University of Twente

department of
Electrical Engineering

Demonstration of the Software-Radio Concept

R. Schiphorst

M.Sc. Thesis

Report no: EL-S&S-005.00

Date: 14-06-2000

January – June 2000

Under supervision of: Prof. dr. ir. C.H. Slump
Dr. ir. S.H. Gerez
Ing. G.J. Laanstra

University of Twente, Department of Electrical Engineering, Signals & Systems PO Box 217, 7500 AE Enschede, The Netherlands

Abstract

Since the early 1980's an explosion-like increase of cellular mobile systems can be observed. A side effect of this rapid growth is an excess of mobile system standards. In fact, every major country has its own standard(s). Therefore, the s*oftware-radio* concept is emerging as a potential pragmatic solution: a software implementation of the user terminal able to dynamically adapt to the radio environment in which it is located.

First, this report presents a global overview of the *software-radio* concept. Furthermore it explains a software-radio transmitter and receiver built for demonstration purposes. The source code has been written as much as possible in C and is running on an evaluation module of Spectrum Digital (with the TMS320LC549 DSP from Texas Instruments). The software-radio transmitter and receiver are based on a modified version of the China Wireless Telecommunication Standard (CWTS). Because the used evaluation module has been designed for audio applications the software radio uses frequencies that are about 5000 times lower than the CWTS standard uses. Both the receiver and transmitter require about 75 million instructions for decoding/encoding one frame. Because the source code is not optimized, the total number of instructions can be reduced significantly. Finally the system has been verified by measuring the BER versus the bit-energy-to-noise-energy ratio.

Acknowledgements

This Master-of-Science assignment*, Demonstration of the Software-Radio Concept*, has been performed at the Signals & Systems chair of the University of Twente.

I would like to thank the following persons. First, my daily supervisors and graduation committee, Sabih Gerez, Kees Slump and Geert-Jan Laanstra for their ideas and help of during the assignment. Furthermore I would like to thank Eelke Blok for his comments on this report.

Of course, I would like to thank everyone else who helped me or showed interest in my assignment.

Enschede, June 2000

Roel Schiphorst

Table of contents

CHAPTER 1

Introduction

Nowadays the magic word in consumer electronics is *mobile*. In the Netherlands almost seven million people [1] have a mobile telephone and their number is increasing every year with tens of percents. The current second-generation mobile network (such as GSM) is however developed for speech; it is not suited for multi-media applications. Therefore, a new third-generation (3G) mobile network is in development, which can handle these new speech and data services. However, this new generation does not imply that the old one will be replaced, both systems will exist. Furthermore, other countries have other standards, which use other modulation techniques and other frequency bands. So there are tens of standards worldwide and their number is increasing. Efforts to reduce the number of standards by defining an integrating standard, results often in a new extra standard [2].

For worldwide roaming it would be nice to have an intelligent handheld able to dynamically adapt to the radio environment in which it is located. The handheld should independently determine with which modulation technique, frequency band and protocol, transmission could take place in a certain place and time. A possible solution for this problem is the *software radio*: a system in which adaptation is achieved by executing the program appropriate to the situation.

1.1 Assignment

The chair Signal & Systems has set up a project [35] in collaboration with the telecommunication industry for investigating the software-radio paradigm. The final goal of the project is to build a software-radio prototype. When I started with my Master's project the software-radio project was not started yet. Therefore my first assignment was to make a study of literature about software radio.

The second assignment was to build a part of the software radio on a re-configurable platform such as a *digital signal processor* (DSP). This part, the front end, starts in the transmitter with the modulation of binary data. Then, the modulated signal is transformed to a channel frequency in the system band. In the next step, the signal is transmitted and received by the receiver. The receiver selects digitally the used channel and demodulates the data. Together these parts form the front end of the software radio transceiver. The emphasis of this task is the projection of the required algorithms on the resources and to design, to implement and to test the required functionality. This should result in a demonstration model of the front end of the software radio.

1.2 Overview

Chapter 2, *Software radio*, gives a global overview of the software-radio subject. It will deal with the definition, the architecture, technological challenges and commercial aspects of software radio. Also two second-generation standards will be discussed. Finally this chapter will discuss the front-end design of software radio in more detail.

Chapter 3, *Hardware*, deals with the hardware that is used in this project. Two evaluation modules of Spectrum Digital (with a Texas-Instruments DSP) are used. The design of these evaluation modules and the used programming tool, Code Composer, will be discussed in this chapter.

Chapter 4, *Software*, discusses the design of the software, which is running on the two evaluation modules. First, it will deal with the used mobile system standard, the *China wireless telecommunication standard* (CWTS). Finally the practical implementation of this standard will be discussed. Also the design of the transmitter and receiver are discussed.

Chapter 5, *Experiments*, discusses some experiments with the software radio transmitter/receiver. First the working of the transmitter and receiver is verified with the programming tool, Code Composer. Finally the BER versus bit-energy-to-noise-energy is measured and verified.

Chapter 6, *Final Review*, draws conclusions about this project. Furthermore it gives recommendations for further research in the field of software radio.

CHAPTER 2

Software radio

2.1 Introduction

Since the early 1980's an explosion-like increase of cellular mobile systems can be observed. Nowadays, mobile communication has become a major worldwide business. A side effect of this rapid growth is an excess of analog and digital mobile system standards such as TACS, GSM, DCS-1800, IS-95 CDMA etc. In fact, every major country has its own standard(s). Efforts to define a unique worldwide standard result often in a new, extra standard [2]. The excess of standards is not only bad for manufacturers but also for consumers. Manufacturers have to develop a new telephone for each standard. This results in extra development costs and small divided markets. It is also bad for consumers because they cannot use their mobile telephones abroad.

A unique common worldwide standard has benefits, but the industrial competition between Asians, Europeans and Americans makes it very difficult. It is therefore in this field that the *software-radio* concept is emerging as a potential pragmatic solution: a software implementation of the user terminal able to dynamically adapt to the radio environment in which it located [3]. Aside of the standardization issues, one should also view the software radio concept as a means to make users, service providers, and manufacturers more independent of standards as such. The benefits of this approach are that air interfaces may, in principle, especially be tailored to the specific needs of a particular service for a particular user in a given environment at a given time. For a manufacturer, a single design is sufficient for the whole world and consumers can use their telephones in every country.

Software Radio can also be described as *radio functionalities defined by software* [4], meaning the possibility to define by software, the typical functionalities of a radio interface. Currently the radio interface in mobile telephones is usually implemented by dedicated hardware. The presence of software defining the radio interface implies the use DSPs replacing dedicated hardware to execute, in real time, the necessary software. Although digital signal processing has developed exponentially since the 1980's, the processing power of DSPs and programmable logic such as *field programmable gate arrays* (FPGAs) is today still too small for a complete digital programmable transceiver. The required processing power (several thousands of MIPS [5]) is expected to become available in the near future.

The structure of this chapter is as follows. First the definition of the software radio will be dealt. Then, two second-generation mobile system standards (GSM *(Global System for Mobile Computing*) and the American *code division multiple access* (CDMA) system (IS-95)) will be discussed. Similarities and differences between standards affect the architecture of the software radio. This issue is discussed after the description of these two standards. There are several problems, which have to be overcome before the software radio can be manufactured. These problems are discussed in the final section.

2.2 Definition

There is not one unique definition for the software radio concept. The most common definitions are summed up below and quoted from [3]:

- *Flexible transceiver architecture, controlled and programmable by software*
- *Signal processing able to replace, as much as possible, radio functionalities*
- *air-interface-download ability: dynamically re-configurable radio equipment by downloadable software, at every level of the protocol stack*
- *Software realization of terminals multiple mode/standard*
- *Transceiver where the following parts can be defined by software:*
  - o *frequency band & radio channel bandwidth*
  - o *modulation & coding scheme*
  - o *radio resource and mobility management protocols*
  - o *user applications*

  These parameters can be adapted and changed by:
  1. *network operator*
  2. *service provider*
  3. *final user*

Software radios use digital techniques, but software-controlled digital radios are generally not software radios [5]. The difference between software-controlled digital radios and software radios is the total programmability of software radio. This programmability includes programmable *radio-frequency* (RF) bands, channel access modes, and channel modulation.

Therefore, in summary, the following definition is proposed (which is quoted from [3]):

> *Software radio is an emerging technology, aimed to build flexible radio systems, which are multiple-service, multi-standard, multi-band, re-configurable and re-programmable, by software.*

A software-radio system can operate in *multi-service* environments. This means that the software radio is able to offer services of any already standardized systems or future ones, on any radio frequency band and is not constrained to a particular standard. For that reason the software-radio system is very flexible. The compatibility of a software-radio system with any defined mobile-system standard is guaranteed by its re-configurability, which is supplied by DSP processors. These processors implement in real time radio interface and upper-layer protocols.

A software radio does not just transmit only, to quote Mitola [5]:

> *In an advanced application, a software radio does not just transmit: it characterizes the available transmission channels, probes, the propagation path, constructs an appropriate channel modulation, electronically steers its transmit beam in the right direction, selects the appropriate power level, and then transmits. Again, in an advanced*

application, a software radio does not just receive: it characterizes the energy distribution in the channel and in adjacent channels, recognizes the mode of the incoming transmission, adaptively nulls interferers, estimates the dynamic properties of desire-signal multi-path, coherently combines desired-signal multi-path, adaptively equalizes this ensemble, trellis decodes the channel modulation, and then corrects residual errors via forward error control (FEC) decoding to receive the signal with lowest possible bit-error rate (BER).

Finally, the software radio supports incremental service enhancements through a wide range of software tools. These tools assist in analyzing the radio environment, defining the required enhancements, prototyping incremental enhancements via software, testing the enhancements in the radio environment, and finally delivering the service enhancements via software and/or hardware.

2.3 Second-generation standards

Ideally, the software radio should be able to carry out every mobile system standard in the world. A good understanding of those standards is thus required. Therefore this section will discuss two major second-generation standards. The first one is GSM, developed in Europe and the second one is IS-95, developed in the USA. An overview of all major second-generation standards and their technical details is given in Table 1.

The structure of this section is as follows. First a functional architecture of a mobile telephone receiver is discussed. After this description, the differences and similarities between GSM and IS-95 are discussed for every segment in this functional architecture.

System	TACS	GSM	DCS-1800	Qualcomm IS-95 CDMA	IS-54 DAMPS	JDC	CT2	DECT	PHS	PACS
Origin	UK	Europe	Europe	USA	USA	Japan	UK	Europe	Japan	USA
Forward Band (MHz) (base station → mobile station = down link)	935-950	935-960	1805-1880	869-894	869-894	810-826 1477-1489 1501-1513	864-868	1880-1900	1985-1918	1930-1990
Reverse Band (MHz) (mobile station → base station = up link)	890-905	890-915	1710-1785	824-849	824-849	940-956 1429-1441 1453-1465	(TDD)	(TDD)	(TDD)	1850-1910
Multiple Access	FDMA	TDMA	TDMA	CDMA	TDMA	TDMA	FDMA	TDMA	TDMA	TDMA
Duplex	FDD	FDD	FDD	FDD	FDD	FDD	TDD	TDD	TDD	FDD
Carrier Spacing	25	200	200	1250	30	25	100	1728	300	300
Channels/carrier (kHz)	1/pair	8/pair	8/pair	55-62	3	3	1	12	4	8/pair
Modulation	FM	GMSK	GMSK	QPSK	π/4-DQPSK	π/4-DQPSK	FSK	GMSK	π/4-DQPSK	π/4-QPSK
Modulation Rate (kBd)	N/A	271	271	1228	48.6	42	72	1152	192	192
Voice + FEC Rate (kbps)	N/A	22.8	22.8	8/Var.	11.2	13	32	32	32	32
Speech codec	N/A	RPE-LTP	RPE-LTP	CELP	VSELP	VSELP	ADPCM	ADPCM	ADPCM	ADPCM
Unprotected Voice Rate (kbps)	N/A	13	13	1.2-9.6	7.95	6.7	32	32	32	32
Control Chan. Name		SACCH	SACCH	SACCH	SACCH	SACCH	D	C	SACCH	
Control Chan. Rate (bps)		967	967	800	600		2000	640		4000
Control Message Size (bits)		184	184	1	65		64	64		10/2.5 ms
Control Delay (ms)		480	480	1.25	240		32	10		
Peak Power (Mobile) (W)	0.6-10	2-20	0.25-2	0.6-3	0.6-3	0.3-3	10 mW	250 mW	80 mW	200 mW
Mean Power (Mobile) (W)	0.6-10	0.25-2.5	0.03-0.25	0.2-1	0.6-3	0.1-1	5 mW	10 mW	10 mW	
Power Control	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Opt.	
Voice Activity Detection	Yes	Yes	Yes	Yes	Opt.	Opt.	No	No		
Handover	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Dynamic Channel Allocation	No	No	No	N/A	No	Opt.	Yes	Yes	Yes	autonom.
Min. Cluster Size	7	3	3	1	7	4	N/A	N/A	N/A	N/A
Capacity (Dpx ch/cell/MHz)	2.8	6.7	6.7	16.5*	7		N/A	N/A	N/A	N/A
Frame duration (ms)		4.615	4.615	20	40	40	2	10	5	2.5
Speech FEC	N/A	Conv. (2,1,5)	Conv. (2,1,5)	Conv. Fwd: (2,1,9) Rev: R = 1/3	Conv. (2,1,5)	R = 9/17	No	No	CRC	CRC
Channel Eq.	N/A	Yes	Yes	Yes	Opt.	Opt.	No	No	No	No
Half-rate codec (kbps)	N/A	5.6	5.6	No	No	3.2	No	No	No	No
Half-rate + FEC (kbps)	N/A	11.4	11.4			5.6				
Enhanced Full-rate (kbps)	N/A	12.2	12.2	Yes	7.4	No	No	No	No	No

DCS-1800: GSM-like European system in the 1800 MHz band	IS-54: American digital advanced mobile phone system (DAMPS)
IS-95: American CDMA system	CT2: British cordless telephone system
PHS: Japanese personal handy phone system	PACS: personal access communications system
TACS: total-access communications system	GSM: global system for mobile computing
CDMA: code-division multiple access	DECT: digital European cordless telephone
FDMA: frequency-division multiple access	FDD: frequency-division duplex
TDMA: time-division multiple access	TDD: time-division duplex
FM: frequency modulation	GMSK: Gaussian minimum shift keying
DQPSK: differential quadrature phase shift keying	GFSK: Gaussian phase shift keying
RPE-LTP: regular-pulse-exited – long-term-predicted	CELP: code-excited linear predicted
VSELP: vector-sum-excited linear predictive	ADPCM: adaptive differential pulse-code modulation
SACCH: slow-associated control channel	JDC: Japanese Digital Cellular

Table 1: Overview of second-generation standards [6]

2.3.1 Functional architecture of a transceiver

Although every standard is different, all standards have the same global steps in decoding a radio signal into speech or data and vice versa. Figure 1 shows a global schematic of a transceiver. The first step of a transmitter is to convert the speech signal into a bit stream. This bit stream must be tiny, because a smaller bit stream leads to less channel bandwidth, which increases the capacity of the network. Therefore, GSM and IS-95 use complex algorithms for encoding speech. The output of the speech encoder is between 9 – 15 kbps (*kilo bits per second*) [Table 1]. Then, error correction code is added to this bit stream. Although a speech encoder reduces the bit stream significantly, the importance of every bit is increased. The system becomes more sensitive for bit errors. To overcome this, error correction code is added. Thus, the real objective of the speech encoder is to minimize this bit stream, to have a small error-insensitive bit stream.

The next step is to cipher the bit stream for privacy reasons. After ciphering, the bit stream will be modulated. This modulated signal uses the bandwidth of one channel. Standards, which use CDMA as modulation technique, multiply this modulated signal with a spreading code ([4], [6], [7] and [8]). So, every channel uses a (small) part of the total bandwidth of the mobile network. The next step is to convert the *base band* (BB) signal into the appropriate part of the system bandwidth. This step marks the border between BB processing and *intermediate-frequency* (IF) processing. The IF stage has a bandwidth of several tens of MHz. Finally, this band has to be converted into the real *radio frequency* (RF) band, which is used by the system (for GSM this band is 890 – 960 MHz).

The transmitted signal is picked up by the receiver, which converts the RF signal into speech or data. The first step is to isolate the bandwidth, which is used by the system. In this step the signal is converted from RF band into IF band. The next block, IF processing, selects one channel from the system band and converts this channel into BB. Then the BB signal will be demodulated. First, it will de-spread the bit stream (if a CDMA standard is used) followed by an equalization process. The last steps of the signal processing are the symbol-to-bit conversion and *forward error correction* (FEC) decoding. The final step is to convert the received bit stream into a speech signal.

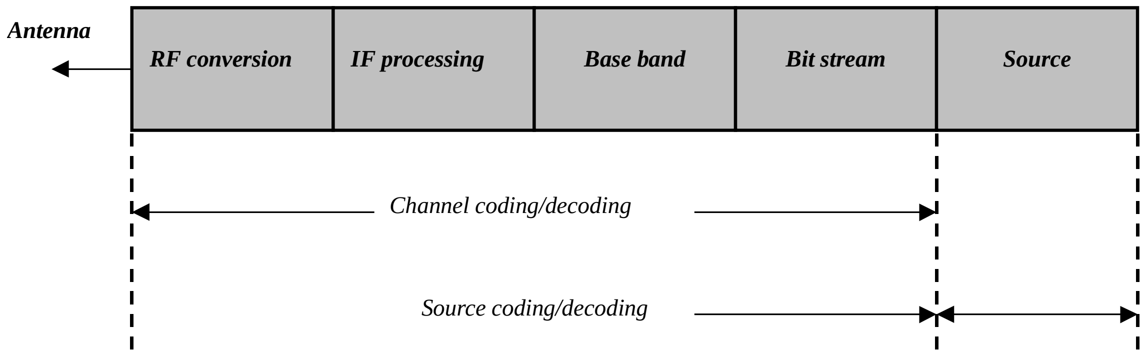


Figure 1: functional architecture of a mobile system transceiver

2.3.2 GSM and IS-95

This section describes how each segment of the transceiver has been implemented by GSM and IS-95. Both standards use for example a different speech *encoder/decoder* (codec). GSM uses the *regular-pulse-exited – long-term-predicted* codec (RPE-LTP) and IS-95 uses the *code-exited-linear-prediction* (CELP) codec. Both methods are complex; for a more detailed explanation, the reader is referred to [9] and [10]. In short, both methods have a model of the speech system. First the optimal parameters of this model are determined for each frame (of 20 ms). The output after this model, the residual signal (which is minimized by the choosing the optimal parameters), can be approximated by different methods. The CELP codec has a codebook, which contains the most used residual fragments. The index of the best match will be transmitted. For optimal performance the codebook is adapted at every frame. The RPE-LTP codec uses another, more difficult method for approximating the residual signal. The residual signal does not vary quickly in time. The codec uses this property and tries to match the signal with a sequence from the past. The best match is subtracted from the signal and the outcome can be considered as noise. This noise is then roughly encoded and transmitted.

After the speech codec, the signal can be considered as a stream of data packets. (The speech encoder encodes the speech per frame (of 20 – 30 ms)). To reduce errors a FEC code is added. Both methods use a convolutional code as FEC. The output of a convolutional code does not only depend on the inputs but also on the state of the encoder. An example of such an encoder is given in Figure 2. From the state diagram, it can be seen that the next state depends on the current input and the current state. There are several algorithms, which can be used to decode this bit stream, for example the *Viterbi*-algorithm ([7], [11]).

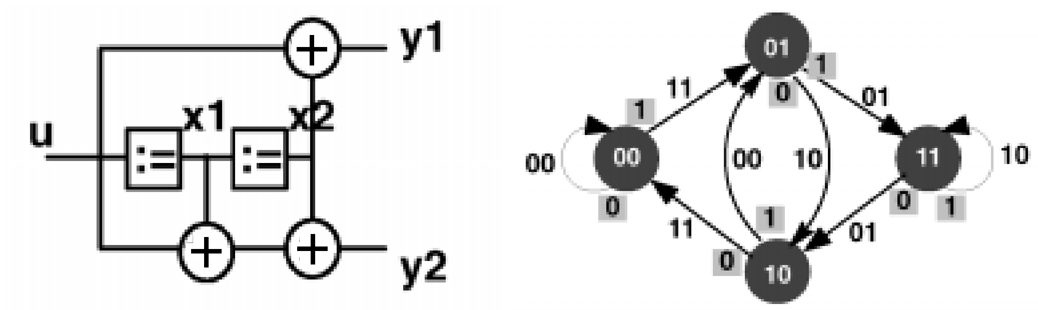


Figure 2: a simple convolutional encoder and its state diagram [11]

After the convolutional encoder, the data packet is led to an interleaver. Errors, in mobile communication do not occur uniformly in time, they occur in short bursts. The best performance of FEC codes is achieved when bit errors occur uniformly [7]. Therefore, the bits of the data packet are mixed, interleaved, for optimal performance. GSM and IS-95 use the same methods, although the implementation differs. Mobile communication must also be safe; therefore the bit stream will be ciphered before it will be modulated. Both standards use ciphering algorithms, which change the encryption from call to call. Therefore, it is nearly impossible to tap a channel.

After ciphering, the bit stream can be modulated and transmitted. Both standards use very different methods. Therefore they will be discussed separately, starting with GSM. GSM uses the G*aussian-minimum-shift-keying* (GMSK) - modulation technique. This is a spectrally efficient variant of the *frequency-shift-keying* (FSK) -modulation technique. For a binary "1", frequency f_1 is transmitted and for a "0", frequency f_2. The GMSK variant, however, uses a minimal frequency distance (by meeting several conditions) and the output is led through a Gaussian filter, to reduce bandwidth. More information can be found in [7] and [19].

GSM uses a combination of FDMA (*frequency division multiple access*) and TDMA (*time-division multiple access*) for communication between mobiles and base stations. First, the frequency band of GSM is divided into channels (FDMA). At most eight users can use each channel by using TDMA. Thus, every user has his own channel and time slot. So the GMSK-signal has to be converted into the right frequency band and must be transmitted in the right time slot.

As said before, IS-95 uses a very different modulation technique because it uses another method, CDMA for communication between mobiles and base stations. First, the bit stream is modulated using the *quadrature-phase-shift-keying* (QPSK) - modulation technique. This technique modulates two bits a time; the phase of a QPSK-signal has namely four values: $\pi/4$, $3\pi/4$, $5\pi/4$, and $7\pi/4$. So, for every bit combination, there is another phase. The bit stream is then multiplied with a fast random binary signal (called the *pseudo-noise* (PN) code). More information can be found in [7] and [19].

So, every user has another PN code, which is orthogonal to other PN codes. This means that the output of multiplication of two PN codes is zero. Because the bit stream is spread by the PN code, it uses a relative large frequency band. Other users can nevertheless share this frequency band, because each PN code is orthogonal to other PN codes. The capacity of CDMA is therefore in theory unlimited, but PN codes are in reality not purely orthogonal. Thus, every extra user increases the noise floor. The first step in recovering the original bit stream is to de-spread the received signal. This is achieved by multiplying it with the same PN code used by the transmitter. The next and final step is to demodulate the QPSK signal into a bit stream. More information about IS-95 and GSM can be found in [6], [7], [8], [10] and [12].

It should be noted that the standards mentioned in this chapter (GSM and IS-95) are not discussed completely. Only the link between handset and base station is discussed. Besides this connection, there are, for example, service channels. The transmitted power is regulated and there are other aspects of a mobile system standard on a higher level such as the billing system. These aspects are not discussed here because they are irrelevant for the handset to base station link. More information can be found in the literature cited above.

2.4 Architecture

In the previous section, the functional diagram of radio transceivers is described. This section discusses the architecture of those transceivers. Currently transmitters and receivers are based on the traditional super-heterodyne scheme (Figure 3). The RF and IF stages are completely analog. Only the BB stage is digital, usually built in dedicated hardware. In Figure 3 the signal is picked up by the antenna. The next step is to filter the signal with a *band-pass filter* (BPF) and to amplify it with a *low-noise amplifier* (LNA). The resulting system band is converted to a lower frequency band by multiplying it with a *local oscillator* (LO). A *low-pass filter* (LPF) isolates the down-converted system band. Then the *analog gain control* (AGC) block tries to normalize the signal power for an optimal use of the *analog digital converter* (ADC). The next step is to isolate one channel from the system band. First the signal is multiplied with a *voltage-controlled oscillator* (VCO). The Digital-Base-Band block controls this VCO. A *digital analog converter* (DAC) is used to convert the digital control signal of the Digital-Base-Band block to an analog signal. This analog signal controls the VCO. After the signal is multiplied with the VCO, the signal is filtered with a LPF and finally sampled (ADC). Because some mobile system standards use quadrature modulation techniques, both the *in-phase* (I) and *quadrature*-phase (Q) component are extracted and sampled. These two bit streams are sent to the digital base band processing. This block also controls the channel selection. More information can be found in section 2.7, Front-end design.

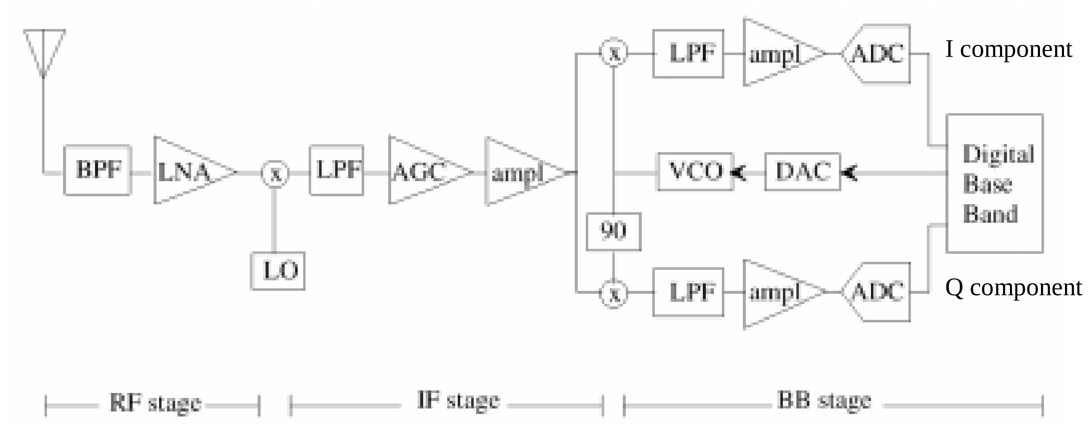


Figure 3: traditional heterodyne receiver [3]

In contrast, the ideal software-radio receiver is shown in Figure 4. The analog stage is as small as possible. The analog stage consists only of the antenna, the BPF and the LNA. The A/D conversion (ADC) is done immediately after the LNA, in order maximize the re-programmability of the system.

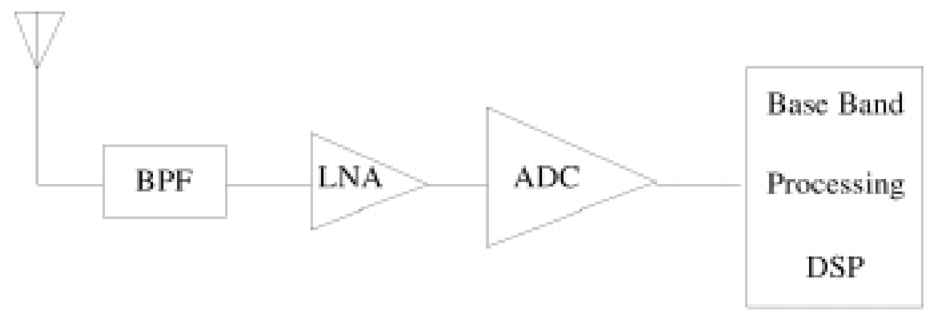


Figure 4: the ideal software-radio receiver [3]

At this moment, the ideal software radio is not realizable. There are several matters, which cause this [13]. For example it is impossible to build antenna and LNAs on a working bandwidth ranging from hundreds MHz to units or tens of GHz. The only way to guarantee the *multi-band* feature is to have more RF stages. Also, jitter effects limit the possibility of A/D conversion directly at the RF band. The most promising solution for the moment is known as *Digital Radio receiver* [3], shown in Figure 5. The RF stage is still completely analog, but the A/D converter samples the spectrum allocated to the system immediately after the RF stage. The IF stage of the Digital Radio transceiver consists of the *programmable down converter* (PDC) which provides the following operations [3]:

- *down conversion*: digital conversion from IF to BB, by using a look-up table containing the samples of a sinusoidal carrier. The look-up table replaces the local oscillator used in the analog down converter.
- *channelization*: selection of the carrier and channel which is performed by digital filtering. In analog receivers, analog filters with very stringent requirements are used.
- *sample-rate adaptation*: under-sampling of the channelization-filter-signal output, to match the sample rate to the selected channel bandwidth. The bandwidth of a channel is compared to the spectrum of the A/D input signal a narrow-band signal. Therefore the sample rate can be much lower to enlighten the required processing power.

After these three operations, the digital PDC output is base-band processed. An example of a PDC can be found in [14].

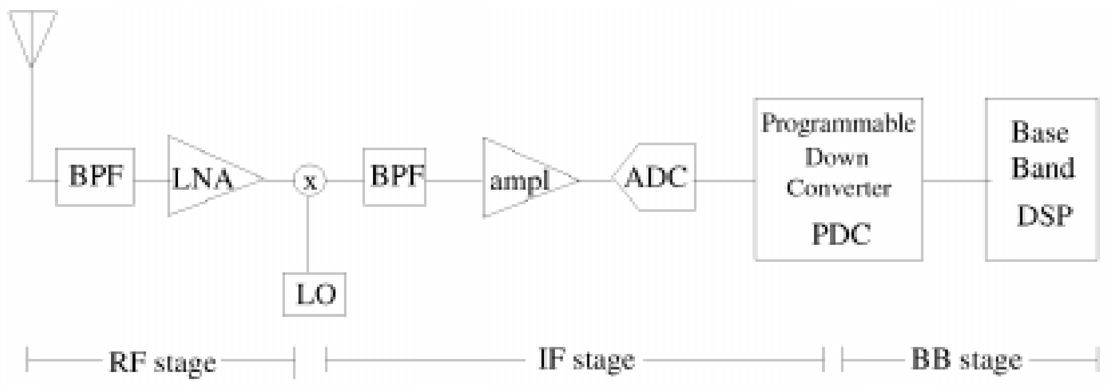


Figure 5: Digital Radio receiver [3]

2.5 Technological challenges

This section discusses the technological issues, which have to be solved before the software radio can be manufactured (commercially). The development of a software radio system, implies, above all, the achievement of two main targets [3]:

1. To move, in transceivers, the border between analog and digital world, as much as possible, towards RF. This requires A/D & D/A wide-band converters placed as near as possible to the antenna.
2. To replace dedicated hardware (ASICs (*application specific integrated circuits*)) with DSPs or FPGAs. In other words, to define, as much as possible, radio functionalities in software.

The first target, indeed, is not software radio exclusive. Much research has been carried out in the w*ide-band transceiver* realization [3]. The primary goal of this transceiver was to extend the digital domain at the IF stage and keeping the RF stage analog.

The second goal is also not only applicable to software radio. A global trend in the industry is to replace hardware by software, because software is flexible. Besides DSPs there are other solutions to make the transceiver flexible, for example FPGAs [2]. There are advantages of using FPGAs instead of DSPs for signal processing in commercial telecommunication systems. The power consumption is lower, the size is smaller and the costs are much lower in comparison with DSPs. A disadvantage of FPGAs is the lack of good tools for efficient mapping of the algorithms. For a commercial product this mapping should be very efficient. This does not only apply for FPGAs but also for DSPs. Only for DSPs more tools are available.

Besides these two important issues there are other challenges, which have to be solved [15]. The first challenge is the power management. For example, sleep modes of DSPs or other hardware save power but introduce a probability that the radio will be asleep during a paging message. A possible solution is a structured timing of paging messages, which reduces the miss probability, and further conserves battery life.

The second challenge is the clock generation and distribution. Every standard such as GSM or IS-95 has its own clock rate. Using one reference oscillator per standard increases parts count, complexity, and therefore cost. A single master clock may use the *least common multiple* (LCM) of the required clocks, but this leads to a very high clock rate, which is very power inefficient. A possible solution is to normalize standards to avoid clock rates with large LCMs.

Receiver complexity is typically four or more times the transmitter complexity. Thus, the receiver architecture has a first order impact on handset cost. The challenge is thus to develop a simple receiver. With the current technology, the support of many standards leads to complex and power-inefficient solutions. ASICs are very power efficient but inflexible. FPGAs could be a possible solution.

The final challenge is the handset production. The cost of handsets in volume production is a nearly linear function of parts count. A possible solution to this problem is a new emerging technology, called MEMS (*micro electro-mechanical systems*) [36]. MEMS is a technology that combines computers with tiny mechanical devices such as sensors, valves, gears, mirrors, and actuators embedded in semiconductor chips. Finally the handset software has to be extraordinarily efficient in use of computational resources. Therefore the handset software has to be very efficient. Every extra line in the source code leads to extra costs. For more information about technical challenges the reader is referred to [15]. That article describes also the technical challenges in the infrastructure of the mobile network and base stations.

2.6 Commercial aspects

So far, only technical details on the implementation of the software radio have been discussed, but there are other aspects, for example the standardization of the software radio. Should every manufacturer develop its own software radio standard, making it incompatible with other systems, or should there be one standard for the software radio. In [16], an overview is given on this area. The article describes which standards can be developed for the software radio: standardized interface to applications (API (*application program interface*)), standardized means for delivering new software to a software radio, standardized ways for allowing a software radio to access a system, standardized software specification language. Furthermore, the question arises how *intellectual property* (IP) can be protected, allowing manufacturers to develop their own features. Too many standards may limit the competition between manufacturers and the development of new hardware. In addition, too many standards will prevent the software radio to become a success.

2.7 Front-end design

2.7.1 Several receiver designs

A global introduction to the software radio was given in the previous section. This section describes in more detail the front end of the software-radio receiver, the RF/IF section.

The conventional heterodyne-transceiver architecture requires some analog filters fitted to the carrier frequency and channel bandwidth of each of the communication standards. The disadvantage of this architecture is that it uses fixed narrow-band passive components that don't fit in a broadband system with multi-mode operation. Therefore a general-purpose common RF stage is required for standards with different RF specifications. These specifications include carrier frequency, bandwidth, modulation scheme, and transmission power [17].

The ability of the software-radio architecture to support a communication waveform is predominantly determined by [18]:

- the largest instantaneous signal bandwidth (W)
- the frequency range and bandwidth of the RF;
- the ADC sampling rate (greater than 2W)
- the maximum dynamic range
- DSP throughput requirements including translation of IF to base band, modulation, demodulation, coding, and decoding.

In most systems the conventional heterodyne receiver is used as shown in Figure 6. It works as follows. First a band-pass filter isolates the system band (Figure 7a). This band is then moved towards a lower frequency band (Figure 7b) by multiplying it with a tuned LO. The system band is then converted from RF to IF. In the next stage the signal is multiplied with a second (programmable) local oscillator (NCO). With this oscillator a channel can be selected. The selected channel will be moved to base-band frequencies (Figure 7c). Finally a low-pass filter will isolate the channel (Figure 7d). Then the signal is digitized where it can be digitally processed.

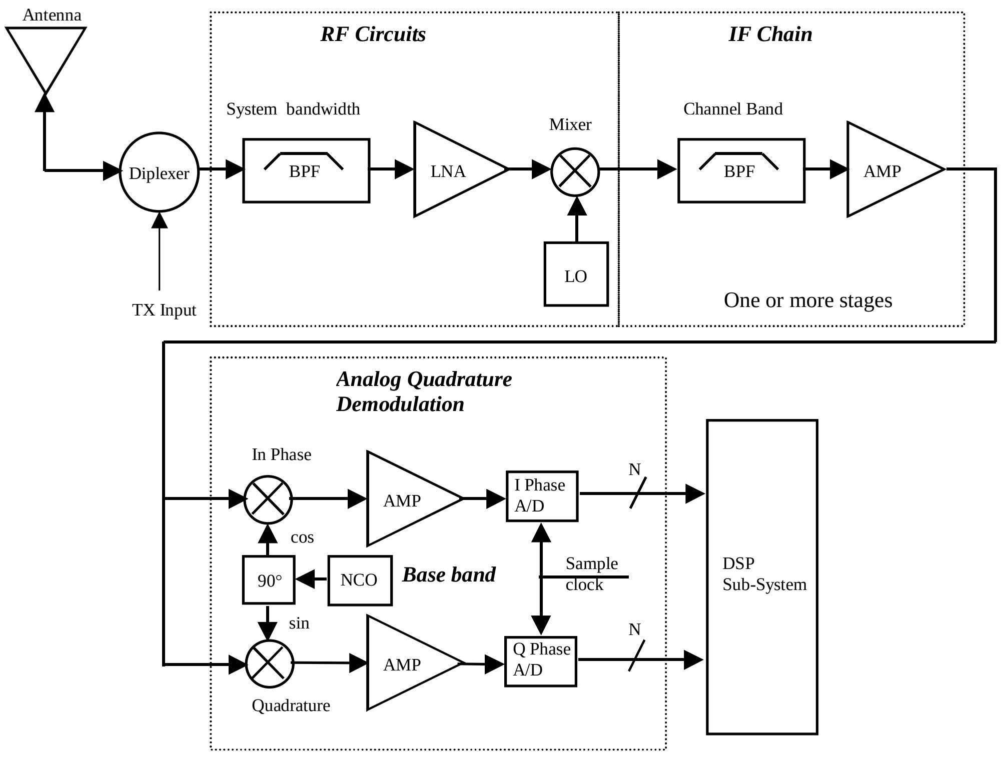


Figure 6: conventional heterodyne receiver [18]

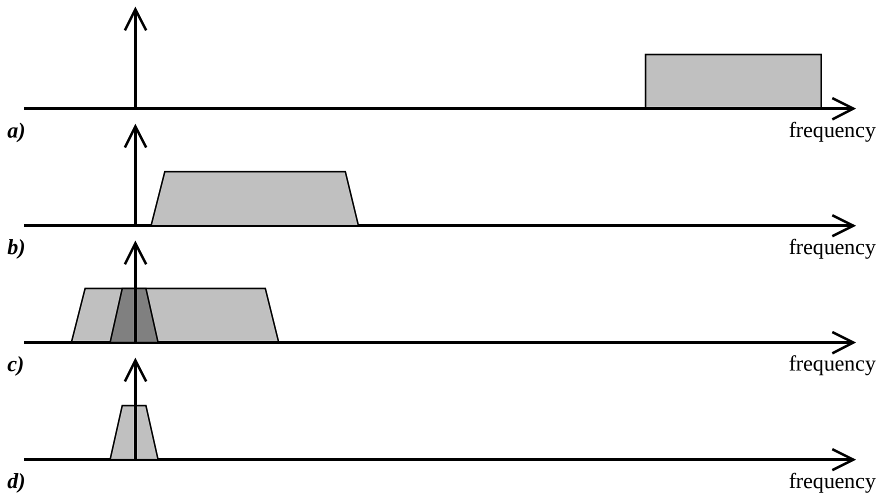


Figure 7: global steps of a heterodyne receiver, a) normal radio spectrum, b) filtering and movement of system band to intermediate frequencies, c) selection of channel and d) filtering and digitalization of channel band width.

A second configuration is the pass-band super-heterodyne receiver that translates the RF signal in one or more IF stages to a final pass-band IF frequency where it is digitized, as shown in Figure 8. The sampling rate of the pass-band signal must be at least twice the bandwidth of the system. Pass-band sampling is popular because only one ADC converter is required, simplifying the component configuration. An example of the pass-band super-heterodyne receiver is given in [3].

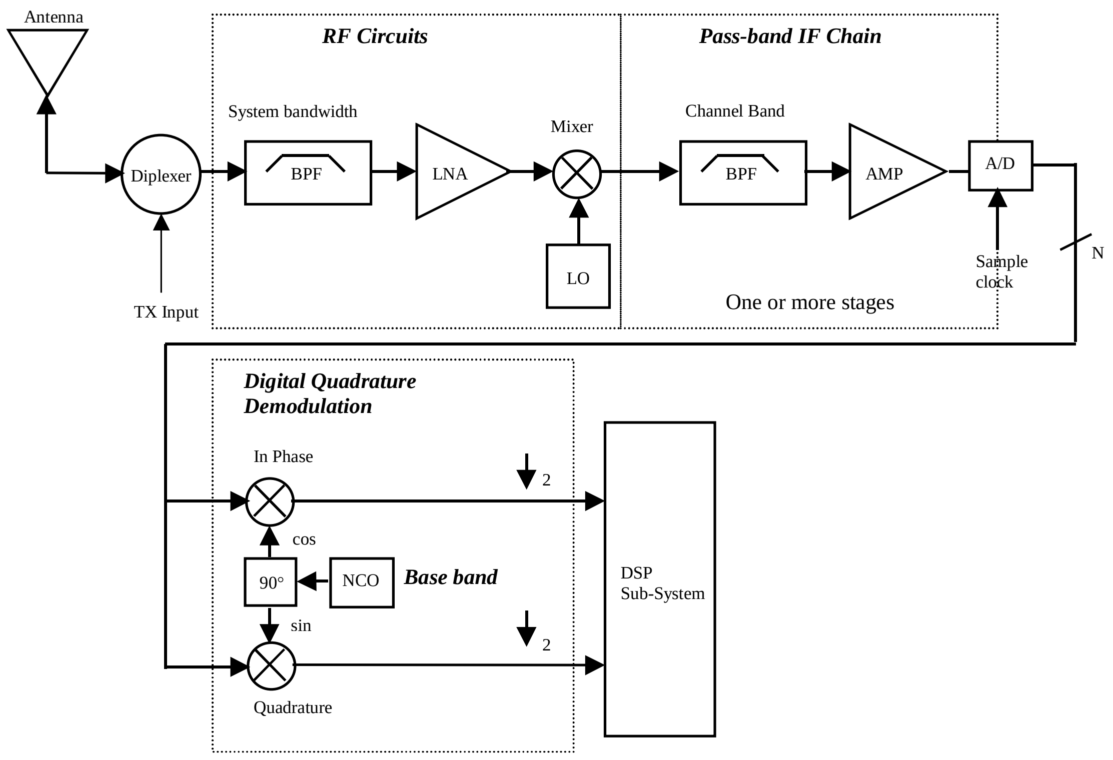


Figure 8: pass-band super-heterodyne receiver [18]

Finally, Figure 9 shows a direct-conversion (or homodyne or zero-IF) receiver where the RF signal is translated directly to base band. Therefore a NCO or synthesizer must be locked to the carrier. There are several advantages of the direct-conversion receiver: fewer signal translation steps and the ability to use simpler receiver filters cascaded with low-pass base-band digital filters in the DSP. The result is a more flexible (wider) tuning range and potentially larger channel bandwidths. But there are also a number of disadvantages of the direct-conversion receiver. These disadvantages are leakage from high-gain low-noise mixers, requirements for very-high-dynamic-range analog components, the requirement for higher sensitivity than a comparable super-heterodyne receiver, the need for precise *in* (I) and *quadrature* (Q) phase balancing, DC-offset cancellation, antenna isolation, and high-selectivity filters. As a result, homodyne receivers are extremely difficult to implement [18].

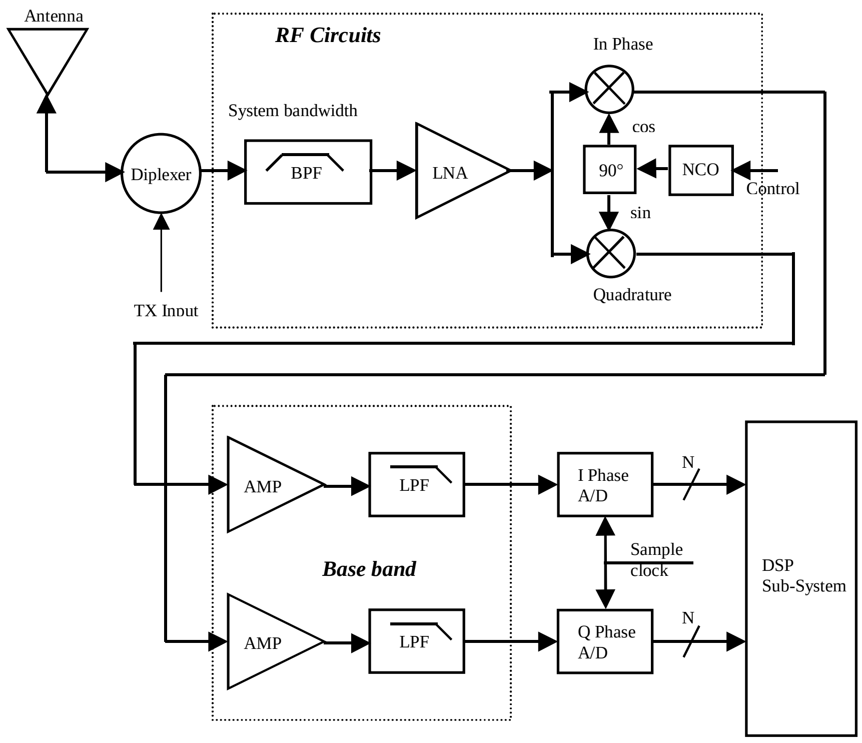


Figure 9: direct-conversion receiver [18]

2.7.2 Receiver challenges

In a receiver several effects appear which have to be compensated. These effects appear both in the analog and digital stage. These items are [17]:

- Analog stage (LNA, mixer, LPF)
    - Signal saturation (i.e. gain control)
    - Nonlinear distortion
    - DC offset
- Base band or digital stage
    - I/Q gain mismatch
    - I/Q phase mismatch
    - DC offset

Base-band-stage compensation is generally achieved by DSP algorithms using received-signal sequences stored in memory [17]. On the other hand, RF analog stage performance depends on circuit linearity. The laws of physics define this linearity. A device-level approach is not recommended because it results in excessive demands on the circuit in order to make it meet every standard. The results are increased terminal cost and size. Therefore, system-level compensation techniques are desirable.

The first problem is signal saturation. The receiver gain must be set to feed the received signal linearly to the ADC over approximately an 80 - 90 dB input range for practical applications. A too low or too high receiver gain leads to an increased BER. Therefore the gain should be controlled to achieve an optimal BER.

The second problem is the non-linear distortion. In mobile communication systems, the radio terminal must receive the desired signal even when an undesired signal approximately 60 dB stronger than the desired signal is present [17]. Any second-order non-linearity produces an undesired distortion signal in the base band. This non-linear distortion is produced mainly in the mixer. Therefore a highly linear mixer is required which reduces second-order distortion. The realization of such a mixer is, however, difficult. This distortion signal must always be lower than the desired signal output (e.g., approximately 15 dB for QPSK modulation [17]).

Finally the DC offset is a problem. This DC offset is mainly caused by *self-mixing* in the mixer circuit. This offset can be divided into two components: time-invariant offset inherent to the mixer circuit and the timer-variant offset caused by local-leakage reflections. The latter is mainly caused by variations in the environment. For a QPSK differential detection the DC offset must be approximately 30 dB lower [17] than the desired signal at the ADC input. A possible solution for removing the DC offset is a DC-offset-canceller circuit. If a small signal is received, only a digital-stage canceller is sufficient. However, if a large desired signal is received, the DC offset will saturate the ADC. Then an analog-stage canceller is also needed.

CHAPTER 3

Hardware

3.1 Introduction

Software runs on hardware. This chapter describes the hardware that is used in this project. Two DSP-evaluation modules from Spectrum Digital are chosen as re-configurable platform. The laboratory has namely two of these evaluation modules. So, one can be used as receiver whereas the other can be used as software-radio transmitter. The next section describes the DSP-evaluation module. Also the DSP and the AD/DA chip from Texas Instruments are discussed in this section. The last section discusses the programming tool, Code Composer that is used for programming the software-radio concept into the DSP.

3.2 DSP Evaluation Module

The TMS320LC549 Evaluation Module is a stand-alone card that lets evaluators examine certain characteristics of the TMS320LC549 DSP to determine if this DSP meets their application requirements. Other important parts on the evaluation module, besides the DSP are the AD/DA-chip and the external memory. The next sections discuss those parts in more detail. Figure 10 shows how these parts are connected on the evaluation module (Note: The AD/DA-chip is the TLC320AD55 chip).

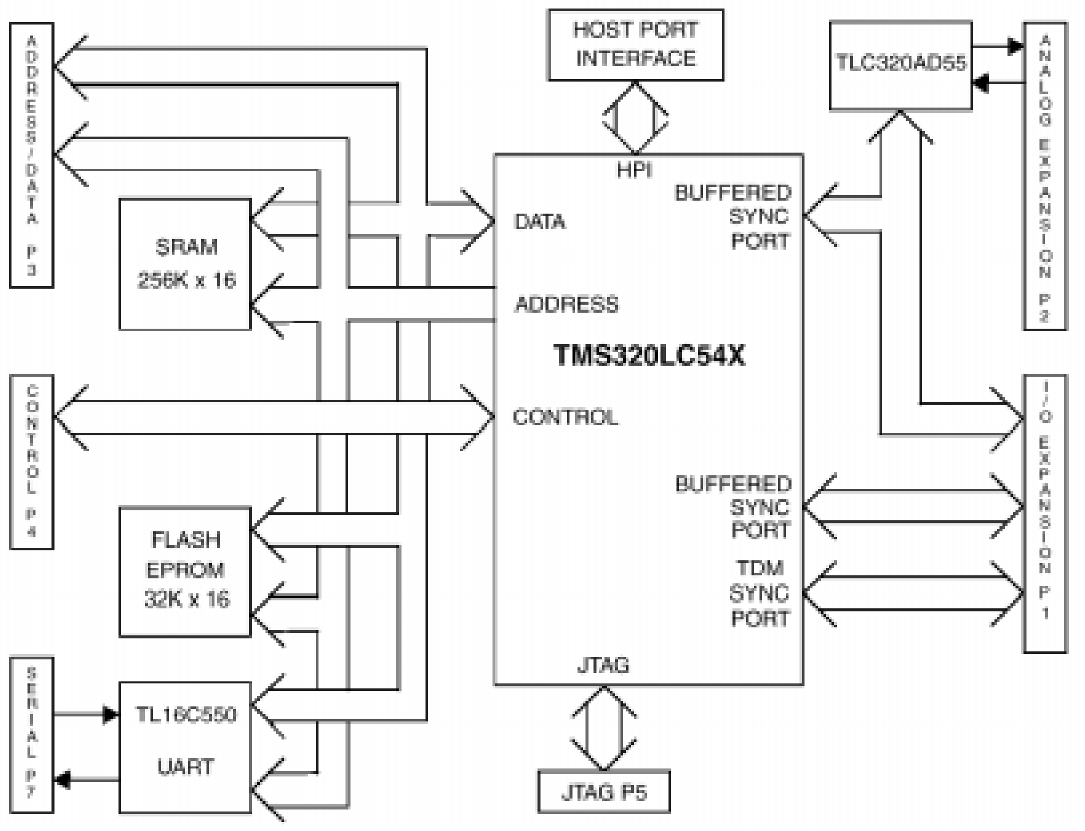


Figure 10: the TMS320C549 Evaluation Module [20]

3.2.1 The TMS320LC549 DSP chip

The homepage of the TMS320LC549 DSP [21] gives a good overview of the capabilities of the processor. A summary of this homepage is given below:

> *The TMS320LC549 fixed-point DSP is based on a modified Harvard architecture that has one program-memory bus and three data-memory buses. The processor also provides an arithmetic-logic unit (ALU) that has a high degree of parallelism, application-specific hardware logic, on-chip memory, and additional on-chip peripherals. Separate program and data spaces allow simultaneous access to program instructions and data, providing a high degree of parallelism. Two read operations and one write operation can be performed in a single cycle. Instructions with parallel store and application-specific instructions can fully utilize this architecture. In addition, data can be transferred between data and program spaces. Such parallelism supports a powerful set of arithmetic, logic, and bit-manipulation operations that can all be performed in a single machine cycle. In addition, the TMS320LC549 includes the control mechanisms to manage interrupts, repeated operations and function calls.*

The TMS320LC549 DSP consists of the following parts [22]:

- 40-bit ALU
- Two 40-bit accumulators
- Barrel shifter
- 17x17-bit multiplier
- 40-bit adder
- Compare, select, and store unit (CSSU)
- Data address generation unit

- Program address generation unit

These parts are connected internally as shown in Figure 11 and the most important features of TMS320LC549 DSP are shown in Table 2.

Parameter Name	**TMS320LC549-100**
Frequency (MHz)	100
MIPS	100
Cycle Time (ns)	10
Data / Program Memory (Words)	64K/8M
RAM (Words)	32K
ROM (Words)	16K
Timers	1
Total Serial Ports	3
TDM Serial Ports[1]	1
Buffered Serial Ports	2
COM[2]	HPI[3]
Boot Loader Available	YES
Core Supply (Volts)	3.3
IO Supply (Volts)	3.3

Table 2: TMS320LC549 specifications [21]

[1] A *time-division multiplexed* (TDM) serial port is a synchronous serial port that is enhanced to allow time-division multiplexing of the data.
[2] COM = port of the TMS320C549 DSP used to interface a host processor or device.
[3] HPI = *host port interface*, more information can be found in [32].

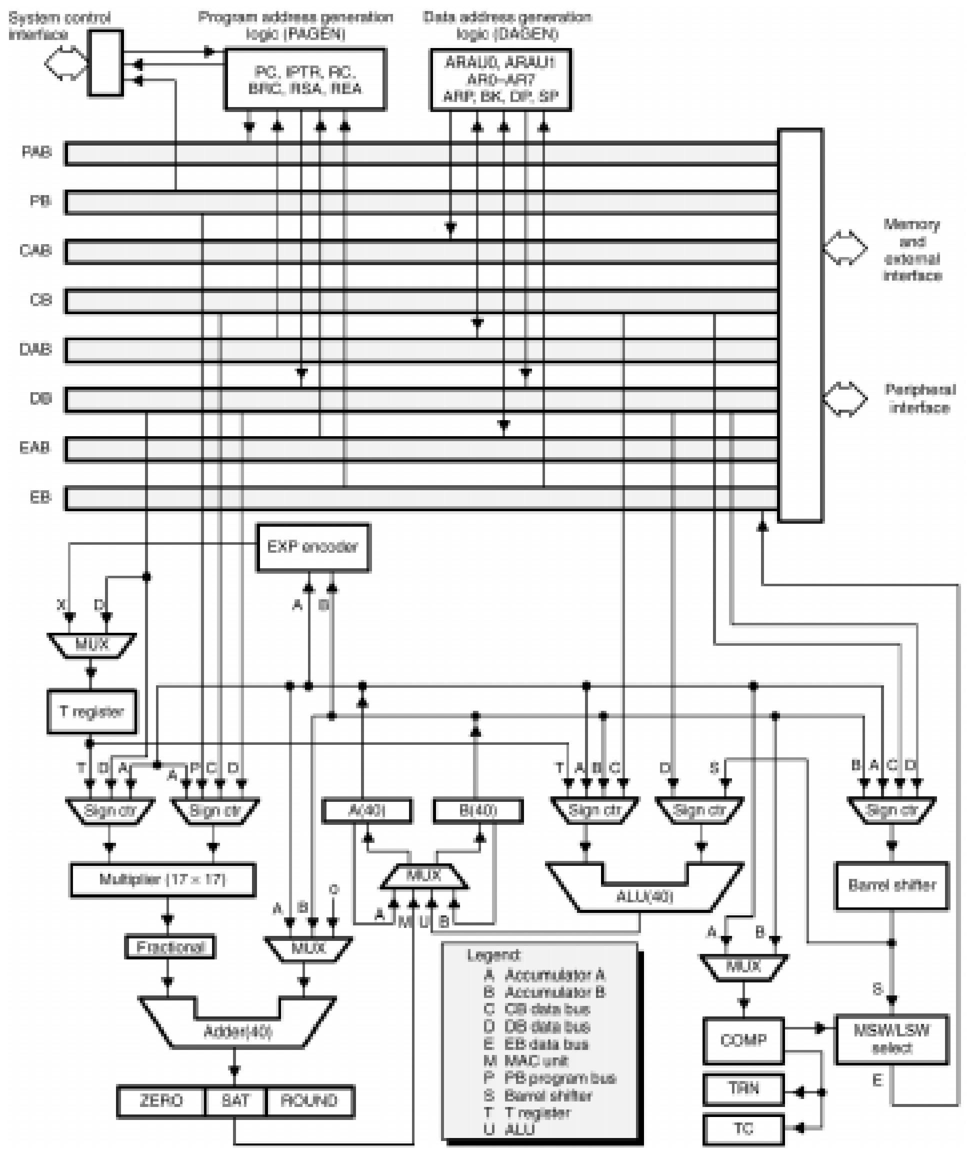


Figure 11: block diagram of TMS320C549 internal hardware [22]

3.2.2 The TMS320AD55 AD/DA chip

The capabilities of the TMS320AD55 AD/DA chip are described in its datasheet [23]. Below a summary is given from this datasheet.

> *The TLC320AD55 provides high-resolution low-speed signal conversion from digital-to-analog (D/A) and from analog-to-digital (A/D) using over-sampling sigma-delta technology. This device consists of two, serial, synchronous conversion paths (one for each data direction) and includes an interpolation filter before the digital-to-analog converter (DAC) and a decimation filter after the analog-to-digital converter (ADC) (see Figure 12). Other overhead functions provide analog filtering and on-chip timing and control. The sigma-delta architecture produces high resolution, A/D and D/A conversion at low system speeds and low cost. The options and the circuit configurations of this device can be programmed through the serial interface. The options include reset, power-down, communications protocol, serial clock rate, signal sampling rate, and test mode. The circuit configurations could include a selection of input ports to the ADC, analog loop-back, digital loop-back, decimator-sinc-filter output, decimator Finite-*

duration-Impulse-Response (FIR) filter output, interpolator-sinc-filter output, and interpolator FIR filter output. The TLC320AD55 is characterized for operation from 0℃ to 70℃.

Features of the TLC320AD55 AD/DA-chip are:

- *Single 5-V power supply*
- *Power dissipation (P_D) of 150 mW maximum in the operating mode*
- *Power-down mode to 1 mW*
- *General-purpose 16-bit signal processing*
- *Two's-complement format*
- *Serial port interface*
- *Minimum 80-dB harmonic distortion plus noise*
- *Differential architecture*
- *Internal reference voltage (V_{ref})*
- *Internal 64-oversampling*
- *Analog output with programmable gain of 1, 1/2, 1/4, and 0 (squelch)*
- *Phone-mode output control*
- *Variable sampling (conversion) frequency (F_s). F_s = MCLK/($F_k \cdot 256$), F_k = 1,2,3,...,256 and MCLK = master clock*
- *System test mode:*
  - *Digital loop-back test*
  - *Analog loop-back test*

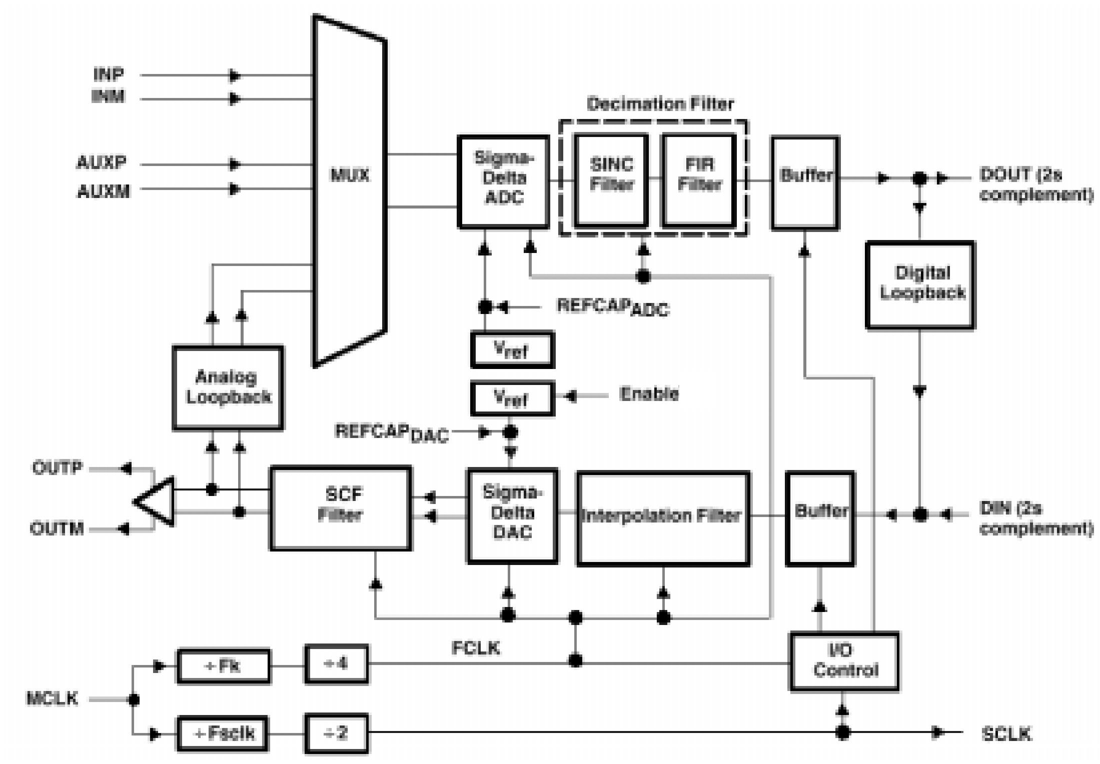


Figure 12: function block diagram of the TLC320AD55 IC [23]

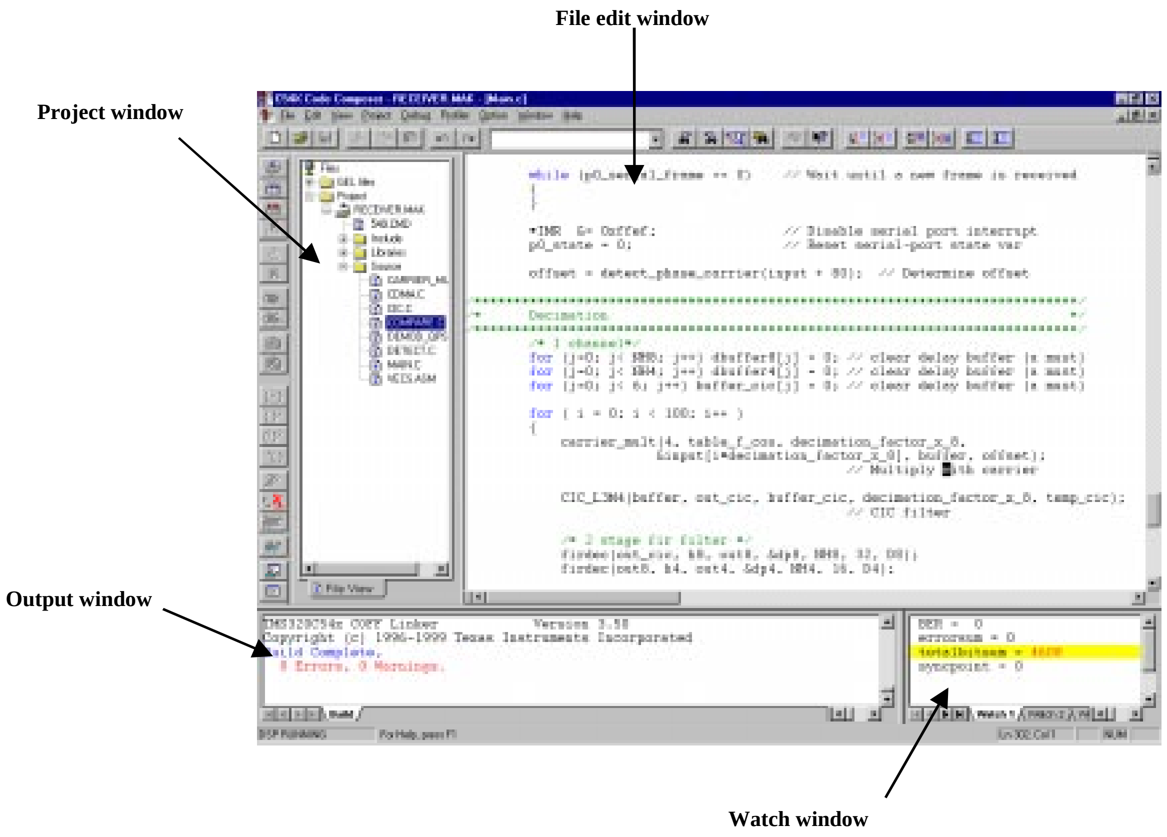
3.2.3 External memory

The evaluation module has 192k words of one wait-state program *read-access memory* (RAM) and 64k words of one wait-state data RAM memory. Besides the

256k of off-chip static RAM the board has also two 32k flash *read-only memories* (ROMs) for boot loading.

3.3 Code composer

Code Composer is a powerful tool for programming DSPs. With Code Composer source files can be edited and projects can be built, debugged, profiled and managed from a single unified environment. Other features include graphical signal analysis, injection/extraction of data signals via file I/O, multi-processor debugging, automated testing and customization via a scripting language [24]. With this language, the *general extension language* (GEL) the development environment can be customized and a *graphical user interface* (GUI) can be created to control the target DSP application. Figure 13 shows a typical Code Composer screen.



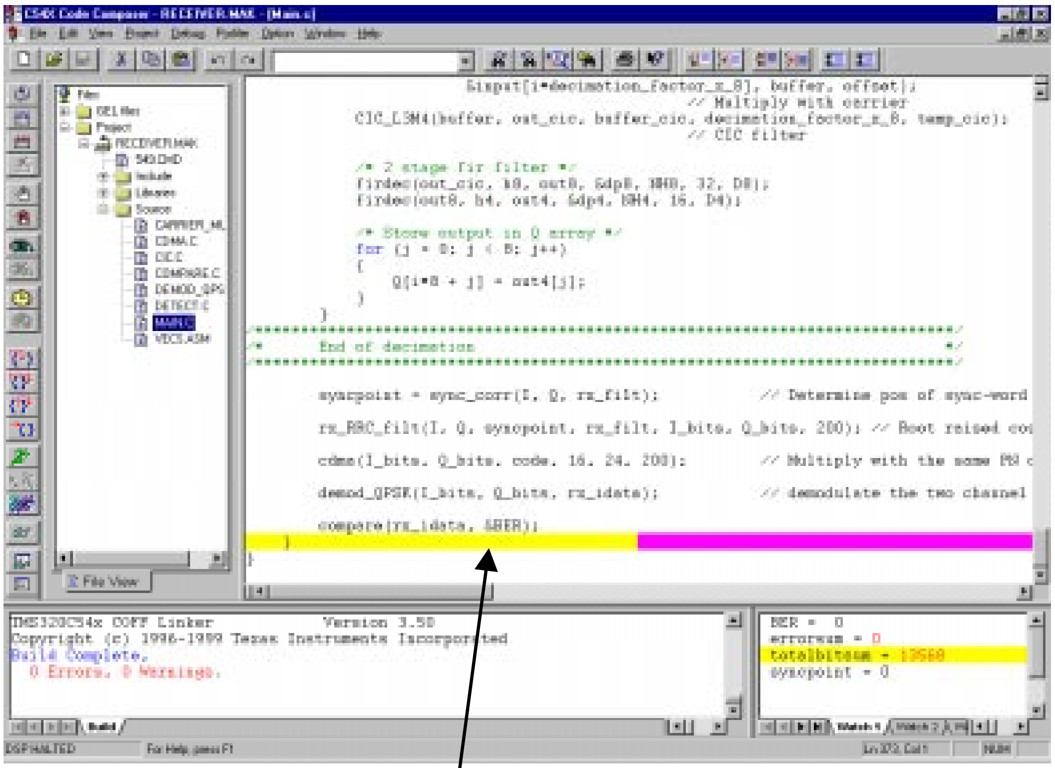
```
while (p0_serial_frame == 0)    // Wait until a new frame is received
{
}

*IMR  &= 0xffef;                 // Disable serial port interrupt
p0_state = 0;                    // Reset serial-port state var

offset = detect_phase_carrier(input + 80);  // Determine offset

/*****************************************************************/
/*     Decimation                                                */
/*****************************************************************/
/* 1 channel*/
for (j=0; j< NH8; j++) dbuffer8[j] = 0; // clear delay buffer (a must)
for (j=0; j< NH4; j++) dbuffer4[j] = 0; // clear delay buffer (a must)
for (j=0; j< 6; j++) buffer_cic[j] = 0; // clear delay buffer (a must)

for ( i = 0; i < 100; i++ )
{
    carrier_mult(4, table_f_cos, decimation_factor_x_8,
                 &input[i*decimation_factor_x_8], buffer, offset);
                                       // Multiply with carrier

    CIC_L3M4(buffer, out_cic, buffer_cic, decimation_factor_x_8, temp_cic);
                                       // CIC filter

    /* 2 stage fir filter */
    firdec(out_cic, h8, out8, &dp8, NH8, 32, D8);
    firdec(out8, h4, out4, &dp4, NH4, 16, D4);
```

```
TMS320C54x COFF Linker          Version 3.50
Copyright (c) 1996-1999 Texas Instruments Incorporated
Build Complete,
  0 Errors, 0 Warnings.
```

```
BER =  0
errorsum = 0
totalbitsum = 4608
syncpoint = 0
```

Figure 13: typical Code Composer screen

In Figure 13 the left-most window is the Project window. In this window files can be added or removed to the project. By double clicking on the file names in the Project window, files will be opened for editing (in the File-edit window). After compiling the output file must be loaded into the DSP. Possible compile errors are shown in the Output window. While the program is running variables can be viewed in the Watch window. Also break points can be set in the source code as shown in Figure 14. Not only can variables be viewed in the Watch window, but they can also be shown graphically in the Graph window, shown in Figure 15. Another feature of Code Composer is the use of Profile points. Profile points are similar to breakpoints, but instead of halting the target processor they count occurrences, and they collect statistics on the number of instruction cycles, or other events, that have elapsed since the previous profile point was encountered. In Figure 16 two Profile points are added. In the Statistics window one can see how many instructions have elapsed between the current profile point and the next one. Other data in the Statistics window are the minimal and maximal value. All these features make Code Composer a powerful tool for programming DSPs.



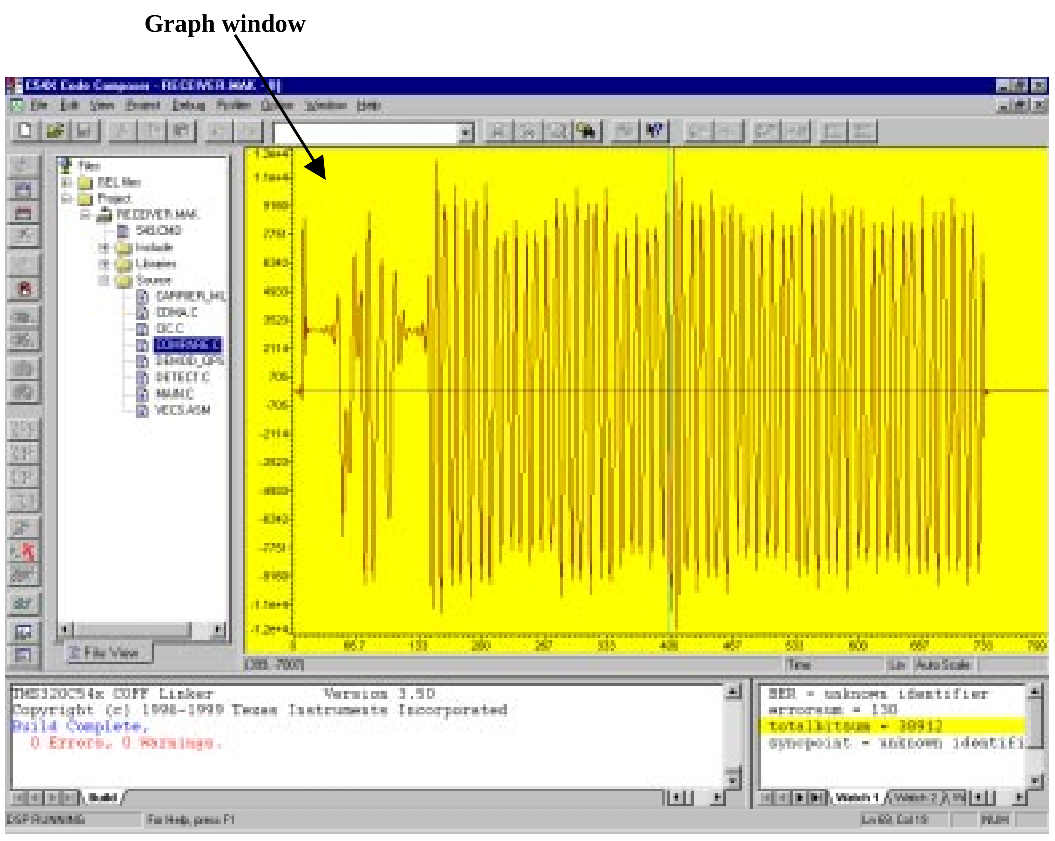
```
                    &input[i*decimation_factor_x_8], buffer, offset);
                                            // Multiply with carrier
        CIC_L3M4(buffer, out_cic, buffer_cic, decimation_factor_x_8, temp_cic);
                                            // CIC filter

        /* 2 stage fir filter */
        firdec(out_cic, h8, out8, &dp8, NH8, 32, D8);
        firdec(out8, h4, out4, &dp4, NH4, 16, D4);

        /* Store output in Q array */
        for (j = 0; j < 8; j++)
        {
            Q[i*8 + j] = out4[j];
        }
    }
/*****************************************************************************/
/*      End of decimation                                                    */
/*****************************************************************************/

    syncpoint = sync_corr(I, Q, rx_filt);           // Determine pos of sync-word

    rx_RRC_filt(I, Q, syncpoint, rx_filt, I_bits, Q_bits, 200); // Root raised co

    cdma(I_bits, Q_bits, code, 16, 24, 200);        // Multiply with the same PN c

    demod_QPSK(I_bits, Q_bits, rx_idata);           // demodulate the two channel

    compare(rx_idata, &BER);
  }
}
```

```
TMS320C54x COFF Linker          Version 3.50
Copyright (c) 1996-1999 Texas Instruments Incorporated
Build Complete,
  0 Errors, 0 Warnings.
```

```
BER =  0
errorsum = 0
totalbitsum = 13560
syncpoint = 0
```

Break point

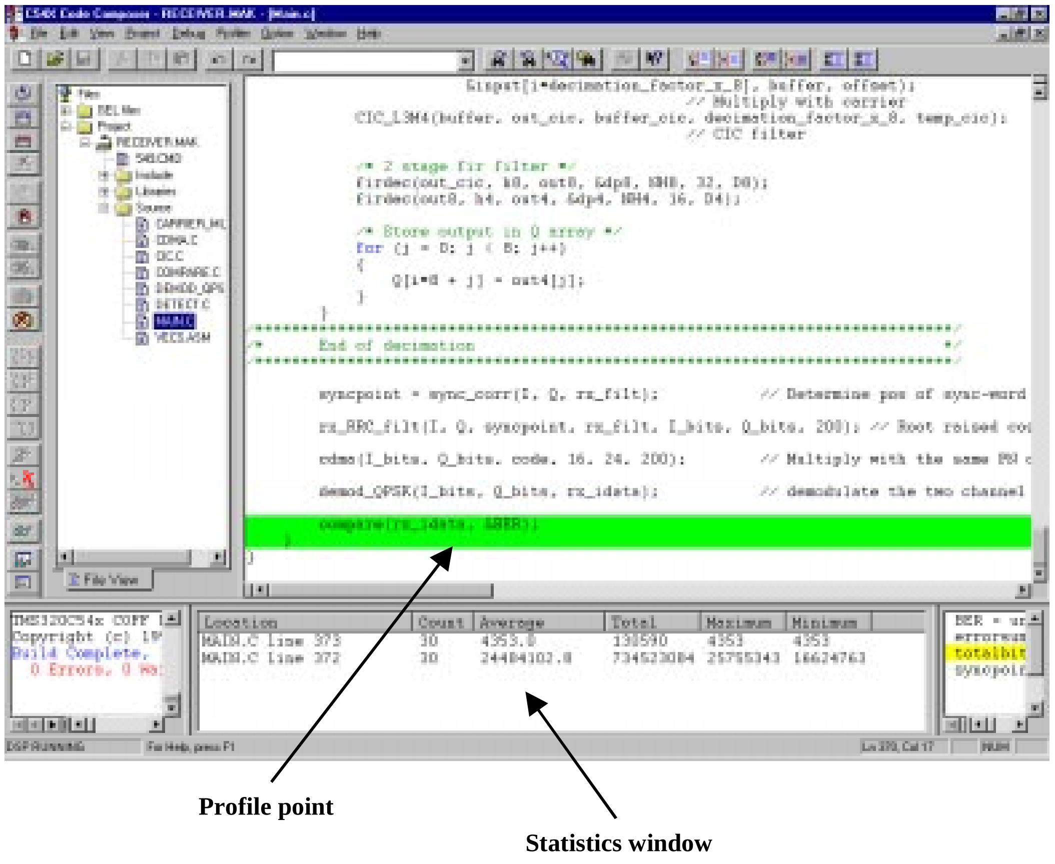
Figure 14: break points in Code Composer



```
TMS320C54x COFF Linker             Version 3.50
Copyright (c) 1996-1999 Texas Instruments Incorporated
Build Complete,
  0 Errors, 0 Warnings.
```

```
BER = unknown identifier
errorsum = 130
totalbitsum = 38912
syncpoint = unknown identifi
```

Figure 15: graph window in Code Composer



```
                        &input[i*decimation_factor_x_8], buffer, offset);
                                                 // Multiply with carrier
            CIC_L3M4(buffer, out_cic, buffer_cic, decimation_factor_x_8, temp_cic);
                                                 // CIC filter

            /* 2 stage fir filter */
            firdec(out_cic, h8, out8, &dp8, NH8, 32, D8);
            firdec(out8, h4, out4, &dp4, NH4, 16, D4);

            /* Store output in Q array */
            for (j = 0; j < 8; j++)
            {
                Q[i*8 + j] = out4[j];
            }
        }
/*****************************************************************************/
/*      End of decimation                                                    */
/*****************************************************************************/

        syncpoint = sync_corr(I, Q, rx_filt);            // Determine pos of sync-word

        rx_RRC_filt(I, Q, syncpoint, rx_filt, I_bits, Q_bits, 200); // Root raised co

        cdma(I_bits, Q_bits, code, 16, 24, 200);         // Multiply with the same PN c

        demod_QPSK(I_bits, Q_bits, rx_idata);            // demodulate the two channel

        compare(rx_idata, &BER);
    }
}
```

Location	Count	Average	Total	Maximum	Minimum
MAIN.C line 373	30	4353.0	130590	4353	4353
MAIN.C line 372	30	24484102.8	734523084	25755343	16624763

Figure 16: profile points

CHAPTER 4

Software

4.1 Introduction

The previous chapter describes the hardware that is used in this project. This chapter, Software, discusses the software part of the software radio. The source code is written as much as possible in C; assembler statements are only used for the initialization of the DSP. For that reason, porting the source code to other hardware should be easy. The next section describes the CWTS standard. The software radio uses a standard that is derived from CWTS. The final section will deal with the practical implementation of the software radio. The architecture of the receiver and transmitter and also design choices are discussed in this section.

4.2 The China wireless telecommunication standard (CWTS)

A wireless telecommunication standard is very complex; besides the physical layer there are many other aspects, such as the billing system, service channels, etc. This section only describes the physical layer of the communication between base station and user terminal of the CWTS standard. The physical layer consists of the following parts: multiple access, frame structure, modulation, and filter characteristics. These parts will also be discussed in this order.

4.2.1 Multiple access

The CWTS standard uses a combination of CDMA and TDMA for multiple access. The used CDMA component is *direct-sequence code division multiple access* (DS-CDMA). In DS-CDMA systems [28], the spreading code is a sequence of bits (known as chips). First an XOR operation is carried out between the message and the spreading code. This XOR operation is also known as *chipping*. So a "0" is represented by a chip sequence and a "1" is represented by the inverse of this chip sequence. Thus instead of transmitting the message bits the accompanying chip sequences are transmitted. The DS-CDMA technique is shown in Figure 17.

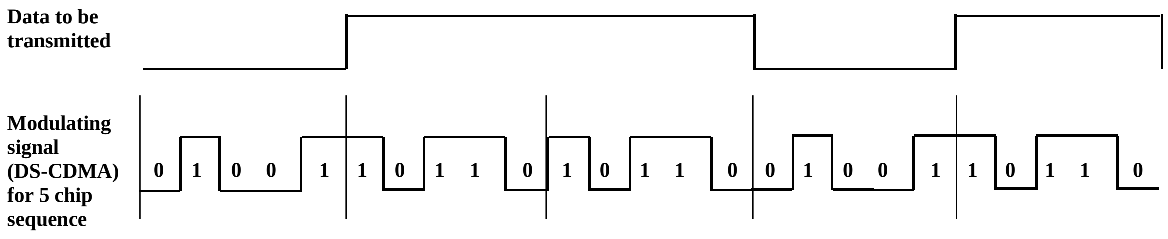


Figure 17: DS-CDMA modulation

The chip rate of the DS-CDMA technique in CWTS is equal to 1.28 Mchips/s which results in a bandwidth of approximately 1.6 MHz. Furthermore, the system uses a 200-kHz carrier raster. So, eight channels utilize the same 200-kHz band. CWTS uses besides DS-CDMA a TDMA component, namely TDD (*time-division duplex*). TDD mode is defined as follows:

TDD [27]:

> *A duplex method whereby forward link and reverse link transmissions are carried over same radio frequency by using synchronized time intervals. In the TDD, time slots in a physical channel are divided into transmission and reception part. Information on forward link and reverse link are transmitted reciprocally.*

4.2.2 Frame structure

The CWTS standard uses a four-layer structure. The first level consists of super frames, which contain 72 radio frames of 10 ms. Radio frames are divided into two 5-ms sub frames. In each sub frame, there are 7 main time slots and 3 special time slots. The complete physical channel signal format is presented in Figure 18. The 7 main time slots can be used for down-link communication (DL#n) or up-link communication (UL#m). Between the down-link and up-link communication there are 3 special time slots. The first one is the *down-link pilot symbol* (DwPTS) that is used for down-link synchronization. The next time slot is a *guard period* (GP) used to separate down and up-link communication. The last special time slot is *up-link pilot symbol* (UpPTS) which is used for up-link synchronization.

So, all physical channels consist of a four-layer structure of super frames, radio frames, sub frames and time slots/codes. The configuration of sub frames or time slots depends on the resource allocation. Between every time slot there are guard symbols for separation. The basic physical channel is defined as the association of one code, one time slot and one frequency.

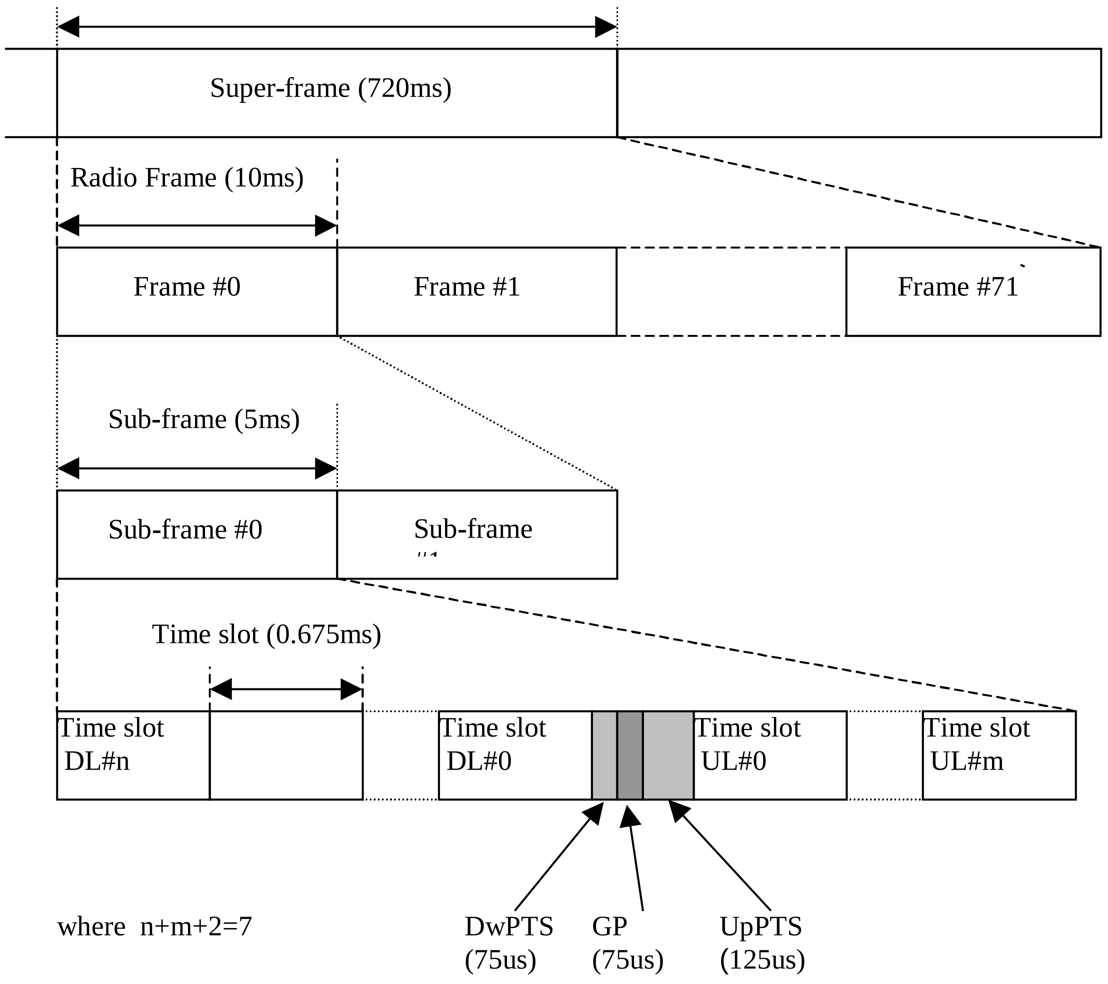


Figure 18: physical channel structure [26]

Sub-frame structure

As mention before, each sub frame (Figure 19) is subdivided into 7 main time slots (TS) of 675 µs duration each and 3 special time slots: DwPTS (down-link pilot symbol), GP (guard period) and UpPTS (up-link pilot symbol). The 7 main time slots can be used both for up and down-link communication.

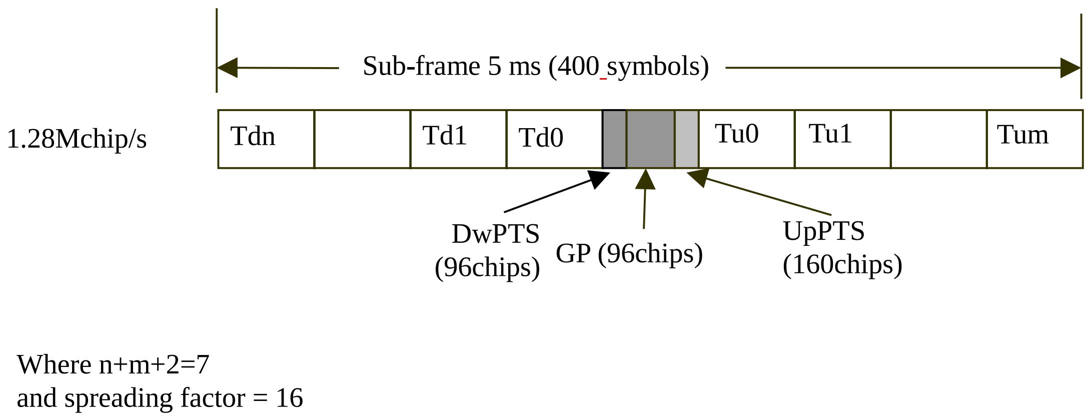


Figure 19: sub frame structure [26]

The DwPTS in each sub-frame is designed for both down-link pilot and synchronization channel. This time slot is usually composed of 64 chips of synchronization word (SYNC) and 32 chips of guard period as shown in Figure 20. The contents in the SYNC are a set of Gold codes. The Gold-code set is designed to distinguish nearby cells for the purpose of easier cell measurement. More information about the construction of the Gold-code set can be found in [25].

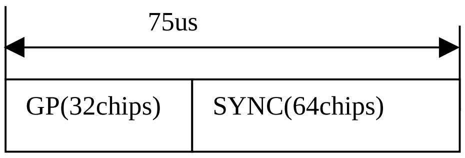


Figure 20: burst structure of DwPTS [25]

The UpPTS in each sub frame is designed for both up-link pilot and synchronization channel. The time slot is usually composed of 128 chips of synchronization word (SYNC1) and 32 chips of GP as shown in Figure 21. The contents in the SYNC1 are a set of Gold codes. The Gold code set is designed to distinguish different mobile telephones in access procedures. More information about the construction of the Gold-code set for the up-link synchronization can be found in [25].

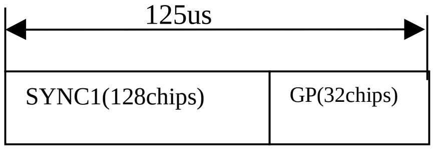


Figure 21: burst structure of UpPTS [25]

The guard period (GP) between the DwPTS and the UpPTS time slot has the duration of 75 µs (96 chips).

Time slot structure

The information rate of the channel varies with the symbol rate being derived from the 1.28 Mchips/s chip rate and the spreading factor. The spreading factor for both up link and down link has a range from 16 to 1. Thus the respective modulation symbol rates vary from 80.0K symbols/s to 1.28M symbols/s (Table 3). Table 4 shows the detailed time-slot structure that is also shown graphically in Figure 22. The training sequences, i.e. mid amble, of different users active in the same time slot are time-shifted versions of one single periodic basic code. Different cells use different periodic basic codes, i.e. different mid amble sets. In this way joint channel estimation for the channel impulse responses of all active users within one time slot can be done by one single cyclic correlation. The different user-specific channel-impulse-response estimations are obtained sequentially in time at the output of the correlator.

Spreading factor (Q)	**Number of symbols (N) per data field in Burst**
1	352
2	176
4	88
8	44
16	22

Table 3: number of symbols per data field in bursts [25]

Chip number (CN)	**Length of field in Chips**	**Length of field in Symbols**	**Length of field in μs**	**Contents of field**
0-351	352	Cf Table 3	275	Data symbols
352-495	144	9	112.5	Mid amble
496-847	352	Cf Table 3	275	Data symbols
848-863	16	1	12.5	Guard period

Table 4: contents of the burst fields [26]

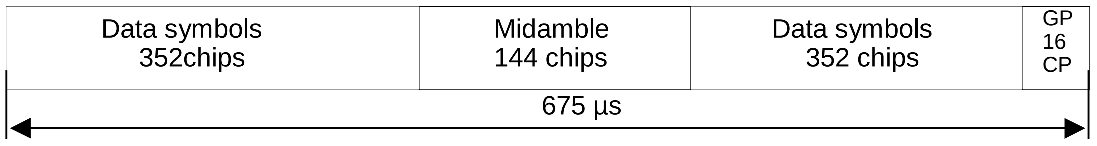
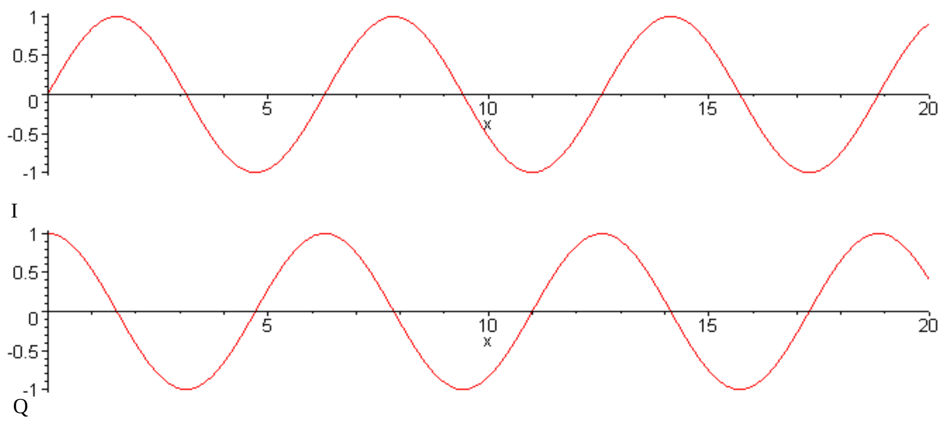


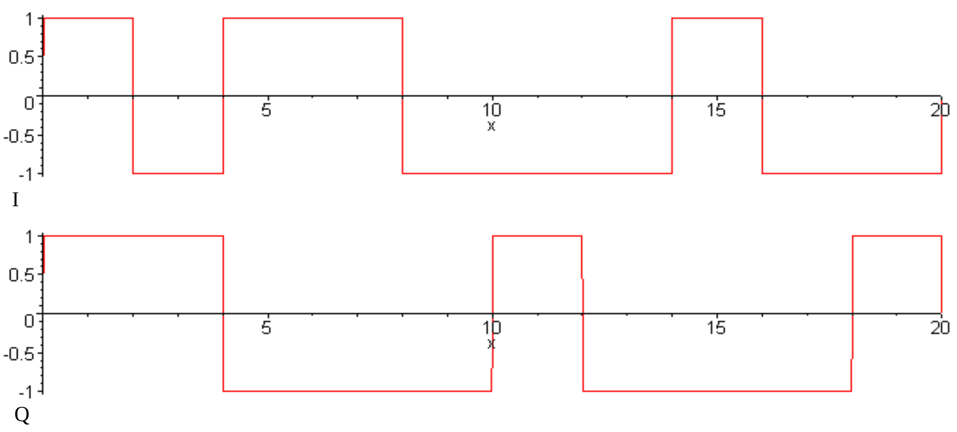
Figure 22: burst structure (GP denotes the guard period and CP the chip period.) [26]

4.2.3 QPSK modulation

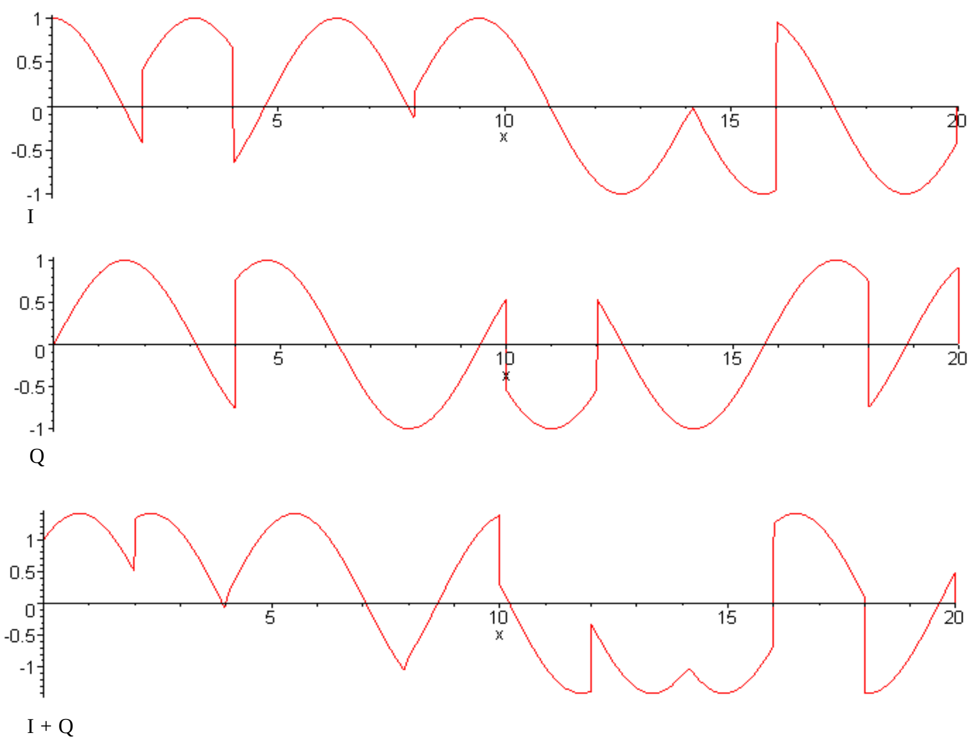
The CWTS standard uses the *quadrature-phase-shift-keying* (QPSK) -modulation technique for transmission. QPSK uses two channels for transmission, an in-phase (I) and quadrature phase (Q) channel. The I channel is multiplied with a cosine and the Q channel with a sine. Because a sine is orthogonal to a cosine, both carriers can use the same frequency. Data is sent by changing the phase of the carriers, A binary "1" is leads to a phase shift of 180 degrees, whereas a binary "0" leads to no phase shift. A phase shift of 180 degrees equals multiplying the sine with "-1" and no phase shift equals multiplying with "1". Figure 23 shows the different steps of QPSK modulation. Figure 23a shows the two carrier signals (I and Q) and Figure 23b represents the data signals. The modulated signal is the multiplication of the carriers with the data signals and is shown in Figure 23c. To reduce *inter-symbol interference* (ISI), the output of the QPSK modulation is led through a *root-raised-cosine filter* with a roll-off factor α of 0.22. The roll-off factor α defines the sharpness of the filter. If $\alpha = 1$, the root-raised-cosine filter is an ideal low-pass filter. On the other hand if $\alpha = 0$ the root-raised-cosine filter is an all-pass filter. More information about root-raised-cosine filter can be found in [19]. The basic modulation parameters used by CWTS are listed in Table 5.



a) carriers



b) data signals



c) modulated signal

Figure 23: QPSK modulation

Chip rate	1.28Mcps
Carrier spacing	1.6MHz
Data modulation	QPSK
Chip modulation	Root-raised cosine Roll-off $\alpha = 0.22$
Spreading characteristics	Orthogonal Qchips/symbol, where $Q = 2^p$, $0 <= p <= 4$

Table 5: basic modulation parameters of CWTS [25]

4.2.4 Filter characteristics

A software-radio transceiver is used for user terminals, not for base stations. Therefore, the filter characteristics of the user terminal are only discussed in this section. More information about base-station filter characteristics can be found in [29] and user-terminal filter characteristics are described in [30]. The input signal at the receiver can have a large input range. The minimal (reference) sensitivity is –135 dB when the data rate is 12.2 kbps. An input signal at the reference sensitivity level should not have a BER larger than 0.001. The maximal input level on the other hand cannot be described by an absolute value. It depends on the BER which should not be larger than 0.001. More information can be found in [31].

The filter characteristics of the receiver cannot be obtained directly from the CWTS standard. But the standard describes several definitions and requirements for these definitions from which the filter characteristics can be estimated [30].

- *Adjacent Channel Selectivity (ACS) [31]:*

  Adjacent Channel Selectivity is a measure of the capability of the user-terminal receiver to receive a wanted signal at its assigned channel frequency in the presence of adjacent channel signal at a given frequency offset from the center frequency of the assigned channel. ACS is the ratio of the receive-filter attenuation on the assigned channel frequency to the receiver filter attenuation on the adjacent channel(s).

  The minimal requirement of ACS is 33 dB [31]. The BER should also be smaller than 0.001.

- *Blocking characteristics [31]:*

  The blocking characteristics is a measure of the receiver ability to receive a wanted signal at is assigned channel frequency in the presence of an unwanted interferer on frequencies other than those of the spurious response or the adjacent channels. The blocking performance shall apply at all frequencies except those at which a spurious response occur.

  Multiple requirements are given for the blocking characteristics. The BER should not be larger than 0.001 when an interferer is 58 dB [31] stronger than the wanted signal.

- *Spurious response characteristics [31]:*

  Spurious response is a measure of the receiver's ability to receive a wanted signal on its assigned channel frequency without exceeding a given degradation due to the presence of an unwanted unmodulated *interfering signal at any other frequency at which a response is obtained i.e. for which the blocking limit is not met.*

The BER should not be larger than 0.001 when an unmodulated interferer is 58 dB [31] stronger than the wanted signal.

- *Inter-modulation response characteristics [31]:*
  Third and higher order mixing of the two interfering RF signals can produce an interfering signal in the band of the desired channel. Inter-modulation response rejection is a measure of the capability of the user terminal receiver to receive a wanted signal on its assigned channel frequency in the presence of two or more interfering signals, which have a specific frequency relationship to the wanted signal.

  The BER should also not be larger than 0.001 when the interfering signals are 56 dB [31] stronger than the wanted signal.

- *Spurious emissions [31]:*
  The Spurious Emissions Power is the power of emissions generated or amplified in a receiver that appear at the user terminal antenna connector.

  The most stringent requirement for spurious emissions is –90 dB [31].

A combination of the above mentioned definitions and accompanying requirements results in the following estimated filter characteristics, shown in Table 6 and Figure 24. The most important requirements are the ACS and the intermodulation response characteristics. The latter one is the most critical. Therefore the minimal stop band attenuation is equal to this requirement, –56 dB.

Parameter	**Value**	**Unit**
Minimal attenuation stop band	56	dB
Pass band	0.64 (0.5 *1.28)	MHz
Transition band	0.16 (0.80 – 0.64)	MHz

Table 6: filter characteristics of CWTS

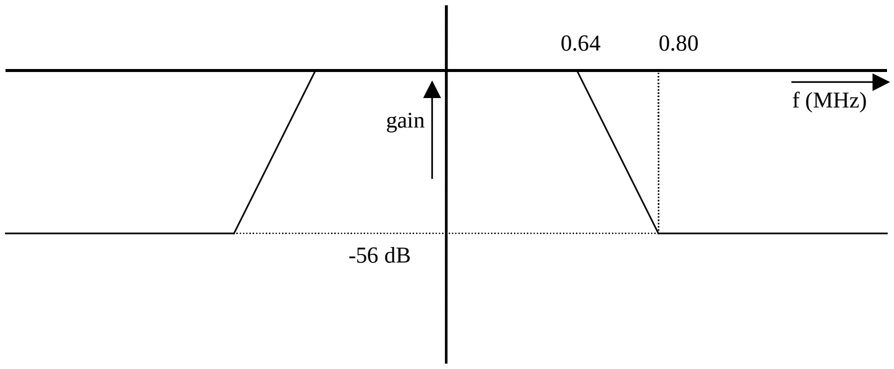


Figure 24: filter characteristics of CWTS

4.3 Practical implementation

4.3.1 Hardware limitations

The Evaluation Module from Spectrum Digital is designed for audio applications. For that reason the maximal sample frequency of the AD/DA converter is low (64 kHz). The software radio cannot utilize this maximum sample frequency because of the time requirements of the interrupt routine for the ADC of the software radio receiver. The receiver must detect a frame burst, which is carried out in this interrupt. If the interrupt routine is not finished when the next sample arrives, data is lost. Therefore the sample frequency is limited to 16 kHz. Another limitation of the Evaluation Modules is the size of the data memory (64 kb). This size is too small for a direct implementation of the CWTS standard. Therefore a derived frame structure is used, shown in Figure 25. This frame structure contains only one sync word and one time slot. The sync word is chosen arbitrarily and is not based on the gold-code set, defined by the CWTS standard.

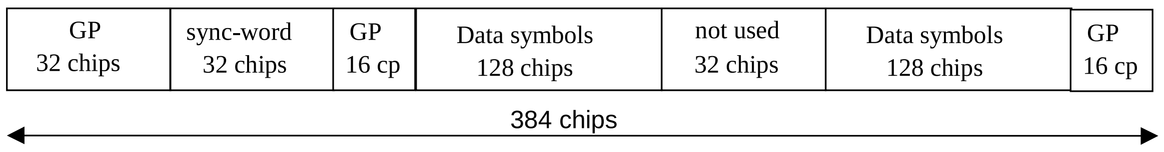


Figure 25: new frame structure (GP denotes the guard period and CP the chip period)

4.3.2 Cascaded integrated comb (CIC) filters

The software-radio receiver filters a channel in multiple stages. A multi-stage filter requires less computational power than a one-stage filter [33]. The receiver uses special filters in the first stages of the decimation (filtering) process. These special filters are *cascaded integrated comb* (CIC) filters. More information about CIC filters can be found in [9] and [34]. CIC filters can only be used in the first steps of the decimation process because the frequency response is bad. (An example is shown in Figure 27.) CIC filters requires only additions and subtractions, which is a great advantage because normal FIR filters require additions and multiplications. Multiplications are difficult to calculate on DSPs and therefore slow compared with an addition. An example of a CIC filter is shown in Figure 26. The frequency response of CIC filters can be captured in the following equation (where M = decimation factor and L = number of integrators/combs) [34]:

$$H(e^{j\omega}) = \left(\frac{\sin \frac{\omega M}{2}}{M \sin \frac{\omega}{2}} e^{-j\omega[(M-1)/2]} \right)^{L} \qquad (1)$$

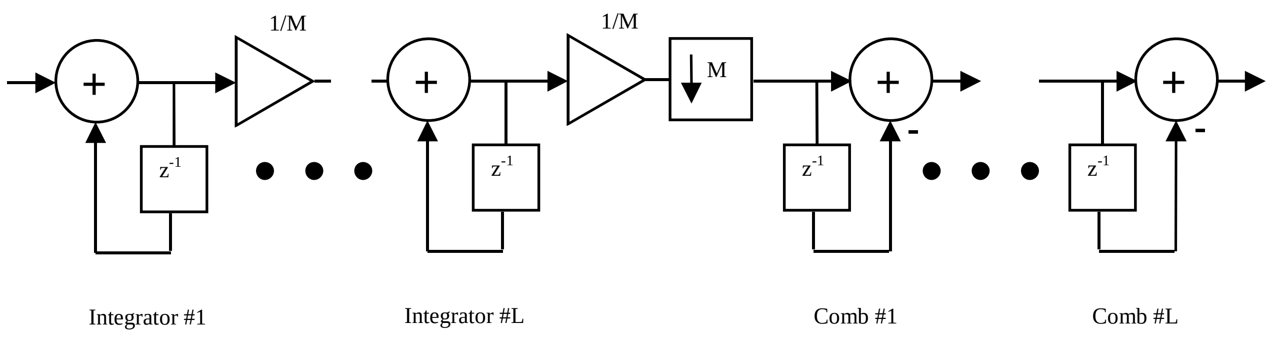


Figure 26: cascaded integrated comb filter

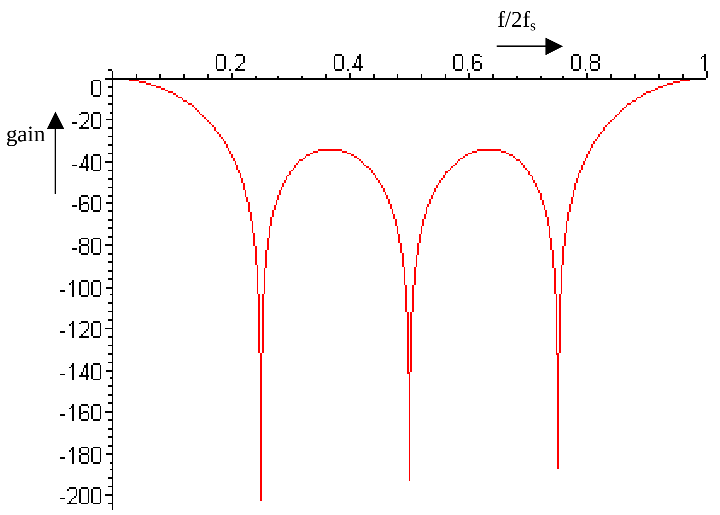


Figure 27: frequency response of a CIC filter

4.3.3 TMS320C54x DSPLIB

The Texas Instruments TMS320C54x DSPLIB is an optimized DSP function library for TMS320C54x processors. It contains more than 50 assembly-optimized general-purpose signal-processing routines that can be called from C. This project uses several functions of this library. The used functions are *firdec* and *firinterp*. Firdec is an assembler optimized decimating FIR filter and firinterp is an interpolating FIR filter. The DSPLIB library is freeware and can be downloaded from the following location:

https://www-a.ti.com/apps/c5000/xt_download.asp?sku=C54x_DSPLib

More information about the DSPLIB can also be found in the accompanying manual [37].

4.3.4 Transmitter

This section discusses the internal structure of the software-radio transmitter. The transmitter uses the frame-structure shown in Figure 25. Figure 28 shows the internal structure of the transmitter. It contains eight steps that are described in the following subsections. The source code of the transmitter can be found in Appendix I.

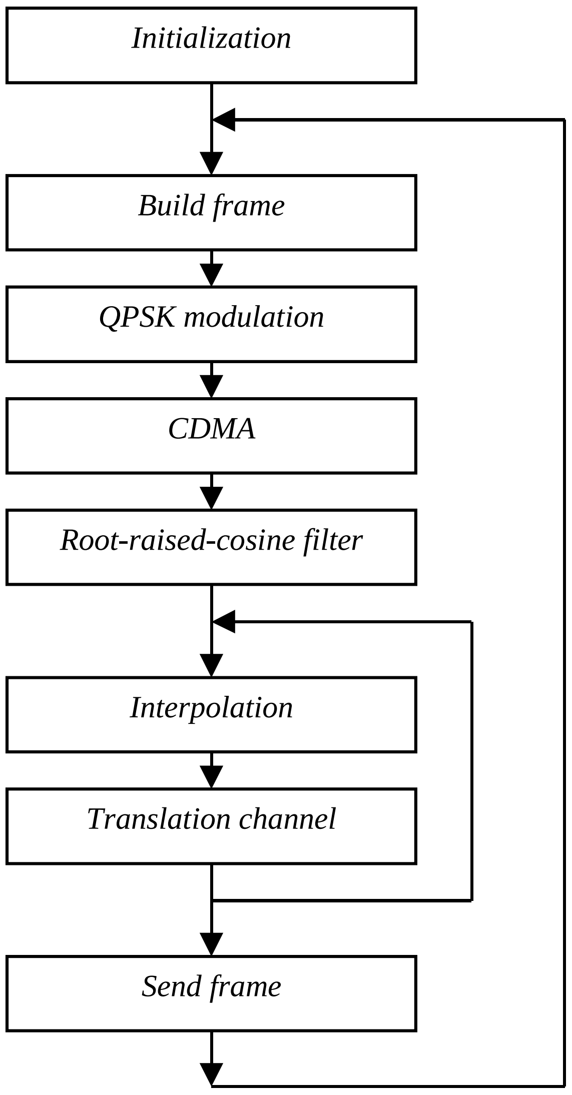


Figure 28: functional block diagram of the software-radio transmitter

Initialization

This step initializes the DSP and the AD/DA chip of the evaluation module. Also the communication between the two chips is set up. Figure 29 shows the initialization function in more detail. The first step is to set the location of the interrupt table. In this table functions are assigned to interrupts. If an interrupt occurs, the DSP uses this table to call the appropriate function. Then, the serial-port receive interrupt is enabled. The DSP uses this port to communicate with the AD/DA chip. The next step is the configuration of the AD/DA chip. All control registers of the AD/DA chip are set to the appropriate value. After the configuration the serial-port interrupt is disabled.

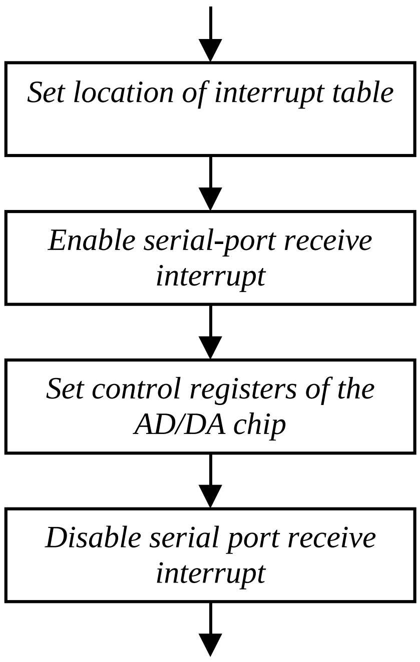

Set location of interrupt table
Enable serial-port receive interrupt
Set control registers of the AD/DA chip
Disable serial port receive interrupt

Figure 29: initialization block diagram

Build frame

In this block, a frame is built up using the structure of Figure 25. At this point the frame consists only of 0's and 1's and the frame has a size of 384 bits. Figure 30 shows a detailed block diagram of the build-frame block. First the *build-frame-data* function is called. This function creates the data part of the frame (256 bits). The *spreading factor* defines how many times an information bit is repeated in the frame. If, for example, the spreading factor is 16 the frame consists only of 16 information bits (, which are repeated 16 times). In the next function, *build frame,* the frame is built up using the structure of Figure 25.

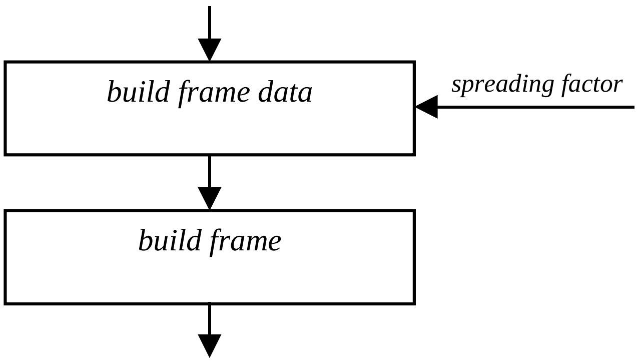


Figure 30: build-frame block diagram

QPSK modulation

This step modulates the frame with QPSK. Odd bits are transmitted via the I channel (cosine) and even bits via the Q channel (sine). A one causes a 180 degrees phase shift, so a –1 is transmitted. On the other hand if a bit is 0 (0 degrees phase shift), a 1 is transmitted. The QPSK modulation can be represented by the following formula:

$$out = 1 - 2 * in \qquad (2)$$

where *in* $\in$ {0, 1}

After modulation there are two arrays, one with I-channel symbols and one with Q-channel symbols. They have an equal size of 192 16-bit integers (2 * 192 = 384).

CDMA

This block multiplies the two arrays with a PN code. The length of the PN code depends on the spreading factor. If the spreading code is 16 for example, the PN code is also 16 chips long and every information bit in the frame is repeated 16 times. Note: only the data is multiplied, the sync word is unaffected. The CDMA process can be captured in the following pseudo source code:

```
j = 0;

for (i = data_start; i < data_end; i++)
{
      I[i] = I[i] * PN_code[i];
      Q[i] = Q[i] * PN_code[i];

      //update index of PN code:
      j = (j + 1) % spreading_factor;
}
```

Figure 31: pseudo source code of CDMA block

Root-raised-cosine filter

To reduce ISI a root-raised-cosine filter is used with α = 0.22. The first step of this block (Figure 32) is to interpolate the I and Q channel with a factor 4. In this interpolation process only zeros are added. Then the data stream is led through a 48-tap root-raised-cosine filter with α = 0.22. The results of this step are two arrays (I and Q channel) which have a size of 768 (4*192) 16-bit fixed-point numbers.

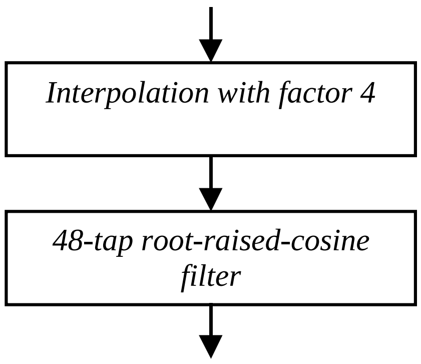


Figure 32: root-raised-cosine filter block diagram of the transmitter

Interpolation

The carrier frequency has a much higher sample rate than the symbol rate of the frame. Therefore, the I and Q channel are interpolated with a factor 16 in this block to match with the sample rate of the carrier frequency. To reduce memory space, the frame is also broken into pieces. Once one piece has been interpolated, the next step, Translation channel, will be carried out. Furthermore the interpolation process is divided into 4 stages for efficient use of processing power. Figure 33 shows a block diagram of the interpolation step. The frequency responses of the used interpolation filters are shown in Appendix II.

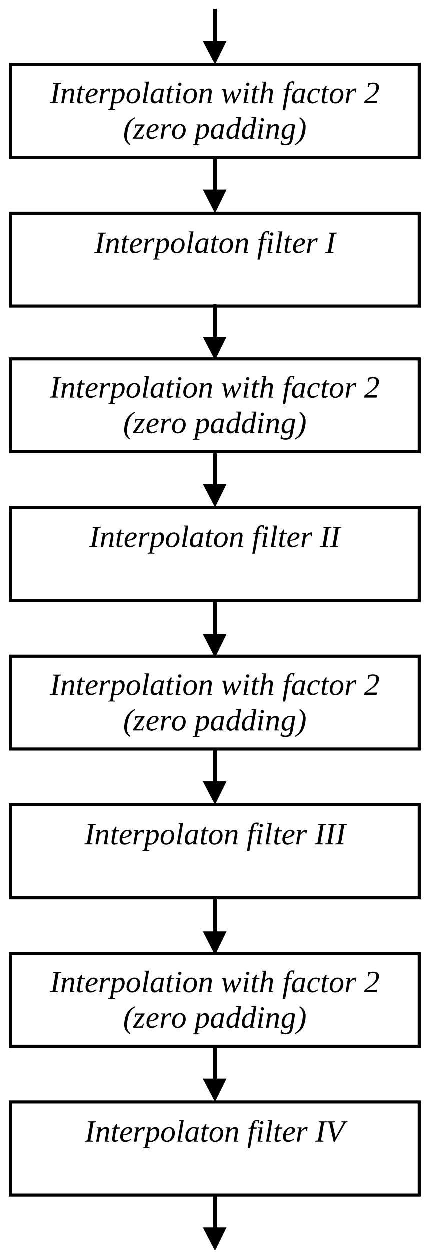

Interpolation with factor 2 (zero padding)
Interpolaton filter I
Interpolation with factor 2 (zero padding)
Interpolaton filter II
Interpolation with factor 2 (zero padding)
Interpolaton filter III
Interpolation with factor 2 (zero padding)
Interpolaton filter IV

Figure 33: interpolation block diagram

Translation channel

This block transforms the QPSK signal to a channel of the system band. So the signal is transformed to a higher frequency. The interpolated part of the I channel is multiplied with a cosine and the Q channel with a sine. First the I channel is multiplied with values from a cosine table and the result is stored in an output table. Then, the Q channel is interpolated and multiplied with values from a sine table. The result is added to the output table. Figure 34 shows a block diagram of the translation-channel step. (The output vector has a size of 11648 16-bit integers.)

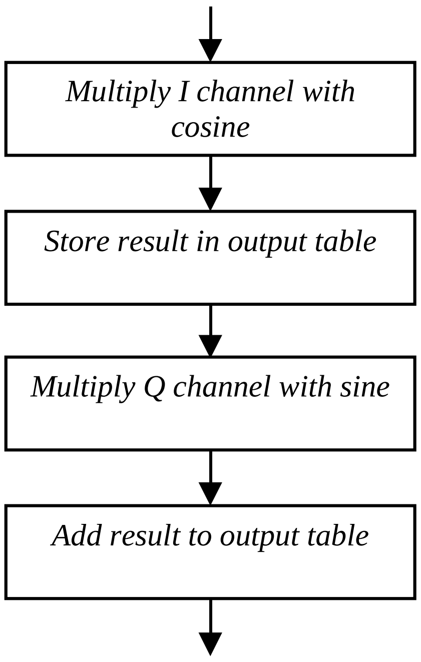

Multiply I channel with cosine
Store result in output table
Multiply Q channel with sine
Add result to output table

Figure 34: translation-channel block diagram

Send frame

The calculated output vector is transmitted in this step. By enabling the serial-port-receive interrupt the interrupt function sends the contents of the output vector to the AD/DA chip. Once the last byte has been transmitted the interrupt function disables itself by disabling the serial-port-receive interrupt.

4.3.5 Receiver

This section discusses the internal structure of the software-radio receiver. The receiver is more complex than the transmitter described in the previous section. A functional block diagram of the software-radio receiver is shown in Figure 35. All blocks are described separately in the subsections below. The source code of the software-radio receiver can be found in Appendix III.

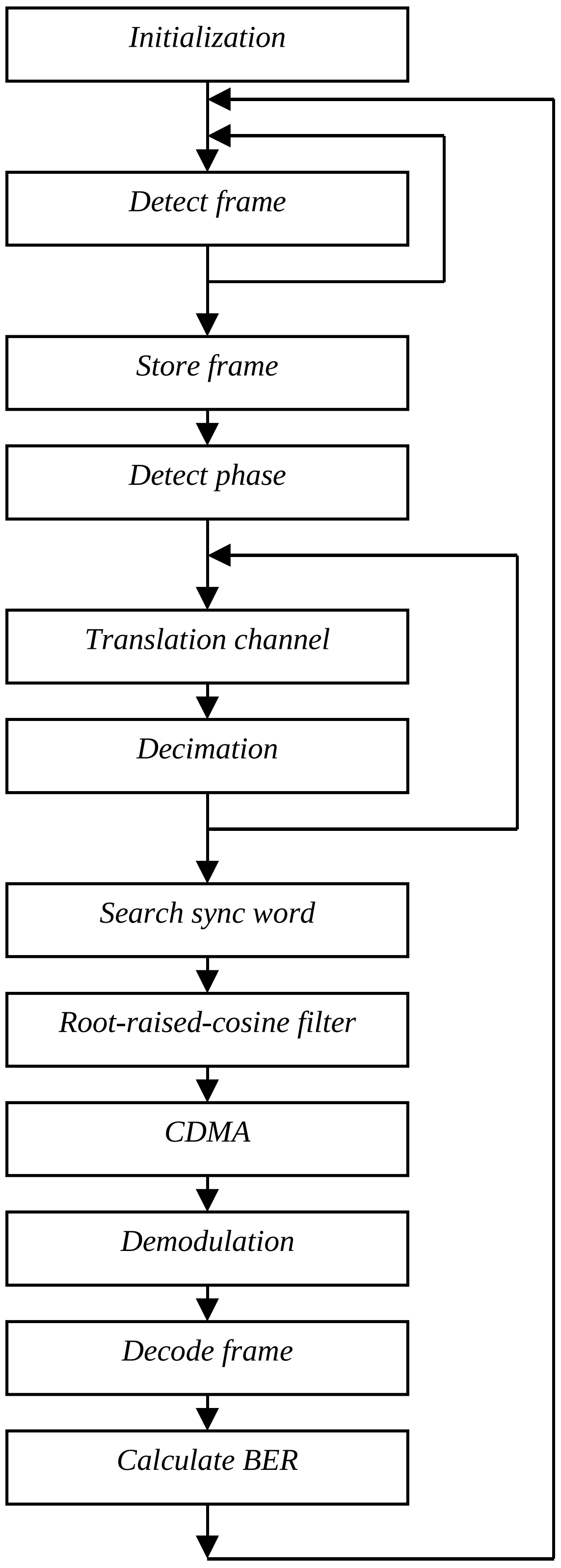


Figure 35: functional block diagram of the software-radio receiver

Initialization
This block is equal to the initialization block of the transmitter, which is discussed on page 37.

Detect frame

The receiver stays in this block/state until a frame is being received. If an incoming frame has been detected the receiver will go to the next block, Store frame. The internal structure of this block is shown in Figure 36. First the average value of the input must be smaller than a certain threshold I. If so, no transmission is taking place and only noise has being received. If it is established no frame is currently being transmitted, the system should wait for the start of the next frame. A frame is assumed to be transmitted if the signal level rises above a certain threshold 2. The next block, Store frame will store the incoming frame. The possibility exists that this function becomes active during a frame transmission. By using two thresholds this function will only proceed until the next frame is transmitted. If only one threshold is used, the receiver does not wait until the next frame and stores an incomplete frame into memory.

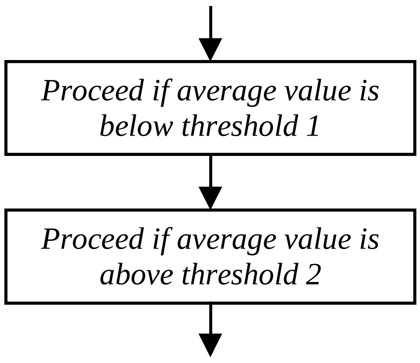

Proceed if average value is below threshold 1
Proceed if average value is above threshold 2

Figure 36: detect-frame block diagram

Store frame

This step stores 12800 samples into a vector after a frame has been detected. The size of this vector is large enough to contain one frame. This block and the previous block have been implemented in the serial-port receive interrupt routine. Therefore these two blocks must be programmed very efficiently. Only necessary functionality has been implemented in this routine. Furthermore the average value is only taken over 4 samples, to reduce the number of instructions.

Detect phase

The first step in decoding a frame is to detect the phase of the carrier. The phase of the carrier (which is the sum of a sine and cosine) is derived from a zero crossing of the signal. The zero crossing shown in Figure 37 has a phase of $-\frac{1}{4}\pi$. From this position the phase of the start position can be calculated. Accuracy is achieved by interpolating the input signal.

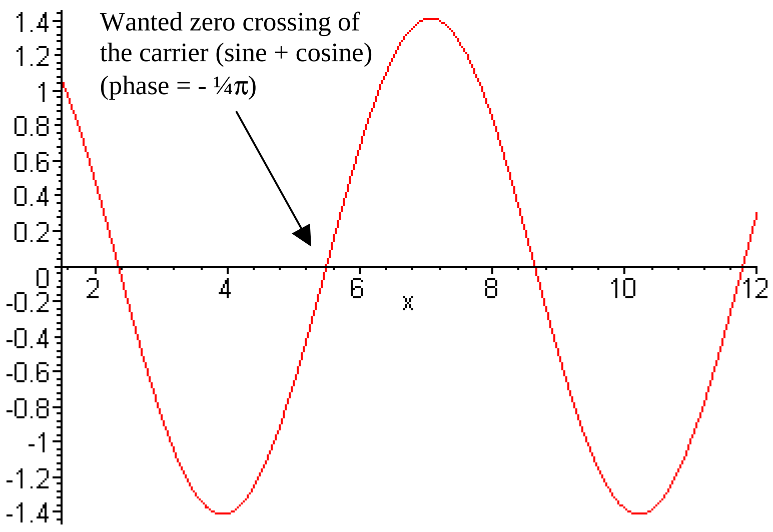


Figure 37: detection of the carrier phase

Figure 38 shows the internal structure of this block. The first step is to detect a zero crossing. If the previous sample has a negative value and the current sample a positive value, the zero crossing must be between these two samples. The next step is to save the position of the zero crossing. The sine table contains 32 samples and the carrier has a period of 4 samples. Therefore the samples around the zero crossing must be interpolated (linearly) with a factor 8. From these 8 samples the closest match of the zero crossing is determined. With these data the zero-phase point can be calculated. This point is used in the next block, Translation channel.

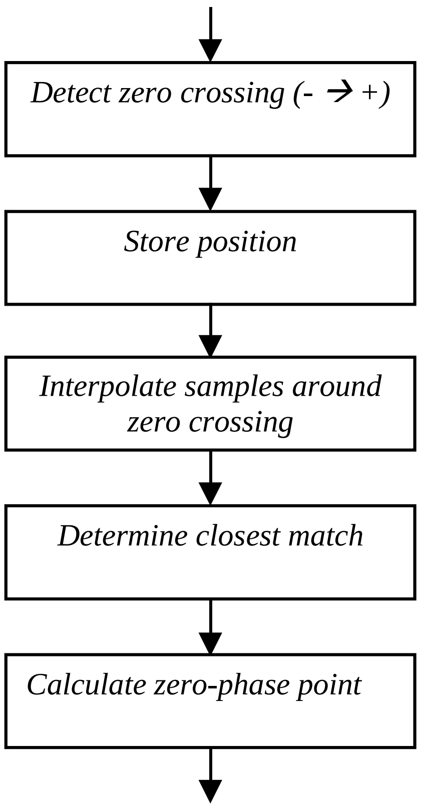

Detect zero crossing (- → +)
Store position
Interpolate samples around zero crossing
Determine closest match
Calculate zero-phase point

Figure 38: detect-phase block diagram

Translation channel
The channel is transformed to base-band frequencies by multiplying it with the carrier frequency. At this point two channels, the I channel and the Q channel are decoded from the input signal. The I channel is regained by multiplying the input signal with a cosine and the Q channel is recovered by multiplying the input with a sine. The zero-phase point from the previous block is used to align the initial phase of the cosine/sine with the initial phase of the carrier. Otherwise separation of the I channel and Q channel is not possible. This step and the following step are calculated by breaking the input signal into pieces to reduce memory space.

Decimation
The symbol rate is 64 times lower than the input sample rate. This step reduces the signal rate with a factor 16. Multi-stage decimation and the use of CIC filters accomplish an efficient decimation. Figure 39 shows the decimation process in more detail. The first stage is a CIC filter that decimates the data rate with a factor 4. Then, two decimating FIR filters are used to reduce the output of the CIC filter with a factor 4. The total decimation factor is thus 16. The results of this step are two arrays (I and Q channel) with a size of 800 16-bit integers. The frequency responses of the used decimation filters are shown in Appendix IV. The responses are based on the filter characteristics of CWTS, described in 4.2.

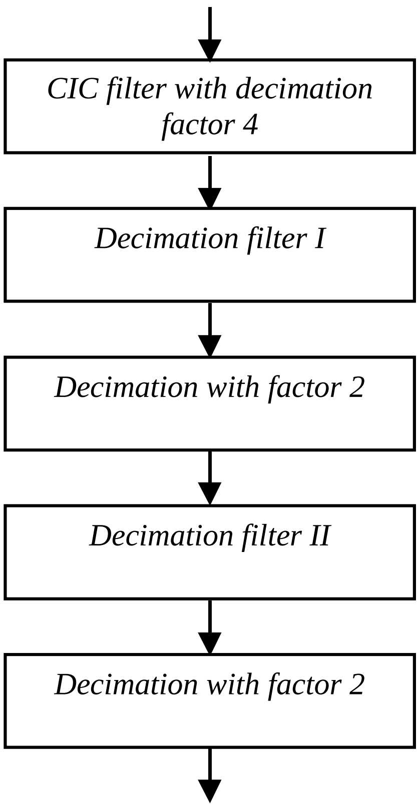


Figure 39: decimation block diagram

Search sync word
At this point the sample rate is still four times the symbol rate. Figure 40 shows the internals of this block. First the I and Q channel are filtered with a root-raised-cosine filter. The same filter is used at the transmitter and only the first parts of both vectors are filtered. The next step correlates the filtered data with the synchronization word. The maximal correlation indicates the position of the

sync word in the received frame. The position of the sync word is used in the next step to provide optimal decoding.

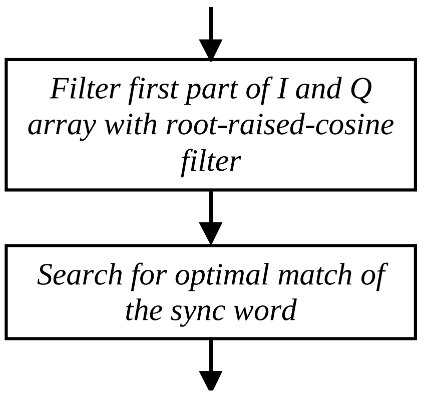

Filter first part of I and Q array with root-raised-cosine filter
Search for optimal match of the sync word

Figure 40: search-sync-word block diagram

Root-raised-cosine filter

The ISI is minimized if the receiver uses the same root-raised-cosine filter as the transmitter. So this block uses also a root-raised-cosine filter with α = 0.22. Figure 41 shows the internals of this block. Because the sample rate is 4 times the symbol rate, not all data is filtered, only the necessary filter output is calculated. The filtering begins with the start of the synchronization word. Then, every fourth output of the filter is calculated. Because the position of the sync word is determined very accurately, optimal decoding of the frame is achieved. The results of this step are two arrays (I and Q channel) which contain 188 samples. These samples represent the phase of both channels.

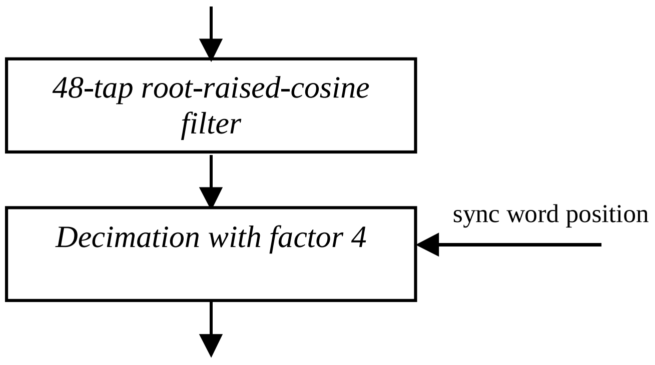


Figure 41: root-raised-cosine block diagram of the receiver

CDMA

After decimation the two arrays are multiplied with the same PN-code of the transmitter. The result is the original QPSK-signal. More information can be found in the CDMA block of the transmitter.

```
j = 0;

for (i = data_start; i < data_end; i++)
{
      I[i] = I[i] * PN_code[i];
      Q[i] = Q[i] * PN_code[i];

      //update index of PN code:
      j = (j + 1) % spreading_factor;
}
```

Figure 42: pseudo source code of CDMA block

Demodulation

The resulting I and Q array can easily be demodulated into data. If a symbol value is greater than zero the received bit is zero, and otherwise one. The demodulation process can be captured in the following pseudo source code (Figure 43).

```
j = 0;

for (i = data_start; i < data_end; i++)
{
      if (I_symbol > 0)
      {
            I_bit = 1;
      }
      else
      {
            I_bit = 0;
      }
      if (Q_symbol > 0)
      {
            Q_bit = 1;
      }
      else
      {
            Q_bit = 0;
      }
}
```

Figure 43: pseudo source code of demodulation block

Decode frame

This step extracts the data bits from the received frame data. The frame structure, described in Figure 25, is used to extract the data bits.

Calculate BER

The last step calculates the number of bit errors in the received frame. The received data is compared with a reference frame (The transmitter transmits always the same frame.) The BER is calculated by dividing the number of bit errors by the total number of received bits.

CHAPTER 5

Experiments

5.1 Introduction

This chapter describes several experiments that have been carried out with the implemented software-radio system. The purpose of these experiments was to verify the constructed system. First, Code Composer has been used as debugger to verify the software-radio transmitter and receiver. Section 5.2 describes the internals of the transmitter by using Code Composer and section 5.3 discusses the software-radio receiver. Finally, the performance of the software-radio system has been evaluated by measuring the BER versus bit-energy-to-noise-energy in section 5.4. Below the used hardware setup of the software-radio-system is discussed.

5.1.1 Software-radio-system setup

A complete radio system transmits and receives data through the air. In this project, only the IF stage of the software-radio system has been implemented. Therefore, to simulate a real radio system, the output of the software-radio transmitter is led through an FM transmitter. This transmitter modulates an analog input signal with FM modulation at 433 MHz, which is a free frequency band. An FM receiver receives this 433-MHz signal and demodulates the signal back to an analog signal. This analog signal is amplified in order to use the full range of the ADC of the DSP evaluation module. Finally, the software-radio receiver demodulates the analog signal into data. More information about the FM receiver and transmitter can be found in the datasheets [38] and [39]. The total software-radio-system setup is shown in Figure 44.

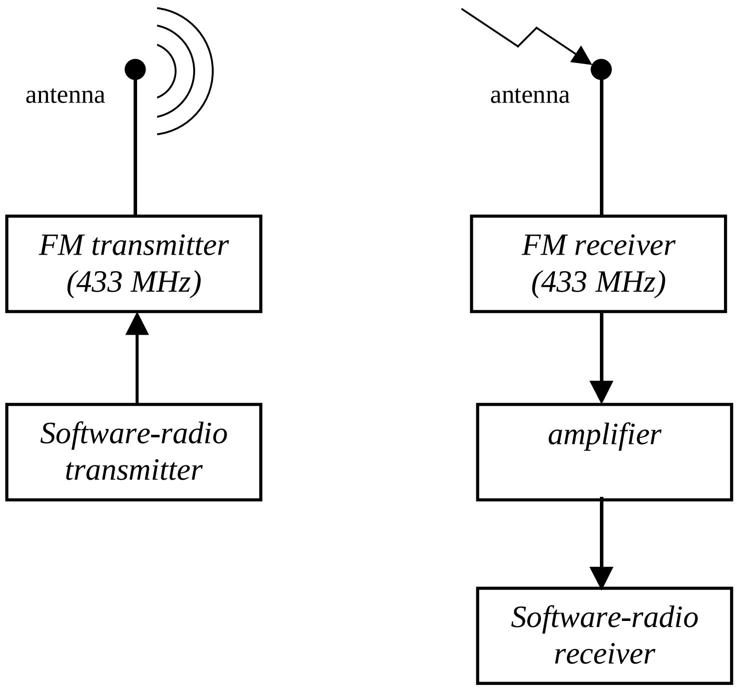


Figure 44: Software-radio-system set up

5.2 Inside the transmitter using Code Composer

Code Composer is a very useful program to verify the programmed software-radio transmitter. This section describes the output of every block of Figure 28. First the transmitter constructs a frame. The output of this block is shown in Figure 45. Then, the frame is QPSK modulated (Figure 46). After modulation the data part of the frame is multiplied with a PN code (Figure 47). This signal is led through a root-raised-cosine filter (Figure 48). The output of this filter is interpolated with a factor 16. Finally, the I channel is multiplied with a cosine and the Q channel is multiplied with a sine. These two channels are combined and transmitted (Figure 49). The DSP evaluation module is capable of transmitting 0.73 frames per second. If the serial-port receive interrupt is disabled, so no transmission takes place, the DSP is capable of calculating 1.33 frames per second. Furthermore, the DSP can process 100 million instruction cycles per second, so about 75 million instructions are needed for calculating one frame.

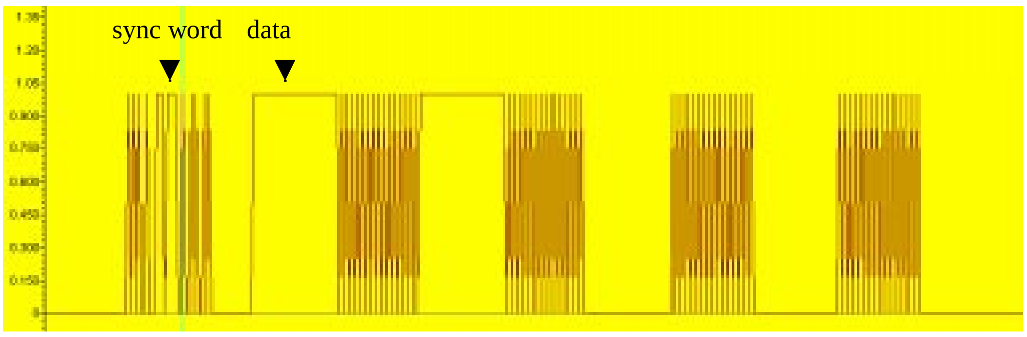


Figure 45: example of a frame

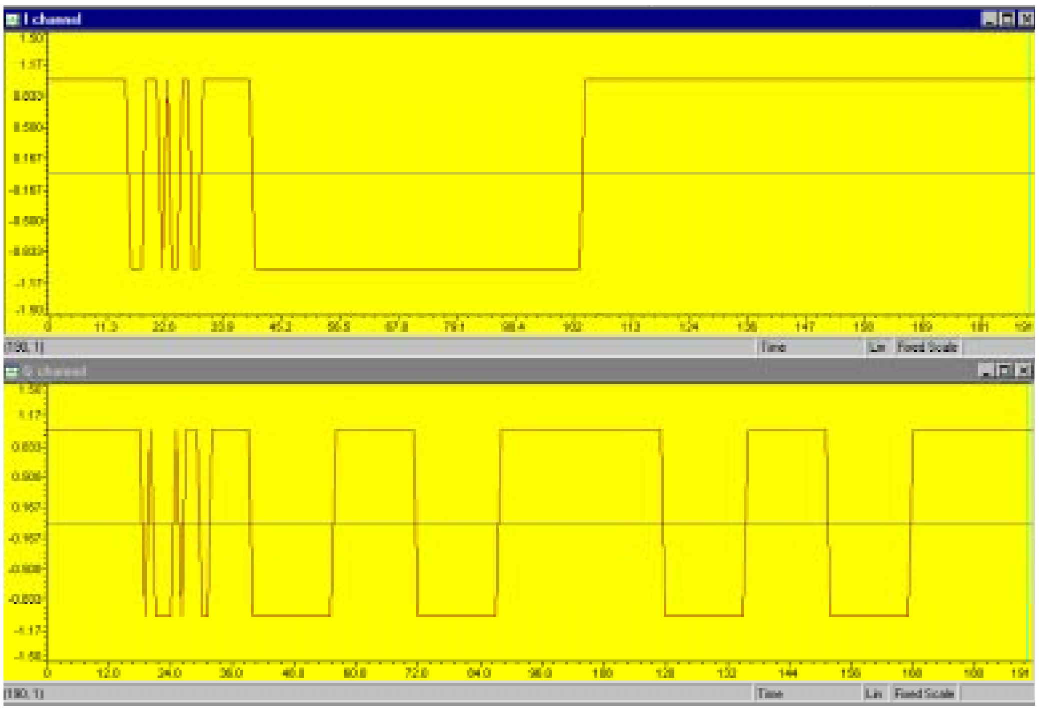


Figure 46: frame after QPSK modulation

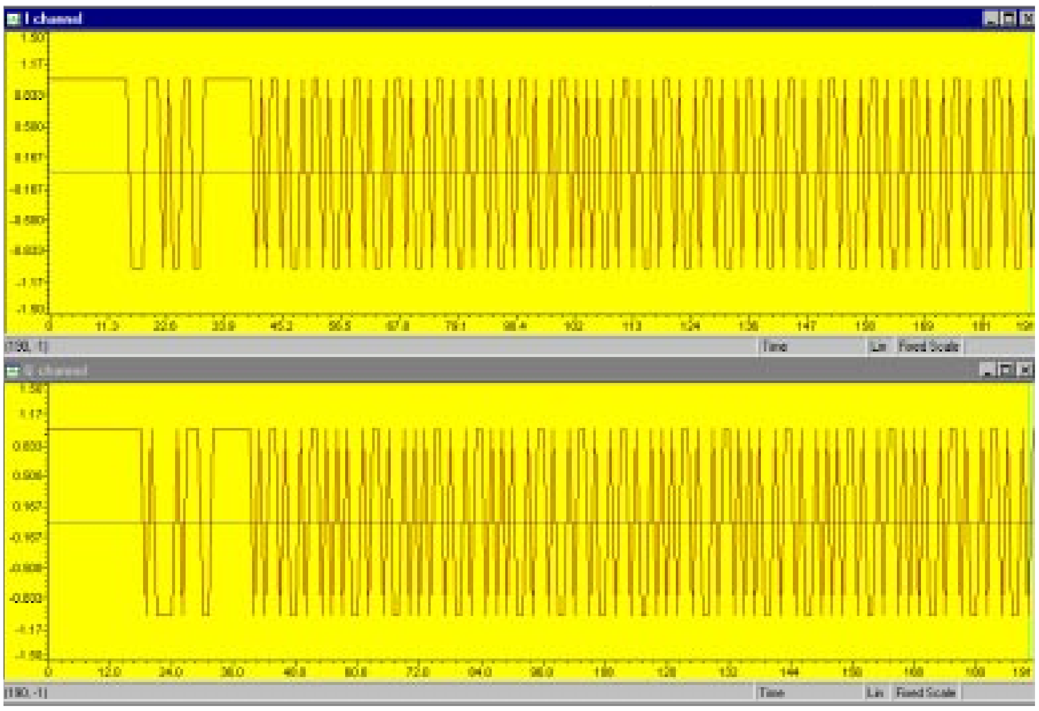


Figure 47: frame after CDMA

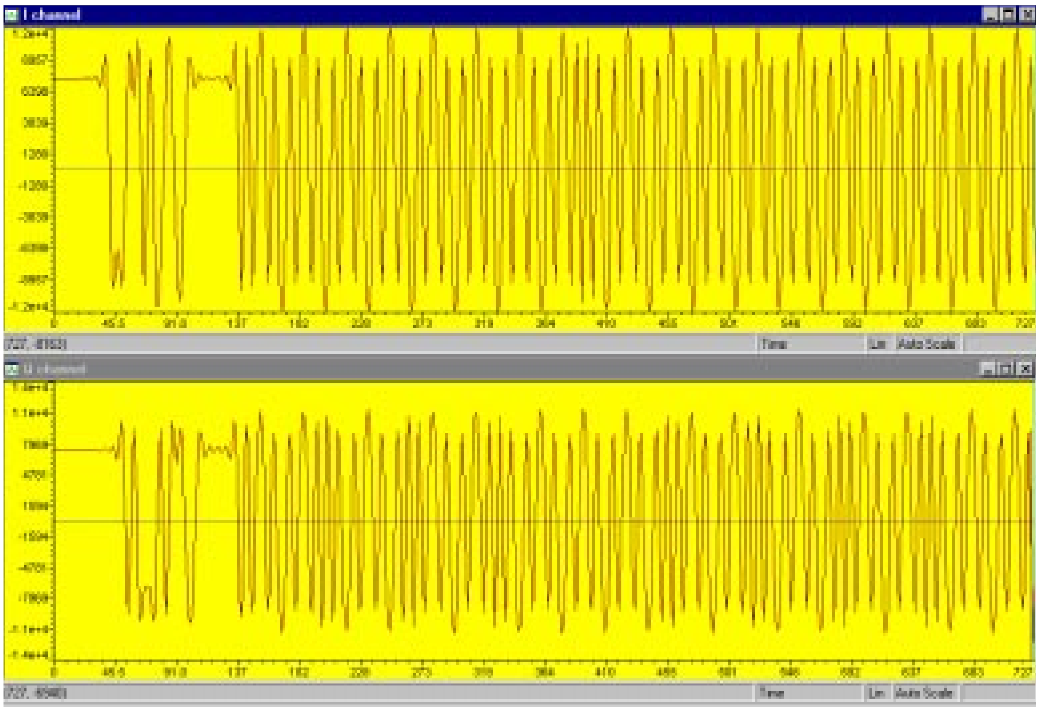


Figure 48: frame after root-raised-cosine filter

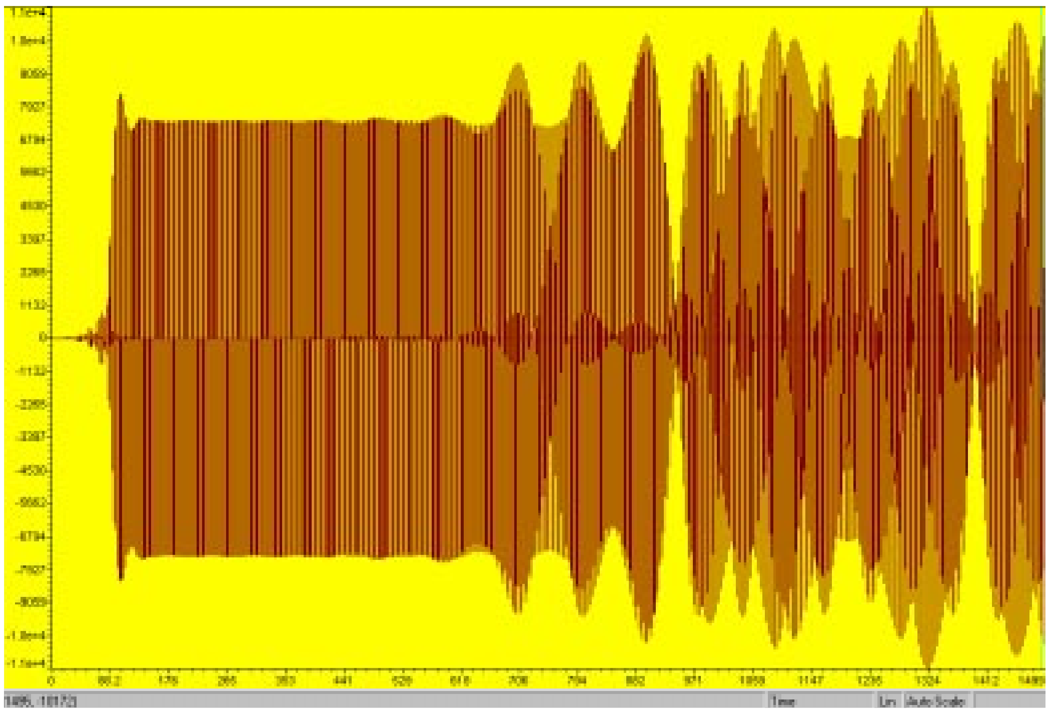


Figure 49: transmitted frame (first part)

5.3 Inside the receiver using Code Composer

This section describes the output of every block of Figure 35. The output plots are created with Code Composer. First the receiver detects a frame and stores it into a large vector (Figure 50). The next step is to multiply the frame with a sine and cosine to regain the I and Q channel. The output after decimation is shown in Figure 51. Then the signals are led through a root-raised-cosine filter (Figure 52). The original bit sequence is regained by multiplying the frame with the same PN code of the transmitter (Figure 53). Finally, the frame is demodulated (Figure 54) and the data is extracted. The receiver can decode 0.33 frames per second. If the serial-port receive interrupt is disabled the receiver can decode 1.27 frames per second. Furthermore the DSP can process 100 million instruction cycles per second, so decoding one frame takes about 80 million instructions.

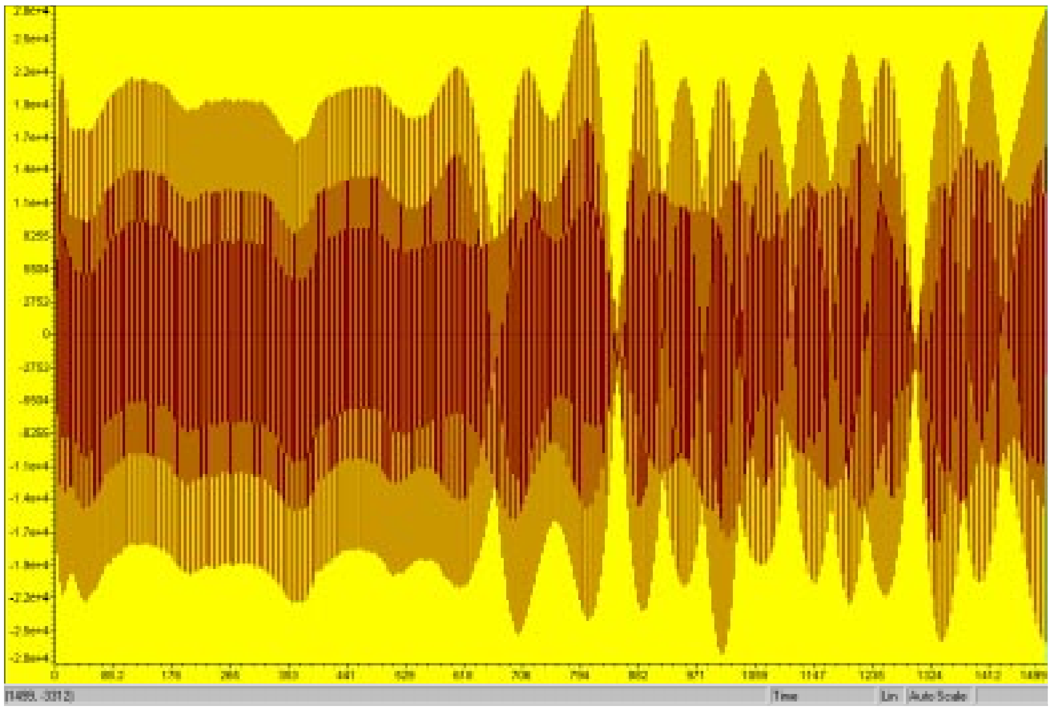


Figure 50: received frame (first part)

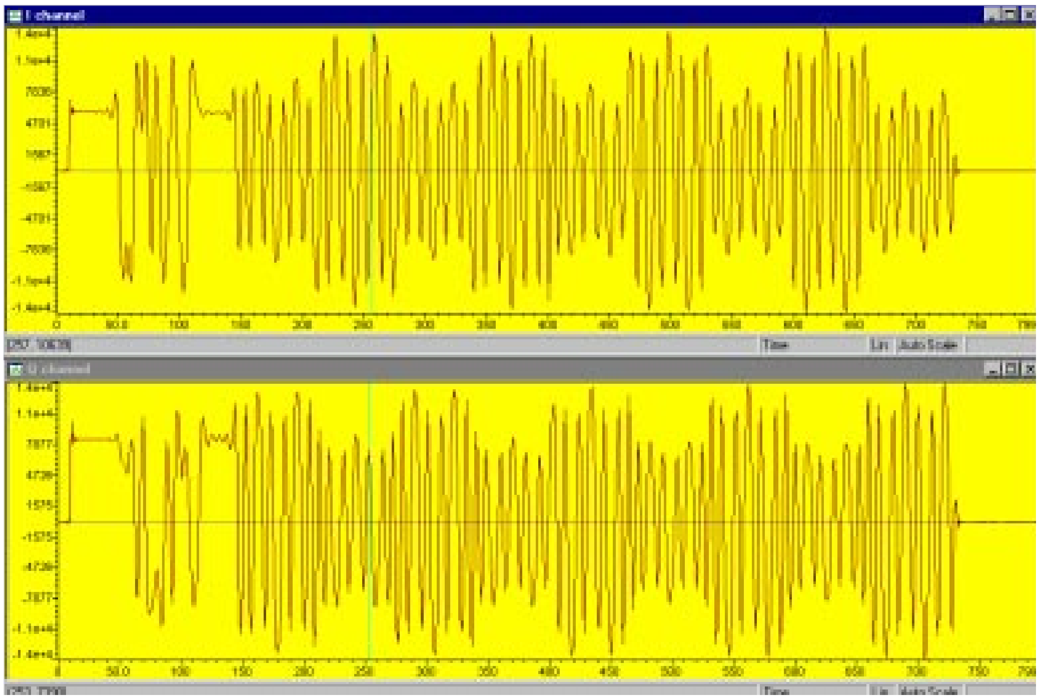


Figure 51: frame after channel separation and decimation

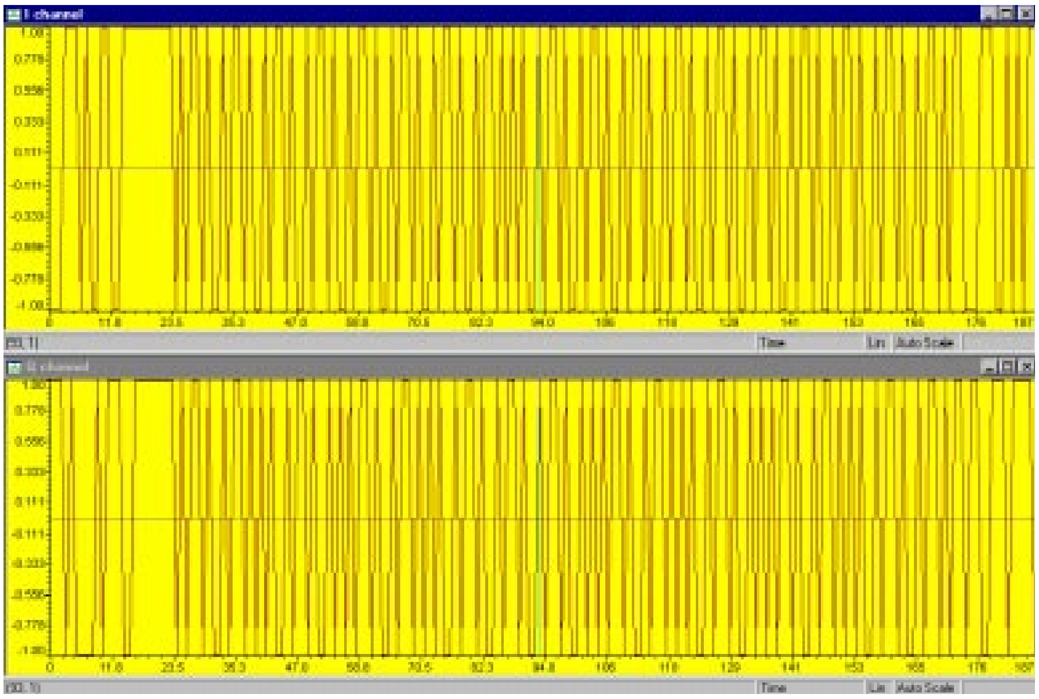


Figure 52: frame after root-raised-cosine filter

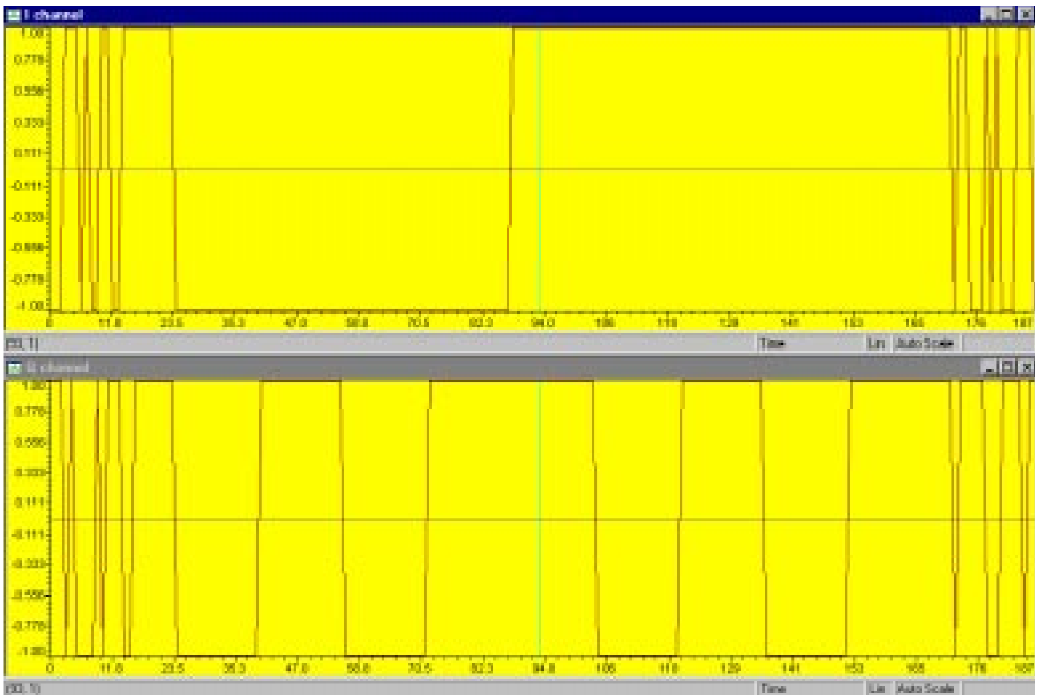


Figure 53: frame after CDMA

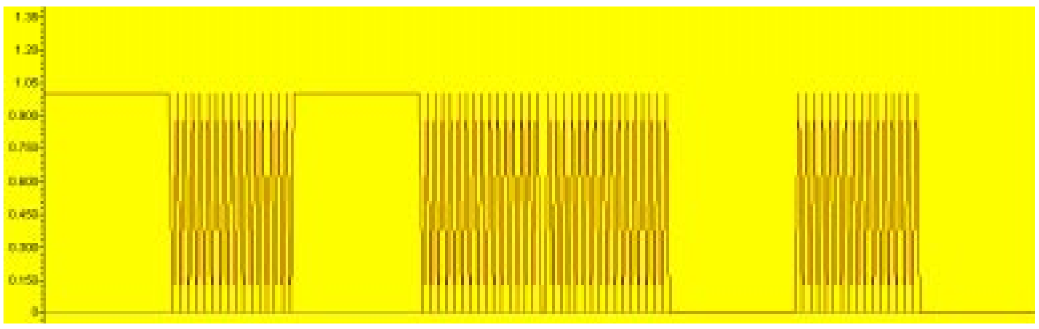


Figure 54: demodulated frame (only data), identical to the data part of Figure 45

5.4 *Performance of the software radio*

Another way to verify the built software-radio system is to measure the BER. This BER depends on the bit-energy-to-noise-energy-ratio and this relation is known from literature. So, in this section, the BER has been measured with different levels of noise. There are different ways to introduce noise in the system. In this experiment noise is added in the transmitter by software. Only noise is added to the data part of the frame. In this way the frame detection is unaffected by noise and only bit errors are counted which are caused by noise and not by wrong frame synchronization. Furthermore the analog output of the software-radio transmitter is directly connected to the analog input of the software-radio receiver (Figure 55). In this way, all noise other than the software-generated noise is eliminated.

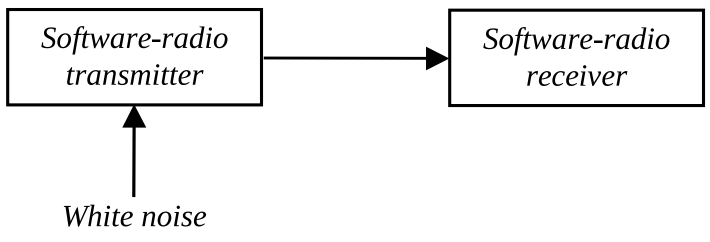


Figure 55: Set up for the BER versus bit-energy-to-noise-energy experiment

To measure the BER versus bit-energy-to-noise-energy relation, 3 variables are needed: bit energy (E_b), noise energy (N_0) and the accompanying BER. E_b can be captured in the following formula:

$$E_b = \tfrac{1}{2} A^2 T \qquad (3)$$

where A = amplitude and T = bit period

The noise energy, N_0, can be determined from the power spectrum. The noise floor of the power spectrum is equal to $N_0/2$. Finally the BER is measured in the software-radio receiver. Table 7 shows the BER at different noise levels. As expected, the BER increases when the E_b/N_0 ratio is decreasing. The received-bits column indicates the reliability of the BER. If more bits are received the BER is more reliable.

	E_b/N_0 (dB)	**BER**	**Received bits**
1	-0.7	0.02307291700	28400
2	-0.7	0.01847061300	38656
3	0.3	0.01779600000	96256
4	1.6	0.00543766840	617728
5	1.6	0.00547201190	245246
6	2.8	0.00192662610	320768
7	2.8	0.00214667780	255744
8	4.1	0.00040860355	122368
9	4.1	0.00048182820	616224
10	4.9	0.00045772895	225024
11	4.9	0.00046588303	251136

Table 7: BER versus bit-energy-to-noise-energy results

The BER for white noise can also be described by the following formula [19]:

$$BER = Q\left(\sqrt{2\left(E_b / N_0\right)}\right) \qquad (4)$$

where $Q(z) = \frac{1}{\sqrt{2\pi}} \int_z^{\infty} e^{-\lambda^2/2} \partial\lambda$ **(5)**

Figure 56 shows this equation in a doubly logarithmic plot. The BER decreases quickly when the bit-energy-to-noise-energy is increasing.

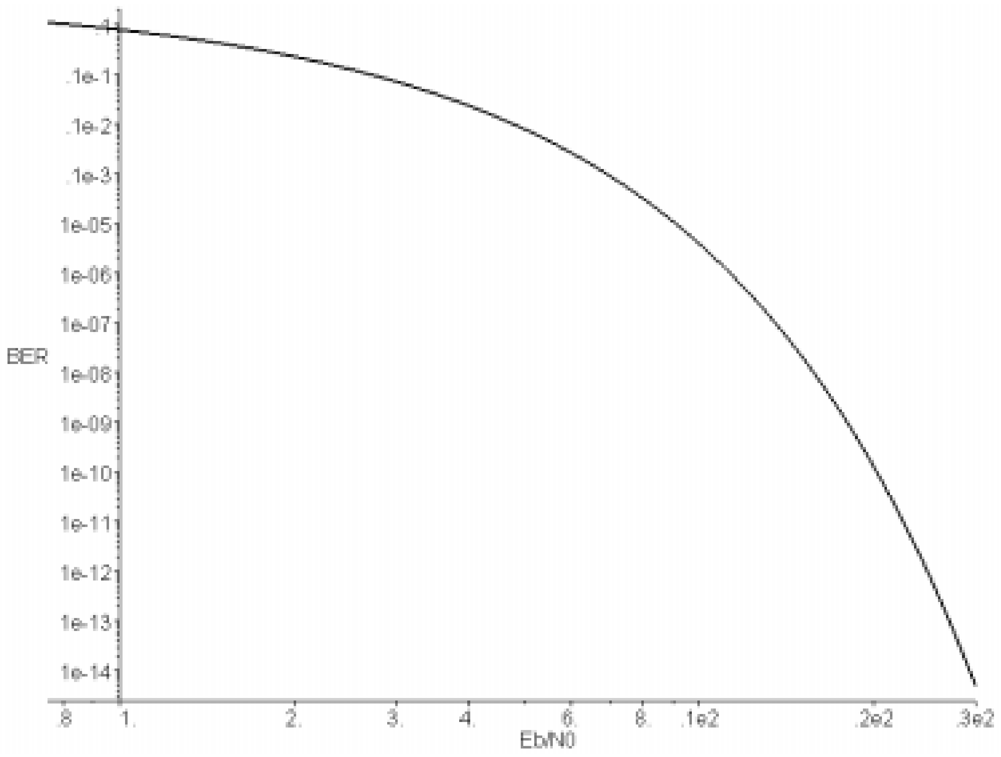


Figure 56: theoretical BER curve

The results of Table 7 are plotted in Figure 57. The black line, in Figure 57 (through the data points), is the best-fitted curve when assuming an exponential relation ($y = ae^{bx}$). The least-square method is used to determine this line. The theoretical line is also shown in Figure 57 (the uppermost line).

The theoretical and measured BER curves are different. They have almost the same shape, but the measured BER curve is much lower. The theoretical formula assumes a QPSK signal surrounded by white noise. The measured BER curve on the other hand has not been measured with white noise, because the noise has been generated by software. If noise is generated by software, only noise is added in the frequency band from zero to half the sample rate. On the other hand white noise has equal energy in all frequencies. Furthermore the sample rate of the software-radio is relatively low. So the generated noise cannot be considered as an approximation of white noise. In addition the receiver uses low-pass filters to extract the QPSK signal. Thus only noise in the pass band remains; the low-pass filters have eliminated other noise.

If the noise floor of the power spectrum is equal for white noise and generated noise, the noise energy of both noises are not equal. White noise has much more energy (in fact infinite energy) than the generated noise. The low-pass filters of the receiver also attenuate the energy of the generated noise. Therefore the BER is larger for white noise than the generated noise for equal bit-energy-to-noise-energy ratios. This effect can also been seen in Figure 57. Other methods for generating noise are difficult to implement. For example a noise generator which adds noise at the analog output of the software-radio transmitter also affects the synchronization word and thus the synchronization. Because synchronization errors do not count for the BER, the BER is difficult to measure.

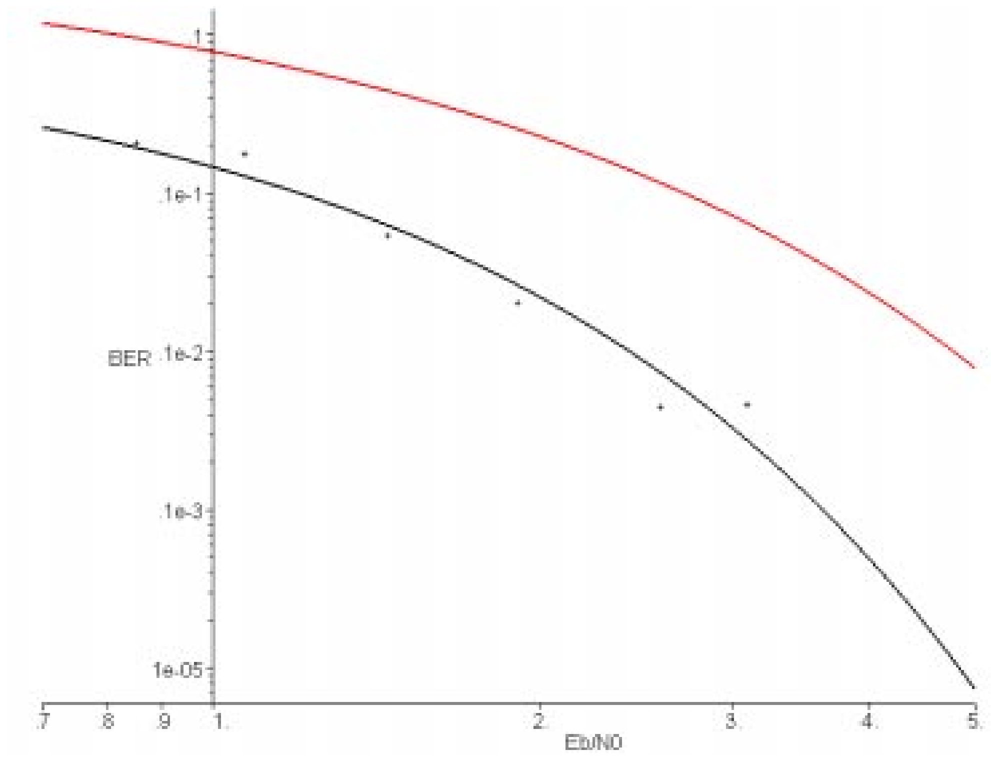


Figure 57: measured BER curve

CHAPTER 6

Final Review

6.1 Conclusions

In this project two major tasks are fulfilled. First a study of literature has been made about software radio and secondly a software-radio transmitter and receiver have been built for demonstration purposes. Besides these major tasks there are also minor ones. The programming environment Code Composer has been evaluated and the BER curve of the software radio has been measured.

The first task of this master's thesis was to make a study of literature about software radio. Software radio is a very large subject with many aspects. A global overview of the software-radio subject is given in this thesis; Some of the most important aspects of software radio such as the definition, the architecture, technological challenges, the front-end design and commercial aspects are described. The definition of software radio varies in literature. This paper uses the following definition:

Software radio is an emerging technology, aimed to build flexible radio systems, which are multiple-service, multi-standard, multi-band, re-configurable and re-programmable, by software.

The second task was to build a software-radio transmitter and receiver for demonstration purposes. Software radio requires both hardware and software. Two evaluation modules of Spectrum Digital are used as configurable hardware. One module is used as transmitter and the other is used as receiver. The modules are equipped with the TMS320LC549 DSP from Texas Instruments. This processor is capable of processing 100 million instructions per second. Because the evaluation module is designed for audio applications, the maximal sample frequency is limited to 64 kHz. Therefore the whole software-radio system uses a sample rate which is 5000 times lower than desired. Apart from the low sample rate, the evaluation module is equipped with 64-kb data RAM which is too small for a direct implementation of a mobile-system standard.

The software has been written as much as possible in C, only the initialization of the DSP has been programmed in assembly. Furthermore the software uses an optimized DSP function library of Texas Instruments for the implementation of the interpolation and decimation filters. This library is assembly-optimized and can be called from C. Other parts of the software radio are written fully in C and are not optimized for the used DSP. The software-radio transmitter requires about 75 million instruction cycles for encoding one frame. If the code is optimized the required amount of instructions is expected to be reduced significantly. The software-radio receiver requires about 80 million instruction cycles. Not all functions of the receiver have been (fully) implemented though. The automatic-gain-control block is missing and furthermore the software-radio receiver uses a very simple frame-detection method. So the total amount of required instruction cycles for the software-radio receiver is larger than 80 million.

The software radio uses a standard that is derived from CWTS. This Chinese standard uses a chip rate of 1.28 Mchips/s. Because the evaluation module is designed for audio applications, the software radio utilizes a chip rate of 250 chips/s. This chip rate is about 5000 times lower than CWTS uses.

Code Composer is used in this project as the programming environment. It is a useful tool, developed by Texas Instruments for programming DSPs. With Code Composer source files can be edited and projects can be built, debugged, profiled and managed from a single unified environment. Especially debugging with Code Composer is very powerful. Statistical data about program flow can be shown both numerically and graphically. Furthermore variables and (control) registers can be changed in run time, so the DSP can be fully controlled.

A complete radio system transmits and receives data through the air. In this project, only the IF stage of the software-radio system has been implemented. Therefore, to simulate a real radio system, the output of the software-radio transmitter is led through an FM transmitter (at 433 MHz). An FM receiver receives the signal and demodulates the signal back to an analog signal. This signal becomes after amplification the input of the software-radio receiver. With this setup the software-radio system has been tested successfully.

Another way to verify the software-radio system is to measure the BER. This BER depends on the bit-energy-to-noise-energy-ratio and this relation is known from literature. The BER curve has been measured and compared with the theoretical curve. They have almost the same shape, but the measured BER curve is much lower. The difference is caused by difference in noise. The theoretical BER curve assumes a QPSK signal surrounded by white noise. The generated noise on the other hand is not white but band-limited noise, because it is generated by software. Furthermore this generated noise is reduced at the receiver by the decimation filters. So, at equal noise floors, white noise has more energy than the generated noise. For that reason the measured BER curve is lower than the theoretical curve. Other methods for generating noise are difficult to implement, because the receiver uses a simple frame-detection method and synchronization errors do not count for the BER.

6.2 Recommendations

The built software radio is not complete. Only the IF stage of the software radio has been implemented. Therefore further research should focus on a complete implementation of a mobile-system standard in software. At this moment two major stages are missing, the RF and BB stage.

The IF stage is also not completely implemented. The built software-radio receiver uses a very simple frame-detection method, which can only be used in environments with little noise. A possible solution to this problem is the use of one threshold instead of two. Currently, two thresholds are used in the frame-detection block to ensure that the beginning of a frame is detected. If the beginning of the frame has a higher signal level, one threshold will also satisfy. This solution has the advantage that it is less sensitive to noise. Furthermore this detection method is simpler, so the size of the interrupt routine can be reduced.

Besides the frame-detection method, the automatic-gain-control block is not implemented in this project. This block is required when the received-signal power has a large range. In real radio environments this is the case. So further research should focus on implementing these two blocks, frame detection and automatic gain control.

At this moment the realization of a software-radio system, which operates at intended frequencies, is not possible. Both the transmitter and receiver would require about 5000 TMS320LC549 DSPs. However, the source code of the software-radio system has not been optimized. There are several possibilities that can reduce the source code and hardware requirements significantly. The software-radio system has been programmed in an old version of Code Composer. The new Code Composer Studio is more advanced and allows the evaluation of every procedure. In this way the most critical functions of the software-radio system can be determined and optimized. Furthermore the stages of the interpolation and decimation filters are not optimized; the filter requirements are met easily. Therefore the order and thus the number of coefficients can also be reduced. At this moment the interpolation and decimation block have been split up in pieces due to the lack of memory. Sufficient memory allows interpolation and decimation in one step, which is more efficient. These solutions should reduce the code size with about a factor 10 but this is still not enough for a software-radio implementation.

Several parts of the transmitter and receiver, such as multiplication by the carrier and the use of CIC filters are equal with every mobile-system standard. Therefore these parts could also be implemented in a FPGA. In addition FPGA are more power efficient. If these parts are implemented in an FPGA the requirements for the DSPs can be dropped again with a factor 10 because these parts are in the front-end of the transmitter/receiver. Thus if the code is optimized and the several parts are implemented in an FPGA the requirements for the DSPs are lowered with a factor 100. So 5000/100 = 50 TMS320LC549 DSPs are required. However, the TMS320LC549 DSP is not new. The newest DSP of Texas Instruments is about 25 times more powerful than the TMS320LC549 (for example the TMS320C64X-family). If the newest DSP is used in combination with FPGAs, only 2 or 3 DSPs are required for the implementation of a software-radio transmitter or receiver operating at intended frequencies.

Bibliography

References

[1] E-news Home Page*, Nieuwsbrief week 2.*
http://www.allcity.demon.nl/archief/week02.htm

[2] Cummings, M. and Haruyama*, S., "FPFA in the Software Radio,"* IEEE Communications Magazine*, pp. 108 – 112, February 1999.*

[3] Buracchini, E., *SORT & SWRADIO concept*
http://www.ifn.et.tu-dresden.de/~sort/

[4] Herbrig, H., Lundheim, L., Rossing, N. K., *SORT SW-Radio - From Concept Towards Demonstration*
http://www.ifn.et.tu-dresden.de/~sort/

[5] Mitola III, J., "The Software Radio Architecture," *IEEE Communications Magazine*, pp. 26 – 38, May 1995.

[6] Steele, R. and Hanzo, L., *Mobile Radio Communications second Edition*, John Wiley & Sons, London, 1999.

[7] Rappaport, T. S., *Wireless communications*, Prentice Hall, Upper Saddle River, 1996.

[8] Whipple, D. P., "The CDMA Standard," T.S. Rappaport, *Cellular Radio & Personal Communications*, IEEE, Piscataway, pp. 501 – 509, 1994.

[9] Frerking, M. E., *Digital signal processing in communication systems*, Kluwer Academic Publishers, Boston, 1993

[10] Mehrotra, A., *GSM System Engineering*, Artech house publishers, Boston, 1997

[11] Smit, J. and Snijders, H., *Collegedictaat VLSI System Design en VLSI Signal Processing*, Universiteit Twente, 1997.

[12] Bekkers, R. and Smits, J. *GSM in detail*, Kluwer/Segment, Beek, 1999.

[13] Kraemer, B., Chen, P., Damerow, D., Bacrania, K., *Advances in Semiconductor Technology - Enabling Software Radio*
http://www.infowin.org/ACTS/ANALYSYS/CONCERTATION/MOBILITY/swr.htm

[14] Intersil Corporation, *Datasheet Programmable Down Converter HSP50214B* http://www.intersil.com/data/fn/fn4/fn4266/index.asp

[15] Mitola III, J., "Technical challenges in the globalization of Software Radio," *IEEE Communications Magazine,* pp. 84 – 89, February 1999.

[16] Robinson, B., *Software radio: The standards perspective* http://www.infowin.org/ACTS/ANALYSYS/CONCERTATION/MOBILITY/swr.htm

[17] Tsurumi, H. and Suzuki, Y., " Broadband RF stage architecture for software-defined radio in handheld terminal applications," *IEEE Communications Magazine*, pp. 90 – 95, February 1999.

[18] Gunn, J. E., "A low-power DSP core-based software radio architecture," *IEEE Journal on selected areas in communications*, vol. 17, no. 4, pp. 574 – 589, April 1999.

[19] Couch, L.W., *Modern communication systems,* Prentice Hall, Englewood Cliffs, 1995

[20] Spectrum digital, *TMS320LC54x Evaluation Module: Technical Reference*, 503482-0001 Rev. E, 1998. http://www.spectrumdigital.com/technical/c54x_man.pdf

[21] Texas Instruments, *TMS320LC549, Digital Signal Processor Home Page*. http://www.ti.com/sc/docs/products/dsp/tms320vc549.html

[22] Texas Instruments, *TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals,* SPRU131F, 1999. http://www-s.ti.com/sc/psheets/spru131f/spru131f.pdf

[23] Texas Instruments, *TLC320AD55C Data Manual*, SLAS085, 1995. http://www-s.ti.com/sc/psheets/slas085/slas085.pdf

[24] Texas Instruments, *Code Composer Home Page*. http://dspvillage.ti.com/docs/tools/dsp/ccomposer/index.htm

[25] Yuezhen, W., *China Wireless Telecommunication Standard (CWTS); Working Group 1 (WG1); Spreading and modulation*, CWTS, 1999. http://www.cwts.org/tdd/R1_c104_v301.zip

[26] Yang, G., *China Wireless Telecommunication Standard (CWTS); Working Group 1 (WG1); Physical Channels and Mapping of Transport Channels onto Physical Channels*, CWTS, 1999. http://www.cwts.org/tdd/R1_c102_v300.zip

[27] Li, S., *China Wireless Telecommunication Standard (CWTS); Working Group 1 (WG1); Physical layer – General description*, CWTS, 1999. http://www.cwts.org/tdd/R1_c101_v300.zip

[28] Schwartz, S.M., *Frequency Hopping Spread Spectrum (FHSS) vs. Direct Sequence Spread Spectrum (DSSS) in the IEEE 802.11 Wireless Local Area Network arena,* BreezeCOM, 1997. http://www.breezecom.com/TechSupport/fhvsds.htm

[29] Yang, G. and Hu, J., *China Wireless Telecommunication Standard (CWTS); Working Group 1 (WG1); Base Station Conformance testing*, CWTS, 1999.

http://www.cwts.org/tdd/R1_c301_v100.zip

[30] Yang, G. and Hu, J, *China Wireless Telecommunication Standard (CWTS); Working Group 1 (WG1); Mobile Station Conformance Testing*, CWTS, 1999. http://www.cwts.org/tdd/R1_c302_v100.zip

[31] Li, S. *China Wireless Telecommunication Standard (CWTS); Working Group 1 (WG1); TD-SCDMA (UE); Radio Transmission and Reception*, CWTS, 1999. http://www.cwts.org/tdd/R1_c401_v300.zip

[32] Chhabra, A. and Iyer R.A., *A Practical Application of the TMS320C54x Host Port Interface (HPI),* Texas Instruments, 1999. http://www-s.ti.com/sc/psheets/spra574/spra574.pdf

[33] Roberts, R. A., and Mullis, C. T., *Digital signal processing*, Addison-Wesley Publishing Company, 1987.

[34] Kwentus, A.Y., Jiang, Z. and Willson, A.N., "Application of filter Sharpening to Cascaded Integrator-Comb Decimation Filters," *IEEE Transactions on Signal Processing*, vol. 45, no. 2, pp. 457 – 467, February 1997.

[35] Nauta, B. and Slump, C.H., *Development of a Software-Radio Based Embedded Mobile Terminal*, Universiteit Twente, 1999.

[36] DARPA MEMS Program http://darpa.mil

[37] Texas Instruments, *Optimized DSP Library for C Programmers*, SPRA480A, 2000. http://www-s.ti.com/sc/psheets/spra480a/spra480a.pdf

[38] Radiometrix Ltd, *UHF Radio Telemetry Transmit Module*, 1995. http://www.radiometrix.co.uk/dsheets/txm.pdf

[39] Radiometrix Ltd, *UHF Radio Telemetry Receive Module*, 1998. http://www.radiometrix.co.uk/dsheets/rxm.pdf

Consulted Works

Bose, V. G. and Shah, A. B., *Software Radios for Wireless Networking* http://www.vanu.com/publications/

Mitola III, J., *Software Radio Technology Challenges and Opportunities* http://www.infowin.org/ACTS/ANALYSYS/CONCERTATION/MOBILITY/swr.htm

Hentschel, T., Fettweis, G., Bronzel, M., *Channelization and sample rate adaptation in software radio terminals* http://www.ifn.et.tu-dresden.de/~sort/

Davis, R., Prabhu, V., *Architectures for Wideband CDMA Software Radios* http://infopad.eecs.berkeley.edu/~vanp/courses/cs252/

Tuttlebee, W. H. W., "Software radio technology: A European perspective," *IEEE Communications Magazine,* pp. 118 – 123, February 1999.

Patti, J. J., Husnay, M., Pintar, J., "A smart software radio: Concept development and demonstration," *IEEE Journal on selected areas in communications*, vol. 17, no. 4, pp. 631 – 649, April 1999.

Baines, R., "The DSP bottleneck", *IEEE Communications Magazine*, pp. 46 – 54, May 1999.

Chester, D. B., "Digital IF filter technology for 3G systems: an introduction," *IEEE Communications Magazine*, pp. 102 – 107, February 1999.

Internet

China Wireless Telecommunication Standard:
http://www.cwts.org

Texas Instruments:
http://www.ti.com

Spectrum Digital:
http://www.spectrumdigital.com/

Intersil Corporation
http://www.intersil.com/

Abbreviations

AC	alternating current
ACS	adjacent channel selectivity
A/D	analog/digital
AD/DA	analog/digital-digital/analog
ADC	analog digital converter
ADPCM	adaptive differential pulse-code modulation
AGC	analog gain control
ALU	arithmetic-logic unit
ampl	amplifier
API	application program interface
ASIC	application-specific integrated circuit
BB	base band
BER	bit error rate
BPF	band pass filter
bps	bits per second
CDMA	code-division multiple access
CELP	code excited linear predicted
codec	encoder/decoder
CSSU	compare, select, and store unit
CT2	British cordless telephone system
CWTS	China wireless telecommunication standard
D/A	digital/analog
DAC	digital analog converter
DAMPS	American digital advanced mobile phone system
dB	decibel
dBm	decibels relative to one milliwatt
DC	direct current
DCS-1800	GSM-like European system in the 1800 MHz band
DECT	digital European cordless telephone
DL	down link
Dpx	duplex
DQPSK	differential quadrature phase shift keying
DSP	digital signal processor
DwPTS	down-link pilot symbol
FDD	frequency-division duplex
FDMA	frequency-division multiple access
FEC	forward error control
FIR	finite-impulse response
FM	frequency modulation
FPGA	field-programmable gate array
FSK	frequency shift keying
GEL	general extension language
GFSK	Gaussian frequency shift keying
GMSK	Gaussian minimum shift keying
GP	guard period

GSM	global system for mobile computing
GUI	graphical user interface
HPI	host-port interface
I	in phase
IF	intermediate fre uency
ISI	inter-symbol interference
IP	intellectual property
IS-	American digital advanced mobile phone system (A S)
IS-9	American A system
JDC	apanese digital cellular
kBd	kilobaud
kbps	kilobits per second
kHz	kilohert
LCM	least common multiple
LNA	low noise amplifier
LO	local oscillator
LPF	low-pass filter
MCLK	master clock
MHz	megahert
MEMS	micro electro-mechanical systems
MIPS	million instructions per second
ms	millisecond
NCO	numerically-controlled oscillator
PACS	personal access communications system
PDC	programmable down converter
PHS	apanese personal handy phone system
PN code	pseudo-noise code
Q	uadrature phase
QPSK	uadrature phase shift keying
RAM	random-access memory
RF	radio fre uency
ROM	read-only memory
RPE-LTP	regular-pulse-exited long-term-predicted
SACCH	slow-associated control channel
SNR	signal-to-noise ratio
TACS	total-access communications system
TDD	time-division duplex
TDM	time-division multiplexed
TDMA	time-division multiple access
UL	up link
UpPTS	up-link pilot symbol
VCO	voltage-controlled oscillator
VSELP	vector-sum-excited linear predictive

. /

' ! ! %) 0 &(

- 89)
  + 2
- ', -. *01 + 2/"4
- F
- 89)
- 7
- = &
  + ' /+ '
- 89
  ', -. *01 + 2/"4
- 8 + 2
- *+,
-
- 8 #
- FD! E
  J2 >
-

' ! !

Figure 58: file hierarchy of the software-radio transmitter

cdma.c

```
/*********************************************************************/
/* Functions for implementing Code Division Multiple Access (CDMA)      */
/*                                                                    */
/* File:                   cdma.c                                     */
/* Version:                1.0                                        */
/* Last update:            28-04-2000                                 */
/* Creation date:          28-04-2000                                 */
/* Author:                 Roel Schiphorst                            */
/*                                                                    */
/*********************************************************************/

#include <tms320.h>

/* Function Prototypes */
void cdma(DATA*, DATA*, DATA*, unsigned int, unsigned int, unsigned int);

/*********************************************************************/
/*                                                                    */
/* Function: cdma()                                                   */
/*                                                                    */
/* This function multiplies the data part of a frame with a pseudo      */
/* noise (PN) code. The sync-part is not multiplied! Note: The input    */
/* vectors are overwritten with the new values                        */
/*                                                                    */
/* args:                                                              */
/*          I                                                          */
/*                 In-phase-component with "length" elements           */
/*                                                                    */
/*          Q                                                          */
/*                 Quadrature component with "length" elements         */
/*                                                                    */
/*          code                                                       */
/*                 Vector with 16 elements which contains the PN-code  */
/*                                                                    */
/*          spreading_factor                                           */
/*                 Indicates the spreading factor. If spreading_factor */
/*                 = 1 there is no spreading. If spreading_factor      */
/*                 = 16, maximal spreading occurs.                     */
/*                                                                    */
/*          offset                                                     */
/*                 receiver:          offset = 24 ((sync-word + 16)/2)*/
/*                 transmitter:  offset = 40 ((32 + sync-word + 12)/2)  */
/*                                                                    */
/*          length                                                    */
/*                 Number of elements of both input vectors           */
/*                                                                    */
/* Return Value :                                                     */
/*         NONE                                                       */
/*                                                                    */
/*********************************************************************/

void cdma(DATA *I, DATA *Q, DATA* code, unsigned int spreading_factor,
unsigned int offset, unsigned int length)
{
       int i, j = 0;

       for (i = offset; i < length; i++)

       {
              //Multiplication with the PN-code

              I[i] *= code[j];
              Q[i] *= code[j];

              j = (j + 1) % spreading_factor;  //update index of the PN-code
       }
}
```

diffenc

```
/**********************************************************************/
/*            IS54 Baseband Simulation Software                        */
/*                                                                     */
/*   File Description : QPSK Differential Encoder                            */
/*   File Name        : diffenc.c                                            */
/*   Date             : 20/04/00                                             */
/*                                                                           */
/*   Differential Encoder for QPSK                                           */
/*                                                                           */
/*   This encoder receives two bits from the data stream, and encodes them*/
/*   into QPSK modulation.  It outputs I and Q for each input symbol pair.*/
/*                                                                           */
/*   The input symbols and their bit relationship and phase changes          */
/*   are listed below:                                                       */
/*                                                                           */
/*   input data           phase changes                                      */
/*                                                                           */
/*   00                   -pi/2, -pi                                         */
/*   01                    pi/2, -pi                                         */
/*   10                   -pi/2,   0                                         */
/*   11                    pi/2,   0                                         */
/*                                                                           */
/*   Inputs :                                                                */
/*            packed : Pointer to array containing 384 symbol                */
/*                     bits to be encoded. Note that each element            */
/*                     of the array consists of a single bit (i.e.           */
/*                     a 0 or 1 only)                                        */
/*                                                                           */
/*    Outputs :                                                              */
/*            out_I   : Pointer to floating point array where                */
/*                      192 modulation vector I-components shall             */
/*                      be stored.                                           */
/*                                                                           */
/*            out_Q   : Pointer to floating point array where                */
/*                      192 modulation vector Q-components shall             */
/*                      be stored.                                           */
/*                                                                           */
/*    Return Value :                                                         */
/*            NONE                                                           */
/*                                                                           */
/****************************************************************************/

/* Include Files */

#include           <stdio.h>
#include           <stdlib.h>
#include           <math.h>
#include <tms320.h>

/* Defines */

/* Function Prototypes */

void diffenc( unsigned*, DATA*, DATA* );

/* External Function Prototypes */

/* Data */

/* External Data */

/* Code */


void diffenc( unsigned *packed, DATA *out_I, DATA *out_Q )
{

    int      i;
    int phase_i, phase_r;

    for (i = 0 ; i < 192 ; i++)
    {
        phase_i =  1;                   /* 0 == 1, 1 == -1*/
        phase_r =  1;                   /* assume both bits are zero     */

        if( *(packed++) != 0 )  phase_i = -1;    // 1st bit determines imag
                                                // phase
        if( *(packed++) != 0 )  phase_r = -1;    // 2nd bit determines real
                                                 // phase
        *(out_I++) = phase_i;
        *(out_Q++) = phase_r;
    }
    return;
}
```

frame.c

```
/*************************************************************************/
/* Functions for building up a frames                                    */
/*                                                                       */
/* File:                  frame.c                                        */
/* Version:               1.0                                            */
/* Last update:           01-05-2000                                     */
/* Creation date:         01-05-2000                                     */
/* Author:                Roel Schiphorst                                */
/*                                                                       */
/*************************************************************************/

/* Function Prototypes */
void build_frame_data(unsigned *, unsigned int);
void build_frame(unsigned *, unsigned *);

/*************************************************************************/
/*                                                                       */
/* Function: build_frame_data()                                          */
/*                                                                       */
/* This function creates the data-part of the frame. The output is      */
/* choosen in such a manner that verification of the received frame at  */
/* the receiver is easy.                                                 */
/*                                                                       */
/* args:                                                                 */
/*           slot                                                        */
/*                  256-bits of data                                     */
/*                                                                       */
/*           spreading_factor                                            */
/*                  Every symbol is repeated n-times. Each frame        */
/*                  consists 256/Spreading_factor number of data bits.  */
/*                                                                       */
/* Return Value :                                                        */
/*      NONE                                                             */
/*                                                                       */
/*************************************************************************/

void build_frame_data(unsigned *frame, unsigned int spreading_factor)
{
    int i, j;

       /* Build up data part of the frame */
       /* layout:   I channel: 1111 0000 1111 */
       /*           Q channel: 1010 1010 1010 */

    for(i = 0; i < 256; )
    {
             for(j = 0; j < spreading_factor; j++)   // repeat bit
                                                     // spreading_factor
                                                     // times
             {
                    frame[i]    = 1;    //even bit, I channel
                    i++;
                    frame[i]    = 1;    //odd bit, Q channel
                    i++;
             }

             for(j = 0; j < spreading_factor; j++)  // repeat bit
                                                    // spreading_factor
                                                    // times
             {
                    frame[i]    = 1;     //even bit, I channel
                    i++;
                    frame[i]    = 0;     //odd bit, Q channel
                    i++;
             }

             for(j = 0; j < spreading_factor; j++)  // repeat bit
                                                    // spreading_factor
```

```
                                                       // times
            {
                  frame[i]    = 1;     //even bit, I channel
                  i++;
                  frame[i]    = 1;     //odd bit, Q channel
                  i++;
            }

            for(j = 0; j < spreading_factor; j++)   // repeat bit
                                                    // spreading_factor
                                                    // times
            {
                  frame[i]    = 1;      //even bit, I channel
                  i++;
                  frame[i]    = 0;      //odd bit, Q channel
                  i++;
            }
            for(j = 0; j < spreading_factor; j++)   // repeat bit
                                                    // spreading_factor
                                                    // times
            {
                  frame[i]    = 0;      //even bit, I channel
                  i++;
                  frame[i]    = 1;      //odd bit, Q channel
                  i++;
            }
            for(j = 0; j < spreading_factor; j++)    // repeat bit
                                                     // spreading_factor
                                                     // times
            {
                  frame[i]    = 0;      //even bit, I channel
                  i++;
                  frame[i]    = 0;      //odd bit, Q channel
                  i++;
            }

            for(j = 0; j < spreading_factor; j++)     // repeat bit
                                                      // spreading_factor
                                                      // times
            {
                  frame[i]    = 0;     //even bit, I channel
                  i++;
                  frame[i]    = 1;     //odd bit, Q channel
                  i++;
            }

            for(j = 0; j < spreading_factor; j++)     // repeat bit
                                                      // spreading_factor
                                                      // times
            {
                  frame[i]    = 0;     //even bit, I channel
                  i++;
                  frame[i]    = 0;     //odd bit, Q channel
                  i++;
            }
    }

    return;
}

/*********************************************************************/
/*                                                                   */
/* Function: build_frame()                                           */
/*                                                                   */
/* This routine constructs a 384-bit formatted slot from 256-bit slot   */
/* data according to the following:                                  */
/*                                                                   */
/*                 16 bits  : 0, guard period                        */
/*                 32 bits  : Sync word                              */
/*                 16 bits  : 0, guad period                         */
/*                 128 bits : 1st 128 bits of slot data              */
/*                 48 bits  : 0                                      */
/*                 128 bits : 2nd 128 bits of slot data              */
/*                 16 bits  : 0                                      */
/*                                                                   */
/* args:                                                             */
/*    data                                                            */
/*           pointer to 256-element bit array representing the data bits*/
/*          the data bits for the slot. (Note that all elements should  */
/*          elements should be binary, i.e. 0's and 1's only).          */
/*                                                                      */
/*      frame                                                           */
/*              pointer to 384-element bit array where the resultant    */
/*          formatted frame shall be stored.                            */
/*                                                                      */
/* Return Value :                                                       */
/*      NONE                                                            */
/*                                                                      */
/************************************************************************/

void build_frame(unsigned *data, unsigned *frame)
{
    int         i;
    unsigned    sync1[32]= {1,0,1,0,1,0,0,1,0,0,0,1,1,1,0,1,1,1,1,0,
                            0,1,0,0,1,0,1,0,0,1,0,1};       // sync word

    for( i = 0; i < 32;  i++ )  *(frame++) = 0;             // guard period
    for( i = 0; i < 32;  i++ )  *(frame++) = sync1[i];      // sync word
    for( i = 0; i < 16;  i++ )  *(frame++) = 0;             // guard period
    for( i = 0; i < 128; i++ )  *(frame++) = *(data++);     // 1st data
                                                            // block
    for( i = 0; i < 32;  i++ )  *(frame++) = 0;          // zeros
    for( i = 0; i < 128; i++ )  *(frame++) = *(data++);  // 2nd data block
    for( i = 0; i < 16;  i++ )  *(frame++) = 0;          // guard period
    return;

}
```

main.c

```
/************************************************************************/
/* This file contains the main loop of the QPSK transmitter. Besides the*/
/* main loop it contains functions for initilization of the DSP and the */
/* AD/DA chip.                                                          */
/*                                                                      */
/* File:                  main.c                                        */
/* Version:               1.0                                           */
/* Last update:           02-05-2000                                    */
/* Creation date:         02-05-2000                                    */
/* Author:                Roel Schiphorst                               */
/*                                                                      */
/************************************************************************/

#include <tms320.h>
#include <dsplib.h>

#include "reg549.h"
#include "table.h"
#include "filter_int.h"

/* Function Prototypes */
void main(void);

volatile int p0_serialflag;     // 0 = normal operation 1 = programming
                                // AD/DA chip
volatile int p0_serialint;      // 1 = serial port receive interrupt has
                                // occured
volatile int p0_ready;          // 1 = frame has been sent

int output[frame_length];       // contains the data that is being send to
                                // the AD/DA chip

DATA  I_RRC[728];                // I channel after root raised cosine filter

DATA  Q_RRC[728];                // Q channel after root raised cosine filter

unsigned int tx_data[256];       // array of data bits that are transmitted
unsigned int tx_frame[384];      // total frame (data + sync word)

DATA out_I[192];                        // QPSK modulated I channel
DATA out_Q[192];                        // QPSK modulated Q channel
```

```
float tx_filt[48];  // array which contains the root raised cosine table:
                    // the input is interpolated 4 times. Therefore 3/4 of
                    // the samples are zero. The filter table is so
                    // rearranged that the first 12 elements contain the
                    // filter coeffs for the first output value, the next 12
                    // for the second output, the next 12, for the third
                    // interpolated output and the last 12 filter coefs
                    // values for the last interpolated output value.

DATA tx_filt_data[48];     // same as tx_filt, only the data is converted
                           // from float to DATA

DATA buffer[8];            // contains a small piece of the I_RRC/Q_RRC
                           // array, used for interpolation process

DATA out16[16];            // output first interpolation filter
DATA out32[32];            // output second interpolation filter
DATA out64[64];            // output third interpolation filter
DATA out128[128];          // output fourth interpolation filter

float const1_1 = 1.1;             // constant value
unsigned int shift_value = 15;

unsigned int spreading_factor = 16;
       // defines the spreading factor for the frame

/*
DATA code1[16]= { 1,1,1,1, 1,1,1,1, 1,1,1,1, 1,1,1,1 };
DATA code2[16]= { 1,1,1,1, 1,1,1,1, -1,-1,-1,-1, -1,-1,-1,-1 };
DATA code3[16]= { 1,1,1,1, -1,-1,-1,-1, 1,1,1,1, -1,-1,-1,-1 };
DATA code4[16]= { 1,1,1,1, -1,-1,-1,-1, -1,-1,-1,-1, 1,1,1,1 };
DATA code5[16]= { 1,1,-1,-1, 1,1,-1,-1, 1,1,-1,-1, 1,1,-1,-1 };
DATA code6[16]= { 1,1,-1,-1, 1,1,-1,-1, -1,-1,1,1, -1,-1,1,1 };
DATA code7[16]= { 1,1,-1,-1, -1,-1,1,1, 1,1,-1,-1, -1,-1,1,1 };
DATA code8[16]= { 1,1,-1,-1, -1,-1,1,1, -1,-1,1,1, 1,1,-1,-1 };
DATA code9[16]=  { 1,-1,1,-1, 1,-1,1,-1, 1,-1,1,-1, 1,-1,1,-1 };
DATA code10[16]= { 1,-1,1,-1, 1,-1,1,-1, -1,1,-1,1, -1,1,-1,1 };
DATA code11[16]= { 1,-1,1,-1, -1,1,-1,1, 1,-1,1,-1, -1,1,-1,1 };
DATA code12[16]= { 1,-1,1,-1, -1,1,-1,1, -1,1,-1,1, 1,-1,1,-1 };
DATA code13[16]= { 1,-1,-1,1, 1,-1,-1,1, 1,-1,-1,1, 1,-1,-1,1 };
DATA code14[16]= { 1,-1,-1,1, 1,-1,-1,1, -1,1,1,-1, -1,1,1,-1 };
DATA code15[16]= { 1,-1,-1,1, -1,1,1,-1, 1,-1,-1,1, -1,1,1,-1 };
DATA code16[16]= { 1,-1,-1,1, -1,1,1,-1, -1,1,1,-1, 1,-1,-1,1 };
*/
// There are 16 different CDMA codes, listed above. One code is choosen for
// the communication and is equal to the transmitter CDMA code
DATA code[16]= { 1,-1,1,-1, -1,1,-1,1, 1,-1,1,-1, -1,1,-1,1 };

/*************************************************************************/
/*************************************************************************/

void inline disable() {
       asm("  ssbx INTM");             // Disable all interrupts of the DSP
}

void inline enable() {
       asm ("  rsbx INTM");            // Enable all interrupts of the DSP
}

/***********************************************************************/
/*                                                                     */
/* Function : waitintr()                                               */
/*                                                                     */
/* This function is used by the progreg()-function. It will only exit */
/* the function if the serial port has received a new word.            */
/*                                                                     */
/* args:                                                               */
/*          NONE                                                       */
/*                                                                     */
/* Return Value :                                                      */
/*     NONE                                                            */
/*                                                                     */
/***********************************************************************/

void waitintr(void)
{
       while (p0_serialint == 0); // Wait until data is received from the
```

```
                                  // AD-DA chip

       p0_serialint = 0;                 // Reset p0_serialint
       return;
}

/**************************************************************************/
/*                                                                        */
/* Function : progreg()                                                   */
/*                                                                        */
/* This function programs a register of the AD/DA chip via the serial    */
/* port                                                                   */
/*                                                                        */
/* args:                                                                  */
/*          progword                                                      */
/*                 a 16 bit word which contains besides the address of */
/*                 the register, also the new contents of the register */
/*                 See the datasheet of the AD/DA-chip for more         */
/*                 information                                            */
/*                                                                        */
/* Return Value :                                                         */
/*    NONE                                                               */
/*                                                                        */
/**************************************************************************/

void progreg(int progword)
{
       *BDXR0  = 0x0001;                 // Request secondary communication
                                         // (read of write to registers)
       waitintr();                       // Wait for serial port receive
                                         // interrupt

       *BDXR0  = progword;               // Send configuration word to AD-DA
                                         // chip

       waitintr();                       // Wait for serial port receive
                                         // interrupt
       return;
}

/**************************************************************************/
/*                                                                        */
/* Function : init()                                                      */
/*                                                                        */
/* This function initializes the DSP and the AD/DA-chip                  */
/*                                                                        */
/* args:                                                                  */
/*          NONE                                                          */
/*                                                                        */
/* Return Value :                                                         */
/*    NONE                                                                */
/*                                                                        */
/**************************************************************************/

void init()
{
       *SWWSR = 0x1209;               // Set wait states for memory
        *PMST = 0x0ffc0;              // Set location of interrupt table
       disable();                     // Disable all interrupts
       *BSPC0 = 0x0000;               // Initialize SPC
       *BSPC0 = *BSPC0 | 0x00C0;      // Initialize SPC
       *IMR  = 0x0010;            // Enable serial port 0 receive interupt
       *IFR  = 0xffff;            // Clear any pending interrupts

       enable();                  // Enable all interrupts

       p0_serialflag = 1;         // Indicated whether the AD-DA chip is
                                  // configured (1)
                                  // or normal use (0)
       waitintr();

       progreg(0x0100);           // 01 = control 1 register, value 00
                                  // (default)

       progreg(0x0200);           // 02 = control 2 register, value 00
                                  // (default)
                                  // Decimator and interpolator filters are
```

```
                                 // disabled
        progreg(0x0304);         // 03 = Fk divide register, value 10
                                 // controls filter clock rate and sample
                                 // period
        progreg(0x0404);         // 04 = Fsclk divide register, value 10
                                 // controls the shift (data) clock rate
        progreg(0x0500);         // 05 = control 3 register, value 00
                                 // DAC reference enabled
        *BDXR0 = 0x0000;         // Send zero to end configuration

        p0_serialflag = 0;       // Configuration is finished.

        *IMR  = *IMR & 0xffef;   // disable serial port receive interrupts

}

/**************************************************************************/
/*                                                                        */
/* Function : codrx()                                                     */
/*                                                                        */
/* This function is called when a serial port receive interrupt occurs.  */
/*                                                                        */
/* args:                                                                  */
/*          NONE                                                          */
/*                                                                        */
/* Return Value :                                                         */
/*    NONE                                                                */
/*                                                                        */
/**************************************************************************/
void interrupt codrx()
{
        static int index_RX = 0;
        p0_serialint = 1;              // Set the interrupt variable to 1
        *AL = *BDRR0;                  // Read data (and trash it)

        if (p0_serialflag == 0)        // If 'normal' operation (no
                                       // programming of regs)
        {
                *BDXR0 = output[index_RX]; // Send data to serial port 0

                if (index_RX == frame_length_min_1)
                // If last sample is transmitted then disable serial port
                // communication
                {
                    *IMR  = 0x0000;    // Disable serial port interrupt
                    *BDXR0 = 0x0000;   // Send a zero to clear the transmit
                                       // buffer
                       p0_ready = 1;   // frames is sent, set var to 1
                       index_RX = -1;  // reset index var
                }

                index_RX++;
        }
}

/**************************************************************************/
/*                                                                        */
/* Function : main()                                                      */
/*                                                                        */
/* Main loop of the QPSK transmitter.                                     */
/*                                                                        */
/* args:                                                                  */
/*          NONE                                                          */
/*                                                                        */
/* Return Value :                                                         */
/*          NONE                                                          */
/*                                                                        */
/**************************************************************************/

void main(void)
{
   int i = 0, j = 0, k = 0;
   DATA noise;

   LDATA sample_data;      // temp value used for multiplication with carrier
   DATA sample_data2;      // temp value used for multiplication with carrier
   int interpolation_factor_x_8 = interpolation_factor*8;
   p0_serialint = 0;   // Variable used to recognize a serial port
                       // receive interrupt
   p0_ready = 1;
   init();          // Initialization of the DSP and AD/DA chip

   rand16init();           // initialize random generator, used to create
                           // noise

   k = 0;
   for ( i = 3; i >= 0; i-- )
   {
        for( j = i; j < 48; j += 4 )
        {
            tx_filt[k++] = tableRaisCosFilt[j]/const1_1;
                           // Read root raised cosine filter coefficients
                           // and convert them to 4 filter banks
        }                  // data rate = 4*symbol rate
   }
   /* convert filter coeffs to DATA-type */
   fltoq15(tx_filt, tx_filt_data, 48);

   build_frame_data(tx_data, spreading_factor);
             // create data bits for the frame

   build_frame(tx_data, tx_frame);
             // build frame (sync word + data)

   mod_QPSK(tx_frame, out_I, out_Q);
             // QPSK modulation

   cdma(out_I, out_Q, code, spreading_factor, 40, 192);
             // multiply with PN-code

   tx_RRC_filt(out_I, out_Q, 182, tx_filt_data, I_RRC, Q_RRC);
             // Root raised cosine filter (4x oversampling)

    while (1)
    {
/***************************************************************************/
/*           Interpolation                                                  */
/***************************************************************************/
        /* I channel*/
      for (j=0; j< NH16; j++) dbuffer16[j] = 0;
             // clear delay buffer (a must)
      for (j=0; j< NH32; j++) dbuffer32[j] = 0;
             // clear delay buffer (a must)
      for (j=0; j< NH64; j++) dbuffer64[j] = 0;
             // clear delay buffer (a must)
      for (j=0; j< NH128; j++) dbuffer128[j] = 0;
             // clear delay buffer (a must)

      for (i = 0; i < 91; i++ )
             // Interpolate only a part of the I_RRC array a time
      {      // interpolation occurs in 91 steps
             for (j = 0; j < 8; j++)
             {
                   buffer[j] = I_RRC[i*8 + j];
                   // use next 8 samples for interpolation
             }

      /* 4-stage Interpolation FIR filter, interpolation factor = 16*/
             firinterp(buffer, h16, out16,  &dp16,  NH16,   8, 2);
             firinterp(out16, h32,  out32,  &dp32,  NH32,  16, 2);
             firinterp(out32, h64,  out64,  &dp64,  NH64,  32, 2);
             firinterp(out64, h128, out128, &dp128, NH128, 64, 2);

             /* multiply with carrier and store result in output array*/
             k = 0;
             for (j = 0; j < interpolation_factor*8; j++)
             {
                   sample_data = out128[j];

                   sample_data *= table_f_cos[k];   // multiply with cosine
                   sample_data >>= shift_value;      // scale
                   output[i*interpolation_factor_x_8 + j] = sample_data;
                         // store in output array
```

```
                k = (k + 8) % 32;
                      // update index to cosine table
          }
    }

/**************************************************************************/
/*          Interpolation                                                 */
/**************************************************************************/
       /* Q channel*/
    for (j=0; j< NH16; j++) dbuffer16[j] = 0;
          // clear delay buffer (a must)
    for (j=0; j< NH32; j++) dbuffer32[j] = 0;
          // clear delay buffer (a must)
    for (j=0; j< NH64; j++) dbuffer64[j] = 0;
          // clear delay buffer (a must)
    for (j=0; j< NH128; j++) dbuffer128[j] = 0;
          // clear delay buffer (a must)

    for (i = 0; i < 91; i++)
          // Interpolate only a part of the Q_RRC array a time
    {     // interpolation occurs in 91 steps
          for (j = 0; j < 8; j++)
          {
                buffer[j] = Q_RRC[i*8 + j];
                      // use next 8 samples for interpolation
          }

    /* 4-stage Interpolation FIR filter, interpolation factor = 16*/
          firinterp(buffer, h16, out16,  &dp16,  NH16,   8, 2);
          firinterp(out16, h32,  out32,  &dp32,  NH32,  16, 2);
          firinterp(out32, h64,  out64,  &dp64,  NH64,  32, 2);
          firinterp(out64, h128, out128, &dp128, NH128, 64, 2);

    /* multiply with carrier and add result in output array*/

          k = 0;
          for (j = 0; j < interpolation_factor*8; j++)
          {
                sample_data = out128[j];

                sample_data *= table_f_sin[k];    // multiply with sine
                sample_data >>= shift_value;      // scale
                sample_data2 = output[i*128 + j];
                      // save current output value (I channel)
                sample_data += sample_data2;
                      // add current output value
                sample_data &= 0xfffffffe;
                      // reset lsb of word, required for AD/DA chip

                output[i*interpolation_factor_x_8 + j] = sample_data;
                      // Store output value
                k = (k + 8) % 32;
                      // update index to sine table
          }
    }
/************************************************************************/
/* add noise                                                            */
/************************************************************************/

    for (i = 1150; i < frame_length; i++) // leave sync word unaffected
    {
          rand16(&noise,1);
          noise >>= 3;
          output[i] >>= 2;
          output[i] += noise;
          output[i] &= 0xfffffffe;
                // reset lsb of word, required for AD/DA chip
    }
/************************************************************************/
/* end noise part                                                       */
/************************************************************************/

/**************************************************************************/
/*          End of interpolation                                          */
/**************************************************************************/
```

```
        p0_ready = 0;       // reset var
        *IMR  |= 0x0010;    // enable serial port receive interrupt

        // frame has been calculated and is ready to send

        build_frame_data(tx_data, spreading_factor);
              // create data bits for the frame

        build_frame(tx_data, tx_frame);
              // build frame (sync word + data)

        mod_QPSK(tx_frame, out_I, out_Q);       // QPSK modulation

        cdma(out_I, out_Q, code, spreading_factor, 40, 192);
              // multiply with PN-code

        tx_RRC_filt(out_I, out_Q, 182, tx_filt_data, I_RRC, Q_RRC);
              // Root raised cosine filter (4x oversampling)



        while (p0_ready == 0);
        // only proceed to next step if frame has been sent
    }
}
```

mod_QPSK.c

```
/*************************************************************************/
/* Functions for modulating a QPSK signal                                */
/*                                                                       */
/* File:                 mod_QPSK.c                                      */
/* Version:              1.0                                             */
/* Last update:          01-05-2000                                      */
/* Creation date:        01-05-2000                                      */
/* Author:               Roel Schiphorst                                 */
/*                                                                       */
/*************************************************************************/

#include <tms320.h>

/* Function Prototypes */
void mod_QPSK( unsigned*, DATA*, DATA* );
void tx_RRC_filt(DATA*, DATA*, int, DATA*, DATA*, DATA*);

/*************************************************************************/
/*                                                                       */
/* Function: mod_QPSK()                                                  */
/*                                                                       */
/* This function modulates frame, using QPSK modulation. The output      */
/* are two arrays (I channel and Q channel).                             */
/*                                                                       */
/* args:                                                                 */
/*          data                                                         */
/*                384-bits frame (sync word + data)                      */
/*                                                                       */
/*          out_I                                                        */
/*                QPSK modulated I channel with 192 elements             */
/*                                                                       */
/*          out_Q                                                        */
/*                QPSK modulated Q channel with 192 elements             */
/*                                                                       */
/* Return Value :                                                        */
/*     NONE                                                              */
/*                                                                       */
/*************************************************************************/

void mod_QPSK(unsigned *data, DATA *out_I, DATA *out_Q )
{

    int     i;
    int phase_i, phase_r;

    for (i = 0 ; i < 192 ; i++)
    {
        phase_i =  1;                   /* 0 == 1, 1 == -1*/
        phase_r =  1;                   /* assume both bits are zero */

        if( *(data++) != 0 )  phase_i = -1;
             /* 1st bit determines imag phase */
        if( *(data++) != 0 )  phase_r = -1;
             /* 2nd bit determines real phase */
        *(out_I++) = phase_i;           // store result into arrays
        *(out_Q++) = phase_r;
    }
    return;
}

/*************************************************************************/
/*                                                                       */
/* Function: tx_RRC_filt()                                               */
/*                                                                       */
/* This filters the QPSK signal with a root raised cosine filter.        */
/* This reduces the ISI (Inter Symbol Interference). The same filter is */
/* used at the receiver.                                                 */
/*                                                                       */
/* args:                                                                 */
/*           I                                                           */
/*                  input vector of the in-phase channel                 */
/*                                                                       */
/*           Q                                                           */
/*                  input vector of the quadrature channel               */
/*                                                                       */
/*           num_of_syms                                                 */
/*                  lenght of I (and Q) - 11 (filter length -1)          */
/*                                                                       */
/*           filt                                                        */
/*                  pointer to 48-taps-root-raised-cosine-filter vector  */
/*                                                                       */
/*           I_SRC                                                       */
/*                  output vector after Root Raised Cosine filter        */
/*                  length = 4 * num_of_syms                             */
/*                                                                       */
/*           Q_SRC                                                       */
/*                  output vector after Root Raised Cosine filter        */
/*                  length = 4 * num_of_syms                             */
/*                                                                       */
/* Return Value :                                                        */
/*           NONE                                                        */
/*                                                                       */
/*************************************************************************/

void tx_RRC_filt(DATA *I, DATA *Q, int num_of_syms, DATA *filt, DATA *I_SRC,
                 DATA *Q_SRC)
{
    int     i,j,k;
    long int i_temp, q_temp; // temp value for saving filter output value
    DATA     *iptr, *qptr;  // pointer to input data

    /* Set up Pointers to input data */
    iptr = I;
    qptr = Q;

  /* Main Filter Loop */
    for (i = 0 ; i < num_of_syms ; i++)
    {
        for (j = 0 ; j < 4 ; j++)  /* 4-bank filtering, 12 taps a piece */
        {
          /* Clear Filter Sums */
            i_temp = 0;
            q_temp = 0;

          /* Filter Input Data */
            for (k = 0 ; k < 12 ; k++)
            {
                i_temp += (*(iptr++)) * (*filt);
                q_temp += (*(qptr++)) * (*(filt++));
            }

          /* Store Filter Output */
            *(I_SRC++) = i_temp >> 2;
            *(Q_SRC++) = q_temp >> 2;
          /* Use Same Input Data Next Time, but different filter bank */
            iptr -= 12;
            qptr -= 12;
        }

        iptr++;
        qptr++;         /* Proceed to next symbol */

        filt -= 48;     /* Set Filter Pointer back to Start */
    }
    return;
}
```

reg549.h

```
/**************************************************************************/
/*                                                                        */
/* Memory-mapped registers for TMS320C549 DSP                             */
/*                                                                        */
/* File:            reg549.h                                              */
/* Version:         1.0                                                   */
/* Last update:     01-05-2000                                            */
/* Creation date:   01-05-2000                                            */
/* Author:          Roel Schiphorst                                       */
/*                                                                        */
/**************************************************************************/
#ifndef _reg549
#define _reg549
/* Interrupt mask register */
volatile unsigned int* IMR = (volatile unsigned int *) 0x0000;

/* Interrupt flag register */
volatile unsigned int* IFR = (volatile unsigned int *) 0x0001;

/* status register 0*/
volatile unsigned int* ST0 = (volatile unsigned int *) 0x0006;

/* status register 1*/
volatile unsigned int* ST1 = (volatile unsigned int *) 0x0007;

/* Accumulator A low word*/
volatile unsigned int* AL = (volatile unsigned int *) 0x0008;

/* Accumulator A high word*/
volatile unsigned int* AH = (volatile unsigned int *) 0x0009;

/* Accumulator A guard bits*/
volatile unsigned int* AG = (volatile unsigned int *) 0x000A;

/* Accumulator B low word*/
volatile unsigned int* BL = (volatile unsigned int *) 0x000B;

/* Accumulator B high word*/
volatile unsigned int* BH = (volatile unsigned int *) 0x000C;

/* Accumulator B guard bits*/
volatile unsigned int* BG = (volatile unsigned int *) 0x000D;

/* Temporary register*/
volatile unsigned int* T = (volatile unsigned int *) 0x000E;

/* Transistion register*/
volatile unsigned int* TRN = (volatile unsigned int *) 0x000F;

/* Auxiliiary register 0*/
volatile unsigned int* AR0 = (volatile unsigned int *) 0x0010;

/* Auxiliiary register 1*/
volatile unsigned int* AR1 = (volatile unsigned int *) 0x0011;

/* Auxiliiary register 2*/
volatile unsigned int* AR2 = (volatile unsigned int *) 0x0012;

/* Auxiliiary register 3*/
```

```
volatile unsigned int* AR3 = (volatile unsigned int *) 0x0013;

/* Auxiliiary register 4*/
volatile unsigned int* AR4 = (volatile unsigned int *) 0x0014;

/* Auxiliiary register 5*/
volatile unsigned int* AR5 = (volatile unsigned int *) 0x0015;

/* Auxiliiary register 6*/
volatile unsigned int* AR6 = (volatile unsigned int *) 0x0016;

/* Auxiliiary register 7*/
volatile unsigned int* AR7 = (volatile unsigned int *) 0x0017;

/* Stack pointer*/
volatile unsigned int* SP = (volatile unsigned int *) 0x0018;

/* Circular-buffer size register*/
volatile unsigned int* BK = (volatile unsigned int *) 0x0019;

/* Block-repeat counter*/
volatile unsigned int* BRC = (volatile unsigned int *) 0x001A;

/* Block-repeat start address*/
volatile unsigned int* RSA = (volatile unsigned int *) 0x001B;

/* Block-repeat end address*/
volatile unsigned int* REA = (volatile unsigned int *) 0x001C;

/* Processor mode status register*/
volatile unsigned int* PMST = (volatile unsigned int *) 0x001D;

/* Program counter extension register*/
volatile unsigned int* XPC = (volatile unsigned int *) 0x001E;

/* Synchronous buffered serial port registers */
volatile unsigned int* BDRR0 = (volatile unsigned int *) 0x0020;
volatile unsigned int* BDXR0 = (volatile unsigned int *) 0x0021;
volatile unsigned int* BSPC0 = (volatile unsigned int *) 0x0022;

/* timer registers */
volatile unsigned int* TIM = (volatile unsigned int *) 0x0024;
volatile unsigned int* PRD = (volatile unsigned int *) 0x0025;
volatile unsigned int* TCR = (volatile unsigned int *) 0x0026;

/* various */
volatile unsigned int* SWWSR = (volatile unsigned int *) 0x0028;
volatile unsigned int* BSCR = (volatile unsigned int *) 0x0029;
volatile unsigned int* XWSR = (volatile unsigned int *) 0x002B;
volatile unsigned int* HPIC = (volatile unsigned int *) 0x002C;

/* TDM serial port registers */
volatile unsigned int* TRCV = (volatile unsigned int *) 0x0030;
volatile unsigned int* TDXR = (volatile unsigned int *) 0x0031;
volatile unsigned int* TDPC = (volatile unsigned int *) 0x0032;
volatile unsigned int* TCSR = (volatile unsigned int *) 0x0033;
volatile unsigned int* TRTA = (volatile unsigned int *) 0x0034;
volatile unsigned int* TRAD = (volatile unsigned int *) 0x0035;

/* ABU (automatic buffering unit registers */
volatile unsigned int* AXR0 = (volatile unsigned int *) 0x0038;
volatile unsigned int* BKX0 = (volatile unsigned int *) 0x0039;
volatile unsigned int* ARR0 = (volatile unsigned int *) 0x003A;
volatile unsigned int* BKR0 = (volatile unsigned int *) 0x003B;
volatile unsigned int* AXR1 = (volatile unsigned int *) 0x003C;
volatile unsigned int* BKX1 = (volatile unsigned int *) 0x003D;
volatile unsigned int* ARR1 = (volatile unsigned int *) 0x003E;

/* serial port 1 */
volatile unsigned int* BDRR1 = (volatile unsigned int *) 0x0040;
volatile unsigned int* BDXR1 = (volatile unsigned int *) 0x0041;
volatile unsigned int* BSPC1 = (volatile unsigned int *) 0x0042;
volatile unsigned int* BSPCE1 = (volatile unsigned int *) 0x0043;

/* clock mode register */
volatile unsigned int* CLKMD = (volatile unsigned int *) 0x0058;
```

```
#endif
```

table.h

```
/*********************************************************************/
/*                                                                   */
/* Contains tables for the root-raised-cosine filter (48 elements) and  */
/* a sine and cosine table of 32 elements.                           */
/*                                                                   */
/* File:           table.h                                           */
/* Version:        1.0                                               */
/* Last update:    02-05-2000                                        */
/* Creation date:  02-05-2000                                        */
/* Author:         Roel Schiphorst                                   */
/*                                                                   */
/*********************************************************************/

#pragma DATA_SECTION (tableRaisCosFilt,".rcf")
       //store coeffs in a special defined memory region

float tableRaisCosFilt[] = { /* Sum  = 4 (interpolation factor = 4) */
-0.005844,
-0.004486,
0.002716,
0.009177,
0.007490,
-0.002346,
-0.011605,
-0.010247,
0.002016,
0.013292,
0.009547,
-0.009547,
-0.025432,
-0.014774,
0.025597,
0.065309,
0.057078,
-0.022058,
-0.135144,
-0.188601,
-0.084650,
0.206831,
0.607737,
0.957119,
1.094239,
0.957119,
0.607737,
0.206831,
-0.084650,
-0.188601,
-0.135144,
-0.022058,
0.057078,
0.065309,
0.025597,
-0.014774,
-0.025432,
-0.009547,
0.009547,
0.013292,
0.002016,
-0.010247,
-0.011605,
-0.002346,
0.007490,
0.009177,
0.002716,
-0.004486,
};
#pragma DATA_SECTION (table_f_sin,".sin")
       //store coeffs in a special defined memory region
int table_f_sin[] ={
       /*step = 11.25 degrees, max int value (0x8000) = 32768 == 1.0 */
     0, // 0.000000000,
  6393, // 0.195090322,
 12540, // 0.382683432,
 18205, // 0.555570233,
 23170, // 0.707106781,
 27246, // 0.831469612,
 30274, // 0.923879533,
 32138, // 0.980785280,
 32767, // 1.000000000,
 32138, // 0.980785280,
 30274, // 0.923879533,
 27246, // 0.831469612,
 23170, // 0.707106781,
 18205, // 0.555570233,
 12540, // 0.382683432,
  6393, // 0.195090322,
     0, // 0.000000000,
 -6393, //-0.195090322,
-12540, //-0.382683432,
-18205, //-0.555570233,
-23170, //-0.707106781,
-27246, //-0.831469612,
-30274, //-0.923879533,
-32138, //-0.980785280,
-32767, //-1.000000000,
-32138, //-0.980785280,
-30274, //-0.923879533,
-27246, //-0.831469612,
-23170, //-0.707106781,
-18205, //-0.555570233,
-12540, //-0.382683432,
 -6393, //-0.195090322,
};
#pragma DATA_SECTION (table_f_cos,".cos")
       //store coeffs in a special defined memory region

int table_f_cos[] ={
       /*step = 11.25 degrees, max int value (0x8000) = 32768 == 1.0 */
 32767, // 1.000000000,
 32138, // 0.980785280,
 30274, // 0.923879533,
 27246, // 0.831469612,
 23170, // 0.707106781,
 18205, // 0.555570233,
 12540, // 0.382683432,
  6393, // 0.195090322,
     0, // 0.000000000,
 -6393, //-0.195090322,
-12540, //-0.382683432,
-18205, //-0.555570233,
-23170, //-0.707106781,
-27246, //-0.831469612,
-30274, //-0.923879533,
-32138, //-0.980785280,
-32767, //-1.000000000,
-32138, //-0.980785280,
-30274, //-0.923879533,
-27246, //-0.831469612,
-23170, //-0.707106781,
-18205, //-0.555570233,
-12540, //-0.382683432,
 -6393, //-0.195090322,
     0, // 0.000000000,
  6393, // 0.195090322,
 12540, // 0.382683432,
 18205, // 0.555570233,
 23170, // 0.707106781,
 27246, // 0.831469612,
 30274, // 0.923879533,
 32138, // 0.980785280,
};
```

filter_int.h

```
/************************************************************************/
/*                                                                      */
/* Contains the filter coeffs tables for the FIR interpolation filters  */
/*                                                                      */
/* File:            filter_int.h                                        */
/* Version:         1.0                                                 */
/* Last update:     02-05-2000                                          */
/* Creation date:   02-05-2000                                          */
/* Author:          Roel Schiphorst                                     */
/*                                                                      */
/************************************************************************/

#define NH16  19    // number of filter coefs for 1st filter
#define NH32  10    // number of filter coefs for 2nd filter
#define NH64  6     // number of filter coefs for 3rd filter
#define NH128 6     // number of filter coefs for 4th filter
#define interpolation_factor 16    // total interpolation factor
#define frame_length 728*interpolation_factor   // define total frame length
#define frame_length_min_1 728*interpolation_factor - 1

#pragma DATA_SECTION (h16,".co_h16")
       //store coeffs in a special defined memory region
DATA h16[NH16] ={
       /* total sum = 2 (interpolation = 2, so 1/2 of samples are 0 */
82,    /* max int value = 32767 (0x7fff) == 1.0 */
0,
-498,
-2,
1774,
2,
-5166,
-4,
20194,
32760,
20194,
-4,
-5166,
2,
1774,
-2,
-498,
0,
82,
};

#pragma DATA_SECTION (h32,".co_h32")
       //store coeffs in a special defined memory region
DATA h32[NH32] ={
       /* total sum = 2 (interpolation = 2, so 1/2 of samples are 0 */
586,   /* max int value = 32767 (0x7fff) == 1.0 */
-1006,
-3892,
7416,
29654,
29654,
7416,
-3892,
-1006,
586,
};

#pragma DATA_SECTION (h64,".co_h64")
       //store coeffs in a special defined memory region
DATA h64[NH64] ={
       /* total sum = 2 (interpolation = 2, so 1/2 of samples are 0 */
-2888, /* max int value = 32767 (0x7fff) == 1.0 */
5518,
30170,
30170,
5518,
-2888,
};

#pragma DATA_SECTION (h128,".co_h128")
       //store coeffs in a special defined memory region
DATA h128[NH128] ={
       /* total sum = 2 (interpolation = 2, so 1/2 of samples are 0 */
-2946, /* max int value = 32767 (0x7fff) == 1.0 */
5308,
30410,
30410,
5308,
-2946,
};

#pragma DATA_SECTION (dbuffer16,".db16")
       //store delay buffer in a special defined memory region
DATA dbuffer16[NH16];      //define buffer
DATA *dp16 = dbuffer16;    //define pointer to buffer

#pragma DATA_SECTION (dbuffer32,".db32")
       //store delay buffer in a special defined memory region
DATA dbuffer32[NH32];      //define buffer
DATA *dp32 = dbuffer32;    //define pointer to buffer

#pragma DATA_SECTION (dbuffer64,".db64")
       //store delay buffer in a special defined memory region
DATA dbuffer64[NH64];      //define buffer
DATA *dp64 = dbuffer64;    //define pointer to buffer

#pragma DATA_SECTION (dbuffer128,".db128")
       //store delay buffer in a special defined memory region
DATA dbuffer128[NH128];    //define buffer
DATA *dp128 = dbuffer128;  //define pointer to buffer
```

vecs.asm

```
;/**********************************************************************/
;/* Setting up the interrupt vector table                              */
;/*                                                                    */
;/* File:           vecs.asm                                           */
;/* Version:        1.0                                                */
;/* Last update:    01-05-2000                                         */
;/* Creation date:  01-05-2000                                         */
;/* Author:         Roel Schiphorst                                    */
;/*                                                                    */
;/**********************************************************************/

       .title "vecs.asm"
       .ref   _c_int00,_codrx

       .sect     ".vectors"
       .mmregs

;vector table:
RESET   bd       _c_int00  ;Start of main program,
                           ;_c_int00 provided by rts.lib
              nop
              nop
       .space 76*16
BRINT0 bd   _codrx                ;serial port receive interrupt
       nop                        ;goto codrx-function
       nop
       .space  44*16
       .end
```

549.cmd

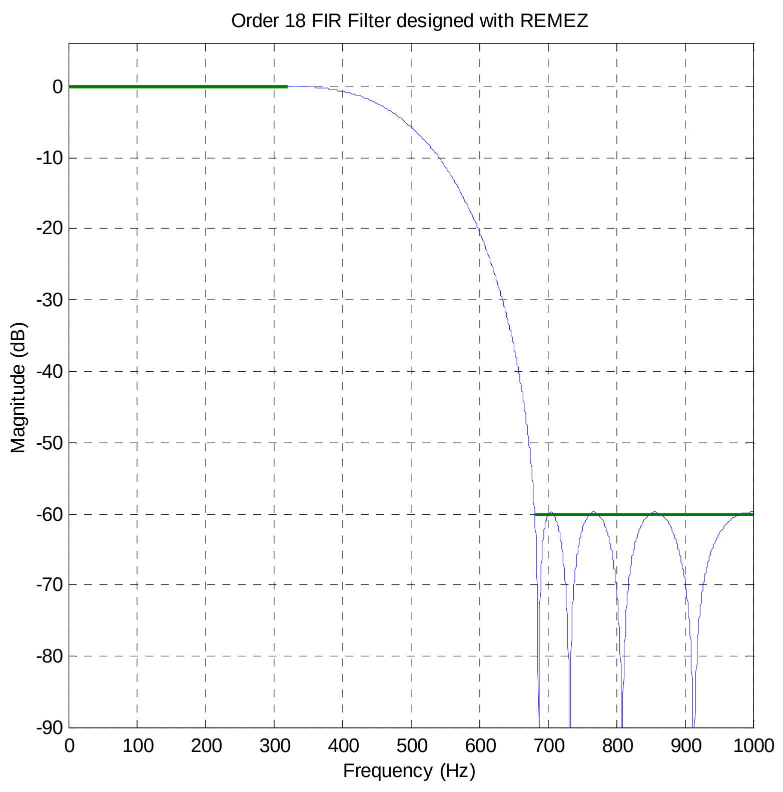
```
/**********************************************************************/
/* Functions for setting up memory map of the 549 DSP processor       */
/*                                                                    */
/* File:                  549.cmd                                     */
/* Version:               1.0                                         */
/* Last update:           01-05-2000                                  */
/* Creation date:         01-05-2000                                  */
/* Author:                Roel Schiphorst                             */
/*                                                                    */
/**********************************************************************/
main.obj
cdma.obj
frame.obj
mod_qpsk.obj
```

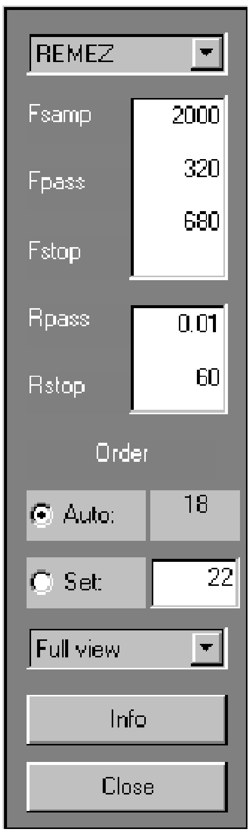
```
-o transmit.out

-m transmit.map

-e RESET

MEMORY
{
      PAGE 0:      ROM:          origin = 00000h, length = 007FFh
                   PRAM0:        origin = 00800h, length = 0F780h
                   VECTOR:       origin = 0ff80h, length = 00080h
      PAGE 1:      REGS:         origin = 00000h, length = 00050h
                   I_O:          origin = 00050h, length = 00010h
                   DRAM0:        origin = 0x0060, length = 0x0020
                   DRAM1:        origin = 0x0080, length = 0x1f80
                   DRAM2:        origin = 0x2000, length = 0x6000
}

SECTIONS
{
    .vectors :     > VECTOR
      .text :      > PRAM0
      .cinit :     > ROM
      .bss :       > DRAM2
      .cos :       > DRAM2
      .sin :       > DRAM2
      .rcf :       > DRAM2
      .co_h16 :    > DRAM2, align(128) /* alignment of filter coeffs */
      .co_h32 :    > DRAM2, align(128)
      .co_h64 :    > DRAM2, align(128)
      .co_h128 :   > DRAM2, align(128)
      .db16 :      > DRAM2, align(128) /* and filter buffers */
      .db32 :      > DRAM2, align(128)
      .db64 :      > DRAM2, align(128)
      .db128 :     > DRAM2, align(128)
      .stack :     > DRAM1
}
```

Appendix II: Interpolation-filter data

The transmitter uses a four-stage interpolation filter. Below the frequency responses are shown and the accompanying filter coefficients.

Interpolation filter I





REMEZ

Fsamp 2000

Fpass 320

Fstop 680

Rpass 0.01

Rstop 60

Order

● Auto: 18

○ Set: 22

Full view

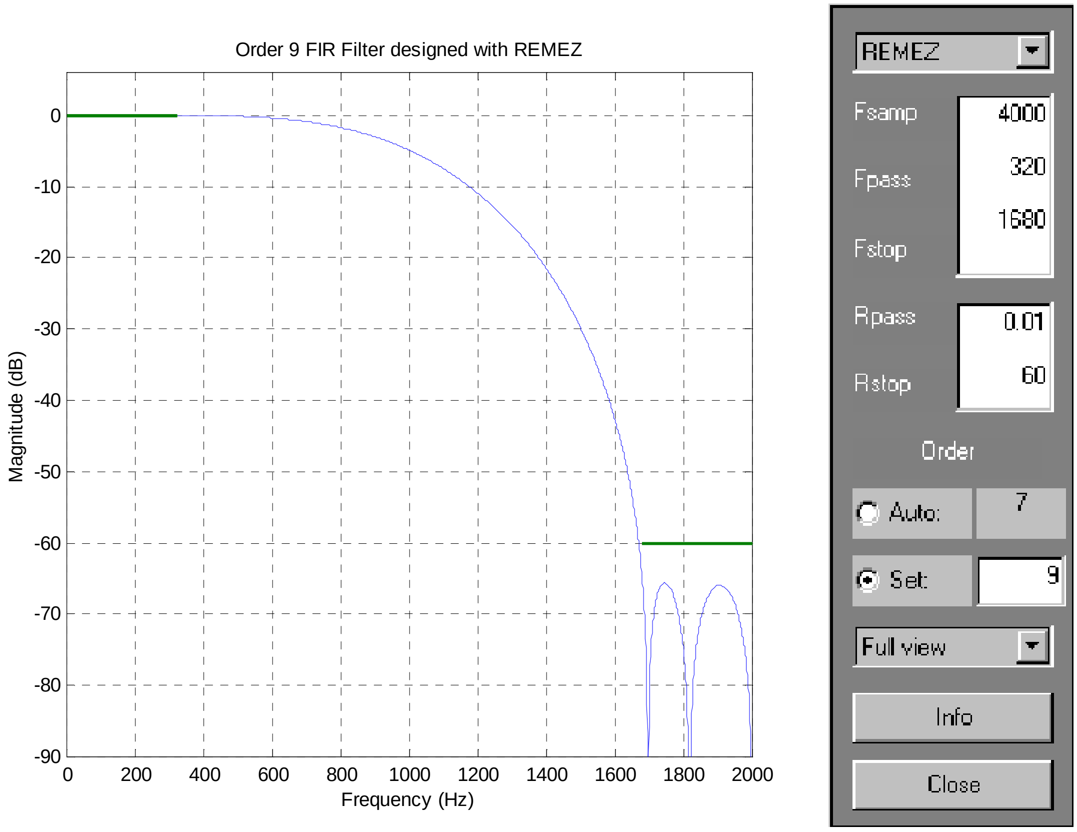
Figure 59: frequency response of interpolation filter I

The filter coefficients of the interpolation filter I are shown in Table 8.

Coefficients	**Interpolation filter I**
C0	0.00297634096950
C1	0.00000572929590
C2	-0.01193960904697
C3	-0.00001163211854
C4	0.03357607686325
C5	0.00002113699900
C6	-0.08497456198584
C7	-0.00002556341517
C8	0.31067825918356
C9	0.31067825918356
C10	0.31067825918356
C11	-0.00002556341517
C12	-0.08497456198584
C13	0.00002113699900
C14	0.03357607686325
C15	-0.00001163211854
C16	-0.01193960904697
C17	0.00000572929590
C18	0.00297634096950

Table 8: filter coefficients of interpolation filter I

Interpolation filter II



REMEZ

Fsamp 4000

Fpass 320

Fstop 1680

Rpass 0.01

Rstop 60

Order

Auto: 7

[✓ Set:] 9

Full view

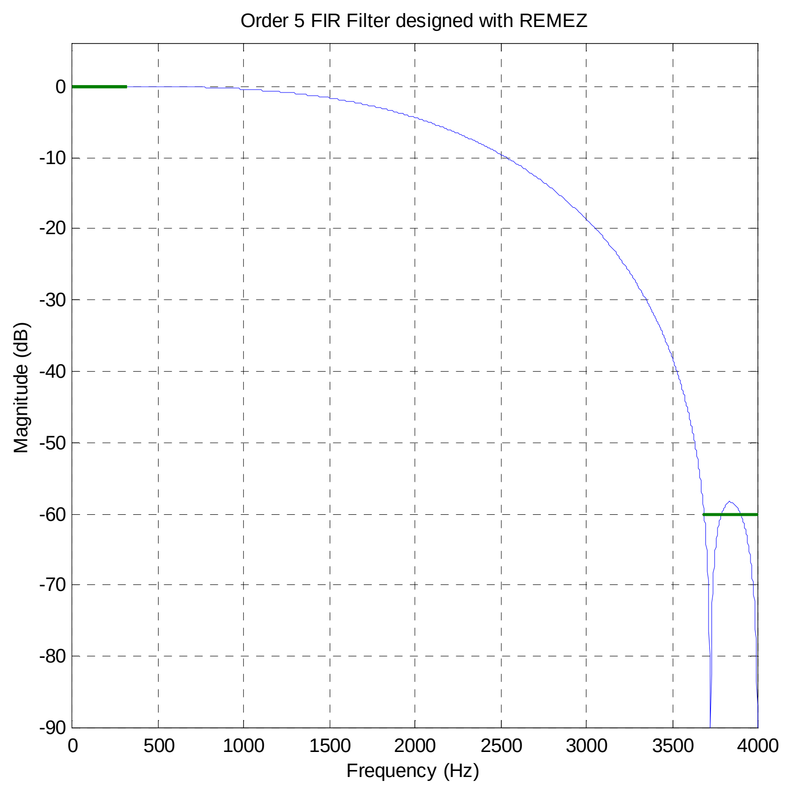
Figure 60: frequency response of interpolation filter II

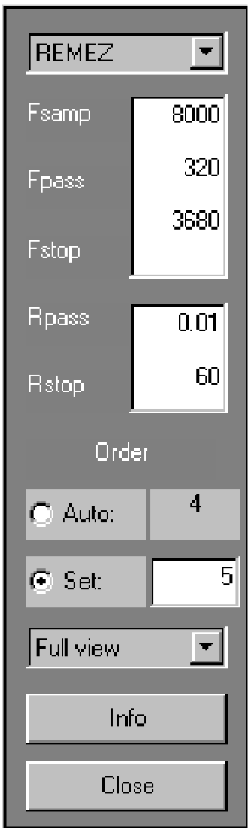
The filter coefficients of the interpolation filter II are shown in Table 9.

Coefficients	Interpolation filter II
C0	0.00910909968705
C1	-0.01590330913436
C2	-0.05958142823839
C3	-0.05958142823839
C4	0.45198770231206
C5	0.45198770231206
C6	0.11414753579084
C7	-0.05958142823839
C8	-0.01590330913436
C9	0.00910909968705

Table 9: filter coefficients of interpolation filter II

Interpolation filter III




REMEZ

Fsamp	8000
Fpass	320
Fstop	3680
Rpass	0.01
Rstop	60

Order

Auto: 4

[✓ Set:] 5

Full view

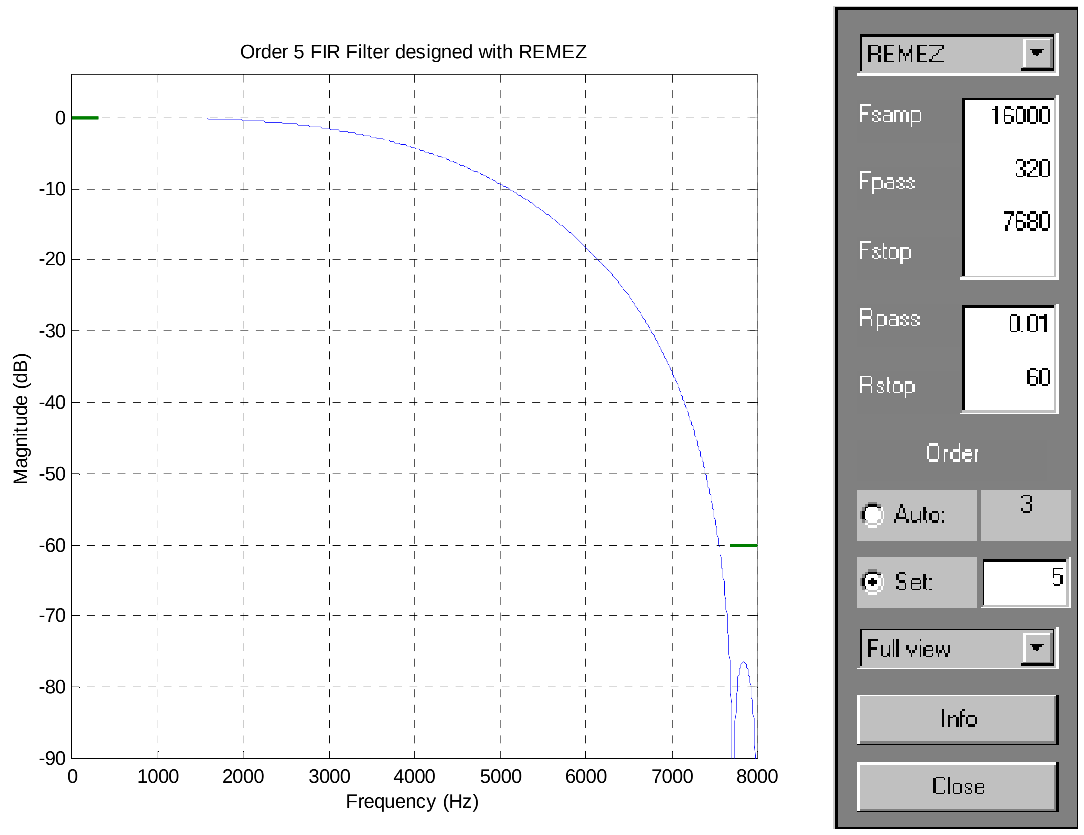
Figure 61: frequency response of interpolation filter III

The filter coefficients of the interpolation filter III are shown in Table 10.

Coefficients	Interpolation filter III
C0	-0.04395008588476
C1	0.08475721686120
C2	0.45978475413957
C3	0.45978475413957
C4	0.08475721686120
C5	-0.04395008588476

Table 10: filter coefficients of interpolation filter III

Interpolation filter IV



REMEZ

Fsamp 16000

Fpass 320

Fstop 7680

Rpass 0.01

Rstop 60

Order

Auto: 3

[✓ Set:] 5

Full view

Figure 62: frequency response of interpolation filter IV

The filter coefficients of the interpolation filter IV are shown in Table 11

Coefficients	**Interpolation filter IV**
C0	-0.04493820850244
C1	0.08098307261048
C2	0.46403046514701
C3	0.46403046514701
C4	0.08098307261048
C5	-0.04493820850244

Table 11: filter coefficients of interpolation filter IV

Appendix III: Source code of the software-radio receiver

The software-radio receiver consists of the following files (See also Figure 63):

- *549.cmd*
  defines the memory map of the DSP
- *dsplib.h*
  header file of TMS320C54x DSPLIB library
- *filter_dec.h*
  filter coefficients of the decimation filters
- *reg549.h*
  defines symbols for memory-mapped registers

- *table.h*
  header file for root-raised-cosine filter and cosine/sine table
- *tms320.h*
  defines data types DATA and LDATA
- *54xdsp.lib*
  TMS320C54x DSPLIB library
- *rts.lib*
  standard library for initialization and booting up of DSP
- *carrier_mult.c*
  functions for multiplying a carrier with a cosine/sine table
- *cdma.c*
  implementation of the CDMA block
- *cic.c*
  implementation of the CIC filter
- *compare.c*
  calculates the BER
- *demod_QPSK.c*
  functions for QPSK demodulation and root-raised-cosine filter
- *detect.c*
  functions for detecting the initial phase of the carrier
- *main.c*
  implementation of the main loop and other functions such as initialization and decimation block
- *vecs.asm*
  defines the interrupt table

The source code of these files is shown in the subsections below.

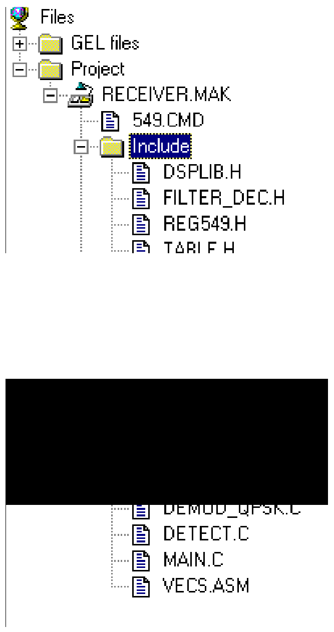


Figure 63: file hierarchy of the software-radio receiver

carrier_mult.c

```
/**************************************************************************/
/* Functions for multiplying a signal with a carrier                    */
/*                                                                      */
/* File:                    carrier_mult.c                              */
/* Version:                1.0                                          */
/* Last update:            28-04-2000                                   */
/* Creation date:          28-04-2000                                   */
/* Author:                 Roel Schiphorst                              */
/*                                                                      */
/************************************************************************/

/* Function Prototypes */
void carrier_mult( int, int*, int, int*, int*, int);

/************************************************************************/
/*                                                                      */
/* Function: carrier_mult()                                             */
/*                                                                      */
/* This function multiplies a vector with a cosine or sine. The         */
/* cosine or sine is derived from a table with 32 elements              */
/*                                                                      */
/* args:                                                                */
/*          NumOfSampl_f                                                */
/*                 Length of the cosine/sine period                     */
/*                                                                      */
/*          f                                                           */
/*                 Pointer to a sine/cosine table with 32 elements      */
/*                                                                      */
/*          NumOfSampl_in                                               */
/*                 Length of input vector, this must be a multiple of */
/*                 NumOfSampl_f                                         */
/*                                                                      */
/*          in                                                          */
/*                 Pointer to input vector with length NumOfSampl_in    */
/*                                                                      */
/*          out                                                         */
/*                 Pointer to output vector with length NumOfSampl_in   */
/*                                                                      */
/*          offset                                                      */
/*                 Indicates the zero-phase point in the input vector   */
/*                 Range: 0..32                                         */
/*                                                                      */
/* Return Value :                                                       */
/*         NONE                                                         */
/*                                                                      */
/************************************************************************/

void carrier_mult( int NumOfSampl_f, int *f, int NumOfSampl_in, int* in,
                   int* out, int offset)
{
       int i,j;
       int f_index;
       long int temp;

       f_index = 32 - offset;     //Match cos/sin with the input vector phase

       for (i = 0; i < NumOfSampl_in; i += NumOfSampl_f)
       {
              for (j = 0; j < NumOfSampl_f; j++)
              {
                     temp = f[f_index] * (*in++);
                            //Multiply the input with the cosine/sine value

                     (*out++) = temp >> 16;     //Use only the 16 MSBs of temp

                     f_index = (f_index + (32/NumOfSampl_f)) % 32;
                            //Change pointer to the next cosine/sine value
              }
       }
    return;
}
```

cdma.c

See appendix I.

cic.c

```
/**********************************************************************/
/* Functions for implementing the Cascaded Integrated Comb(CIC) filters */
/*                                                                    */
/* File:                  cic.c                                       */
/* Version:               1.0                                         */
/* Last update:           28-04-2000                                  */
/* Creation date:         28-04-2000                                  */
/* Author:                Roel Schiphorst                             */
/*                                                                    */
/**********************************************************************/

#include <tms320.h>
/* Function Prototypes */
void CIC_L3M4(DATA*, DATA*, LDATA*, int, LDATA*);

/**********************************************************************/
/*                                                                    */
/* Function: CIC_L3M4()                                              */
/*                                                                    */
/* This function implements a 3 stage CIC-filter with a decimation    */
/* factor of 4                                                        */
/*                                                                    */
/* args:                                                              */
/*          in                                                        */
/*                input vector with NumberOfInputSamples elements     */
/*                                                                    */
/*          out                                                       */
/*                output vector with NumberOfInputSamples/4 elements  */
/*                                                                    */
/*          state_buffer                                              */
/*                Vector with 6 elements which contains the buffer    */
/*                values of the CIC filter                            */
/*                                                                    */
/*          NumberOfInputSamples                                      */
/*                Length of the input vector                          */
/*                                                                    */
/*          temp                                                      */
/*                temporary buffer of length NumberOfInputSamples     */
/*                                                                    */
/* Return Value :                                                     */
/*         NONE                                                       */
/*                                                                    */
/**********************************************************************/

void CIC_L3M4(DATA *in, DATA* out, LDATA* state_buffer,
              int NumberOfInputSamples, LDATA* temp)
{
      int i;
      int temp_in;

      for(i = 0; i < NumberOfInputSamples; i++)
      {
            temp_in = in[i]/30;  //scale input vector to prevent a
                                 // quick overflow in the filter buffers
            state_buffer[2] += state_buffer[1];     //integrator #3
            state_buffer[1] += state_buffer[0];     //integrator #2
            state_buffer[0] += temp_in;             //integrator #1

            temp[i] = state_buffer[2];              //store temp data
      }

      for(i = 0; i < NumberOfInputSamples/4; i++)    //downsample factor 4
      {
            out[i] = temp[4*i] - 3*state_buffer[3] + 3*state_buffer[4] –
                     state_buffer[5];

            //calculate output sample
            state_buffer[5] = state_buffer[4];      //update buffer values
            state_buffer[4] = state_buffer[3];
            state_buffer[3] = temp[4*i];
      }
}
```

compare.c

```
/***********************************************************************/
/* Functions for calculation the BER (Bit Error Rate)                 */
/*                                                                     */
/* File:                    compare.c                                  */
/* Version:                 1.0                                        */
/* Last update:             03-05-2000                                 */
/* Creation date:           03-05-2000                                 */
/* Author:                  Roel Schiphorst                            */
/*                                                                     */
/***********************************************************************/

#include <tms320.h>

/* Function Prototypes */
void compare(DATA*, float*);

/***********************************************************************/
/*                                                                     */
/* Function: cdma()                                                    */
/*                                                                     */
/* This function determines the number of errors in the received frame */
/*                                                                     */
/* args:                                                               */
/*          rx_data                                                    */
/*                 Received frame data (256 bits)                      */
/*                                                                     */
/*          ber                                                        */
/*                 Bit Error Rate                                      */
/*                                                                     */
/* Return Value :                                                      */
/*         NONE                                                        */
/*                                                                     */
/***********************************************************************/

DATA reference_frame_data[256] = {
             1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1,
             1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1,
             1, 0, 1, 0,  1, 0, 1, 0,  1, 0, 1, 0,  1, 0, 1, 0,
             1, 0, 1, 0,  1, 0, 1, 0,  1, 0, 1, 0,  1, 0, 1, 0,

             1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1,
             1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1,
             1, 0, 1, 0,  1, 0, 1, 0,  1, 0, 1, 0,  1, 0, 1, 0,
             1, 0, 1, 0,  1, 0, 1, 0,  1, 0, 1, 0,  1, 0, 1, 0,

             0, 1, 0, 1,  0, 1, 0, 1,  0, 1, 0, 1,  0, 1, 0, 1,
             0, 1, 0, 1,  0, 1, 0, 1,  0, 1, 0, 1,  0, 1, 0, 1,
             0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0,
             0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0,

             0, 1, 0, 1,  0, 1, 0, 1,  0, 1, 0, 1,  0, 1, 0, 1,
             0, 1, 0, 1,  0, 1, 0, 1,  0, 1, 0, 1,  0, 1, 0, 1,
             0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0,
             0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0
             };

void compare(DATA *rx_data, float* ber)
{
      static unsigned long errorsum = 0;
      static unsigned long totalbitsum = 0;
      float temp_error;
      float temp_totalbit;
      int i = 0;

      for (i = 0; i < 256; i++)
      {
            if (rx_data[i] != reference_frame_data[i])
            {
                  errorsum++;
            }
      }
      totalbitsum += 256;

      temp_error = errorsum;
      temp_totalbit = totalbitsum;
      temp_error = temp_error/temp_totalbit;
```

```
       *ber = temp_error;
}
```

demod_QPSK.c

```
/**********************************************************************/
/* Functions for demodulating a QPSK signal                          */
/*                                                                    */
/* File:                 demod_QPSK.c                                 */
/* Version:              1.0                                          */
/* Last update:          01-05-2000                                   */
/* Creation date:        01-05-2000                                   */
/* Author:               Roel Schiphorst                              */
/*                                                                    */
/**********************************************************************/

#include <tms320.h>

/* Function Prototypes */

int  sync_corr(DATA*, DATA*, DATA* );
void rx_RRC_filt(DATA*, DATA*, unsigned int, DATA*, DATA*, DATA*,
                    unsigned int);
void demod_QPSK(DATA*, DATA*, unsigned*);

/* Data */
DATA   cor_sync[] = {      -1, 1, // vector which contains the sync-word
                           -1, 1,  // the left row is sent via the I channel
                           -1, 1, // the other via the Q channel
                            1,-1,
                            1, 1,
                            1,-1,
                           -1,-1,
                            1,-1,
                           -1,-1,
                           -1, 1,
                            1,-1,
                            1, 1,
                           -1, 1,
                           -1, 1,
                            1,-1,
                            1,-1
                    };

/*********************************************************************/
/*                                                                   */
/* Function: sync_corr()                                             */
/*                                                                   */
/* This function searches the best match of the sync-word in the input */
/* vector sync word. This done via correlation. The position of the    */
/* best match is returned and used by the demodulation process.       */
/*                                                                   */
/* args:                                                             */
/*          I                                                        */
/*                 input vector of the in-phase channel              */
/*                                                                   */
/*          Q                                                        */
/*                 input vector of the quadrature channel            */
/*                                                                   */
/*          filt                                                     */
/*                 pointer to 48-taps-root-raised-cosine-filter vector */
/*                                                                   */
/* Return Value :                                                     */
/*          sync_point                                                */
/*          position of the correlation-peak in the input vectors     */
/*                                                                   */
/**********************************************************************/


int sync_corr(DATA *I, DATA *Q, DATA *filt)
{
    LDATA   temp_rc[2*100];      // Temporary storage array for post-
                                 // filtered root-raised-cosine data.

    int     i, j,
            sync_point;           // synchronization point
    static long int max;          // contains the maximal correlation value
    long int *rc_data,            // pointer to temp_rc array
             corr_mag,            // correlation value of one posibility
             I_temp, Q_temp,      // temporary filter output values
             rc_i, rc_q;          // used for calculating correlation
    DATA     *sync_ptr,           // pointer to synchronization table
             sync_i, sync_q;      // used for calculating correlation

/***************************************/
/* Filter just enough data to span    */
/* correlation window                 */
/*                                    */
/* Store resultant 100 I,Q pairs      */
/* into temp_rc[] storage.            */
/***************************************/

    rc_data = temp_rc; // Set pointer to temporary storage for filter output

    I += 24;         // Intentional offset to start 6 symbols
    Q += 24;         // to left of expected sync point.

    for (i = 0 ; i < 100; i++)
    {
        I_temp = 0;  // Zero filter sums
        Q_temp = 0;

        for (j = 0 ; j < 48 ; j++)      // Calculate output value of root
                                        // raised cosine filter
        {
            I_temp += (*(I++)) * (*filt);
            Q_temp += (*(Q++)) * (*filt++);
        }
        filt  -= 48;       // Set Filter pointer back to start

        I -= 47;           // Start 1 sample over next time
        Q -= 47;

        *(rc_data++) = I_temp/16;   // Store Filter Output I term and divide
                                    // by 16 to prevent overflow
        *(rc_data++) = Q_temp/16;   // Store Filter Output Q term and divide
                                    // by 16 to prevent overflow
    }

/*****************************************/
/* Correlate with Sync Word             */
/* Keep track of correlation peak.      */
/*****************************************/

    max = 0;                      // Correlation Magnetude Maximum

    sync_point = 0;             // Index into data to sync point

    rc_data = temp_rc;          // Set pointer to beginning of filtered data

    for (i = 0; i < 10; i++)
    {

        I_temp = 0; // Set correlation values to zero
        Q_temp = 0;


        sync_ptr = cor_sync;   // Set pointer to sync-word table

       /* Perform Correlation at offset 'i' into data */
        for (j = 0; j < 16; j++)
        {
            sync_i = *(sync_ptr++);
            sync_q = *(sync_ptr++);
            rc_i   = *(rc_data++);
            rc_q   = *(rc_data++);
            rc_data += 6;
            I_temp += sync_i*rc_i;
            Q_temp += sync_q*rc_q;
        }
        rc_data -= 126;    // 128 would set pointer back to start we will
                           // start 1 sample (I,Q pair) over next time
       /* Compute Correlation Magnetude & Maximize */
        corr_mag = I_temp + Q_temp;

        if( corr_mag > max ) //if new point is greater than it is new max.
        {
            sync_point = i;
            max = corr_mag;
        }
    }

    return(sync_point + 24); // return correlation-peak position

                    // 24 is added because we started after 24 samples
                    // (to reduce calculations)
}

/*************************************************************************/
/*                                                                       */
/* Function: rx_RRC_filt()                                               */
/*                                                                       */
/* This filters the QPSK signal with a root raised cosine filter.       */
/* This reduces the ISI (Inter Symbol Interference). The same filter is */
/* used at the transmitter.                                              */
/*                                                                       */
/* args:                                                                 */
/*            I                                                          */
/*                   input vector of the in-phase channel                */
/*                                                                       */
/*            Q                                                          */
/*                   input vector of the quadrature channel              */
/*                                                                       */
/*            sync_pos                                                   */
/*                   location of the start of the sync-word in the input */
/*                   vectors                                             */
/*                                                                       */
/*            filt                                                       */
/*                   pointer to 48-taps-root-raised-cosine-filter vector */
/*                                                                       */
/*            I_bits                                                     */
/*                   decoded I channel which contains length/4 elements  */
/*                                                                       */
/*            Q_bits                                                     */
/*                   decoded Q channel which contains length/4 elements  */
/*                                                                       */
/*            length                                                     */
/*                   number of elements of the both input vectors        */
/*                                                                       */
/* Return Value :                                                        */
/*            NONE                                                       */
/*                                                                       */
/*************************************************************************/

void rx_RRC_filt(DATA* I, DATA* Q, unsigned int sync_pos, DATA* filt,
                 DATA* I_bits, DATA* Q_bits, unsigned int length)
{
    int     i,j;
    long int I_temp, Q_temp; // output values of root raised cosine filter
    DATA    *filt_ptr,       // pointer to the filter table, filt
            *isrc_ptr,       // pointer to the input vector I
            *qsrc_ptr;       // pointer to the input vector Q


/* Update pointers to SRC data to start at sync point */

    I += sync_pos;         // Start at the begin of the sync word
    Q += sync_pos;

/********************/
/* Filter Data Loop */
/********************/

    for (i = 0 ; i < length ; i++)
    {
       /* Set filter sums to zero */
        I_temp = 0;
        Q_temp = 0;
       /* Initialize pointers to input data */
        isrc_ptr = I;
        qsrc_ptr = Q;

       /* Set filter pointer to filter */
        filt_ptr = filt;

       /* Perform 48-tap filter */
        for (j = 0 ; j < 48 ; j++)
        {
            I_temp += (*(isrc_ptr++)) * (*filt_ptr);
            Q_temp += (*(qsrc_ptr++)) * (*(filt_ptr++));
        }

       /* Store output samples */
       if (I_temp > 0)
       {
                    *(I_bits++) = 1;
       }
       else
       {
                    *(I_bits++) = -1;
       }
       if (Q_temp > 0)
       {
                    *(Q_bits++) = 1;
       }
       else
       {
                    *(Q_bits++) = -1;
       }

       /* Update start sample pointers */
        I += 4;
        Q += 4;
    }

    return;
}

/*********************************************************************/
/*                                                                   */
/* Function : demod_QPSK()                                           */
/*                                                                   */
/* This routine extracts 256-bit slot data from a demodulated I and Q   */
/* channel A demodulated 1 means 0 and -1 implies a 1.               */
/*                                                                   */
/* args:                                                             */
/*          I                                                        */
/*                 input vector of the demodulated in-phase channel    */
/*                                                                   */
/*          Q                                                        */
/*                 input vector of the demodulated quadrature channel */
/*                                                                   */
/*          data                                                     */
/*          output array, which contains the decoded data bits of the */
/*                 received frame (256 bits)                         */
/*                                                                   */
/* Return Value :                                                    */
/*         NONE                                                      */
/*                                                                   */
/*********************************************************************/

void demod_QPSK(DATA* I, DATA* Q, unsigned *data)
{
    DATA *I_ptr, *Q_ptr;   // pointer to both input vectors, I and Q
    int i;

    I_ptr = I + 24; // start at the begin position of the first data field
    Q_ptr = Q + 24;

    for( i = 0; i < 64; i++ )  // decode QPSK signal
    {
       if (*(I_ptr++) < 0) // -1 means bit = 1 and 1 means bit = 0
       {
            *(data++) = 1;
       }
       else
       {
             *(data++) = 0;
       }
       if (*(Q_ptr++) < 0) // -1 means bit = 1 and 1 means bit = 0
       {
             *(data++) = 1;
       }
       else
       {
             *(data++) = 0;
       }
    }
    I_ptr = I + 104; // start at the begin position of the second data field
    Q_ptr = Q + 104;

    for( i = 0; i < 64; i++ )    // decode QPSK signal
    {
       if (*(I_ptr++) < 0) // -1 means bit = 1 and 1 means bit = 0
       {
             *(data++) = 1;
       }
       else
       {
             *(data++) = 0;
       }

       if (*(Q_ptr++) < 0) // -1 means bit = 1 and 1 means bit = 0
       {
             *(data++) = 1;
       }
       else
       {
             *(data++) = 0;
       }
    }
    return;
}
```

detect.c

```
/**********************************************************************/
/* Function for detecting the phase of the input signal               */
/*                                                                    */
/* File:                  detect.c                                    */
/* Version:               1.0                                         */
/* Last update:           01-05-2000                                  */
/* Creation date:         01-05-2000                                  */
/* Author:                Roel Schiphorst                             */
/*                                                                    */
/**********************************************************************/

/* Function Prototypes */
int detect_phase_carrier(int*);

/**********************************************************************/
/*                                                                    */
/* Function: detect_phase_carrier()                                   */
/*                                                                    */
/* This function detects the phase of the input vector. It returns the  */
/* best match of the zero phase. This zero-phase position is used by    */
/* the alignment of the sin/cosine table to the input signal. Therefore */
/* the range = 0..32.                                                 */
/*                                                                    */
/* args:                                                              */
/*          in                                                        */
/*                 input vector                                       */
/*                                                                    */
/* Return Value :                                                     */
/*          offset                                                    */
/*          Zero-phase-position                                       */
/*                                                                    */
/**********************************************************************/
```

```
int detect_phase_carrier(int* in)
{
      int phase_negmin = 0;
            // position of the closest match in the input vector
            // of the zero-crossing (- --> +)
            // This point is the nearest sample at the negative part.

      int phase;   // return-value of the function
      int phase2;   // output value of the interpolation
      int negmin = 32000;     // value of phase_negmin
      int posmin = 32000;
            // value of phase_negmin + 1 (other side of the crossing)
      int min = 32000;
            // used in the interpolation to determine the position
      int i;
      int temp = 30000, temp_old = 30000;
            //temp value to determine the crossing
      int x[9];    // array which contains the interpolated value

      for (i = 0; i < 20; i++)
            // Evaluate the first 20 sample for a zero-crossing
      {
            temp_old = temp;
            temp = (*in);

            in++;

            if ((temp_old <= 0) && (temp >= 0))
                  // if previous sample = - and this sample
                  // = + then zero-crossing
            {
                  phase_negmin = (4 + (i - 1)) % 4;
                        // carrier-period is 4 samples
                  posmin = temp;      // save + sample of the crossing
                  negmin = temp_old;  // save - sample of the crossing
                  break;              // exit loop
            }
      }

      x[0] = negmin;                  // interpolation
      x[1] = 7*(negmin/8) + posmin/8;
      x[2] = 3*(negmin/4) + posmin/4;
      x[3] = 5*(negmin/8) + 3*(posmin/8);
      x[4] = negmin/2 + posmin/2;
      x[5] = 3*(negmin/8) + 5*(posmin/8);
      x[6] = negmin/4 + 3*(posmin/4);
      x[7] = negmin/8 + 7*(posmin/8);
      x[8] = posmin;

      for (i = 0; i < 9; i++)          // determine closest match to

      // zero-crossing
      {
            if (abs(x[i]) < min)
            {
                  min = abs(x[i]);    // save new minimum
                  phase2 = i;         // save position
            }
      }

      phase = 8*phase_negmin;    // Calculate zero-phase position =
      phase += phase2;           // 8*input-match + interpolation position
      phase += 4;                // phase value of zero-crossing = -1/4 pi

      // so 4 (32/8) has to be added to the phase
      phase = phase % 32;        // normalize phase

      return phase;
}
```

main.c

```
/**************************************************************************/
/* This file contains the main loop of the QPSK receiver. Besides the   */
/* main loop it contains functions for initilization of the DSP and the */
/* AD/DA chip.                                                         */
/*                                                                     */
/* File:                  main.c                                       */
/* Version:               1.0                                          */
/* Last update:           03-05-2000                                   */
/* Creation date:         03-05-2000                                   */
/* Author:                Roel Schiphorst                               */
/*                                                                     */
/***********************************************************************/

#include <tms320.h>
#include <dsplib.h>

#include "filter_dec.h"
#include "reg549.h"
#include "table.h"

/* Function Prototypes */
void main(void);

volatile int p0_serialflag;       // 0 = normal operation 1 = programming
                                  // AD/DA chip
volatile int p0_serialint;        // 1 = serial port receive interrupt has
                                  // occured
volatile int p0_serial_frame;    // 1 = complete frame has been received
volatile int p0_state;           // state variable used in the detection of
                                  // a frame
                                  // 0 = wait for no signal
                                  // 1 = wait for signal
                                  // 2 = record frame
int p0_buffer[4];                 // buffer used by the receive interrupt
                                  //routine to detect a frame


DATA  I[800];       // Array for the I channel before the root raised cosine
                    // filter
DATA  Q[800];       // Array for the Q channel before the root raised cosine
                    // filter
DATA  I_bits[188];  // Array for the decoded I symbols
DATA  Q_bits[188];  // Array for the decode Q symbols

int rx_idata[256];  // Array for the data bits of the received frame
DATA rx_filt[48];   // Array for the root raised cosine filter
int input[frame_length];   // Array of the input samples which contain the
                           // frame

DATA buffer[128];   // Buffer which contains a small part of the input array

DATA out_cic[32];   // output array of the cic filter
LDATA buffer_cic[6];       // state array of the cic filter
LDATA temp_cic[128];       // temp data of the cic filter

DATA out8[16];             // output of the first FIR decimation filter
DATA out4[8];              // output of the second FIR decimation filter

float const1_1 = 1.1;      // constant value

unsigned int syncpoint = 0;       // Synchronisation point
float BER;                        // Bit Error Rate
/*
DATA code1[16]= { 1,1,1,1, 1,1,1,1, 1,1,1,1, 1,1,1,1 };
DATA code2[16]= { 1,1,1,1, 1,1,1,1, -1,-1,-1,-1, -1,-1,-1,-1 };
DATA code3[16]= { 1,1,1,1, -1,-1,-1,-1, 1,1,1,1, -1,-1,-1,-1 };
DATA code4[16]= { 1,1,1,1, -1,-1,-1,-1, -1,-1,-1,-1, 1,1,1,1 };
DATA code5[16]= { 1,1,-1,-1, 1,1,-1,-1, 1,1,-1,-1, 1,1,-1,-1 };
DATA code6[16]= { 1,1,-1,-1, 1,1,-1,-1, -1,-1,1,1, -1,-1,1,1 };
DATA code7[16]= { 1,1,-1,-1, -1,-1,1,1, 1,1,-1,-1, -1,-1,1,1 };
DATA code8[16]= { 1,1,-1,-1, -1,-1,1,1, -1,-1,1,1, 1,1,-1,-1 };
DATA code9[16]=  { 1,-1,1,-1, 1,-1,1,-1, 1,-1,1,-1, 1,-1,1,-1 };
DATA code10[16]= { 1,-1,1,-1, 1,-1,1,-1, -1,1,-1,1, -1,1,-1,1 };
DATA code11[16]= { 1,-1,1,-1, -1,1,-1,1, 1,-1,1,-1, -1,1,-1,1 };
DATA code12[16]= { 1,-1,1,-1, -1,1,-1,1, -1,1,-1,1, 1,-1,1,-1 };
DATA code13[16]= { 1,-1,-1,1, 1,-1,-1,1, 1,-1,-1,1, 1,-1,-1,1 };
DATA code14[16]= { 1,-1,-1,1, 1,-1,-1,1, -1,1,1,-1, -1,1,1,-1 };
DATA code15[16]= { 1,-1,-1,1, -1,1,1,-1, 1,-1,-1,1, -1,1,1,-1 };
DATA code16[16]= { 1,-1,-1,1, -1,1,1,-1, -1,1,1,-1, 1,-1,-1,1 };
*/
```

```
// There are 16 different CDMA codes, listed above. One code is choosen for
// the communication and is equal to the transmitter CDMA code
DATA code[16]= { 1,-1,1,-1, -1,1,-1,1, 1,-1,1,-1, -1,1,-1,1 };

/***************************************************************************/
/***************************************************************************/

void inline disable() {
      asm("  ssbx INTM");              // Disable all interrupts of the DSP
}

void inline enable() {
      asm ("  rsbx INTM");             // Enable all interrupts of the DSP
}

/*************************************************************************/
/*                                                                       */
/* Function : waitintr()                                                 */
/*                                                                       */
/* This function is used by the progreg()-function. It will only exit   */
/* the function if the serial port has received a new word.              */
/*                                                                       */
/* args:                                                                 */
/*          NONE                                                         */
/*                                                                       */
/* Return Value :                                                        */
/*    NONE                                                               */
/*                                                                       */
/*************************************************************************/

void waitintr(void)
{
      while (p0_serialint == 0);
             // Wait until data is received from the AD-DA chip

      p0_serialint = 0;                // Reset p0_serialint
      return;
}

/*************************************************************************/
/*                                                                       */
/* Function : progreg()                                                  */
/*                                                                       */
/* This function programs a register of the AD/DA chip via the serial   */
/* port                                                                  */
/*                                                                       */
/* args:                                                                 */
/*          progword                                                     */
/*                a 16 bit word which contains besides the address of */
/*                the register, also the new contents of the register */
/*                See the datasheet of the AD/DA-chip for more        */
/*                information                                          */
/*                                                                       */
/* Return Value :                                                        */
/*    NONE                                                               */
/*                                                                       */
/*************************************************************************/

void progreg(int progword)
{
      *BDXR0  = 0x0001;          // Request secondary communication
                                 // (read of write to registers)
      waitintr();                // Wait for serial port receive interrupt

      *BDXR0  = progword;        // Send configuration word to AD-DA chip

      waitintr();                // Wait for serial port receive interrupt
      return;
}

/*************************************************************************/
/*                                                                       */
/* Function : init()                                                     */
/*                                                                       */
/* This function initializes the DSP and the AD/DA-chip                 */
/*                                                                       */
/* args:                                                                 */
/*            NONE                                                       */
/*                                                                       */
/* Return Value :                                                        */
/*     NONE                                                              */
/*                                                                       */
/*************************************************************************/

void init()
{
      *SWWSR = 0x1209;           // Set wait states for memory
      *PMST = 0x0ffc0;           // Set location of interrupt table

      disable();                 // Disable all interrupts
      *BSPC0 = 0x0000;           // Initialize SPC
      *BSPC0 = *BSPC0 | 0x00C0;  // Initialize SPC
      *IMR  = 0x0010;            // Enable serial port 0 receive interupt
      *IFR  = 0xffff;            // Clear any pending interrupts

      enable();                  // Enable all interrupts

      p0_serialflag = 1;         // Indicated whether the AD-DA chip is
                                 // configure (1) or normal use (0)
      waitintr();

      progreg(0x0100);    // 01 = control 1 register, value 00 (default)

      progreg(0x0200);    // 02 = control 2 register, value 00 (default)
                          // Decimator and interpolator filters are disabled

      progreg(0x0304);    // 03 = Fk divide register, value 10
                          // controls filter clock rate and sample period

      progreg(0x0404);    // 04 = Fsclk divide register, value 10
                          // controls the shift (data) clock rate
      progreg(0x0500);    // 05 = control 3 register, value 00
                          // DAC reference enabled
      *BDXR0 = 0x0000;    // Send zero to end configuration

      p0_serialflag = 0;  // Configuration is finished.

      *IMR  = *IMR & 0xffef;     // disable serial port receive interrupts
}

/*************************************************************************/
/*                                                                       */
/* Function : codrx()                                                    */
/*                                                                       */
/* This function is called when a serial port receive interrupt occurs. */
/*                                                                       */
/* args:                                                                 */
/*            NONE                                                       */
/*                                                                       */
/* Return Value :                                                        */
/*     NONE                                                              */
/*                                                                       */
/*************************************************************************/

void interrupt codrx()
{     int sample_int;            // var to save sample value
      static int i = 0, j = 0, k = 0;
      static int long temp = 0;

      p0_serialint = 1;          // Set the interrupt variable to 1
      sample_int = *BDRR0;       // save sample value

      // If 'normal' operation (no programming of regs)
      if (p0_serialflag == 0)
      {
            if (p0_state < 2) // if p0_state = 0 or 1
            {
                  if (sample_int == -32768) sample_int = 32767;

                  // Store sample in buffer
                  p0_buffer[i] = abs(sample_int);

                  i = (i + 1) % 4;
```

```
                    temp = 0;
                    for (j = 0; j < 4; j++)
                    // Buffer is used for averaging the input
                    {
                          temp += p0_buffer[j];
                          // calculate abs sum
                    }

                    // if temp is great enough, signal detected, record
                    // frame
                    if (temp > 0x00007000)
                    {
                          if (p0_state == 1) p0_state = 2;
                    }

                    // if temp is small enough, no signal, goto next state
                    if (temp < 0x00000010)
                    {
                          if (p0_state == 0) p0_state = 1;
                    }

              }
              else
              {     // record the first frame_length samples
                    input[k] = sample_int;

                    // If last sample is transmitted then
                    if (k == frame_length_min_1)
                    {
                          // set p0_serial_frame var to 1;
                          p0_serial_frame = 1;
                          k = -1;       // reset index k
                    }
                    k++;
              }
        }
}

/*********************************************************************/
/*                                                                   */
/* Function : main()                                                 */
/*                                                                   */
/* Main loop of the QPSK receiver.                                   */
/*                                                                   */
/* args:                                                             */
/*          NONE                                                     */
/*                                                                   */
/* Return Value :                                                    */
/*     NONE                                                          */
/*                                                                   */
/*********************************************************************/

void main(void)
{
    int i,j;
    int offset;     // Zero-phase position of the input signal

    int decimation_factor_x_8 = decimation_factor*8;

    p0_serialint = 0;      // reset var
    p0_state = 0;          // reset var

    init();                // Initialization of AD-DA chip

       for( j = 0; j < 48; j ++)
       {
             tableRaisCosFilt[j] /= const1_1;
             // Normalize Root Raise Cosine table
       }

       fltoq15(tableRaisCosFilt, rx_filt, 48); // convert to DATA array

       for( j = 0; j < 48; j ++)
       {
             // divide by 4 (total sum of the filter = 4)
             // same filter is used by the transmitter
             // the transmitter interpolates the signal 4 times
             // so 3/4 of the input is zero and therefore the
             // total sum of the filter = 4

             rx_filt[j] >>= 2;
       }

    while (1)              // receive frames forever
    {
       p0_serial_frame = 0;// reset var
       *IMR  |= 0x0010;    // enable serial port receive interrupt

       // Wait until a new frame is received
       while (p0_serial_frame == 0)
       {
       }

       *IMR  &= 0xffef;           // Disable serial port interrupt
             p0_state = 0;        // Reset serial-port state var


       offset = detect_phase_carrier(input + 80);     // Determine offset

/***************************************************************************/
/*          Decimation                                                     */
/***************************************************************************/
        /* I channel*/
       for (j=0; j< NH8; j++) dbuffer8[j] = 0; // clear delay buffer (a must)
       for (j=0; j< NH4; j++) dbuffer4[j] = 0; // clear delay buffer (a must)
       for (j=0; j< 6; j++) buffer_cic[j] = 0; // clear delay buffer (a must)

       for ( i = 0; i < 100; i++ )
       {
             // Multiply with carrier
             carrier_mult(4, table_f_cos, decimation_factor_x_8,
                          &input[i*decimation_factor_x_8], buffer, offset);


             // CIC filter
             CIC_L3M4(buffer, out_cic, buffer_cic, decimation_factor_x_8,
                      temp_cic);


             /* 2 stage fir filter */
             firdec(out_cic, h8, out8, &dp8, NH8, 32, D8);
             firdec(out8, h4, out4, &dp4, NH4, 16, D4);

             /* Store output in I array */
             for (j = 0; j < 8; j++)
             {
                    I[i*8 + j] = out4[j];
             }
       }

       /* Q channel*/
       for (j=0; j< NH8; j++) dbuffer8[j] = 0; // clear delay buffer (a must)
       for (j=0; j< NH4; j++) dbuffer4[j] = 0; // clear delay buffer (a must)
       for (j=0; j< 6; j++) buffer_cic[j] = 0; // clear delay buffer (a must)

       for ( i = 0; i < 100; i++ )
       {
             // Multiply with carrier
             carrier_mult(4, table_f_sin, decimation_factor_x_8,
                          &input[i*decimation_factor_x_8], buffer, offset);

             // CIC filter
             CIC_L3M4(buffer, out_cic, buffer_cic, decimation_factor_x_8,
                      temp_cic);

             /* 2 stage fir filter */
             firdec(out_cic, h8, out8, &dp8, NH8, 32, D8);
             firdec(out8, h4, out4, &dp4, NH4, 16, D4);

             /* Store output in Q array */
             for (j = 0; j < 8; j++)
             {
                    Q[i*8 + j] = out4[j];
             }
       }
/**********************************************************************/
/*          End of decimation                                         */
/**********************************************************************/

       syncpoint = sync_corr(I, Q, rx_filt);
             // Determine pos of sync-word

       rx_RRC_filt(I, Q, syncpoint, rx_filt, I_bits, Q_bits, 188);
             // Root raised cosine filter

       cdma(I_bits, Q_bits, code, 16, 24, 188);
             // Multiply with the same PN code to become original signal

       demod_QPSK(I_bits, Q_bits, rx_idata);
             // demodulate the two channel into data

       compare(rx_idata, &BER);
     }
}
```

vecs.asm

See Appendix I.

reg549.h

See Appendix I.

table.h

See Appendix I.

filter_dec.h

```
/********************************************************************/
/*                                                                  */
/* Contains the filter coeffs tables for the FIR decimation filters */
/*                                                                  */
/* File:                  filter_dec.h                              */
/* Version:               1.0                                       */
/* Last update:           01-05-2000                                */
/* Creation date:         01-05-2000                                */
/* Author:                Roel Schiphorst                           */
/*                                                                  */
/********************************************************************/

#ifndef _filter_dec
#define _filter_dec

#define NH8  10           // number of filter coefs for 1st filter
#define D8   2            // decimation factor filter 1
#define NH4  35           // number of filter coefs for 2nd filter
#define D4   2            // decimation factor filter 2

#define decimation_factor 16 // total decimation factor (CIC + FIR)
#define frame_length 800*decimation_factor     //define total frame length
#define frame_length_min_1 800*decimation_factor - 1

#pragma DATA_SECTION (h8,".co_h8")      //store coeffs in a special defined
                                        // memory region
DATA h8[NH8] ={
      //filter coeffs 1st filter
2985,  //max int value (0x8000) = 32768 == 1.0
-521.
-1952,
3740,
14810,
14810,
3740,
-1952,
-521,
2985,
};

#pragma DATA_SECTION (dbuffer8,".db8")  //store delay buffer in a special
                                        //defined memory region
DATA dbuffer8[NH8];                     //define buffer
DATA *dp8 = dbuffer8;                   //define pointer to buffer

#pragma DATA_SECTION (h4,".co_h4")      //store coeffs in a special defined
                                        //memory region
DATA h4[NH4] ={
       //filter coeffs 2nd filter
38,     //max int value (0x8000) = 32768 == 1.0
0,
-90,
0,
189,
0,
-352,
0,
609,
1,
-1019,
-1,
1724,
1,
-3248,
-1,
10352,
16385,
10352,
-1,
-3248,
1,
1724,
-1,
-1019,
1,
609,
0,
-352,
0,
189,
0,
-90,
0,
38,
};

#pragma DATA_SECTION (dbuffer4,".db4")  //store delay buffer in a special
                                        //defined memory region
DATA dbuffer4[NH4];                     //define delay buffer
DATA *dp4 = dbuffer4;                   //define pointer to this buffer

#endif
```

549.cmd

```
/***********************************************************************/
/* Functions for setting up memory map of the 549 DSP processor        */
/*                                                                     */
/* File:                    549.cmd                                    */
/* Version:                 1.0                                        */
/* Last update:             01-05-2000                                 */
/* Creation date:           01-05-2000                                 */
/* Author:                  Roel Schiphorst                            */
/*                                                                     */
/***********************************************************************/
main.obj
carrier_mult.obj
cic.obj
detect.obj
demod_qpsk.obj
```

```
cdma.obj

-o receive.out

-m receive.map

-e RESET

MEMORY
{
       PAGE 0:       ROM:          origin = 00000h, length = 007FFh
                     PRAM0:        origin = 00800h, length = 0F780h
                     VECTOR:       origin = 0ff80h, length = 00080h
       PAGE 1:       REGS:         origin = 00000h, length = 00050h
                     I_O:          origin = 00050h, length = 00010h
                     DRAM0:        origin = 0x0060, length = 0x0020
                     DRAM1:        origin = 0x0080, length = 0x1f80
                     DRAM2:        origin = 0x2000, length = 0x6000
}

SECTIONS
{
    .vectors :       > VECTOR
       .text :       > PRAM0
       .cinit :      > ROM
       .bss :        > DRAM2
       .cos :        > DRAM2
       .sin :        > DRAM2
       .rcf :        > DRAM2

       /* alignment of filter coeffs and filter buffers */
       .co_h8:       > DRAM2, align(128)
       .db8   :      > DRAM2, align(128)
       .co_h4:       > DRAM2, align(128)
       .db4   :      > DRAM2, align(128)
       .stack :      > DRAM1
}
```

Appendix IV: Decimation-filter data

The receiver uses a three-stage decimation. Below the frequency responses are shown and the accompanying filter coefficients.

Decimation filter I (CIC filter)

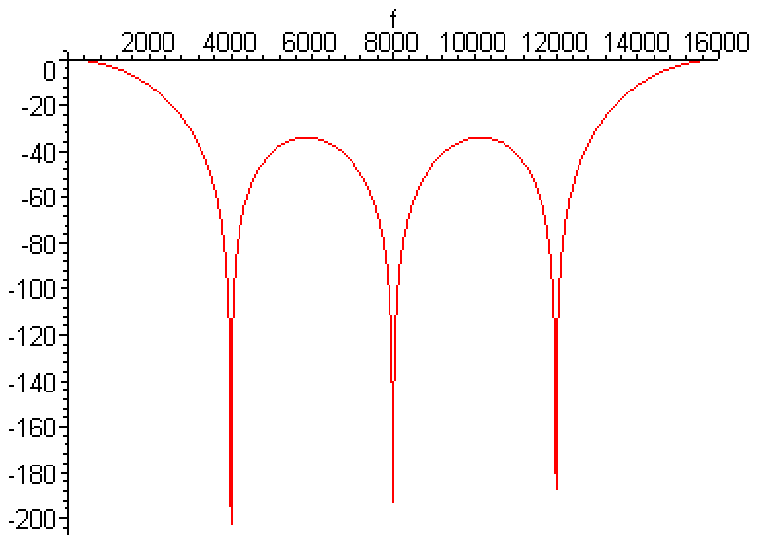
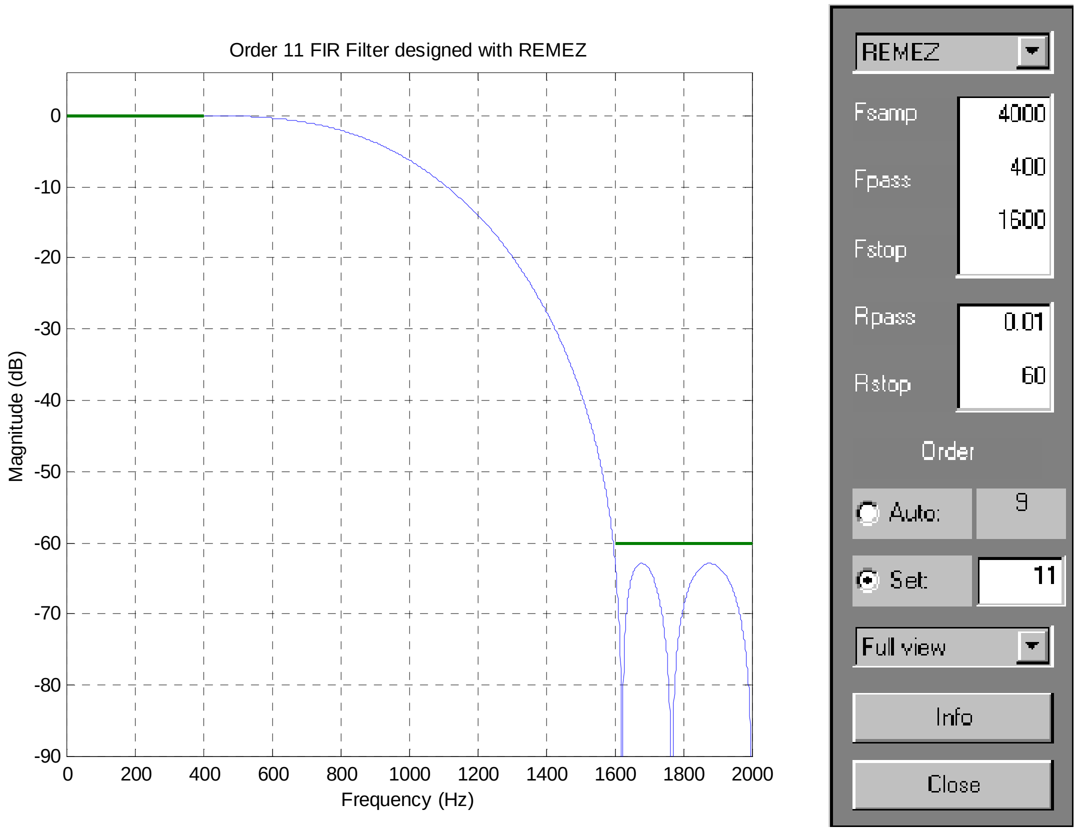


Figure 64: frequency response of decimation filter I

Decimation filter II



Fsamp 4000
Fpass 400
Fstop 1600
Rpass 0.01
Rstop 60
Order
Auto: 9
Set: 11

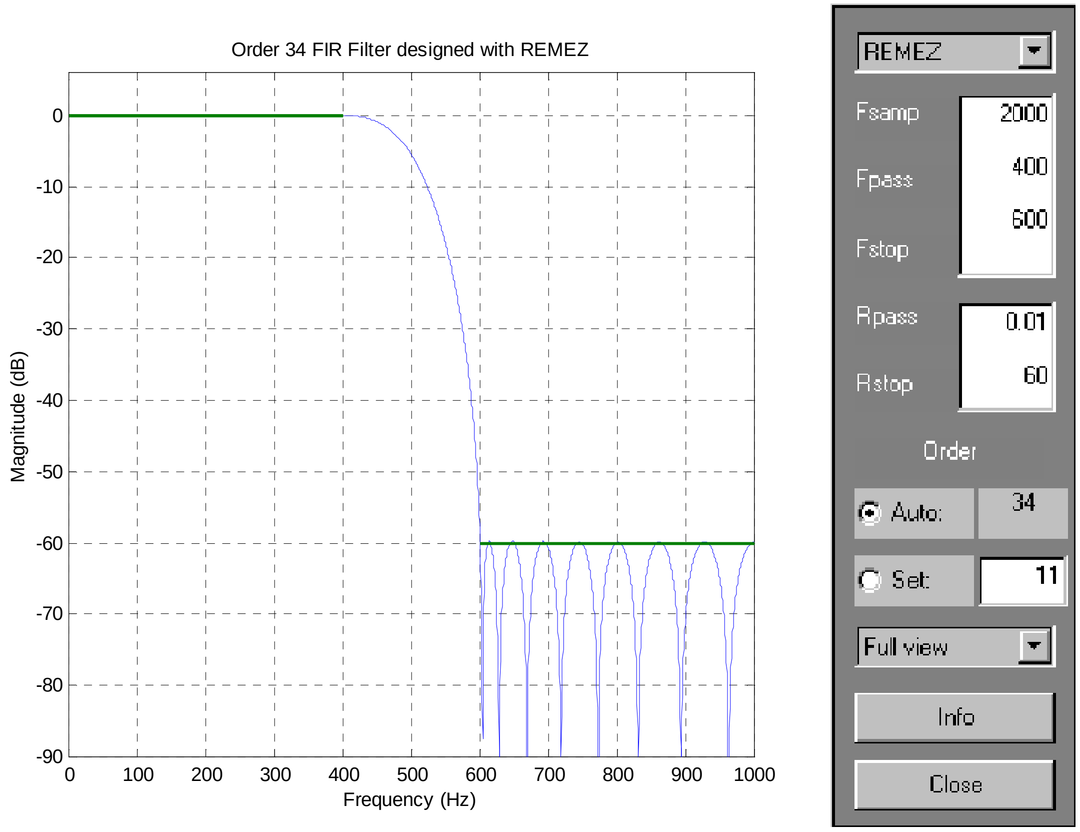
Figure 65: frequency response of decimation filter II

The filter coefficients of the decimation filter II are shown in Table 12

Coefficients	**Decimation filter II**
C0	0.00501366817478
C1	0.00893665588294
C2	-0.03164948271466
C3	-0.05402394353169
C4	0.13631696318803
C5	0.43563345229614
C6	0.43563345229614
C7	0.13631696318803
C8	-0.05402394353169
C9	-0.03164948271466
C10	0.00893665588294
C11	0.00501366817478

Table 12: filter coefficients of decimation filter II

Decimation filter III



REMEZ

Fsamp 2000

Fpass 400

Fstop 600

Rpass 0.01

Rstop 60

Order

[✓ Auto:] 34

Set: 11

Full view

Figure 66: frequency response of decimation filter III

The filter coefficients of the decimation filter III are shown in Table 13.

Coefficients	Decimation filter III
C0	0.00115231466580
C1	0.00000537777114
C2	0.00000537777114
C3	-0.00000479327194
C4	0.00576296767733
C5	0.00001184899847
C6	-0.01073431260741
C7	-0.00001277015431
C8	0.01858639414522
C9	0.00002058156598
C10	-0.03109221158338
C11	-0.00002132190202
C12	0.05259937703532
C13	0.00002796029231
C14	-0.09913078385136
C15	-0.00002627823095
C16	0.31592463163592
C17	0.50003103100212
C18	0.31592463163592
C19	-0.00002627823095
C20	-0.09913078385136
C21	0.00002796029231
C22	0.05259937703532
C23	-0.00002132190202

C24	-0.03109221158338
C25	0.00002058156598
C26	0.01858639414522
C27	-0.00001277015431
C28	-0.01073431260741
C29	0.00001184899847
C30	0.00576296767733
C31	-0.00000479327194
C32	-0.00274720845232
C33	0.00000537777114
C34	0.00115231466580

Table 13: filter coefficients of decimation filter III