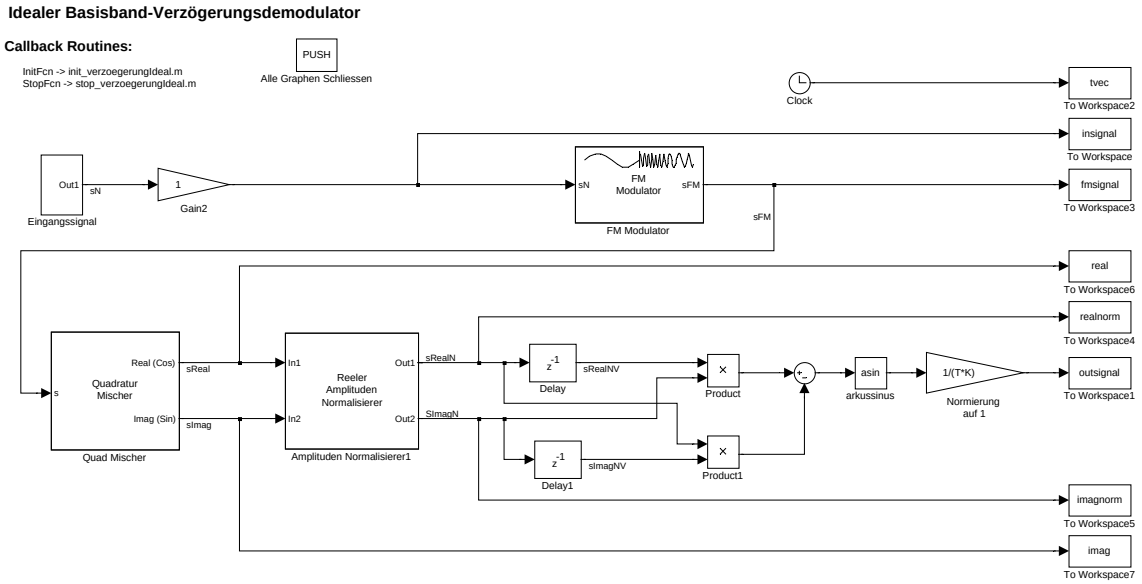


Figure E.1.: *Blockschaltbild Simulink Aufbau Idealer Baseband-Delaydemodulator*

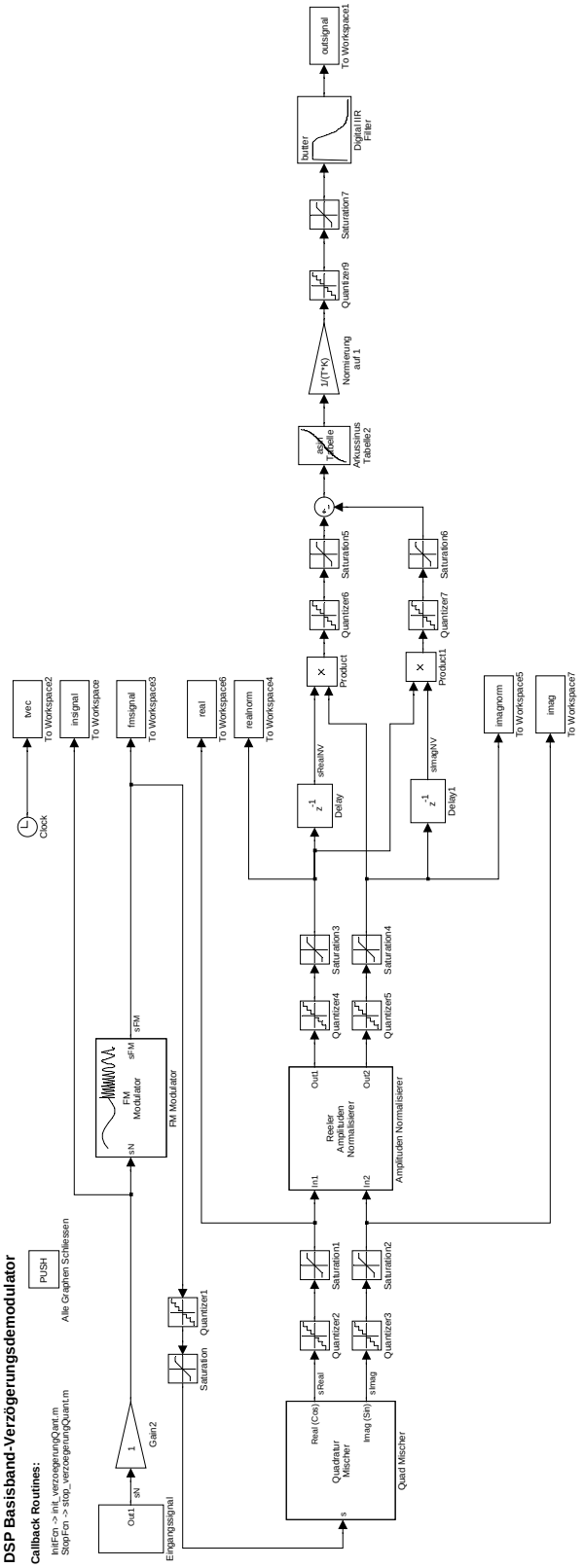


Figure E.2.: *Blockschaltbild Simulink Aufbau DSP Baseband-Delaydemodulator*

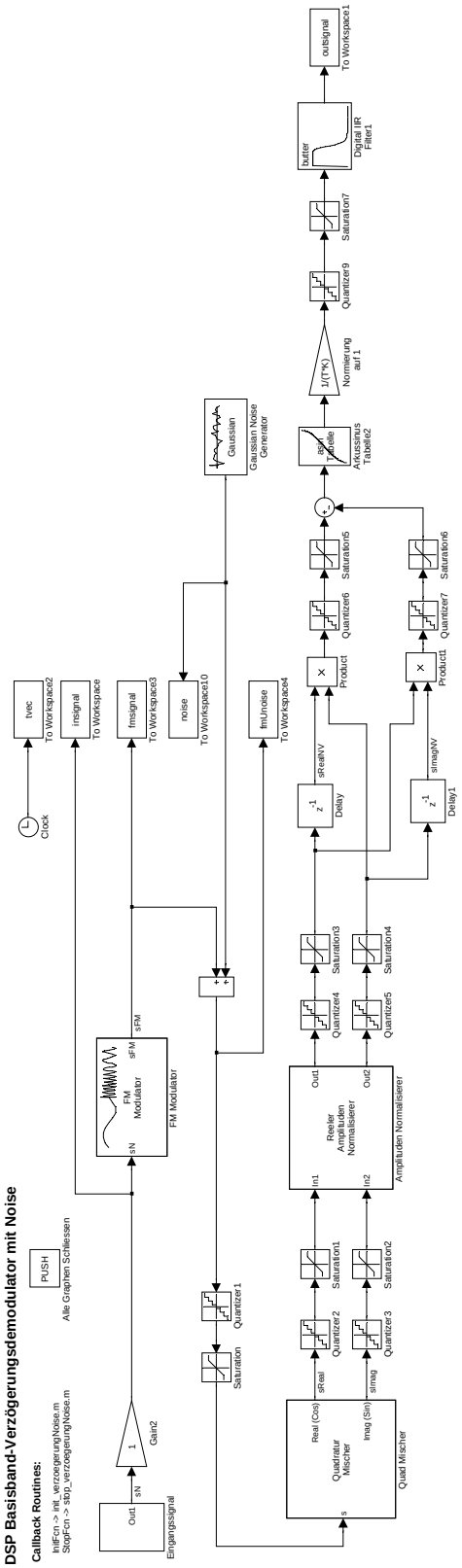


Figure E.3.: *Blockschaltbild Simulink Aufbau DSP Baseband-Delaydemodulator mit gausschem Rauschen*

E.2. Phase-Adapter-Demodulator

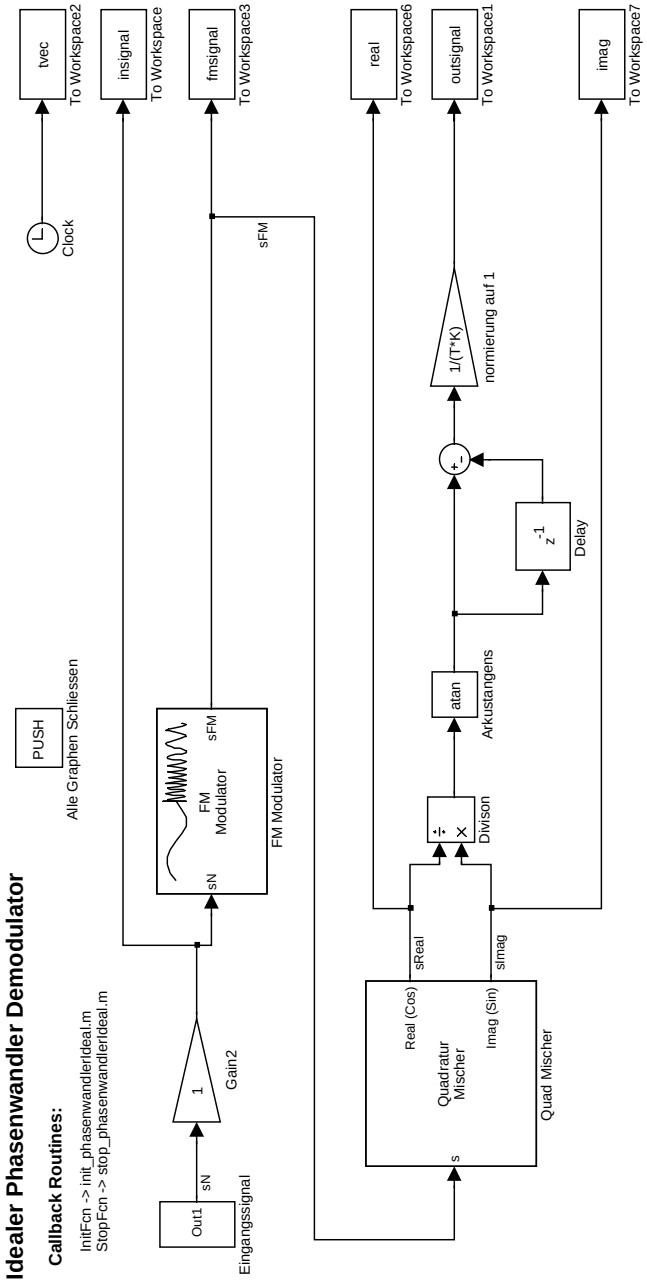


Figure E.4.: *Blockschaltbild Simulink Aufbau Idealer Phase-Adapter-Demodulator*

E.3. Phasenregelschleife

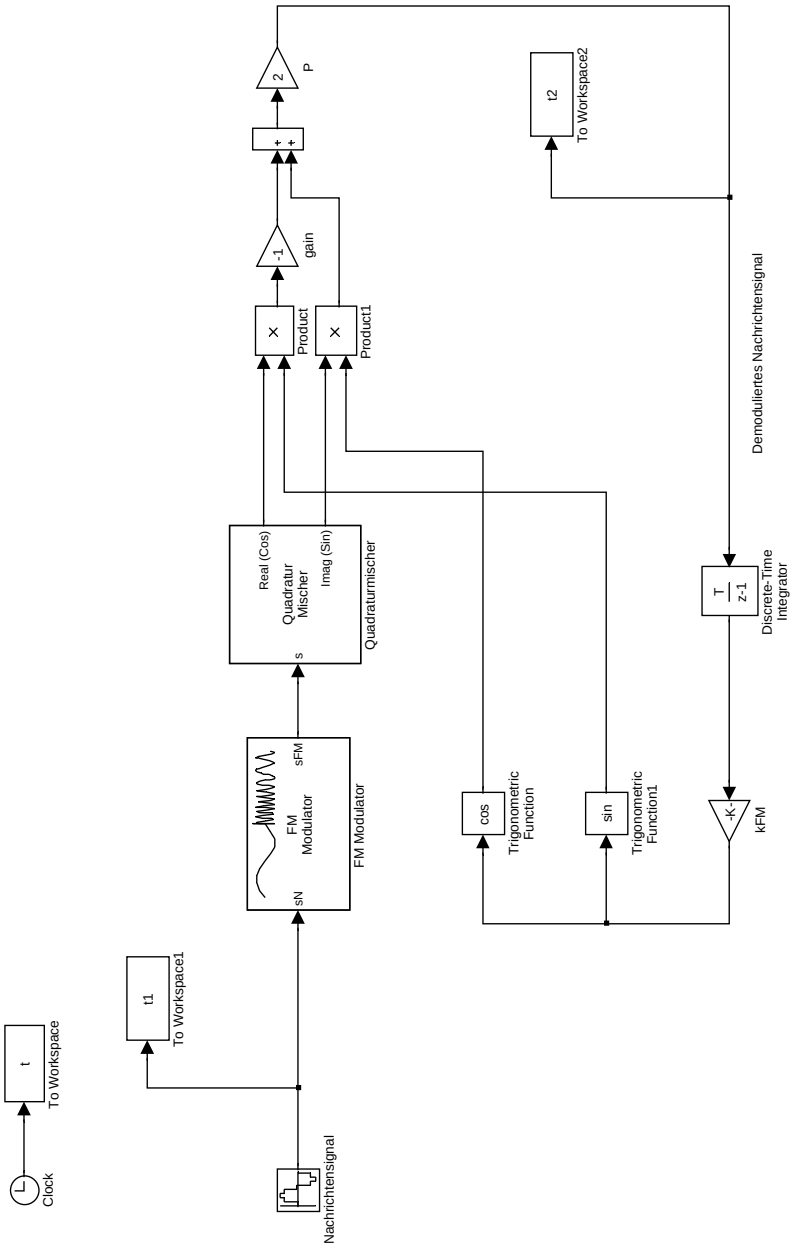


Figure E.5.: *Blockschaltbild Simulink Aufbau Idealer PLL*

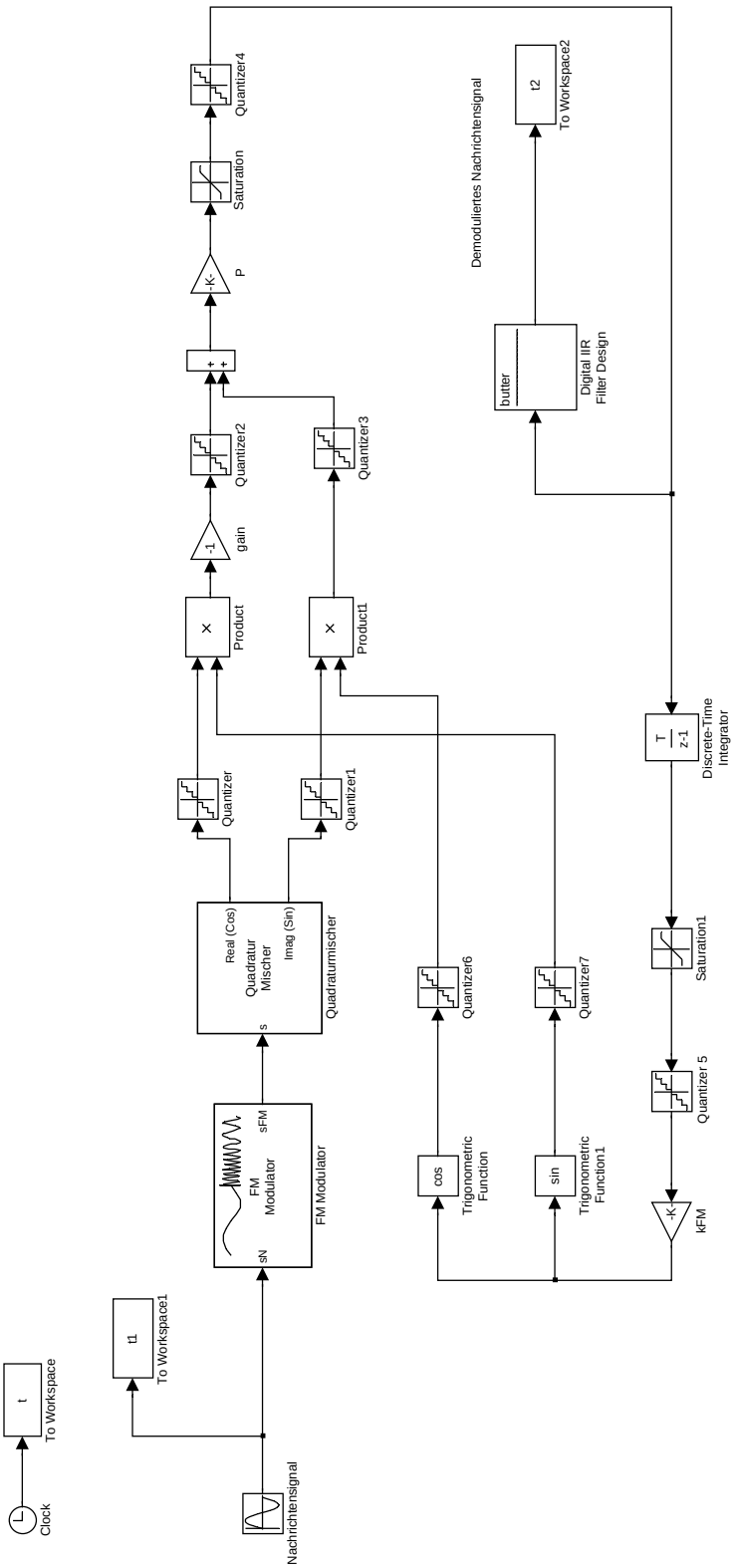


Figure E.6.: *Blockschaltbild Simulink Aufbau DSP PLL*

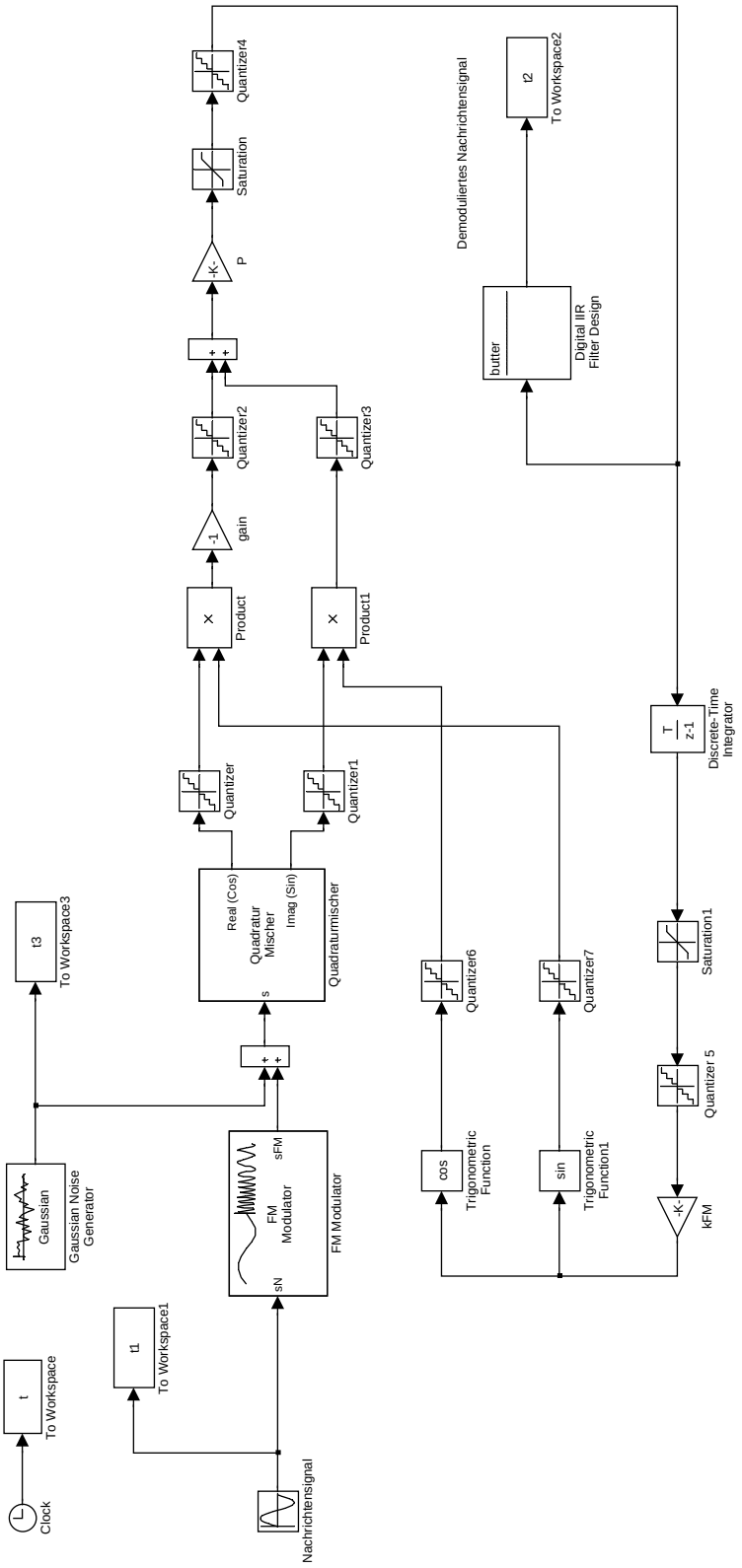


Figure E.7.: *Blockschaltbild Simulink Aufbau DSP PLL mit gausschem Rauschen*

E.4. Mix Demodulator

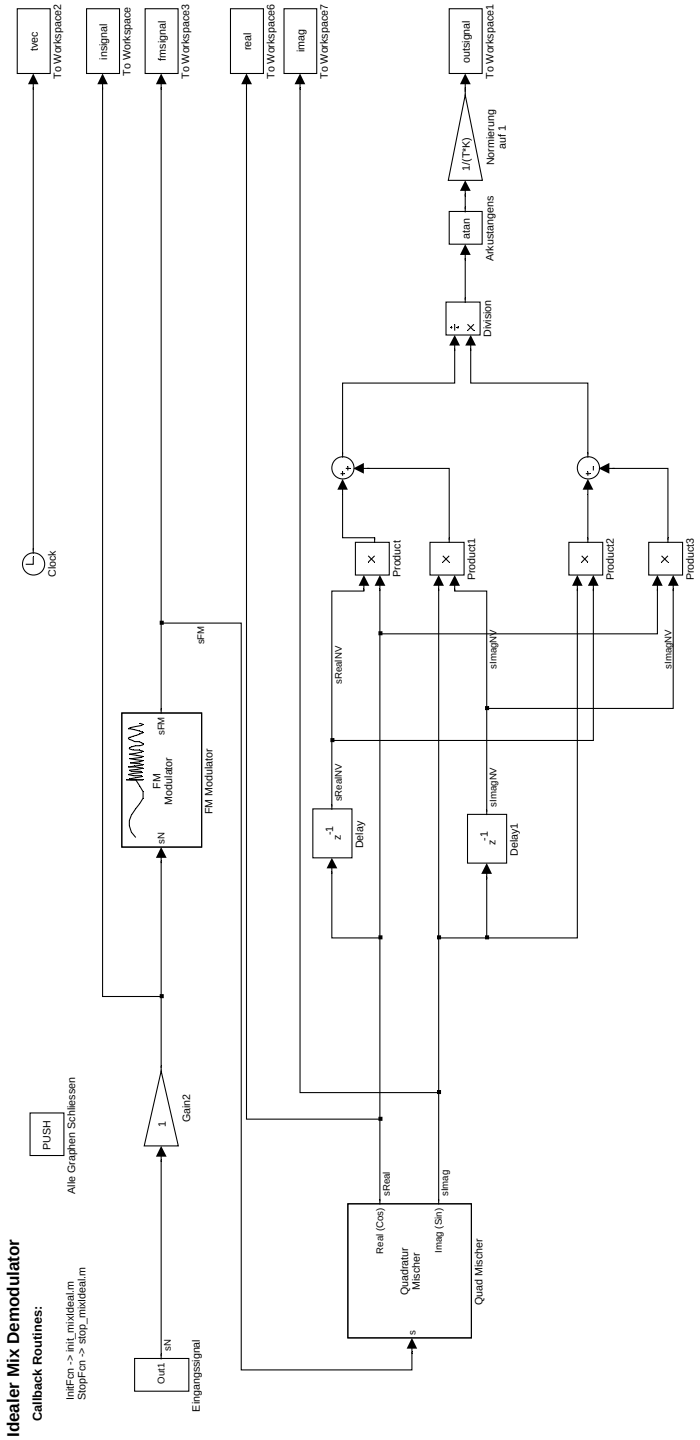


Figure E.8.: *Blockschaltbild Simulink Aufbau Idealer Mix Demodulator*

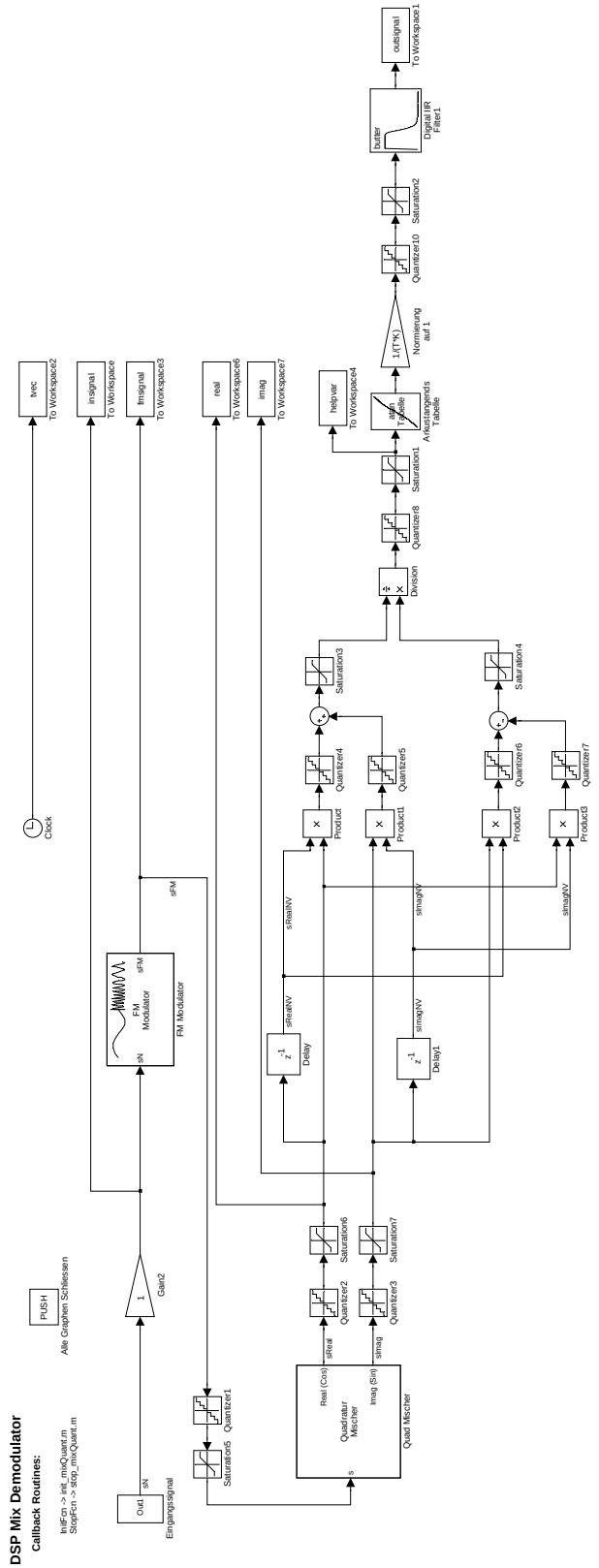


Figure E.9.: *Blockschaltbild Simulink Aufbau DSP Mix Demodulator*

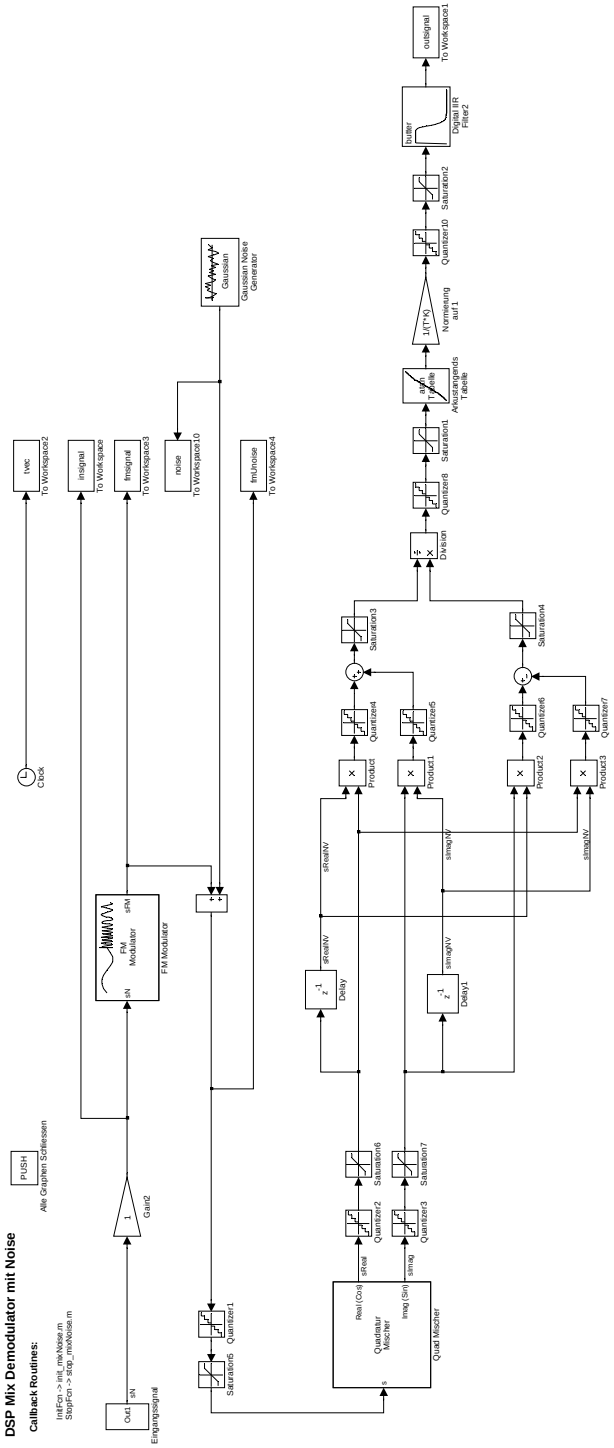


Figure E.10.: *Blockschaltbild Simulink Aufbau DSP Mix Demodulator mit gausschem Rauschen*

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*

```
/*_____________________________________________________________________
   ANALOG DIGITAL CONVERTER THS1408 - VERSION 1

   file  :  adc_THS1408.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
  _____________________________________________________________________
*/

#ifndef _ADC_THS1408_
#define _ADC_THS1408_

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/

   #include <csl_timer.h>
   #include <csl_emif.h>
   #include <csl_irq.h>
   #include <csl_edma.h>

   #include "global_settings.h"

/*---------------------------------------------------------------------
   debug defines
  ---------------------------------------------------------------------*/
   #define THS1408_ENABLE_SET_FREQ  0
      /* enable or disable the function ths1408_setFreq */

/*---------------------------------------------------------------------
   defines EDMA TCC for the two buffers
  ---------------------------------------------------------------------*/
   #define THS1408_BUFFER_A_TCC  (4)  /* */
   #define THS1408_BUFFER_B_TCC  (5)  /* */

/*---------------------------------------------------------------------
   defines for Control Register of the THS1408 (refer SLAS248)
  ---------------------------------------------------------------------*/
   /* Power Down D13    */
   #define THS1408_PWD_NORMAL (0)  /* normal operation  */
   #define THS1408_PWD_DOWN  (1)  /* power down    */
   /* Reference Select D12  */
   #define THS1408_REF_INTERNAL (0)  /* internal reference */
   #define THS1408_REF_EXTERNAL (1)  /* external reference */
   /* Output Format D11  */
   #define THS1408_FOR_STRAIGHT (0)  /* straight binary   */
   #define THS1408_FOR_2SCOMPL (1)  /* 2s complement  */
   /* Test Mode D8-D10   */
   #define THS1408_TM_NORMAL (0x0) /* normal operation  */
   #define THS1408_TM_REFN_REFN (0x1) /* both inputs=REF-  */
   #define THS1408_TM_VREF_REFN (0x2) /* IN+=Vref/2,IN-=REF- */
   #define THS1408_TM_REFP_REFN (0x3) /* IN+=REF+,IN-=REF- */
   #define THS1408_TM_NORMAL2 (0x4) /* normal operation  */
   #define THS1408_TM_REFP_REFP (0x5) /* both inputs=REF+  */
   #define THS1408_TM_REFN_VREF (0x6) /* IN+=REF-,IN-=Vref/2 */
   #define THS1408_TM_REFN_REFP (0x7) /* IN+=REF-,IN-=REF+ */
   /* Offset correction D7  */
   #define THS1408_OF_ENABLE (0)  /* enable        */
   #define THS1408_OF_DISABLE (1)  /* disable      */
   /* Reserved Fields D0-D6  */
   #define THS1408_LSB_RESERVED (0x00) /* zeroes for RES Field */
   /* Reserved Fileds D14-D15 */
   #define THS1408_MSB_RESERVED (0x0) /* zeroes for RES Field */

/*---------------------------------------------------------------------
   Parameters for the THS1408EVM Analog Digital Converter
   to TMS320C6711
  ---------------------------------------------------------------------*/
   /*-- Handle parameters --*/
   #define THS1408_RES_ADDR (0xA0000040) /* adress of RES register  */
   #define THS1408_PGA_ADDR (0xA0000044) /* adress of PGA register  */
   #define THS1408_OFF_ADDR (0xA0000048) /* adress of OFF register  */
   #define THS1408_CTL_ADDR (0xA000004C) /* adress of CTL register  */
   /*-- EMIF Interface parameters --*/ /*Org*/
   #define THS1408_RDSETUP  (1)      /* read             */
   #define THS1408_RDSTRB   (3)
   #define THS1408_RDHLD    (1)
   #define THS1408_WRSETUP  (4)      /* write            */
   #define THS1408_WRHLD    (6)
   #define THS1408_WRSTRB   (3)
   /*-- Global Parameters */
   #define THS1408_DSP_TYPE TMS320C6711 /* DSP Type      */
   #define THS1408_DSP_FREQ (150)    /* in MHz           */

/*---------------------------------------------------------------------
   Default Initial Parameters
  ---------------------------------------------------------------------*/
   /* Sample Frequency  */
   #define THS1408_SAMPLE_FREQ (64)   /* in kHz          */
                                      /*   DSP_FREQ      */
   #define THS1408_TIM_PERIOD (0x125) /* =------------   */
                                      /* 8*SAMPLE_FREQ    */
   /* Register Parameters */
    /* CTL */
   #define THS1408_POW  THS1408_PWD_NORMAL /* normal operation */
   #define THS1408_REF  THS1408_REF_INTERNAL /* internal reference  */
   #define THS1408_FORMAT THS1408_FOR_2SCOMPL /* 2s complement  */
   #define THS1408_TM   THS1408_TM_NORMAL /* normal operation */
   #define THS1408_OFF  THS1408_OF_ENABLE /* enable      */
    /* PGA */
   #define THS1408_PGA      (0x0)       /* gain of 7dB   */
    /*- OFF -*/
   #define THS1408_OFFSET   (0)         /* no offset     */

/*---------------------------------------------------------------------
   structs and unions
  ---------------------------------------------------------------------*/
   typedef union
   {
      unsigned short value;
      struct
      {
       unsigned int lsb_reserved  :7;
       unsigned int off       :1;
       unsigned int tm        :3;
       unsigned int format    :1;
       unsigned int ref       :1;
       unsigned int pwd       :1;
       unsigned int msb_reserved :2;
      } control_bit;
```

```
   } THS1408_CTRL;  /* the Control Register of ADC (refer SLAS248 P.15)*/

   typedef struct
   {
      unsigned short pga;
      unsigned short offset;
      THS1408_CTRL ctrl;
      Uint32      tim_period;
   } THS1408_CONFIG; /* Configuration Struct*/

   typedef struct
   {
      volatile short *const res_addr;
      volatile short *const pga_addr;
      volatile short *const off_addr;
      volatile short *const ctl_addr;
   } THS1408_HANDLE; /* Handle to the ADC */

/*---------------------------------------------------------------------
   global variables ( extern defined in .c )
  ---------------------------------------------------------------------*/
   extern THS1408_HANDLE hThs1408;  /* handle to the AD Converter */
   extern TIMER_Handle hTimer0;  /* handle to the Timer 0 */
   extern EDMA_Handle hEdmaIn; /* handle to the Input EDMA */

/*---------------------------------------------------------------------
   global functions
  ---------------------------------------------------------------------*/
   /*
      Notice:
      =======

      - The function 'CSL_init()' must have been called before any
        of the following functions can be called.

      - First call 'ths1408_init' before using the other functions

   */

   /*
      Initialize the THS1408:
         Sets the values for the EMIF and the Timer according
         to the Default Initial Parameters.
         Sets up the EDMA for Ping-Pong-Buffering (inBufferA und
         inBufferB).
         Maps the EDMA interrupt and enables it.

      Parameters :
         inBufferA pointer to Ping Buffer
         inBufferB pointer to Pong Buffer
   */
   void  ths1408_init(short inBufferA[],short inBufferB[]);

   /*
      Changes the configuration of the THS1408:
         Sets the the THS1408 Registers and the Timer Period
         according to the config struct.

      Parameters :
         config   new configuration calues
   */
   void  ths1408_config(THS1408_CONFIG config);

   /*
      Starts the THS1408:
         Starts the Timer 0 and enables the EDMA interrupt
   */
   void  ths1408_start();

   #if THS1408_ENABLE_SET_FREQ
      /*
         Changes the sample frequency of the THS1408:
            Sets a new Timer Period.
         !! Only available if THS1408_ENABLE_SET_FREQ=1 !!

         Parameters :
            freq     wanted frequency
         Return :
            (double) exact frequency of THS1408
      */
      double  ths1408_setFreq(double freq);
   #endif

#endif /* _ADC_THS1408_ */

/*_____________________________________________________________________
   END OF THE FILE adc_THS1408.h
  _____________________________________________________________________*/
```

Listing F.2: adc_THS1408.c

```
/*_____________________________________________________________________
   ANALOG DIGITAL CONVERTER THS1408 - VERSION 1

   file  :  adc_THS1408.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
  _____________________________________________________________________
*/

/*---------------------------------------------------------------------
   debug defines
  ---------------------------------------------------------------------*/
   #define THS1408_CF_ERROR_DISP 1
      /* Enable Error Log
        If enabled a 'error_log' LOG needs to be defined in the
        DSP/BIOS Config -> Log Manager
      */

   #if THS1408_CF_ERROR_DISP
      #include <std.h>
      #include <log.h>
      extern far LOG_Obj error_log;
   #endif

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "adc_THS1408.h"

/*---------------------------------------------------------------------
   local functions
  ---------------------------------------------------------------------*/
   void  emif_init_THS1408();
   void  timer_init_THS1408();
   void  adc_edma_init(short inBufferA_ptr[],short inBufferB_ptr[]);
   void  adc_irq_init();

/*---------------------------------------------------------------------
   global varibles
  ---------------------------------------------------------------------*/
   THS1408_HANDLE hThs1408={
            (volatile short *)THS1408_RES_ADDR,
            (volatile short *)THS1408_PGA_ADDR,
            (volatile short *)THS1408_OFF_ADDR,
            (volatile short *)THS1408_CTL_ADDR,
   };                      /* handle to the AD Converter */

   TIMER_Handle hTimer0;   /* handle to the Timer 0 */
   EDMA_Handle hEdmaIn;    /* handle to the Input EDMA */
   EDMA_Handle hEdmaInLinkA; /* EDMA link for Buffer A */
   EDMA_Handle hEdmaInLinkB; /* EDMA link for Buffer B */

/*---------------------------------------------------------------------
   implementation of functions
  ---------------------------------------------------------------------*/
   void  ths1408_init(short inBufferA[],short inBufferB[])
   {
      THS1408_CONFIG defconfig; /* default config */

      /* Setup the emif parameters */
      emif_init_THS1408();

      /* Open and init the Timer0 */
      timer_init_THS1408();

      /* Setup the EDMA */
      adc_edma_init(inBufferA,inBufferB);
```

```
69
70      /* Configure the THS1408 */
71      defconfig.ctrl.control_bit.pwd=THS1408_POW;
72      defconfig.ctrl.control_bit.ref=THS1408_REF;
73      defconfig.ctrl.control_bit.format=THS1408_FORMAT;
74      defconfig.ctrl.control_bit.tm=THS1408_TM;
75      defconfig.ctrl.control_bit.off=THS1408_OFF;
76      defconfig.ctrl.control_bit.msb_reserved=THS1408_MSB_RESERVED;
77      defconfig.ctrl.control_bit.lsb_reserved=THS1408_LSB_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
87  void  ths1408_config(THS1408_CONFIG config)
88  {
89      *hThs1408.ctl_addr=config.ctrl.value;
90      *hThs1408.off_addr=config.offset;
91      *hThs1408.pga_addr=config.pga;
92
93      TIMER_setPeriod(hTimer0,config.tim_period);
94      /* set Count to 0 because it could be bigger than Period
95        -> would need one overflow*/
96      TIMER_setCount(hTimer0,0x0000);
97  }
98
99  void  emif_init_THS1408()
100 {
101     /* set the EMIF parameters for the CE2 (THS1408) */
102     EMIF_FSET(CECTL2, WRSETUP , THS1408_WRSETUP);
103     EMIF_FSET(CECTL2, WRSTRB , THS1408_WRSTRB);
104     EMIF_FSET(CECTL2, WRHLD , THS1408_WRHLD);
105     EMIF_FSET(CECTL2, RDSETUP , THS1408_RDSETUP);
106     EMIF_FSET(CECTL2, RDSTRB , THS1408_RDSTRB);
107     EMIF_FSET(CECTL2, RDHLD , THS1408_RDHLD);
108     EMIF_FSET(CECTL2, MTYPE , EMIF_CECTL_MTYPE_ASYNC32);
109     EMIF_FSET(CECTL2, TA , EMIF_CECTL_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, PWID , TIMER_CTL_PWID_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_CPUOVR4)
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 edmaConfig_A;
148     EDMA_Config edmaConfig_B;
149
150     /* open one EDMA Channel */
151     hEdmaIn = EDMA_open(EDMA_CHA_TINT0,EDMA_OPEN_RESET);
152
153     #if THS1408_CF_ERROR_DISP
154     if (hEdmaIn==EDMA_HINV)
155     {
156        LOG_printf(&error_log,"Error THS1408 : Could Not Open EDMA Channel !");
157     }
158     #endif
159
160     /* Get a link table for A */
161     hEdmaInLinkA = EDMA_allocTable(-1);
162
163     #if THS1408_CF_ERROR_DISP
164     if (hEdmaInLinkA==EDMA_HINV)
165     {
166        LOG_printf(&error_log,"Error THS1408 : Could Not Allocate EDMA LINK A !");
167     }
168     #endif
169
170     /* Get a link tabel for B */
171     hEdmaInLinkB = EDMA_allocTable(-1);
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_bufferA */
180     edmaConfig_A.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     edmaConfig_A.src = (Uint32) hThs1408.res_addr;
191     edmaConfig_A.cnt = BUFFERSIZE_INPUT;
192     edmaConfig_A.dst = (Uint32) inBufferA_ptr;
193     edmaConfig_A.idx = EDMA_FMK(IDX , FRMIDX , EDMA_IDX_FRMIDX_DEFAULT)
194                 | EDMA_FMK(IDX , ELEIDX , EDMA_IDX_ELEIDX_DEFAULT);
195     edmaConfig_A.rld = EDMA_FMK(RLD , ELERLD , EDMA_RLD_ELERLD_DEFAULT)
196                 | EDMA_FMK(RLD , LINK , EDMA_RLD_LINK_OF(hEdmaInLinkB));
197
198     /* configuration for in_bufferB */
199     edmaConfig_B.opt = EDMA_FMK(OPT , PRI , EDMA_OPT_PRI_HIGH)
200                 | EDMA_FMK(OPT , ESIZE , EDMA_OPT_ESIZE_16BIT)
201                 | EDMA_FMK(OPT , 2DS , EDMA_OPT_2DS_NO)
202                 | EDMA_FMK(OPT , SUM , EDMA_OPT_SUM_NONE)
203                 | EDMA_FMK(OPT , 2DD , EDMA_OPT_2DD_NO)
204                 | EDMA_FMK(OPT , DUM , EDMA_OPT_DUM_INC)
205                 | EDMA_FMK(OPT , TCINT , EDMA_OPT_TCINT_YES)
206                 | EDMA_FMK(OPT , TCC , EDMA_OPT_TCC_OF(THS1408_BUFFER_B_TCC))
207                 | EDMA_FMK(OPT , LINK , EDMA_OPT_LINK_YES)
208                 | EDMA_FMK(OPT , FS , EDMA_OPT_FS_NO);
209     edmaConfig_B.src = (Uint32) hThs1408.res_addr;
210     edmaConfig_B.cnt = BUFFERSIZE_INPUT;
211     edmaConfig_B.dst = (Uint32) inBufferB_ptr;
212     edmaConfig_B.idx = EDMA_FMK(IDX , FRMIDX , EDMA_IDX_FRMIDX_DEFAULT)
213                 | EDMA_FMK(IDX , ELEIDX , EDMA_IDX_ELEIDX_DEFAULT);
214     edmaConfig_B.rld = EDMA_FMK(RLD , ELERLD , EDMA_RLD_ELERLD_DEFAULT)
215                 | EDMA_FMK(RLD , LINK , EDMA_RLD_LINK_OF(hEdmaInLinkA));
216
217     /* configure channel and links */
218     EDMA_config(hEdmaIn,&edmaConfig_A);
219     EDMA_config(hEdmaInLinkA,&edmaConfig_A);
220     EDMA_config(hEdmaInLinkB,&edmaConfig_B);
221
222     /* enable transfer completion interrupt for A and B */
223     EDMA_intEnable(THS1408_BUFFER_A_TCC);
224     EDMA_intEnable(THS1408_BUFFER_B_TCC);
225 }
226
227 void  adc_irq_init()
228 {
```

```
      /* Map and enable EDMA interrupt */
      IRQ_map(IRQ_EVT_EDMAINT,8);
      IRQ_enable(IRQ_EVT_EDMAINT);
   }

   void  ths1408_start()
   {
      TIMER_start(hTimer0);
      EDMA_enableChannel(hEdmaIn);
   }

   #if THS1408_ENABLE_SET_FREQ
      double  ths1408_setFreq(double freq)
      {
         Uint32 period=0;
         double ret=0;
         period=(1000.0*THS1408_DSP_FREQ)/(8*freq);
         ret=(1000.0*THS1408_DSP_FREQ)/(8*period);

         TIMER_setPeriod(hTimer0,period);
         /* set Count to 0 because it could be bigger than Period
            -> would need one overflow*/
         TIMER_setCount(hTimer0,0x0000);

         return ret;
      }
   #endif

/*_____________________________________________________________________
   END OF THE FILE adc_THS1408.c
 _____________________________________________________________________*/
```

dac_codec

Listing F.3: *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
18    #include <csl_irq.h>
19    #include <csl_mcbsp.h>
20    #include <bsl_ad535.h>
21
22 /*---------------------------------------------------------------------
23    global variables ( extern defined in .c )
24   ---------------------------------------------------------------------*/
25
26    extern AD535_Handle outAD535h ;
27
28 /*---------------------------------------------------------------------
29    global functions
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    */
41    void codec_start();
42
43    /*
44       Writes value to the input of the DAC. Unlike
45       the AD535_write API, it does not use polling to establish
46       that the McBSP is ready to write another sample. Rather, it
47       requires the McBSP to already be ready. In other words,
48       the AD535_writeHwi is for use within an Interrupt Service
49       Routine. The face that you arrived at an McBSP transmit ISR
50       signifies that the McBSP is ready with another sample.
51
52       Parameters :
53          handle   handle to codec channel
54          out_data value to be written to DAC.
55    */
56    void AD535_HWI_write(AD535_Handle handle, short out_data);
57
58    /*
59       Sets 1 to the FREE field of the SPCR register of the
60       given McBSP port. When FREE is set to 1, the serial
61       clocks continue to run during an emulation halt.
62
63       Parameters :
64          port     McBSP port ( 0 or 1 )
65    */
66    void MCBSP_setfree(int port);
67
68
69 #endif /* _DAC_CODEC_ */
70
71 /*_____________________________________________________________________
72    END OF THE FILE
73  _____________________________________________________________________*/
```

Listing F.4: *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
15    #include "dac_codec.h"
16
17
18 /*---------------------------------------------------------------------
19    global varibles
20   ---------------------------------------------------------------------*/
21    AD535_Handle outAD535h ;
22
23 /*---------------------------------------------------------------------
24    implementation of functions
25   ---------------------------------------------------------------------*/
26
27    void codec_init()
28    {
29       AD535_Config outAD535c ={
30                          AD535_LOOPBACK_DISABLE,/*Loopback mode */
31                          AD535_MICGAIN_OFF,  /*Microphone gain */
32                          AD535_GAIN_0DB,   /*ADC input gain */
33                          AD535_GAIN_0DB    /*DAC output gain */
34                         };
35
36       outAD535h = AD535_open(AD535_localId); /* open the AD535 for codec use */
37       AD535_reset(outAD535h);      /* reset it */
38       AD535_config(outAD535h,& outAD535c ); /* setup according to outAD535c */
39       MCBSP_setfree(MCBSP_DEV0);
40
41       IRQ_map(IRQ_EVT_XINT0,9); /* map the codec interrupt to the dsp */
42       IRQ_enable(IRQ_EVT_XINT0); /* enable the codec interrupt  */
```

```
43    }
44
45    void codec_start()
46    {
47       AD535_write(outAD535h,0); /* codec starts to after the fisrt value is
48                                    written to it                 */
49    }
50
51    void AD535_HWI_write(AD535_Handle handle, short out_data)
52    {
53       MCBSP_Handle h;
54       h = AD535_getMcbspHandle(handle);
55       out_data &= 0xFFFE;
56       MCBSP_write(h,(int)out_data);
57    }
58
59    void MCBSP_setfree(int port)
60    {
61       switch (port)
62       {
63          case 0:
64             MCBSP_FSET(SPCR0,FREE,1);
65
66             break;
67          case 1:
68             MCBSP_FSET(SPCR0,FREE,1);
69             break;
70       }
71    }
72
73 /*_____________________________________________________________________
74    END OF THE FILE
75  _____________________________________________________________________*/
```

F.1.2. Floating-Point

global_settings

Listing F.5: *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
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.6: *Floating-Point :: fm_dem_main.c*

```
1  /*_____________________________________________________________________
2     FM DEMODULATION MAIN - VERSION FLOATING POINT 1.0
3
4     file  :  fm_dem_main.c
5     date  :  oct 2001 - jan 2002
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 <csl.h>
26    #include <bsl.h>
27
28    #include "global_settings.h"
29    #include "adc_THS1408.h"
30    #include "dac_codec.h"
31    #include "quad_mix.h"
32    #include "downsample_one.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    #include "out_filter.h"
41    #include "downsample_two.h"
42
43 /*---------------------------------------------------------------------
44    debug defines
45  ---------------------------------------------------------------------*/
46    #define FM_DEM_CF_DEBUG    0
47       /* to print comments to debug_log*/
48
49    #define FM_DEM_CF_INT_OCCUR_LED  0
50       /* Led toggles when an interrupt occurs   */
51       /* LED 1 codec interrupt (output)    */
52       /* LED 2 edma interrupt (input)     */
53       /* LED 3 timming error              */
54
55    #define FM_DEM_CF_INT_TIME_LED  0
56       /* Led ON by interrupt start OFF at the end  */
57       /* LED 1 codec interrupt (output)    */
58       /* LED 2 edma interrupt (input)     */
59       /* LED 3 software interrupt         */
60
61    #define FM_DEM_CF_INT_TIME_STS  0
62       /* Enable STS Objects:                */
63       /* -dac_sts                           */
64       /* -demodulate_sts                    */
65       /* -edma_sts                          */
66       /* to measure the duration time of the interrupts */
67
68    #define FM_DEM_CF_FUNC_TIME_STS  1
69       /* Enable STS Objects:                */
70       /* -mix_sts                           */
71       /* -dem_sts                           */
72       /* -filter_sts                        */
73       /* to measure the duration time of the functions */
74
75    #if FM_DEM_CF_DEBUG
76       #include <log.h>
77       extern far LOG_Obj debug_log;
78    #endif
79
80    #if FFM_DEM_CF_INT_OCCUR_LED
81       #include <bsl_led.h>
82    #endif
83
84    #if FFM_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 varibles
108   -----------------------------------------------------------------*/
109
110   extern far SWI_Obj demodulate_swi;
111   extern HWI_enable();
112   extern HWI_disable();
113
114 /*-----------------------------------------------------------------
115    global varibles
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
137   short* in_buffer_ptr;
138   short* out_buffer_ptr;
139   Bool  a_inEDMA=TRUE;
140   short outcounter=0;
141
142 /*-----------------------------------------------------------------
143    the main function
144   -----------------------------------------------------------------*/
145   void main()
146   {
147      HWI_disable();
148
149      #if FM_DEM_CF_DEBUG
150         LOG_printf(&debug_log,"Info : Begin Of main ");
151      #endif
152
153      CSL_init();  /* on chip init   */
154      BSL_init();  /* on board init  */
155
156      ths1408_init(in_buffer_A,in_buffer_B);
157      codec_init();
158
159      HWI_enable();
160
161      ths1408_start();
162      codec_start();
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
177   void edmaInt(void)
178   {
179      #if FM_DEM_CF_INT_OCCUR_LED
180         LED_toggle(LED_2);
181      #endif
182
183      #if FM_DEM_CF_INT_TIME_LED
184         LED_on(LED_2);
185      #endif
186
187      #if FM_DEM_CF_INT_TIME_STS
188         STS_set(&edma_sts,CLK_gethtime());
189      #endif
190
191      /* reset outcounter */
192      outcounter=0;
193
194      /* select EDMA channel */
195      if (EDMA_intTest(THS1408_BUFFER_A_TCC))
196      {
197         #if FM_DEM_CF_DEBUG
198            LOG_printf(&debug_log," In_A");
199         #endif
200         /* Clear transfer completion interrupt flag */
201         EDMA_intClear(THS1408_BUFFER_A_TCC);
202
203         /* set buffer pointers */
204         a_inEDMA=FALSE;
205         in_buffer_ptr=in_buffer_A;
206         out_buffer_ptr=out_buffer_A;
207
208         /* post software interrupt -> start demodulation*/
209         SWI_post(&demodulate_swi);
210      }
211      if (EDMA_intTest(THS1408_BUFFER_B_TCC))
212      {
213         #if FM_DEM_CF_DEBUG
214            LOG_printf(&debug_log," In_B");
215         #endif
216         /* Clear transfer completion interrupt flag */
217         EDMA_intClear(THS1408_BUFFER_B_TCC);
218
219         /* set buffer pointers */
220         a_inEDMA=TRUE;
221         in_buffer_ptr=in_buffer_B;
222         out_buffer_ptr=out_buffer_B;
223
224         /* post software interrupt -> start demodulation*/
225         SWI_post(&demodulate_swi);
226      }
227
228      #if FM_DEM_CF_INT_TIME_STS
229         STS_delta(&edma_sts,CLK_gethtime());
230      #endif
231
232      #if FM_DEM_CF_INT_TIME_LED
233         LED_off(LED_2);
234      #endif
235   }
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       #endif
246
247       #if FM_DEM_CF_INT_TIME_STS
248          STS_set(&dac_sts,CLK_gethtime());
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_HWI_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          }else
265          {
266             AD535_HWI_write(outAD535h,((out_buffer_B[15]+out_buffer_A[0])/2));
267             #if FM_DEM_CF_DEBUG
268                LOG_printf(&debug_log,"Out B error :%d",outcounter);
269             #endif
270          }
271       }else /* output a value */
272       {
273          /* select the buffer */
274          if(a_inEDMA)
275          {
276             AD535_HWI_write(outAD535h,out_buffer_A[outcounter]);
277             #if FM_DEM_CF_DEBUG
278                LOG_printf(&debug_log,"Out A :%d",outcounter);
279             #endif
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          /* update outcounter */
288          outcounter++;
289       }
290
291       #if FM_DEM_CF_INT_TIME_STS
292          STS_delta(&dac_sts,CLK_gethtime());
293       #endif
294
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_gethtime());
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;
325       }
326
327       #if FM_DEM_CF_FUNC_TIME_STS
328          STS_set(&mix_sts,CLK_gethtime());
329       #endif
330       quad_mix(in_buffer,re_buffer,im_buffer);
331       #if FM_DEM_CF_FUNC_TIME_STS
332          STS_delta(&mix_sts,CLK_gethtime());
333       #endif
334
335       #if FM_DEM_CF_FUNC_TIME_STS
336          STS_set(&down_one_sts,CLK_gethtime());
337       #endif
338       downsample_one(re_buffer,im_buffer,re_buffer_down_one,im_buffer_down_one);
339       #if FM_DEM_CF_FUNC_TIME_STS
340          STS_delta(&down_one_sts,CLK_gethtime());
341       #endif
342
343       #if FM_DEM_CF_FUNC_TIME_STS
344          STS_set(&dem_sts,CLK_gethtime());
345       #endif
346       #if   (FM_DEMODULATOR==FM_PLL_DEM)
347          pll_demodulate(re_buffer_down_one,im_buffer_down_one,out_buffer_down_one);
348       #elif (FM_DEMODULATOR==FM_MIXED_DEM)
349          mixed_demodulate(re_buffer_down_one,im_buffer_down_one,out_buffer_down_one);
350       #else
351          #error No demodulator algorithm defined
352       #endif
353       #if FM_DEM_CF_FUNC_TIME_STS
354          STS_delta(&dem_sts,CLK_gethtime());
355       #endif
356
357       #if FM_DEM_CF_FUNC_TIME_STS
358          STS_set(&filter_sts,CLK_gethtime());
359       #endif
360       out_filter(out_buffer_down_one);
361       #if FM_DEM_CF_FUNC_TIME_STS
362          STS_delta(&filter_sts,CLK_gethtime());
363       #endif
364
365
366       #if FM_DEM_CF_FUNC_TIME_STS
367          STS_set(&down_two_sts,CLK_gethtime());
368       #endif
369       downsample_two(out_buffer_down_one,out_buffer_down_two);
370       #if FM_DEM_CF_FUNC_TIME_STS
371          STS_delta(&down_two_sts,CLK_gethtime());
372       #endif
373
374       /*covert from float to short*/
375       for(i=0;i<BUFFERSIZE_OUTPUT;i++)
376       {
377          out_buffer_ptr[i]=out_buffer_down_two[i]*32767;
378       }
379
380       #if FM_DEM_CF_INT_TIME_STS
381          STS_delta(&demodulate_sts,CLK_gethtime());
382       #endif
383
384       #if FM_DEM_CF_INT_TIME_LED
385          LED_off(LED_3);
386       #endif
387    }
```

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   _____________________________________________________________________
```

```
*/

#ifndef _QUAD_MIX_
#define _QUAD_MIX_

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "global_settings.h"

/*---------------------------------------------------------------------
   global functions
  ---------------------------------------------------------------------*/
   /*
      The in_buffer is quadrature mixed to the two
      buffers re_buffer and im_buffer.

      Parameters :

      in_buffer   buffer of input signal
      re_buffer   buffer of I signal
      im_buffer   buffer of Q signal
   */
   void quad_mix(float in_buffer[] , float re_buffer[] , float im_buffer[]);

#endif /* _QUAD_MIX_ */

/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

Listing F.8: *Floating-Point :: quad_mix.c*

```
/*_____________________________________________________________________
   QUADRATURE MIXER - VERSION FLOATING POINT 1.0

   file  :  quad_mix.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
  _____________________________________________________________________
*/

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "quad_mix.h"


/*---------------------------------------------------------------------
   global varibles
  ---------------------------------------------------------------------*/
   /*coefficients of iir filter for cosine/sine generation*/

   float bc2=-0.27697926505;/*cosine*/
   float bs2=0.960875895593;/*sine*/
   float asc2=-0.5539585301;/*sine/cosine*/

   /*delay variabels of iir filter for cosine/sine generation*/

   float zsc1=0.0;
   float zsc2=0.0;
   float zsc3=1.0;/*for first value of delta function*/

   /*coefficients of iir filter for low pass filtering*/

   float blp1=0.0015;
   float blp2=0.0076;
   float blp3=0.0151;
   float blp4=0.0151;
   float blp5=0.0076;
   float blp6=0.0015;
   float alp2=-2.8957;
   float alp3=3.6344;
   float alp4=-2.3928;
   float alp5=0.8176;
   float alp6=-0.1151;

   /*delay variabels of iir filter for low pass filtering the I-path*/

   float zlpc1=0.0;
   float zlpc2=0.0;
   float zlpc3=0.0;
   float zlpc4=0.0;
   float zlpc5=0.0;
   float zlpc6=0.0;

   /*delay variabels of iir filter for low pass filtering the Q-path*/

   float zlps1=0.0;
   float zlps2=0.0;
   float zlps3=0.0;
   float zlps4=0.0;
   float zlps5=0.0;
   float zlps6=0.0;

/*---------------------------------------------------------------------
   implementation of function
  ---------------------------------------------------------------------*/
   void quad_mix(float in_buffer[] , float re_buffer[] , float im_buffer[])
   {
      int i;

      /*__cosine/sine generation__*/

      for(i=0;i<BUFFERSIZE_INPUT;i++)
      {
         /*feedback*/
         zsc1=-(zsc2*asc2+zsc3);

         /*foreward*/
         re_buffer[i]=zsc1+zsc2*bc2;/*cosine*/
         im_buffer[i]=zsc2*bs2;/*sine*/

         /*shift delays*/
         zsc3=zsc2;
         zsc2=zsc1;
      }

      /*__mixing__*/

      for(i=0;i<BUFFERSIZE_INPUT;i++)
      {
         /*mixing*/
         re_buffer[i]=re_buffer[i]*in_buffer[i];/*I-path*/
         im_buffer[i]=im_buffer[i]*in_buffer[i];/*Q-path*/
      }

      /*__low pass__*/

      for(i=0;i<BUFFERSIZE_INPUT;i++)
      {
         /*feedback*/
         zlpc1=re_buffer[i]-(alp2*zlpc2+alp3*zlpc3+alp4*zlpc4+alp5*zlpc5+alp6*zlpc6);
               /*I-path*/
         zlps1=im_buffer[i]-(alp2*zlps2+alp3*zlps3+alp4*zlps4+alp5*zlps5+alp6*zlps6);
               /*Q-path*/

         /*foreward*/
         re_buffer[i]=blp1*zlpc1+blp2*zlpc2+blp3*zlpc3+blp4*zlpc4+blp5*zlpc5+blp6*
               zlpc6;/*I-path*/
         im_buffer[i]=-(blp1*zlps1+blp2*zlps2+blp3*zlps3+blp4*zlps4+blp5*zlps5+blp6*
               zlps6);/*Q-path*/

         /*shift delays*/
         /*I-path*/
         zlpc6=zlpc5;
         zlpc5=zlpc4;
         zlpc4=zlpc3;
         zlpc3=zlpc2;
         zlpc2=zlpc1;
         /*Q-path*/
         zlps6=zlps5;
         zlps5=zlps4;
         zlps4=zlps3;
         zlps3=zlps2;
         zlps2=zlps1;
      }
   }
```

```
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
11 #ifndef _DOWNSAMPLE_ONE_
12 #define _DOWNSAMPLE_ONE_
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         buffer of Q signal
30       re_buffer_down_one  buffer of downsampled I signal
31       im_buffer_down_one  buffer of downsampled Q signal
32
33    */
34    void downsample_one(float re_buffer[] , float im_buffer[] , float
            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  :  downsamle_one.c
5     date  :  oct 2001 - jan 2002
6     by    :  c. haller
7              f. schnyder
8  _____________________________________________________________________
9  */
10
11 /*---------------------------------------------------------------------
12    includes
13   ---------------------------------------------------------------------*/
14    #include "downsample_one.h"
15
16 /*---------------------------------------------------------------------
17    implementation of function
18   ---------------------------------------------------------------------*/
19    void downsample_one(float re_buffer[] , float im_buffer[] , float
            re_buffer_down_one[] , float im_buffer_down_one[])
20    {
21       int i;
22
23       for(i=0;i<BUFFERSIZE_DEMODULATOR;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    includes
16   ---------------------------------------------------------------------*/
17    #include "global_settings.h"
18    #include <math.h>
19
20 /*---------------------------------------------------------------------
21    global functions
22   ---------------------------------------------------------------------*/
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
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 mixed_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] ,
            float out_buffer_down_one[]);
34
35 #endif /* _MIXED_DEMODULATE_ */
36
37 /*_____________________________________________________________________
38    END OF THE FILE
39  _____________________________________________________________________*/
```

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 /*---------------------------------------------------------------------
```

```
   includes
  ------------------------------------------------------------------*/
   #include "mixed_demodulate.h"

/*-------------------------------------------------------------------
   global varibles
  ------------------------------------------------------------------*/
   /*variabels of demodulating (attension division by zero!)*/

   float coold=0.0;/*old value of I-path*/
   float conew=1.0;/*new value of I-path*/
   float siold=0.0;/*old value of Q-path*/
   float sinew=1.0;/*new value of Q-path*/
   float demo=0.0;/*value demodulating*/

   /*constant of demodulating*/

   float decon=0.7;


/*-------------------------------------------------------------------
   implementation of function
  ------------------------------------------------------------------*/
   void mixed_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] ,
         float out_buffer_down_one[])
   {
      int i;

      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         /*__shifting values__*/
         coold=conew;
         siold=sinew;
         conew=re_buffer_down_one[i];
         sinew=im_buffer_down_one[i];

         /*__calculating demodulated value__*/
         demo=(coold*sinew-siold*conew)/(coold*conew+siold*sinew);/*atan argument*/
         demo=atanf(demo);/*atan value*/
         demo=demo*decon;/*demodulated value*/
         out_buffer_down_one[i]=demo;
      }
   }


/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

pll_demodulate

Listing F.13: *Floating-Point :: pll_demodulate.h*

```
/*_____________________________________________________________________
   PLL DEMODULATOR - VERSION FLOATING POINT 1.0

   file  :  pll_demodulate.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
  _____________________________________________________________________
*/

#ifndef _PLL_DEMODULATE_
#define _PLL_DEMODULATE_

/*-------------------------------------------------------------------
   includes
  ------------------------------------------------------------------*/
   #include "global_settings.h"
   #include <math.h>

/*-------------------------------------------------------------------
   global functions
  ------------------------------------------------------------------*/
   /*
      The re_buffer_down_one and im_buffer_down_one are demodulated
      with the pll algorithm to the out_buffer_down_one.

      Parameters :

      re_buffer_down_one  buffer of downsampled I signal
      im_buffer_down_one  buffer of downsampled Q signal
      out_buffer_down_one  buffer of demodulated signal
   */
   void pll_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] ,
         float out_buffer_down_one[]);

#endif /* _PLL_DEMODULATE_ */


/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

Listing F.14: *Floating-Point :: pll_demodulate.c*

```
/*_____________________________________________________________________
   PLL DEMODULATOR - VERSION FLOATING POINT 1.0

   file  :  pll_demodulate.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
  _____________________________________________________________________
*/

/*-------------------------------------------------------------------
   includes
  ------------------------------------------------------------------*/
   #include "pll_demodulate.h"

/*-------------------------------------------------------------------
   global varibles
  ------------------------------------------------------------------*/
   /*constants for demodulate*/

   float P_PLL=2.0;
   float K_PLL=2.5;
   float GAIN_PLL=1.5;

   /*variables for demodulate*/

   float arg=0.0;  /*argument of sine and cosine*/
   float i_path=0.0; /*I path*/
   float q_path=0.0; /*Q path*/

/*-------------------------------------------------------------------
   implementation of function
  ------------------------------------------------------------------*/
   void pll_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] ,
         float out_buffer_down_one[])
   {
      int i;

      float demo;  /*value of demodulation*/

      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         /*__calculation of demodulation__*/
         i_path=re_buffer_down_one[i]*sinf(arg);
         q_path=im_buffer_down_one[i]*cosf(arg);

         demo=q_path-i_path;
         demo=demo*P_PLL;

         out_buffer_down_one[i]=demo*GAIN_PLL;

         /*__summation of argument__*/
         arg=arg+K_PLL*demo;

         /*__limitation of argument__*/
         if (arg>2*PI)
         {
```

```
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 /*_____________________________________________________________________
69    END OF THE FILE
70  _____________________________________________________________________*/
```

out_filter

Listing F.15: *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
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       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 F.16: *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 /*---------------------------------------------------------------------
12    includes
13   ---------------------------------------------------------------------*/
14    #include "out_filter.h"
15
16
17 /*---------------------------------------------------------------------
18    global varibles
19   ---------------------------------------------------------------------*/
20
21 /*---------------------------------------------------------------------
22   coefficients of output filter (band pass) after demodulation
23   ---------------------------------------------------------------------*/
24    /*band pass order : 10    */
25    /*cutoff frequency 1 : 200 Hz */
26    /*cutoff frequency 2 : 3600 Hz */
27
28    /*coefficients of iir filtering for band pass filtering*/
29
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 variabels of iir filtering for band pass filtering*/
53
54    float zbp1=0.0;
55    float zbp2=0.0;
56    float zbp3=0.0;
57    float zbp4=0.0;
58    float zbp5=0.0;
59    float zbp6=0.0;
60    float zbp7=0.0;
61    float zbp8=0.0;
62    float zbp9=0.0;
63    float zbp10=0.0;
64    float zbp11=0.0;
65
66
67 /*---------------------------------------------------------------------
68    implementation of function
69   ---------------------------------------------------------------------*/
70    void out_filter(float out_buffer_down_one[])
71    {
72       int i;
73
74       for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
75       {
76          /*feedback*/
77          zbp1=out_buffer_down_one[i]-(abp2*zbp2+abp3*zbp3+abp4*zbp4+abp5*zbp5+abp6*
                 zbp6+abp7*zbp7+abp8*zbp8+abp9*zbp9+abp10*zbp10+abp11*zbp11);
78
79          /*foreward*/
80          out_buffer_down_one[i]=bbp1*zbp1+bbp2*zbp2+bbp3*zbp3+bbp4*zbp4+bbp5*zbp5+
                 bbp6*zbp6+bbp7*zbp7+bbp8*zbp8+bbp9*zbp9+bbp10*zbp10+bbp11*zbp11;
81
82          /*shift delays*/
83          zbp11=zbp10;
84          zbp10=zbp9;
85          zbp9=zbp8;
86          zbp8=zbp7;
87          zbp7=zbp6;
88          zbp6=zbp5;
89          zbp5=zbp4;
90          zbp4=zbp3;
91          zbp3=zbp2;
92          zbp2=zbp1;
93       }
```

```
    }

/*_____________________________________________________________________
   END OF THE FILE
 _____________________________________________________________________*/
```

downsample_two

Listing F.17: *Floating-Point :: downsample_two.h*

```
/*_____________________________________________________________________
   DOWNSAMPLE TWO - VERSION FLOATING POINT 1.0

   file  :  downsample_two.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _____________________________________________________________________
*/

#ifndef _DOWNSAMPLE_TWO_
#define _DOWNSAMPLE_TWO_

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "global_settings.h"

/*---------------------------------------------------------------------
   global functions
  ---------------------------------------------------------------------*/
   /*
      The out_buffer_down_one is downsampled to the
      out_buffer_down_two.

      Parameters :

      out_buffer_down_one  buffer of demodulated signal.
      out_buffer_down_two  buffer of downsampled demodulated signal


   */
   void downsample_two(float out_buffer_down_one[] , float out_buffer_down_two[]);
#endif /* _DOWNSAMPLE_TWO_ */

/*_____________________________________________________________________
   END OF THE FILE
 _____________________________________________________________________*/
```

Listing F.18: *Floating-Point :: downsample_two.c*

```
/*_____________________________________________________________________
   DOWNSAMPLE TWO - VERSION FLOATING POINT 1.0

   file  :  downsamle_two.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _____________________________________________________________________
*/

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "downsample_two.h"

/*---------------------------------------------------------------------
   implementation of function
  ---------------------------------------------------------------------*/
   void downsample_two(float out_buffer_down_one[] , float out_buffer_down_two[])
   {
      int i;

      for(i=0;i<BUFFERSIZE_OUTPUT;i++)
      {
         out_buffer_down_two[i]=out_buffer_down_one[i*DOWNSAMPLE_TWO];
      }
   }

/*_____________________________________________________________________
   END OF THE FILE
 _____________________________________________________________________*/
```

F.1.3. Fixed-Point

global_settings

Listing F.19: *Fixed-Point :: global_settings.h*

```
/*_____________________________________________________________________
   GLOBAL SETTINGS - - VERSION FIXED POINT 0.5

   file  :  global_settings.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _____________________________________________________________________
*/

#define BUFFERSIZE 128
#define DOWNSAMPLE_ONE 4
#define DOWNSAMPLE_TWO 2

#define BUFFERSIZE_INPUT  (BUFFERSIZE)
#define BUFFERSIZE_DEMODULATOR (BUFFERSIZE/DOWNSAMPLE_ONE)
#define BUFFERSIZE_OUTPUT  (BUFFERSIZE/(DOWNSAMPLE_ONE*DOWNSAMPLE_TWO))

#define PI 3.14159265359
```

fm_dem_main

Listing F.20: *Fixed-Point :: fm_dem_main.c*

```
/*_____________________________________________________________________
   FM DEMODULATION MAIN - VERSION FIXED POINT 0.5

   file  :  fm_dem_main.c
   date  :  nov 2001
   by    :  c. haller
            f. schnyder
 _____________________________________________________________________
*/

/*---------------------------------------------------------------------
   demodulator define
  ---------------------------------------------------------------------*/
   #define FM_MIXED_DEM    10
   #define FM_PLL_DEM      20

   #define FM_DEMODULATOR   FM_MIXED_DEM

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/

   #include <std.h>
   #include <swi.h>
   #include <csl.h>
   #include <bsl.h>

   #include "global_settings.h"
   #include "adc_THS1408.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 timming 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 <bsl_led.h>
80    #endif
81
82    #if FFM_DEM_CF_INT_TIME_LED
83       #include <bsl_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 varibles
104  ---------------------------------------------------------------*/
105
106   extern far SWI_Obj demodulate_swi;
107   extern HWI_enable();
108   extern HWI_disable();
109
```

```
110 /*---------------------------------------------------------------
111   global varibles
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
126   short* in_buffer_ptr;
127   short* out_buffer_ptr;
128   Bool  a_inEDMA=TRUE;
129   short outcounter=0;
130
131 /*---------------------------------------------------------------
132   the main function
133  ---------------------------------------------------------------*/
134   void main()
135   {
136      HWI_disable();
137
138      #if FM_DEM_CF_DEBUG
139         LOG_printf(&debug_log,"Info : Begin Of main ");
140      #endif
141
142      CSL_init();  /* on chip init   */
143      BSL_init();  /* on board init  */
144
145      ths1408_init(in_buffer_A,in_buffer_B);
146      codec_init();
147
148      HWI_enable();
149
150      ths1408_start();
151      codec_start();
152
153      #if FM_DEM_CF_INT_TIME_LED
154         LED_off(LED_ALL);
155      #endif
156
157      #if FM_DEM_CF_DEBUG
158         LOG_printf(&debug_log,"Info : End Of main ");
159      #endif
160   }
161
162 /*---------------------------------------------------------------
163   hardware interrupt functions
164  ---------------------------------------------------------------*/
165
166   void edmaInt(void)
167   {
168      #if FM_DEM_CF_INT_OCCUR_LED
169         LED_toggle(LED_2);
170      #endif
171
172      #if FM_DEM_CF_INT_TIME_LED
173         LED_on(LED_2);
174      #endif
175
176      #if FM_DEM_CF_INT_TIME_STS
177         STS_set(&edma_sts,CLK_gethtime());
178      #endif
179
180      /* reset outcounter */
181      outcounter=0;
182
183      /* select EDMA channel */
184      if (EDMA_intTest(THS1408_BUFFER_A_TCC))
185      {
186         #if FM_DEM_CF_DEBUG
187            LOG_printf(&debug_log," In_A");
188         #endif
189         /* Clear transfer completion interrupt flag */
```

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

quad_mix

Listing F.21: Fixed-Point :: quad_mix.h

```c
/*______________________________________________________________
   QUADRATURE MIXER - VERSION FIXED POINT 0.5

   file  :  quad_mix.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _______________________________________________________________
*/

#ifndef _QUAD_MIX_
#define _QUAD_MIX_

/*-----------------------------------------------------------------
   includes
 -----------------------------------------------------------------*/
   #include "global_settings.h"

/*-----------------------------------------------------------------
   global functions
 -----------------------------------------------------------------*/
   /*
      the signal in the in_buffer is quadrature mixed to the two
      buffers re_buffer and im_buffer and is also downsampled by
      DOWNSAMPLE_ONE

      Parameters :
         in_buffer buffer with input signal
         re_buffer buffer of signal mixed with cosine
         out_buffer buffer of signal mixed with sine
   */
   void quad_mix(short in_buffer[] , short re_buffer[] , short im_buffer[]);

#endif /* _QUAD_MIX_ */

/*______________________________________________________________
   END OF THE FILE
 _______________________________________________________________*/
```

Listing F.22: Fixed-Point :: quad_mix.c

```c
/*______________________________________________________________
   QUADRATURE MIXER - VERSION FIXED POINT 0.5

   file  :  quad_mix.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _______________________________________________________________
*/

/*-----------------------------------------------------------------
   includes
 -----------------------------------------------------------------*/
   #include "quad_mix.h"


/*-----------------------------------------------------------------
   global varibles
 -----------------------------------------------------------------*/


/*-----------------------------------------------------------------
  coefficients of sine and cosine oszilator
 -----------------------------------------------------------------*/
   /* Frequency : 13139.93 HZ*/

   short gen_b1_cos = -9076;  /*cos*/
   short gen_b1_sin = 31486;  /*sin*/
   short gen_a1 = -18152;  /*sin/cos*/

   /* delay variabels */
   short gen_z [3] = {0,0,26213}; /* first value (delta function) */
                              /* 0.8 to avoid overflows  */

/*-----------------------------------------------------------------
  coefficients of mixer low-pass (mix_lp)
 -----------------------------------------------------------------*/
   /* Order   : 5            */
   /* Pass-band : 2 dB @ 6300.00 Hz */
   /* Stop-band : 60 dB @ 19000.00 Hz */

   #define MIX_LP_CASCADES 3
   short const mix_lp_a[][3]={ /* a_i_1a a_i_1b a_i_2 */
                    { -8092 , -8092 ,  0 },
                    { -17444 , -17444 , 11166 },
                    { -21907 , -21907 , 22406 },
                 };

   short const mix_lp_b[][3]={ /* b_i_0 b_i_1 b_i_2 */
                    {  4523 , 4523 ,   0 },
                    {  2725 , 5450 , 2725 },
                    {  8537 , 17074 , 8537 },
                 };

   short const mix_lp_scale=16583; /*16583 to avoid overflows */

   /* delay variabels */
   short mix_lp_z_re[][3]={
                     { 0,0,0},
                     { 0,0,0},
                     { 0,0,0},
                  };
   short mix_lp_z_im[][3]={
                     { 0,0,0},
                     { 0,0,0},
                     { 0,0,0},
                  };

/*-----------------------------------------------------------------
   implementation of function
 -----------------------------------------------------------------*/
   void quad_mix(short in_buffer[] , short re_buffer[] , short im_buffer[])
   {
      int i,k;

      /*__ sine and cosine generation __*/

      for(i=0;i<BUFFERSIZE_INPUT;i++)
      {
         /* feedback */
         gen_z[0]=-( ((gen_a1*gen_z[1])>>15) + gen_z[2] );

         /* foreward */
         re_buffer[i]=gen_z[0] +((gen_b1_cos*gen_z[1])>>15);
         im_buffer[i]=(gen_b1_sin*gen_z[1])>>15;

         /* shift delays */
         gen_z[2]=gen_z[1];
         gen_z[1]=gen_z[0];
      }

      /* __ mixing and scaling __ */

      for(i=0;i<BUFFERSIZE_INPUT;i++)
      {
         /* mixing */
         re_buffer[i]=(re_buffer[i]*in_buffer[i])>>15;
         im_buffer[i]=(im_buffer[i]*in_buffer[i])>>15;

         /* scaling and neg of im_path */
         re_buffer[i]=(re_buffer[i]*mix_lp_scale)>>15;
         im_buffer[i]=-((im_buffer[i]*mix_lp_scale)>>15);
      }

      /* __ low-pass filtering__*/

      /* -- Cascade 0 (first Order)-- */
      for(i=0;i<BUFFERSIZE_INPUT;i++)
      {
         /* feedback */
         mix_lp_z_re[0][0] = re_buffer[i]-( ( mix_lp_a[0][0]*mix_lp_z_re[0][1] + /*
               a_1a*z1*/
```

```
                                mix_lp_a[0][1]*mix_lp_z_re[0][1] /*a_1b*z1*/
                               )>>15);

    mix_lp_z_im[0][0] = im_buffer[i]-( ( mix_lp_a[0][0]*mix_lp_z_im[0][1] + /*
           a_1a*z1*/
                                mix_lp_a[0][1]*mix_lp_z_im[0][1] /*a_1b*z1*/
                               )>>15);

    /* foreward */
    re_buffer[i]= ( mix_lp_b[0][0]*mix_lp_z_re[0][0]+ /* b_0 * z0 */
                    mix_lp_b[0][1]*mix_lp_z_re[0][1] /* b_1 * z1 */
                  )>>15;

    im_buffer[i]= ( mix_lp_b[0][0]*mix_lp_z_im[0][0]+ /* b_0 * z0 */
                     mix_lp_b[0][1]*mix_lp_z_im[0][1] /* b_1 * z1 */
                  )>>15;

    /* shift delays */
    mix_lp_z_re[0][1]=mix_lp_z_re[0][0];

    mix_lp_z_im[0][1]=mix_lp_z_im[0][0];
}

/* -- Cascade 1 (Second Order)--*/
for(i=0;i<BUFFERSIZE_INPUT;i++)
{
    /* feedback */
    mix_lp_z_re[1][0] = re_buffer[i]-( ( mix_lp_a[1][0]*mix_lp_z_re[1][1] + /*
           a_1a*z1*/
                                mix_lp_a[1][1]*mix_lp_z_re[1][1] + /*a_1b*z1*/
                                mix_lp_a[1][2]*mix_lp_z_re[1][2]  /*a_2 *z2*/
                               )>>15);

    mix_lp_z_im[1][0] = im_buffer[i]-( ( mix_lp_a[1][0]*mix_lp_z_im[1][1] + /*
           a_1a*z1*/
                                mix_lp_a[1][1]*mix_lp_z_im[1][1] + /*a_1b*z1*/
                                mix_lp_a[1][2]*mix_lp_z_im[1][2]  /*a_2 *z2*/
                               )>>15);

    /* foreward */
    re_buffer[i]= ( mix_lp_b[1][0]*mix_lp_z_re[1][0]+ /* b_0 * z0 */
                     mix_lp_b[1][1]*mix_lp_z_re[1][1]+ /* b_1 * z1 */
                     mix_lp_b[1][2]*mix_lp_z_re[1][2] /* b_2 * z2 */
                   )>>15;

    im_buffer[i]= ( mix_lp_b[1][0]*mix_lp_z_im[1][0]+ /* b_0 * z0 */
                     mix_lp_b[1][1]*mix_lp_z_im[1][1]+ /* b_1 * z1 */
                     mix_lp_b[1][2]*mix_lp_z_im[1][2] /* b_2 * z2 */
                    )>>15;

    /* shift delays */
    mix_lp_z_re[1][2]=mix_lp_z_re[1][1];
    mix_lp_z_re[1][1]=mix_lp_z_re[1][0];

    mix_lp_z_im[1][2]=mix_lp_z_im[1][1];
    mix_lp_z_im[1][1]=mix_lp_z_im[1][0];
}

/* -- Cascade 2 (second Order)--*/ /*including downsampling*/
for(i=0,k=0;i<BUFFERSIZE_INPUT;i++)
{
    /* feedback */
    mix_lp_z_re[2][0] = re_buffer[i]-( ( mix_lp_a[2][0]*mix_lp_z_re[2][1] + /*
           a_1a*z1*/
                                mix_lp_a[2][1]*mix_lp_z_re[2][1] + /*a_1b*z1*/
                                mix_lp_a[2][2]*mix_lp_z_re[2][2]  /*a_2 *z2*/
                               )>>15);

    mix_lp_z_im[2][0] = im_buffer[i]-( ( mix_lp_a[2][0]*mix_lp_z_im[2][1] + /*
           a_1a*z1*/
                                mix_lp_a[2][1]*mix_lp_z_im[2][1] + /*a_1b*z1*/
                                mix_lp_a[2][2]*mix_lp_z_im[2][2]  /*a_2 *z2*/
                               )>>15);

    /* foreward including downsampling -> take every 4th value*/
    if(i%DOWNSAMPLE_ONE==0)
    {
       re_buffer[k]= ( mix_lp_b[2][0]*mix_lp_z_re[2][0]+ /* b_0 * z0 */
                       mix_lp_b[2][1]*mix_lp_z_re[2][1]+ /* b_1 * z1 */
                       mix_lp_b[2][2]*mix_lp_z_re[2][2] /* b_2 * z2 */
                     )>>15;

       im_buffer[k]= ( mix_lp_b[2][0]*mix_lp_z_im[2][0]+ /* b_0 * z0 */
                        mix_lp_b[2][1]*mix_lp_z_im[2][1]+ /* b_1 * z1 */
                        mix_lp_b[2][2]*mix_lp_z_im[2][2] /* b_2 * z2 */
                      )>>15;
       k++;
    }

    /* shift delays */
    mix_lp_z_re[2][2]=mix_lp_z_re[2][1];
    mix_lp_z_re[2][1]=mix_lp_z_re[2][0];

    mix_lp_z_im[2][2]=mix_lp_z_im[2][1];
    mix_lp_z_im[2][1]=mix_lp_z_im[2][0];
  }
}


/*_____________________________________________________________________
  END OF THE FILE
 _____________________________________________________________________*/
```

mixed_demodulate

Listing F.23: *Fixed-Point :: mixed_demodulate.h*

```
/*_____________________________________________________________________
  MIXED DEMODULATOR - VERSION FIXED POINT 0.5

  file  :  mixed_demodulate.h
  date  :  oct 2001 - jan 2002
  by    :  c. haller
           f. schnyder
 _____________________________________________________________________
*/

#ifndef _MIXED_DEMODULATE_
#define _MIXED_DEMODULATE_

/*--------------------------------------------------------------------
  includes
 --------------------------------------------------------------------*/
  #include "global_settings.h"
  #include "atan.h"

/*--------------------------------------------------------------------
  global functions
 --------------------------------------------------------------------*/
  /*
     FM demodulates the the two basebandsignal re_buffer and im_buffer
     into the out_buffer with the mixed algorithm

     Parameters :
        re_buffer reel FM path in baseband
        im_buffer imag FM path in baseband
        out_buffer demodulated signal
  */
  void mixed_demodulate(short re_buffer[] , short im_buffer[] , short out_buffer[]);

#endif /* _MIXED_DEMODULATE_ */

/*_____________________________________________________________________
  END OF THE FILE
 _____________________________________________________________________*/
```

Listing F.24: *Fixed-Point :: mixed_demodulate.c*

```
/*_____________________________________________________________________
  MIXED DEMODULATOR - VERSION FIXED POINT 0.5

  file  :  mixed_demodulate.c
  date  :  oct 2001 - jan 2002
```

```
   by    :  c. haller
            f. schnyder
 ____________________________________________________________________
*/

/*--------------------------------------------------------------------
   includes
 --------------------------------------------------------------------*/
   #include "mixed_demodulate.h"

/*--------------------------------------------------------------------
   global varibles
 --------------------------------------------------------------------*/
   /* delay varibales of the mixed demodulator */
   short dem_z_re=16384; /* !=0 to avoid division with 0 */
   short dem_z_im=16384; /* !=0 to avoid division with 0 */

/*--------------------------------------------------------------------
   implementation of function
 --------------------------------------------------------------------*/
   void mixed_demodulate(short re_buffer[] , short im_buffer[] , short out_buffer[])
   {
      int i;
      int re_path=0;
      int im_path=0;
      int dem_arg=0;


      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         re_path = ( im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
         im_path = ( re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */

         /* shift delays */
         dem_z_re=re_buffer[i];
         dem_z_im=im_buffer[i];

         /* calculate the argument */
         dem_arg=re_path/(im_path>>14); /* Q17.14 */

         /* atan lookup */
         if(dem_arg>0)     /*positiv*/
         {
            dem_arg=dem_arg>>(ATAN_SHIFT);
            if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
            out_buffer[i]=atan_buffer[dem_arg];
         }else             /*negative*/
         {
            dem_arg=-dem_arg;
            dem_arg=dem_arg>>(ATAN_SHIFT);
            if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
            out_buffer[i]=-atan_buffer[dem_arg];
         }
      }
   }

/*____________________________________________________________________
   END OF THE FILE
 ____________________________________________________________________*/
```

Listing F.25: *Fixed-Point Interpolated :: mixed_demodulate.h*

```
/*____________________________________________________________________
   MIXED DEMODULATOR WITH INTERPOLATION - VERSION FIXED POINT 0.5

   file  :  mixed_demodulate.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 ____________________________________________________________________
*/

#ifndef _MIXED_DEMODULATE_
#define _MIXED_DEMODULATE_

/*--------------------------------------------------------------------
   includes
 --------------------------------------------------------------------*/
   #include "global_settings.h"
   #include "atan.h"

/*--------------------------------------------------------------------
   global functions
 --------------------------------------------------------------------*/
   /*
      FM demodulates the the two basebandsignal re_buffer and im_buffer
      into the out_buffer with the mixed algorithm

      Parameters :
         re_buffer reel FM path in baseband
         im_buffer imag FM path in baseband
         out_buffer demodulated signal
   */
   void mixed_demodulate(short re_buffer[] , short im_buffer[] , short out_buffer[]);

#endif /* _MIXED_DEMODULATE_ */

/*____________________________________________________________________
   END OF THE FILE
 ____________________________________________________________________*/
```

Listing F.26: *Fixed-Point Interpolated :: mixed_demodulate.c*

```
/*____________________________________________________________________
   MIXED DEMODULATOR WITH INTERPOLATION - VERSION FIXED POINT 0.5

   file  :  mixed_demodulate.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 ____________________________________________________________________
*/

/*--------------------------------------------------------------------
   includes
 --------------------------------------------------------------------*/
   #include "mixed_demodulate.h"

/*--------------------------------------------------------------------
   global varibles
 --------------------------------------------------------------------*/
   /* delay varibales of the mixed demodulator */
   short dem_z_re=16384; /* !=0 to avoid division with 0 */
   short dem_z_im=16384; /* !=0 to avoid division with 0 */

/*--------------------------------------------------------------------
   implementation of function
 --------------------------------------------------------------------*/
   void mixed_demodulate(short re_buffer[] , short im_buffer[] , short out_buffer[])
   {
      int i;
      int re_path=0;
      int im_path=0;
      int dem_arg=0;
      short cutoff=0;


      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {

         re_path = ( im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
         im_path = ( re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */

         /* shift delays */
         dem_z_re=re_buffer[i];
         dem_z_im=im_buffer[i];

         /* calculate the argument */
         dem_arg=re_path/(im_path>>14); /* Q17.14 */

         /* atan lookup */
         if(dem_arg>0)     /*positiv*/
         {
            cutoff=dem_arg%ATAN_INTERPOL;
            dem_arg=dem_arg>>(ATAN_SHIFT);
            if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
            out_buffer[i]=atan_buffer[dem_arg]+((cutoff*atan_grad_buffer[dem_arg])>>
                  ATAN_SHIFT);
```

```
55          }else             /*negative*/
56          {
57
58             dem_arg=-dem_arg;
59             cutoff=dem_arg%ATAN_INTERPOL;
60             dem_arg=dem_arg>>(ATAN_SHIFT);
61             if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
62             out_buffer[i]=-(atan_buffer[dem_arg]+((cutoff*atan_grad_buffer[dem_arg])
                     >>ATAN_SHIFT));
63          }
64       }
65    }
66
67 /*_____________________________________________________________________
68    END OF THE FILE
69  _____________________________________________________________________*/
```

pll_demodulate

Listing F.27: *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
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       FM demodulates the the two basebandsignal 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 F.28: *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 varibles
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_PLL 55721  /*GAIN_PLL=1.7, using q15*/
29    #define TWO_PI  25735   /*2*PI, using q3.12*/
30
31
32 /*---------------------------------------------------------------------
33    implementation of function
34   ---------------------------------------------------------------------*/
35    void pll_demodulate(short re_buffer[] , short im_buffer[] , short out_buffer[])
36    {
37       int i;
38       int flag=0;
39
40       short demo;    /*index for sine_buffer access*/
41       short index;   /*value of demodulation*/
42
43
44       for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
45       {
46          /*__calculation of demodulated value, using Q15__*/
47
48          i_path=(re_buffer[i]*i_path)>>15;
49          q_path=(im_buffer[i]*q_path)>>15;
50
51          demo=q_path-i_path;
52          demo=(demo*P_PLL)>>15;
53
54          out_buffer[i]=(demo*GAIN_PLL)>>15;
55
56
57          /*__summation of demodulated signal, using Q3.12__*/
58
59          demo=demo>>3;
60          arg=arg+((demo*K_PLL)>>12);
61
62
63          /*__limitation of argument to 2*PI, using Q3.12__*/
64
65          if (arg>TWO_PI)
66          {
67             arg=arg-TWO_PI;
68          }
69
70          else if (arg<-TWO_PI)
71          {
72             arg=arg+TWO_PI;
73          }
74
75
76          /*__calculation of sine and cosine value of argument__*/
77
78          /*calculating of index for sine_buffer access, using new Q format*/
79
80          index=arg>>REDUCTION;
81
82          if (index<0)
83          {
84             index=-index;
85             flag=1;
86          }
87
88          index=(index*STEP_INVERSE)>>(2*Q_FORMAT);
89
90          if (flag==1)
91          {
```

```
92              index=-index;
93              flag=0;
94           }
95
96           /*access on sine_buffer with index*/
97
98           if(index>=0)
99           {
100             if(index<(BUFFERSIZE_SINE-1))
101             {
102                /*sine value*/
103                i_path=sine_buffer[index];
104                /*cosine value*/
105                index=(BUFFERSIZE_SINE-1)-index;
106                q_path=sine_buffer[index];
107             }
108             else if(index<2*(BUFFERSIZE_SINE-1))
109             {
110                index=index % (BUFFERSIZE_SINE-1);
111                /*sine value*/
112                index=(BUFFERSIZE_SINE-1)-index;
113                i_path=sine_buffer[index];
114                /*cosine value*/
115                index=(BUFFERSIZE_SINE-1)-index;
116                q_path=-sine_buffer[index];
117             }
118             else if (index<3*(BUFFERSIZE_SINE-1))
119             {
120                index=index % (BUFFERSIZE_SINE-1);
121                /*sine value*/
122                i_path=-sine_buffer[index];
123                /*cosine value*/
124                index=(BUFFERSIZE_SINE-1)-index;
125                q_path=-sine_buffer[index];
126             }
127             else if (index<4*(BUFFERSIZE_SINE-1))
128             {
129                index=index % (BUFFERSIZE_SINE-1);
130                /*sine value*/
131                index=(BUFFERSIZE_SINE-1)-index;
132                i_path=-sine_buffer[index];
133                /*cosine value*/
134                index=(BUFFERSIZE_SINE-1)-index;
135                q_path=sine_buffer[index];
136             }
137             else
138             {
139                index=index % (BUFFERSIZE_SINE-1);
140                /*sine value*/
141                i_path=sine_buffer[index];
142                /*cosine value*/
143                index=(BUFFERSIZE_SINE-1)-index;
144                q_path=sine_buffer[index];
145             }
146          }
147          else
148          {
149             if(index>-(BUFFERSIZE_SINE-1))
150             {
151                index=-index;
152                /*sine value*/
153                i_path=-sine_buffer[index];
154                /*cosine value*/
155                index=(BUFFERSIZE_SINE-1)-index;
156                q_path=sine_buffer[index];
157             }
158             else if(index>-2*(BUFFERSIZE_SINE-1))
159             {
160                index=-(index % (BUFFERSIZE_SINE-1));
161                /*sine value*/
162                index=(BUFFERSIZE_SINE-1)-index;
163                i_path=-sine_buffer[index];
164                /*cosine value*/
165                index=(BUFFERSIZE_SINE-1)-index;
166                q_path=-sine_buffer[index];
167             }
168             else if (index>-3*(BUFFERSIZE_SINE-1))
169             {
170                index=-(index % (BUFFERSIZE_SINE-1));
171                /*sine value*/
172                i_path=sine_buffer[index];
173                /*cosine value*/
174                index=(BUFFERSIZE_SINE-1)-index;
175                q_path=-sine_buffer[index];
176             }
177             else if (index>-4*(BUFFERSIZE_SINE-1))
178             {
179                index=-(index % (BUFFERSIZE_SINE-1));
180                /*sine value*/
181                index=(BUFFERSIZE_SINE-1)-index;
182                i_path=sine_buffer[index];
183                /*cosine value*/
184                index=(BUFFERSIZE_SINE-1)-index;
185                q_path=sine_buffer[index];
186             }
187             else
188             {
189                index=-(index % (BUFFERSIZE_SINE-1));
190                /*sine value*/
191                i_path=-sine_buffer[index];
192                /*cosine value*/
193                index=(BUFFERSIZE_SINE-1)-index;
194                q_path=sine_buffer[index];
195             }
196          }
197       }
198    }
199
200 /*_____________________________________________________________________
201    END OF THE FILE
202  _____________________________________________________________________*/
```

Listing F.29: *Fixed-Point Interpolated :: pll_demodulate.h*

```
1  /*_____________________________________________________________________
2     PLL DEMODULATOR WITH INTERPOLATION - 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
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       FM demodulates the the two basebandsignal 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 F.30: *Fixed-Point Interpolated :: pll_demodulate.c*

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

```
   file  :  pll_demodulate.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 ____________________________________________________________________
*/

/*--------------------------------------------------------------------
   includes
  --------------------------------------------------------------------*/
   #include "pll_demodulate.h"

/*--------------------------------------------------------------------
   global varibles
  --------------------------------------------------------------------*/
   short arg=0;   /*argument of sine and cosine*/
   short i_path=0;  /*I path*/
   short q_path=0;  /*Q path*/

/*--------------------------------------------------------------------
   global defines
  --------------------------------------------------------------------*/
   #define P_PLL  65536    /*P_PLL=2.0, using q15*/
   #define K_PLL  10240    /*K_PLL=2.5, using q12*/
   #define GAIN_PLL 55721  /*GAIN_PLL=1.7, using q15*/
   #define TWO_PI  25735   /*2*PI, using q12*/


/*--------------------------------------------------------------------
   implementation of function
  --------------------------------------------------------------------*/
   void pll_demodulate(short re_buffer[] , short im_buffer[] , short out_buffer[])
   {
      int i;
      int flag=0;  /*flag to check sign of index*/


      short index;   /*index for sine_buffer access*/
      short correction; /*correction used for interpolation*/
      short rest;    /*rest used for interpolation*/
      short demo;    /*value of demodulation*/


      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         /*__calculation of demodulated value, using Q15__*/

         i_path=(re_buffer[i]*i_path)>>15;
         q_path=(im_buffer[i]*q_path)>>15;

         demo=q_path-i_path;
         demo=(demo*P_PLL)>>15;

         out_buffer[i]=(demo*GAIN_PLL)>>15;


         /*__summation of demodulated signal, using Q3.12__*/

         demo=demo>>3;
         arg=arg+((demo*K_PLL)>>12);


         /*__limitation of argument to 2*PI, using Q3.12__*/

         if (arg>TWO_PI)
         {
            arg=arg-TWO_PI;
         }

         else if (arg<-TWO_PI)
         {
            arg=arg+TWO_PI;
         }


         /*__calculation of sine and cosine value of argument__*/

         /*calculating of index for sine_buffer access, using new Q format*/

         index=arg>>REDUCTION;


         if (index<0)
         {
            index=-index;
            flag=1;
         }

         index=(index*STEP_INVERSE)>>Q_FORMAT;
         rest=index%FORMAT;
         index=index>>Q_FORMAT;

         if (flag==1)
         {
            index=-index;
            flag=0;
         }

         /*access on sine_buffer with index*/

         if(index>=0)
         {
            if(index<(BUFFERSIZE_SINE-1))
            {
               /*sine value*/
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               i_path=sine_buffer[index]+correction;
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               q_path=sine_buffer[index]+correction;
            }
            else if(index<2*(BUFFERSIZE_SINE-1))
            {
               index=index % (BUFFERSIZE_SINE-1);
               /*sine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               i_path=sine_buffer[index]+correction;
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               q_path=-(sine_buffer[index]+correction);
            }
            else if (index<3*(BUFFERSIZE_SINE-1))
            {
               index=index % (BUFFERSIZE_SINE-1);
               /*sine value*/
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               i_path=-(sine_buffer[index]+correction);
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               q_path=-(sine_buffer[index]+correction);
            }
            else if (index<4*(BUFFERSIZE_SINE-1))
            {
               index=index % (BUFFERSIZE_SINE-1);
               /*sine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               i_path=-(sine_buffer[index]+correction);
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               q_path=sine_buffer[index]+correction;
            }
            else
            {
               index=index % (BUFFERSIZE_SINE-1);
               /*sine value*/
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               i_path=sine_buffer[index]+correction;
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               q_path=sine_buffer[index]+correction;
            }
         }
         else
         {
```

```
            if(index>-(BUFFERSIZE_SINE-1))
            {
               index=-index;
               /*sine value*/
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               i_path=-(sine_buffer[index]+correction);
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               q_path=sine_buffer[index]+correction;
            }
            else if(index>-2*(BUFFERSIZE_SINE-1))
            {
               index=-(index % (BUFFERSIZE_SINE-1));
               /*sine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               i_path=-(sine_buffer[index]+correction);
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               q_path=-(sine_buffer[index]+correction);
            }
            else if (index>-3*(BUFFERSIZE_SINE-1))
            {
               index=-(index % (BUFFERSIZE_SINE-1));
               /*sine value*/
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               i_path=sine_buffer[index]+correction;
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               q_path=-(sine_buffer[index]+correction);
            }
            else if (index>-4*(BUFFERSIZE_SINE-1))
            {
               index=-(index % (BUFFERSIZE_SINE-1));
               /*sine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               i_path=sine_buffer[index]+correction;
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               q_path=sine_buffer[index]+correction;
            }
            else
            {
               index=-(index % (BUFFERSIZE_SINE-1));
               /*sine value*/
               correction=(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*positive*/
               i_path=-(sine_buffer[index]+correction);
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               correction=-(sine_grad_buffer[index]*rest)>>Q_FORMAT;/*negative*/
               q_path=sine_buffer[index]+correction;
            }
         }
      }
   }

/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

out_filter

Listing F.31: *Fixed-Point :: out_filter.h*

```
/*_____________________________________________________________________
   OUTPUT AND NOISE FILTER - VERSION FIXED POINT 0.5

   file  :  out_filter.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
  _____________________________________________________________________
*/

#ifndef _OUT_FILTER_
#define _OUT_FILTER_

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "global_settings.h"

/*---------------------------------------------------------------------
   global functions
  ---------------------------------------------------------------------*/
   /*
      low-pass filters the out_buffer_down_one signal downsamples it
      and then filters it with an high-pass and writes it to
      out_buffer_down_two

      Parameters :
         out_buffer_down_one  input buffer
         out_buffer_down_two  output buffer
   */
   void out_filter(short out_buffer[] , short out_buffer_down[]);

#endif /* _OUT_FILTER_ */

/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

Listing F.32: *Fixed-Point :: out_filter.c*

```
/*_____________________________________________________________________
   OUTPUT AND NOISE FILTER - VERSION FIXED POINT 0.5

   file  :  out_filter.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
  _____________________________________________________________________
*/

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "out_filter.h"


/*---------------------------------------------------------------------
   global varibles
  ---------------------------------------------------------------------*/

/*---------------------------------------------------------------------
  coefficients of noise low-pass (no_lp) after demodulation
  ---------------------------------------------------------------------*/
   /* Order    : 13           */
   /* Pass-band : 2 dB @ 3500.00 Hz */
   /* Stop-band : 20 dB @ 3999.57 Hz */

   #define NO_LP_CASCADES 7
   short no_lp_a[][2]={ /* a_i_1 a_i_2 */
                        { -2888 ,   0 },
                        { -5861 ,  738 },
                        { -6124 , 2244 },
                        { -6600 , 4962 },
                        { -7352 , 9261 },
                        { -8497 , 15808 },
                        { -10247 , 25815 },
                      };

   short no_lp_b[][3]={ /* b_i_0 b_i_1 b_i_2 */
                        { 13822 , 13822 ,  0 },
                        {  7222 , 14444 , 7222 },
                        {  7783 , 15565 , 7783 },
                        {  8669 , 17339 , 8669 },
                        { 10020 , 20040 , 10020 },
                        {  7947 , 15893 , 7947 },
```

```
                  { 12457 , 24914 , 12457 },
                };

   short no_lp_scale=23904;  /* 29880*0.8 0.911862 to avoid overflows */

   /* delay variabels */
   short no_lp_z[][3]={
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                };
/*-------------------------------------------------------------------
  coefficients of end high-pass (en_hp) before dac
  -------------------------------------------------------------------*/
   /* Order    : 4            */
   /* Pass-band : 0.1 dB @ 300.00 Hz */
   /* Stop-band : 40.0 dB @ 50.00 Hz */

   #define EN_HP_CASCADES 2
   short en_hp_a[][3]={ /* aa_i_1a a_i_1b a_i_2 */
                  { -29180 , -29180 , 26045 },
                  { -31046 , -31046 , 29805 },
                };

   /*
   short en_hp_b[][3]={ b_i_0 b_i_1 b_i_2
                  {    1 ,   -2 ,    1 },
                  {    1 ,   -2 ,    1 },
                };
   */

   short en_hp_scale=13931;   /* 0.5*0.850269 to avoid overflows */

   #define EN_HP_UPSCALE 2

   /* delay variabels */
   short en_hp_z[][3]={
                  { 0,0,0},
                  { 0,0,0},
                };

/*-------------------------------------------------------------------
   implementation of function
  -------------------------------------------------------------------*/
   void out_filter(short out_buffer[] , short out_buffer_down[])
   {
      int i,j,k;

      k=0;

      /*__ the low-pass __*/

      /* scaling */
      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         out_buffer[i]=(out_buffer[i]*no_lp_scale)>>15;
      }

      /* all cascades except the last one */
      for(j=0;j<(NO_LP_CASCADES-1);j++)

      {
         for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
         {
            /* feedback */
            no_lp_z[j][0] = out_buffer[i]-( ( no_lp_a[j][0]*no_lp_z[j][1] + /*a_1*z1
                  */
                                 no_lp_a[j][1]*no_lp_z[j][2] /*a_2 *z2*/
                               )>>15);

            /* foreward */
            out_buffer[i]= ( no_lp_b[j][0]*no_lp_z[j][0]+ /* b_0 * z0 */
                        no_lp_b[j][1]*no_lp_z[j][1]+ /* b_1 * z1 */
                        no_lp_b[j][2]*no_lp_z[j][2] /* b_2 * z2 */
                       )>>15;


            /* shift delays */
            no_lp_z[j][2]=no_lp_z[j][1];
            no_lp_z[j][1]=no_lp_z[j][0];
         }
      }

      /* last cascade including downsampling*/
      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         /* feedback */
         no_lp_z[j][0] = out_buffer[i]-( ( no_lp_a[j][0]*no_lp_z[j][1] + /*a_1*z1*/
                                 no_lp_a[j][1]*no_lp_z[j][2]  /*a_2 *z2*/
                               )>>15);

         /* foreward ncluding downsampling -> take every 2nd value*/
         if(i%DOWNSAMPLE_TWO==0)
         {
            out_buffer_down[k]= ( no_lp_b[j][0]*no_lp_z[j][0]+ /* b_0 * z0 */
                           no_lp_b[j][1]*no_lp_z[j][1]+ /* b_1 * z1 */
                           no_lp_b[j][2]*no_lp_z[j][2] /* b_2 * z2 */
                          )>>15;
            k++;
         }

         /* shift delays */
         no_lp_z[j][2]=no_lp_z[j][1];
         no_lp_z[j][1]=no_lp_z[j][0];
      }


      /*__ the high-pass __*/

      /* scaling */
      for(i=0;i<BUFFERSIZE_OUTPUT;i++)
      {
         out_buffer_down[i]=(out_buffer_down[i]*en_hp_scale)>>15;
      }

      /* high-pass in transposed form */
      for(j=0;j<(EN_HP_CASCADES);j++)
      {
         for(i=0;i<BUFFERSIZE_OUTPUT;i++)
         {
            en_hp_z[j][2]= en_hp_z[j][1]+out_buffer_down[i];

            en_hp_z[j][1]= -out_buffer_down[i]-out_buffer_down[i]-
                        ((en_hp_a[j][0]*en_hp_z[j][2]+
                        en_hp_a[j][1]*en_hp_z[j][2])>>15)+
                        en_hp_z[j][0];

            en_hp_z[j][0]= out_buffer_down[i]-
                       ((en_hp_a[j][2]*en_hp_z[j][2])>>15);
            out_buffer_down[i]=(en_hp_z[j][2]);
         }
      }
      for(i=0;i<BUFFERSIZE_OUTPUT;i++)
      {
         out_buffer_down[i]=out_buffer_down[i]*EN_HP_UPSCALE;
      }

   }

/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

F.1.4. Fixed-Point Optimized

global_settings

Listing F.33: *Fixed-Point Optimized :: global_settings.h*

```
/*_____________________________________________________________________
```

```
   GLOBAL SETTINGS - VERSION FIXED POINT 1

   file  :  global_settings.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 ____________________________________________________________________
*/

#define BUFFERSIZE 128
#define DOWNSAMPLE_ONE 4
#define DOWNSAMPLE_TWO 2

#define BUFFERSIZE_INPUT  (BUFFERSIZE)
#define BUFFERSIZE_DEMODULATOR (BUFFERSIZE/DOWNSAMPLE_ONE)
#define BUFFERSIZE_OUTPUT  (BUFFERSIZE/(DOWNSAMPLE_ONE*DOWNSAMPLE_TWO))

#define PI 3.14159265359

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

fm_dem_main

Listing F.34: *Fixed-Point Optimized :: fm_dem_main.c*

```
/*____________________________________________________________________
   FM DEMODULATION MAIN - VERSION FIXED POINT 1

   file  :  fm_dem_main.c
   date  :  nov 2001
   by    :  c. haller
            f. schnyder
 ____________________________________________________________________
*/

/*--------------------------------------------------------------------
   demodulator define
  --------------------------------------------------------------------*/
   #define FM_MIXED_DEM    10
   #define FM_PLL_DEM      20

   #define FM_DEMODULATOR   FM_MIXED_DEM

/*--------------------------------------------------------------------
   includes
  --------------------------------------------------------------------*/

   #include <std.h>
   #include <swi.h>
   #include <csl.h>
   #include <bsl.h>

   #include "global_settings.h"
   #include "adc_THS1408.h"
   #include "dac_codec.h"
   #include "quad_mix.h"
   #include "out_filter.h"
   #if   (FM_DEMODULATOR==FM_PLL_DEM)
      #include "pll_demodulate.h"
   #elif (FM_DEMODULATOR==FM_MIXED_DEM)
      #include "mixed_demodulate.h"
   #else
      #error No demodulator algorithm defined
   #endif



/*--------------------------------------------------------------------
   debug defines
  --------------------------------------------------------------------*/
   #define FM_DEM_CF_DEBUG    0
      /* to print comments to debug_log*/

   #define FM_DEM_CF_INT_OCCUR_LED  0
      /* Led toggles when an interrupt occurs   */
      /* LED 1 codec interrupt (output)    */
      /* LED 2 edma interrupt (input)      */
      /* LED 3 timming error               */

   #define FM_DEM_CF_INT_TIME_LED  0
      /* Led ON by interrupt start OFF at the end  */
      /* LED 1 codec interrupt (output)    */
      /* LED 2 edma interrupt (input)      */
      /* LED 3 software interrupt          */

   #define FM_DEM_CF_INT_TIME_STS  0
      /* Enable STS Objects:                */
      /* -dac_sts                           */
      /* -demodulate_sts                    */
      /* -edma_sts                          */
      /* to measure the duration time of the interrupts */

   #define FM_DEM_CF_FUNC_TIME_STS  0
      /* Enable STS Objects:                */
      /* -mix_sts                           */
      /* -dem_sts                           */
      /* -filter_sts                        */
      /* to measure the duration time of the functions */

   #if FM_DEM_CF_DEBUG
      #include <log.h>
      extern far LOG_Obj debug_log;
   #endif

   #if FFM_DEM_CF_INT_OCCUR_LED
      #include <bsl_led.h>
   #endif

   #if FFM_DEM_CF_INT_TIME_LED
      #include <bsl_led.h>
   #endif

   #if FM_DEM_CF_INT_TIME_STS
      #include <clk.h>
      #include <sts.h>
      extern far STS_Obj dac_sts;
      extern far STS_Obj demodulate_sts;
      extern far STS_Obj edma_sts;
   #endif

   #if FM_DEM_CF_FUNC_TIME_STS
      #include <clk.h>
      #include <sts.h>
      extern far STS_Obj mix_sts;
      extern far STS_Obj dem_sts;
      extern far STS_Obj filter_sts;
   #endif

/*--------------------------------------------------------------------
   DSP/BIOS extern varibles
  --------------------------------------------------------------------*/

   extern far SWI_Obj demodulate_swi;
   extern HWI_enable();
   extern HWI_disable();

/*--------------------------------------------------------------------
   global varibles
  --------------------------------------------------------------------*/

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

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

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

```
131    short out_buffer_B[BUFFERSIZE_OUTPUT];
132
133
134
135    short* in_buffer_ptr;
136    short* out_buffer_ptr;
137    Bool  a_inEDMA=TRUE;
138    short outcounter=0;
139
140 /*-----------------------------------------------------------------------
141    the main function
142   -----------------------------------------------------------------------*/
143    void main()
144    {
145       HWI_disable();
146
147       #if FM_DEM_CF_DEBUG
148          LOG_printf(&debug_log,"Info : Begin Of main ");
149       #endif
150
151       CSL_init();  /* on chip init   */
152       BSL_init();  /* on board init  */
153
154       ths1408_init(in_buffer_A,in_buffer_B);
155       codec_init();
156
157       in_buffer_A[0]=10;
158
159       HWI_enable();
160
161       ths1408_start();
162       codec_start();
163
164
165       #if FM_DEM_CF_INT_TIME_LED
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_gethtime());
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_gethtime());
231       #endif
232
233       #if FM_DEM_CF_INT_TIME_LED
234          LED_off(LED_2);
235       #endif
236    }
237
238    void dacInt(void)
239    {
240       #if FM_DEM_CF_INT_OCCUR_LED
241          LED_toggle(LED_1);
242       #endif
243
244       #if FM_DEM_CF_INT_TIME_LED
245          LED_on(LED_1);
246       #endif
247
248       #if FM_DEM_CF_INT_TIME_STS
249          STS_set(&dac_sts,CLK_gethtime());
250       #endif
251
252       /* interpolate a missing sample */
253       if (outcounter>15)
254       {
255          #if FM_DEM_CF_INT_OCCUR_LED
256             LED_toggle(LED_3);
257          #endif
258          /* select the buffer */
259          if(a_inEDMA)
260          {
261             AD535_HWI_write(outAD535h,(( out_buffer_A[15]+out_buffer_B[0])/2));
262             #if FM_DEM_CF_DEBUG
263                LOG_printf(&debug_log,"Out A error :%d",outcounter);
264             #endif
265          }else
266          {
267             AD535_HWI_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_HWI_write(outAD535h,out_buffer_A[outcounter]);
278             #if FM_DEM_CF_DEBUG
279                LOG_printf(&debug_log,"Out A :%d",outcounter);
280             #endif
281          }else
282          {
283             AD535_HWI_write(outAD535h,out_buffer_B[outcounter]);
284             #if FM_DEM_CF_DEBUG
285                LOG_printf(&debug_log,"Out B :%d",outcounter);
286             #endif
287          }
288          /* update outcounter */
289          outcounter++;
290       }
```

```
291
292       #if FM_DEM_CF_INT_TIME_STS
293          STS_delta(&dac_sts,CLK_gethtime());
294       #endif
295
296       #if FM_DEM_CF_INT_TIME_LED
297          LED_off(LED_1);
298       #endif
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_gethtime());
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_gethtime());
324       #endif
325       quad_mix(in_buffer_ptr,re_buffer,im_buffer);
326       #if FM_DEM_CF_FUNC_TIME_STS
327          STS_delta(&mix_sts,CLK_gethtime());
328       #endif
329
330       #if FM_DEM_CF_FUNC_TIME_STS
331          STS_set(&dem_sts,CLK_gethtime());
332       #endif
333       #if   (FM_DEMODULATOR==FM_PLL_DEM)
334          pll_demodulate(re_buffer,im_buffer,out_buffer);
335       #elif (FM_DEMODULATOR==FM_MIXED_DEM)
336          mixed_demodulate(re_buffer,im_buffer,out_buffer);
337       #else
338          #error No demodulator algorithm defined
339       #endif
340       #if FM_DEM_CF_FUNC_TIME_STS
341          STS_delta(&dem_sts,CLK_gethtime());
342       #endif
343
344       #if FM_DEM_CF_FUNC_TIME_STS
345          STS_set(&filter_sts,CLK_gethtime());
346       #endif
347       out_filter(out_buffer,out_buffer_ptr);
348       #if FM_DEM_CF_FUNC_TIME_STS
349          STS_delta(&filter_sts,CLK_gethtime());
350       #endif
351
352       #if FM_DEM_CF_INT_TIME_STS
353          STS_delta(&demodulate_sts,CLK_gethtime());
354       #endif
355
356       #if FM_DEM_CF_INT_TIME_LED
357          LED_off(LED_3);
358       #endif
359    }
```

quad_mix

Listing F.35: *Fixed-Point Optimized :: quad_mix.h*

```
1  /*_____________________________________________________________
2     QUADRATURE MIXER - VERSION FIXED POINT 1
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    extern 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.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 varibles
18   ---------------------------------------------------------------*/
19
20 /*---------------------------------------------------------------
21    coefficients of sine and cosine oszilator
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 variabels */
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
```

```
   #define MIX_LP_CASCADES 3
   short mix_lp_a[][3]={ /* a_i_1a a_i_1b a_i_2 */
                         { -8092 , -8092 ,  0 },
                         { -17444 , -17444 , 11166 },
                         { -21907 , -21907 , 22406 },
                       };

   short mix_lp_b[][3]={ /* b_i_0 b_i_1 b_i_2 */
                         {  4523 , 4523 ,   0 },
                         {  2725 , 5450 , 2725 },
                         {  8537 , 17074 , 8537 },
                       };

   /*Scaling included in output of oszis*/

   /* delay variabels */
   short mix_lp_z_re[][3]={
                         { 0,0,0},
                         { 0,0,0},
                         { 0,0,0},
                       };
   short mix_lp_z_im[][3]={
                         { 0,0,0},
                         { 0,0,0},
                         { 0,0,0},
                       };

   /* delay variabels */
   short as_mix_lp_z_re[][3]={
                         { 0,0,0},
                         { 0,0,0},
                         { 0,0,0},
                       };
   short as_mix_lp_z_im[][3]={
                         { 0,0,0},
                         { 0,0,0},
                         { 0,0,0},
                       };

/*-------------------------------------------------------------------
   implementation of function
 -------------------------------------------------------------------*/

   /* implementation in linear assembly in
      file quad_mix_lin_ass !!!      */

/*____________________________________________________________________
   END OF THE FILE
 ____________________________________________________________________*/
```

mixed_demodulate

Listing F.37: *Fixed-Point Optimized :: mixed_demodulate.h*

```
/*____________________________________________________________________
   MIXED DEMODULATOR - VERSION FIXED POINT 1

   file  :  mixed_demodulate.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 ____________________________________________________________________
*/

#ifndef _MIXED_DEMODULATE_
#define _MIXED_DEMODULATE_

/*-------------------------------------------------------------------
   includes
 -------------------------------------------------------------------*/
   #include "global_settings.h"
   #include "atan.h"

/*-------------------------------------------------------------------
   global functions
 -------------------------------------------------------------------*/
   /*
      FM demodulates the the two basebandsignal re_buffer and im_buffer
      into the out_buffer with the mixed algorithm

      Parameters :
         re_buffer reel FM path in baseband
         im_buffer imag FM path in baseband
         out_buffer demodulated signal
   */
   void mixed_demodulate(short re_buffer[] , short im_buffer[] , short out_buffer[]);

#endif /* _MIXED_DEMODULATE_ */

/*____________________________________________________________________
   END OF THE FILE
 ____________________________________________________________________*/
```

Listing F.38: *Fixed-Point Optimized :: mixed_demodulate.c*

```
/*____________________________________________________________________
   MIXED DEMODULATOR - VERSION FIXED POINT 1

   file  :  mixed_demodulate.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 ____________________________________________________________________
*/

/*-------------------------------------------------------------------
   includes
 -------------------------------------------------------------------*/
   #include "mixed_demodulate.h"

/*-------------------------------------------------------------------
   global varibles
 -------------------------------------------------------------------*/
   /* delay varibales of the mixed demodulator */
   short dem_z_re=16384; /* !=0 to avoid division with 0 */
   short dem_z_im=16384; /* !=0 to avoid division with 0 */

/*-------------------------------------------------------------------
   implementation of function
 -------------------------------------------------------------------*/
   void mixed_demodulate(short re_buffer[restrict] , short im_buffer[restrict] ,
         short out_buffer[restrict])
   {
      int   i;
      int   re_path=0;
      int   im_path=0;
      int   dem_arg=0;

      WORD_ALIGNED(re_buffer);
      WORD_ALIGNED(im_buffer);
      WORD_ALIGNED(out_buffer);

      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         re_path = ( im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
         im_path = ( re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */

         /* shift delays */
         dem_z_re=re_buffer[i];
         dem_z_im=im_buffer[i];

         /* calculate the argument */
         dem_arg=re_path/(im_path>>14); /* Q17.14 */

         /* atan lookup */
         if(dem_arg>0)     /*positiv*/
         {
            dem_arg=dem_arg>>(ATAN_SHIFT);
            if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
            out_buffer[i]=atan_buffer[dem_arg];
         }else             /*negative*/
         {
            dem_arg=-dem_arg;
            dem_arg=dem_arg>>(ATAN_SHIFT);
```

```
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  /*_____________________________________________________________________
67     END OF THE FILE
68    _____________________________________________________________________*/
```

Listing F.39: *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 the two basebandsignal re_buffer and im_buffer
25        into the out_buffer with the mixed 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 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 F.40: *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 varibles
18    --------------------------------------------------------------------*/
19     /* delay varibales 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] ,
             short out_buffer[restrict])
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        WORD_ALIGNED(re_buffer);
35        WORD_ALIGNED(im_buffer);
36        WORD_ALIGNED(out_buffer);
37
38
39        for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
40        {
41
42           re_path = ( im_buffer[i]*dem_z_re - re_buffer[i]*dem_z_im)<<1; /* Q.31 */
43           im_path = ( re_buffer[i]*dem_z_re + im_buffer[i]*dem_z_im)<<1; /* Q.31 */
44
45           /* shift delays */
46           dem_z_re=re_buffer[i];
47           dem_z_im=im_buffer[i];
48
49           /* calculate the argument */
50           dem_arg=re_path/(im_path>>14); /* Q17.14 */
51
52           /* atan lookup */
53           if(dem_arg>0)     /*positiv*/
54           {
55              cutoff=dem_arg%ATAN_INTERPOL;
56              dem_arg=dem_arg>>(ATAN_SHIFT);
57              if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
58              out_buffer[i]=atan_buffer[dem_arg]+((cutoff*atan_grad_buffer[dem_arg])>>
                    ATAN_SHIFT);
59           }else             /*negative*/
60           {
61
62              dem_arg=-dem_arg;
63              cutoff=dem_arg%ATAN_INTERPOL;
64              dem_arg=dem_arg>>(ATAN_SHIFT);
65              if(dem_arg>=BUFFERSIZE_ATAN) dem_arg=BUFFERSIZE_ATAN-1;
66              out_buffer[i]=-(atan_buffer[dem_arg]+((cutoff*atan_grad_buffer[dem_arg])
                    >>ATAN_SHIFT));
67           }
68        }
69     }
70
71  /*_____________________________________________________________________
72     END OF THE FILE
73    _____________________________________________________________________*/
```

pll_demodulate

Listing F.41: *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  /*--------------------------------------------------------------------
```

```
   includes
  ------------------------------------------------------------------*/
   #include "global_settings.h"
   #include "sine.h"

/*------------------------------------------------------------------
   global functions
  ------------------------------------------------------------------*/
   /*
      FM demodulates the the two basebandsignal re_buffer and im_buffer
      into the out_buffer with the pll algorithm

      Parameters :
         re_buffer reel FM path in baseband
         im_buffer imag FM path in baseband
         out_buffer demodulated signal
   */
   void pll_demodulate(short re_buffer[] , short im_buffer[] , short out_buffer[]);

#endif /* _PLL_DEMODULATE_ */

/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

Listing F.42: *Fixed-Point Optimized :: pll_demodulate.c*

```
/*_____________________________________________________________________
   PLL DEMODULATOR - VERSION FIXED POINT 1

   file  :  pll_demodulate.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
  _____________________________________________________________________
*/

/*------------------------------------------------------------------
   includes
  ------------------------------------------------------------------*/
   #include "pll_demodulate.h"

/*------------------------------------------------------------------
   global varibles
  ------------------------------------------------------------------*/
   short arg=0;   /*argument of sine and cosine*/
   short i_path=1000; /*I path*/
   short q_path=0;  /*Q path*/

/*------------------------------------------------------------------
   global defines
  ------------------------------------------------------------------*/
   #define P_PLL  65536    /*P_PLL=2.0, using q15*/
   #define K_PLL  10240    /*K_PLL=2.5, using q3.12*/
   #define GAIN_PLL 55705  /*GAIN_PLL=1.7, using q15*/
   #define TWO_PI  25735   /*2*PI, using q3.12*/


/*------------------------------------------------------------------
   implementation of function
  ------------------------------------------------------------------*/
   void pll_demodulate(short re_buffer[restrict] , short im_buffer[restrict] , short
         out_buffer[restrict])
   {
      int i;
      int flag=0;

      short demo;    /*index for sine_buffer access*/
      short index;   /*value of demodulation*/

      WORD_ALIGNED(re_buffer);
      WORD_ALIGNED(im_buffer);
      WORD_ALIGNED(out_buffer);


      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         /*__calculation of demodulated value, using Q15__*/

         i_path=(re_buffer[i]*i_path)>>15;
         q_path=(im_buffer[i]*q_path)>>15;

         demo=q_path-i_path;
         demo=(demo*P_PLL)>>15;

         out_buffer[i]=(demo*GAIN_PLL)>>15;


         /*__summation of demodulated signal, using Q3.12__*/

         demo=demo>>3;
         arg=arg+((demo*K_PLL)>>12);


         /*__limitation of argument to 2*PI, using Q3.12__*/

         if (arg>TWO_PI)
         {
            arg=arg-TWO_PI;
         }

         else if (arg<-TWO_PI)
         {
            arg=arg+TWO_PI;
         }


         /*__calculation of sine and cosine value of argument__*/

         /*calculating of index for sine_buffer access, using new Q format*/

         index=arg>>REDUCTION;

         if (index<0)
         {
            index=-index;
            flag=1;
         }

         index=(index*STEP_INVERSE)>>(2*Q_FORMAT);

         if (flag==1)
         {
            index=-index;
            flag=0;
         }

         /*access on sine_buffer with index*/

         if(index>=0)
         {
            if(index<(BUFFERSIZE_SINE-1))
            {
               /*sine value*/
               i_path=sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=sine_buffer[index];
            }
            else if(index<2*(BUFFERSIZE_SINE-1))
            {
               index=index % (BUFFERSIZE_SINE-1);
               /*sine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               i_path=sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=-sine_buffer[index];
            }
            else if (index<3*(BUFFERSIZE_SINE-1))
            {
               index=index % (BUFFERSIZE_SINE-1);
               /*sine value*/
               i_path=-sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=-sine_buffer[index];
            }
            else if (index<4*(BUFFERSIZE_SINE-1))
```

```
            {
               index=index % (BUFFERSIZE_SINE-1);
               /*sine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               i_path=-sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=sine_buffer[index];
            }
            else
            {
               index=index % (BUFFERSIZE_SINE-1);
               /*sine value*/
               i_path=sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=sine_buffer[index];
            }
         }
         else
         {
            if(index>-(BUFFERSIZE_SINE-1))
            {
               index=-index;
               /*sine value*/
               i_path=-sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=sine_buffer[index];
            }
            else if(index>-2*(BUFFERSIZE_SINE-1))
            {
               index=-(index % (BUFFERSIZE_SINE-1));
               /*sine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               i_path=-sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=-sine_buffer[index];
            }
            else if (index>-3*(BUFFERSIZE_SINE-1))
            {
               index=-(index % (BUFFERSIZE_SINE-1));
               /*sine value*/
               i_path=sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=-sine_buffer[index];
            }
            else if (index>-4*(BUFFERSIZE_SINE-1))
            {
               index=-(index % (BUFFERSIZE_SINE-1));
               /*sine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               i_path=sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=sine_buffer[index];
            }
            else
            {
               index=-(index % (BUFFERSIZE_SINE-1));
               /*sine value*/
               i_path=-sine_buffer[index];
               /*cosine value*/
               index=(BUFFERSIZE_SINE-1)-index;
               q_path=sine_buffer[index];
            }
         }
      }
   }

/*_____________________________________________________________________
   END OF THE FILE
 _____________________________________________________________________*/
```


out_filter

Listing F.43: *Fixed-Point Optimized :: out_filter.h*

```
/*_____________________________________________________________________
   OUTPUT AND NOISE FILTER - VERSION FIXED POINT 1

   file  :  out_filter.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _____________________________________________________________________
*/

#ifndef _OUT_FILTER_
#define _OUT_FILTER_

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "global_settings.h"

/*---------------------------------------------------------------------
   global functions
  ---------------------------------------------------------------------*/
   /*
      low-pass filters the out_buffer_down_one signal downsamples it
      and then filters it with an high-pass and writes it to
      out_buffer_down_two

      Parameters :
         out_buffer_down_one  input buffer
         out_buffer_down_two  output buffer
   */
   void out_filter(short out_buffer[] , short out_buffer_down[]);

#endif /* _OUT_FILTER_ */

/*_____________________________________________________________________
   END OF THE FILE
 _____________________________________________________________________*/
```

Listing F.44: *Fixed-Point Optimized :: out_filter.c*

```
/*_____________________________________________________________________
   OUTPUT AND NOISE FILTER - VERSION FIXED POINT 1

   file  :  out_filter.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _____________________________________________________________________
*/

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "out_filter.h"


/*---------------------------------------------------------------------
   global varibles
  ---------------------------------------------------------------------*/

/*---------------------------------------------------------------------
  coefficients of noise low-pass (no_lp) after demodulation
  ---------------------------------------------------------------------*/
  /* Order   : 13           */
  /* Pass-band : 2 dB @ 3500.00 Hz */
  /* Stop-band : 20 dB @ 3999.57 Hz */

  #define NO_LP_CASCADES 7
  short no_lp_a[][2]={ /* a_i_1 a_i_2 */
                  { -2888 ,    0 },
                  { -5861 ,  738 },
                  { -6124 , 2244 },
                  { -6600 , 4962 },
                  { -7352 , 9261 },
```

```
                  { -8497 , 15808 },
                  { -10247 , 25815 },
               };

   short no_lp_b[][3]={ /* b_i_0 b_i_1 b_i_2 */
                  { 13822 , 13822 ,  0 },
                  {  7222 , 14444 , 7222 },
                  {  7783 , 15565 , 7783 },
                  {  8669 , 17339 , 8669 },
                  { 10020 , 20040 , 10020 },
                  {  7947 , 15893 , 7947 },
                  {  5296 , 10592 , 5296 },
               };

   /*short no_lp_scale= 23904;*/  /* 0.911862*0.8 to avoid overflows */

   /* delay variabels */
   short no_lp_z[][3]={
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
                  { 0,0,0},
               };

/*-------------------------------------------------------------------
   coefficients of end high-pass (en_hp) before dac
  -------------------------------------------------------------------*/
   /* Order    : 4           */
   /* Pass-band : 0.1 dB @ 300.00 Hz */
   /* Stop-band : 40.0 dB @ 50.00 Hz */

   #define EN_HP_CASCADES 2
   short en_hp_a[][3]={ /* aa_i_1a a_i_1b a_i_2 */
                  { -29180 , -29180 , 26045 },
                  { -31046 , -31046 , 29805 },
               };

   /*
   short en_hp_b[][3]={ b_i_0 b_i_1 b_i_2
                  {    1 ,   -2 ,    1 },
                  {    1 ,   -2 ,    1 },
               };
   */

   short en_hp_scale=13931;   /* 0.5*0.850269 to avoid overflows */

   #define EN_HP_UPSCALE 2

   /* delay variabels */
   short en_hp_z[][3]={
                  { 0,0,0},
                  { 0,0,0},
               };

/*-------------------------------------------------------------------
   implementation of function
  -------------------------------------------------------------------*/
   void out_filter(short out_buffer[restrict] , short out_buffer_down[restrict])
   {
      int i,j,k;

      WORD_ALIGNED(out_buffer);
      WORD_ALIGNED(out_buffer_down);

      k=0;

      /*__ the low-pass __*/

      /* all cascades except fisrt the last one */
      for(j=0;j<(NO_LP_CASCADES-1);j++)

      {
         for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
         {
            /* feedback */
            no_lp_z[j][0] = out_buffer[i]-( ( no_lp_a[j][0]*no_lp_z[j][1] + /*a_1*z1
                  */
                                    no_lp_a[j][1]*no_lp_z[j][2] /*a_2 *z2*/
                                  )>>15);

            /* foreward */
            out_buffer[i]= ( no_lp_b[j][0]*no_lp_z[j][0]+ /* b_0 * z0 */
                        no_lp_b[j][1]*no_lp_z[j][1]+ /* b_1 * z1 */
                        no_lp_b[j][2]*no_lp_z[j][2] /* b_2 * z2 */
                       )>>15;


            /* shift delays */
            no_lp_z[j][2]=no_lp_z[j][1];
            no_lp_z[j][1]=no_lp_z[j][0];
         }
      }

      /* last cascade including downsampling*/
      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         /* feedback */
         no_lp_z[NO_LP_CASCADES-1][0] = out_buffer[i]-( ( no_lp_a[NO_LP_CASCADES
               -1][0]*no_lp_z[NO_LP_CASCADES-1][1] + /*a_1*z1*/
                                    no_lp_a[NO_LP_CASCADES-1][1]*no_lp_z[
                                          NO_LP_CASCADES-1][2]  /*a_2 *z2*/
                                 )>>15);

         /* foreward ncluding downsampling -> take every 2nd value*/
         if(i%DOWNSAMPLE_TWO==0)
         {
            out_buffer_down[k]= ( no_lp_b[NO_LP_CASCADES-1][0]*no_lp_z[NO_LP_CASCADES
                  -1][0]+ /* b_0 * z0 */
                              no_lp_b[NO_LP_CASCADES-1][1]*no_lp_z[NO_LP_CASCADES
                                    -1][1]+ /* b_1 * z1 */
                              no_lp_b[NO_LP_CASCADES-1][2]*no_lp_z[NO_LP_CASCADES
                                    -1][2] /* b_2 * z2 */
                             )>>15;
            k++;
         }

         /* shift delays */
         no_lp_z[NO_LP_CASCADES-1][2]=no_lp_z[NO_LP_CASCADES-1][1];
         no_lp_z[NO_LP_CASCADES-1][1]=no_lp_z[NO_LP_CASCADES-1][0];
      }


      /*__ the high-pass __*/

      /* high-pass in transposed form */
      for(j=0;j<(EN_HP_CASCADES);j++)
      {
         for(i=0;i<BUFFERSIZE_OUTPUT;i++)
         {
            en_hp_z[j][2]= en_hp_z[j][1]+out_buffer_down[i];

            en_hp_z[j][1]= -out_buffer_down[i]-out_buffer_down[i]-
                        ((en_hp_a[j][0]*en_hp_z[j][2]+
                        en_hp_a[j][1]*en_hp_z[j][2])>>15)+
                        en_hp_z[j][0];

            en_hp_z[j][0]= out_buffer_down[i]-
                        ((en_hp_a[j][2]*en_hp_z[j][2])>>15);
            out_buffer_down[i]=(en_hp_z[j][2]);
         }
      }
      for(i=0;i<BUFFERSIZE_OUTPUT;i++)
      {
         out_buffer_down[i]=out_buffer_down[i]*EN_HP_UPSCALE;
      }

   }

/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

F.1.5. Floating-Point Adaptive

mixed_demodulate

Listing F.45: *Floating-Point Optimized Adaptive :: mixed_demodulate.h*

```
/*_____________________________________________________________________
   MIXED DEMODULATOR WITH CARRIER CORRECTION
                              - VERSION FLOATING POINT 1.1

   file  :  mixed_demodulate.h
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _____________________________________________________________________
*/

#ifndef _MIXED_DEMODULATE_
#define _MIXED_DEMODULATE_

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "global_settings.h"
   #include <math.h>

/*---------------------------------------------------------------------
   global functions
  ---------------------------------------------------------------------*/
   /*
      The re_buffer_down_one and im_buffer_down_one are demodulated
      with the mixed demodulator algorithm to out_buffer_down_one.

      Parameters :

      re_buffer_down_one  buffer of downsampled I signal
      im_buffer_down_one  buffer of downsampled Q signal
      out_buffer_down_one  buffer of demodulated signal
   */
   void mixed_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] ,
         float out_buffer_down_one[]);

#endif /* _MIXED_DEMODULATE_ */

/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

Listing F.46: *Floating-Point Optimized Adaptive :: mixed_demodulate.c*

```
/*_____________________________________________________________________
   MIXED DEMODULATOR WITH CARRIER CORRECTION
                              - VERSION FLOATING POINT 1.1

   file  :  mixed_demodulate.c
   date  :  oct 2001 - jan 2002
   by    :  c. haller
            f. schnyder
 _____________________________________________________________________
*/

/*---------------------------------------------------------------------
   includes
  ---------------------------------------------------------------------*/
   #include "mixed_demodulate.h"

/*---------------------------------------------------------------------
   global varibles
  ---------------------------------------------------------------------*/
   /*variabels of demodulating (attension division by zero!)*/

   float coold=0.0;/*old value of I-path*/
   float conew=1.0;/*new value of I-path*/
   float siold=0.0;/*old value of Q-path*/
   float sinew=1.0;/*new value of Q-path*/
   float demo=0.0;/*value demodulating*/

   /*constant of demodulating*/

   float decon=0.7;

   /*variabels of adaptive filtering*/

   int counter=0;

   float sum=0;
   float wMe=82561;/*exstant mixer angular frequency*/
   float wMs=0;/*supposed mixer angular frequency*/

   extern float bc2;
   extern float bs2;
   extern float asc2;

   /*constants of adaptive filtering*/

   #define N 32000/*x*BUFFERSIZE_DEMODULATOR*/
   #define derivation 3000

   float T=0.000015625;/*sample time*/
   float adcon=2*PI*derivation/N;


/*---------------------------------------------------------------------
   implementation of function
  ---------------------------------------------------------------------*/
   void mixed_demodulate(float re_buffer_down_one[] , float im_buffer_down_one[] ,
         float out_buffer_down_one[])
   {
      int i;

      for(i=0;i<BUFFERSIZE_DEMODULATOR;i++)
      {
         /*__shifting values__*/
         coold=conew;
         siold=sinew;
         conew=re_buffer_down_one[i];
         sinew=im_buffer_down_one[i];

         /*__calculating demodulated value__*/
         demo=(coold*sinew-siold*conew)/(coold*conew+siold*sinew);/*atan argument*/
         demo=atanf(demo);/*atan value*/
         demo=demo*decon;/*demodulated value*/
         out_buffer_down_one[i]=demo;

         sum=sum+demo;
         counter++;
      }


      if(counter==N)
      {
         wMs=wMe;
         wMe=wMs+sum*adcon;

         bc2=-cosf(wMe*T);
         bs2=sinf(wMe*T);
         asc2=-2*cosf(wMe*T);

         sum=0;
         counter=0;
      }

   }


/*_____________________________________________________________________
   END OF THE FILE
  _____________________________________________________________________*/
```

F.2. Linear Assembly Listings

F.2.1. Fixed-Point Optimized

quad_mix

Listing F.47: *Fixed-Point Optimized :: quad_mix_lin_ass.sa*

```
;_____________________________________________________________________
;  QUADRATURE MIXER - VERSION FIXED POINT 1
;
;
;  file  :  quad_mix.c
;  date  :  oct 2001 - jan 2002
;  by    :  c. haller
;           f. schnyder
;_____________________________________________________________________


   .global _quad_mix
   .global _mix_lp_a
   .global _mix_lp_b
   .global _mix_lp_z_re
   .global _mix_lp_z_im
   .global _gen_b0_cos
   .global _gen_b1_cos
   .global _gen_b1_sin
   .global _gen_a1
   .global _gen_z

   .sect ".text"

_quad_mix: .cproc in_s , re_b_s , im_b_s

      .reg  re_b , im_b
      .reg  cnt
      .reg  ptr_re , ptr_im

      .reg  re_in , re_prod1 , re_prod2
      .reg  im_in , im_prod1 , im_prod2

      .reg  re_a_1_a , re_a_1_b , re_a_2 , re_b_0 , re_b_1 , re_b_2
      .reg  im_a_1_a , im_a_1_b , im_a_2 , im_b_0 , im_b_1 , im_b_2

      .reg  re_z_0 , re_z_1 , re_z_2
      .reg  im_z_0 , im_z_1 , im_z_2

      .no_mdep


;=============================================================
;  BEGIN mixing
;=============================================================
      ; copy address of input buffers
         MV     re_b_s , re_b
         MV     im_b_s , im_b

      ; load filter coefficients
         MVKL  _gen_a1  ,  ptr_re
         MVKL  _gen_b1_sin , ptr_im
         MVKH  _gen_a1  ,  ptr_re
         MVKH  _gen_b1_sin , ptr_im

         LDH   *ptr_re  ,  re_a_1_a
         LDH   *ptr_im  ,  im_b_1

         MVKL  _gen_b0_cos , ptr_re
         MVKL  _gen_b1_cos , ptr_im
         MVKH  _gen_b0_cos , ptr_re
         MVKH  _gen_b1_cos , ptr_im

         LDH   *ptr_re  ,  re_b_0
         LDH   *ptr_im  ,  re_b_1

      ; load delay states
         MVKL  _gen_z  , ptr_re
         MVKH  _gen_z  , ptr_re

         LDH   *++ptr_re, re_z_1
         LDH   *++ptr_re, re_z_2

      ; setup loop
         MVK      128 , cnt

MIX_LOOP_GEN: .trip 128 , 128  ; loop with min 128 iter and max 128 iter

      ; load input values
         LDH   *in_s++ , re_in

      ; feedback of oszi
         MPY   re_a_1_a , re_z_1 , re_z_0 ; z1*a1
         SHR   re_z_0 , 15 , re_z_0  ; back to Q15
         ADD   re_z_0 , re_z_2 , re_z_0 ; (z1*a1)+z2
         NEG   re_z_0 , re_z_0      ; -(z1*a1)+z2

      ; foreward of oszi
         MPY   re_z_0 , re_b_0 , re_prod1

         MPY   re_z_1 , re_b_1 , re_prod2
         MPY   re_z_1 , im_b_1 , im_prod2

         ADD   re_prod1,re_prod2,re_prod2

         SHR   re_prod2 , 15 , re_prod2
         SHR   im_prod2 , 15 , im_prod2

      ; shift delays
         MV       re_z_1 , re_z_2
         MV       re_z_0 , re_z_1

      ; mixing
         MPY      im_prod2 , re_in , im_in
         MPY      re_prod2 , re_in , re_in
         SHR      im_in , 15 , im_in
         SHR      re_in , 15 , re_in

      ; store output values amd set Pointer to next value
         STH      re_in  , *re_b++
         STH      im_in  , *im_b++

      ; increment counter and branch till loop cnt=0
   [cnt]  SUB     cnt, 1, cnt
   [cnt]  B       MIX_LOOP_GEN

      ; store delay states
         MVKL  _gen_z  , ptr_re
         MVKH  _gen_z  , ptr_re

         STH   re_z_1 , *++ptr_re
         STH   re_z_2 , *++ptr_re

;=============================================================
;  END mixing
;=============================================================

;=============================================================
;  BEGIN low-pass filter
;=============================================================

   ;-------------------------------------------------------------
   ;  cascade 0 (first order)
   ;-------------------------------------------------------------


      ; copy address of input buffers
         MV    re_b_s , re_b
         MV    im_b_s , im_b

      ; load filter coefficients
         MVKL  _mix_lp_a  , ptr_re
         MVKL  _mix_lp_b  , ptr_im
         MVKH  _mix_lp_a  , ptr_re
         MVKH  _mix_lp_b  , ptr_im

         LDH   *ptr_re++ , re_a_1_a
```

```
148          LDH   *ptr_im++ , re_b_0
149          LDH   *ptr_re++ , re_a_1_b
150          LDH   *ptr_im++ , re_b_1
151
152          ;copy to have them for both paths
153          MV    re_a_1_a  ,  im_a_1_a
154          MV    re_b_0    ,  im_b_0
155          MV    re_a_1_b ,   im_a_1_b
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_1_a , re_prod1 ; z1 * a1a
178          MPY   im_z_1 , im_a_1_a , im_prod1
179
180          MPY   re_z_1 , re_a_1_b , re_z_0 ; z1 * a1b
181          MPY   im_z_1 , im_a_1_b , 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       ; foreward 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 amd 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
230          ; copy address of input buffers
231             MV     re_b_s , re_b
232             MV     im_b_s , im_b
233
234          ; load coefficients
235             MVKL  _mix_lp_a  , ptr_re
236             MVKL  _mix_lp_b  , ptr_im
237             MVKH  _mix_lp_a  , ptr_re
238             MVKH  _mix_lp_b  , ptr_im
239
240             ADD   6 , ptr_re , ptr_re
241             ADD   6 , ptr_im , ptr_im
242
243             LDH   *ptr_re++ , re_a_1_a
244             LDH   *ptr_im++ , re_b_0
245             LDH   *ptr_re++ , re_a_1_b
246             LDH   *ptr_im++ , re_b_1
247             LDH   *ptr_re++ , re_a_2
248             LDH   *ptr_im++ , re_b_2
249
250             MV    re_a_1_a  ,  im_a_1_a
251             MV    re_b_0    ,  im_b_0
252             MV    re_a_1_b ,   im_a_1_b
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
277    MIX_LOOP_C1: .trip 128 , 128  ; loop with min 128 iter and max 128 iter
278
279          ; load input values
280             LDH      *re_b , re_in
281             LDH      *im_b , im_in
282
283          ; feedback of filter
284             MPY      re_z_1 , re_a_1_a , re_prod1 ; z1 * a1a
285             MPY      im_z_1 , im_a_1_a , im_prod1
286
287             MPY      re_z_1 , re_a_1_b , re_prod2 ; z1 * a1b
288             MPY      im_z_1 , im_a_1_b , im_prod2
289
290             MPY      re_z_2 , re_a_2 , re_z_0 ; z2 * a2
291             MPY      im_z_2 , im_a_2 , im_z_0
292
293             ADD      re_z_0 , re_prod1 , re_z_0 ; z1*a1a + z2*a2
294             ADD      im_z_0 , im_prod1 , im_z_0
295
296             ADD      re_z_0 , re_prod2 , re_z_0 ; z1*a1a + z1*a1b + z2*a2
297             ADD      im_z_0 , im_prod2 , im_z_0
298
299             SHR      re_z_0 , 15    , re_z_0 ; back to Q15
300             SHR      im_z_0 , 15    , im_z_0
301
302             SUB      re_in  , re_z_0 , re_z_0 ; in-(z1*a1a + z1*a1b + z2*a2)
303             SUB      im_in  , im_z_0 , im_z_0
304
305          ; foreward of filter
306             MPY      re_z_0 , re_b_0 , re_prod1 ; z0*b0
307             MPY      im_z_0  , im_b_0 , im_prod1
```

```
308
309         MPY      re_z_1 , re_b_1 , re_prod2 ; z1*b1
310         MPY      im_z_1 , im_b_1 , im_prod2
311
312         MPY      re_z_2 , re_b_2 , re_in  ; z2*b2
313         MPY      im_z_2 , im_b_2 , im_in
314
315         ADD      re_in  , re_prod1 , re_in  ; z0*b0+z2*b2
316         ADD      im_in  , im_prod1 , im_in
317
318         ADD      re_in  , re_prod2 , re_in  ; z0*b0+z1*b1+z2*b2
319         ADD      im_in  , im_prod2 , im_in
320
321         SHR      re_in  , 15    , re_in  ; back to Q15
322         SHR      im_in  , 15    , im_in
323
324      ; shift delays
325         MV       re_z_1 , re_z_2
326         MV       im_z_1 , im_z_2
327
328         MV       re_z_0 , re_z_1
329         MV       im_z_0 , im_z_1
330
331      ; store output values amd set Pointer to next value
332         STH      re_in  , *re_b++
333         STH      im_in  , *im_b++
334
335      ; increment counter and branch till loop cnt=0
336   [cnt] SUB      cnt , 1 , cnt
337   [cnt] B        MIX_LOOP_C1
338
339      ; store delay states
340         MVKL  _mix_lp_z_re  , ptr_re
341         MVKL  _mix_lp_z_im  , ptr_im
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   ;---------------------------------------------------------------
355   ;  cascade 2 (second order)
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         MVKH  _mix_lp_z_re  , ptr_re
392         MVKH  _mix_lp_z_im  , ptr_im
393
394         ADD   6 , ptr_re , ptr_re
395         ADD   6 , ptr_im , ptr_im ; add 12
396
397         ADD   6 , ptr_re , ptr_re
398         ADD   6 , ptr_im , ptr_im ; add 12
399
400         LDH   *++ptr_re, re_z_1
401         LDH   *++ptr_im, im_z_1
402
403         LDH   *++ptr_re, re_z_2
404         LDH   *++ptr_im, im_z_2
405
406      ; setup loop
407         MVK      128 , cnt
408
409   MIX_LOOP_C2: .trip 128 , 128 ; loop with min 128 iter and max 128 iter
410
411      ; load input values
412         LDH   *re_b , re_in
413         LDH   *im_b++ , im_in
414
415      ; feedback of filter
416         MPY   re_z_1 , re_a_1_a , re_prod1 ; z1 * a1a
417         MPY   im_z_1 , im_a_1_a , im_prod1
418
419         MPY   re_z_1 , re_a_1_b , re_prod2 ; z1 * a1b
420         MPY   im_z_1 , im_a_1_b , im_prod2
421
422         MPY   re_z_2 , re_a_2 , re_z_0 ; z2 * a2
423         MPY   im_z_2 , im_a_2 , im_z_0
424
425         ADD   re_z_0 , re_prod1 , re_z_0 ; z1*a1a + z2*a2
426         ADD   im_z_0 , im_prod1 , im_z_0
427
428         ADD   re_z_0 , re_prod2 , re_z_0 ; z1*a1a + z1*a1b + z2*a2
429         ADD   im_z_0 , im_prod2 , im_z_0
430
431         SHR   re_z_0 , 15    , re_z_0 ; back to Q15
432         SHR   im_z_0 , 15    , im_z_0
433
434         SUB   re_in  , re_z_0 , re_z_0 ; in-(z1*a1a + z1*a1b + z2*a2)
435         SUB   im_in  , im_z_0 , im_z_0
436
437      ; foreward of filter
438         AND   0x7 , re_b , ptr_re ; do forward only every 4th time
439         ADD   0x2 , re_b , re_b ; redirect pointer
440
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          MV    re_z_0  , re_z_1
469          MV    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          MVKL  _mix_lp_z_re  , ptr_re
477          MVKL  _mix_lp_z_im  , ptr_im
478          MVKH  _mix_lp_z_re  , ptr_re
479          MVKH  _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=linspace(0,max,N+1);
24 values=atan(v)*scale;
25
26 % scaleing 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
37 % generate .c and .h file
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 % compaire polt
94 testv=0:(2^(inbits+Mbit));
95 testv=testv(1:length(testv)-1);
96 indexv=floor(testv./(2^shift));
97 cutv=testv-((2^shift)*indexv); % modula
98 direct=testv;
99 for i=1:length(testv)
100    direct(i)=values(indexv(i)+1);
101 end
102 interpol=testv;
103 for i=1:length(testv)
104    if indexv(i)<2^Nbit-1
105       interpol(i)=(values(indexv(i)+2)*cutv(i)+values(indexv(i)+1)*((2^shift)-cutv(i)
              ))/2^shift;
106    else
107       interpol(i)=values(indexv(i)+1);
108    end
109 end
110 testv=(testv./(2^inbits));
```

```
plot(testv,direct./2^15,testv,interpol./2^15,testv,atan(testv)*scale);
legend('Direct Lookup','Interpolation Lookup','Atan')
```

Listing F.49: tanTabGenInterp.m

```
%  Arc Tangent Table With Interpolation Generator - Version 0.5
%
%  file  :  tanTabGenInterp.m
%  date  :  oct 2001 - jan 2002
%  by    :  c. haller
%           f. schnyder
%_____________________________________________________________________

%input parameters
name='atan';         % name of the output files
Nbit=4;              % lenght of table = 2^Nbit
scale=0.8;           % scale factor included in table
Mbit=2;              % bits of max input -> table from 0 to 2^bit
inbits=14;           % Q format of input

% calculate rest of parameters
N=2^Nbit;
max=2^Mbit;
shift=inbits-(Nbit-Mbit);
shiftval=2^shift;

% calculation of values
v=linspace(0,max,N+1);
values=atan(v)*scale;

% scaleing to Q15
values=round(values*(2^15));
values=values(1:N);

% avoid overflows (saturation)
for i=1:N
   if values(i)>(2^15-1)
      values(i)=2^15-1;
   end
end
gradient=values(2:N)-values(1:N-1);
gradient(N)=0;

% generate .c and .h file

% .c file
outfile=fopen(strcat(name,'.c'),'w');
fprintf(outfile,'/*_____________________________________________________________________\
      \n');
fprintf(outfile,'  ARCTANGENT FUNCTION TABLE WITH INTERPOLATION \n');
fprintf(outfile,'                        - VERSION FIXED POINT 0.5 \n');
fprintf(outfile,' generated with tanTabGen.m\n');
fprintf(outfile,'  N   = %i\n',N);
fprintf(outfile,'  Range = %i\n',max);
fprintf(outfile,'  Scale = %f\n',scale);
fprintf(outfile,'  \n');
fprintf(outfile,' file : %s\n',strcat(name,'.c'));
fprintf(outfile,' date : oct 2001 - jan 2002\n');
fprintf(outfile,' by :  c. haller\n');
fprintf(outfile,'       f. schnyder\n');
fprintf(outfile,' _____________________________________________________________________\
      n');
fprintf(outfile,'*/\n');
fprintf(outfile,'\n');
fprintf(outfile,' #include "%s"\n',strcat(name,'.h'));
fprintf(outfile,'\n');
fprintf(outfile,' short atan_buffer[]={\n');
fprintf(outfile,'              %i ,\n',values);
fprintf(outfile,'             };\n');
fprintf(outfile,'\n');
fprintf(outfile,' short atan_grad_buffer[]={\n');
fprintf(outfile,'                   %i ,\n',gradient);
fprintf(outfile,'                  };\n');
fprintf(outfile,'\n');
fclose(outfile);

% .h file
outfile=fopen(strcat(name,'.h'),'w');
fprintf(outfile,'/*_____________________________________________________________________\
      \n');
fprintf(outfile,'  ARCTANGENT FUNCTION TABLE WITH INTERPOLATION \n');
fprintf(outfile,'                        - VERSION FIXED POINT 0.5 \n');
fprintf(outfile,' generated with tanTabGen.m\n');
fprintf(outfile,'  N   = %i\n',N);
fprintf(outfile,'  Range = %i\n',max);
fprintf(outfile,'  Scale = %f\n',scale);
fprintf(outfile,'  \n');
fprintf(outfile,' file : %s\n',strcat(name,'.h'));
fprintf(outfile,' date : oct 2001 - jan 2002\n');
fprintf(outfile,' by :  c. haller\n');
fprintf(outfile,'       f. schnyder\n');
fprintf(outfile,' _____________________________________________________________________\
      n');
fprintf(outfile,'*/\n');
fprintf(outfile,'\n');
fprintf(outfile,'#ifndef _%s_\n',upper(name));
fprintf(outfile,'#define _%s_\n',upper(name));
fprintf(outfile,'\n');
fprintf(outfile,' #define BUFFERSIZE_ATAN %i\n',N);
fprintf(outfile,' #define ATAN_SHIFT %i\n',shift);
fprintf(outfile,' #define ATAN_INTERPOL %i\n',shiftval);
fprintf(outfile,'\n');
fprintf(outfile,' extern short atan_buffer[];\n');
fprintf(outfile,'\n');
fprintf(outfile,' extern short atan_grad_buffer[];\n');
fprintf(outfile,'\n');
fprintf(outfile,'#endif /* _%s_ */\n',upper(name));
fprintf(outfile,'\n');
fclose(outfile);

% compaire polt
testv=0:(2^(inbits+Mbit));
testv=testv(1:length(testv)-1);
indexv=floor(testv./(2^shift));
cutv=testv-((2^shift)*indexv); % modula
direct=testv;
for i=1:length(testv)
   direct(i)=values(indexv(i)+1);
end
interpol=testv;
for i=1:length(testv)
   if indexv(i)<2^Nbit-1
      interpol(i)=values(indexv(i)+1)+(gradient(indexv(i)+1)*cutv(i))/2^shift;
   else
      interpol(i)=values(indexv(i)+1);
   end
end
testv=(testv./(2^inbits));
plot(testv,direct./2^15,testv,interpol./2^15,testv,atan(testv)*scale);
legend('Direct Lookup','Interpolation Lookup','Atan')
```

Listing F.50: SineTabGen.m

```
%  Sine Table Generator - Version 1
%
%  file  :  SineTabGen.m
%  date  :  oct 2001 - jan 2002
%  by    :  c. haller
%           f. schnyder
%_____________________________________________________________________

% name of the output files
name='sine';

% file input: bitnumber of buffer lenght (4, 5, 6, 7)
n=7;

% calculation of constants
len=2^n;                    %table length
q_format=15-log2(4*len);      %size of needed Q format
format=2^q_format;          %format to calculate step_inverse
reduction=12-q_format;      %change from Q3.12 to new Q format
step=(pi/2)/(len-1);        %step between the arguments
step_inverse=round((1/step)*format); %step_inverse

% calculation of values
arg=linspace(0,(pi/2),len);
```

```
values=sin(arg);

% scaling to Q15
values=round(values*((2^15)-1)); %conversion from float to q15


% generate .c and .h file
outfile=fopen(strcat(name,'.c'),'w');

fprintf(outfile,'/*____________________________________________________________\
      \n');
fprintf(outfile,'  SINE FUNCTION TABLE \n');
fprintf(outfile,' generated with SineTabGen.m\n');
fprintf(outfile,'  \n');
fprintf(outfile,' file : %s\n',strcat(name,'.c'));
fprintf(outfile,' date : oct 2001 - jan 2002\n');
fprintf(outfile,' by :  c. haller\n');
fprintf(outfile,'       f. schnyder\n');
fprintf(outfile,' ____________________________________________________________\
      n');
fprintf(outfile,'*/\n');
fprintf(outfile,'\n');
fprintf(outfile,' #include "%s"\n',strcat(name,'.h'));
fprintf(outfile,'\n');
fprintf(outfile,' short sine_buffer[]={\n');
fprintf(outfile,'                %i ,\n',values);
fprintf(outfile,'              };\n');
fprintf(outfile,'\n');


fclose(outfile);

outfile=fopen(strcat(name,'.h'),'w');

fprintf(outfile,'/*____________________________________________________________\
      \n');
fprintf(outfile,'  SINE FUNCTION TABLE \n');
fprintf(outfile,' generated with SineTabGen.m\n');
fprintf(outfile,'  \n');
fprintf(outfile,' file : %s\n',strcat(name,'.h'));
fprintf(outfile,' date : oct 2001 - jan 2002\n');
fprintf(outfile,' by :  c. haller\n');
fprintf(outfile,'       f. schnyder\n');
fprintf(outfile,' ____________________________________________________________\
      n');
fprintf(outfile,'*/\n');
fprintf(outfile,'\n');
fprintf(outfile,'#ifndef _%s_\n',upper(name));
fprintf(outfile,'#define _%s_\n',upper(name));
fprintf(outfile,'\n');
fprintf(outfile,' #define BUFFERSIZE_SINE %i\n',len);
fprintf(outfile,' #define STEP_INVERSE %i\n',step_inverse);
fprintf(outfile,' #define Q_FORMAT %i\n',q_format);
fprintf(outfile,' #define FORMAT %i\n',format);
fprintf(outfile,' #define REDUCTION %i\n',reduction);
fprintf(outfile,'\n');
fprintf(outfile,' extern short sine_buffer[];\n');
fprintf(outfile,'\n');
fprintf(outfile,'#endif /* _%s_ */\n',upper(name));
fprintf(outfile,'\n');

fclose(outfile);

plot(values./2^15)
```

Listing F.51: SineTabGenInt.m

```
%  Sine Table With Interpolation Generator - Version 1
%
%  file  :  SineTabGenInt.m
%  date  :  oct 2001 - jan 2002
%  by    :  c. haller
%           f. schnyder
%_____________________________________________________________

% name of the output files
name='sine';

% file input: bitnumber of buffer lenght (4, 5, 6, 7)
n=4;

% calculation of constants
len=2^n;                        %table length
q_format=15-log2(4*len);          %size of needed Q format
format=2^q_format;              %format to calculate step_inverse
reduction=12-q_format;          %change from Q3.12 to new Q format
step=(pi/2)/(len-1);            %step between the arguments
step_inverse=round((1/step)*format); %step_inverse

% calculation of sine table
arg=linspace(0,(pi/2),len);
values=sin(arg);

% scaling to Q15
values=round(values*((2^15)-1)); %conversion from float to q15

%calculation of sine gradient table
gradient=values(2:len)-values(1:len-1);
gradient(len)=0;

% generate .c and .h file
outfile=fopen(strcat(name,'.c'),'w');

fprintf(outfile,'/*____________________________________________________________\
      \n');
fprintf(outfile,'  SINE FUNCTION TABLE \n');
fprintf(outfile,' generated with SineTabGenInt.m\n');
fprintf(outfile,'  \n');
fprintf(outfile,' file : %s\n',strcat(name,'.c'));
fprintf(outfile,' date : oct 2001 - jan 2002\n');
fprintf(outfile,' by :  c. haller\n');
fprintf(outfile,'       f. schnyder\n');
fprintf(outfile,' ____________________________________________________________\
      n');
fprintf(outfile,'*/\n');
fprintf(outfile,'\n');
fprintf(outfile,' #include "%s"\n',strcat(name,'.h'));
fprintf(outfile,'\n');
fprintf(outfile,' short sine_buffer[]={\n');
fprintf(outfile,'                %i ,\n',values);
fprintf(outfile,'              };\n');
fprintf(outfile,'\n');
fprintf(outfile,' short sine_grad_buffer[]={\n');
fprintf(outfile,'                   %i ,\n',gradient);
fprintf(outfile,'                 };\n');
fprintf(outfile,'\n');


fclose(outfile);

outfile=fopen(strcat(name,'.h'),'w');

fprintf(outfile,'/*____________________________________________________________\
      \n');
fprintf(outfile,'  SINE FUNCTION TABLE \n');
fprintf(outfile,' generated with SineTabGenInt.m\n');
fprintf(outfile,'  \n');
fprintf(outfile,' file : %s\n',strcat(name,'.h'));
fprintf(outfile,' date : oct 2001 - jan 2002\n');
fprintf(outfile,' by :  c. haller\n');
fprintf(outfile,'       f. schnyder\n');
fprintf(outfile,' ____________________________________________________________\
      n');
fprintf(outfile,'*/\n');
fprintf(outfile,'\n');
fprintf(outfile,'#ifndef _%s_\n',upper(name));
fprintf(outfile,'#define _%s_\n',upper(name));
fprintf(outfile,'\n');
fprintf(outfile,' #define BUFFERSIZE_SINE %i\n',len);
fprintf(outfile,' #define STEP_INVERSE %i\n',step_inverse);
fprintf(outfile,' #define Q_FORMAT %i\n',q_format);
fprintf(outfile,' #define FORMAT %i\n',format);
fprintf(outfile,' #define REDUCTION %i\n',reduction);
fprintf(outfile,'\n');
fprintf(outfile,' extern short sine_buffer[];\n');
fprintf(outfile,'\n');
fprintf(outfile,' extern short sine_grad_buffer[];\n');
fprintf(outfile,'\n');
fprintf(outfile,'#endif /* _%s_ */\n',upper(name));
```

```
fprintf(outfile,'\n');

fclose(outfile);

plot(values./2^15)
```

Filter Coefficients

Listing F.52: *mixer_design.m*

```
%  Mixer Design - Version 0.5
%
%  file  :  mixer_design.m
%  date  :  oct 2001 - jan 2002
%  by    :  c. haller
%           f. schnyder
%_____________________________________________________________________

format long

fa=150e6/(8*293); % sampling rate
fp=6300;          % pass-band edge frequency
fs=19000;         % stop-band edge frequency
dp=2;             % pass-band ripple
ds=60;            % stop-band ripple
fc=10.7e6;        % carier frequency


% the sine and cosine oszilator
   sub=floor(fc/fa);
   foszi=fc-sub*fa;

   oszi_a1=-2*cos(2*pi*foszi*(1/fa));
   oszi_b1_cos=-cos(2*pi*foszi*(1/fa));
   oszi_b1_sin=sin(2*pi*foszi*(1/fa));
   % to Q15
   ioszi_a1=round(oszi_a1*2^15);
   ioszi_b1_cos=round(oszi_b1_cos*2^15);
   ioszi_b1_sin=round(oszi_b1_sin*2^15);

% the filter
   [n,Wn]=buttord(fp*2/fa, fs*2/fa, dp, ds); % determine the order

   [b,a]=butter(n,Wn);             % determine direct coefficients
   freqz(b,a,256,fa)               % and plot the filter

   [z,p,k]=butter(n,Wn);           % derermine second order coefficients
   [coeff,g]=zp2sos(z,p,k,'up','inf');

   % to Q15
   icoeff=coeff;
   Ns=size(coeff,1);
   for i=1:Ns
      icoeff(i,4)=coeff(i,5)/2;
      icoeff(i,5)=coeff(i,5)/2;
      icoeff(i,:)=round(icoeff(i,:)*2^15);
   end

% write data output

fprintf('/*--------------------------------------------------------------------\n');
fprintf(' coefficients of sine and cosine oszilator \n');
fprintf(' -------------------------------------------------------------------*/\n');
fprintf(' /* Frequency : %10.2f HZ*/ \n',foszi);
fprintf('\n');
fprintf(' short gen_b1_cos = %i;  /*cos*/ \n',ioszi_b1_cos);
fprintf(' short gen_b1_sin = %i;  /*sin*/ \n',ioszi_b1_sin);
fprintf(' short gen_a1 = %i;  /*sin/cos*/ \n',ioszi_a1);
fprintf('\n');
fprintf(' /* delay variabels */ \n');
fprintf(' short gen_z [3] = {0,0,26213}; /* 0.8 for first value of delta function*/ \
      n');
fprintf('\n');


```

```
fprintf('/*--------------------------------------------------------------------\n');
fprintf(' coefficients of mixer low-pass (mix_lp) \n');
fprintf(' -------------------------------------------------------------------*/\n');
fprintf(' /* Order  : %2i          */ \n',n);
fprintf(' /* Pass-band : %2i dB @ %6.2f Hz */ \n',dp,fp);
fprintf(' /* Stop-band : %2i dB @ %6.2f Hz */ \n',ds,fs);
fprintf('\n');
fprintf(' #define MIX_LP_CASCADES %i',Ns);
fprintf('\n');
fprintf(' short mix_lp_a[][3]={ /* a_i_1a a_i_1b a_i_2 */\n');
fprintf('                { %6i , %6i , %6i },\n',icoeff(:,4:6));
fprintf('              };\n');
fprintf('\n');
fprintf(' short mix_lp_b[][3]={ /* b_i_0 b_i_1 b_i_2 */\n');
fprintf('                { %6i , %6i , %6i },\n',icoeff(:,1:3));
fprintf('              };\n');
fprintf('\n');
fprintf(' short mix_lp_scale= %i; \t/* %f to avoid overflows */\n',(round(g*2^15)),g)
      ;
fprintf('\n');
fprintf(' /* delay variabels */ \n');
fprintf(' short mix_lp_z_re[][3]={ \n');
for i=1:Ns
   fprintf('                { 0,0,0},\n',icoeff(:,5:6));
end
fprintf('              };\n');
fprintf(' short mix_lp_z_im[][3]={ \n');
for i=1:Ns
   fprintf('                { 0,0,0},\n',icoeff(:,5:6));
end
fprintf('              };\n');
fprintf('\n');

% Filter Test for overflows
Nx=200;
y=zeros(Ns,Nx+2);


% create input signal
t=linspace(0,(1/fa*(Nx+1)),Nx+2);
%x=sin(2*pi*3000*t);
%x=chirp(t,100,t(Nx+2),10000);
%x=[0 0 (rand(1,Nx)-0.5)*2];
x=[0 1 ones(1,Nx/2) (ones(1,Nx/2)*-1) ];

xsave=x;

% scaling from the mixer
x=x*g;%*0.8;
d=zeros(Ns,Nx+2);

% do the filtering
for n=3:Nx+2
   for i=1:Ns
      d(i,n)=(x(n)-coeff(i,5)*d(i,n-1)-coeff(i,6)*d(i,n-2));
      y(i,n)=(coeff(i,1)*d(i,n)+coeff(i,2)*d(i,n-1)+coeff(i,3)*d(i,n-2));
      x(n) = y(i,n);
   end
end

%Plot delay states and output values

figure
for i=1:Ns
   subplot(2,Ns,i);
   plot(d(i,:),'k');
   axis([-10 Nx -1.4 1.4])
   legend(['Delay Cascade_',int2str(i-1)],4);

   subplot(2,Ns,i+Ns);
   plot(y(i,:),'k');
   axis([-10 Nx -1.4 1.4])
   legend(['Output Cascade_',int2str(i-1)],4);
end

%plot(y(Ns,:));
hold on
plot(xsave,'k:');
legend('Output Signal' , 'Input Signal',4);
hold off
```

Listing F.53: *lowpass_design_noisefilter.m*

```
%  Low-Pass Filter Design - Version 0.5
%
%  filter : low-pass before dac
%  file   :  lowpass_design_noisefilter.m
%  date   :  oct 2001 - jan 2002
%  by     :  c. haller
%            f. schnyder
%_____________________________________________________________________

fa=150000000/(8*293*4); % sample rate
fp=3500;            % pass-band edge frequency
fs=fa/4;            % stop-band edge frequency
dp=2;               % pass-band ripple
ds=20;              % stop-band ripple

[n,Wn]=buttord(fp*2/fa, fs*2/fa, dp, ds); % determine the order

[b,a]=butter(n,Wn);             % determine direct coefficients
freqz(b,a,256,fa)

[z,p,k]=butter(n,Wn);           % derermine second order coefficients
[coeff,g]=zp2sos(z,p,k,'up','inf');

coeffsave=coeff;
icoeff=coeff;

Ns=size(coeff,1);

for i=1:Ns
   icoeff(i,:)=round(icoeff(i,:)*2^15);
end

fprintf('/*-------------------------------------------------------------------\n');
fprintf(' coefficients of noise low-pass (no_lp) after demodulation \n');
fprintf(' -------------------------------------------------------------------*/\n');
fprintf(' /* Order  : %2i          */ \n',n);
fprintf(' /* Pass-band : %2i dB @ %6.2f Hz */ \n',dp,fp);
fprintf(' /* Stop-band : %2i dB @ %6.2f Hz */ \n',ds,fs);
fprintf('\n');
fprintf(' #define NO_LP_CASCADES %i',Ns);
fprintf('\n');
fprintf(' short no_lp_a[][2]={ /* a_i_1 a_i_2 */\n');
fprintf('               { %6i , %6i },\n',icoeff(:,5:6)');
fprintf('             };\n');
fprintf('\n');
fprintf(' short no_lp_b[][3]={ /* b_i_0 b_i_1 b_i_2 */\n');
fprintf('               { %6i , %6i , %6i },\n',icoeff(:,1:3)');
fprintf('             };\n');
fprintf('\n');
fprintf(' short no_lp_scale= %i; \t/* %f*0.8 to avoid overflows */\n',(round((g*0.8)
      *2^15)),g);
fprintf('\n');
fprintf(' /* delay variabels */ \n');
fprintf(' short no_lp_z[][3]={ \n');
for i=1:Ns
   fprintf('               { 0,0,0},\n',icoeff(:,5:6)');
end
fprintf('             };\n');

% Filter Test
Nx=300;
y=zeros(Ns,Nx+2);

%x=[0 0 (rand(1,Nx)-0.5)*2];

t=linspace(0,(1/fa*(Nx+1)),Nx+2);
%x=sin(2*pi*3000*t);
%x=chirp(t,100,t(Nx+2),10000);

x=[0 1 ones(1,Nx)];

xsave=x;

x=x*g*0.8;

d=zeros(Ns,Nx+2);

for n=3:Nx+2
   for i=1:Ns
      d(i,n)=(x(n)-coeff(i,5)*d(i,n-1)-coeff(i,6)*d(i,n-2));
      y(i,n)=(coeff(i,1)*d(i,n)+coeff(i,2)*d(i,n-1)+coeff(i,3)*d(i,n-2));
      x(n) = y(i,n);
   end
end

figure
for i=1:Ns
   subplot(2,Ns,i);
   plot(d(i,:));
   legend(['D',int2str(i)]);

   subplot(2,Ns,i+Ns);
   plot(y(i,:));
   legend('Y');
end

figure
plot(y(Ns,:));
hold on
plot(xsave,'r');
hold off
```

Listing F.54: *highpass_design_noisefilter.m*

```
%  High-Pass Filter Design - Version 0.5
%
%  filter : high-pass before dac
%  file   :  highpass_design_noisefilter.m
%  date   :  oct 2001 - jan 2002
%  by     :  c. haller
%            f. schnyder
%_____________________________________________________________________

fa=150000000/(8*293*4*2);; % sample rate
fp=300;                    % pass-band edge frequency
fs=50;                     % stop-band edge frequency
dp=0.1;                    % pass-band ripple
ds=40;                     % stop-band ripple

[n,Wn]=buttord(fp*2/fa, fs*2/fa, dp, ds); % determine the order

[b,a]=butter(n,Wn,'high'); % determine direct coefficients

freqz(b,a,256,fa)

[z,p,k]=butter(n,Wn,'high'); % derermine second order coefficients
[sos,g]=zp2sos(z,p,k);
nc=size(sos,1);
A=sos(:,4:6);
B=sos(:,1:3);
iA=A;
iB=B;

for i=1:nc
   iA(i,1)=iA(i,2)/2;
   iA(i,2)=iA(i,2)/2;
   iA(i,:)=round(iA(i,:)*2^15);
   iB(i,:)=round(iB(i,:)*2^15);
end

fprintf('/*-------------------------------------------------------------------\n');
fprintf(' coefficients of end high-pass (en_hp) before dac \n');
fprintf(' -------------------------------------------------------------------*/\n');
fprintf(' /* Order  : %2i          */ \n',n);
fprintf(' /* Pass-band : %2.1f dB @ %6.2f Hz */ \n',dp,fp);
fprintf(' /* Stop-band : %2.1f dB @ %6.2f Hz */ \n',ds,fs);
fprintf('\n');
fprintf(' #define EN_HP_CASCADES %i',nc);
fprintf('\n');
fprintf(' short en_hp_a[][2]={ /* aa_i_1a a_i_1b a_i_2 */\n');
fprintf('               { %6i , %6i , %6i },\n',iA');
fprintf('             };\n');
fprintf('\n');
fprintf(' /*\n');
fprintf(' short en_hp_b[][3]={ /* b_i_0 b_i_1 b_i_2 */\n');
fprintf('               { %6i , %6i , %6i },\n',B');
fprintf('             };\n');
fprintf(' */\n');
```

```
fprintf('\n');
fprintf(' short en_hp_scale= %i; \t/* 0.5*%f to avoid overflows */\n',(round(g
       *2^15/2)),g);
fprintf('\n');
fprintf(' #define EN_HP_UPSCALE 2\n');
fprintf('\n');
fprintf(' /* delay variables */ \n');
fprintf(' short en_hp_z[][3]={ \n');
for i=1:nc
   fprintf('                { 0,0,0},\n');
end
fprintf('             };\n');
```

```
nx=100
t=linspace(0,(1/fa*(nx-1)),nx);
x=sin(2*pi*300*t(1,1:50));
x=[x,cos(2*pi*300*t(1,51:nx))];
%x=[0 1 ones(1,nx-2)];


xsave=x;
x=x*g;

zeta=zeros(nc,nx);
kappa=zeros(nc,nx);
y=zeros(nc,nx);

for i=1:length(t)
   for c=1:nc
      y(c,i)=zeta(c,i)+(B(c,1)*x(i));
      zeta(c,i+1)=B(c,2)*x(i)-A(c,2)*y(c,i)+kappa(c,i);
      kappa(c,i+1)=B(c,3)*x(i)-A(c,3)*y(c,i);
      x(i)=y(c,i);
   end
end

alpha=zeros(nc,nx);
beta=zeros(nc,nx);
gamma=zeros(nc,nx);
y2=zeros(nc,nx);

x=xsave;

for i=1:length(t)
   for c=1:nc
      alpha(c,i)=(-A(c,3)*gamma(c,i))+(-A(c,2)*beta(c,i))+x(i);
      y2(c,i)=(B(c,1)*alpha(c,i))+(B(c,2)*beta(c,i))+(B(c,3)*gamma(c,i));
      gamma(c,i+1)=beta(c,i);
      beta(c,i+1)=alpha(c,i);
      x(i)=y(c,i);
   end
end

labels=int2str([1:nc]);

plot(t,xsave,t,y);
legend('in','out1','out2');
figure;
subplot(2,1,1)
plot(t,zeta(:,1:nx))
legend('zeta1','zeta2');
subplot(2,1,2)
plot(t,kappa(:,1:nx))
legend('kappa1','kappa2');

figure
plot(t,xsave,t,y);
legend('in n','out1','out2');
figure;
subplot(3,1,1)
plot(t,alpha(:,1:nx))
legend('alpha1','alpha2');
subplot(3,1,2)
plot(t,beta(:,1:nx))
legend('beta1','beta2');
subplot(3,1,3)
plot(t,gamma(:,1:nx))
legend('gamma1','gamma2');
```