Real-Time Digital Signal Processing

Implementations and Applications

Second Edition

Sen M. Kuo, Bob. H. Lee and Wenshun Tian

WILEY

Real-Time Digital Signal Processing

Implementations and Applications

Second Edition

Sen M Kuo
Northern Illinois University, USA

Bob H Lee
Ingenient Technologies Inc., USA

Wenshun Tian
UTStarcom Inc., USA

John Wiley & Sons, Ltd

Real-Time Digital Signal Processing

Second Edition

Real-Time Digital Signal Processing

Implementations and Applications

Second Edition

Sen M Kuo
Northern Illinois University, USA

Bob H Lee
Ingenient Technologies Inc., USA

Wenshun Tian
UTStarcom Inc., USA

John Wiley & Sons, Ltd

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada M9W 1L1

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Kuo, Sen M. (Sen-Maw)
Real-time digital signal processing : implementations, applications and experiments with the TMS320C55X / Sen M Kuo, Bob H Lee, Wenshun Tian. – 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-470-01495-4 (cloth)
1. Signal processing–Digital techniques. 2. Texas Instruments TMS320 series microprocessors. I. Lee, Bob H. II. Tian, Wenshun. III. Title.

TK5102 .9 .K86 2006
621.382′2-dc22 2005036660

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN-13 978-0-470-01495-0
ISBN-10 0-470-01495-4

Typeset in 9/11pt Times by TechBooks, New Delhi, India
Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Preface

In recent years, digital signal processing (DSP) has expanded beyond filtering, frequency analysis, and signal generation. More and more markets are opening up to DSP applications, where in the past, real-time signal processing was not feasible or was too expensive. Real-time signal processing using general-purpose DSP processors provides an effective way to design and implement DSP algorithms for real-world applications. However, this is very challenging work in today's engineering fields. With DSP penetrating into many practical applications, the demand for high-performance digital signal processors has expanded rapidly in recent years. Many industrial companies are currently engaged in real-time DSP research and development. Therefore, it becomes increasingly important for today's students, practicing engineers, and development researchers to master not only the theory of DSP, but also the skill of real-time DSP system design and implementation techniques.

This book provides fundamental real-time DSP principles and uses a hands-on approach to introduce DSP algorithms, system design, real-time implementation considerations, and many practical applications. This book contains many useful examples like hands-on experiment software and DSP programs using MATLAB, Simulink, C, and DSP assembly languages. Also included are various exercises for further exploring the extensions of the examples and experiments. The book uses the Texas Instruments' Code Composer Studio (CCS) with the Spectrum Digital TMS320VC5510 DSP starter kit (DSK) development tool for real-time experiments and applications.

This book emphasizes real-time DSP applications and is intended as a text for senior/graduate-level college students. The prerequisites of this book are signals and systems concepts, microprocessor architecture and programming, and basic C programming knowledge. These topics are covered at the sophomore and junior levels of electrical and computer engineering, computer science, and other related engineering curricula. This book can also serve as a desktop reference for DSP engineers, algorithm developers, and embedded system programmers to learn DSP concepts and to develop real-time DSP applications on the job. We use a practical approach that avoids numerous theoretical derivations. A list of DSP textbooks with mathematical proofs is given at the end of each chapter. Also helpful are the manuals and application notes for the TMS320C55x DSP processors from Texas Instruments at `www.ti.com`, and for the MATLAB and Simulink from Math Works at `www.mathworks.com`.

This is the second edition of the book titled 'Real-Time Digital Signal Processing: Implementations, Applications and Experiments with the TMS320C55x' by Kuo and Lee, John Wiley & Sons, Ltd. in 2001. The major changes included in the revision are:

1. To utilize the effective software development process that begins from algorithm design and verification using MATLAB and floating-point C, to finite-wordlength analysis, fixed-point C implementation and code optimization using intrinsics, assembly routines, and mixed C-and-assembly programming

on fixed-point DSP processors. This step-by-step software development and optimization process is applied to the finite-impulse response (FIR) filtering, infinite-impulse response (IIR) filtering, adaptive filtering, fast Fourier transform, and many real-life applications in Chapters 8–15.

2. To add several widely used DSP applications such as speech coding, channel coding, audio coding, image processing, signal generation and detection, echo cancelation, and noise reduction by expanding Chapter 9 of the first edition to eight new chapters with the necessary background to perform the experiments using the optimized software development process.

3. To design and analyze DSP algorithms using the most effective MATLAB graphic user interface (GUI) tools such as Signal Processing Tool (SPTool), Filter Design and Analysis Tool (FDATool), etc. These tools are powerful for filter designing, analysis, quantization, testing, and implementation.

4. To add step-by-step experiments to create CCS DSP/BIOS applications, configure the TMS320VC5510 DSK for real-time audio applications, and utilize MATLAB's Link for CCS feature to improve DSP development, debug, analyze, and test efficiencies.

5. To update experiments to include new sets of hands-on exercises and applications. Also, to update all programs using the most recent version of software and the TMS320C5510 DSK board for real-time experiments.

There are many existing DSP algorithms and applications available in MATLAB and floating-point C programs. This book provides a systematic software development process for converting these programs to fixed-point C and optimizing them for implementation on commercially available fixed-point DSP processors. To effectively illustrate real-time DSP concepts and applications, MATLAB is used for analysis and filter design, C program is used for implementing DSP algorithms, and CCS is integrated into TMS320C55x experiments and applications. To efficiently utilize the advanced DSP architecture for fast software development and maintenance, the mixing of C and assembly programs is emphasized.

This book is organized into two parts: DSP implementation and DSP application. Part I, DSP implementation (Chapters 1–7) discusses real-time DSP principles, architectures, algorithms, and implementation considerations. Chapter 1 reviews the fundamentals of real-time DSP functional blocks, DSP hardware options, fixed- and floating-point DSP devices, real-time constraints, algorithm development, selection of DSP chips, and software development. Chapter 2 introduces the architecture and assembly programming of the TMS320C55x DSP processor. Chapter 3 presents fundamental DSP concepts and practical considerations for the implementation of digital filters and algorithms on DSP hardware. Chapter 4 focuses on the design, implementation, and application of FIR filters. Digital IIR filters are covered in Chapter 5, and adaptive filters are presented in Chapter 7. The development, implementation, and application of FFT algorithms are introduced in Chapter 6.

Part II, DSP application (Chapters 8–15) introduces several popular real-world applications in signal processing that have played important roles in the realization of the systems. These selected DSP applications include signal (sinewave, noise, and multitone) generation in Chapter 8, dual-tone multifrequency detection in Chapter 9, adaptive echo cancelation in Chapter 10, speech-coding algorithms in Chapter 11, speech enhancement techniques in Chapter 12, audio coding methods in Chapter 13, error correction coding techniques in Chapter 14, and image processing fundamentals in Chapter 15.

As with any book attempting to capture the state of the art at a given time, there will certainly be updates that are necessitated by the rapidly evolving developments in this dynamic field. We are certain that this book will serve as a guide for what has already come and as an inspiration for what will follow.

Software Availability

This text utilizes various MATLAB, floating-point and fixed-point C, DSP assembly and mixed C and assembly programs for the examples, experiments, and applications. These programs along with many other programs and real-world data files are available in the companion CD. The directory structure and the subdirectory names are explained in Appendix B. The software will assist in gaining insight into the understanding and implementation of DSP algorithms, and it is required for doing experiments in the last section of each chapter. Some of these experiments involve minor modifications of the example code. By examining, studying, and modifying the example code, the software can also be used as a prototype for other practical applications. Every attempt has been made to ensure the correctness of the code. We would appreciate readers bringing to our attention (`kuo@ceet.niu.edu`) any coding errors so that we can correct, update, and post them on the website `http://www.ceet.niu.edu/faculty/kuo`.

Acknowledgments

We are grateful to Cathy Wicks and Gene Frantz of Texas Instruments, and to Naomi Fernandes and Courtney Esposito of The MathWorks for providing us with the support needed to write this book. We would like to thank several individuals at Wiley for their support on this project: Simone Taylor, Executive Commissioning Editor; Emily Bone, Assistant Editor; and Lucy Bryan, Executive Project Editor. We also thank the staff at Wiley for the final preparation of this book. Finally, we thank our families for the endless love, encouragement, patience, and understanding they have shown throughout this period.

Sen M. Kuo, **Bob H. Lee** and **Wenshun Tian**

1 Introduction to Real-Time Digital Signal Processing

Signals can be divided into three categories: continuous-time (analog) signals, discrete-time signals, and digital signals. The signals that we encounter daily are mostly analog signals. These signals are defined continuously in time, have an infinite range of amplitude values, and can be processed using analog electronics containing both active and passive circuit elements. Discrete-time signals are defined only at a particular set of time instances. Therefore, they can be represented as a sequence of numbers that have a continuous range of values. Digital signals have discrete values in both time and amplitude; thus, they can be processed by computers or microprocessors. In this book, we will present the design, implementation, and applications of digital systems for processing digital signals using digital hardware. However, the analysis usually uses discrete-time signals and systems for mathematical convenience. Therefore, we use the terms 'discrete-time' and 'digital' interchangeably.

Digital signal processing (DSP) is concerned with the digital representation of signals and the use of digital systems to analyze, modify, store, or extract information from these signals. Much research has been conducted to develop DSP algorithms and systems for real-world applications. In recent years, the rapid advancement in digital technologies has supported the implementation of sophisticated DSP algorithms for real-time applications. DSP is now used not only in areas where analog methods were used previously, but also in areas where applying analog techniques is very difficult or impossible.

There are many advantages in using digital techniques for signal processing rather than traditional analog devices, such as amplifiers, modulators, and filters. Some of the advantages of a DSP system over analog circuitry are summarized as follows:

1. *Flexibility*: Functions of a DSP system can be easily modified and upgraded with software that implements the specific applications. One can design a DSP system that can be programmed to perform a wide variety of tasks by executing different software modules. A digital electronic device can be easily upgraded in the field through the onboard memory devices (e.g., flash memory) to meet new requirements or improve its features.

2. *Reproducibility*: The performance of a DSP system can be repeated precisely from one unit to another. In addition, by using DSP techniques, digital signals such as audio and video streams can be stored, transferred, or reproduced many times without degrading the quality. By contract, analog circuits

will not have the same characteristics even if they are built following identical specifications due to analog component tolerances.

3. *Reliability*: The memory and logic of DSP hardware does not deteriorate with age. Therefore, the field performance of DSP systems will not drift with changing environmental conditions or aged electronic components as their analog counterparts do.

4. *Complexity*: DSP allows sophisticated applications such as speech recognition and image compression to be implemented with lightweight and low-power portable devices. Furthermore, there are some important signal processing algorithms such as error correcting codes, data transmission and storage, and data compression, which can only be performed using DSP systems.

With the rapid evolution in semiconductor technologies, DSP systems have a lower overall cost compared to analog systems for most applications. DSP algorithms can be developed, analyzed, and simulated using high-level language and software tools such as C/C++ and MATLAB (matrix laboratory). The performance of the algorithms can be verified using a low-cost, general-purpose computer. Therefore, a DSP system is relatively easy to design, develop, analyze, simulate, test, and maintain.

There are some limitations associated with DSP. For instance, the bandwidth of a DSP system is limited by the sampling rate and hardware peripherals. Also, DSP algorithms are implemented using a fixed number of bits with a limited precision and dynamic range (the ratio between the largest and smallest numbers that can be represented), which results in quantization and arithmetic errors. Thus, the system performance might be different from the theoretical expectation.

1.1 Basic Elements of Real-Time DSP Systems

There are two types of DSP applications: non-real-time and real-time. Non-real-time signal processing involves manipulating signals that have already been collected in digital forms. This may or may not represent a current action, and the requirement for the processing result is not a function of real time. Real-time signal processing places stringent demands on DSP hardware and software designs to complete predefined tasks within a certain time frame. This chapter reviews the fundamental functional blocks of real-time DSP systems.

The basic functional blocks of DSP systems are illustrated in Figure 1.1, where a real-world analog signal is converted to a digital signal, processed by DSP hardware, and converted back into an analog

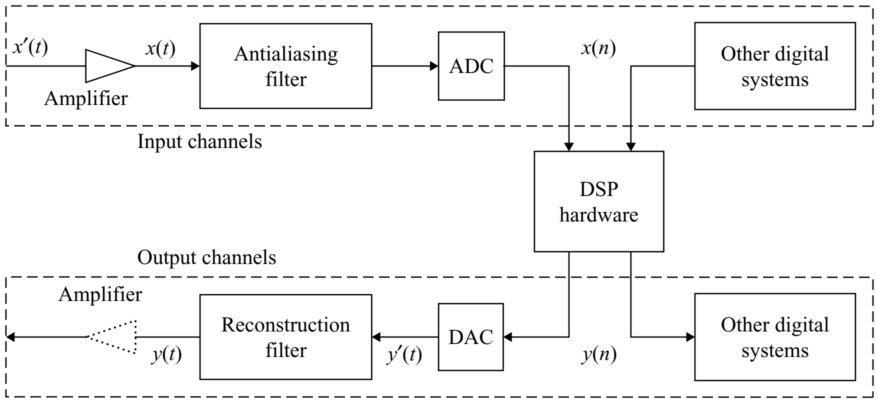


Figure 1.1 Basic functional block diagram of a real-time DSP system

signal. Each of the functional blocks in Figure 1.1 will be introduced in the subsequent sections. For some applications, the input signal may already be in digital form and/or the output data may not need to be converted to an analog signal. For example, the processed digital information may be stored in computer memory for later use, or it may be displayed graphically. In other applications, the DSP system may be required to generate signals digitally, such as speech synthesis used for computerized services or pseudo-random number generators for CDMA (code division multiple access) wireless communication systems.

1.2 Analog Interface

In this book, a time-domain signal is denoted with a lowercase letter. For example, $x(t)$ in Figure 1.1 is used to name an analog signal of x which is a function of time t. The time variable t and the amplitude of $x(t)$ take on a continuum of values between $-\infty$ and ∞. For this reason we say $x(t)$ is a continuous-time signal. The signals $x(n)$ and $y(n)$ in Figure 1.1 depict digital signals which are only meaningful at time instant n. In this section, we first discuss how to convert analog signals into digital signals so that they can be processed using DSP hardware. The process of converting an analog signal to a digital signal is called the analog-to-digital conversion, usually performed by an analog-to-digital converter (ADC).

The purpose of signal conversion is to prepare real-world analog signals for processing by digital hardware. As shown in Figure 1.1, the analog signal $x'(t)$ is picked up by an appropriate electronic sensor that converts pressure, temperature, or sound into electrical signals. For example, a microphone can be used to collect sound signals. The sensor signal $x'(t)$ is amplified by an amplifier with gain value g. The amplified signal is

$$x(t) = gx'(t). \tag{1.1}$$

The gain value g is determined such that $x(t)$ has a dynamic range that matches the ADC used by the system. If the peak-to-peak voltage range of the ADC is ± 5 V, then g may be set so that the amplitude of signal $x(t)$ to the ADC is within ± 5 V. In practice, it is very difficult to set an appropriate fixed gain because the level of $x'(t)$ may be unknown and changing with time, especially for signals with a larger dynamic range such as human speech.

Once the input digital signal has been processed by the DSP hardware, the result $y(n)$ is still in digital form. In many DSP applications, we need to reconstruct the analog signal after the completion of digital processing. We must convert the digital signal $y(n)$ back to the analog signal $y(t)$ before it is applied to an appropriated analog device. This process is called the digital-to-analog conversion, typically performed by a digital-to-analog converter (DAC). One example would be audio CD (compact disc) players, for which the audio music signals are stored in digital form on CDs. A CD player reads the encoded digital audio signals from the disk and reconstructs the corresponding analog waveform for playback via loudspeakers.

The system shown in Figure 1.1 is a real-time system if the signal to the ADC is continuously sampled and the ADC presents a new sample to the DSP hardware at the same rate. In order to maintain real-time operation, the DSP hardware must perform all required operations within the fixed time period, and present an output sample to the DAC before the arrival of the next sample from the ADC.

1.2.1 Sampling

As shown in Figure 1.1, the ADC converts the analog signal $x(t)$ into the digital signal $x(n)$. Analog-to-digital conversion, commonly referred as digitization, consists of the sampling (digitization in time) and quantization (digitization in amplitude) processes as illustrated in Figure 1.2. The sampling process depicts an analog signal as a sequence of values. The basic sampling function can be carried out with an ideal 'sample-and-hold' circuit, which maintains the sampled signal level until the next sample is taken.

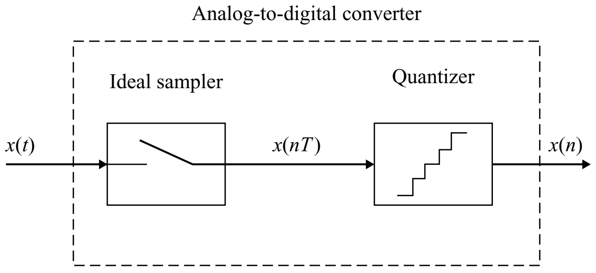


Figure 1.2 Block diagram of an ADC

Quantization process approximates a waveform by assigning a number for each sample. Therefore, the analog-to-digital conversion will perform the following steps:

1. The bandlimited signal $x(t)$ is sampled at uniformly spaced instants of time nT, where n is a positive integer and T is the sampling period in seconds. This sampling process converts an analog signal into a discrete-time signal $x(nT)$ with continuous amplitude value.

2. The amplitude of each discrete-time sample is quantized into one of the 2^B levels, where B is the number of bits that the ADC has to represent for each sample. The discrete amplitude levels are represented (or encoded) into distinct binary words $x(n)$ with a fixed wordlength B.

The reason for making this distinction is that these processes introduce different distortions. The sampling process brings in aliasing or folding distortion, while the encoding process results in quantization noise. As shown in Figure 1.2, the sampler and quantizer are integrated on the same chip. However, high-speed ADCs typically require an external sample-and-hold device.

An ideal sampler can be considered as a switch that periodically opens and closes every T s (seconds). The sampling period is defined as

$$T = \frac{1}{f_s}, \tag{1.2}$$

where f_s is the sampling frequency (or sampling rate) in hertz (or cycles per second). The intermediate signal $x(nT)$ is a discrete-time signal with a continuous value (a number with infinite precision) at discrete time nT, $n = 0, 1, \ldots, \infty$, as illustrated in Figure 1.3. The analog signal $x(t)$ is continuous in both time and amplitude. The sampled discrete-time signal $x(nT)$ is continuous in amplitude, but is defined only at discrete sampling instants $t = nT$.

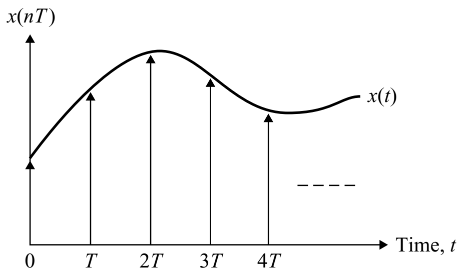


Figure 1.3 Example of analog signal $x(t)$ and discrete-time signal $x(nT)$

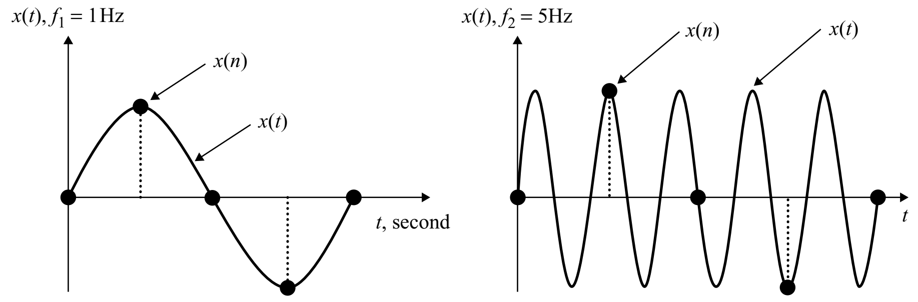
In order to represent an analog signal $x(t)$ by a discrete-time signal $x(nT)$ accurately, the sampling frequency f_s must be at least twice the maximum frequency component (f_M) in the analog signal $x(t)$. That is,

$$f_s \geq 2f_M, \tag{1.3}$$

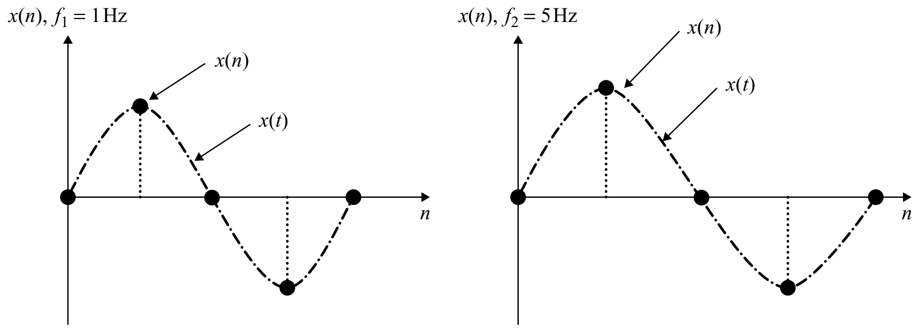
where f_M is also called the bandwidth of the signal $x(t)$. This is Shannon's sampling theorem, which states that when the sampling frequency is greater than twice of the highest frequency component contained in the analog signal, the original signal $x(t)$ can be perfectly reconstructed from the corresponding discrete-time signal $x(nT)$.

The minimum sampling rate $f_s = 2f_M$ is called the Nyquist rate. The frequency $f_N = f_s/2$ is called the Nyquist frequency or folding frequency. The frequency interval $[-f_s/2,\ f_s/2]$ is called the Nyquist interval. When an analog signal is sampled at f_s, frequency components higher than $f_s/2$ fold back into the frequency range $[0,\ f_s/2]$. The folded back frequency components overlap with the original frequency components in the same range. Therefore, the original analog signal cannot be recovered from the sampled data. This undesired effect is known as aliasing.

Example 1.1: Consider two sinewaves of frequencies $f_1 = 1$ Hz and $f_2 = 5$ Hz that are sampled at $f_s = 4$ Hz, rather than at 10 Hz according to the sampling theorem. The analog waveforms are illustrated in Figure 1.4(a), while their digital samples and reconstructed waveforms are illustrated



(a) Original analog waveforms and digital samplses for $f_1 = 1$ Hz and $f_2 = 5$ Hz.



(b) Digital samples for $f_1 = 1$ Hz and $f_2 = 5$ Hz and reconstructed waveforms.

Figure 1.4 Example of the aliasing phenomenon: (a) original analog waveforms and digital samples for $f_1 = 1$ Hz and $f_2 = 5$ Hz; (b) digital samples of $f_1 = 1$ Hz and $f_2 = 5$ Hz and reconstructed waveforms

in Figure 1.4(b). As shown in the figures, we can reconstruct the original waveform from the digital samples for the sinewave of frequency $f_1 = 1$ Hz. However, for the original sinewave of frequency $f_2 = 5$ Hz, the reconstructed signal is identical to the sinewave of frequency 1 Hz. Therefore, f_1 and f_2 are said to be aliased to one another, i.e., they cannot be distinguished by their discrete-time samples.

Note that the sampling theorem assumes that the signal is bandlimited. For most practical applications, the analog signal $x(t)$ may have significant energies outside the highest frequency of interest, or may contain noise with a wider bandwidth. In some cases, the sampling rate is predetermined by a given application. For example, most voice communication systems use an 8 kHz sampling rate. Unfortunately, the frequency components in a speech signal can be much higher than 4 kHz. To guarantee that the sampling theorem defined in Equation (1.3) can be fulfilled, we must block the frequency components that are above the Nyquist frequency. This can be done by using an antialiasing filter, which is an analog lowpass filter with the cutoff frequency

$$f_c \leq \frac{f_s}{2}. \tag{1.4}$$

Ideally, an antialiasing filter should remove all frequency components above the Nyquist frequency. In many practical systems, a bandpass filter is preferred to remove all frequency components above the Nyquist frequency, as well as to prevent undesired DC offset, 60 Hz hum, or other low-frequency noises. A bandpass filter with passband from 300 to 3200 Hz can often be found in telecommunication systems.

Since antialiasing filters used in real-world applications are not ideal filters, they cannot completely remove all frequency components outside the Nyquist interval. In addition, since the phase response of the analog filter may not be linear, the phase of the signal will not be shifted by amounts proportional to their frequencies. In general, a lowpass (or bandpass) filter with steeper roll-off will introduce more phase distortion. Higher sampling rates allow simple low-cost antialiasing filter with minimum phase distortion to be used. This technique is known as oversampling, which is widely used in audio applications.

Example 1.2: The range of sampling rate required by DSP systems is large, from approximately 1 GHz in radar to 1 Hz in instrumentation. Given a sampling rate for a specific application, the sampling period can be determined by (1.2). Some real-world applications use the following sampling frequencies and periods:

1. In International Telecommunication Union (ITU) speech compression standards, the sampling rate of ITU-T G.729 and G.723.1 is $f_s = 8$ kHz, thus the sampling period $T = 1/8000$ s $=$ 125 μs. Note that 1 μs $= 10^{-6}$ s.

2. Wideband telecommunication systems, such as ITU-T G.722, use a sampling rate of $f_s =$ 16 kHz, thus $T = 1/16\,000$ s $= 62.5$ μs.

3. In audio CDs, the sampling rate is $f_s = 44.1$ kHz, thus $T = 1/44\,100$ s $= 22.676$ μs.

4. High-fidelity audio systems, such as MPEG-2 (moving picture experts group) AAC (advanced audio coding) standard, MP3 (MPEG layer 3) audio compression standard, and Dolby AC-3, have a sampling rate of $f_s = 48$ kHz, and thus $T = 1/48\,000$ s $= 20.833$ μs. The sampling rate for MPEG-2 AAC can be as high as 96 kHz.

The speech compression algorithms will be discussed in Chapter 11 and the audio coding techniques will be introduced in Chapter 13.

1.2.2 Quantization and Encoding

In previous sections, we assumed that the sample values $x(nT)$ are represented exactly with an infinite number of bits (i.e., $B \rightarrow \infty$). We now discuss a method of representing the sampled discrete-time signal $x(nT)$ as a binary number with finite number of bits. This is the quantization and encoding process. If the wordlength of an ADC is B bits, there are 2^B different values (levels) that can be used to represent a sample. If $x(n)$ lies between two quantization levels, it will be either rounded or truncated. Rounding replaces $x(n)$ by the value of the nearest quantization level, while truncation replaces $x(n)$ by the value of the level below it. Since rounding produces less biased representation of the analog values, it is widely used by ADCs. Therefore, quantization is a process that represents an analog-valued sample $x(nT)$ with its nearest level that corresponds to the digital signal $x(n)$.

We can use 2 bits to define four equally spaced levels (00, 01, 10, and 11) to classify the signal into the four subranges as illustrated in Figure 1.5. In this figure, the symbol ‘o’ represents the discrete-time signal $x(nT)$, and the symbol ‘•’ represents the digital signal $x(n)$. The spacing between two consecutive quantization levels is called the quantization width, step, or resolution. If the spacing between these levels is the same, then we have a uniform quantizer. For the uniform quantization, the resolution is given by dividing a full-scale range with the number of quantization levels, 2^B.

In Figure 1.5, the difference between the quantized number and the original value is defined as the quantization error, which appears as noise in the converter output. It is also called the quantization noise, which is assumed to be random variables that are uniformly distributed. If a B-bit quantizer is used, the signal-to-quantization-noise ratio (SQNR) is approximated by (will be derived in Chapter 3)

$$\text{SQNR} \approx 6B \text{ dB}. \tag{1.5}$$

This is a theoretical maximum. In practice, the achievable SQNR will be less than this value due to imperfections in the fabrication of converters. However, Equation (1.5) still provides a simple guideline for determining the required bits for a given application. For each additional bit, a digital signal will have about 6-dB gain in SQNR. The problems of quantization and their solutions will be further discussed in Chapter 3.

Example 1.3: If the input signal varies between 0 and 5 V, we have the resolutions and SQNRs for the following commonly used data converters:

1. An 8-bit ADC with 256 (2^8) levels can only provide 19.5 mV resolution and 48 dB SQNR.
2. A 12-bit ADC has 4096 (2^{12}) levels of 1.22 mV resolution, and provides 72 dB SQNR.

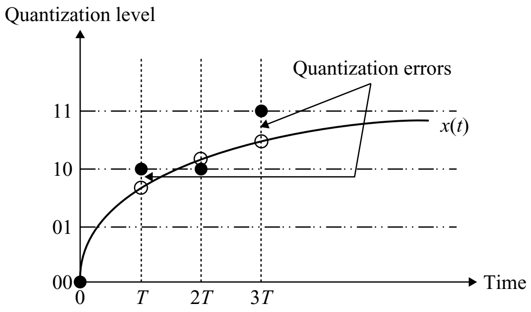


Figure 1.5 Digital samples using a 2-bit quantizer

3. A 16-bit ADC has 65 536 (2^{16}) levels, and thus provides 76.294 μV resolution with 96 dB SQNR.

Obviously, with more quantization levels, one can represent analog signals more accurately.

The dynamic range of speech signals is very large. If the uniform quantization scheme shown in Figure 1.5 can adequately represent loud sounds, most of the softer sounds may be pushed into the same small value. This means that soft sounds may not be distinguishable. To solve this problem, a quantizer whose quantization level varies according to the signal amplitude can be used. In practice, the nonuniform quantizer uses uniform levels, but the input signal is compressed first using a logarithm function. That is, the logarithm-scaled signal, rather than the original input signal itself, will be quantized. The compressed signal can be reconstructed by expanding it. The process of compression and expansion is called companding (compressing and expanding). For example, the ITU-T G.711 μ-law (used in North America and parts of Northeast Asia) and A-law (used in Europe and most of the rest of the world) companding schemes are used in most digital telecommunications. The A-law companding scheme gives slightly better performance at high signal levels, while the μ-law is better at low levels.

As shown in Figure 1.1, the input signal to DSP hardware may be a digital signal from other DSP systems. In this case, the sampling rate of digital signals from other digital systems must be known. The signal processing techniques called interpolation and decimation can be used to increase or decrease the existing digital signals' sampling rates. Sampling rate changes may be required in many multirate DSP systems such as interconnecting DSP systems that are operated at different rates.

1.2.3 Smoothing Filters

Most commercial DACs are zero-order-hold devices, meaning they convert the input binary number to the corresponding voltage level and then hold that value for T s until the next sampling instant. Therefore, the DAC produces a staircase-shape analog waveform $y'(t)$ as shown by the solid line in Figure 1.6, which is a rectangular waveform with amplitude equal to the input value and duration of T s. Obviously, this staircase output contains some high-frequency components due to an abrupt change in signal levels. The reconstruction or smoothing filter shown in Figure 1.1 smoothes the staircase-like analog signal generated by the DAC. This lowpass filtering has the effect of rounding off the corners (high-frequency components) of the staircase signal and making it smoother, which is shown as a dotted line in Figure 1.6. This analog lowpass filter may have the same specifications as the antialiasing filter with cutoff frequency $f_c \leq f_s/2$. High-quality DSP applications, such as professional digital audio, require the use

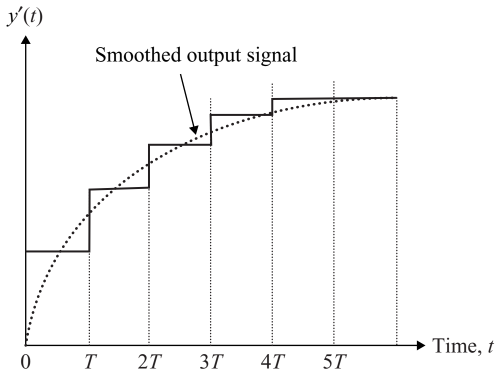


Figure 1.6 Staircase waveform generated by a DAC

of reconstruction filters with very stringent specifications. To reduce the cost of using high-quality analog filter, the oversampling technique can be adopted to allow the use of low-cost filter with slower roll off.

1.2.4 Data Converters

There are two schemes of connecting ADC and DAC to DSP processors: serial and parallel. A parallel converter receives or transmits all the B bits in one pass, while the serial converters receive or transmit B bits in a serial bit stream. Parallel converters must be attached to the DSP processor's external address and data buses, which are also attached to many different types of devices. Serial converters can be connected directly to the built-in serial ports of DSP processors. This is why many practical DSP systems use serial ADCs and DACs.

Many applications use a single-chip device called an analog interface chip (AIC) or a coder/decoder (CODEC), which integrates an antialiasing filter, an ADC, a DAC, and a reconstruction filter all on a single piece of silicon. In this book, we will use Texas Instruments' TLV320AIC23 (AIC23) chip on the DSP starter kit (DSK) for real-time experiments. Typical applications using CODEC include modems, speech systems, audio systems, and industrial controllers. Many standards that specify the nature of the CODEC have evolved for the purposes of switching and transmission. Some CODECs use a logarithmic quantizer, i.e., A-law or μ-law, which must be converted into a linear format for processing. DSP processors implement the required format conversion (compression or expansion) in hardware, or in software by using a lookup table or calculation.

The most popular commercially available ADCs are successive approximation, dual slope, flash, and sigma–delta. The successive-approximation ADC produces a B-bit output in B clock cycles by comparing the input waveform with the output of a DAC. This device uses a successive-approximation register to split the voltage range in half to determine where the input signal lies. According to the comparator result, 1 bit will be set or reset each time. This process proceeds from the most significant bit to the least significant bit. The successive-approximation type of ADC is generally accurate and fast at a relatively low cost. However, its ability to follow changes in the input signal is limited by its internal clock rate, and so it may be slow to respond to sudden changes in the input signal.

The dual-slope ADC uses an integrator connected to the input voltage and a reference voltage. The integrator starts at zero condition, and it is charged for a limited time. The integrator is then switched to a known negative reference voltage and charged in the opposite direction until it reaches zero volts again. Simultaneously, a digital counter starts to record the clock cycles. The number of counts required for the integrator output voltage to return to zero is directly proportional to the input voltage. This technique is very precise and can produce ADCs with high resolution. Since the integrator is used for input and reference voltages, any small variations in temperature and aging of components have little or no effect on these types of converters. However, they are very slow and generally cost more than successive-approximation ADCs.

A voltage divider made by resistors is used to set reference voltages at the flash ADC inputs. The major advantage of a flash ADC is its speed of conversion, which is simply the propagation delay of the comparators. Unfortunately, a B-bit ADC requires $(2^B - 1)$ expensive comparators and laser-trimmed resistors. Therefore, commercially available flash ADCs usually have lower bits.

Sigma–delta ADCs use oversampling and quantization noise shaping to trade the quantizer resolution with sampling rate. The block diagram of a sigma–delta ADC is illustrated in Figure 1.7, which uses a 1-bit quantizer with a very high sampling rate. Thus, the requirements for an antialiasing filter are significantly relaxed (i.e., the lower roll-off rate). A low-order antialiasing filter requires simple low-cost analog circuitry and is much easier to build and maintain. In the process of quantization, the resulting noise power is spread evenly over the entire spectrum. The quantization noise beyond the required spectrum range can be filtered out using an appropriate digital lowpass filter. As a result, the noise power within the frequency band of interest is lower. In order to match the sampling frequency with the system and increase its resolution, a decimator is used. The advantages of sigma–delta

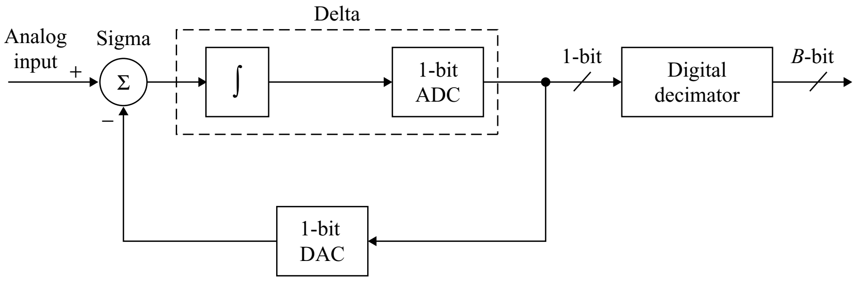


Figure 1.7 A conceptual sigma–delta ADC block diagram

ADCs are high resolution and good noise characteristics at a competitive price because they use digital filters.

Example 1.4: In this book, we use the TMS320VC5510 DSK for real-time experiments. The C5510 DSK uses an AIC23 stereo CODEC for input and output of audio signals. The ADCs and DACs within the AIC23 use the multi-bit sigma–delta technology with integrated oversampling digital interpolation filters. It supports data wordlengths of 16, 20, 24, and 32 bits, with sampling rates from 8 to 96 kHz including the CD standard 44.1 kHz. Integrated analog features consist of stereo-line inputs and a stereo headphone amplifier with analog volume control. Its power management allows selective shutdown of CODEC functions, thus extending battery life in portable applications such as portable audio and video players and digital recorders.

1.3 DSP Hardware

DSP systems are required to perform intensive arithmetic operations such as multiplication and addition. These tasks may be implemented on microprocessors, microcontrollers, digital signal processors, or custom integrated circuits. The selection of appropriate hardware is determined by the applications, cost, or a combination of both. This section introduces different digital hardware implementations for DSP applications.

1.3.1 DSP Hardware Options

As shown in Figure 1.1, the processing of the digital signal $x(n)$ is performed using the DSP hardware. Although it is possible to implement DSP algorithms on any digital computer, the real applications determine the optimum hardware platform. Five hardware platforms are widely used for DSP systems:

1. special-purpose (custom) chips such as application-specific integrated circuits (ASIC);
2. field-programmable gate arrays (FPGA);
3. general-purpose microprocessors or microcontrollers (μP/μC);
4. general-purpose digital signal processors (DSP processors); and
5. DSP processors with application-specific hardware (HW) accelerators.

The hardware characteristics of these options are summarized in Table 1.1.

Table 1.1 Summary of DSP hardware implementations

	ASIC	FPGA	μP/μC	DSP processor	DSP processors with HW accelerators
Flexibility	None	Limited	High	High	Medium
Design time	Long	Medium	Short	Short	Short
Power consumption	Low	Low–medium	Medium–high	Low–medium	Low–medium
Performance	High	High	Low–medium	Medium–high	High
Development cost	High	Medium	Low	Low	Low
Production cost	Low	Low–medium	Medium–high	Low–medium	Medium

ASIC devices are usually designed for specific tasks that require a lot of computations such as digital subscriber loop (DSL) modems, or high-volume products that use mature algorithms such as fast Fourier transform and Reed–Solomon codes. These devices are able to perform their limited functions much faster than general-purpose processors because of their dedicated architecture. These application-specific products enable the use of high-speed functions optimized in hardware, but they are deficient in the programmability to modify the algorithms and functions. They are suitable for implementing well-defined and well-tested DSP algorithms for high-volume products, or applications demanding extremely high speeds that can be achieved only by ASICs. Recently, the availability of core modules for some common DSP functions has simplified the ASIC design tasks, but the cost of prototyping an ASIC device, a longer design cycle, and the lack of standard development tools support and reprogramming flexibility sometimes outweigh their benefits.

FPGAs have been used in DSP applications for years as glue logics, bus bridges, and peripherals for reducing system costs and affording a higher level of system integration. Recently, FPGAs have been gaining considerable attention in high-performance DSP applications, and are emerging as coprocessors for standard DSP processors that need specific accelerators. In these cases, FPGAs work in conjunction with DSP processors for integrating pre- and postprocessing functions. FPGAs provide tremendous computational power by using highly parallel architectures for very high performance. These devices are hardware reconfigurable, thus allowing the system designer to optimize the hardware architectures for implementing algorithms that require higher performance and lower production cost. In addition, the designer can implement high-performance complex DSP functions in a small fraction of the total device, and use the rest to implement system logic or interface functions, resulting in both lower costs and higher system integration.

Example 1.5: There are four major FPGA families that are targeted for DSP systems: Cyclone and Stratix from Altera, and Virtex and Spartan from Xilinx. The Xilinx Spartan-3 FPGA family (introduced in 2003) uses 90-nm manufacturing technique to achieve low silicon die costs. To support DSP functions in an area-efficient manner, Spartan-3 includes the following features:

- embedded 18×18 multipliers;
- distributed RAM for local storage of DSP coefficients;
- 16-bit shift register for capturing high-speed data; and
- large block RAM for buffers.

The current Spartan-3 family includes XC3S50, S200, S400, S1000, and S1500 devices. With the aid of Xilinx System Generation for DSP, a tool used to port MATLAB Simulink model to Xilinx hardware model, a system designer can model, simulate, and verify the DSP algorithms on the target hardware under the Simulink environment.

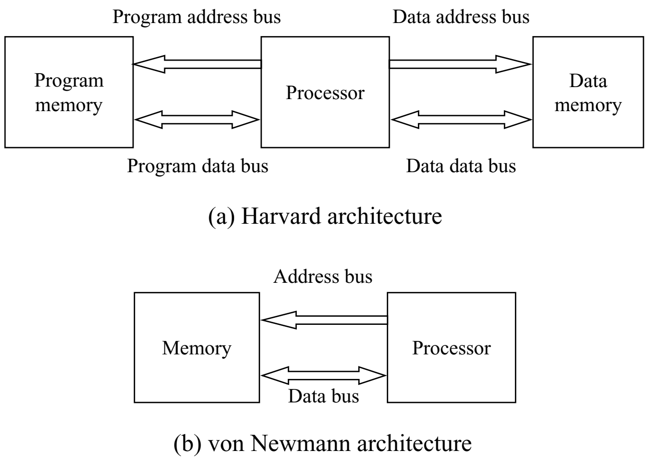


Figure 1.8 Different memory architectures: (a) Harvard architecture; (b) von Newmann architecture

General-purpose μP/μC becomes faster and increasingly able to handle some DSP applications. Many electronic products are currently designed using these processors. For example, automotive controllers use microcontrollers for engine, brake, and suspension control. If a DSP application is added to an existing product that already contains a μP/μC, it is desired to add the new functions in software without requiring an additional DSP processor. For example, Intel has adopted a native signal processing initiative that uses the host processor in computers to perform audio coding and decoding, sound synthesis, and so on. Software development tools for μP/μC devices are generally more sophisticated and powerful than those available for DSP processors, thus easing development for some applications that are less demanding on the performance and power consumption of processors.

General architectures of μP/μC fall into two categories: Harvard architecture and von Neumann architecture. As illustrated in Figure 1.8(a), Harvard architecture has a separate memory space for the program and the data, so that both memories can be accessed simultaneously. The von Neumann architecture assumes that there is no intrinsic difference between the instructions and the data, as illustrated in Figure 1.8(b). Operations such as add, move, and subtract are easy to perform on μPs/μCs. However, complex instructions such as multiplication and division are slow since they need a series of shift, addition, or subtraction operations. These devices do not have the architecture or the on-chip facilities required for efficient DSP operations. Their real-time DSP performance does not compare well with even the cheaper general-purpose DSP processors, and they would not be a cost-effective or power-efficient solution for many DSP applications.

Example 1.6: Microcontrollers such as Intel 8081 and Freescale 68HC11 are typically used in industrial process control applications, in which I/O capability (serial/parallel interfaces, timers, and interrupts) and control are more important than speed of performing functions such as multiplication and addition. Microprocessors such as Pentium, PowerPC, and ARM are basically single-chip processors that require additional circuitry to improve the computation capability. Microprocessor instruction sets can be either complex instruction set computer (CISC) such as Pentium or reduced instruction set computer (RISC) such as ARM. The CISC processor includes instructions for basic processor operations, plus some highly sophisticated instructions for specific functions. The RISC processor uses hardwired simpler instructions such as LOAD and STORE to execute in a single clock cycle.

It is important to note that some microprocessors such as Pentium add multimedia extension (MMX) and streaming single-instruction, multiple-data (SIMD) extension to support DSP operations. They can run in high speed (>3 GHz), provide single-cycle multiplication and arithmetic operations, have good memory bandwidth, and have many supporting tools and software available for easing development.

A DSP processor is basically a microprocessor optimized for processing repetitive numerically intensive operations at high rates. DSP processors with architectures and instruction sets specifically designed for DSP applications are manufactured by Texas Instruments, Freescale, Agere, Analog Devices, and many others. The rapid growth and the exploitation of DSP technology is not a surprise, considering the commercial advantages in terms of the fast, flexible, low power consumption, and potentially low-cost design capabilities offered by these devices. In comparison to ASIC and FPGA solutions, DSP processors have advantages in easing development and being reprogrammable in the field to allow a product feature upgrade or bug fix. They are often more cost-effective than custom hardware such as ASIC and FPGA, especially for low-volume applications. In comparison to the general-purpose μP/μC, DSP processors have better speed, better energy efficiency, and lower cost.

In many practical applications, designers are facing challenges of implementing complex algorithms that require more processing power than the DSP processors in use are capable of providing. For example, multimedia on wireless and portable devices requires efficient multimedia compression algorithms. The study of most prevalent imaging coding/decoding algorithms shows some DSP functions used for multimedia compression algorithms that account for approximately 80 % of the processing load. These common functions are discrete cosine transform (DCT), inverse DCT, pixel interpolation, motion estimation, and quantization, etc. The hardware extension or accelerator lets the DSP processor achieve high-bandwidth performance for applications such as streaming video and interactive gaming on a single device. The TMS320C5510 DSP used by this book consists of the hardware extensions that are specifically designed to support multimedia applications. In addition, Altera has also added the hardware accelerator into its FPGA as coprocessors to enhance the DSP processing abilities.

Today, DSP processors have become the foundation of many new markets beyond the traditional signal processing areas for technologies and innovations in motor and motion control, automotive systems, home appliances, consumer electronics, and vast range of communication systems and devices. These general-purpose-programmable DSP processors are supported by integrated software development tools that include C compilers, assemblers, optimizers, linkers, debuggers, simulators, and emulators. In this book, we use Texas Instruments' TMS320C55x for hands-on experiments. This high-performance and ultralow power consumption DSP processor will be introduced in Chapter 2. In the following section, we will briefly introduce some widely used DSP processors.

1.3.2 DSP Processors

In 1979, Intel introduced the 2920, a 25-bit integer processor with a 400 ns instruction cycle and a 25-bit arithmetic-logic unit (ALU) for DSP applications. In 1982, Texas Instruments introduced the TMS32010, a 16-bit fixed-point processor with a 16×16 hardware multiplier and a 32-bit ALU and accumulator. This first commercially successful DSP processor was followed by the development of faster products and floating-point processors. The performance and price range among DSP processors vary widely. Today, dozens of DSP processor families are commercially available. Table 1.2 summarizes some of the most popular DSP processors.

In the low-end and low-cost group are Texas Instruments' TMS320C2000 (C24x and C28x) family, Analog Devices' ADSP-218x family, and Freescale's DSP568xx family. These conventional DSP processors include hardware multiplier and shifters, execute one instruction per clock cycle, and use the complex instructions that perform multiple operations such as multiply, accumulate, and update address

Table 1.2 Current commercially available DSP processors

Vendor	Family	Arithmetic type	Clock speed
Texas instruments	TMS320C24x	Fixed-point	40 MHz
	TMS320C28x	Fixed-point	150 MHz
	TMS320C54x	Fixed-point	160 MHz
	TMS320C55x	Fixed-point	300 MHz
	TMS320C62x	Fixed-point	300 MHz
	TMS320C64x	Fixed-point	1 GHz
	TMS320C67x	Floating-point	300 MHz
Analog devices	ADSP-218x	Fixed-point	80 MHz
	ADSP-219x	Fixed-point	160 MHz
	ADSP-2126x	Floating-point	200 MHz
	ADSP-2136x	Floating-point	333 MHz
	ADSP-BF5xx	Fixed-point	750 MHz
	ADSP-TS20x	Fixed/Floating	600 MHz
Freescale	DSP56300	Fixed, 24-bit	275 MHz
	DSP568xx	Fixed-point	40 MHz
	DSP5685x	Fixed-point	120 MHz
	MSC71xx	Fixed-point	200 MHz
	MSC81xx	Fixed-point	400 MHz
Agere	DSP1641x	Fixed-point	285 MHz

Source: Adapted from [11]

pointers. They provide good performance with modest power consumption and memory usage, thus are widely used in automotives, appliances, hard disk drives, modems, and consumer electronics. For example, the TMS320C2000 and DSP568xx families are optimized for control applications, such as motor and automobile control, by integrating many microcontroller features and peripherals on the chip.

The midrange processor group includes Texas Instruments' TMS320C5000 (C54x and C55x), Analog Devices' ADSP219x and ADSP-BF5xx, and Freescale's DSP563xx. These enhanced processors achieve higher performance through a combination of increased clock rates and more advanced architectures. These families often include deeper pipelines, instruction cache, complex instruction words, multiple data buses (to access several data words per clock cycle), additional hardware accelerators, and parallel execution units to allow more operations to be executed in parallel. For example, the TMS320C55x has two multiply–accumulate (MAC) units. These midrange processors provide better performance with lower power consumption, thus are typically used in portable applications such as cellular phones and wireless devices, digital cameras, audio and video players, and digital hearing aids.

These conventional and enhanced DSP processors have the following features for common DSP algorithms such as filtering:

- Fast MAC units – The multiply–add or multiply–accumulate operation is required in most DSP functions including filtering, fast Fourier transform, and correlation. To perform the MAC operation efficiently, DSP processors integrate the multiplier and accumulator into the same data path to complete the MAC operation in single instruction cycle.

- Multiple memory accesses – Most DSP processors adopted modified Harvard architectures that keep the program memory and data memory separate to allow simultaneous fetching of instruction and data. In order to support simultaneous access of multiple data words, the DSP processors provide multiple on-chip buses, independent memory banks, and on-chip dual-access data memory.

- Special addressing modes – DSP processors often incorporate dedicated data-address generation units for generating data addresses in parallel with the execution of instruction. These units usually support circular addressing and bit-reversed addressing for some specific algorithms.
- Special program control – Most DSP processors provide zero-overhead looping, which allows the programmer to implement a loop without extra clock cycles for updating and testing loop counters, or branching back to the top of loop.
- Optimize instruction set – DSP processors provide special instructions that support the computational intensive DSP algorithms. For example, the TMS320C5000 processors support compare-select instructions for fast Viterbi decoding, which will be discussed in Chapter 14.
- Effective peripheral interface – DSP processors usually incorporate high-performance serial and parallel input/output (I/O) interfaces to other devices such as ADC and DAC. They provide streamlined I/O handling mechanisms such as buffered serial ports, direct memory access (DMA) controllers, and low-overhead interrupt to transfer data with little or no intervention from the processor's computational units.

These DSP processors use specialized hardware and complex instructions for allowing more operations to be executed in every instruction cycle. However, they are difficult to program in assembly language and also difficult to design efficient C compilers in terms of speed and memory usage for supporting these complex-instruction architectures.

With the goals of achieving high performance and creating architecture that supports efficient C compilers, some DSP processors, such as the TMS320C6000 (C62x, C64x, and C67x), use very simple instructions. These processors achieve a high level of parallelism by issuing and executing multiple simple instructions in parallel at higher clock rates. For example, the TMS320C6000 uses very long instruction word (VLIW) architecture that provides eight execution units to execute four to eight instructions per clock cycle. These instructions have few restrictions on register usage and addressing modes, thus improving the efficiency of C compilers. However, the disadvantage of using simple instructions is that the VLIW processors need more instructions to perform a given task, thus require relatively high program memory usage and power consumption. These high-performance DSP processors are typically used in high-end video and radar systems, communication infrastructures, wireless base stations, and high-quality real-time video encoding systems.

1.3.3 Fixed- and Floating-Point Processors

A basic distinction between DSP processors is the arithmetic formats: fixed-point or floating-point. This is the most important factor for the system designers to determine the suitability of a DSP processor for a chosen application. The fixed-point representation of signals and arithmetic will be discussed in Chapter 3. Fixed-point DSP processors are either 16-bit or 24-bit devices, while floating-point processors are usually 32-bit devices. A typical 16-bit fixed-point processor, such as the TMS320C55x, stores numbers in a 16-bit integer or fraction format in a fixed range. Although coefficients and signals are only stored with 16-bit precision, intermediate values (products) may be kept at 32-bit precision within the internal 40-bit accumulators in order to reduce cumulative rounding errors. Fixed-point DSP devices are usually cheaper and faster than their floating-point counterparts because they use less silicon, have lower power consumption, and require fewer external pins. Most high-volume, low-cost embedded applications, such as appliance control, cellular phones, hard disk drives, modems, audio players, and digital cameras, use fixed-point processors.

Floating-point arithmetic greatly expands the dynamic range of numbers. A typical 32-bit floating-point DSP processor, such as the TMS320C67x, represents numbers with a 24-bit mantissa and an 8-bit exponent. The mantissa represents a fraction in the rang -1.0 to $+1.0$, while the exponent is an integer that represents the number of places that the binary point must be shifted left or right in order to obtain the true value. A 32-bit floating-point format covers a large dynamic range, thus the data dynamic range restrictions may be virtually ignored in a design using floating-point DSP processors. This is in contrast to fixed-point designs, where the designer has to apply scaling factors and other techniques to prevent arithmetic overflow, which are very difficult and time-consuming processes. As a result, floating-point DSP processors are generally easy to program and use, but are usually more expensive and have higher power consumption.

Example 1.7: The precision and dynamic range of commonly used 16-bit fixed-point processors are summarized in the following table:

	Precision	Dynamic range
Unsigned integer	1	$0 \leq x \leq 65\,535$
Signed integer	1	$-32\,768 \leq x \leq 32\,767$
Unsigned fraction	2^{-16}	$0 \leq x \leq (1 - 2^{-16})$
Signed fraction	2^{-15}	$-1 \leq x \leq (1 - 2^{-15})$

The precision of 32-bit floating-point DSP processors is 2^{-23} since there are 24 mantissa bits. The dynamic range is $1.18 \times 10^{-38} \leq x \leq 3.4 \times 10^{38}$.

System designers have to determine the dynamic range and precision needed for the applications. Floating-point processors may be needed in applications where coefficients vary in time, signals and coefficients require a large dynamic range and high precisions, or where large memory structures are required, such as in image processing. Floating-point DSP processors also allow for the efficient use of high-level C compilers, thus reducing the cost of development and maintenance. The faster development cycle for a floating-point processor may easily outweigh the extra cost of the DSP processor itself. Therefore, floating-point processors can also be justified for applications where development costs are high and production volumes are low.

1.3.4 Real-Time Constraints

A limitation of DSP systems for real-time applications is that the bandwidth of the system is limited by the sampling rate. The processing speed determines the maximum rate at which the analog signal can be sampled. For example, with the sample-by-sample processing, one output sample is generated when one input sample is presented to the system. Therefore, the delay between the input and the output for sample-by-sample processing is at most one sample interval (T s). A real-time DSP system demands that the signal processing time, t_p, must be less than the sampling period, T, in order to complete the processing task before the new sample comes in. That is,

$$t_\mathrm{p} + t_\mathrm{o} < T, \tag{1.6}$$

where t_o is the overhead of I/O operations.

This hard real-time constraint limits the highest frequency signal that can be processed by DSP systems using sample-by-sample processing approach. This limit on real-time bandwidth f_M is given as

$$f_\mathrm{M} \leq \frac{f_\mathrm{s}}{2} < \frac{1}{2\left(t_\mathrm{p} + t_\mathrm{o}\right)}. \tag{1.7}$$

It is clear that the longer the processing time t_p, the lower the signal bandwidth that can be handled by a given processor.

Although new and faster DSP processors have continuously been introduced, there is still a limit to the processing that can be done in real time. This limit becomes even more apparent when system cost is taken into consideration. Generally, the real-time bandwidth can be increased by using faster DSP processors, simplified DSP algorithms, optimized DSP programs, and parallel processing using multiple DSP processors, etc. However, there is still a trade-off between the system cost and performance.

Equation (1.7) also shows that the real-time bandwidth can be increased by reducing the overhead of I/O operations. This can be achieved by using block-by-block processing approach. With block processing methods, the I/O operations are usually handled by a DMA controller, which places data samples in a memory buffer. The DMA controller interrupts the processor when the input buffer is full, and a block of signal samples will be processed at a time. For example, for a real-time N-point fast Fourier transform (will be discussed in Chapter 6), the N input samples have to be buffered by the DMA controller. The block of N samples is processed after the buffer is full. The block computation must be completed before the next block of N samples is arrived. Therefore, the delay between input and output in block processing is dependent on the block size N, and this may cause a problem for some applications.

1.4 DSP System Design

A generalized DSP system design process is illustrated in Figure 1.9. For a given application, the theoretical aspects of DSP system specifications such as system requirements, signal analysis, resource analysis, and configuration analysis are first performed to define system requirements.

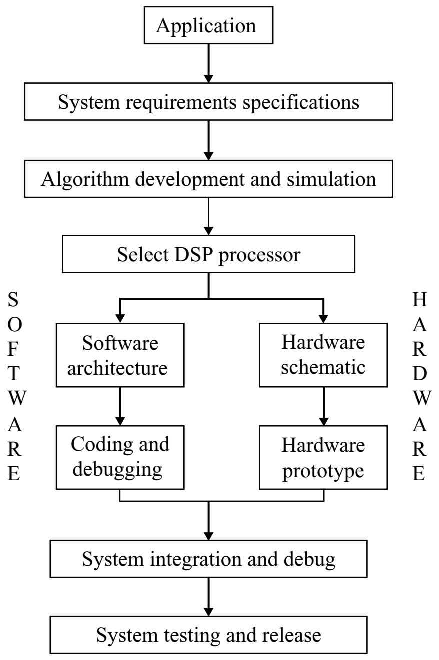

Application
System requirements specifications
Algorithm development and simulation
Select DSP processor
SOFTWARE
Software architecture
Coding and debugging
HARDWARE
Hardware schematic
Hardware prototype
System integration and debug
System testing and release

Figure 1.9 Simplified DSP system design flow

1.4.1 Algorithm Development

DSP systems are often characterized by the embedded algorithm, which specifies the arithmetic operations to be performed. The algorithm for a given application is initially described using difference equations or signal-flow block diagrams with symbolic names for the inputs and outputs. In documenting an algorithm, it is sometimes helpful to further clarify which inputs and outputs are involved by means of a data-flow diagram. The next stage of the development process is to provide more details on the sequence of operations that must be performed in order to derive the output. There are two methods of characterizing the sequence of operations in a program: flowcharts or structured descriptions.

At the algorithm development stage, we most likely work with high-level language DSP tools (such as MATLAB, Simulink, or C/C++) that are capable of algorithmic-level system simulations. We then implement the algorithm using software, hardware, or both, depending on specific needs. A DSP algorithm can be simulated using a general-purpose computer so that its performance can be tested and analyzed. A block diagram of general-purpose computer implementation is illustrated in Figure 1.10. The test signals may be internally generated by signal generators or digitized from a real environment based on the given application or received from other computers via the networks. The simulation program uses the signal samples stored in data file(s) as input(s) to produce output signals that will be saved in data file(s) for further analysis.

Advantages of developing DSP algorithms using a general-purpose computer are:

1. Using high-level languages such as MATLAB, Simulink, C/C++, or other DSP software packages on computers can significantly save algorithm development time. In addition, the prototype C programs used for algorithm evaluation can be ported to different DSP hardware platforms.

2. It is easy to debug and modify high-level language programs on computers using integrated software development tools.

3. Input/output operations based on disk files are simple to implement and the behaviors of the system are easy to analyze.

4. Floating-point data format and arithmetic can be used for computer simulations, thus easing development.

5. We can easily obtain bit-true simulations of the developed algorithms using MATLAB or Simulink for fixed-point DSP implementation.

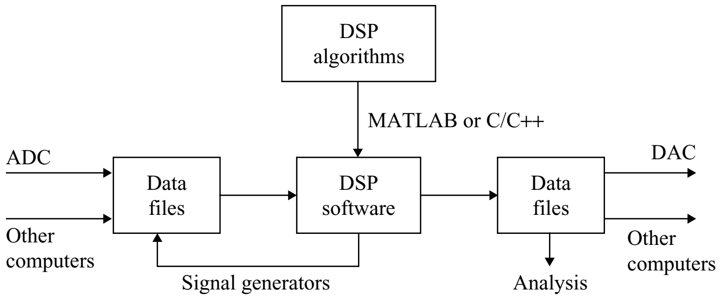


Figure 1.10 DSP software developments using a general-purpose computer

1.4.2 Selection of DSP Processors

As discussed earlier, DSP processors are used in a wide range of applications from high-performance radar systems to low-cost consumer electronics. As shown in Table 1.2, semiconductor vendors have responded to this demand by producing a variety of DSP processors. DSP system designers require a full understanding of the application requirements in order to select the right DSP processor for a given application. The objective is to choose the processor that meets the project's requirements with the most cost-effective solution. Some decisions can be made at an early stage based on arithmetic format, performance, price, power consumption, ease of development, and integration, etc. In real-time DSP applications, the efficiency of data flow into and out of the processor is also critical. However, these criteria will probably still leave a number of candidate processors for further analysis.

Example 1.8: There are a number of ways to measure a processor's execution speed. They include:

- MIPS – millions of instructions per second;
- MOPS – millions of operations per second;
- MFLOPS – millions of floating-point operations per second;
- MHz – clock rate; and
- MMACS – millions of multiply–accumulate operations.

In addition, there are other metrics such as milliwatts for measuring power consumption, MIPS per mw, or MIPS per dollar. These numbers provide only the sketchiest indication about performance, power, and price for a given application. They cannot predict exactly how the processor will measure up in the target system.

For high-volume applications, processor cost and product manufacture integration are important factors. For portable, battery-powered products such as cellular phones, digital cameras, and personal multimedia players, power consumption is more critical. For low- to medium-volume applications, there will be trade-offs among development time, cost of development tools, and the cost of the DSP processor itself. The likelihood of having higher performance processors with upward-compatible software in the future is also an important factor. For high-performance, low-volume applications such as communication infrastructures and wireless base stations, the performance, ease of development, and multiprocessor configurations are paramount.

Example 1.9: A number of DSP applications along with the relative importance for performance, price, and power consumption are listed in Table 1.3. This table shows that the designer of a handheld device has extreme concerns about power efficiency, but the main criterion of DSP selection for the communications infrastructures is its performance.

When processing speed is at a premium, the only valid comparison between processors is on an algorithm-implementation basis. Optimum code must be written for all candidates and then the execution time must be compared. Other important factors are memory usage and on-chip peripheral devices, such as on-chip converters and I/O interfaces.

Table 1.3 Some DSP applications with the relative importance rating

Application	Performance	Price	Power consumption
Audio receiver	1	2	3
DSP hearing aid	2	3	1
MP3 player	3	1	2
Portable video recorder	2	1	3
Desktop computer	1	2	3
Notebook computer	3	2	1
Cell phone handset	3	1	2
Cellular base station	1	2	3

Source: Adapted from [12]
Note: Rating – 1–3, with 1 being the most important

In addition, a full set of development tools and supports are important for DSP processor selection, including:

- Software development tools such as C compilers, assemblers, linkers, debuggers, and simulators.
- Commercially available DSP boards for software development and testing before the target DSP hardware is available.
- Hardware testing tools such as in-circuit emulators and logic analyzers.
- Development assistance such as application notes, DSP function libraries, application libraries, data books, and low-cost prototyping, etc.

1.4.3 Software Development

The four common measures of good DSP software are reliability, maintainability, extensibility, and efficiency. A reliable program is one that seldom (or never) fails. Since most programs will occasionally fail, a maintainable program is one that is easily correctable. A truly maintainable program is one that can be fixed by someone other than the original programmers. In order for a program to be truly maintainable, it must be portable on more than one type of hardware. An extensible program is one that can be easily modified when the requirements change.

A program is usually tested in a finite number of ways much smaller than the number of input data conditions. This means that a program can be considered reliable only after years of bug-free use in many different environments. A good DSP program often contains many small functions with only one purpose, which can be easily reused by other programs for different purposes. Programming tricks should be avoided at all costs, as they will often not be reliable and will almost always be difficult for someone else to understand even with lots of comments. In addition, the use of variable names should be meaningful in the context of the program.

As shown in Figure 1.9, the hardware and software design can be conducted at the same time for a given DSP application. Since there are a lot of interdependent factors between hardware and software, an ideal DSP designer will be a true ‘system’ engineer, capable of understanding issues with both hardware and software. The cost of hardware has gone down dramatically in recent years, thus the majority of the cost of a DSP solution now resides in software.

The software life cycle involves the completion of a software project: the project definition, the detailed specification, coding and modular testing, integration, system testing, and maintenance. Software maintenance is a significant part of the cost for a DSP system. Maintenance includes enhancing the software functions, fixing errors identified as the software is used, and modifying the software to work with new hardware and software. It is essential to document programs thoroughly with titles and comment statements because this greatly simplifies the task of software maintenance.

As discussed earlier, good programming techniques play an essential role in successful DSP applications. A structured and well-documented approach to programming should be initiated from the beginning. It is important to develop an overall specification for signal processing tasks prior to writing any program. The specification includes the basic algorithm and task description, memory requirements, constraints on the program size, execution time, and so on. A thoroughly reviewed specification can catch mistakes even before code has been written and prevent potential code changes at the system integration stage. A flow diagram would be a very helpful design tool to adopt at this stage.

Writing and testing DSP code is a highly interactive process. With the use of integrated software development tools that include simulators or evaluation boards, code may be tested regularly as it is written. Writing code in modules or sections can help this process, as each module can be tested individually, thus increasing the chance of the entire system working at the system integration stage.

There are two commonly used methods in developing software for DSP devices: using assembly program or C/C++ program. Assembly language is similar to the machine code actually used by the processor. Programming in assembly language gives the engineers full control of processor functions and resources, thus resulting in the most efficient program for mapping the algorithm by hand. However, this is a very time-consuming and laborious task, especially for today's highly paralleled DSP architectures. A C program, on the other hand, is easier for software development, upgrade, and maintenance. However, the machine code generated by a C compiler is inefficient in both processing speed and memory usage. Recently, DSP manufacturers have improved C compiler efficiency dramatically, especially with the DSP processors that use simple instructions and general register files.

Often the ideal solution is to work with a mixture of C and assembly code. The overall program is controlled and written by C code, but the run-time critical inner loops and modules are written in assembly language. In a mixed programming environment, an assembly routine may be called as a function or intrinsics, or in-line coded into the C program. A library of hand-optimized functions may be built up and brought into the code when required. The assembly programming for the TMS320C55x will be discussed in Chapter 2.

1.4.4 High-Level Software Development Tools

Software tools are computer programs that have been written to perform specific operations. Most DSP operations can be categorized as being either analysis tasks or filtering tasks. Signal analysis deals with the measurement of signal properties. MATLAB is a powerful environment for signal analysis and visualization, which are critical components in understanding and developing a DSP system. C programming is an efficient tool for performing signal processing and is portable over different DSP platforms.

MATLAB is an interactive, technical computing environment for scientific and engineering numerical analysis, computation, and visualization. Its strength lies in the fact that complex numerical problems can be solved easily in a fraction of the time required with a programming language such as C. By using its relatively simple programming capability, MATLAB can be easily extended to create new functions, and is further enhanced by numerous toolboxes such as the *Signal Processing Toolbox* and *Filter Design Toolbox*. In addition, MATLAB provides many graphical user interface (GUI) tools such as Filter Design and Analysis Tool (FDATool).

The purpose of a programming language is to solve a problem involving the manipulation of information. The purpose of a DSP program is to manipulate signals to solve a specific signal processing problem. High-level languages such as C and C++ are computer languages that have English-like commands and

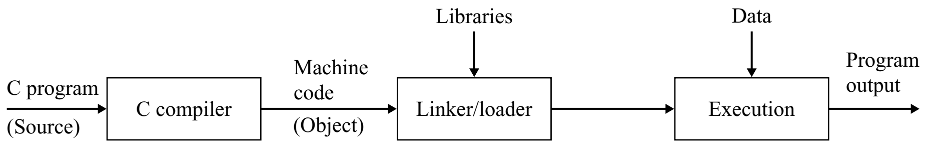


Figure 1.11 Program compilation, linking, and execution flow

instructions. High-level language programs are usually portable, so they can be recompiled and run on many different computers. Although C/C++ is categorized as a high-level language, it can also be written for low-level device drivers. In addition, a C compiler is available for most modern DSP processors such as the TMS320C55x. Thus C programming is the most commonly used high-level language for DSP applications.

C has become the language of choice for many DSP software development engineers not only because it has powerful commands and data structures but also because it can easily be ported on different DSP processors and platforms. The processes of compilation, linking/loading, and execution are outlined in Figure 1.11. C compilers are available for a wide range of computers and DSP processors, thus making the C program the most portable software for DSP applications. Many C programming environments include GUI debugger programs, which are useful in identifying errors in a source program. Debugger programs allow us to see values stored in variables at different points in a program, and to step through the program line by line.

1.5 Introduction to DSP Development Tools

The manufacturers of DSP processors typically provide a set of software tools for the user to develop efficient DSP software. The basic software development tools include C compiler, assembler, linker, and simulator. In order to execute the designed DSP tasks on the target system, the C or assembly programs must be translated into machine code and then linked together to form an executable code. This code conversion process is carried out using software development tools illustrated in Figure 1.12.

The TMS320C55x software development tools include a C compiler, an assembler, a linker, an archiver, a hex conversion utility, a cross-reference utility, and an absolute lister. The C55x C compiler generates assembly source code from the C source files. The assembler translates assembly source files, either hand-coded by DSP programmers or generated by the C compiler, into machine language object files. The assembly tools use the common object file format (COFF) to facilitate modular programming. Using COFF allows the programmer to define the system's memory map at link time. This maximizes performance by enabling the programmer to link the code and data objects into specific memory locations. The archiver allows users to collect a group of files into a single archived file. The linker combines object files and libraries into a single executable COFF object module. The hex conversion utility converts a COFF object file into a format that can be downloaded to an EPROM programmer or a flash memory program utility.

In this section, we will briefly describe the C compiler, assembler, and linker. A full description of these tools can be found in the user's guides [13, 14].

1.5.1 C Compiler

C language is the most popular high-level tool for evaluating algorithms and developing real-time software for DSP applications. The C compiler can generate either a mnemonic assembly code or an algebraic assembly code. In this book, we use the mnemonic assembly (ASM) language. The C compiler package includes a shell program, code optimizer, and C-to-ASM interlister. The shell program supports

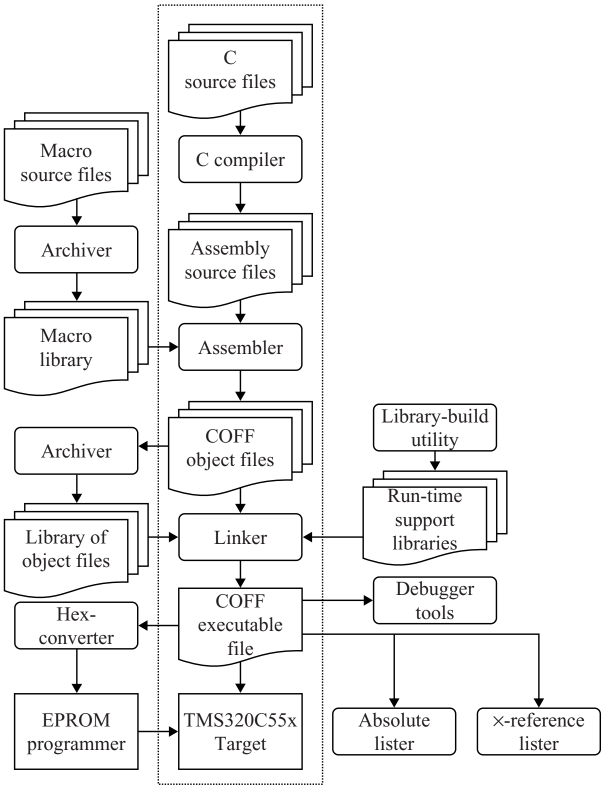


Figure 1.12 TMS320C55x software development flow and tools

automatically compiled, assembled, and linked modules. The optimizer improves run-time and code density efficiency of the C source file. The C-to-ASM interlister inserts the original comments in C source code into the compiler's output assembly code so users can view the corresponding assembly instructions for each C statement generated by the compiler.

The C55x C compiler supports American National Standards Institute (ANSI) C and its run-time support library. The run-time support library `rts55.lib` (or `rts55x.lib` for large memory model) includes functions to support string operation, memory allocation, data conversion, trigonometry, and exponential manipulations.

C language lacks specific features of DSP, especially those fixed-point data operations that are necessary for many DSP algorithms. To improve compiler efficiency for DSP applications, the C55x C compiler supports in-line assembly language for C programs. This allows adding highly efficient assembly code directly into the C program. Intrinsics are another improvement for substituting DSP arithmetic operation with DSP assembly intrinsic operators. We will introduce more compiler features in Chapter 2 and subsequent chapters.

1.5.2 Assembler

The assembler translates processor-specific assembly language source files (in ASCII format) into binary COFF object files. Source files can contain assembler directives, macro directives, and instructions.

Assembler directives are used to control various aspects of the assembly process, such as the source file listing format, data alignment, section content, etc. Binary object files contain separate blocks (called sections) of code or data that can be loaded into memory space.

Once the DSP algorithm has been written in assembly, it is necessary to add important assembly directives to the source code. Assembler directives are used to control the assembly process and enter data into the program. Assembly directives can be used to initialize memory, define global variables, set conditional assembly blocks, and reserve memory space for code and data.

1.5.3 Linker

The linker combines multiple binary object files and libraries into a single executable program for the target DSP hardware. It resolves external references and performs code relocation to create the executable module. The C55x linker handles various requirements of different object files and libraries, as well as targets system memory configurations. For a specific hardware configuration, the system designers need to provide the memory mapping specification to the linker. This task can be accomplished by using a linker command file. The visual linker is also a very useful tool that provides a visualized memory usage map directly.

The linker commands support expression assignment and evaluation, and provides `MEMORY` and `SECTION` directives. Using these directives, we can define the memory model for the given target system. We can also combine object file sections, allocate sections into specific memory areas, and define or redefine global symbols at link time.

An example linker command file is listed in Table 1.4. The first portion uses the `MEMORY` directive to identify the range of memory blocks that physically exist in the target hardware. These memory blocks

Table 1.4 Example of linker command file used by TMS320C55x

```
/* Specify the system memory map */
MEMORY
{
    RAM  (RWIX) : o = 0x000100, l = 0x00feff  /* Data memory     */
    RAM0 (RWIX) : o = 0x010000, l = 0x008000  /* Data memory     */
    RAM1 (RWIX) : o = 0x018000, l = 0x008000  /* Data memory     */
    RAM2 (RWIX) : o = 0x040100, l = 0x040000  /* Program memory  */
    ROM  (RIX)  : o = 0x020100, l = 0x020000  /* Program memory  */
    VECS (RIX)  : o = 0xffff00, l = 0x000100  /* Reset vector    */
}
/* Specify the sections allocation into memory */
SECTIONS
{
    vectors  > VECS   /* Interrupt vector table     */
   .text     > ROM    /* Code                       */
   .switch   > RAM    /* Switch table info          */
   .const    > RAM    /* Constant data              */
   .cinit    > RAM2   /* Initialization tables      */
   .data     > RAM    /* Initialized data           */
   .bss      > RAM    /* Global & static vars       */
   .stack    > RAM    /* Primary system stack       */
   .sysstack > RAM    /* Secondary system stack     */
    expdata0 > RAM0   /* Global & static vars       */
    expdata1 > RAM1   /* Global & static vars       */
}
```

are available for the software to use. Each memory block has its name, starting address, and the length of the block. The address and length are given in bytes for C55x processors and in words for C54x processors. For example, the data memory block called RAM starts at the byte address 0x100, and it has a size of 0xFEFF bytes. Note that the prefix 0x indicates the following number is represented in hexadecimal (hex) form.

The `SECTIONS` directive provides different code section names for the linker to allocate the program and data within each memory block. For example, the program can be loaded into the `.text` section, and the uninitialized global variables are in the `.bss` section. The attributes inside the parentheses are optional to set memory access restrictions. These attributes are:

`R` – Memory space can be read.

`W` – Memory space can be written.

`X` – Memory space contains executable code.

`I` – Memory space can be initialized.

Several additional options used to initialize the memory can be found in [13].

1.5.4 Other Development Tools

Archiver is used to group files into a single archived file, that is, to build a library. The archiver can also be used to modify a library by deleting, replacing, extracting, or adding members. Hex-converter converts a COFF object file into an ASCII hex format file. The converted hex format files are often used to program EPROM and flash memory devices. Absolute lister takes linked object files to create the `.abs` files. These `.abs` files can be assembled together to produce a listing file that contains absolute addresses of the entire system program. Cross-reference lister takes all the object files to produce a cross-reference listing file. The cross-reference listing file includes symbols, definitions, and references in the linked source files.

The DSP development tools also include simulator, EVM, XDS, and DSK. A simulator is the software simulation tool that does not require any hardware support. The simulator can be used for code development and testing. The EVM is a hardware evaluation module including I/O capabilities to allow developers to evaluate the DSP algorithms for the specific DSP processor in real time. EVM is usually a computer board to be connected with a host computer for evaluating the DSP tasks. The XDS usually includes in-circuit emulation and boundary scan for system development and debug. The XDS is an external stand-alone hardware device connected to a host computer and a DSP board. The DSK is a low-cost development board for the user to develop and evaluate DSP algorithms under a Windows operation system environment. In this book, we will use the Spectrum Digital's TMS320VC5510 DSK for real-time experiments.

The DSK works under the Code Composer Studio (CCS) development environment. The DSK package includes a special version of the CCS [15]. The DSK communicates with CCS via its onboard universal serial bus (USB) JTAG emulator. The C5510 DSK uses a 200 MHz TMS320VC5510 DSP processor, an AIC23 stereo CODEC, 8 Mbytes synchronous DRAM, and 512 Kbytes flash memory.

1.6 Experiments and Program Examples

Texas Instruments' CCS Integrated Development Environment (IDE) is a DSP development tool that allows users to create, edit, build, debug, and analyze DSP programs. For building applications, the CCS provides a project manager to handle the programming project. For debugging purposes, it provides breakpoints, variable watch windows, memory/register/stack viewing windows, probe points to stream data to and from the target, graphical analysis, execution profile, and the capability to display mixed disassembled and C instructions. Another important feature of the CCS is its ability to create and manage large projects from a GUI environment. In this section, we will use a simple sinewave example to introduce the basic editing features, key IDE components, and the use of the C55x DSP development tools. We also demonstrate simple approaches to software development and the debug process using the TMS320C55x simulator. Finally, we will use the C5510 DSK to demonstrate an audio loop-back example in real time.

1.6.1 Experiments of Using CCS and DSK

After installing the DSK or CCS simulator, we can start the CCS IDE. Figure 1.13 shows the CCS running on the DSK. The IDE consists of the standard toolbar, project toolbar, edit toolbar, and debug toolbar. Some basic functions are summarized and listed in Figure 1.13. Table 1.5 briefly describes the files used in this experiment.

Procedures of the experiment are listed as follows:

1. *Create a project for the CCS*: Choose `Project→New` to create a new project file and save it as `useCCS.pjt` to the directory `..\experiments\exp1.6.1_CCSandDSK`. The CCS uses the project to operate its built-in utilities to create a full-build application.

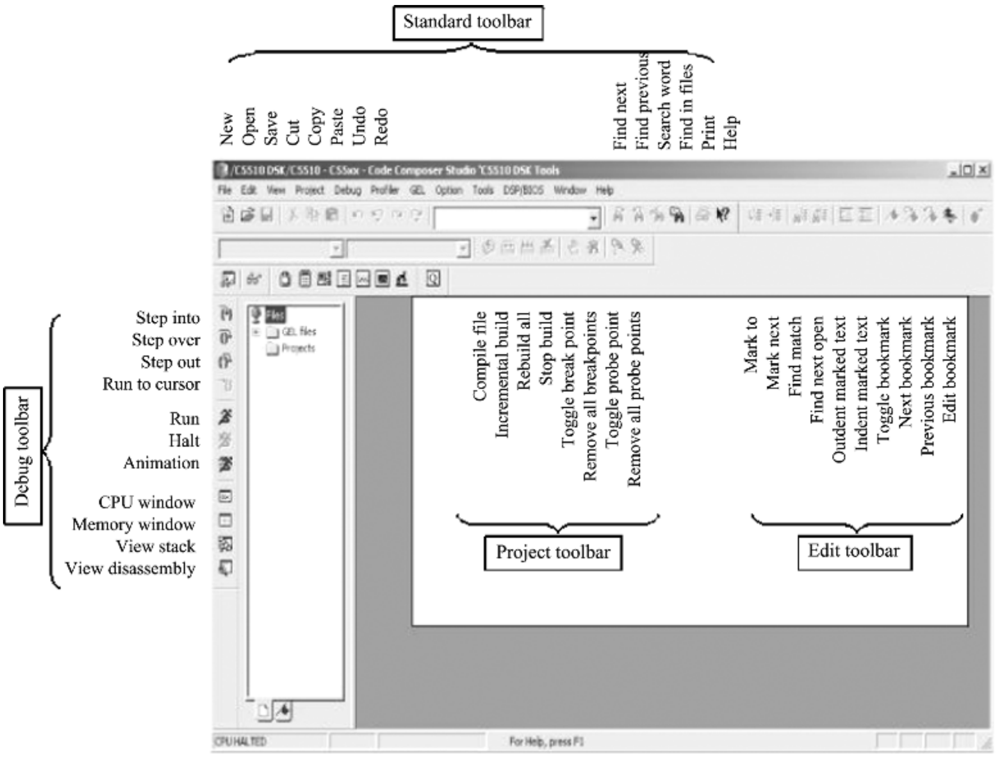


Figure 1.13 CCS IDE

Table 1.5 File listing for experiment exp1.6.1CCSandDSK

Files	Description
useCCS.c	C file for testing experiment
useCCS.h	C header file
useCCS.pjt	DSP project file
useCCS.cmd	DSP linker command file

2. *Create C program files using the CCS editor*: Choose File→New to create a new file, type in the example C code listed in Tables 1.6 and 1.7. Save C code listed in Table 1.6 as useCCS.c to ..\experiments\exp1.6.1_CCSandDSK\src, and save C code listed in Table 1.7 as useCCS.h to the directory ..\experiments\exp1.6.1_CCSandDSK\inc. This example reads precalculated sine values from a data table, negates them, and stores the results in a reversed order to an output buffer. The programs useCCS.c and useCCS.h are included in the companion CD. However, it is recommended that we create them using the editor to become familiar with the CCS editing functions.

3. *Create a linker command file for the simulator*: Choose File→New to create another new file, and type in the linker command file as listed in Table 1.4. Save this file as useCCS.cmd to the directory ..\experiments\exp1.6.1_CCSandDSK. The command file is used by the linker to map different program segments into a prepartitioned system memory space.

4. *Setting up the project*: Add useCCS.c and useCCS.cmd to the project by choosing Project→Add Files to Project, then select files useCCS.c and useCCS.cmd. Before building a project, the search paths of the included files and libraries should be setup for C compiler, assembler, and linker. To setup options for C compiler, assembler, and linker choose Project→Build Options. We need to add search paths to include files and libraries that are not included in the C55x DSP tools directories, such as the libraries and included files we created

Table 1.6 Program example, useCCS.c

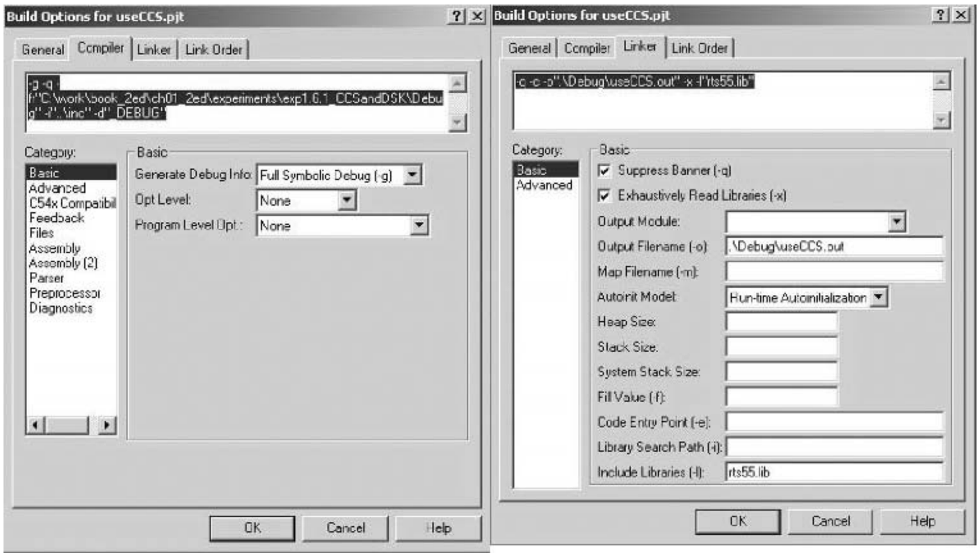
```
#include "useCCS.h"

short outBuffer[BUF_SIZE];

void main()
{
     short i, j;

     j = 0;
     while (1)
     {
       for (i=BUF_SIZE-1; i>= 0;i--)
       {
           outBuffer [j++] = 0 - sineTable[i]; //  <- Set breakpoint

           if (j >= BUF_SIZE)
               j = 0;
       }
       j++;
     }
}
```

Table 1.7 Program example header file, `useCCS.h`

```
#define  BUF_SIZE 40
const short sineTable[BUF_SIZE]=
     {0x0000, 0x000f, 0x001e, 0x002d, 0x003a, 0x0046, 0x0050, 0x0059,
      0x005f, 0x0062, 0x0063, 0x0062, 0x005f, 0x0059, 0x0050, 0x0046,
      0x003a, 0x002d, 0x001e, 0x000f, 0x0000, 0xfff1, 0xffe2, 0xffd3,
      0xffc6, 0xffba, 0xffb0, 0xffa7, 0xffa1, 0xff9e, 0xff9d, 0xff9e,
      0xffa1, 0xffa7, 0xffb0, 0xffba, 0xffc6, 0xffd3, 0xffe2, 0xfff1};
```

in the working directory. Programs written in C language require the use of the run-time support library, either `rts55.lib` or `rts55x.lib`, for system initialization. This can be done by selecting the compiler and linker dialog box and entering the C55x run-time support library, `rts55.lib`, and adding the header file path related to the source file directory. We can also specify different directories to store the output executable file and map file. Figure 1.14 shows an example of how to set the search paths for compiler, assembler, and linker.

5. *Build and run the program*: Use `Project→Rebuild All` command to build the project. If there are no errors, the CCS will generate the executable output file, `useCCS.out`. Before we can run the program, we need to load the executable output file to the C55x DSK or the simulator. To do so, use `File→Load Program` menu and select the `useCCS.out` in `..\expriments\exp1.6.1_CCSandDSK\Debug` directory and load it. Execute this program by choosing `Debug→Run`. The processor status at the bottom-left-hand corner of the CCS will change from **CPU HALTED** to **CPU RUNNING**. The running process can be stopped by the `Debug→Halt` command. We can continue the program by reissuing the **Run** command or exiting the DSK or the simulator by choosing `File→Exit` menu.



```
-g -q -fr"C:\work\book_2ed\ch01_2ed\experiments\exp1.6.1_CCSandDSK\Debug" -i"..\inc" -d"_DEBUG"
```

Basic
Generate Debug Info: Full Symbolic Debug (-g)
Opt Level: None
Program Level Opt.: None

```
-c -c -o".\Debug\useCCS.out" -x -l"rts55.lib"
```

Basic
☑ Suppress Banner (-q)
☑ Exhaustively Read Libraries (-x)
Output Module:
Output Filename (-o): .\Debug\useCCS.out
Map Filename (-m):
Autoinit Model: Run-time Autoinitialization
Heap Size:
Stack Size:
System Stack Size:
Fill Value (-f):
Code Entry Point (-e):
Library Search Path (-i):
Include Libraries (-l): rts55.lib

(a) Setting the include file searching path. (b) Setting the run-time support library.

Figure 1.14 Setup search paths for C compiler, assembler, and linker: (a) setting the include file searching path; (b) setting the run-time support library

1.6.2 Debugging Program Using CCS and DSK

The CCS IDE has extended traditional DSP code generation tools by integrating a set of editing, emulating, debugging, and analyzing capabilities in one entity. In this section, we will introduce some program building steps and software debugging capabilities of the CCS.

The standard toolbar in Figure 1.13 allows users to create and open files, cut, copy, and paste text within and between files. It also has undo and redo capabilities to aid file editing. Finding text can be done within the same file or in different files. The CCS built-in context-sensitive help menu is also located in the standard toolbar menu. More advanced editing features are in the edit toolbar menu, including mark to, mark next, find match, and find next open parenthesis for C programs. The features of out-indent and in-indent can be used to move a selected block of text horizontally. There are four bookmarks that allow users to create, remove, edit, and search bookmarks.

The project environment contains C compiler, assembler, and linker. The project toolbar menu (see Figure 1.13) gives users different choices while working on projects. The compile only, incremental build, and build all features allow users to build the DSP projects efficiently. Breakpoints permit users to set software breakpoints in the program and halt the processor whenever the program executes at those breakpoint locations. Probe points are used to transfer data files in and out of the programs. The profiler can be used to measure the execution time of given functions or code segments, which can be used to analyze and identify critical run-time blocks of the programs.

The debug toolbar menu illustrated in Figure 1.13 contains several stepping operations: step-into-a-function, step-over-a-function, and step-out-off-a-function. It can also perform the run-to-cursor-position operation, which is a very convenient feature, allowing users to step through the code. The next three hot buttons in the debug toolbar are run, halt, and animate. They allow users to execute, stop, and animate the DSP programs. The watch windows are used to monitor variable contents. CPU registers and data memory viewing windows provide additional information for ease of debugging programs. More custom options are available from the pull-down menus, such as graphing data directly from the processor memory.

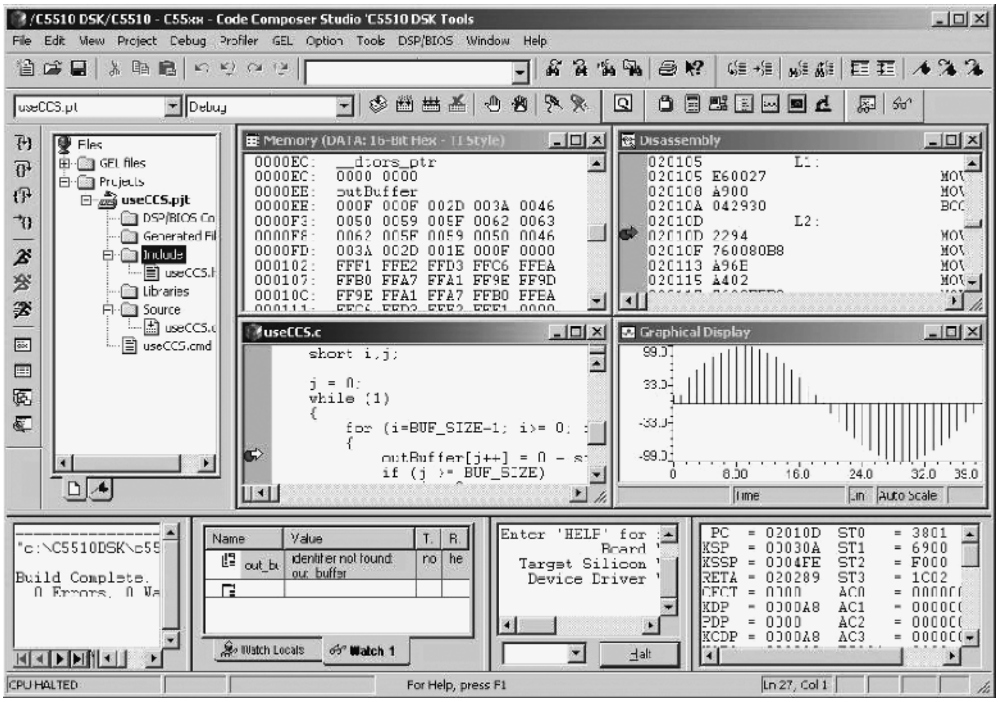
We often need to check the changing values of variables during program execution for developing and testing programs. This can be accomplished with debugging settings such as breakpoints, step commands, and watch windows, which are illustrated in the following experiment.

Procedures of the experiment are listed as follows:

1. *Add and remove breakpoints*: Start with `Project→Open`, select `useCCS.pjt` from the directory `..\experiments\exp1.6.2_CCSandDSK`. Build and load the example project `useCCS.out`. Double click the C file, `useCCS.c`, in the project viewing window to open it in the editing window. To add a breakpoint, move the cursor to the line where we want to set a breakpoint. The command to enable a breakpoint can be given either from the **Toggle Breakpoint** hot button on the project toolbar or by clicking the mouse button on the line of interest. The function key <F9> is a shortcut that can be used to toggle a breakpoint. Once a breakpoint is enabled, a red dot will appear on the left to indicate where the breakpoint is set. The program will run up to that line without passing it. To remove breakpoints, we can either toggle breakpoints one by one or select the **Remove All Breakpoints** hot button from the debug toolbar to clear all the breakpoints at once. Now load the `useCCS.out` and open the source code window with source code `useCCS.c`, and put the cursor on the line:

   ```
   outBuffer[j++] = 0 - sineTable[i]; // <- set breakpoint
   ```

   Click the **Toggle Breakpoint** button (or press <F9>) to set the breakpoint. The breakpoint will be set as shown in Figure 1.15.



```
0000EC:   __dtors_ptr
0000EC:   0000 0000
0000EE:   outBuffer
0000EE:   000F 000F 002D 003A 0046
0000F3:   0050 0059 005F 0062 0063
0000F8:   0062 005F 0059 0050 0046
0000FD:   003A 002D 001E 000F 0000
000102:   FFF1 FFE2 FFD3 FFC6 FFEA
000107:   FFB0 FFA7 FFA1 FF9E FF9D
00010C:   FF9E FFA1 FFA7 FFB0 FFEA
```

```
020105              L1:
020105 E60027                  MOV
020108 A900                    MOV
02010A 042930                  BCC
02010D              L2:
02010D 2294                    MOV
02010F 760080B8                MOV
020113 A96E                    MOV
020115 A402                    MOV
```

```
    short i,j;

    j = 0;
    while (1)
    {
        for (i=BUF_SIZE-1; i>= 0;
        {
            outBuffer[j++] = 0 - s
            if (j >= BUF_SIZE)
```

```
"c:\C5510DSK\c55

Build Complete.
  0 Errors, 0 Wa
```

Name	Value	T.	R.
out_bu	identifier not found: out_buffer	no	he

```
Enter 'HELP' for
             Board
  Target Silicon
   Device Driver
```

```
 PC  = 02010D  ST0  = 3801
XSP  = 00030A  ST1  = 6900
XSSP = 0004FE  ST2  = F000
RETA = 020289  ST3  = 1C02
CFCT = 0000    AC0  = 0000C
XDP  = 0000A8  AC1  = 0000C
DP   = 0000    AC2  = 0000C
XCDP = 0000A8  AC3  = 0000C
```

Figure 1.15 CCS screen snapshot of the example using CCS

2. *Set up viewing windows*: CCS IDE provides many useful windows to ease code development and the debugging process. The following are some of the most often used windows:

   CPU register viewing window: On the standard tool menu bar, click `View`→Registers→CPU Registers to open the CPU registers window. We can edit the contents of any CPU register by double clicking it. If we right click the **CPU Register Window** and select **Allow Docking**, we can move the window around and resize it. As an example, try to change the temporary register T0 and accumulator AC0 to new values of T0 = 0x1234 and AC0 = 0x56789ABC.

   Command window: From the CCS menu bar, click `Tools→Command Window` to add the command window. We can resize and dock it as well. The command window will appear each time when we rebuild the project.

   Disassembly window: Click `View→Disassembly` on the menu bar to see the disassembly window. Every time we reload an executable out file, the disassembly window will appear automatically.

3. *Workspace feature*: We can customize the CCS display and settings using the workspace feature. To save a workspace, click `File→Workspace→Save Workspace` and give the workspace a name and path where the workspace will be stored. When we restart CCS, we can reload the workspace by clicking `File→Workspace→Load Workspace` and use a workspace from previous work. Now save the workspace for your current CCS settings then exit the CCS. Restart CCS and reload the workspace. After the workspace is reloaded, you should have the identical settings restored.

4. *Using the single-step features*: When using C programs, the C55x system uses a function called `boot` from the run-time support library `rts55.lib` to initialize the system. After we load the `useCCS.out`,

the program counter (PC) will be at the start of the boot function (in assembly code `boot.asm`). This code should be displayed in the disassembly window. For a project starting with C programs, there is a function called `main( )` from which the C program begins to execute. We can issue the command **Go Main** from the **Debug** menu to start the C program after loading the `useCCS.out`. After the **Go Main** command is executed, the processor will be initialized for `boot.asm` and then halted at the location where the function `main( )` is. Hit the <F8> key or click the single-step button on the debug toolbar repeatedly to single step through the program `useCCS.c`, and watch the values of the CPU registers change. Move the cursor to different locations in the code and try the **Run to Cursor** command (hold down the <Ctrl> and <F10> keys simultaneously).

5. *Resource monitoring*: CCS provides several resource viewing windows to aid software development and the debugging process.

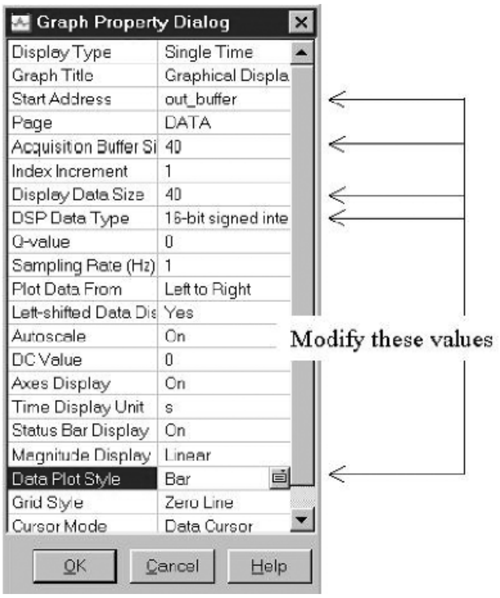
   Watch windows: From `View→Watch Window`, open the watch window. The watch window can be used to show the values of listed variables. Type the output buffer name, `outBuffer`, into the expression box and click **OK**. Expand the `outBuffer` to view each individual element of the buffer.

   Memory viewing: From `View→Memory`, open a memory window and enter the starting address of the `outBuffer` in the data page to view the output buffer data. Since global variables are defined globally, we can use the variable name as its address for memory viewing. Is memory viewing showing the same data values as the watch window in previous step?

   Graphics viewing: From `View→Graph→Time/Frequency`, open the graphic property dialog. Set the display parameters as shown in Figure 1.16. The CCS allows the user to plot data directly from memory by specifying the memory location and its length.

   Set a breakpoint on the line of the following C statement:

```
outBuffer[j++] = 0 - sineTable[i]; // <- set breakpoint
```



Display Type	Single Time
Graph Title	Graphical Displa
Start Address	out_buffer
Page	DATA
Acquisition Buffer Si	40
Index Increment	1
Display Data Size	40
DSP Data Type	16-bit signed inte
Q-value	0
Sampling Rate (Hz)	1
Plot Data From	Left to Right
Left-shifted Data Dis	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s
Status Bar Display	On
Magnitude Display	Linear
Data Plot Style	Bar
Grid Style	Zero Line
Cursor Mode	Data Cursor

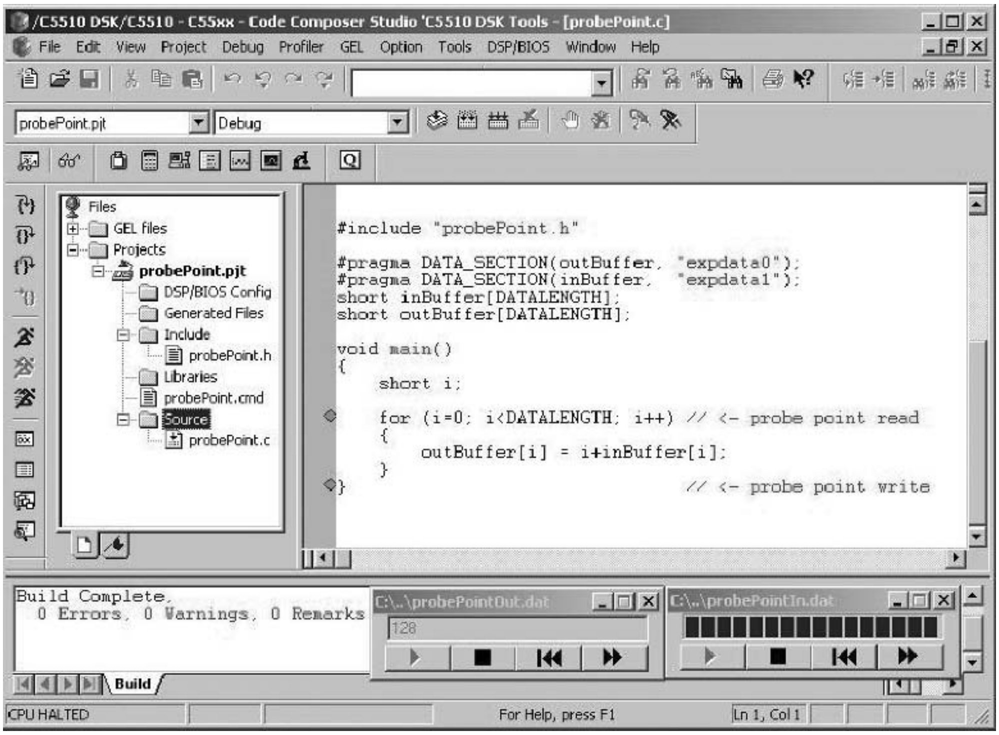
Figure 1.16 Graphics display settings

Start animation execution (<F12> hot key), and view DSP CPU registers and `outBuffer` data in watch window, memory window, and graphical plot window. Figure 1.15 shows one instant snapshot of the animation. The yellow arrow represents the current program counter's location, and the red dot shows where the breakpoint is set. The data and register values in red color are the ones that have just been updated.

1.6.3 File I/O Using Probe Point

Probe point is a useful tool for algorithm development, such as simulating the real-time input and output operations with predigitized data in files. When a probe point is reached, the CCS can either read the selected amount of data samples from a file of the host computer to the target processor memory or write the processed data samples from the target processor to the host computer as an output file for analysis. In the following example, we will learn how to setup probe points to transfer data between the example program `probePoint.c` and a host computer. In the example, the input data is read into `inBuffer[ ]` via probe point before the for loop, and the output data in `outBuffer[ ]` is written to the host computer at the end of the program. The program `probePoint.c` is listed in Figure 1.17.

Write the C program as shown in Figure 1.17. This program reads in 128 words of data from a file, and adds each data value with the loop counter, `i`. The result is saved in `outBuffer[ ]`, and written out to the host computer at the end of the program. Save the program in `..\experiments \exp1.6.3_probePoint\src` and create the linker command file `probePoint.cmd` based on the



```
#include "probePoint.h"

#pragma DATA_SECTION(outBuffer, "expdata0");
#pragma DATA_SECTION(inBuffer,  "expdata1");
short inBuffer[DATALENGTH];
short outBuffer[DATALENGTH];

void main()
{
    short i;

    for (i=0; i<DATALENGTH; i++) // <- probe point read
    {
        outBuffer[i] = i+inBuffer[i];
    }
}                                // <- probe point write
```

```
Build Complete,
  0 Errors, 0 Warnings, 0 Remarks
```

Figure 1.17 CCS screen snapshot of example of using probe point: (a) set up probe point address and length for output data buffer; (b) set up probe point address and length for input data buffer; and (c) connect probe points with files

Table 1.8 File listing for experiment `exp1.6.3_probePoint`

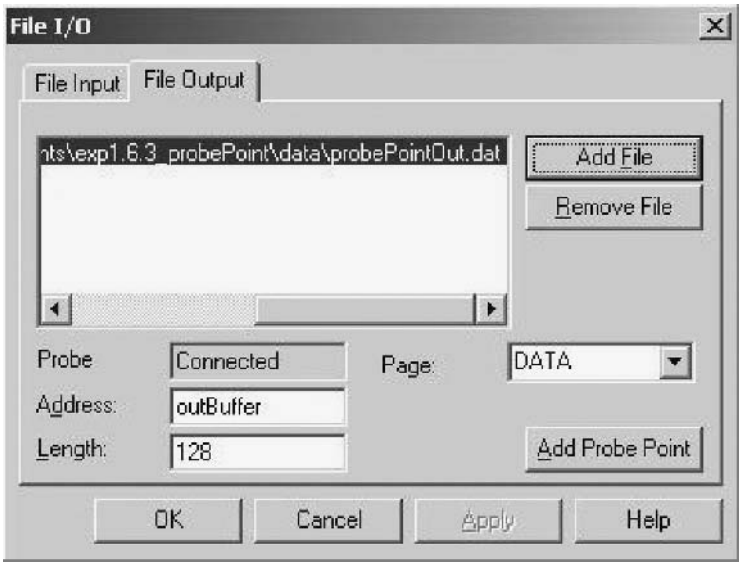
Files	Description
`probePoint.c`	C file for testing probe point experiment
`probePoint.h`	C header file
`probePoint.pjt`	DSP project file
`probePoint.cmd`	DSP linker command file

previous example `useCCS`. The sections `expdata0` and `expdata1` are defined for the input and output data buffers. The `pragma` keyword in the C code will be discussed in Chapter 2. Table 1.8 gives a brief description of the files used for CCS probe point experiment.

Procedures of the experiment are listed as follows:

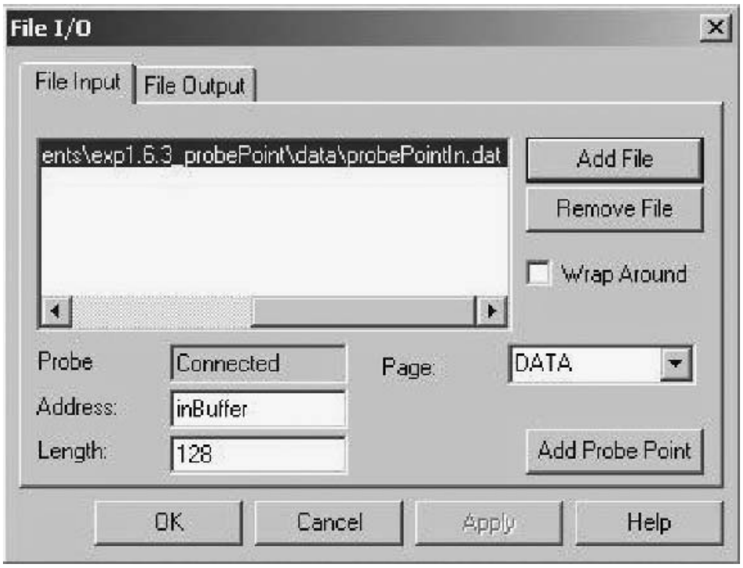
1. *Set probe point position*: To set a probe point, put the cursor on the line where the probe point will be set and click the **Toggle Probe Point** hot button. A blue dot to the left indicates that the probe point is set (see Figure 1.17). The first probe point on the line of the for loop reads data into the `inBuffer[ ]`, while the second probe points at the end of the main program and writes data from the `outBuffer[ ]` to the host computer.

2. *Connect probe points*: From `File→File I/O`, open the file I/O dialog box and select **File Output** tab. From the **Add File** tab, enter `probePointOut.dat` as the filename from the directory `..\experiments\exp1.6.3_probePoint\data` and select *.dat (Hex) as the file type and then click **Open** tab. Use the output buffer name `outBuffer` as the address and 128 as the length of the data block for transferring 128 data to the host computer from the output buffer when the probe point is reached as shown in Figure 1.18(a). Also connect the input data probe point to the DSP processor. Select the **File Input** tab from the **File I/O** dialog box and click **Add File** tab. Navigate to the


File I/O
File Input | File Output
nts\exp1.6.3_probePoint\data\probePointOut.dat
Probe: Connected
Address: outBuffer
Length: 128
Page: DATA

(a) Set up probe point address and length for output data buffer.

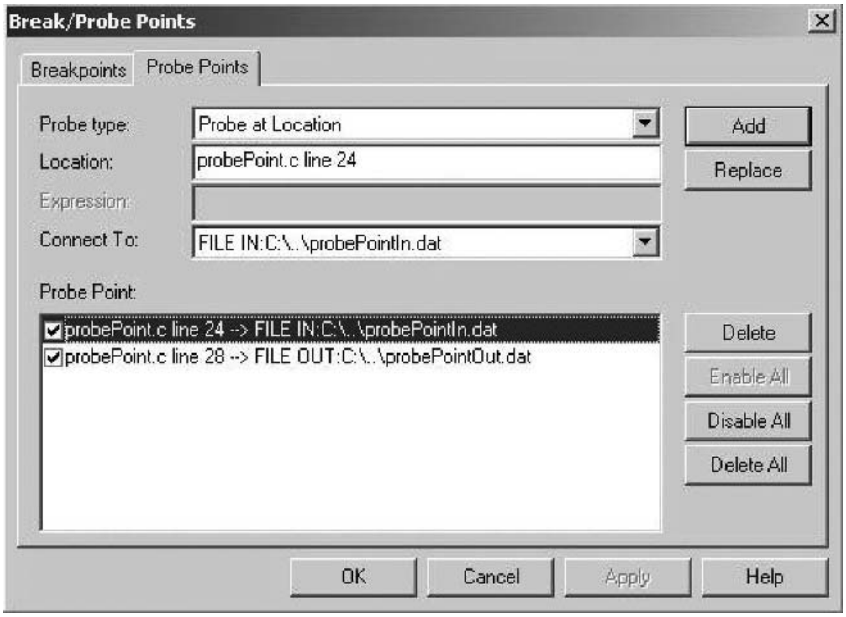
Figure 1.18 Connect probe points



ents\exp1.6.3_probePoint\data\probePointIn.dat

Probe: Connected
Page: DATA
Address: inBuffer
Length: 128

(b) Set up probe point address and length for input data buffer.



Probe type: Probe at Location
Location: probePoint.c line 24
Expression:
Connect To: FILE IN:C:\..\probePointIn.dat

Probe Point:

- ✓ probePoint.c line 24 --> FILE IN:C:\..\probePointIn.dat
- ✓ probePoint.c line 28 --> FILE OUT:C:\..\probePointOut.dat

(c) Connect probe points with files.

Figure 1.18 (*Continued*)

Table 1.9 Data input file used by CCS probe point

```
1651 1 c000 1 80
0x0000
0x0001
0x0002
0x0003
0x0004
. . .
```

folder `..\experiments\exp1.6.3_probePoint\data` and choose `probePointIn.dat` file. In the **Address** box, enter `inBuffer` for the input data buffer and set length to 128 (see Figure 1.18(b)). Now select **Add Probe Point** tab to connect the probe point with the output file `probePointOut.dat` and input data file `probePointIn.dat`. A new dialog box, **Break/Probe Points**, as shown in Figure 1.18(c), will pop up. From this window, highlight the probe point and click the **Connect** pull-down tab to select the output data file `probePointOut.dat` for the output data file, and select the input file `probePointIn.dat` for the input data file. Finally, click the **Replace** button to connect the input and output probe points to these two files. After closing the **Break/Probe Points** dialog box, the **File I/O** dialog box will be updated to show that the probe point has been connected. Restart the program and run the program. After execution, view the data file `probePointOut.dat` using the built-in editor by issuing `File → Open` command. If there is a need to view or edit the data file using other editors/viewers, exit the CCS or disconnect the file from the **File I/O**.

3. *Probe point results*: Input data file for experiment is shown in Table 1.9, and the output data file is listed in Table 1.10. The first line contains the header information in hexadecimal format, which uses the syntax illustrated in Figure 1.19.

For this example, the data shown in Tables 1.9 and 1.10 are in hexadecimal format, with the address of `inBuffer` at 0xC000 and `outBuffer` at 0x8000; both are at the data page, and each block contains 128 (0x80) data samples.

1.6.4 File I/O Using C File System Functions

As shown in Figure 1.17, the probe point can be used to connect data files to the C55x system via CCS. The CCS uses only the ACSII file format. Binary file format, on the other hand, is more efficient for storing in the computers. In real-world applications, many data files are digitized and stored in binary format instead of ASCII format. In this section, we will introduce the C file-I/O functions.

The CCS supports standard C library I/O functions and include `fopen( ), fclose( ), fread ( ), fwrite( )`, and so on. These functions not only provide the ability of operating on different

Table 1.10 Data output file saved by CCS probe point

```
1651 1 8000 1 80
0x0000
0x0002
0x0004
0x0006
0x0008
. . .
```

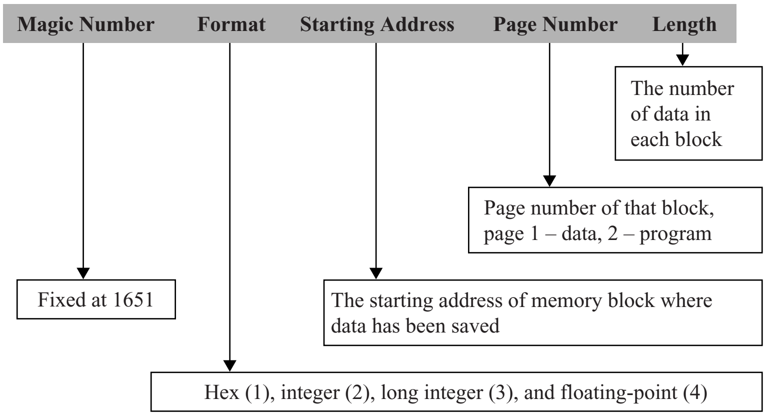

Magic Number
Format
Starting Address
Page Number
Length
The number of data in each block
Page number of that block, page 1 – data, 2 – program
Fixed at 1651
The starting address of memory block where data has been saved
Hex (1), integer (2), long integer (3), and floating-point (4)

Figure 1.19 CCS file header format

file formats, but also allow users to directly use the functions on computers. Comparing with probe point introduced in the previous section, these file I/O functions are functions that are portable to other development environments.

Table 1.11 shows an example of C program that uses `fopen( )`, `fclose( )`, `fread( )`, and `fwrite( )` functions. The input is a stereo data file in linear PCM WAV (Microsoft file format for using pulse code modulation audio data) file format. In this WAV file, a dialog is carried out between the left and right stereo channels. The stereo audio data file is arranged such that the even samples are the left-channel data and the odd samples are the right-channel data. The example program reads in audio data samples in binary form from the input WAV file and writes out the left and right channels separately into two binary data files. The output files are written using the linear PCM WAV file format and will have the same sampling rate as the input file. These WAV files can be played back via the Windows Media Player.

In this example, the binary files are read and written in byte units as `sizeof(char)`. The CCS file I/O for C55x only supports this data format. For data units larger than byte, such as 16-bit short data type, the read and write must be done in multiple byte accesses. In the example, the 16-bit linear PCM data is read in and written out with 2 bytes at a time. When running this program on a computer, the data access can be changed to its native data type `sizeof(short)`. The files used are in linear PCM WAV file format. WAV file format can have several different file types and it supports different sampling frequencies. Different WAV file formats are given as an exercise in this chapter for readers to explore further. The detailed WAV file format can be found in references [18–20]. The files used for this experiment are given in Table 1.12 with brief descriptions.

Procedures of the experiment are listed as follows:

1. Create the project `fielIO.pjt` and save it in the directory `..\experiments\exp1.6.4_fileIO`. Copy the linker command file from the previous experiment and rename it as `fileIO.cmd`. Write the experiment program `fielIO.c` as shown in Table 1.9 and save it to the directory `..\experiments\exp1.6.4_fileIO\src`. Write the WAV file header as shown in Table 1.13 and save it as `fielIO.h` in the directory `..\experiments\exp1.6.4_fileIO\inc`. The input data file inStereo.wav is included in the CD and located in the directory ..\experiments\exp1.6.4_fileIO\data.

2. Build the `fielIO` project and test the program.

3. Listen to the output WAV files generated by the experiment using computer audio player such as Windows Media Player and compare experimental output WAV file with the input WAV file.

Table 1.11 Program of using C file system, `fielIO.c`

```
#include <stdio.h>
#include "fielIO.h"

void main()
{
  FILE *inFile;       // File pointer of input signal
  FILE *outLeftFile;  // File pointer of left channel output signal
  FILE *outRightFile; // File pointer of right channel output signal
  short x[4];
  char wavHd[44];
  inFile = fopen("..\\data\\inStereo.wav", "rb");
  if (inFile == NULL)
  {
      printf("Can't open inStereo.wav");
      exit(0);
  }
  outLeftFile = fopen("..\\data\\outLeftCh.wav", "wb");
  outRightFile = fopen("..\\data\\outRightCh.wav", "wb");

  // Skip input wav file header
  fread(wavHd, sizeof(char), 44, inFile);
  // Add wav header to left and right channel output files
  fwrite(wavHeader, sizeof(char), 44, outLeftFile);
  fwrite(wavHeader, sizeof(char), 44, outRightFile);

  // Read stereo input and write to left/right channels
  while( (fread(x, sizeof(char), 4, inFile) == 4) )
  {
       fwrite(&x[0], sizeof(char), 2, outLeftFile);
       fwrite(&x[2], sizeof(char), 2, outRightFile);
  }
  fclose(inFile);
  fclose(outLeftFile);
  fclose(outRightFile);
}
```

Table 1.12 File listing for experiment `exp1.6.4_fileIO`

Files	Description
`fileIO.c`	C file for testing file IO experiment
`fileIO.h`	C header file
`fileIO.pjt`	DSP project file
`fileIO.cmd`	DSP linker command file

1.6.5 Code Efficiency Analysis Using Profiler

The profiler of the CCS measures the execution status of specific segments of a project. This is a very useful tool for analyzing and optimizing large and complex DSP projects. In this experiment, we will use the CCS profiler to obtain statistics of the execution time of DSP functions. The files used for this experiment are listed in Table 1.14.

Table 1.13 Program example of header file, `fielIO.h`

```
// This wav file header is pre-calculated
// It can only be used for this experiment
short wavHeader[44]={
0x52,  0x49,  0x46,  0x46,  //  RIFF
0x2E,  0x8D,  0x01,  0x00,  //  101678 (36 bytes + 101642 bytes data)
0x57,  0x41,  0x56,  0x45,  //  WAVE
0x66,  0x6D,  0x74,  0x20,  //  Formatted
0x10,  0x00,  0x00,  0x00,  //  PCM audio
0x01,  0x00,  0x01,  0x00,  //  Linear PCM, 1-channel
0x40,  0x1F,  0x00,  0x00,  //  8 kHz sampling
0x80,  0x3E,  0x00,  0x00,  //  Byte rate = 16000
0x02,  0x00,  0x10,  0x00,  //  Block align = 2, 16-bit/sample
0x64,  0x61,  0x74,  0x61,  //  Data
0x0A,  0x8D,  0x01,  0x00}; //  101642 data bytes
```

Table 1.14 File listing for experiment `exp1.6.5_profile`

Files	Description
`profile.c`	C file for testing DSP profile experiment
`profile.h`	C header file
`profile.pjt`	DSP project file
`profile.cmd`	DSP linker command file

Procedures of experiment are listed as follows:

1. *Creating the DSP project*: Create a new `project profile.pjt` and write a C program `profile.c` as shown in Table 1.15. Build the project and load the program. For demonstration purposes, we will profile a function and a segment of code in the program.

   This program calls the sine function from the C math library to generate 1000 Hz tone at 8000 Hz sampling rate. The generated 16-bit integer data is saved on the computer in WAV file format. As an example, we will use CCS profile feature to profile the function `sinewave( )` and the code segment in `main( )` which calls the `sinewave( )` function.

2. *Set up profile points*: Build and load the program `profile.out`. Open the source code `profile.c`. From the **Profile Point** menu, select `Start New Session`. This opens the profile window. We can give a name to the profile session. Select **Functions** tab from the window. Click the function name (not the calling function inside the main function), `sinewave( )`, and drag this function into the profile session window. This enables the CCS profiler to profile the function `sinewave( )`. We can profile a segment of the source code by choosing the **Ranges** tab instead of the **Functions** tab. Highlight two lines of source code in `main( )` where the sine function is called, and drag them into the profiler session window. This enables profiling two lines of code segment. Finally, run the program and record the cycle counts shown on the profile status window. The profile is run with the assistance of breakpoints, so it is not suitable for real-time analysis. But, it does provide useful analysis using the digitized data files. We use profile to identify the critical run-time code segments and functions. These functions, or code segments, can then be optimized to improve their real-time performances. Figure 1.20 shows the example profile results. The sine function in C library uses an average of over 4000 cycles to generate one data sample. This is very inefficient due to the use of floating-point arithmetic for calculation of the sine function. Since the TMS320C55x is a fixed-point DSP processor, the processing speed has been dramatically slowed by emulating floating-point

Table 1.15 Program example using CCS profile features, `profile.c`

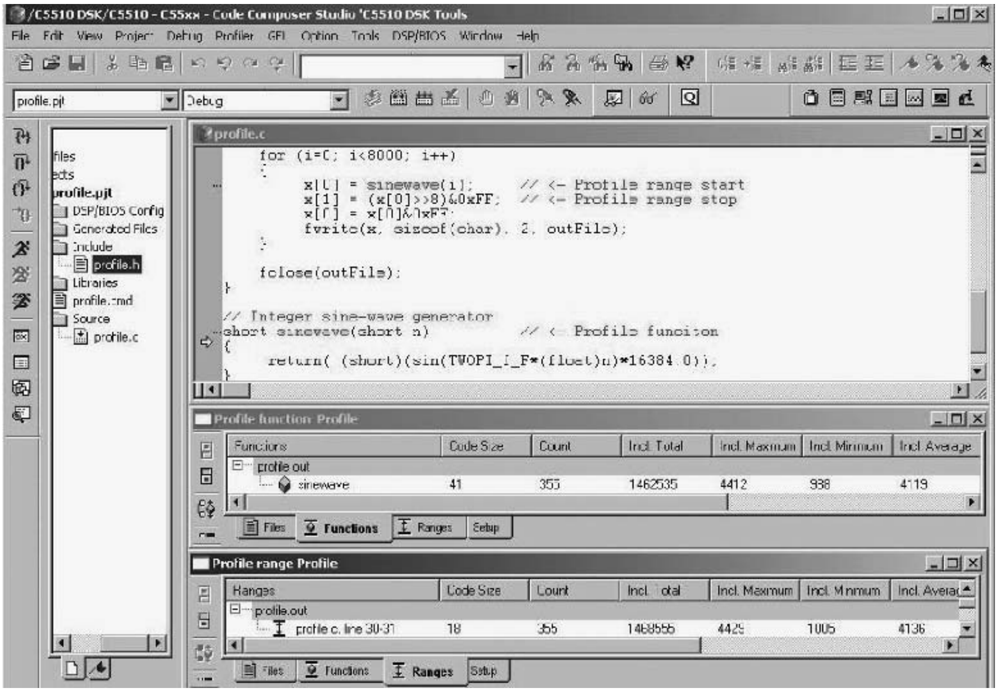
```
#include <stdio.h>
#include <math.h>
#include "profile.h"

void main()
{
   FILE *outFile; // File pointer of output file
   short x[2],i;

   outFile = fopen("..\\data\\output.wav", "wb");

   // Add wav header to left and right channel output files
   fwrite(wavHeader, sizeof(char), 44, outFile);

   // Generate 1 second 1 kHz sine wave at 8kHz sampling rate
   for (i=0; i<8000; i++)
   {
         x[0] = sinewave(i); // <- Profile range start
         x[1] = (x[0]>>8)&0xFF; // <- Profile range stop
         x[0] = x[0]&0xFF;
         fwrite(x, sizeof(char), 2, outFile);
   }
   fclose(outFile);

}

// Integer sine-wave generator
short sinewave(short n)        // <- Profile function
{
   return( (short)(sin(TWOPI_f_F*(float)n)*16384.0));
}
```

arithmetic. We will discuss the implementation of fixed-point arithmetic for the C55x processors in Chapter 3. We will also present a more efficient way to generate variety of digital signals including sinewave in the following chapters.

1.6.6 Real-Time Experiments Using DSK

The programs we have presented in previous sections can be used either on a C5510 DSK or on a C55x simulator. In this section, we will focus on the DSK for real-time experiments. The DSK is a low-cost DSP development and evaluation hardware platform. It uses USB interface for connecting to the host computer. A DSK can be used either for DSP program development and debugging or for real-time demonstration and evaluation. We will introduce detailed DSK functions and features in Chapter 2 along with the C5510 peripherals.

The DSK has several example programs included in its package. We modified an audio loop-back demo that takes an audio input through the line-in jack, and plays back via the headphone output in real time. The photo of the C5510 DSK is shown in Figure 1.21. For the audio demo example, we connect DSK with an audio player as the input audio source and a headphone (or loudspeaker) as the audio output. The demo program is listed in Table 1.16.



profile.c

```
    for (i=0; i<8000; i++)
    {
        x[0] = sinewave(i);       // <- Profile range start
        x[1] = (x[0]>>8)&0xFF;    // <- Profile range stop
        x[0] = x[0]&0xFF;
        fwrite(x, sizeof(char), 2, outFile);
    }

    fclose(outFile);
}

// Integer sine-wave generator
short sinewave(short n)           // <- Profile funciton
{
    return( (short)(sin(TWOPI_I_F*(float)n)*16384.0));
}
```

Profile function: Profile

Funcions	Code Size	Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
profile.out						
sinewave	41	355	1462535	4412	938	4119

Profile range Profile

Ranges	Code Size	Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Averag
profile.out						
profile.c, line 30-31	18	355	1468555	4429	1005	4136

Figure 1.20 Profile window of DSP profile status

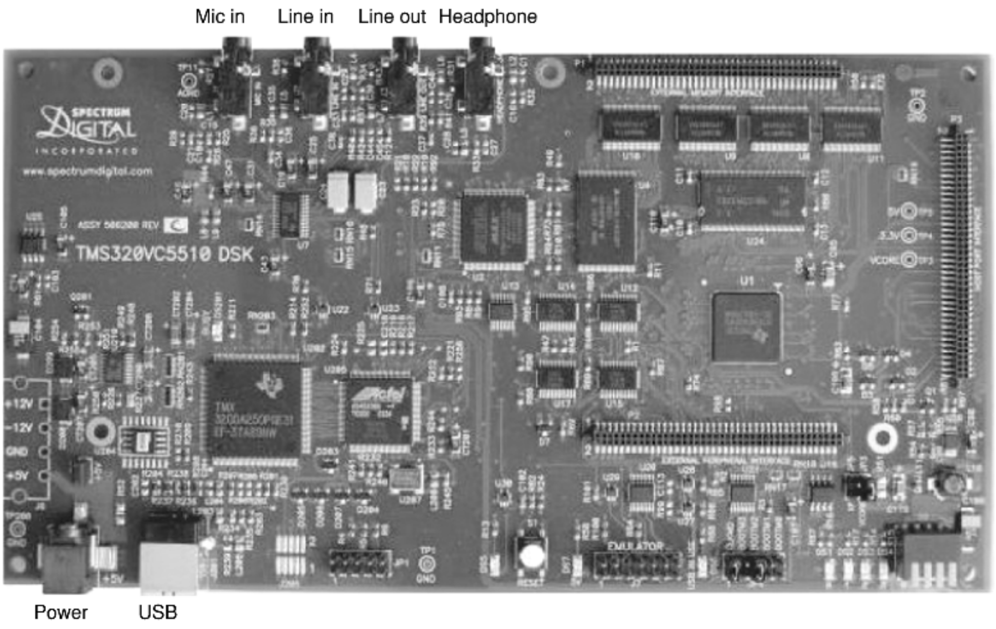


Figure 1.21 TMSVC 5510 DSK

Table 1.16 Program example of DSK audio loop-back, `loopback.c`

```c
#include "loopbackcfg.h"
#include "dsk5510.h"
#include "dsk5510_aic23.h"

/* Codec configuration settings */
DSK5510_AIC23_Config config = { \
   0x0017, /* 0 DSK5510_AIC23_LEFTINVOL  Left line input channel
volume */ \
   0x0017, /* 1 DSK5510_AIC23_RIGHTINVOL Right line input channel
volume */\
   0x01f9, /* 2 DSK5510_AIC23_LEFTHPVOL  Left channel headphone
volume */  \
   0x01f9, /* 3 DSK5510_AIC23_RIGHTHPVOL Right channel headphone
volume */ \
   0x0011, /* 4 DSK5510_AIC23_ANAPATH    Analog audio path
control */      \
   0x0000, /* 5 DSK5510_AIC23_DIGPATH    Digital audio path
control */     \
   0x0000, /* 6 DSK5510_AIC23_POWERDOWN  Power down
control */             \
   0x0043, /* 7 DSK5510_AIC23_DIGIF      Digital audio interface
format */ \
   0x0081, /* 8 DSK5510_AIC23_SAMPLERATE Sample rate
control */            \
   0x0001, /* 9 DSK5510_AIC23_DIGACT     Digital interface
activation */   \
};

void main()
{
    DSK5510_AIC23_CodecHandle hCodec;
    Int16 i,j,left,right;

    /* Initialize the board support library, must be called first */
    DSK5510_init();
    /* Start the codec */
    hCodec = DSK5510_AIC23_openCodec(0, &config);

    /* Loop back line-in audio for 30 seconds at 48 kHz sampling rate */
    for (i = 0; i < 30; i++)
    {
        for (j = 0; j < 48000; j++)
        {
            /* Read a sample from the left input channel */
            while (!DSK5510_AIC23_read16(hCodec, &left));
            /* Write a sample to the left output channel */
            while (!DSK5510_AIC23_write16(hCodec, left));
            /* Read a sample from the right input channel */
            while (!DSK5510_AIC23_read16(hCodec, &right));
            /* Write a sample to the right output channel */
            while (!DSK5510_AIC23_write16(hCodec, right));
        }
    }
    /* Close the codec */
    DSK5510_AIC23_closeCodec(hCodec);
}
```

Table 1.17 File listing for experiment `exp1.6.6_loopback`

Files	Description
`loopback.c`	C file for testing DSK real-time loopback experiment
`Loopback.cdb`	DSP BIOS configuration file
`loopbackcfg.h`	C header file
`loopback.pjt`	DSP project file
`loopbackcfg.cmd`	DSP linker command file
`desertSun.wav`	Test data file
`fools8k.wav`	Test data file

This experimental program first initializes the DSK board and the AIC23 CODEC. It starts the audio loopback at 48 kHz sampling rate for 1 min. Finally, it stops and closes down the AIC23. The settings of the AIC23 will be presented in detail in Chapter 2. This example is included in the companion CD and can be loaded into DSK directly. In the subsequent chapters, we will continue to modify this program for use in other audio signal processing experiments. As we have discussed in this chapter, the signal processing can be either in a sample-by-sample or in a block-by-block method. This audio loopback experiment is implemented in the sample-by-sample method. It is not very efficient in terms of processing I/O overhead. We will introduce block-by-block method to reduce the I/O overhead in the following chapters.

The files used for this experiment are listed in Table 1.17. In addition, there are many built-in header files automatically included by CCS.

Procedures of experiment are listed as follows:

1. Play the WAV file `desertSun.wav` in the directory `..\experiments\exp1.6.6_loopback\data` using media player in loop mode on a host computer, or use an audio player as audio source.

2. Connect one end of a stereo cable to the computer's audio output jack, and the other end to the DSK's line-in jack. Connect a headphone to the DSK headphone output jack.

3. Start CCS and open `loopback.pjt` from the directory `..\experiments\exp1.6.6_loopback`, and then build and load the `loopback.out`.

4. Play the audio by the host computer in loop mode and run the program on the DSK. The DSK will acquire the signal from the line-in input, and send it out to the DSK headphone output.

1.6.7 Sampling Theory

Aliasing is caused by using sampling frequency incorrectly. A chirp signal (will be discussed in Chapter 8) is a sinusoid function with changing frequency, which is good for observing aliasing. This experiment uses audible and visual results from the MATLAB to illustrate the aliasing phenomenon. Table 1.18 lists the MATLAB code for experiment.

In the program given in Table 1.18, `fl` and `fh` are the low and high frequencies of the chirp signals, respectively. The sampling frequency `fs` is set to 800 Hz. This experiment program generates 1 s of chirp signal. The experiment uses MATLAB function `sound( )` as the audio tool for listening to the chirp signal and uses the `plot( )` function as a visual aid to illustrate the aliasing result.

Table 1.18 MATLAB code to demonstrate aliasing

```
fl = 0;                  % Low frequency
fh = 200;                % High frequency
fs = 800;                % Sampling frequency
n = 0:1/fs:1;            % 1 seconds of data
phi = 2*pi*(fl*n + (fh-fl)*n.*n/2);
y = 0.5*sin(phi);
sound(y, fs);
plot(y)
```

Procedures of the experiment are listed as follows:

1. Start MATLAB and set MATLAB path to the experiment directory.
2. Type `samplingTheory` in the MATLAB command window to start the experiment.
3. When the sampling frequency is set to 800 Hz, the code will generate a chirp signal sweeping from 0 to 200 Hz. The MATLAB uses the function `sound` to play the continuous chirp signal and plot the entire signal as shown in Figure 1.22(a).
4. Now change the sampling frequency to `fs` = 200 Hz. Because the sampling frequency `fs` does not meet the sampling theorem, the chirp signal generated will have aliasing. The result is audible and can be viewed from a MATLAB plot. The sweeping frequency folded at 100 Hz is shown in Figure 1.22(b).

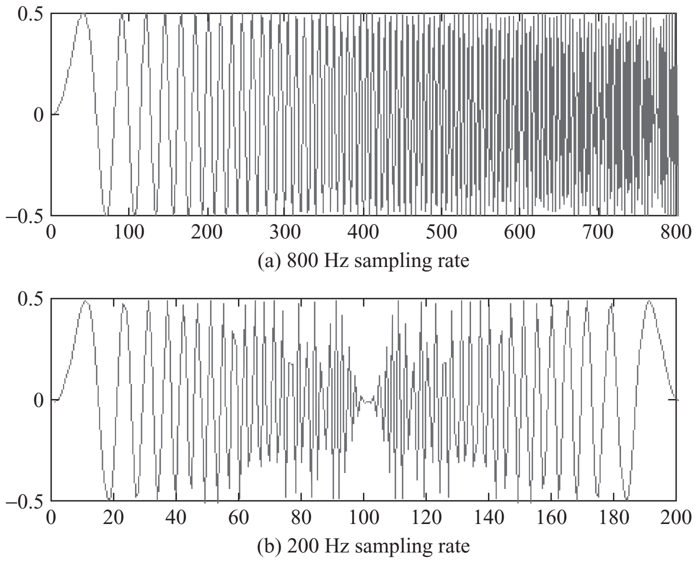


Figure 1.22 Sampling theory experiment using chirp signal: (a) 800 Hz sampling rate; (b) 200 Hz sampling rate

5. Now use the chirp experiment as reference, and write a signal generator using the sine function available in MATLAB. Set the sampling frequency to 800 Hz. Start with sine function frequency 200 Hz, generate 2 s of audio, and plot the signal. Repeat this experiment five times by incrementing the sine function frequency by 100 Hz each time.

1.6.8 Quantization in ADCs

Quantization is an important factor when designing a DSP system. We will discuss quantization in Chapter 3. For this experiment, we use MATLAB to show that different ADC wordlengths have different quantization errors. Table 1.19 shows a portion of the MATLAB code for this experiment.

Procedures of the experiment are listed as follows:

1. Start MATLAB and set the MATLAB path to the experiment directory.

2. Type `ADCQuantization` in the MATLAB command window to start the experiment.

3. When MATLAB prompts for input, enter the desired ADC peak voltage and wordlength.

4. Enter an input voltage to the ADC to compute the digital output and error. This experiment will calculate the ADC resolution and compute the error in million volts; it will also display the corresponding hexadecimal numbers that will be generated by ADC for the given voltage.

An example of output is listed in Table 1.20.

Table 1.19 MATLAB code for experiment of ADC quantization

```
peak = input('Enter the ADC peak voltage (0 - 5) = ');
bits = input('Enter the ADC wordlength (4 - 12) = ');
volt = input('Enter the analog voltage = ');

% Calculate resolution
resolution = peak / power(2, bits);

% Find digital output
digital = round(volt/resolution);

% Calculate error
error = volt - digital*resolution;
```

Table 1.20 Output of ADC quantization

```
>> ADCQuantization
Enter the ADC peak voltage (0 - 5) = 5
Enter the ADC wordlength (4 - 12) = 12
Enter the analog voltage (less than or equal to peak voltage) = 3.445
ADC resolution (mv) = 1.2207
ADC corresponding output (HEX) = B06
ADC quantization error (mv)= 0.1758
```

References

[1] ITU Recommendation G.729, *Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction* (*CS-ACELP*), Mar. 1996.
[2] ITU Recommendation G.723.1, *Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*, Mar. 1996.
[3] ITU Recommendation G.722, *7 kHz Audio-Coding within 64 kbit/s*, Nov. 1988.
[4] 3GPP TS 26.190, *AMR Wideband Speech Codec: Transcoding Functions, 3GPP Technical Specification*, Mar. 2002.
[5] ISO/IEC 13818-7, *MPEG-2 Generic Coding of Moving Pictures and Associated Audio Information*, Oct. 2000.
[6] ISO/IEC 11172-3, *Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s - Part 3: Audio*, Nov. 1992.
[7] ITU Recommendation G.711, *Pulse Code Modulation (PCM) of Voice Frequencies*, Nov. 1988.
[8] Texas Instruments, *TLV320AIC23B Data Manual*, Literature no. SLWS106H, 2004.
[9] Spectrum Digital, Inc., *TMS320VC5510 DSK Technical Reference*, 2002.
[10] S. Zack and S. Dhanani, 'DSP co-processing in FPGA: Embedding high-performance, low-cost DSP functions,' *Xilinx White Paper*, WP212, 2004.
[11] Berkeley Design Technology, Inc., 'Choosing a DSP processor,' *White-Paper*, 2000.
[12] G. Frantz and L. Adams, 'The three Ps of value in selecting DSPs,' *Embedded System Programming*, Oct. 2004.
[13] Texas Instruments, Inc., *TMS320C55x Optimizing C Compiler User's Guide*, Literature no. SPRU281E, Revised 2003.
[14] Texas Instruments, Inc., *TMS320C55x Assembly Language Tools User's Guide*, Literature no. SPRU280G, Revised 2003.
[15] Spectrum Digital, Inc., *TMS320C5000 DSP Platform Code Composer Studio DSK v2 IDE,* DSK Tools for C5510 Version 1.0, Nov. 2002.
[16] Texas Instruments, Inc., *Code Composer Studio User's Guide (Rev B)*, Literature no. SPRU328B, Mar. 2000.
[17] Texas Instruments, Inc., *Code Composer Studio v3.0 Getting Start Guide*, Literature no. SPRU509E, Sept. 2004.
[18] IBM and Microsoft, *Multimedia Programming Interface and Data Specification* 1.0, Aug. 1991.
[19] Microsoft, *New Multimedia Data Types and Data Techniques*, Rev 1.3, Aug. 1994.
[20] Microsoft, *Multiple Channel Audio Data and WAVE Files*, Nov. 2002.
[21] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1989.
[22] S. J. Orfanidis, *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1996.
[23] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd Ed., Englewood Cliffs, NJ: Prentice Hall, 1996.
[24] A. Bateman and W. Yates, *Digital Signal Processing Design*, New York: Computer Science Press, 1989.
[25] S. M. Kuo and D. R. Morgan, *Active Noise Control Systems – Algorithms and DSP Implementations*, New York: John Wiley & Sons, Inc., 1996.
[26] J. H. McClellan, R. W. Schafer, and M. A. Yoder, *DSP First*: *A Multimedia Approach*, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1998.
[27] S. M. Kuo and W. S. Gan, *Digital Signal Processors – Architectures, Implementations, and Applications*, Upper Saddle River, NJ: Prentice Hall, 2005.
[28] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals: Architectures and Features*, Piscataway, NJ: IEEE Press, 1997.
[29] Berkeley Design Technology, Inc., 'The evolution of DSP processor,' *A BDTi White Paper*, 2000.

Exercises

1. Given an analog audio signal with frequencies up to 10 kHz.

   (a) What is the minimum required sampling frequency that allows a perfect reconstruction of the signal from its samples?

   (b) What will happen if a sampling frequency of 8 kHz is used?

(c) What will happen if the sampling frequency is 50 kHz?

(d) When sampled at 50 kHz, if only taking every other samples (this is a decimation by 2), what is the frequency of the new signal? Is this causing aliasing?

2. Refer to Example 1.1, assuming that we have to store 50 ms (1 ms $= 10^{-3}$ s) of digitized signals. How many samples are needed for (a) narrowband telecommunication systems with $f_s = 8$ kHz, (b) wideband telecommunication systems with $f_s = 16$ kHz, (c) audio CDs with $f_s = 44.1$ kHz, and (d) professional audio systems with $f_s = 48$ kHz.

3. Assume that the dynamic range of the human ear is about 100 dB, and the highest frequency the human ear can hear is 20 kHz. If you are a high-end digital audio system designer, what size of converters and sampling rate are needed? If your design uses single-channel 16-bit converter and 44.1 kHz sampling rate, how many bits are needed to be stored for 1 min of music?

4. Given a discrete time sinusoidal signal of $x(n) = 5\sin(n\pi/100)$ V.

(a) Find its peak-to-peak range?

(b) What are the quantization resolutions of (i) 8-bit, (ii) 12-bit, (iii) 16-bit, and (iv) 24-bit ADCs for this signal?

(c) In order to obtain the quantization resolution of below 1 mV, how many bits are required in the ADC?

5. A speech file (`timit_1.asc`) was sampled using 16-bit ADC with one of the following sampling rates: 8 kHz, 12 kHz, 16 kHz, 24 kHz, or 32 kHz. We can use MATLAB to play it and find the correct sampling rate. Try to run `exercise1_5.m` under the exercise directory. This script plays the file at 8 kHz, 12 kHz, 16 kHz, 24 kHz, and 32 kHz. Press the Enter key to continue if the program is paused. What is the correct sampling rate?

6. From the **Option** menu, set the CCS for automatically loading the program after the project has been built.

7. To reduce the number of mouse clicks, many pull-down menu items have been mapped to the hot buttons for the standard and advanced edit, project management, and debug toolbars. There are still some functions, however, that do not associate with any hot buttons. Use the **Option** menu to create shortcut keys for the following menu items:

(a) map **Go Main** in the debug menu to Alt + M (<Alt> and <M> keys);

(b) map **Reset** in the debug menu to Alt + R;

(c) map **Restart** in the debug menu to Alt + S; and

(d) map **Reload Program** in the file menu to Ctrl + R.

8. After loading the program into the simulator and enabling Source/ASM mixed display mode from View → Mixed Source/ASM, what is shown in the CCS source display window besides the C source code?

9. How do you change the format of displayed data in the watch window to hex, long, and floating-point format from the integer format?

10. What does File → Workspace do? Try the save and reload workspace commands.

11. Besides using file I/O with the probe point, data values in a block of memory space can also be stored to a file. Try the File → Data → Save and File → Data → Load commands.

12. Using Edit→Memory command we can manipulate (edit, copy, and fill) system memory of the `useCCS.pjt` in section 1.6.1 with the following tasks:

(a) open memory window to view `outBuffer`;

(b) fill `outBuffer` with data 0x5555; and

(c) copy the constant `sineTable[]` to `outBuffer`.

13. Use the CCS context-sensitive online help menu to find the TMS320C55x CUP diagram, and name all the buses and processing units.

14. We have introduced probe point for connecting files in and out of the DSP program. Create a project that will read in 16-bit, 32-bit, and floating-point data files into the DSP program. Perform multiplication of two data and write the results out via probe point.

15. Create a project to use `fgetc( )` and `fputc( )` to get data from the host computer to the DSP processor and write out the data back to the computer.

16. Use probe point to read the unknown 16-bit speech data file (`timit_1.asc`) and write it out in binary format (`timit_1.bin`).

17. Study the WAV file format and write a program that can create a WAV file using the PCM binary file (`timit_1.bin`) from above experiment. Play the created WAV file (`timit_1.wav`) on a personal computer's Windows Media Player.

18. Getting familiar with the DSK examples is very helpful. The DSK software package includes many DSP examples for the DSK. Use the DSK to run some of these examples and observe what these examples do.

2

Introduction to TMS320C55x Digital Signal Processor

To efficiently design and implement digital signal processing (DSP) systems, we must have a sound knowledge of DSP algorithms as well as DSP processors. In this chapter, we will introduce the architecture and programming of the Texas Instruments' TMS320C55x fixed-point processors.

2.1 Introduction

As introduced in Chapter 1, the TMS320 fixed-point processor family consists of C1x, C2x, C5x, C2xx, C54x, C55x, C62x, and C64x. In recent years, Texas Instruments have also introduced application-specific DSP-based processors including DSC2x, DM2xx, DM3xx, DM6xxxx, and OMAP (open multimedia application platform). DSC2x targets low-end digital cameras market. The digital medial processors aim at the rapid developing digital media markets such as portable media players, media centers, digital satellite broadcasting, simultaneously streaming video and audio, high-definition TVs, surveillance systems, as well as high-end digital cameras. The OMAP family is primarily used in wireless and portable devices such as new generation of cell phones and portable multimedia devices. Each generation of the TMS320 family has its own unique central processing unit (CPU) with variety of memory and peripheral configurations.

The widely used TMS320C55x family includes C5501, C5502, C5503, C5509, C5510, and so on. In this book, we use the TMS320C5510 as an example for real-time DSP implementations, experiments, and applications. The C55x processor is designed for low power consumption, optimum performance, and high code density. Its dual multiply-and-accumulate (MAC) architecture provides twice the cycle efficiency for computing vector products, and its scaleable instruction length significantly improves the code density. Some important features of the C55x processors are:

- 64-byte instruction buffer queue that works as an on-chip program cache to support implementation of block-repeat operations efficiently.

- Two 17-bit by17-bit MAC units can execute dual MAC operations in a single cycle.

- A 40-bit arithmetic-and-logic unit (ALU) performs high precision arithmetic and logic operations with an additional 16-bit ALU to perform simple arithmetic operations in parallel to the main ALU.

- Four 40-bit accumulators for storing intermediate computational results in order to reduce memory access.
- Eight extended auxiliary registers (XARs) for data addressing plus four temporary data registers to ease data processing requirements.
- Circular addressing mode supports up to five circular buffers.
- Single-instruction repeat and block-repeat operations of program for supporting zero-overhead looping.
- Multiple data variable and coefficient accesses in single instruction.

Detailed information of the TMS320C55x can be found in the references listed at the end of this chapter.

2.2 TMS320C55x Architecture

The C55x CPU consists of four processing units: instruction buffer unit (IU), program flow unit (PU), address-data flow unit (AU), and data computation unit (DU). These units are connected to 12 different address and data buses as shown in Figure 2.1.

2.2.1 Architecture Overview

IU fetches instructions from the memory into the CPU. The C55x instructions have different lengths for optimum code density. Simple instructions use only 8 bits (1 byte), while complicated instructions may contain as many as 48 bits (6 bytes). For each clock cycle, the IU fetches 4 bytes of instruction code via its 32-bit program-read data bus (PB) and places them into the 64-byte instruction buffer. At the same time, the instruction decoder decodes an instruction as shown in Figure 2.2. The decoded instruction is passed to the PU, AU, or DU.

The IU improves the program execution by maintaining instruction flow between the four units within the CPU. If the IU is able to hold a complete segment of loop code, the program execution can be repeated many times without fetching code from memory. Such capability not only improves the efficiency of loop execution, but also saves the power consumption by reducing memory accesses. The instruction buffer that can hold multiple instructions in conjunction with conditional program flow control is another advantage. This can minimize the overhead caused by program flow discontinuities such as conditional calls and branches.

PU controls program execution. As illustrated in Figure 2.3, the PU consists of a program counter (PC), four status registers, a program address generator, and a pipeline protection unit. The PC tracks the program execution every clock cycle. The program address generator produces a 24-bit address that covers 16 Mbytes of memory space. Since most instructions will be executed sequentially, the C55x utilizes pipeline structure to improve its execution efficiency. However, instructions such as branch, call, return, conditional execution, and interrupt will cause a nonsequential program execution. The dedicated pipeline protection unit prevents program flow from any pipeline vulnerabilities caused by a nonsequential execution.

AU serves as the data access manager. The block diagram illustrated in Figure 2.4 shows that the AU generates the data space addresses for data read and data write. The AU consists of eight 23-bit XARS (XAR0–XAR7), four 16-bit temporary registers (T0–T3), a 23-bit coefficient pointer (XCDP), and a 23-bit extended stack pointer (XSP). It consists of an additional 16-bit ALU that can be used for simple arithmetic operations. The temporary registers can be utilized to expand compiler efficiency by minimizing the need for memory access. The AU allows two address registers and a coefficient pointer

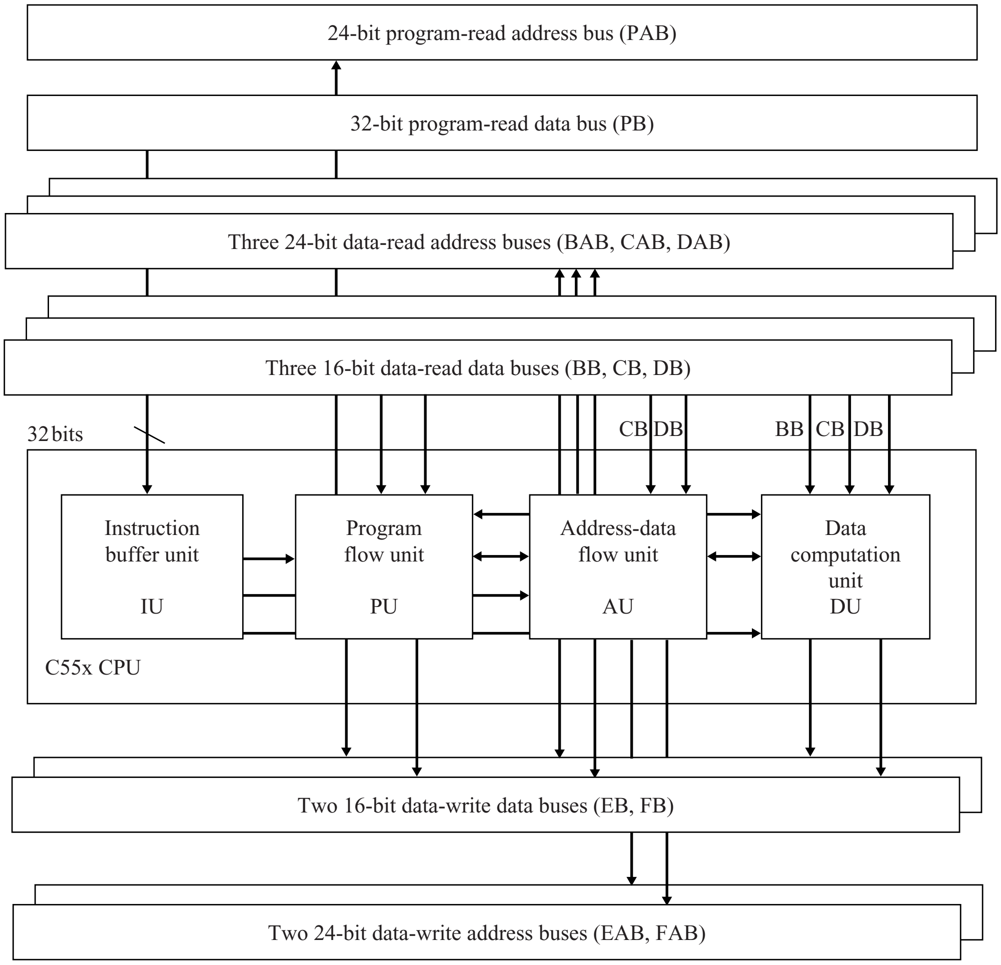

24-bit program-read address bus (PAB)
32-bit program-read data bus (PB)
Three 24-bit data-read address buses (BAB, CAB, DAB)
Three 16-bit data-read data buses (BB, CB, DB)
32 bits
CB DB
BB CB DB
Instruction buffer unit
IU
Program flow unit
PU
Address-data flow unit
AU
Data computation unit
DU
C55x CPU
Two 16-bit data-write data buses (EB, FB)
Two 24-bit data-write address buses (EAB, FAB)

Figure 2.1 Block diagram of TMS320C55x CPU

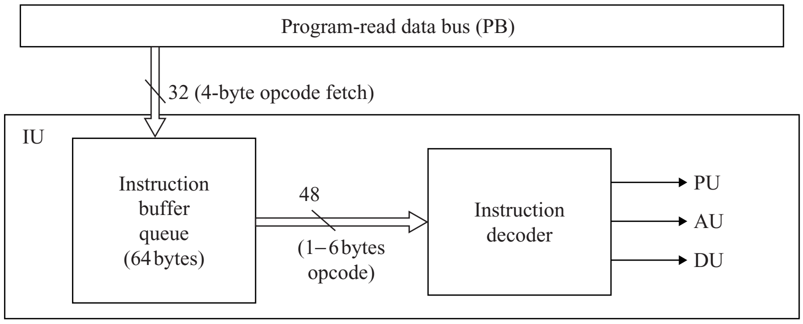


Figure 2.2 Simplified block diagram of the C55x IU

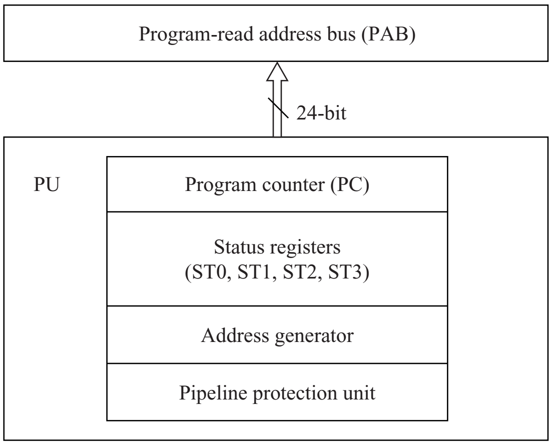


Figure 2.3 Simplified block diagram of the C55x PU

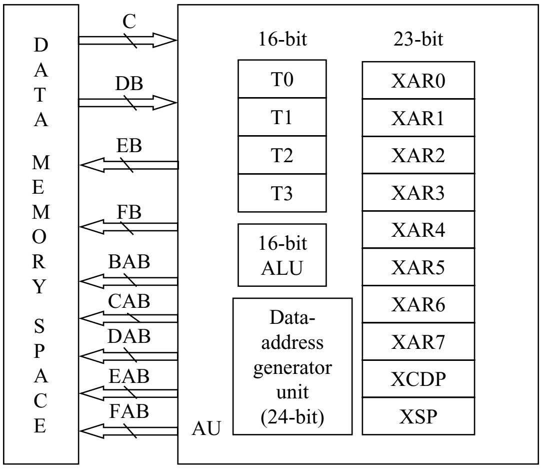


Figure 2.4 Simplified block diagram of the C55x AU

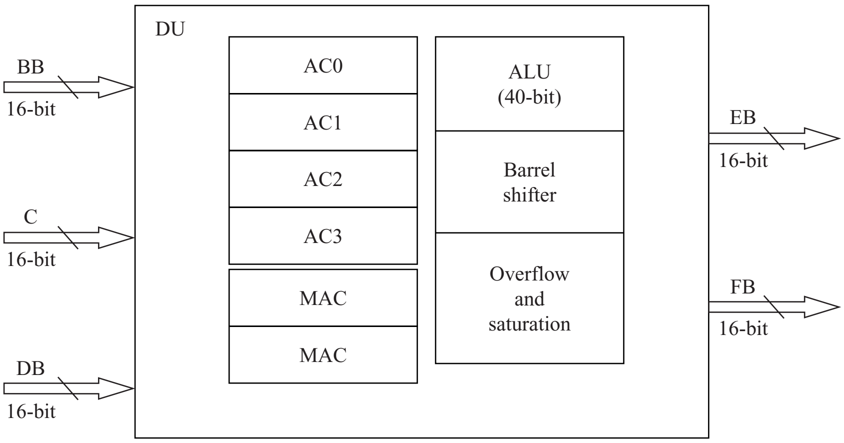


Figure 2.5 Simplified block diagram of the C55x DU

to be used together for some instructions to access two data samples and one coefficient in a single clock cycle. The AU also supports up to five circular buffers, which will be discussed later.

DU handles intensive computation for C55x applications. As illustrated in Figure 2.5, the DU consists of a pair of MAC units, a 40-bit ALU, four 40-bit accumulators (AC0, AC1, AC2, and AC3), a barrel shifter, and rounding and saturation control logic. There are three data-read data buses that allow two data paths and a coefficient path to be connected to the dual MAC units simultaneously. In a single cycle, each MAC unit can perform a 17-bit by 17-bit multiplication and a 40-bit addition (or subtraction) with saturation option. The ALU can perform 40-bit arithmetic, logic, rounding, and saturation operations using the accumulators. It can also be used to achieve two 16-bit arithmetic operations in both the upper and lower portions of an accumulator at the same time. The ALU can accept immediate values from the IU as data and communicate with other AU and PU registers. The barrel shifter may be used to perform data shift in the range of 2^{-32} (shift right 32 bits) to 2^{31} (shift left 31 bits).

2.2.2 Buses

As illustrated in Figure 2.1, the TMS320C55x has one program data bus, five data buses, and six address buses. The C55x architecture is built around these 12 buses. The program buses carry the instruction code and immediate operands from program memory, while the data buses connect various units. This architecture maximizes the processing power by maintaining separate memory bus structures for full-speed execution.

The program buses include a 32-bit PB and a 24-bit program-read address bus (PAB). The PAB carries the program memory address in order to read the code from the program space. The PB transfers 4 bytes of code to the IU at each clock cycle. The unit of program address used by the C55x processors is byte. Thus, the addressable program space is in the range of 0x000000–0xFFFFFF. (The prefix 0x indicates that the following numbers are in hexadecimal format.)

The data buses consist of three 16-bit data-read data buses (BB, CB, and DB) and three 24-bit data-read address buses (BAB, CAB, and DAB). This architecture supports three simultaneous data reads from data memory or I/O space. The CB and DB can send data to the PU, AU, and DU, while the BB can only work with the DU. The primary function of the BB is to connect memory to the dual MAC, so some specific operations can fetch two data and one coefficient simultaneously. The data-write operations use two 16-bit data-write data buses (EB and FB) and two 24-bit data-write address buses (EAB and FAB). For a single 16-bit data write, only the EB is used. A 32-bit data write will use both the the EB and the FB in one cycle. The data-write address buses (EAB and FAB) have the same 24-bit addressing range. The data memory space is 23-bit word addressable from address 0x000000 to 0x7FFFFF.

2.2.3 On-Chip Memories

The C55x uses unified program and data memory configurations with separated I/O space. All 16 Mbytes of memory space are available for program and data. The program memory space is used for program code, which is stored in byte units. The data memory space is used for data storage. The memory mapped registers (MMRs) also reside in data memory space. When the processor fetches instructions from the program memory space, the C55x address generator uses the 24-bit PAB. When the processor accesses data memory space, the C55x address generator masks off the least significant bit (LSB) of the data address line to ensure that the data is stored in memory in 16-bit word entity. The 16 Mbytes memory map is shown in Figure 2.6. Data space is divided into 128 data pages (0–127), and each page has 64 K words.

The C55x on-chip memory from addresses 0x0000 to 0xFFFF uses the dual access RAM (DARAM). The DARAM is divided into eight blocks of 8 Kbytes each, see Table 2.1. Within each block, C55x can perform two accesses (two reads, two writes, or one read and one write) per cycle. The on-chip DARAM can be accessed via the internal program bus, data bus, or direct memory access (DMA) buses. The DARAM is often used for frequently accessed data.

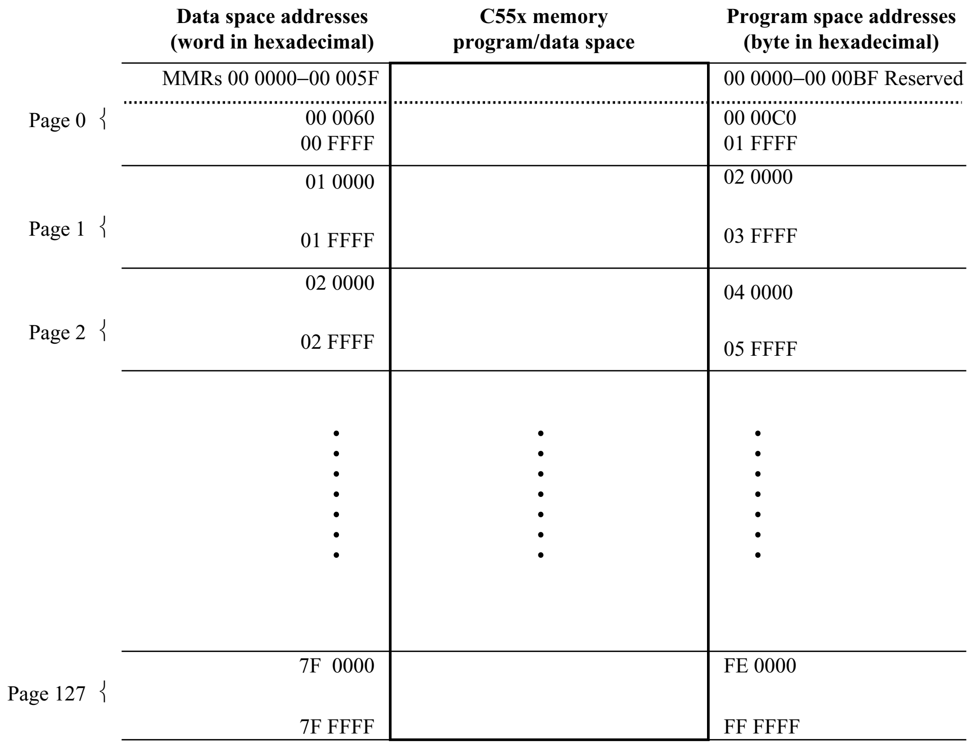


Figure 2.6 TMS320C55x program space and data space memory map

The C55x on-chip memory also includes single-access RAM (SARAM). The SARAM location starts from the byte address 0x10000 to 0x4FFFF. It consists of 32 blocks of 8 Kbytes each (see Table 2.2). Each access (one read or one write) will take one cycle. The C55x on-chip SARAM can be accessed by the internal program, data, or DMA buses.

The C55x contains an on-chip read-only memory (ROM) in a single 32 K-byte block. It starts from the byte address 0xFF8000 to 0xFFFFFF. Table 2.3 shows the addresses and contents of ROM in C5510. The bootloader provides multiple methods to load the program at power up or hardware reset. The bootloader uses vector table for placing interrupts. The 256-value sine lookup table can be used to generate sine function.

Table 2.1 C5510 DARAM blocks and addresses

DARAM byte address range	DARAM memory blocks
`0x0000–0x1FFF`	DARAM 0
`0x2000–0x3FFF`	DARAM 1
`0x4000–0x5FFF`	DARAM 2
`0x6000–0x7FFF`	DARAM 3
`0x8000–0x9FFF`	DARAM 4
`0xA000–0xBFFF`	DARAM 5
`0xC000–0xDFFF`	DARAM 6
`0xE000–0xFFFF`	DARAM 7

Table 2.2 C5510 DARAM blocks and addresses

SARAM byte address range	SARAM memory blocks
`0x10000–0x11FFF`	SARAM 0
`0x12000–0x13FFF`	SARAM 1
`0x14000–0x15FFF`	SARAM 2
:	:
:	:
:	:
`0x4C000–0x4DFFF`	SARAM 30
`0x4E000–0x4FFFF`	SARAM 31

2.2.4 Memory Mapped Registers

The C55x processor has MMRs for internal managing, controlling, and monitoring. These MMRs are located at the reserved RAM block from 0x00000 to 0x0005F. Table 2.4 lists all the CPU registers of C5510.

The accumulators AC0, AC1, AC2, and AC3 are 40-bit registers. They are formed by two 16-bit and one 8-bit registers as shown in Figure 2.7. The guard bits, AG, are used to hold data result of more than 32 bits to prevent overflow during accumulation.

The temporary data registers, T0, T1, T2, and T3, are 16-bit registers. They are used to hold data results less or equal to 16 bits. There are eight auxiliary registers, AR0-AR7, which can be used for several purposes. Auxiliary registers can be used as data pointers for indirect addressing mode and circular addressing mode. The coefficient data pointer (CDP) is a unique addressing register used for accessing coefficients via coefficient data bus during multiple data access operations. Stack pointer tracks the data memory address position at the top of the stack. The stack must be set with sufficient locations at reset to ensure that the system works correctly. Auxiliary registers, CDP register, and stack pointer register are all 23-bit registers. These 23-bit registers are formed by combining two independent registers (see Figure 2.8). The data in lower 16-bit portion will not carry into higher 7-bit portion of the register.

The C55x processor has four system status registers: ST0_C55, ST1_C55, ST2_C55, and ST3_C55. These registers contain system control bits and flag bits. The control bits directly affect the C55x operation conditions. The flag bits report the processor current status or results. These bits are shown in Figure 2.9, and see C55x reference guides for details.

2.2.5 Interrupts and Interrupt Vector

The C5510 has an interrupt vector that serves all the internal and external interrupts. Interrupt vector given in Table 2.5 lists the priorities for all internal and external interrupts. The addresses of the interrupts are the offsets from the interrupt vector pointer.

Table 2.3 C5510 ROM block addresses and contents

SARAM byte address range	SARAM memory blocks
`0xFF8000–0xFF8FFF`	Bootloader
`0xFF9000–0xFFF9FF`	Reserved
`0xFFFA00–0xFFFBFF`	Sine lookup table
`0xFFFC00–0xFFFEFF`	Factory test code
`0xFFFF00–0xFFFEFB`	Vector table
`0xFFFFFC–0xFFFEFF`	ID code

Table 2.4 C5510 MMRs

Reg.	Addr.	Function description	Reg.	Addr.	Function description
IER0	0x00	Interrupt mask register 0	DPH	0x2B	Extended data-page pointer
IFR0	0x01	Interrupt flag register 0		0x2C	Reserved
ST0_55	0x02	Status register 0 for C55x		0x2D	Reserved
ST1_55	0x03	Status register 1 for C55x	DP	0x2E	Memory data-page start address
ST3_55	0x04	Status register 3 for C55x	PDP	0x2F	Peripheral data-page start address
	0x05	Reserved	BK47	0x30	Circular buffer size register for AR[4–7]
ST0	0x06	ST0 (for 54x compatibility)	BKC	0x31	Circular buffer size register for CDP
ST1	0x07	ST1 (for 54x compatibility)	BSA01	0x32	Circular buffer start addr. reg. for AR[0–1]
AC0L	0x08	Accumulator 0 [15 0]	BSA23	0x33	Circular buffer start addr. reg. for AR[2–3]
AC0H	0x09	Accumulator 0 [31 16]	BSA45	0x34	Circular buffer start addr. reg. for AR[4–5]
AC0G	0x0A	Accumulator 0 [39 32]	BSA67	0x35	Circular buffer start addr. reg. for AR[6–7]
AC1L	0x0B	Accumulator 1 [15 0]	BSAC	0x36	Circular buffer coefficient start addr. reg.
AC1H	0x0C	Accumulator 1 [31 16]	BIOS	0x37	Data page ptr storage for 128-word data table
AC1G	0x0D	Accumulator 1 [39 32]	TRN1	0x38	Transition register 1
T3	0x0E	Temporary register 3	BRC1	0x39	Block-repeat counter 1
TRN0	0x0F	Transition register	BRS1	0x3A	Block-repeat save 1
AR0	0x10	Auxiliary register 0	CSR	0x3B	Computed single repeat
AR1	0x11	Auxiliary register 1	RSA0H	0x3C	Repeat start address 0 high
AR2	0x12	Auxiliary register 2	RSA0L	0x3D	Repeat start address 0 low
AR3	0x13	Auxiliary register 3	REA0H	0x3E	Repeat end address 0 high
AR4	0x14	Auxiliary register 4	REA0L	0x3F	Repeat end address 0 low
AR5	0x15	Auxiliary register 5	RSA1H	0x40	Repeat start address 1 high
AR6	0x16	Auxiliary register 6	RSA1L	0x41	Repeat start address 1 low
AR7	0x17	Auxiliary register 7	REA1H	0x42	Repeat end address 1 high
SP	0x18	Stack pointer register	REA1L	0x43	Repeat end address 1 low
BK03	0x19	Circular buffer size register	RPTC	0x44	Repeat counter
BRC0	0x1A	Block-repeat counter	IER1	0x45	Interrupt mask register 1
RSA0L	0x1B	Block-repeat start address	IFR1	0x46	Interrupt flag register 1
REA0L	0x1C	Block-repeat end address	DBIER0	0x47	Debug IER0
PMST	0x1D	Processor mode status register	DBIER1	0x48	Debug IER1
XPC	0x1E	Program counter extension register	IVPD	0x49	Interrupt vector pointer, DSP
	0x1F	Reserved	IVPH	0x4A	Interrupt vector pointer, HOST
T0	0x20	Temporary data register 0	ST2_55	0x4B	Status register 2 for C55x
T1	0x21	Temporary data register 1	SSP	0x4C	System stack pointer
T2	0x22	Temporary data register 2	SP	0x4D	User stack pointer
T3	0x23	Temporary data register 3	SPH	0x4E	Extended data-page pointer for SP and SSP
AC2L	0x24	Accumulator 2 [15 0]	CDPH	0x4F	Main data-page pointer for the CDP
AC2H	0x25	Accumulator 2 [31 16]			
AC2G	0x26	ccumulator 2 [39 32]			
CDP	0x27	Coefficient data pointer			
AC3L	0x28	Accumulator 3 [15 0]			
AC3H	0x29	Accumulator 3 [31 16]			
AC3G	0x2A	Accumulator 3 [39 32]			

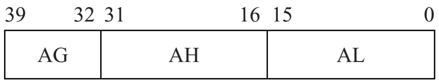


Figure 2.7 TMS320C55x accumulator structure

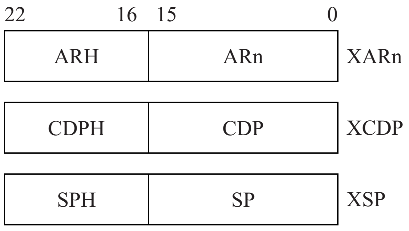


Figure 2.8 TMS320C55x 23-bit MMRs

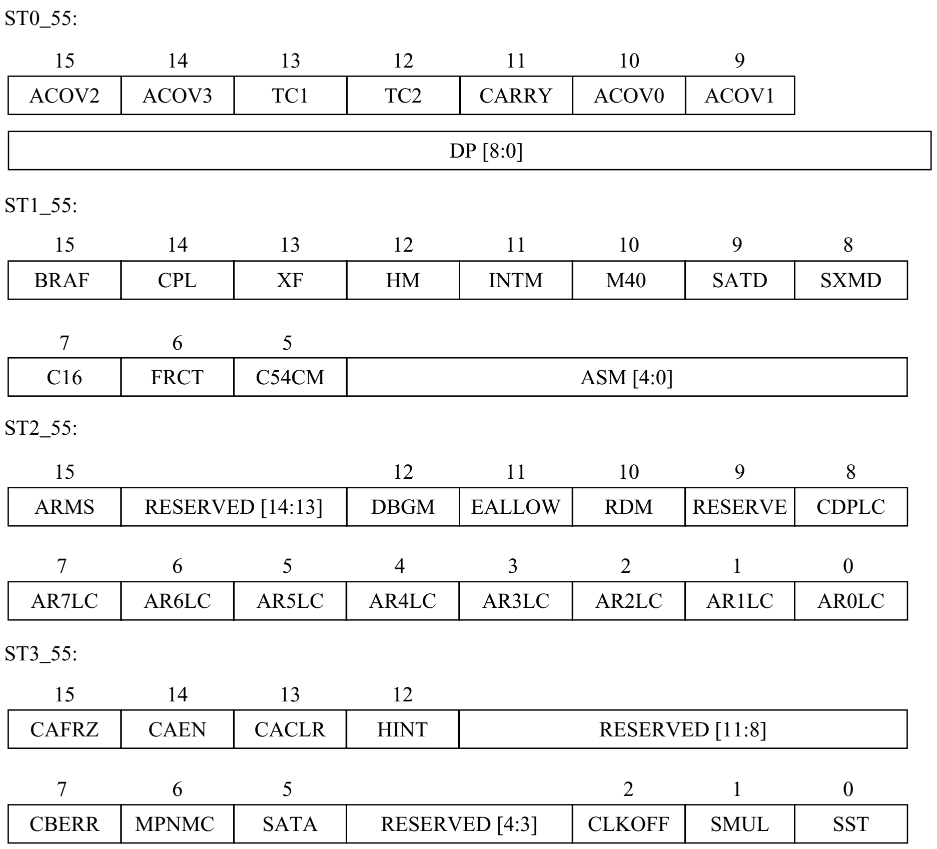


Figure 2.9 TME320C55x status registers

Table 2.5 C5510 interrupt vector

Name	Offset	Priority	Function description
RESET	0x00	0	Reset (hardware and software)
MNI	0x08	1	Nonmaskable interrupt
INT0	0x10	3	External interrupt #0
INT2	0x18	5	External interrupt #2
TINT0	0x20	6	Timer #0 interrupt
RINT0	0x28	7	McBSP #0 receive interrupt
RINT1	0x30	9	McBSP #1 receive interrupt
XINT1	0x38	10	McBSP #1 transmit interrupt
SINT8	0x40	11	Software interrupt #8
DMAC1	0x48	13	DMA channel #1 interrupt
DSPINT	0x50	14	Interrupt from host
INT3	0x58	15	External interrupt #3
RINT2	0x60	17	McBSP #2 receive interrupt
XINT2	0x68	18	McBSP #2 transmit interrupt
DMAC4	0x70	21	DMA channel #4 interrupt
DMAC5	0x78	22	DMA channel #5 interrupt
INT1	0x80	4	External interrupt #1
XINT0	0x88	8	McBSP #0 transmit interrupt
DMAC0	0x90	12	DMA channel #0 interrupt
INT4	0x98	16	External interrupt #4
DMAC2	0xA0	19	DMA channel #2 interrupt
DMAC3	0xA8	20	DMA channel #3 interrupt
TINT1	0xB0	23	Timer #1 interrupt
INT5	0xB8	24	External interrupt #5
BERR	0xC0	2	Bus error interrupt
DLOG	0xC8	25	Data log interrupt
RTOS	0xD0	26	Real-time operating system interrupt
SINT27	0xD8	27	Software interrupt #27
SINT28	0xE0	28	Software interrupt #28
SINT29	0xE8	29	Software interrupt #29
SINT30	0xF0	30	Software interrupt #30
SINT31	0xF8	31	Software interrupt #31

These interrupts can be enabled or disabled (masked) by the interrupt enable registers, IER0 and IER1. The interrupt flag registers, IFR0 and IFR1, indicate if an interrupt has occurred. The interrupt enable bits and flag bits assignments are given in Figure 2.10. When a flag bit of the IFR is set to 1, it indicates an interrupt has happened and that interrupt is pending to be served.

2.3 TMS320C55x Peripherals

In this section, we use the C5510 as an example to introduce some commonly used peripherals of the C55x processors. The C5510 consists of the following peripherals and the functional block diagram is shown in Figure 2.11.

- an external memory interface (EMIF);
- a six-channel DMA controller;
- a 16-bit parallel enhanced host-port interface (EHPI);

IFR0/IER

15	14	13	12	11	10	9	8
DMAC5	DMAC4	XINT2	RINT2	INT3	DSPINT	DMAC1	RESERV

7	6	5	4	3	2		
XINT1	RINT1	RINT0	TINT0	INT2	INT0	RESERVED [1:0]	

IFR1/IER:1

					10	9	8
RESERVED [15:11]					RTOS	DLOG	BER

7	6	5	4	3	2	1	0
INT5	TINT1	DMAC3	DMAC2	INT4	DMAC0	XINT0	INT1

Figure 2.10 TMS320C55x interrupt enable and flag registers

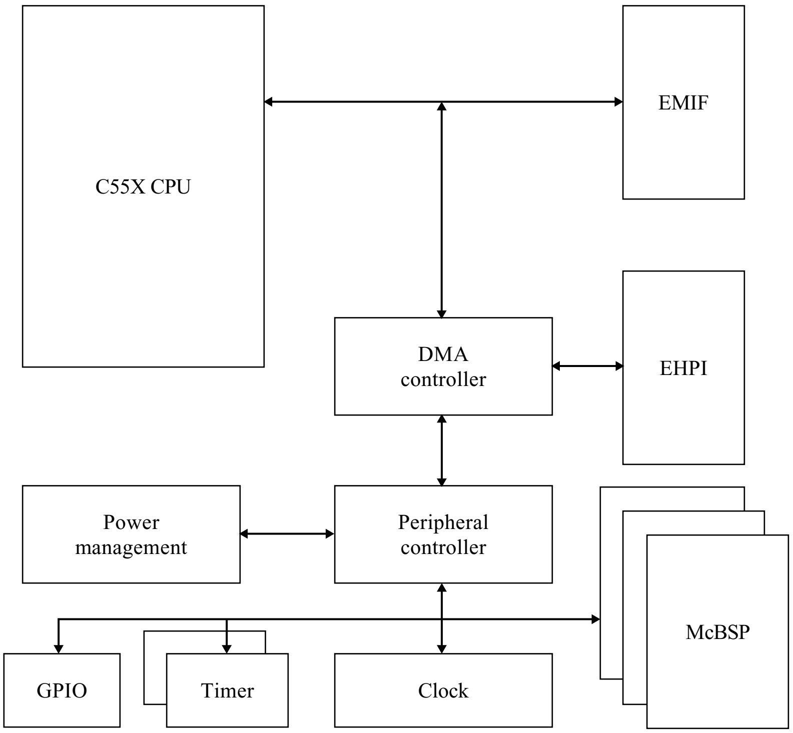


Figure 2.11 TMS320C55x functional blocks

- a digital phase-locked loop (DPLL) clock generator;
- two timers;
- three multichannel buffered serial ports (McBSP); and
- eight configurable general-purpose I/O (GPIO) pins;

2.3.1 External Memory Interface

The C55x EMIF connects the processor with external memory devices. The memory devices can be ROM, Flash, SRAM, synchronous burst SRAM (SBSRAM), and synchronous DRAM (SDRAM). The EMIF supports the program and data memory accesses at 32, 16, or 8 bit.

The C55x external memory is divided into four spaces according to chip enable (CE) settings (see Figure 2.12). The highest address block, 0xFF8000–0xFFFFFF, can be configured either as a continuous CE3 space or shared by internal processor ROM. The configuration depends upon the C55x status register MPNMC bit selection. A memory device must be physically connected to the proper CE pin of the EMIF. For example, an SDRAM memory's chip select pin must be connected to EMIF CE1_pin in order to be used in the CE1 memory space. The EMIF is managed by EMIF registers. The C5510 EMIF registers are given in Table 2.6. Each CE space can support either asynchronous or synchronous memory. For asynchronous memory, it can be a 32-, 16-, or 8-bit device. For synchronous memory, it supports SDRAM and SBSDRAM. More detailed description can be found in reference [7].

2.3.2 Direct Memory Access

The DMA is used to transfer data between the internal memory, external memory, and peripherals. Since the DMA data transfer is independent of the CPU, the C55x processor can simultaneously perform processing tasks at foreground while DMA transfers data at background. There are six DMA channels on

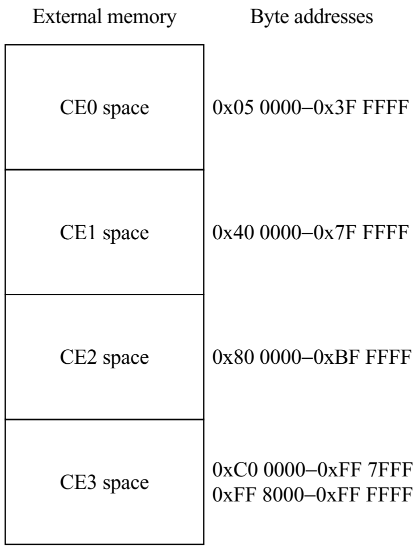


Figure 2.12 TMS320C55x EMIF

Table 2.6 C5510 EMIF registers

Register	Address	Function description
EGCR	0x0800	Global control register
EMI_RST	0x0801	Global reset register
EMI_BE	0x0802	Bus error status register
CE0_1	0x0803	CE0 space control register 1
CE0_2	0x0804	CE0 space control register 2
CE0_3	0x0805	CE0 space control register 3
CE1_1	0x0806	CE1 space control register 1
CE1_2	0x0807	CE1 space control register 2
CE1_3	0x0808	CE1 space control register 3
CE2_1	0x0809	CE2 space control register 1
CE2_2	0x080A	CE2 space control register 2
CE2_3	0x080B	CE2 space control register 3
CE3_1	0x080C	CE3 space control register 1
CE3_2	0x080D	CE3 space control register 2
CE3_3	0x080E	CE3 space control register 3
SDC1	0x080F	SDRAM control register
SDPER	0x0810	SDRAM period register
SDCNT	0x0811	SDRAM counter register
INIT	0x0812	SDRAM init register
SDC2	0x0813	SDRAM control register 2

the C55x processors, thus allowing up to six different operations. Each DMA channel has its own interrupt associated for event control. The DMA uses four standard ports for DARAM, SARAM, peripherals, and external memory. Each DMA channel's priority can be programmed independently. The data transfer source and destination addresses are programmable to provide more flexibility.

Table 2.7 lists the DMA synchronization events. Event sync is determined by the SYNC field in DMA channel control register, DMA_CCR. Channel 1–5 configuration registers are in the same order as channel 0. The DMA global registers and channel 0 configuration registers are given in Table 2.8.

2.3.3 Enhanced Host-Port Interface

The C5510 has an EHPI that allows a host processor to access C55x's internal DARAM and SARAM as well as portions of the external memory within its 20-bit address range. The range of 16-bit data access

Table 2.7 C5510 DMA synchronization events

SYNC field	SYNC event	SYNC field	SYNC event
0x00	No sync event	0x0B	Reserved
0x01	McBSP0 receive event REVT0	0x0C	Reserved
0x02	McBSP0 transmit event XEVT0	0x0D	Timer0 event
0x03	Reserved	0x0E	Timer1 event
0x04	Reserved	0x0F	External interrupt 0
0x05	McBSP1 receive event REVT0	0x10	External interrupt 1
0x06	McBSP1 transmit event XEVT0	0x11	External interrupt 2
0x07	Reserved	0x12	External interrupt 3
0x08	Reserved	0x13	External interrupt 4
0x09	McBSP2 receive event REVT0	0x14	External interrupt 5
0x0A	McBSP2 transmit event XEVT0		Reserved

Table 2.8 C5510 DMA configuration registers (channel 0 only)

Register	Address	Function description
		Global Register
DMA_GCR	0x0E00	DMA global control register
DMA_GSCR	0x0E02	EMIF bus error status register
		Channel #0 Registers
DMA_CSDP0	0x0C00	DMA channel 0 source/destination parameters register
DMA_CCR0	0x0C01	DMA channel 0 control register
DMA_CICR0	0x0C02	DMA channel 0 interrupt control register
DMA_CSR0	0x0C03	DMA channel 0 status register
DMA_CSSA_L0	0x0C04	DMA channel 0 source start address register (low bits)
DMA_CSSA_U0	0x0C05	DMA channel 0 source start address register (up bits)
DMA_CDSA_L0	0x0C06	DMA channel 0 source destination address register (low bits)
DMA_CDSA_U0	0x0C07	DMA channel 0 source destination address register (up bits)
DMA_CEN0	0x0C08	DMA channel 0 element number register
DMA_CFN0	0x0C09	DMA channel 0 frame number register
DMA_CSFI0	0x0C0A	DMA channel 0 source frame index register
DMA_CSEI0	0x0C0B	DMA channel 0 source element index register
DMA_CSAC0	0x0C0C	DMA channel 0 source address counter
DMA_CDAC0	0x0C0D	DMA channel 0 destination address counter
DMA_CDEI0	0x0C0E	DMA channel 0 destination element index register
DMA_CDFI0	0x0C0F	DMA channel 0 destination frame index register

starts from word address 0x00030 to 0xFFFFF, except the spaces for MMRs and peripheral registers. The address auto-increment capability improves the data transfer efficiency. The EHPI provides a 16-bit parallel data access between the host processor and the DSP processor. The data transfer is handled by the DMA controller. There are two configurations: nonmultiplexed mode and multiplexed mode. For the nonmultiplexed mode, the EHPI uses separated address and data buses while the multiplexed mode shares the same bus for both address and data. In order to pass data between C55x's peripherals (or MMRs) and host processor, the data must be first transferred to a shared memory that can be accessed by both the host processor and the DSP processor.

2.3.4 Multichannel Buffered Serial Ports

TMS320C55x processors use McBSP for direct serial interface with other serial devices connected to the system. The McBSP has the following key features:

- full-duplex communication;
- double-buffered transmission and triple-buffered reception;
- independent clocking and framing for receiving and transmitting;
- support external clock generation and sync frame signal;
- programmable sampling rate for internal clock generation and sync frame signal;

- support data size of 8, 12, 16, 20, 24, and 32 bits; and
- ability of performing μ-law and A-law companding.

The McBSP functional block diagram is shown in Figure 2.13. In the receive path, the incoming data is triple buffered. This allows one buffer to receive the data while other two buffers to be used by the processor. In the transmit path, double-buffer scheme is used. This allows one buffer of data to be transmitted out while the other buffer to be filled with new data for transmission. If the data width is 16-bit or less, only one 16-bit register will be used at each stage. These registers are DDR1, RBR1, and RSR1 in the receiving path, and DXR1 and XSR1 in the transmit path. When the data size is greater than 16 bits, two registers will be used at each stage of the data transfer. We will use the most commonly used 16-bit data transfer as an example to explain the functions of McBSP.

When a receive data bit arrives at C55x's DR pin, it will be shifted into the receive shift register (RSR). After all 16 receive bits are shifted into RSR, the whole word will be copied to the receive buffer register (RBR) if the previous data word in the RBR has already been copied. After the previous data in the data receive register (DRR) has been read, the RBR will copy its data to DRR for the processor or DMA to read. The data transmit process starts by the processor (or DMA controller) writing a data word to the data transmit register (DXR). After the last bit in the transmit shift register (XSR) is shifted out through the DX pin, the data in DXR will be copied into XSR. Using the McBSP's hardware companding feature, the linear data word can be compressed into 8-bit byte according to either μ-law or A-law standard while the received μ-law or A-law 8-bit data can be expanded to 16-bit linear data. The companding algorithm follows ITU G.711 recommendation.

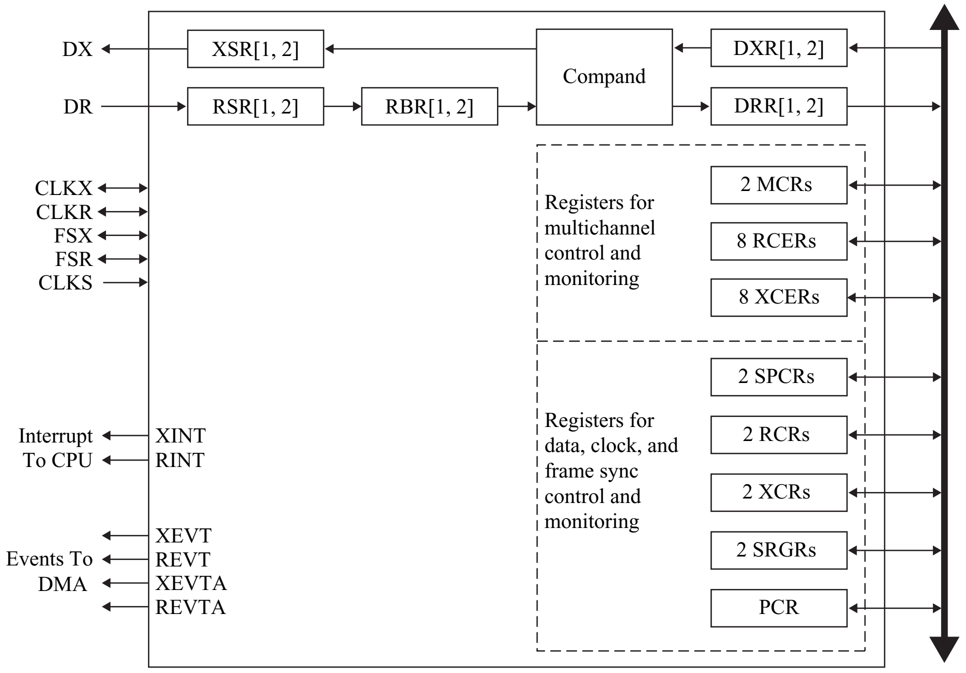


Figure 2.13 TMS320C55x McBSP functional block

As shown in Figure 2.13, the McBSP will send interrupt notification to the processor via XINT or RINT interrupt, and send important events to the DMA controller via REVT, XEVT, REVTA, and XEVTA. These pins are summarized as follows:

RINT – receive interrupt: The McBSP can send receive interrupt request to C55x according to a preselected condition in the receiver of the McBSP.

XINT – transmit interrupt: The McBSP can send transmit interrupt request to C55x according to a preselected condition in the transmitter of the McBSP.

REVT – receive synchronization event: This signal is sent to the DMA controller when a data is received in the DRR.

XEVT – transmit synchronization event: This signal is sent to the DMA controller when the DXR is ready to accept the next serial word data.

The C55x has three McBSPs: McBSP0, McBSP1, and McBSP2. Each McBSP has 31 registers. Table 2.9 lists the registers for McBSP0 as an example.

Table 2.9 Registers and addresses for the McBSP0

Register	Address	Function description
DRR2_0	0x2800	McBSP0 data receive register 2
DRR1_0	0x2801	McBSP0 data receive register 1
DXR2_0	0x2802	McBSP0 data transmit register 2
DXR1_0	0x2803	McBSP0 data transmit register 1
SPCR2_0	0x2804	McBSP0 serial port control register 2
SPCR1_0	0x2805	McBSP0 serial port control register 1
RCR2_0	0x2806	McBSP0 receive control register 2
RCR1_0	0x2807	McBSP0 receive control register 1
XCR2_0	0x2808	McBSP0 transmit control register 2
XCR1_0	0x2809	McBSP0 transmit control register 1
SRGR2_0	0x280A	McBSP0 sample rate generator register 2
SRGR1_0	0x280B	McBSP0 sample rate generator register 1
MCR2_0	0x280C	McBSP0 multichannel register 2
MCR1_0	0x280D	McBSP0 multichannel register 1
RCERA_0	0x280E	McBSP0 receive channel enable register partition A
RCERB_0	0x280F	McBSP0 receive channel enable register partition B
XCERA_0	0x2810	McBSP0 transmit channel enable register partition A
XCERB_0	0x2811	McBSP0 transmit channel enable register partition B
PCR0	0x2812	McBSP0 pin control register
RCERC_0	0x2813	McBSP0 receive channel enable register partition C
RCERD_0	0x2814	McBSP0 receive channel enable register partition D
XCERC_0	0x2815	McBSP0 transmit channel enable register partition C
XCERD_0	0x2816	McBSP0 transmit channel enable register partition D
RCERE_0	0x2817	McBSP0 receive channel enable register partition E
RCERF_0	0x2818	McBSP0 receive channel enable register partition F
XCERE_0	0x2819	McBSP0 transmit channel enable register partition E
XCERF_0	0x281A	McBSP0 transmit channel enable register partition F
RCERG_0	0x281B	McBSP0 receive channel enable register partition G
RCERH_0	0x281C	McBSP0 receive channel enable register partition H
XCERG_0	0x281D	McBSP0 transmit channel enable register partition G
XCERH_0	0x281E	McBSP0 transmit channel enable register partition H

Table 2.10 Registers and addresses for clock generator and timers

Register	Address	Function description
Clock Generator Register		
CLKMD	0x1C00	DMA global control register
Timer Registers		
TIM0	0x1000	Timer0 count register
PRD0	0x1001	Timer0 period register
TCR0	0x1002	Timer0 timer control register
PRSC0	0x1003	Timer0 timer prescale register
TIM1	0x2400	Timer1 count register
PRD1	0x2401	Timer1 period register
TCR1	0x2402	Timer1 timer control register
PRS10	0x2403	Timer1 timer prescale register

2.3.5 Clock Generator and Timers

The C5510 has a clock generator and two general-purpose timers. The clock generator takes an input clock signal from the CLKIN pin, and modifies this signal to generate the output clock signal for processor, peripherals, and other modules inside the C55x. The output clock signal is called the DSP (CPU) clock, which can be sent out via the CLKOUT pin. The clock generator consists of a DPLL circuit for high precision clock signal. An important feature of the clock generator is its idle mode for power conservation applications. The TMS320C55x has two general-purpose timers. Each timer has a dynamic range of 20 bits. The registers for clock generator and timers are listed in Table 2.10.

2.3.6 General Purpose Input/Output Port

TMS320C55x has a GPIO port, which consists of two I/O port registers, an I/O direction register, and an I/O data register. The I/O direction register controls the direction of a particular I/O pin. The C55x has eight I/O pins. Each can be independently configured as input or output. At power up, all the I/O pins are set as inputs. The I/O registers are listed in Table 2.11.

2.4 TMS320C55x Addressing Modes

The TMS320C55x can address 16 Mbytes of memory space using the following addressing modes:

- direct addressing mode;
- indirect addressing mode;
- absolute addressing mode;

Table 2.11 C5510 GPIO registers

Register	Address	Function description
IODIR	0x3400	GPIO direction register
IODATA	0x3401	GPIO data register

Table 2.12 C55x `mov` instruction with different operand forms

Instruction	Description
1. `mov #k,dst`	Load the 16-bit signed constant `k` to the destination register `dst`
2. `mov src,dst`	Load the content of source register `src` to the destination register `dst`
3. `mov Smem,dst`	Load the content of memory location `Smem` to the destination register `dst`
4. `mov Xmem,Ymem,ACx`	The content of `Xmem` is loaded into the lower part of `ACx`, while the content of `Ymem` is sign extended and loaded into upper part of `ACx`
5. `mov dbl(Lmem),pair(TAx)`	Load upper 16-bit data and lower 16-bit data from `Lmem` to the `TAx` and `TA(x+1)`, respectively
6. `amov #k23,xdst`	Load the effective address of `k23` (23-bit constant) into extended destination register (`xdst`)

- MMR addressing mode;
- register bits addressing mode; and
- circular addressing mode.

To explain these different addressing modes, Table 2.12 lists the move (`mov`) instruction with different syntaxes.

As illustrated in Table 2.12, each addressing mode uses one or more operands. Some of the operand types are explained as follows:

- `Smem` means a short data word (16-bit) from data memory, I/O memory, or MMRs.
- `Lmem` means a long data word (32-bit) from either data memory space or MMRs.
- `Xmem` and `Ymem` are used by an instruction to perform two 16-bit data memory accesses simultaneously.
- `src` and `dst` are source and destination registers, respectively.
- `#k` is a signed immediate constant; for example, `#k16` is a 16-bit constant ranging from –32768 to 32767.
- `dbl` is a memory qualifier for memory access for a long data word.
- `xdst` is an extended register (23-bit).

2.4.1 Direct Addressing Modes

There are four types of direct addressing modes: data-page pointer (DP) direct, stack pointer (SP) direct, register-bit direct, and peripheral data-page pointer (PDP) direct.

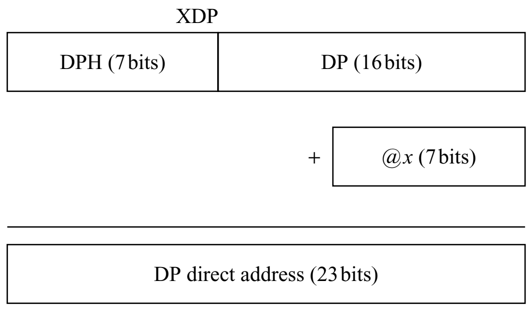


Figure 2.14 Using the DP-direct addressing mode to access variable x

The DP-direct addressing mode uses the main data page specified by the 23-bit extended data-page pointer (XDP). Figure 2.14 shows a generation of DP-direct address. The upper 7-bit DPH determines the main data page (0-127), and the lower 16-bit DP defines the starting address in the data page selected by the DPH. The instruction contains a 7-bit offset in the data page (`@x`) that directly points to the variable `x(Smem)`. The data-page registers DPH, DP, and XDP can be loaded by the `mov` instruction as

```
mov #k7,DPH    ; Load DPH with a 7-bit constant k7
mov #k16,DP    ; Load DP with a 16-bit constant k16
```

The first instruction loads the high portion of the extended data-page pointer, DPH, with a 7-bit constant `k7` to set up the main data page. The second instruction initializes the starting address of the DP. Example 2.1 shows how to initialize the DPH and DP pointers.

Example 2.1: Instruction

```
mov #0x3,DPH
mov #0x0100,DP
```

	Before instruction		After instruction
DPH	0	DPH	03
DP	0000	DP	0100

The XDP also can be initialized in one instruction using a 23-bit constant as

```
amov #k23,XDP ; Load XDP with a 23-bit constant
```

The syntax used in the assembly code is `amov #k23,xdst`, where `#k23` is a 23-bit address, the destination `xdst` is an extended register. Example 2.2 initializes the XDP to data page 1 with starting address 0x4000.

Example 2.2: Instruction

```
amov #0x14000, XDP
```

	Before instruction		After instruction
DPH	0	DPH	01
DP	0000	DP	4000

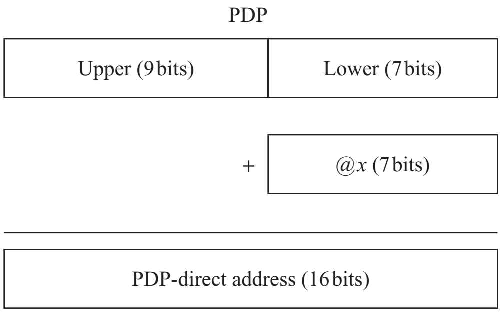


Figure 2.15 Using PDP-direct addressing mode to access variable x

The following code shows how to use DP-direct addressing mode:

```
X .set 0x1FFEF
  mov# 0x1,DPH       ; Load DPH with 1
  mov# 0x0FFEF,DP    ; Load DP with starting address
  .dp  X
  mov# 0x5555,@X     ; Store 0x5555 to memory location X
  mov# 0xFFFF,@(X+5) ; Store 0xFFFF to memory location X+5
```

In this example, the symbol `@` tells the assembler that this instruction uses the direct addressing mode. The directive `.dp` indicates the base address of the variable `X` without using memory space.

The SP-direct addressing mode is similar to the DP-direct addressing mode. The 23-bit address can be formed with the XSP in the same way as XDP. The upper 7 bits (SPH) select the main data page and the lower 16 bits (SP) determine the starting address. The 7-bit stack offset is contained in the instruction. When SPH = 0 (main page 0), the stack must not use the reserved memory space for MMRs from address 0 to 0x5F.

The I/O space addressing mode only has 16-bit addressing range. The 512 peripheral data pages are selected by the upper 9 bits of the PDP register. The 7-bit offset in the lower portion of the PDP register determines the location inside the selected peripheral data page as illustrated in Figure 2.15.

2.4.2 Indirect Addressing Modes

There are four types of indirect addressing modes. The AR-indirect mode uses one of the eight auxiliary registers as a pointer to data memory, I/O space, and MMRs. The dual AR indirect mode uses two auxiliary registers for dual data memory accesses. The CDP indirect mode uses the CDP for pointing to coefficients in data memory space. The coefficient-dual-AR indirect mode uses the CDP and the dual AR indirect modes for generating three addresses. The indirect addressing is the most frequently used addressing mode. It provides powerful pointer update and modification schemes as listed in Table 2.13.

The AR-indirect addressing mode uses auxiliary registers (AR0–AR7) to point to data memory space. The upper 7 bits of the XAR point to the main data page while the lower 16 bits point to a data location in that page. Since the I/O-space address is limited to a 16-bit range, the upper portion of the XAR must be set to zero when accessing I/O space. The maximum block size (32 K words) of the indirect addressing mode is limited by using 16-bit auxiliary registers. Example 2.3 uses the indirect addressing mode to copy the data stored in data memory, pointed by AR0, to the destination register AC0.

Table 2.13 AR and CDP indirect addressing pointer modification schemes

Operand	ARn/CDP pointer modifications
`*ARn` or `*CDP`	`ARn` (or `CDP`) is not modified
`*ARn±` or `*CDP±`	`ARn` (or `CDP`) is modified after the operation by: ±1 for 16-bit operation (`ARn=ARn ±1`) ±2 for 32-bit operation (`ARn=ARn±2`)
`*ARn(#k16)` or `*CDP(#k16)`	`ARn` (or `CDP`) is not modified The signed 16-bit constant `k16` is used as the offset from the base pointer `ARn` (or `CDP`)
`*+ARn(#k16)` or `*+CDP(#k16)`	`ARn` (or `CDP`) is modified before the operation The signed 16-bit constant `k16` is added as the offset to the base pointer `ARn` (or `CDP`) before generating new address
`*(ARn±T0/T1)`	`ARn` is modified after the operation by ±16-bit content in `T0` or `T1`, (`ARn=ARn±T0/T1`)
`*ARn(T0/T1)`	`ARn` is not modified `T0` or `T1` is used as the offset for the base pointer `ARn`

Example 2.3: Instruction

```
mov *AR0, AC0
```

AC0	00 0FAB 8678	AC0	00 0000 12AB
AR0	0100	AR0	0100
Data memory		Data memory	
0x100	12AB	0x100	12AB
	Before instruction		After instruction

The dual AR indirect addressing mode allows two data memory accesses through the auxiliary registers. It can access two 16-bit data in memory using the syntax

```
mov Xmem,Ymem,ACx
```

given in Table 2.12. Example 2.4 performs two 16-bit data loads with AR2 and AR3 as the data pointers to `Xmem` and `Ymem`, respectively. The data pointed at by AR3 is sign extended to 24 bits, loaded into the upper portion of the destination accumulator `AC0(39:16)`, and the data pointed at by AR2 is loaded into the lower portion of `AC0(15:0)`. The data pointers AR2 and AR3 are also updated.

Example 2.4: Instruction

```
mov *AR2+, *AR3-, AC0
```

AC0	FF FFAB 8678	AC0	00 3333 5555
AR2	0100	AR2	0101
AR3	0300	AR3	02FF
Data memory		Data memory	
0x100	5555	0x100	5555
0x300	3333	0x300	3333
	Before instruction		After instruction

The extended coefficient data pointer (XCDP) is the concatenation of the CDPH (the upper 7 bits) and the CDP (the lower 16 bits). The CDP-indirect addressing mode uses the upper 7 bits to define the main data page and the lower 16 bits to point to the memory location within the specified data page. Example 2.5 uses the CDP-indirect addressing mode where CDP contains the address of the coefficient in data memory. This instruction first increases the CDP pointer by 2, then loads a coefficient pointed by the updated coefficient pointer to the destination register AC3.

Example 2.5: Instruction

```
mov *+CDP (#2), AC3
```

AC3	00 0FAB EF45	AC3	00 0000 5631
CDP	0400	CDP	0402
Data memory		Data memory	
0x402	5631	0x402	5631
	Before instruction		After instruction

2.4.3 Absolute Addressing Modes

The memory can also be addressed using either `k16` or `k23` absolute addressing mode. The `k23` absolute mode specifies an address as a 23-bit unsigned constant. Example 2.6 shows an example of loading the data content at address `0x1234` on main data page `1` into the temporary register T2, where the symbol `*( )` represents the absolute addressing mode.

Example 2.6: Instruction

```
mov *(#x011234), T2
```

T2	0000	T2	FFFF
Data memory		Data memory	
0x01 1234	FFFF	0x01 1234	FFFF
	Before instruction		After instruction

The `k16` absolute addressing mode uses the operand `*abs(#k16)`, where `k16` is a 16-bit unsigned constant. In this mode, the DPH (7-bit) is forced to zero and concatenated with the unsigned constant `k16` to form a 23-bit data space memory address. The I/O absolute addressing mode uses the operand `port(#k16)`. The absolute address can also be the variable name such as the variable `x` in the following example:

```
    mov *(x),AC0
```

This instruction loads the accumulator AC0 with a content of variable `x`. When using absolute addressing mode, we do not need to worry about the DP. The drawback is that it needs more code space to represent the 23-bit addresses.

2.4.4 Memory Mapped Register Addressing Mode

The absolute, direct, and indirect addressing modes can be used to address MMRs, which are located in the data memory from address 0x0 to 0x5F on the main data page 0 as shown in Figure 2.6. To access MMRs using the `k16` absolute operand, the DPH must be set to zero. Example 2.7 uses the absolute addressing mode to load the 16-bit content of AR2 into the temporary register T2.

Example 2.7: Instruction

```
mov *abs16(#AR2), T2
```

AR2	1357	AR2	1357
T2	0000	T2	1357
	Before instruction		After instruction

For the MMR-direct addressing mode, the DP-direct addressing mode must be selected. Example 2.8 uses direct addressing mode to load the content of the lower portion of the accumulator AC0 (15:0) into the temporary register T0. When the `mmap()` qualifier is used for the MMR-direct addressing mode, it forces the data-address generator to access the main data page 0. That is, XDP = 0.

Example 2.8: Instruction

```
mov mmap16(@AC0L), T0
```

AC0	00 12DF 0202	AC0	00 12DF 0202
T0	0000	T0	0202
	Before instruction		After instruction

Accessing the MMRs using indirect addressing mode is the same as addressing the data memory space. Since the MMRs are located in data page 0, the XAR and XCDP must be initialized to page 0 by setting the upper 7 bits to zero. The following instructions load the content of AC0 into T1 and T2 registers:

```
amov #AC0H,XAR6

mov  *AR6-,T2

mov  *AR6+,T1
```

In this example, the first instruction loads the effective address of the upper portion of the accumulator AC0 (AC0H, located at address 0x9 on page 0) to XAR6. That is, XAR6 = 0x000009. The second instruction uses AR6 as a pointer to copy the content of AC0H into the T2 register, and then the pointer was decremented by 1 to point to the lower portion of AC0 (AC0L, located at address 0x8). The third instruction copies the content of AC0L into the register T1 and modifies AR6 to point to AC0H again.

2.4.5 Register Bits Addressing Mode

Both direct and indirect addressing modes can be used to address a bit or a pair of bits in a specific register. The direct addressing mode uses a bit offset to access a particular register's bit. The offset is the number of bits counting from the LSB. The instruction of register-bit direct addressing mode is shown in Example 2.9. The bit test instruction `btstp` will update the test condition bits (TC1 and TC2) of the status register ST0.

Example 2.9: Instruction

```
btstp @30, AC1
```

AC1	00 7ADF 3D05	AC1	00 7ADF 3D05
TC1	0	TC1	1
TC2	0	TC2	0
	Before instruction		After instruction

We can also use the indirect addressing modes to specify register bit(s) as follows:

```
mov# 2,AR4     ; AR4 contains the bit offset 2
bset* AR4,AC3  ; Set the AC3 bit pointed by AR4 to 1
btstp* AR4,AC1 ; Test AC1 bit-pair pointed by AR4
```

The register bits addressing mode supports only the bit test, bit set, bit clear, and bit complement instructions in conjunction with the accumulators (AC0–AC3), auxiliary registers (AR0–AR7), and temporary registers (T0–T3).

2.4.6 Circular Addressing Mode

Circular addressing mode updates data pointers in modulo fashion for accessing data buffers continuously without resetting the pointers. When the pointer reaches the end of the buffer, it will wrap back to the beginning of the buffer for the next iteration. Auxiliary registers (AR0–AR7) and the CDP can be used as circular pointers in indirect addressing mode. The following steps are used to set up circular buffers:

1. Initialize the most significant 7 bits of XAR (ARnH or CDPH) to select the main data page for a circular buffer. For example, `mov #k7,AR2H`.

2. Initialize the 16-bit circular pointer (ARn or CDP). The pointer can point to any memory location within the buffer. For example, `mov #k16,AR2`. The initialization of the address pointers in the examples of steps 1 and 2 can be combined using the single instruction: `amov #k23,XAR2`.

3. Initialize the 16-bit circular buffer starting address register (BSA01, BSA23, BSA45, BSA67, or BSAC) associated with the auxiliary registers. For example, if AR2 (or AR3) is used as the circular pointer, we have to use BSA23 and initialize it using `mov #k16,BSA23`. The main data page concatenated with the content of this register defines the 23-bit starting address of the circular buffer.

4. Initialize the data buffer size register (BK03, BK47, or BKC). When using AR0–AR3 (or AR4–AR7) as the circular pointer, BK03 (or BK47) should be initialized. The instruction `mov #16,BK03` sets up a circular buffer of 16 elements for the auxiliary registers AR0–AR3.

5. Enable the circular buffer by setting the appropriate bit in the status register ST2. For example, the instruction `bset AR2LC` enables AR2 for circular addressing.

The following example demonstrates how to initialize a circular buffer `COEFF[4]` with four integers, and how to use the circular addressing mode to access data in the buffer:

```
amov#COEFF,XAR2    ; Main data page for COEFF[4]
mov#COEFF,BSA23    ; Buffer base address is COEFF[0]
mov#0x4,BK03       ; Set buffer size of 4 words
mov#2,AR2          ; AR2 points to COEFF[2]
bset AR2LC         ; AR2 is configured as circular pointer
mov*AR2+,T0        ; T0 is loaded with COEFF[2]
mov*AR2+,T1        ; T1 is loaded with COEFF[3]
mov*AR2+,T2        ; T2 is loaded with COEFF[0]
mov*AR2+,T3        ; T3 is loaded with COEFF[1]
```

Since the circular addressing uses the indirect addressing modes, the circular pointers can be updated using the modifications listed in Table 2.13. The applications of using circular buffers for FIR filtering will be introduced in Chapter 4.

2.5 Pipeline and Parallelism

The pipeline technique has been widely used to improve DSP processors' performance. The pipeline execution breaks a sequence of operations into smaller segments and efficiently executes these smaller pieces in parallel to reduce the overall execution time.

2.5.1 TMS320C55x Pipeline

The C55x has two independent pipelines as illustrated in Figure 2.16: the program fetch pipeline and the program execution pipeline. The numbers on the top of the diagram represent the CPU clock cycle. The program fetch pipeline consists of the following three stages:

PA (program address): Instruction unit places the program address on the PAB.

PM (program memory address stable): The C55x requires one clock cycle for its program memory address bus to be stabilized before that memory can be read.

PB (program fetch from program data bus): Four bytes of the program code are fetched from the program memory via the 32-bit PB. The code is placed into the instruction buffer queue (IBQ).

At the same time, the seven-stage execution pipeline independently performs the sequence of fetch, decode, address, access, read, and execution. The C55x execution pipeline stages are summarized as follows:

F (fetch): An instruction is fetched from the IBQ. The size of the instruction varies from 1 byte up to 6 bytes.

D (decode): Decode logic decodes these bytes as an instruction or a parallel instruction pair. The decode logic will dispatch the instruction to the PU, AU, or DU.

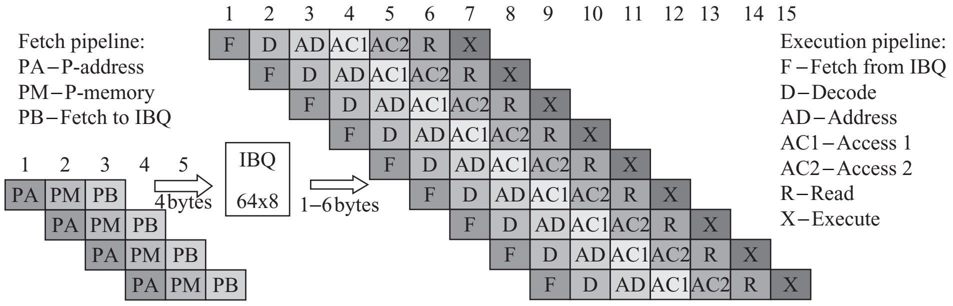


Figure 2.16 The C55x fetch and execution pipelines

AD (address): AU calculates data addresses using its data-address generation unit, modifies pointers if required, and computes the program space address for PC-relative branching instructions.

AC (access cycle 1 and 2): The first cycle is used to send the addresses to the data-read address buses (BAB, CAB, and DAB) for read operations, or transfer an operand to the processor via the CB. The second cycle is inserted to allow the address lines to be stabilized before the memory is read.

R (read): Data and operands are transferred to the processor via the CB for the `Ymem` operand, the BB for the `Cmem` operand, and the DB for the `Smem` or `Xmem` operands. For reading the `Lmem` operand, both the CB and the DB are used. The AU will generate the address for the operand write and send the address to the data-write address buses (EAB and FAB).

X (execute): Most data processing operations are done in this stage. The ALU inside the AU as well as the ALU and dual MAC inside the DU perform data processing, store an operand via the FB, or store a long operand via the EB and FB.

Figure 2.16 shows that the execution pipeline will be full after seven cycles, and every cycle that follows will complete the execution of one instruction. If the pipeline is always full, this technique increases the processing speed seven times. However, when a disturbing execution such as a branch instruction occurs, it breaks the sequential pipeline. Under such circumstances, the pipeline will be flushed and will need to be refilled. This is called pipeline breakdown. The use of IBQ can minimize the impact of pipeline breakdown. Proper use of conditional execution instructions to replace branch instructions can also reduce the pipeline breakdown.

2.5.2 Parallel Execution

The TMS320C55x uses multiple-bus architecture, dual MAC units, and separated PU, AU and DU for parallel execution. The C55x supports two types of parallel processing: implied (built-in) and explicit (user-built). The implied parallel instructions use the parallel columns symbol ': :' to separate the pair of instructions that will be processed in parallel. The explicit parallel instructions use the parallel bar symbol '||' to indicate the pair of parallel instructions. These two types of parallel instructions can be used together to form a combined parallel instruction. The following examples show the user-built, built-in, and combined parallel instructions that can be carried out in just one clock cycle.

User-built:

```
   mpym  *AR1+,*AR2+,AC0    ; User-built parallel instruction
|| and   AR4,T1             ;   using DU and AU
```

Built-in:

```
   mac *AR2-,*CDP-,AC0   ; Built-in parallel instruction
:: mac *AR3+,*CDP-,AC1   ;   using dual-MAC units
```

Built-in and user-built combination:

```
   mpy *AR2+,*CDP+,AC0   ; Combined parallel instruction
:: mpy *AR3+,*CDP+,AC1   ;   using dual-MAC units and PU
|| rpt #15
```

Table 2.14 Partial list of the C55x registers and buses

PU registers/buses	AU registers/buses	DU registers/buses
RPTC BRC0, BRC1 RSA0, RSA1 REA0, REA1	T0, T1, T2, T3 AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7 CDP BSA01, BSA23, BSA45, BSA67 BK01, BK23, BK45, BK67	AC0, AC1, AC2, AC3 TRN0, TRN1
Read buses: CB, DB Write buses: EB, FB	Read buses: CB, DB Write buses: EB, FB	Read buses: BB, CB, DB Write buses: EB, FB

Some of the restrictions for using parallel instructions are summarized as follows:

- For either the user-built or the built-in parallelism, only two instructions can be executed in parallel, and these two instructions must not exceed 6 bytes.
- Not all instructions can be used for parallel operations.
- When addressing memory space, only the indirect addressing mode is allowed.
- Parallelism is allowed between and within execution units, but there cannot be any hardware resources conflicts between units, buses, or within the unit itself.

There are several restrictions that define the parallelism within each unit when applying parallel operations in assembly code. The detailed descriptions are given in the TMS320C55x DSP Mnemonic Instruction Set Reference Guide.

The PU, AU, and DU can be involved in parallel operations. Understanding the register files and buses in each of these units will help to be aware of the potential conflicts when using the parallel instructions. Table 2.14 lists some of the registers and buses in PU, AU, and DU.

The parallel instructions used in the following example are incorrect because the second instruction uses the direct addressing mode:

```
   mov  *AR2,AC0
|| mov  T1,@x
```

We can correct this problem by replacing the direct addressing mode, `@x`, with an indirect addressing mode, `*AR1`, so both memory accesses are using indirect addressing mode as follows:

```
   mov  *AR2,AC0
|| mov  T1,*AR1
```

Consider the following example where the first instruction loads the content of AC0 that resides inside the DU to the auxiliary register AR2 inside the AU. The second instruction attempts to use the content of AC3 as the program address for a function call. Because there is only one link between AU and DU, when both instructions try to access the accumulators in the DU via the single link, it creates a conflict.

```
   mov   AC0,AR2
|| call  AC3
```

To solve this problem, we can change the subroutine call from call by accumulator to call by address as follows:

```
       mov   AC0,AR2
    || call  my_func
```

This is because the instruction `call my_func` uses only the PU.

The coefficient-dual-AR indirect addressing mode is used to perform operations with dual-AR indirect addressing mode. The coefficient indirect addressing mode supports three simultaneous memory accesses (`Xmem`, `Ymem`, and `Cmem`). The FIR filter (will be introduced in Chapter 4) is one of the applications that can effectively use coefficient indirect addressing mode. The following code is an example of using the coefficient indirect addressing mode:

```
       mpy *AR2+,*CDP+,AC2    ; AR1 pointer to data x1
    :: mpy *AR3+,*CDP+,AC3    ; AR2 pointer to data x2
    | | rpt #6                ; Repeat the following 7 times
       mac *AR2+,*CDP+,AC2    ; AC2 has accumulated result
    :: mac *AR3+,*CDP+,AC3    ; AC3 has another result
```

In this example, the memory buffers `Xmem` and `Ymem` are pointed at by AR2 and AR3, respectively, while the coefficient array is pointed at by CDP. The multiplication results are added with the contents in the accumulators AC2 and AC3.

2.6 TMS320C55x Instruction Set

In this section, we will introduce more C55x instructions for DSP applications. In general, we can divide the instruction set into four categories: arithmetic, logic and bit manipulation, move (load and store), and program flow control instructions.

2.6.1 Arithmetic Instructions

Arithmetic instructions include addition (`add`), subtraction (`sub`), and multiplication (`mpy`). The combination of these basic operations produces powerful subset of instructions such as the multiply–accumulation (`mac`) and multiply–subtraction (`mas`) instructions. Most arithmetic operations can be executed conditionally. The C55x also supports extended precision arithmetic such as add-with-carry, subtract-with-borrow, signed/signed, signed/unsigned, and unsigned/unsigned instructions. In Example 2.10, the instruction `mpym` multiplies the data pointed by AR1 and CDP, stores the product in the accumulator AC0, and updates AR1 and CDP after the multiplication.

Example 2.10: Instruction

```
mpym *AR1+, *CDP−, AC0
```

AC0	FF FFFF FF00	AC0	00 0000 0020
FRCT	0	FRCT	0
AR1	02E0	AR1	02E1
CDP	0400	CDP	03FF
Data memory		Data memory	
0x2E0	0002	0x2E0	0002
0x400	0010	0x400	0010
	Before instruction		After instruction

In Example 2.11, the `macmr40` instruction performs MAC operation using AR1 and AR2 as data pointers. At the same time, the instruction also carries out the following operations:

- The keyword '`r`' produces a rounded result in the higher portion of the accumulator AC3. After rounding, the lower portion of AC3(15:0) is cleared.
- 40-bit overflow detection is enabled by the keyword '`40`'. If overflow occurs, the result in accumulator AC3 will be saturated to a 40-bit maximum value.
- The option '`T3=*AR1+`' loads the data pointed at by AR1 into T3.
- Finally, AR1 and AR2 are incremented by 1 to point to the next data memory location.

Example 2.11: Instruction

```
macmr40 T3=*AR1+, *AR2+, AC3
```

AC3	00 0000 0020	AC3	00 235B 0000
FRCT	1	FRCT	1
T3	FFF0	T3	3456
AR1	0200	AR1	0201
AR2	0380	AR2	0381
Data memory		Data memory	
0x200	3456	0x200	3456
0x380	5678	0x380	5678
	Before instruction		After instruction

2.6.2 Logic and Bit Manipulation Instructions

Logic operation instructions such as `and`, `or`, `not`, and `xor` (exclusive-OR) on data values are widely used in decision-making and execution-flow control. They are also found in applications such as error correction coding in data communications, which will be introduced in Chapter 14. For example, the instruction

```
and #0xf,AC0
```

clears all upper bits in the accumulator AC0 but not the four LSBs.

Example 2.12: Instruction

```
and #0xf,AC0
```

AC0	00 1234 5678	AC0	00 0000 0008
	Before instruction		After instruction

The bit manipulation instructions act on an individual bit or a pair of bits of a register or data memory. These instructions include bit clear, bit set, and bit test to a specified bit or a bit pair. Similar to logic operations, the bit manipulation instructions are often used with logic operations in supporting decision-making processes. In Example 2.13, the bit-clear instruction clears the carry bit (bit 11) of the status register ST0.

Example 2.13: Instruction

```
blcr #11, ST0
```

ST0	0800	ST0	0000
	Before instruction		After instruction

2.6.3 Move Instruction

The move instruction copies data values between registers, memory locations, register to memory, or memory to register. Example 2.14 initializes the upper 16 bits of accumulator AC1 with a constant and clears the lower portion of the AC1. We can use the instruction

```
    mov #k<<16,AC1
```

where the constant `k` is shifted left by 16 bits first and then loaded into the upper portion of the accumulator AC1 (31:16), and the lower portion of the accumulator AC1 (15:0) is zero filled. The 16-bit constant that follows the # can be any 16-bit signed number.

Example 2.14: Instruction

```
mov #5<<16,AC1
```

AC1	00 0011 0800	AC1	00 0005 0000
	Before instruction		After instruction

A more complicated instruction given in Example 2.15 completes the following operations in one clock cycle:

- The unsigned data content in AC0 is shifted left according to the content in T2.
- The upper portion of the AC0 (31:16) is rounded.
- The data value in AC0 may be saturated if the left-shift or the rounding process causes the result in AC0 to overflow.
- The final result, after left shifting, rounding, and possible saturation, is stored into the data memory pointed at by the pointer AR1 as an unsigned value.
- Pointer AR1 is automatically incremented by 1.

Example 2.15: Instruction

```
mov uns (rnd(HI(satuate(AC0<<T2)))), *AR1+
```

AC0	00 0FAB 8678	AC0	00 0FAB 8678
AR1	0x100	AR1	0x101
T2	0x2	T2	0x2
Data memory		Data memory	
0x100	1234	0x100	3EAE
	Before instruction		After instruction

2.6.4 Program Flow Control Instructions

The program flow control instructions determine the execution flow of the program, including branching (`b`), subroutine call (`call`), loop operation (`rptb`), return to caller (`ret`), etc. These instructions can be either conditionally or unconditionally executed. Conditional call (`callcc`) along with conditional branch (`bcc`) and conditional return (`retcc`) can be used to control the program flow according to certain conditions. For example,

```
    callcc my_routine, TC1
```

is the conditional instruction that will call the subroutine `my_routine` only if the test control bit `TC1` of the status register ST0 is set.

The conditional execution instruction `xcc` can be implemented in either conditional execution or partial conditional execution. In Example 2.16, the conditional execution instruction tests the TC1 bit. If TC1 is set, the following instruction `mov *AR1+,AC0` will be executed, and both AC0 and AR1 are updated. If the condition is false, AC0 and AR1 will not be changed. Conditional execution instruction `xcc` allows conditional execution of one instruction or two paralleled instructions. The `label` is used for readability, especially when two parallel instructions are used.

Example 2.16:

Instruction

```
        xcc label, TC1
        mov *AR1+, AC0
    label
```

	TC1 = 1		TC1 = 0	
	Before instruction	After instruction	Before instruction	After instruction
AC0	00 0000 0000	00 0000 55AA	00 0000 0000	00 0000 0000
AR1	0100	0101	0100	0101
Data memory 0x100	55AA	55AA	55AA	55AA

In addition to conditional execution, the C55x also provides the capability of partial conditional execution of an instruction. An example of partial conditional execution is given in Example 2.17. When the condition is true, both AR1 and AC0 will be updated. However, if the condition is false, the execution phase of the pipeline will not be carried out. Since the first operand (the address pointer AR1) is updated in the read phase of the pipeline, AR1 will be updated whether or not the condition is true while the accumulator AC0 will remain unchanged at the execution phase. That is, the instruction is only partially executed.

Example 2.17:

Instruction

```
    xccpart label, TC1
    mov *AR1+, AC0
  label
```

	TC1 = 1		TC1 = 0	
	Before instruction	After instruction	Before instruction	After instruction
AC0	00 0000 0000	00 0000 55AA	00 0000 0000	00 0000 0000
AR1	0100	0101	0100	0101
Data memory 0x100	55AA	55AA	55AA	55AA

Many DSP applications, such as filtering, require repeated executions of instructions. These arithmetic operations may be located inside nested loops. If the number of instructions in the inner loop is small, the percentage of overhead for loop control may be very high compared with the instructions used in the inner loop. The loop-control instructions, such as testing and updating the loop counter(s), pointer(s), and branches back to the beginning of the loop, impose a heavy overhead for any tight loop processing. To minimize the overhead, the C55x provides built-in hardware for zero-overhead loop operations.

The single-repeat instruction (`rpt`) repeats the following single-cycle instruction or two single-cycle instructions that can be executed in parallel. For example,

```
rpt #N-1          ; Repeat next instruction N times
mov *AR2+,*AR3+
```

The immediate number, `N-1`, is loaded into the single-repeat counter (RPTC) by the `rpt` instruction. It allows the following instruction, `mov *AR2+,*AR3+`, to be executed `N` times.

The block-repeat instruction (`rptb`) forms a loop that repeats a block of instructions. It supports a nested loop with an inner loop being placed inside the outer loop. Block-repeat operations use block-repeat counters BRC0 and BRC1. For example,

```
      mov   #N-1,BRC0        ; Repeat outer loop N times
      mov   #M-1,BRC1        ; Repeat inner loop M times
      rptb  outloop-1        ; Repeat outer loop up to outloop
      mpy   *AR1+,*CDP+,AC0
      mpy   *AR2+,*CDP+,AC1
      rptb  inloop-1         ; Repeat inner loop up to inloop
      mac   *AR1+,*CDP+,AC0
      mac   *AR2+,*CDP+,AC1
 inloop                      ; End of inner loop
      mov   AC0,*AR3+        ; Save result in AC0
      mov   AC1,*AR4+        ; Save result in AC1
outloop                      ; End of outer loop
```

This example uses two block-repeat instructions to control nested repetitive operations. The following block-repeat structure

```
   rptb label_name-1
   (a block of instructions)
label_name
```

executes a block of instructions between the `rptb` instruction and the end label `label_name`. The maximum size of code that can be used inside a block-repeat loop is limited to 64 Kbytes, and the maximum number of times a loop can be repeated is limited to 65 536 (= 2^{16}) due to the 16-bit block-repeat counters. Because of the pipeline scheme, the minimum cycles within a block-repeat loop are two. For a single loop, the BRC0 should be used as the repeat counter. When implementing nested loops, the repeat counter BRC1 must be used for the inner-loop counter while BRC0 for the outer-loop counter. Since repeat counter BRC1 will be reloaded each time when it reaches zero, it only needs to be initialized once.

The local block-repeat structure is illustrated as follows:

```
   rptblocal label_name-1
   (Instructions of 56 bytes or less)
label_name
```

This has the similar structure as the previous block-repeat loop, but is more efficient because the loop code is placed inside the IBQ. Unlike the previous block-repeat loop, the local block-repeat loop fetches instructions from the memory once only. These instructions are stored in the IU and are used throughout the entire looping operation. The size of IBQ limits the size of local-repeat loop to 56 bytes or less.

Finally, we list some basic C55x mnemonic instructions in Table 2.15. The listed examples are just a small set of the rich C55x assembly instructions, especially the conditional instructions. The complete list of the mnemonic instruction set is provided by the TMS320C55x DSP mnemonic Instruction Set Reference Guide.

Table 2.15 TMS320C55x instruction set

Syntax	Meaning	Example	
`aadd`	Modify AR	`aadd`	`AR0, AR1`
`abdst`	Absolute distance	`abdst`	`*AR0+, *AR1+, AC0, AC1`
`abs`	Absolute value	`abs`	`AC0, AC1`
`add`	Addition	`add`	`uns(*AR4), AC1, AC0`
`addsub`	Addition and subtraction	`addsub`	`*AR3, AC1, TC2, AC0`
`amov`	Modify AR	`amov`	`AR0, AR1`
`and`	Bitwise AND	`and`	`AC0<#16, AC1`
`asub`	Modify AR	`asub`	`AR0, AR1`
`b`	Branch	`bcc`	`AC0`
`bclr`	Bit field clear	`bclr`	`AC2, *AR2`
`bcnt`	Bit filed counting	`bcnt`	`AC1, AC@, TC1, T1`
`bfxpa`	Bit filed expand	`bfxpa`	`#0x4022, AC0, T0`
`bfxtr`	Bit field extract	`bfxtr`	`#0x2204, AC0, T0`
`bnot`	Bit complement	`bnot`	`AC0, *AR3`
`band`	Bit field comparison	`band`	`*AR0, #0x0040, TC1`
`bset`	Bit set	`bset`	`INTM`
`btst`	Bit test	`btst`	`AC0, *AR0, TC2`
`btstclr`	Bit test and clear	`btstset`	`#0xA, *AR1, TC0`
`btstset`	Bit test and set	`btstset`	`#0x8, *AR3, TC1`
`call`	Function call	`call`	`AC1`
`delay`	Memory delay	`delay`	`*AR2+`
`cmp`	Compare	`cmp`	`*AR1+ == #0x200, TC1`
`firsadd`	FIR symmetric add	`firsadd`	`*AR0, *AR1, *CDP, AC0, AC1`
`firssub`	FIR symmetric sub	`firssub`	`*AR0, *AR1, *CDP, AC0, AC1`
`idle`	Force DSP to idle	`idle`	
`intr`	Software interrupt	`intr`	`#3`
`lms`	Least mean square	`lms`	`*AR0+, *AR1+, AC0, AC1`
`mant`	Normalization	`mant`	`AC0 ::`
`nexp AC0, T1`			
`mac`	Multiply–accumulate	`macr :: macr *AR3, *CDP, AC1`	`*AR2, *CDP, AC0`
`mack`	Multiply–accumulate	`mack`	`T0, #0xff00, AC0, AC1`
`mar`	Modify AR register	`amar`	`*AR0+, *AR1-, *CDP`
`mas`	Multiply–subtraction	`mas`	`uns(*AR2), uns(*CDP), AC0`
`max`	Maximum value	`max`	`AC0, AC1`
`maxdiff`	Compare and select maximum	`maxdiff`	`AC0, AC1, AC2, AC1`
`min`	Minimum value	`min`	`AC1, T0`
`mindiff`	Compare and select minimum	`mindiff`	`AC0, AC1, AC2, AC1`
`mov`	Move data	`mov`	`*AR3<<T0, AC0`

continues overleaf

Table 2.15 (*continued*)

Syntax	Meaning	Example	
mpy	Multiply	mpy	*AR2, *CDP, AC0
		:: mpy	*AR3, *CDP, AC1
mpyk	Multiply	mpyk	#-54, AC0, AC1
neg	Negate	neg	AC0, AC1
not	Bitwise complement	not	AC0, AC1
or	Bitwise OR	or	AC0, AC1
pop	POP from stack	popboth	AR5
psh	PSH to stack	psh	AC0
reset	Software reset	reset	
ret	Return	retcc	
reti	Return from interrupt	reti	
rol	Rotate left	rol	CARRY, AC1, TC2, AC1
ror	Rotate right	ror	TC2, AC0, TC2, AC1
round	Rounding	round	AC0, AC2
rpt	Repeat	rpt	#15
rptb	Repeat block	rptblocal	label-1
sat	Saturate	sat	AC0, AC1
sftl	Logic shift	sftl	AC2, #-1
sfts	Signed shift	sfts	AC0, T1, AC1
sqr	Square	sqr	AC1, AC0
sqdst	Square distance	sqdst	*AR0, *AR1, AC0, AC1
sub	Subtraction	sub	dual(*AR4), AC0, AC2
subadd	Subtraction and addition	subadd	T0, *AR0+, AC0
swap	Swap register	swap	AR4, AR5
trap	Software trap	trap	#5
xcc	Execute conditionally	xcc	*AR0 != #0
		add	*AR2+, AC0
xor	Bitwise XOR	xor	AC0, AC1

2.7 TMS320C55x Assembly Language Programming

We will introduce assembly programming in this section. The C55x assembly language source files usually contain assembler directives, macro directives, and assembly instructions.

2.7.1 Assembly Directives

Assembly directives control assembly processes such as the source file listing format, data alignment, and section contents. They also enter data to the program, initialize memory, define global variables, set conditional assembly blocks, and reserve memory space for code and data. Some commonly used C55x assembly directives are described in this section.

The .bss directive reserves uninitialized memory for data variables defined in the .bss section. It is often used to allocate data into RAM for run-time variables such as I/O buffers. For example,

```
.bss xn_buffer,size_in_words
```

where the `xn_buffer` points to the first location of the reserved memory space, and the `size_in_words` specifies the number of words to be reserved in the `.bss` section. If we do not specify names for uninitialized data sections, the assembler will put all the uninitialized data into the `.bss` section, which is word addressable.

The `.data` directive tells the assembler to begin assembling the source code into the `.data` section, which usually contains data tables or preinitialized data such as a sine table. The `.data` section is word addressable.

The `.sect` directive defines a code or data section and tells the assembler to begin assembling source code or data into that section. It is often used to divide long programs into logical partitions. It can separate subroutines from the main program, or separate constants that belong to different tasks. For example,

```
.sect "user_section"
```

assigns the code into the user-defined section called `user_section`. Code sections from different source files with the same section name are placed together.

Similar to the `.bss` directive, the `.usect` directive reserves memory space in an uninitialized section. It places data into user-defined sections. It is often used to divide large data sections into logical partitions, such as separating the transmitter variables from the receiver variables. The syntax for the `.usect` directive is

```
symbol .usect "section_name", size_in_words
```

where `symbol` is the variable, or the starting address of a data array, which will be placed into the section named `section_name`.

The `.text` directive tells the assembler to begin assembling source code into the `.text` section, which is the default section for program code. If we do not specify a code section, the assembler will put all the programs into the `.text` section.

The `.int` (or `.word`) directive places one or more 16-bit integer values into consecutive words in the current memory section. This allows users to initialize memory with constants. For example,

```
data1 .word 0x1234
data2 .int 1010111b
```

In these examples, `data1` is initialized to the hexadecimal number 0x1234 (decimal number 4660), while `data2` is initialized to the binary number 1010111b (decimal 87).

The `.set` (or `.equ`) directive assigns values to symbols. This type of symbols is known as assembly time constants. These symbols can then be used by source statements in the same manner as a numeric constant. The `.set` directive has the form

```
symbol .set value
```

where the `symbol` must appear in the first column. This equates the constant `value` to the `symbol`. The symbolic name used in the program will be replaced with the constant value by the assembler during assembly time, thus allowing programmers to write more readable programs. The `.set` and `.equ` directives can be used interchangeably.

The `.global` (`.def` or `.ref`) directive makes the symbols global to the external functions. The `.def` directive indicates a defined symbol that is given in the current file and known to other external files. The `.ref` directive references an external defined symbol that is defined in another file. The `.def` directive has the form

```
    .def symbol_name
```

The `symbol_name` can be a function name or a variable name that is defined in this file and can be referenced by other functions in different files. The `.global` directive can be interchanged with either `.def` or `.ref` directive.

The `.include` (or `.copy`) directive reads source file from another file. The `.include` directive has the form

```
    .include "file_name"
```

The `file_name` is used to tell the assembler which file to be read in as part of the source file.

2.7.2 Assembly Statement Syntax

The assembly statements may be separated into four ordered fields. The basic syntax for a C55x assembly statement is

```
    [label][:] mnemonic [operand list] [;comment]
```

The elements inside the brackets are optional. Statements must begin with a label, blank, asterisk, or semicolon. Each field must be separated by at least one blank. For ease of reading and maintenance, it is strongly recommended that we use meaningful mnemonics for labels, variables, and subroutine names. An example of C55x assembly statement is given in Table 2.16. The four ordered fields used in assembly program are explained as follows.

The label field can contain up to 32 alphanumeric characters (A–Z, a–z, 0–9, _, and $). It associates a symbolic address with a unique program location. The line that is labeled in the assembly program can then be referenced by the defined symbolic name. This is useful for modular programming and branch instructions. Labels are optional but, if used, they must begin in the first column. Labels are case sensitive and must start with an alphabetic letter or underscore. In the example depicted in Table 2.16, the symbol `start` is a label in the program's text section and is placed in the first column. The symbol `start` is defined as a global function entry point. Functions in other files are able to reference it. The `complex_data_loop` symbol is another label in the text section. This is a local label for setting up the block-repeat loop by the assembler. The symbol `stk_size` is a label used to indicate that the `.set` directive equates the constant `0x100` to be used as the stack size to allocate memory for the stack.

The mnemonic field can contain a mnemonic instruction, assembler directive, macro directive, or macro call. Note, the mnemonic field cannot start in the first column; otherwise, it would be interpreted as a label. In Table 2.16, mnemonic instructions `bset` and `bclr` are used to configure the C55x processor status registers.

The operand field is a list of operands. An operand can be a constant, symbol, or combination of constants and symbols in an expression. An operand can also be an assembly time expression that refers to memory, I/O ports, or pointers. Another category of the operands can be the registers and accumulators. Constants can be expressed in binary, decimal, or hexadecimal formats. For example, a binary constant is a string of binary digits (0s and 1s) followed by the suffix `B` (or `b`) and a hexadecimal constant is a string of hexadecimal digits (0, 1, ..., 9, A, B, C, D, E, and F) followed by the suffix `H` (or `h`). A hexadecimal number can also use the prefix `0x` similar to those used by C language. The prefix # is used to indicate an immediate constant. For example, #123 indicates that the operand is a constant of decimal number 123, while #0x53CD is the hexadecimal number of 53CD (equal to a decimal number of 21 453). Symbols defined in an assembly program with assembler directives may be labels, register names, constants, etc.

Table 2.16 An example of C55x assembly program

```
 ;
 ; Assembly program example
 ;
 N           .set 128
 stk_size    .set 0x100

 stack       .usect ".stack",stk_size  ; Stack
 sysstack    .usect ".stack",stk_size  ; System stack
 _Xin        .usect "in_data",(2*N)    ; Input data array
 _Xout       .usect "out_data",(2*N)   ; Output data array
 _Spectrum   .usect "out_data",N       ; Data spectrum array

         .sect data
input    .copy "input.inc"            ; Copy input.inc into program
         .def start                   ; Define this program' entry point
         .def _Xin,_Xout,_Spectrum    ; Make these data global data
         .ref _dft_128,_mag_128       ; Reference external functions

         .sect text
start
    bset SATD                         ; Set up saturation for D unit
    bset SATA                         ; Set up saturation for A unit
    bset SXMD                         ; Set up sign extension mode
    bclr C54CM                        ; Disable C54x compatibility mode
    bclr CPL                          ; Turn off compiler mode
    amov #(stack+stk_size),XSP        ; Setup DSP stack
    mov  #(sysstack+stk_size),SSP     ; Setup system stack
    mov  #N-1,BRC0                    ; Init counter for loop N times
    amov #input,XAR0                  ; Input data array pointer
    amov #_Xin,XAR1                   ; Xin array pointer
    rptblocal complex_data_loop-1     ; Form complex data
    mov *AR0+,*AR1+
    mov #0,   *AR1+
complex_data_loop
    amov #_Xin,XAR0            ; Xin array pointer
    amov #_Xout,XAR1           ; Xout array pointer
    call _dft_128              ; Perform 128-point DFT
    amov #_Xout,XAR0           ; Xout pointer
    amov #_Spectrum,XAR1       ; Spectrum array pointer
    call _mag_128              ; Compute squared-magnitude response
    .end
```

For example, we use the `.set` directive to assign a value to a symbol `N` as in the example given by Table 2.16. Thus, the symbol `N` becomes a constant value of 128 during assembly time.

The assembly instruction

```
    mov *AR0+,*AR1+
```

located inside the repeat loop copies the content pointed at by address pointer AR0 to a different memory location pointed at by address pointer AR1. The operand can also be a function name, such as

```
call _dft_128
```

Comments are notes about the program, which are significant. A comment can begin with an asterisk or a semicolon in column one. Comments that begin in any other column must begin with a semicolon. The following line

```
amov #(stack+stk_size),XSP ; Setup C55x stack
```

in Table 2.16 uses comment to note that this instruction is used to set up the C55x stack.

Assembly programming involves many considerations: allocating sections for program, data, constant, and variables; initializing the processor mode; deciding the proper addressing mode; and writing the assembly program. The example given in Table 2.16 has a text section, `.sect text`, where the assembly program code resides; a data section, `.sect data`, where a data file is copied into this program; and five uninitialized data sections, `.usect`, for stacks and data arrays. This example program uses indirect addressing mode to read in a set of data samples and performs discrete Fourier transform, which will be discussed in Chapter 6.

2.8 C Language Programming for TMS320C55x

In recent years, high-level languages such as C and C++ for DSP processors have become more popular and the C compilers are better designed to generate more efficient code. In this section, we will introduce some basic C programming considerations for TMS320C55x architecture.

2.8.1 Data Types

The C55x C compiler supports standard C data types. However, the C55x has some different data types as compared with other computer-based architectures (see Table 2.17).

For most general-purpose computers and microprocessors, C data type `char` is 8-bit but the C55x C data type `char` is 16-bit word. The C55x `long long` data type is 40-bit (as accumulators), not 64-bit as most of the general-purpose computers. Also, the data type `int` is defined as 16-bit data by C55x, while many other computers define `int` as 32-bit data. In order to write portable C programs, avoid using `char`, `int`, and `long long` if possible.

Table 2.17 C data types

	Data size (bits)	
Data type	C55x	Computer
`char`	16	8
`int`	16	32
`short`	16	16
`long`	32	32
`long long`	40	64
`float`	32	32
`double`	64	64

2.8.2 Assembly Code Generation by C Compiler

The C55x C compiler translates a C source code into an assembly source code first, the C55x assembler converts the assembly code into a binary object file, and the C55x linker combines it with other object files and library files to create an executable file. Knowing how the C compiler generates the assembly code can help us to write correct and efficient C programs.

Multiplication and addition are two most widely used statements in C programming for DSP applications. One common mistake of working with `short` variables is not using data type cast to force them into 32-bit data. For example, the correct C statement of multiplying two 16-bit numbers is

```
long mult_32bit;
short data1_16bit, data2_16bit;
mult_32bit = (long)data1_16bit * data2_16bit;
```

The C55x compiler will treat it as a 16-bit number times another 16-bit number with 32-bit product because the compiler preprocessor knows that both data operands are 16-bit. The following statements are written incorrectly for the C55x compiler:

```
mult_32bit = data1_16bit * data2_16bit;
mult_32bit = (long)(data1_16bit * data2_16bit);
add_32bit = data1_16bit + data2_16bit;
sub_32bit = (long)(data1_16bit - data2_16bit);
```

The following example shows the CCS debugger's mixed mode display. The C55x compiler generates the assembly code (in gray) that performs a 16-bit data `a` multiplied by another 16-bit data `b`. Only the last C statement

```
 c = (long)a * b;
```

generates the correct 32-bit result `c`.

```
short a=0x4000, b=0x6000;
long c;
void main()
{
   // Wrong: only the lower 16-bit result in AC0 is saved to c
   c = a * b;
   000100 A511018F     mov       *abs16(#018fh),T1
   000104 D31105018E   mpym      *abs16(#018eh),T1,AC0
   000109 A010 98      mov       mmap(AC0L),AC0
   00010C EB11080190   mov       AC0,dbl(*abs16(#0190h))

   // Wrong: only the lower 16-bit result in AC0 is saved to c
   c = (long)(a * b);
   000111 D31105018E   mpym      *abs16(#018eh),T1,AC0
   000116 A010 98      mov       mmap(AC0L),AC0
   000119 EB11080190   mov       AC0,dbl(*abs16(#0190h))

   // Correct: 32-bit result in AC0 is saved to c
   c = (long)a * b;
```

```
        00011E D31105018E   mpym       *abs16(#018eh),T1,AC0
        000123 EB11080190   mov        AC0,dbl(*abs16(#0190h))
    }
```

Understanding the C55x architecture also helps the compiler to generate more efficient code. For example, the loop statement such as

```
    for(i=0; i<count; i++)
```

is widely used in C programming for general-purpose computers. The C55x C compiler will generate different assembly code according to the data type of the loop counter, `i`. As discussed in Section 2.6, a for loop can be efficiently implemented using the C55x block-repeat instructions `rpt` and `rptb`. However, the block-repeat counters BRC0 and BRC1 are 16-bit registers, and this limits the loop counter to 16 bits. If we know that the loop will not exceed the maximum of 65535 ($2^{16}-1$) iterations, we should define the loop counter `i` as either `short` or `unsigned short`. If a `long` data type is used, the C55x compiler will not generate the loop using block-repeat instructions. The following example shows the CCS debugger's mixed mode display. The C55x compiler generates a more efficient code when the loop counter is defined as a 16-bit `short` variable.

```
    short a[300];

    void main()
    {
        long i;
        short *p=a, j;

            000263 760142B8       mov   #322,AR3
            000267 3C00           mov   #0,AC0

        // Inefficient for-loop using long loop counter i
        for (i=0; i<100; i++)
        {
            *p++ =  0;
            000269                L1:
            000269  E66300        mov  #0,*AR3+
            00026C  76006418      mov  #100,AC1
            000270  4010          add  #1,AC0
            000272  120410        cmp  AC0 < AC1, TC1
            000275  0464F1        bcc  L1,TC1
        }
        // More efficient for-loop using short loop counter j
        for (j=0; j<100; j++)
        {
            *p++ = 0;
            000278 4C63           rpt  #99
            00027A E66300         mov  #0,*AR3+
        }
    }
```

Avoiding compiler built-in supporting library functions can also improve the real-time efficiency. This is because the use of library functions requires function calls, which need to set up and pass the arguments before the call and collect return values. Also, some of the C55x registers need to be saved by the caller function so that they can be used by the library functions. An example of modulo statement in C is listed below. It is clear that the C55x C compiler will generate more efficient code when using $2^n - 1$ for modulo-2^n operation.

```
void main()
{
  short a=30;
     000102 E6001E        mov       #30,*SP(#00h)

 // Modulo operation calls library function
    a = a % 7;
    000105  A400 3D75     mov       *SP(#00h),T0 ||  mov #7,T1
    000109  6C000297      call      __remi
    00010D  C400          mov       T0,*SP(#00h)

 // Inefficient modulo operation of a power of 2 number
    a = a % 8;
    00010F  2240 3D7A     mov       T0,AC0  ||  mov #7,AR2
    000113  36AA 11453E   not       AR2,AR2 ||  sfts AC0,#-2,AC1
    000118  76E00091      bfxtr     #57344,AC1,AR1
    00011C  2409          add       AC0,AR1
    00011E  289A          and       AR1,AR2
    000120  26A0          sub       AR2,AC0
    000122  C000          mov       AC0,*SP(#00h)

 // Efficient modulo operation of a power of 2 number
    a = a & (8-1);
    000124  F4000007      and       #7,*SP(#00h)

}
```

2.8.3 Compiler Keywords and Pragma Directives

The C55x C compiler supports the `const` and `volatile` keywords. The `const` keyword controls data objects allocation. It ensures that the data objects will not be changed by placing constant data into ROM space. The `volatile` keyword is especially important when the compiler optimization feature is turned on. This is because the optimizer may aggressively rearrange code around and may even remove segments of code and data variables. However, the optimizer will keep all the variables with `volatile` keyword.

The C55x compiler also supports three new keywords: `ioport`, `interrupt`, and `onchip`. The keyword `ioport` is used for compiler to distinguish memory space related to the I/O. The peripheral registers, such as EMIF, DMA, Timer, McBSP, etc., are all located in the I/O memory space. To access these registers, we must use the `ioport` keyword. The `ioport` keyword can only be applied to global variables and used for the local or global pointers. Since I/O is only addressable in 16-bit range, all variables including pointers are 16-bit even if the program is compiled for the large memory model.

Interrupts and interrupt services are common for DSP programs in real-time applications. Due to the fact that interrupt service routine (ISR) requires specific register handling and relies on special sequences to entry and return, the `interrupt` keyword supported by the C55x compiler specifies the function that is actually an ISR.

To utilize its dual MAC feature in C, the C55x compiler uses a keyword `onchip` to qualify the memory that may be used by dual MAC instructions. This memory must locate at the on-chip DARAM.

Table 2.18 C program example of enabling DPLL

```
// Function of enabling or disabling clock generator PLL

#define  PLLENABLE_SET    1      // PLL enable

#define  CLKMD_ADDR    0x1c00
#define  CLKMD        (ioport volatile unsigned short *)CLKMD_ADDR

#pragma  CODE_SECTION(pllEnable, ".text:C55xCode'');

void pllEnable(short enable)
{
    short clkModeReg;
    clkModeReg = *CLKMD;
    if (enable)
        *CLKMD = clkModeReg |  (PLLENABLE_SET<<4);
    else
        *CLKMD = clkModeReg & 0xFFEF;

}
```

The `CODE_SECTION` pragma allocates a program memory space and places the function associated with the pragma into that section. The `DATA_SECTION` pragma allocates a data memory space and places the data associated with the pragma into that data section instead of the `.bss` section.

Table 2.18 shows a C program example that enables the DPLL. In this example, the function `pllEnable` is placed in the program memory space inside the `.text` section with a subsection of `C55xCode`.

2.9 Mixed C-and-Assembly Language Programming

As discussed in Chapter 1, the mixing of C and assembly programs is used for many DSP applications. High-level C code provides the ease of maintenance and portability, while assembly code has the advantages of run-time efficiency and code density. In this section, we will introduce how to interface C with assembly programs, and review the guidelines of the C function calling conventions. The assembly routines called by a C function can have arguments and return values just like C functions. The following guidelines are used for writing the C55x assembly code that is callable by C functions:

Naming convention: Use the underscore '_' as a prefix for all variables and routine names that will be accessed by C functions. For example, use `_asm_func` as the name of an assembly routine called by a C function. If a variable is defined in the assembly routine, it must use the underscore prefix for C function to access it, such as `_asm_var`. The prefix '_' is used by the C compiler only. When we access assembly routines or variables from C functions, we do not need to use the underscore as a prefix.

Variable definition: The variables that are accessed by both C and assembly routines must be defined as global variables using the directive `.global`, `.def`, or `.ref` by the assembler.

Compiler mode: By using the C compiler, the C55x CPL (compiler mode) bit is automatically set for using stack pointer relative addressing mode when entering an assembly routine. The indirect addressing modes are preferred under this configuration. If we need to use direct addressing modes to access data memory in a C-callable assembly routine, we must change to DP-direct addressing mode. This can be done by clearing the CPL bit. However, before the assembly routine returns to its C caller function, the CPL bit must be restored. The bit clear and bit set instructions, `bclr CPL` and `bset CPL`, can be used to reset and set the CPL bit in the status register ST1, respectively. The following example code can be used to check the CPL bit, turn CPL bit off if it is set, and restore the CPL bit before returning it to the caller:

```
     btstclr #14,*(ST1),TC1 ; Turn off CPL bits if it is set
     (more instructions ...)
     xcc continue,TC1       ; TC1 is set if we turn off CPL bit
     bset CPL               ; Turn on CPL bit
  continue
     ret
```

Passing arguments: To pass arguments from a C function to an assembly routine, we must follow the strict rules of C-callable conversions set by the C55x compiler. When passing an argument, the C compiler assigns it to a particular data type and then places it using a register according to its data type. The C55x C compiler uses the following three classes to define the data types:

- *Data pointer*: `short *`, `int *`, or `long *`;
- *16-bit data*: `char`, `short`, or `int`; and
- *32-bit data*: `long`, `float`, `double`, or function pointers.

If the arguments are pointers to data memory, they are treated as data pointers. If the argument can fit into a 16-bit register such as `short`, `int`, and `char`, it is considered to be 16-bit data. Otherwise, it is 32-bit data. The arguments can also be structures. A structure of two words (32 bits) or less is treated as a 32-bit data argument and is passed using a 32-bit register. For structures larger than two words, the arguments are passed by reference. The C compiler will pass the address of a structure as a pointer, and this pointer is treated like a data argument.

For a subroutine call, the arguments are assigned to registers in the order that the arguments are listed by the function. They are placed in the registers according to their data type, in the order shown in Table 2.19.

Table 2.19 shows the overlap between the AR registers used for data pointers and the registers used for 16-bit data. For example, if T0 and T1 hold 16-bit data arguments and AR0 already holds a data-pointer argument, a third 16-bit data argument would be placed into AR1 (see the second example in Figure 2.17). If the registers of the appropriate type are not available, the arguments are passed onto the stack (see the third example given in Figure 2.17).

Table 2.19 Argument classes assigned to registers

Argument type	Register assignment order
16-bits data pointer	AR0, AR1, AR2, AR3, AR4
23-bit data pointer	XAR0, XAR1, XAR2, XAR3, XAR4
16-bit data	T0, T1, AR0, AR1, AR2, AR3, AR4
32-bit data	AC0, AC1, AC2

```
T0           T0      AC0      AR0
↓            ↓       ↓        ↓
short  func (short i1,  long i2,  short *p3);


AC0                  AR0        T0         T1         AR1
 ↓                    ↓         ↓          ↓          ↓
long func (short *p1, short i2, short i3, short i4);


           AC0  AC1  AC2  Stack  T0
            ↓    ↓    ↓     ↓     ↓
void  func(long l1, ling l2, long l3, long l4, short i5);
```

Figure 2.17 Examples of arguments passing conventions

Return values: The calling function collects the return values from the called function/subroutine. A 16-bit data is returned in the register T0, and a 32-bit data is returned in the accumulator AC0. A data pointer is returned in (X)AR0, and a structure is returned on the local stack.

Register use and preservation: When making a function call, the register assignments and preservations between the caller and the called functions are strictly defined. Table 2.20 describes how the registers are preserved during a function call. The called function must save the contents of the save-on-entry registers (T2, T3, AR5, AR6, and AR7) if it will use these registers. The calling function must push the contents of any other save-on-call registers onto the stack if these registers' contents are needed after the function/subroutine call. A called function can freely use any of the save-on-call registers (AC0–AC3, T0, T1, and AR0–AR4) without saving its value. More detailed descriptions can be found in the TMS320C55x Optimizing C Compiler User's Guide.

As an example of mixed C-and-assembly programming, the following C program calls an assembly function `findMax( )` that returns the maximum value of given array:

```
extern short findMax(short *p, short n);
void main()
{
   short a[8]={19, 55, 2, 28, 19, 84, 12, 10};
```

Table 2.20 Register use and preservation conventions

Registers	Preserved by	Used for
AC0–AC2	Calling function Save-on-call	16-, 32-, or 40-bit data 24-bit code pointers
(X)AR0–(X)AR4	Calling function Save-on-call	16-bit data 16- or 23-bit pointers
T0 and T1	Calling function Save-on-call	16-bit data
AC3	Called function Save-on-entry	16-, 32-, or 40-bit data
(X)AR5–(X)AR7	Called function Save-on-entry	16-bit data 16- or 23-bit pointers
T2 and T3	Called function Save-on-entry	16-bit data

```
     static short max;

     max = findMax(a, 8);
  }
```

The assembly function `findMax( )` is listed as follows:

```
;
;   Function prototype:
;     short findMax(short *p, short n);
;
;   Entry: AR0 is the pointer of p and T0 contains length n
;   Exit:  T0 contains the maximum data value found
;
    .def   _findMax      ; Using "_" prefix for C-callable
    .text
_findMax:
    sub  #2,T0
    mov  T0,BRC0         ; Setup up loop counter
    mov  *AR0+,T0        ; Place the first data in T0
||  rptblocal loop-1     ; Loop through entire array
    mov  *AR0+,AR1       ; Place the next data in AR1
    cmp  T0<AR1,TC1      ; Check to see if new data is greater
||  nop                  ;   than the maximum?
    xccpart TC1          ; If find new maximum, place it in T0
||  mov  AR1,T0
loop
  ret                    ; Return with maximum data in T0
```

The assembly routine `findMax` returns the 16-bit maximum value of the array pointed by the data pointer `*p`. The assembly function uses the underscore '_' as the prefix for function name. The first argument is a 16-bit data pointer and it is passed via auxiliary register AR0. The array size is the second argument, which is a 16-bit data and is passed via the temporary register T0. The return value is a 16-bit data in T0 register.

2.10 Experiments and Program Examples

In this section, we will use the CCS and DSK to show the programming considerations of the C55x processors.

2.10.1 Interfacing C with Assembly Code

In this experiment, we will learn how to write C-callable assembly routines. The C function `main` calls the assembly routine `sum` to add two elements in an array, and returns the sum to the C-calling function. The C program is listed as follows:

```
extern short sum(short *);  /* Assembly routine sum        */
short x[2]={0x1234,0x4321}; /* Define x[] as global array  */
short s;                    /* Define s as global variable */
```

Table 2.21 File listing for experiment `exp2.10.1_C_ASM`

Files	Description
`c_asmTest.c`	C function for testing the experiment
`sum.asm`	Assembly function to compute the sum
`c_asm.pjt`	DSP project file
`c_asm.cmd`	DSP linker command file

```
void main()
{
   s = sum(x);               /* Call assembly routine _sum  */
}
```

The assembly routine `sum` called by the C main program is listed as follows:

```
   .global  _sum
_sum
   mov *AR0+,AC0    ; AC0 = x[1]
   add *AR0+,AC0    ; AC0 = x[1]+x[2]
   mov AC0,T0       ; Return value in T0
   ret              ; Return to calling function
```

The label `_sum` defines the entry point of the assembly subroutine, and the directive `.global` defines this assembly routine `sum(short *)` as a global function. Table 2.21 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Use CCS to create the project `c_asm.pjt`.
2. Add the C and assembly files listed in Table 2.21 to the project from CCS Project→Add Files to Project tab.
3. From the CCS Project→Options→Linker→Library tab, include the run-time support library `rst55.lib`.
4. Compile, debug, and load the program to the CCS (or DSK), then issue the **Go Main** command.
5. Watch and record the contents of AC0, AR0, and T0 in the CPU register window. Watch memory location '`s`' and '`x`' and record when the content of result '`s`' is updated. Why?
6. Single step through the C and assembly code.

2.10.2 Addressing Modes Using Assembly Programming

In this experiment, we will use assembly routines to exercise and understand C55x addressing modes. The assembly routines used for experiment are called by the following C function:

```
short a[8];               /* Define array Ai[] */
short x[8];               /* Define array Xi[] */
short result1, result2;   /* Define variables  */
main()
{
   absAddr(void);
   directAddr(void);
   result1 = indirectAddr(a, x);
   result2 = parallelProc(a, x);
}
```

The assembly routine `absAddr( )` uses the absolute addressing mode to initialize the array `a[8] = {1, 2, 3, 4, 5, 6, 7, 8}` in data memory. Since the array `a[8]` is defined in the C function, the assembly routine references it using the directive `.global` (or `.ref`).

The assembly routine `directAddr( )` uses the direct addressing mode to initialize the array `x[8]= {9, 3, 2, 0, 1, 9, 7, 1 }` in data memory. As mentioned in Section 2.4.1, there are four different direct addressing modes. We use the DP-direct addressing mode for this experiment. The instruction

```
 btstclr #14,*(ST1),TC1
```

tests the CPL (compiler mode) bit of the status register ST1 (bit 14). The compiler mode will be set if this routine is called by a C function. If the test is true, the test flag bit TC1 of status register ST0 will be set, and this instruction clears the CPL bit. This is necessary for using DP-direct addressing mode. At the end of the code, the conditional execution instruction

```
 xcc continue,TC1
```

sets the CPL bit if TC1 is set.

A dot product of two vectors **a** and **x** of length L can be expressed as

$$y = \sum_{i=0}^{L-1} a_i x_i = a_0 x_0 + a_1 x_1 + \cdots + a_{L-1} x_{L-1},$$

where a_i and x_i are elements of vectors **a** and **x**, respectively. In this experiment, we expect the dot product $y = 0x0089$. There are many ways to access the elements of arrays, such as direct, indirect, and absolute addressing modes. In this experiment, we implement the dot-product routine `indirectAddr(short *, short *)` using the indirect addressing mode and store the returned value in the variable `result` in data memory. Arrays **a** and **x** are defined as global arrays, and the return value is passed back to the C function by register T0.

```
_indirectAddr
    mpym  *AR0+,*AR1+,AC0
    mpym  *AR0+,*AR1+,AC1
    add   AC1,AC0
    mpym  *AR0+,*AR1+,AC1
    add   AC1,AC0
    mpym  *AR0+,*AR1+,AC1
    add   AC1,AC0
    mpym  *AR0+,*AR1+,AC1
    add   AC1,AC0
```

```
        mpym  *AR0+,*AR1+,AC1
        add   AC1,AC0
        mpym  *AR0+,*AR1+,AC1
        add   AC1,AC0
        mpym  *AR0+,*AR1+,AC1
        add   AC1,AC0
        mov   AC0,T0
        ret
```

The assembly routine `parallelProc (short *, short *)` uses the indirect addressing mode in conjunction with parallel and repeat instructions to improve the code density and efficiency. The auxiliary registers AR0 and AR1 are used as data pointers to array **a** and array **x**, respectively. The instruction `macm` performs MAC operation. The parallel bar `||` indicates the parallel execution of two instructions. The repeat instruction `rpt #K` will repeat the followed instruction `K+1` times:

```
    _parallelProc
        mpym *AR0+,*AR1+,AC0
    ||  rpt  #6
        macm *AR0+,*AR1+,AC0
        mov  AC0,T0
        ret
```

Table 2.22 briefly describes the files used for this experiment. Procedures of the experiment are listed as follows:

1. Create the project `addrModes.pjt`, add the C and assembly files listed in Table 2.22, and the run-time support library `rts55.lib` to the project. Build and load the project.

2. Use the memory watch window to see how the arrays **a** and **x** are initialized in data memory by routines `absAddr.asm` and `directAddr.asm`.

3. Open the CPU register window to see how the dot product is computed by `indirectAddr.asm` and `parallel.asm`.

4. Use the profiler to measure the sum-of-product operations and compare the cycle difference between the assembly routines `indirectAddr.asm` and `parallel.asm`.

5. Generate map files to compare the code size of assembly routines `indirectAddr.asm` and `paral-lel.asm`. Note that the program size is given in bytes.

Table 2.22 File listing for experiment `exp2.10.2_addrModes`

Files	Description
`addrModesTest.c`	C function for testing the experiment
`absAddr.asm`	Assembly routine uses absolute addressing mode
`dirctAddr.asm`	Assembly routine uses direct addressing mode
`indirectAddr.asm`	Assembly routine uses indirect addressing mode
`parallelProc.asm`	Assembly routine uses parallel instructions
`addrModes.pjt`	DSP project file
`addrModes.cmd`	DSP linker command file

2.10.3 Phase-Locked Loop and Timers

In this experiment, we will set up the PLL for C55x timers. We use the data type qualifiers `ioport` and `volatile` in C language to access MMRs and I/O registers. We use the `pragma` keyword for memory management, and also introduce the in-line assembly insertion.

The C55x peripheral register, CLKMD, controls the clock generator. When the PLL is enabled, the output clock frequency can be obtained by the following equation:

$$\text{Output frequency} = \frac{\text{PLLMULT}}{\text{PLLDIV} + 1}\text{ Input frequency.} \tag{2.1}$$

For example, if the input frequency is 24 MHz with PLLDIV = 2 and PLLMULT = 25, the output frequency of the PLL will be 200 MHz.

DSP processor's timers generate accurate timing pulses; thus, they are widely used in practical applications for managing and controlling the DSP systems. The TMS320C55x has two general-purpose timers; each has a prescale register (PRSC), a 16-bit period register (PRD), a 16-bit main counter (TIM), and a timer control register (TCR). The prescale counter (PSC) and time-divide-down counter (TDDR) of the PRSC are 4-bit. When the timer operates, TIM can be automatically reloaded from the PRD register, and the PSC can be automatically reloaded from the TDDR of the PRSC register when it reaches zero. Both TIM and PSC counters are decrement-by-1 counters. The PSC counter is driven by the processor input clock. For each clock cycle, the PSC is decremented by 1 until it reaches 0. After PSC reaches 0, TIM will be decremented by 1 at the next clock cycle. Once TIM reaches 0, the next clock pulse will make the timer to issue the interrupt TINT to the processor and the timer event TEVT to DMA, as well as generate the output signal TOUT. The timer period is calculated by

$$\text{Period} = (\text{TDDR} + 1)\,(\text{PRD} + 1)\text{ Input clock period.} \tag{2.2}$$

For instance, if the input clock frequency is 160 MHz with TDDR = 9 and PRD = 15999, the timer period will be 1 ms. When the timer expires, the interrupt TINT will automatically set the flag in the interrupt flag register TFR0 for Timer0 (or IFR1 for Timer1).

The C55x C compiler supports the `asm` statement that inserts a single line of assembly code into C programs. In program `initTimer()`, we used inline assembly statement

```
asm("\tBSET #ST1_INTM, ST1_55");
```

within C code to directly control the C55x global interrupt mask register. The syntax of the in-line assembly statement is

```
asm("assembly instruction");
```

Before we change any of the C55x system registers, it is a good practice to disable the processor interrupts. This can prevent unpredictable events caused by the corruption of system registers. The most efficient way to disable all the C55x interrupts is to set the global interrupt mask register, INTM. Since the timer interrupt will be enabled after initializing the timer, this experiment also clears all the pending interrupts before the completion of the DSK initialization by writing 0xFFFF to IFR0 register. The steps of initializing C55x timer can be summarized as follows:

1. Disable the global interrupt to prevent interrupts during timer initialization.

2. Clear any pending interrupts and stop timer interrupt.

3. Initialize the timer with new TDDR, PRD, and PRD-reload values.

4. Enable timer interrupt.

5. Reenable the global interrupt INTM.

The timer can be stopped any time by setting the TCR timer stop bit TSS, and enabled by clearing the TSS bit.

TMS320C55x can support up to 32 interrupts including hardware and software interrupts. These interrupts can be categorized as maskable and non-maskable interrupts. The maskable interrupts can be blocked by the software. If multiple interrupts occur simultaneously, the C55x will serve these interrupts based on their priorities listed in Table 2.5. When an interrupt occurs, an interrupt request will be issued to the C55x processor. The C55x will complete the current instruction, flush the instruction pipeline, issue an interrupt request acknowledgment, and prepare itself to serve the interrupt. The C55x will fetch the corresponding ISR address and branch to that ISR for performing interrupt service. The C55x interrupt vector table given in Table 2.23 defines the ISR address offsets. In this experiment, we use timer interrupt as an example.

Table 2.23 Example of using interrupt vector table, `vectorsTable.asm`

```
;
;  Interrupt table for C5510
;

     .def _Reset
     .ref _c_int00
     .ref _c_tint0

     ; Vector macro
     ;
     vector   .macro   isrName
         b    :isrName:
         nop_16
     ||  nop_16
         .endm


     ; Default handler
     ;

       .sect ".text:example:timer0"
     _no_ISR b _no_ISR


     ; Interrupt vectors
     ;

      .sect ".vectors"

 _Reset: vector _c_int00  ; 0x00 Reset (HW or SW)
 NMI:    vector _no_ISR   ; 0x08 Non-maskable hardware interrupt
 INT0:   vector _no_ISR   ; 0x10 External interrupt INT0
 INT2:   vector _no_ISR   ; 0x18 External interrupt INT2
```

Table 2.23 (*continued*)

```
TINT0:  vector _c_tint0 ; 0x20 Timer 0 interrupt TINT0
RINT0:  vector _no_ISR  ; 0x28 McBSP 0 receive interrupt RINT0
RINT1:  vector _no_ISR  ; 0x30 McBSP 1 receive interrupt RINT1
XINT1:  vector _no_ISR  ; 0x38 McBSP 1 transmit interrupt XINT1
SINT8:  vector _no_ISR  ; 0x40 Software interrupt 8 SINT8
DMAC1:  vector _no_ISR  ; 0x48 DMA channel 1 interrupt DMAC1
DSPINT: vector _no_ISR  ; 0x50 Interrupt from Host EHPI DSPINT
INT3:   vector _no_ISR  ; 0x58 External interrupt 3 INT3
RINT2:  vector _no_ISR  ; 0x60 McBSP 2 receive interrupt RINT2
XINT2:  vector _no_ISR  ; 0x68 McBSP 2 transmit interrupt XINT2
DMAC4:  vector _no_ISR  ; 0x70 DMA channel 4 interrupt DMAC4
DMAC5:  vector _no_ISR  ; 0x78 DMA channel 5 interrupt DMAC5
INT1:   vector _no_ISR  ; 0x80 External interrupt 1 INT1
XINT0:  vector _no_ISR  ; 0x88 McBSP 0 transmit interrupt XINT0
DMAC0:  vector _no_ISR  ; 0x90 DMA channel 0 interrupt DMAC0
INT4:   vector _no_ISR  ; 0x98 External interrupt 4 INT4
DMAC2:  vector _no_ISR  ; 0xa0 DMA channel 2 interrupt DMAC2
DMAC3:  vector _no_ISR  ; 0xa8 DMA channel 3 interrupt DMAC3
TINT1:  vector _no_ISR  ; 0xb0 Timer 1 interrupt TINT1
INT5:   vector _no_ISR  ; 0xb8 External interrupt 5 INT5
BERR:   vector _no_ISR  ; 0xc0 Bus error interrupt BERR
DLOG:   vector _no_ISR  ; 0xc8 Data log interrupt DLOG
RTOS:   vector _no_ISR  ; 0xd0 Real-time OS interrupt RTOS
SINT27: vector _no_ISR  ; 0xd8 Software interrupt 27 SINT27
SINT28: vector _no_ISR  ; 0xe0 Software interrupt 28 SINT28
SINT29: vector _no_ISR  ; 0xe8 Software interrupt 29 SINT29
SINT30: vector _no_ISR  ; 0xf0 Software interrupt 30 SINT30
SINT31: vector _no_ISR  ; 0xf8 Software interrupt 31 SINT31
.end
```

In Table 2.23, each interrupt address occupies 8 bytes. The first 4 bytes contain the address of the ISR. The next 4 bytes are instructions that will be executed before the processor entering the ISR because of the C55x pipeline. Therefore, we can move some of the ISR instructions into this 4-byte space.

In Table 2.23, we use assembly macro to replace the repeated statements used by the program. With assembly macro, we can define our own instructions and shorten the source statements for easy reading. The assembly macro must start with a macro name in the source code label field followed by the identifier keyword `.macro` and end with the keyword `.endm`. The macro statements can have optional parameters. The body of the macro may have assembly instructions and assembly directives. In Table 2.23, the macro

```
vector .macro   isrName
    b  :isrName:
    nop_16
||  nop_16
        .endm
```

uses the macro name `vector`, the optional parameter is `isrName`. There are three assembly instructions inside the macro `vector`. The first instruction is a branch instruction. The next two `nop` instructions are used for the following 4-byte program memory. The total size of this macro is 8 bytes. The branch instruction uses substitution symbol `isrName`. The macro substitution symbol allows user to use macro multiple times with different parameters or data. The interrupt table listed in Table 2.23 has two ISR

Table 2.24 Program example of timer ISR, `tint0.c`

```
#include "timer.h"

#pragma DATA_SECTION(time,        ".bss:example:timer0");
#pragma CODE_SECTION(c_tint0,     ".text:example:timer0");

C55xTimer time;              // Define timer variable

interrupt void c_tint0(void)
{
    time.us += 10;           // Each interrupt is 10 us
    if (time.us == 1000)
    {
        time.us = 0;
        if (time.ms++ == 1000)
        {
            time.ms = 0;
            time.s++;
        }
    }
}
```

addresses: `_c_int00` and `_c_tint0`. The `_c_int00` is the C55x power-on reset ISR address, and `_ctint_0` is the timer0 ISR address. For other vectors in Table 2.23, the vector addresses are replaced with `_no_ISR`, a dummy ISR placeholder.

Table 2.24 shows the example of the timer ISR. Every time when the timer interrupt occurs, this ISR will increment the counter by 10 μs as programmed timer unit. When writing C program for C55x, the ISR must use the keyword `interrupt` qualifier to inform the C compiler for generating interrupt content saving and restore procedures while entering and exiting an interrupt.

The vector table must be placed at the special memory location specified by the linker command file. The code section `.sect ".vectors"` in Table 2.23 matches the vector section defined by the linker command file `timer.cmd` as shown in Table 2.25. When defining memory addresses for the C55x, the linker command file requires the use of byte unit for address and length of the sections. For example, the `VECS` section starts at byte address 0x100, which is the same as 0x80 for the word address.

The program for timer experiment is listed in Table 2.26. This program uses a 10-μs timer to measure the run-time function `application( )`. Because the application function loading is varying, the time measured is also changing. Using the C compiler optimization, we can build an efficient program that

Table 2.25 Example of timer linker command file, `timer.cmd`

```
-stack    0x2000   /* C55x DSP stack size  */
-sysstack 0x1000   /* System stack size    */
-heap     0x2000   /* DSP heap size        */

-c                 /* Use C linking conventions */

MEMORY
{
  PAGE 0:  /* ---- C55x unified memory space -------- */
  VECS (RIX)    : origin=0x000100, length=0x000100 /* 256-byte vectors */
  DARAM (RWIX) : origin=0x000200, length=0x00fe00 /* Internal DARAM   */
```

Table 2.25 *(continued)*

```
    SARAM (RWIX) : origin=0x010000, length=0x040000 /* Internal SARAM   */
    PAGE 2: /* ---- C55x 64K-word I/O memory space --------*/
    IOPORT (RWI) : origin = 0x000000, length = 0x020000
  }

  SECTIONS
  {
      vectors   > VECS  PAGE 0      /* Interrupt vectors            */
     .text      > SARAM PAGE  0     /* Code                         */
     .data      > SARAM PAGE  0     /* Initialized variables        */
     .bss       > DARAM PAGE  0     /* Global & static variables    */
     .const     > DARAM PAGE  0     /* Constant data                */
     .sysmem    > SARAM PAGE  0     /* Dynamic memory (malloc)      */
     .stack     > SARAM PAGE  0     /* Primary system stack         */
     .sysstack  > SARAM PAGE  0     /* Secondary system stack       */
     .cio       > SARAM PAGE  0     /* C I/O buffers                */
     .switch    > SARAM PAGE  0     /* Switch statement tables      */
     .cinit     > SARAM PAGE  0     /* Auto-initialization tables   */
     .pinit     > SARAM PAGE  0     /* Initialization fn tables     */

     .ioport    > IOPORT PAGE 2     /* Global&static IO variables   */
  }
```

runs much faster. Table 2.27 lists the run-time benchmark comparison of this demo program built with and without C compiler optimization option.

Table 2.28 lists the files used for this experiment. Although we have included all the files for the experiment, the readers are strongly encouraged to create this experiment step by step.

Table 2.26 Program example of timer, `timerTest.c`

```
#include <stdio.h>
#include "timer.h"

#pragma CODE_SECTION(main,         ".text:example:timer0");
#pragma CODE_SECTION(application,  ".text:example:timer0");

static unsigned long application(unsigned long cnt, short lp);

void main()
{
    short k,loop;
    unsigned long  sampleCnt;

    asm(" MOV #0x01,mmap(IVPD)");// Set up C55x interrupt vector pointer
    asm(" MOV #0x01,mmap(IVPH)");// Set up HOST interrupt vector pointer
    initCLKMD();                 // Initialize CLKMD register
    initTimer();                 // Initialize timer
    loop = 100;
    for (k=0; k<9; k++)
    {
        time.us = 0;            // Reset time variables
```

continues overleaf

Table 2.26 *(continued)*

```
       time.ms = 0;
       time.s = 0;
       startTimer();           // Start timer
       sampleCnt = application(0, loop);
       stopTimer();            // Stop timer
       loop -= 10;
       printf("samples processed = %10ld\ttime used = %d(s)%d(ms)%d(us)\ n",
               sampleCnt,time.s,time.ms,time.us);
     }
 }
 // Example of DSP application
 static unsigned long application(unsigned long cnt, short lp)
 {
     short i,j,k,n;

     for (i=0; i<lp; i++)
     {
       for (j=0; j<lp; j++)
       {
          for (k=0; k<lp; k++)
          {
             for (n=0; n<lp; n++)
             {
               cnt++;
             }
          }
       }
     }
     return (cnt);
}
```

Procedures of the experiment are listed as follows:

1. Create the project `timer.pjt`, and add all the files listed in Table 2.28 to the project.
2. Include the run-time support library, build the project with compiler optimization option set to no optimization, and load the experiment to DSK. Run the timer experiment and record the time spent for each time the function `application` is called.

Table 2.27 Benchmark of program example, `timerTest.c`

Number of samples processed	Without optimization	With optimization –o2
100 000 000	7(s)407(ms)970(μs)	0(s)10(ms)920(μs)
65 610 000	4(s)868(ms)940(μs)	0(s) 8(ms) 0(μs)
40 960 000	3(s) 44(ms)990(μs)	0(s) 5(ms)650(μs)
24 010 000	1(s)790(ms)170(μs)	0(s) 3(ms)810(μs)
12 960 000	0(s)970(ms)240(μs)	0(s) 2(ms)430(μs)
6 250 000	0(s)470(ms)170(μs)	0(s) 1(ms)420(μs)
2 560 000	0(s)193(ms)970(μs)	0(s) 0(ms)740(μs)
810 000	0(s) 62(ms)120(μs)	0(s) 0(ms)320(μs)
160 000	0(s) 12(ms)570(μs)	0(s) 0(ms)100(μs)

Table 2.28 File listing for experiment `exp2.10.3_DSPTimer`

Files	Description
`timerTest.c.c`	C function for testing DSP timer experiment
`vectorsTable.asm`	Assembly routine sets interrupt vector table
`clkmdInit.c`	C function initializes C55x clock CLKMD register
`timerInit.c`	C function initializes C55x timer0
`timerStart.c`	C function starts C55x timer0
`timerStop.c`	C function stops C55x timer0
`tinit0.c`	C timer0 interrupt service function
`timer.h`	C header file for DSP timer experiment
`timer.pjt`	DSP project file
`timer.cmd`	DSP linker command file

3. Set the compiler optimization option to `-o2` for the file `timerDemo.c`. Rerun the experiment and compare the time spent for the function `application`. By turning the optimization option on, all the C files in the project will be compiled with C compiler optimizer. How to set the **build option** such that the CCS will only optimize one particular file?

2.10.4 EMIF Configuration for Using SDRAM

In this experiment, we will set up the EMIF registers to use the SDRAM for the C5510 DSK with the memory configuration as shown in Figure 2.18.

The DARAM and SARAM are the on-chip memories, while the SDRAM is an external memory. The SDRAM starts at word address 0x028000 controlled by chip enable CE0. The chip enable CE1 controls the Flash memory and the complex programmable logic device.

In this experiment, we configure CE0 for SDRAM and CE1 for flash memory. After resetting the EMIF, the program clears the EMIF error flags. EMIF global control register is set to use half of the CPU clock frequency. MTYPE bit of CE0 is set for SDRAM. The experiment initializes the EMIF and writes data to SDRAM. The data in SDRAM can be viewed using CCS graph display feature. Figure 2.19 shows how to set up the display type as RGB color image and selects data addresses in SDRAM for display, and Figure 2.20 displays the color pattern. Because the SDRAM is an external memory, it requires 23-bit address range to access SDRAM. We compile and link the project using large memory

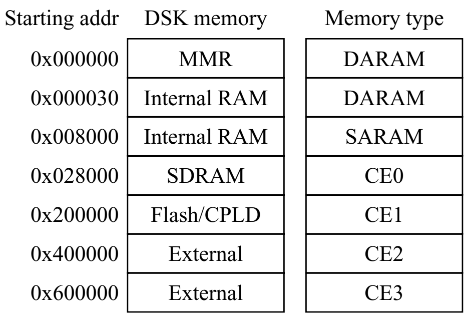
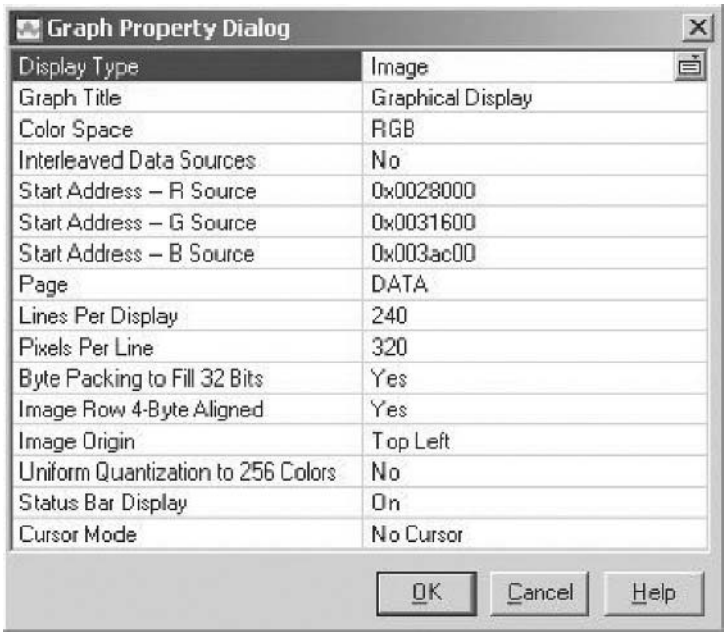


Figure 2.18 TMS320VC5510 DSK memory configuration



Display Type	Image
Graph Title	Graphical Display
Color Space	RGB
Interleaved Data Sources	No
Start Address – R Source	0x0028000
Start Address – G Source	0x0031600
Start Address – B Source	0x003ac00
Page	DATA
Lines Per Display	240
Pixels Per Line	320
Byte Packing to Fill 32 Bits	Yes
Image Row 4-Byte Aligned	Yes
Image Origin	Top Left
Uniform Quantization to 256 Colors	No
Status Bar Display	On
Cursor Mode	No Cursor

Figure 2.19 Graph display setting for EMIF demonstration example

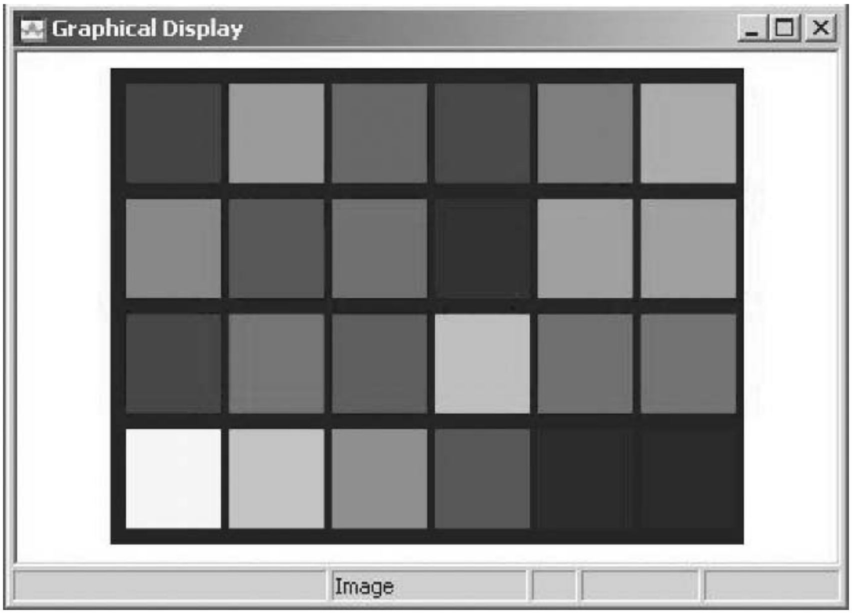


Figure 2.20 The RGB color checker display by the EMIF test program

Table 2.29 File listing for experiment `exp2.10.4_emifSDRAM`

Files	Description
`emifTest.c`	C function for testing EMIF experiment
`emifInit.c`	C function initializes C55x EMIF module
`drawBox.c`	C function writes data to SDRAM
`emif.h`	C header file for EMIF experiment
`emif.pjt`	DSP project file
`emif.cmd`	DSP linker command file

model. When using large memory model, we need to link the program with the run-time support library `rts55x.lib`. For this experiment, we modify the linker command file to include the SDRAM memory space and external SDRAM memory section as following:

```
MEMORY
{
    SDRAM (RWIX): origin=0x050000, length=0x3B0000 /* External SDRAM */
}
SECTIONS
{
   .edata > SDRAM PAGE 0 /* SDRAM memory: external memory */
}
```

The files used for this experiment are listed in Table 2.29. Procedures of the experiment are listed as follows:

1. Create the project `emif.pjt`, and add files listed in Table 2.29 and the run-time support library `rts55x.lib` to the project. Set the **Build Option** to use large memory mode.

2. Build the project and load it to DSK. Run the experiment and view the SDRAM memory using CCS graphic display tool to verify RGB data write to SDRAM. Do you see the data pattern as shown in Figure 2.20?

2.10.5 Programming Flash Memory Devices

Many DSP products use flash memories to store application program and data. The programmable flash memory offers the cost-effective and reliable read/write nonvolatile random accesses. The C5510 DSK consists of 256K words (4 Mbits) external flash memory. The flash memory is mapped to the C55x EMIF CE1 memory space. In this experiment, we will demonstrate several basic functions of flash programming.

The C5510 DSK uses 4 Mbits of flash memory. The starting address of the flash memory is at word address 0x200000. The flash memory device is operated by writing specific data sequences into the flash memory command registers. Writing to incorrect address or using invalid data will cause the flash memory to reset. After resets, the flash device is in read-only state. The flash memory reset is performed by writing reset command word 0xF0 to any valid flash memory location.

The flash memory must be erased before writing new data. The erase operation includes both chip erase and sector erase. The chip erase will erase all the data contained in the entire flash memory device while the sector erase will erase the data contained only in the specific sector. The erase command sequence is writing the specific data pattern 0xAA-0x55-0x80-0xAA-0x55-0x10 to flash memory address offset 0x555 and 0x2AA. After the command sequence has been completed, we check the flash memory status

Table 2.30 File listing for experiment `exp2.10.5_flashProgram`

Files	Description
`flashTest.c`	C function for testing flash programming experiment
`emifInit.c`	C function initializes C55x EMIF module
`flashErase.c`	C function erases flash memory data
`flashID.c`	C function obtains flash device IDs
`flashReset.c`	C function resets flash memory
`flashWrite.c`	C function writes data to flash memory
`emif.h`	C header file used for EMIF settings
`flash.h`	C header file for flash programming experiment
`flash.pjt`	DSP project file
`flash.cmd`	DSP linker command file
`dtmf18005551234.dat`	Data file in ASCII format

bit DQ7 to detect if the erase has completed. The bit DQ5 indicates if the operation has been timed out. If this bit is set, it indicates a timeout error.

When programming flash memory, the write sequence must be issued for each data at each flash memory address. The data can be a 16-bit word or an 8-bit byte. This experiment uses 16-bit flash memory write program. The write command sequence is 0xAA-0x55-0xA0-addr-data. After each data has been written, the processor checks the data ready bit, DQ7, before writing another data.

The program first initializes the EMIF to map the flash device to CE1. It then resets the flash device and uses the function `flashID( )` to obtain the manufacture ID and device ID. This important step allows system to determine exactly which flash memory device has being used, and thus enables flexible programming for using different programming algorithms for different flash memory devices. The program then erases the entire flash memory and reprograms it. Finally, the program reads data back from the flash memory and compares with the original data.

In this experiment, we introduce several basic programming functions and demonstrate flash-erase and flash-write procedures. We also identify specific flash memory devices. Table 2.30 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Create the project `flash.pjt`, and add the files listed in Table 2.30 to the project.

2. Build the project and load it to DSK. Run the experiment and view the flash memory data pattern with CCS memory window after flash programming is completed.

3. Reload the flash program to DSK. Use **Go Main** command to start the experiment. Single step through the program to view the flash memory manufacture ID and chip ID. Use CCS debug window to view flash memory (what is the correct starting address for flash memory?) before calling the `flashErase()` function. Single step into the erase function and view the flash memory using CCS. Finally, run the code to write the data into the flash memory.

2.10.6 Using McBSP

The C5510 DSK uses McBSP1 and McBSP2 for connecting the analog interface chip TLV320AIC23, where McBSP1 is used as control channel and McBSP2 is used as data channel. In this experiment, we will introduce the basic programming of the McBSP and use CCS to build the McBSP DSP library.

Every McBSP consists of two serial port control registers (SPCR) in I/O memory spaces. These control registers are used for controlling digital loopback, sign extension, clock stop, and interrupt modes. There are also several signal pins that allow user to check the current status of transmit and receive operations. Each McBSP port has two transmit control registers (XCR) and two receive control registers (RCR). These I/O MMRs allow users to specify transmit and receive frame phases, wordlength, and the number of the words to transfer.

There are two sample rate generator registers (SRGR) for each McBSP port. Sample rate generator can generate frame sync and clock (CLKG) signals. The SRGRs allow user to choose input clock source (CLKSM), divide output clock via a divide counter (CLKGDV), and set the frame sync pulse width and period. Every McBSP port has eight transmit channel enable registers and eight receive channel enable registers. These registers are used only when the transmittmer and/or receiver are configured to allow individual enabling and disabling of the channels; that is, TMCM = 0 and/or RMCM = 1.

In this experiment, we initialize the registers of the McBSP ports. Since the DSK uses McBSP1 in master mode for AIC23 control channel, McBSP1 must be initialized before it can send configuration commands and parameters to the control registers of the AIC23. The McBSP2 is used as data port for AIC23, which can be configured by the function `mcbsp2Init( )`. After the McBSP2 is reset, the initialization sets the transmitter and receiver control registers. The pin control register is configured for the proper clock polarities. Finally, the sample rate generator is enabled, and then McBSP transmitter and receiver are enabled.

This experiment uses two functions: `mcbsp1CtlTx( )` and `mcbsp2DatTx( )`. The function `mcbsp1CtlTx( )` checks McBSP transmit data ready bit. When this bit is set, the control parameter `regValue` will be written to the AIC23 control registers via the McBSP1. The lower 9 bits of the `regValue` contain the control parameters for setting the AIC register, while the upper 6 bits are used to identify the AIC23 control registers. Similarly, the function `mcbsp2DatTx( )` checks if it is time to write the data via the McBSP2 transmitter. If the `XRDY` bit is set, the data is copied to the McBSP transmit buffer for transmission.

The process of using CCS to create a DSP library for the TMS32C55x processor is similar to create a COFF executable program. First, we create a new project using library (.lib) as the target instead of choosing the COFF executable file type. The library must use the same memory model as the application program that will use the library. As an example, Figure 2.21 shows that the project type is the library

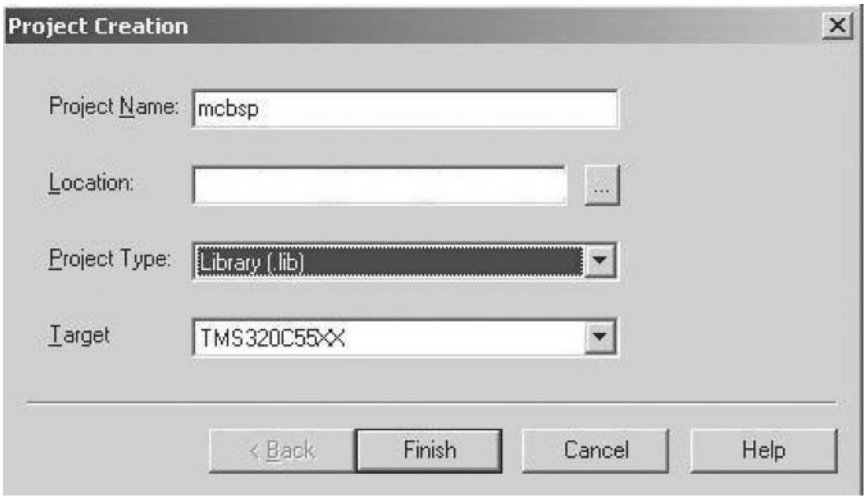
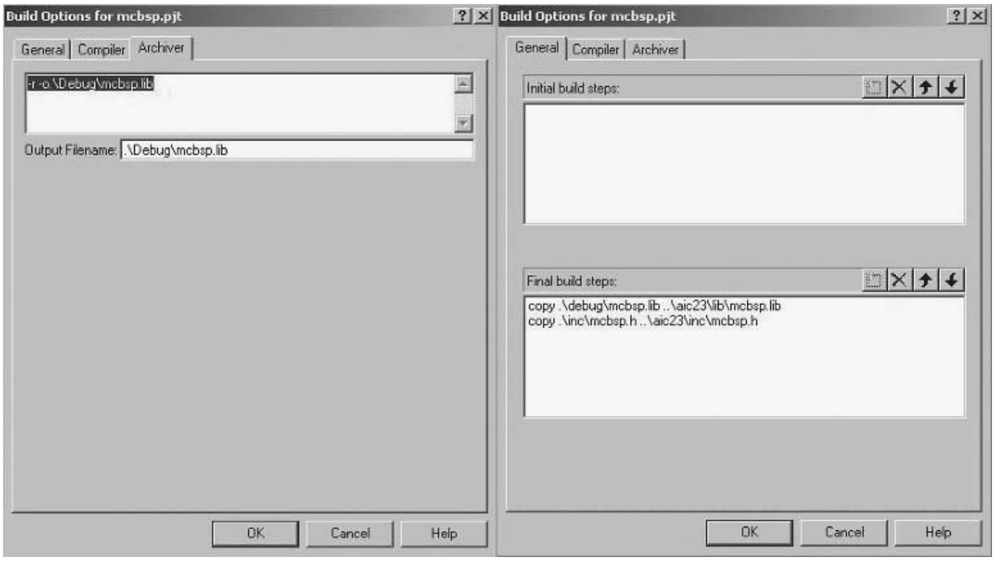

Project Creation
Project Name: mcbsp
Location:
Project Type: Library (.lib)
Target TMS320C55XX

Figure 2.21 The project creation for McBSP library



```
-r -o.\Debug\mcbsp.lib
```

Output Filename: .\Debug\mcbsp.lib

```
copy .\debug\mcbsp.lib ..\aic23\lib\mcbsp.lib
copy .\inc\mcbsp.h ..\aic23\inc\mcbsp.h
```

Figure 2.22 The CCS project configuration for McBSP library, `mcbsp.lib`

when we create the new project. The building option in Figure 2.22 shows the project will create a library `mcbsp.lib`. When creating libraries, only the library functions are included in the project. The application program, test programs, and other functions that are not related to the library shall not be included. We choose optimization option level-2 when building the library, so the library functions will be compiled with optimization option turned on. We also disabled features of generating the debug information when building the library. These settings give us an efficient library. As shown in Figure 2.22, we used two copy statements in the **Final build steps Window** to copy the C header file and library to the working directories.

In practice, the library functions are individually tested, debugged, and verified before being used to create the library. The example `mcbsp.lib` is made with the large memory model and will be used by the next experiment. In Figure 2.22, we also show how to add special commands in the CCS build option to copy the `mcbsp.lib` from the current build directory to the destination. Table 2.31 lists the files used to build the McBSP library for this experiment.

Table 2.31 File listing for experiment `exp2.10.6_mcbsp`

Files	Description
`mcbsp1CtlTx.c`	C function sends command and data via C55x McBSP1
`mcbsp1Init.c`	C function initializes C55x McBSP1 registers
`mcbsp2DatTx.c`	C function writes data to McBSP2
`mcbsp2Init.c`	C function initializes C55x McBSP2 registers
`mcbspReset.c`	C function resets C55x McBSP
`mcbsp.h`	C header file for McBSP experiment
`mcbsp.pjt`	DSP project file

Procedures of the experiment are listed as follows:

1. Create the project `mcbsp.pjt`, and add the following files to the project: `mcbsp1CtlTx.c`, `mcbsp1Init.c`, `mcbsp2DatTx.c`, `mcbsp2Init.c`, and `mcbspReset.c`.

2. Set up the search path for **Include File**. Build the project to create the library.

3. The C55x DSP code generation tools are located in the directory `..\c5500\cgtools\bin`. Open a command window from the host computer by going to the **Windows Start Menu** and select **Run**. In the **Run** dialogue window, type **cmd** and click **OK**.

4. When the command window appears on the computer, we will show how to use the archiver tool `ar55.exe` located in `..\c5500\cgtools\bin directory`. Assuming that the DSP tools are installed in the `C:\ti` directory, type `C:\ti\c5500\cgtools\bin\ar55 -h` from the command window to see the archiver's help information. The following is an example of the archiver help menu:

```
Syntax: ar55 [arxdt][quvse] archive files ...

 Commands: (only one may be selected)
    a - Add file                r - Replace file
    x - Extract file            d - Delete file
    t - Print table of contents
 Options:
    q - Quiet mode - Normal status messages suppressed
    u - Update with newer files (use with 'r' command)
    s - Print symbol table contents
    v - Verbose
```

5. The archiver `ar55.exe` allows us to view (-t) the file list of a library, remove (-d) files from the library, add (-a) and replace (-r) files to existing library, and extract the library files. Use these commands to view and extract the McBSP library that we built for this experiment.

2.10.7 AIC23 Configurations

The C5510 DSK analog inputs include a microphone and a stereo line-in; the analog outputs include a stereo line-out and a stereo headphone. The AIC23 uses McBSP1 for control channel with 16-bit control signal. The lower 9 bits contain the command value that will be written to the specified register, while the upper 7 bits specify the AIC23 control register. The McBSP2 is set as the bidirectional data channel for passing audio samples in and out of the DSK. The AIC23 supports several data formats and can be configured for different sampling frequencies as described in its data manual [12]. In this experiment, we will configure the C55x McBSP to interface with the AIC23 for real-time audio playback.

The experiment program `AIC23Demo( )` configures the AIC23 for stereo output. The digital samples stored in the DSK flash memory will be played at 8 kHz rate. The AIC23 has 11 control registers that must be initialized to satisfy different application requirements. The initialization values are listed in the C header file `aic23.h`.

The AIC23 uses the sigma–delta technologies with built-in headphone amplifier to provide up to 30 mW output level for 32Ω impedance and 40 mW for 16 Ω impedance. It supports sampling rate from 8 to 96 kHz, and data wordlength of 16, 20, 24, and 32 bits. The AIC23 also includes flexible gain and volume controls. Figure 2.23 shows the functional block diagram of AIC23. The serial clock is connected

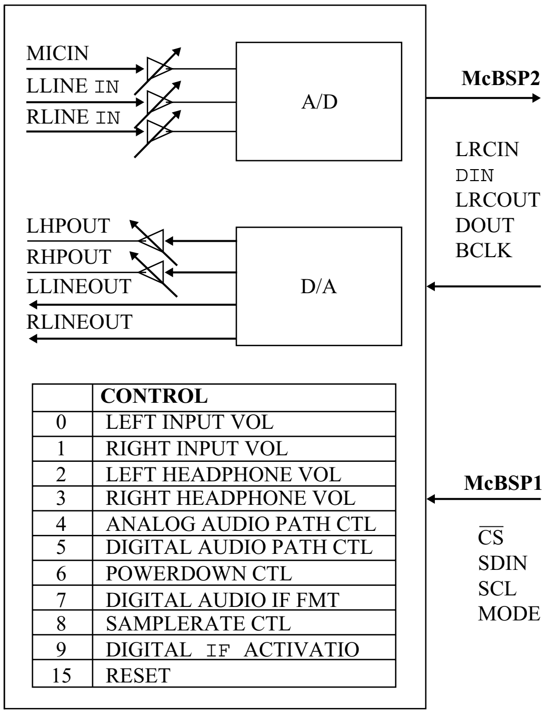


Figure 2.23 Block diagram of connecting AIC23 using McBSPs

to SCLK. The data word is latched by $\overline{\text{CS}}$ signal. The 16-bit control word is latched on the rising edge of $\overline{\text{CS}}$ with the MSB first.

The AIC23 supports stereo audio channels. The left and right channels can be simultaneously locked together or individually controlled. The program listed in Table 2.32 initializes the DSK and AIC23, reads in the data sample from flash memory, and plays this audio signal at 8-kHz sampling rate via the DSK headphone output.

The files used for this AIC23 experiment are listed in Table 2.33. Procedures of the experiment are listed as follows:

1. Use the flash program experiment in Section 2.10.5 to initialize the flash memory with the data file, `dtmf18005551234.dat`. You can also use the same data file from SRAM as shown in `aic23Test.c`. In this case, the data file is included as a header file.

2. Create the `mcbsp.lib` using the experiment from previous experiment given in Section 2.10.6.

3. Create the project `aic23.pjt`, add all the source files listed in Table 2.33 and the `mcbsp.lib` to the project, and build the project (pay attention to memory mode).

4. Connect a headphone (or loudspeaker) to the DSK headphone output and run the DSK experiment to listen to the playback.

Table 2.32 Program of audio playback, `aic23Test.c`

```
#include <stdio.h>
#include "mcbsp.h"
#include "aic23.h"

#define DATASIZE 12320
#if 1
  // Flash memory data
#define flashData 0x200000  // Data has been programmed by flash example
#else
  // Local memory data
short flashData[DATASIZE]={ // Data is in SRAM
#include "dtmf18005551234.dat"
};
#endif

#pragma CODE_SECTION(main, ".text:example:main");

void main()
{
    short i,data;
    unsigned short *flashPtr;

    // Initialize McBSP1 as AIC23 control channel
    mcbsp1Init();

    // Initialize the AIC23
    aic23Init();

    // Initialize McBSP2 as AIC23 data channel
    mcbsp2Init();
    flashPtr = (unsigned short *)flashData;
    // Playback data via AIC23
    for (i=0; i<DATASIZE; i++)
    {
        data = *flashPtr++;
        while (!aic23Tx(data));      // Send data to left channel
        while (!aic23Tx(data));     //  Send data to right channel
    }

    // Power down the AIC23 and reset McBSP
    aic23Powerdown();
    mcbspReset(1);
    mcbspReset(2);
}
```

2.10.8 Direct Memory Access

In this experiment, we will use the C55x DMA controller, which allows data transfers among internal memory, external memory, peripherals, as well as the enhanced host-port interface without intervention by the processor. The DMA controller consists of four standard ports for DARAM, SARAM, external memory, and peripherals. The channel reads data from the given source and writes data to the destination.

Table 2.33 File listing for experiment `exp2.10.7_aic23`

Files	Description
`aic23Test.c`	C function for testing AIC23 experiment
`aic23Init.c`	C function initializes AIC23
`aic23Powerdown.c`	C function powers down AIC23
`aic23Tx.c`	C function sends data to AIC23
`mcbsp.h`	C header file for use McBSP library
`aic23.h`	C header file for AIC23 experiment
`aic23.pjt`	DSP project file
`aic23.cmd`	DSP linker command file
`mcbsp.lib`	McBSP library for AIC23 experiment
`dtmf18005551234.dat`	Data file in ASCII format

The DMA controller uses different channels to perform independent block transfers among the standard ports. There are six channels and each channel can send an interrupt to the processor on the completion of given operations. The DMA data transfers at each channel can be dependent on or be synchronized with occurrences of selected events.

For DMA data transfer, the basic data (in byte unit) is called element, and a group of elements is called frame. An element can have one or more bytes of data, and the frame can have one or more elements. The block is formed by one or more frames. The frame or block transfer can be interrupted, while the element transfer cannot be interrupted.

There are six DMA channels and each channel has 12 DMA registers for DMA control. These registers start from I/O memory address 0x0C00. Each channel is separated by 32 words. The DMA can have many combinations to satisfy different application requirements. In this experiment, we use a simple example to initialize and set up DMA for data transfer.

The program listed in Table 2.34 configures the DMA controller to transfer *N* elements using *M* frames through `DMA_CHANNEL`. The source is a pre-initialized DARAM and the destination is an SDRAM. The demonstration program initializes the DMA channel 3. Because the DMA controller uses only byte addresses, the source and destination address registers must be initialized in byte addresses. If the data type to be transferred is 32-bit, the source and destination must be aligned to the 32-bit boundary in memory using the `DATA_ALIGN` pragma.

Table 2.34 Program of DMA demo, `dmaTest.c`

```
#include <stdio.h>
#include "dma.h"
#include "emif.h"

#define N                  128  // Transfer data elements
#define M                  16   // Transfer data frames
#define DMA_CHANNEL         3   // DMA channel

// SRC is in DARAM and DST is in SDRAM
// Force SRC and DST to align at 32-bit boundary
#pragma DATA_SECTION(src, ".daram:example:dmaDemo")
#pragma DATA_SECTION(dst, ".sdram:example:dmaDemo")
#pragma DATA_ALIGN(src, 2);
#pragma DATA_ALIGN(dst, 2);
unsigned short src[N*M];
unsigned short dst[N*M];
```

Table 2.34 *(continued)*

```
#pragma CODE_SECTION(main, ".text:example:main");

void main(void)
{
    unsigned short i,frame,err;
    short dmaInitParm[DMA_REGS];
    unsigned long srcAddr,dstAddr;

    // Initialize EMIF
    emifInit();

    // Initialize source and destination memory for testing
    for (i = 0; i < (N*M); i++) {
        dst[i] = 0;
        src[i] = i + 1;
    }

    // Convert word address to byte address, DMA uses byte address
    srcAddr = (unsigned long)src;
    dstAddr = (unsigned long)dst;
    srcAddr <<= 1;
    dstAddr <<= 1;

    // Setup DMA initialization values
    dmaInitParm[0]  = DMACSDP_INIT_VAL;
    dmaInitParm[1]  = DMACCR_INIT_VAL;
    dmaInitParm[2]  = DMACICR_INIT_VAL;
    dmaInitParm[3]  = DMACSR_INIT_VAL;
    dmaInitParm[4]  = (short)(srcAddr & 0xFFFF);
    dmaInitParm[5]  = (short)(srcAddr >> 16);
    dmaInitParm[6]  = (short)(dstAddr & 0xFFFF);
    dmaInitParm[7]  = (short)(dstAddr >> 16);
    dmaInitParm[8]  = N;
    dmaInitParm[9]  = M;
    dmaInitParm[10] = DMACSFI_INIT_VAL;
    dmaInitParm[11] = DMACSEI_INIT_VAL;
    dmaInitParm[12] = 0;
    dmaInitParm[13] = 0;
    dmaInitParm[14] = DMACDEI_INIT_VAL;
    dmaInitParm[15] = DMACDFI_INIT_VAL;

    // Initialize DMA channel
    dmaInit(DMA_CHANNEL, dmaInitParm);

    // Enable DMA channel and begin data transfer
    dmaEnable(DMA_CHANNEL);

    // DMA transfer data at background
    frame = M;
    while (frame>0)
    {
        if (dmaFrameStat(DMA_CHANNEL) != 0)
```

continues overleaf

Table 2.34 (*continued*)

```
        {
            frame--;
        }
    }
    // Close DMA channel
    dmaReset(DMA_CHANNEL);

    // Check data transfer is correct or not
    err = 0;
    for (i = 0; i <(N*M); i++)
    {
        if (dst[i] != src[i])
        {
            err++;
        }
    }
    printf("DMA Demo: error found = %d\n", err);
}
```

The DMA initialization function `dmaInit( )` uses the argument `dmaNum` to select a DMA channel and initialize all 16 registers with the context passed in via `dmaInitParm[]`. The DMA channel is disabled by the initialization function. To begin a DMA transfer, the DMA channel must be enabled. The DMA status register bits indicate the DMA status for the given channel. The DMA demo program sets a frame-synchronization-based DMA transfer. After each frame of data has been transferred, the frame sync status bit of the DMA_CSR register will be set. The program checks the frame sync status bit to monitor the data transfer.

It is a good practice to disable the DMA channel when data transfer has been completed and the DMA channel will no longer be needed. This can prevent unpredictable behavior. The function `dmaReset( )` will disable the given DMA channel and reset the DMA registers.

Table 2.35 lists the files used for this DMA experiment. Procedures of the experiment are listed as follows:

1. Create the project `dma.pjt`, and add all the files listed in Table 2.35 and the run-time support library `rts55x.lib` to the project.

2. Build the project and run the program. What memory mode should be used for this experiment?

Table 2.35 File listing for experiment `exp2.10.8_dma`

Files	Description
`dmaTest.c`	C function for testing DMA experiment
`dmaEnable.c`	C function enables DMA
`dmaFrameStat.c`	C function checks DMA status bit
`dmaInit.c`	C function initializes DMA
`dmaReset.c`	C function resets DMA
`emifInit.c`	C function initializes EMIF for DMA experiment
`emif.h`	C header file for using EMIF initialization function
`dma.h`	C header file for DMA experiment
`dma.pjt`	DSP project file
`dma.cmd`	DSP linker command file

References

[1] Texas Instruments, Inc., *TMS320C55x DSP CPU Reference Guide*, Literature no. SPRU371F, 2004.
[2] Texas Instruments, Inc., *TMS320C55x Assembly Language Tools User's Guide*, Literature no. SPRU280G, 2003.
[3] Texas Instruments, Inc., *TMS320C55x Optimizing C Compiler User's Guide*, Literature no. SPRU281E, 2003.
[4] Texas Instruments, Inc., *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*, Literature no. SPRU374G, 2002.
[5] Texas Instruments, Inc., *TMS320C55x DSP Algebraic Instruction Set Reference Guide*, Literature no. SPRU375G, 2002.
[6] Texas Instruments, Inc., *TMS320C55x Programmer's Reference Guide*, Literature no. SPRU376A, 2001.
[7] Texas Instruments, Inc., *TMS320C55x DSP Peripherals Reference Guide*, Literature no. SPRU317G, 2004.
[8] ITU Recommendation G.711, 'Pulse code modulation (PCM) of voice frequencies,' *CCITT Series G Recommendations*, 1988.
[9] Texas Instruments, Inc., *TMS320VC5510 DSP External Memory Interface (EMIF) Reference Guide*, Literature no. SPRU590, Aug. 2004.
[10] Micron, Technology, Inc., *Synchronous DRAM MT48LC2M32B2 – 512K x 32 x 4 Banks*, Specification, Literature no. Advanced Micro Devices, Inc.
[11] Advance Micro Devices, Inc., *Am29LV400B 4 Megabit (512k x 8-Bit/256K x 16-Bit) COMS 3.0 Volt-only Boot Sector FLASH Memory Data Sheet*, July 2003.
[12] Texas Instruments, Inc., *TLV320AIC23B Stereo Audio Codec, 8-96 kHz, With Integrated Headphone Amplifier Data Manual*, Literature no. SLWS106F, 2004.

Exercises

1. Check the following examples to determine if these are correct parallel instructions. If not, correct the problems:

(a)
```
     mov *AR1+,AC1
  :: add @x,AR2
```

(b)
```
     mov AC0,dbl(*AR2+)
  :: mov dbl(*AR1+T0),AC2
```

(c)
```
     mpy *AR1+,*AR2+,AC0
  :: mpy *AR3+,*AR2+,AC1
  || rpt #127
```

2. Given a memory block, XAR0, XDP, and T0 as shown in Figure 2.24. Determine the contents of AC0, AR0, and T0 after the execution of the following instructions:

(a) `mov *(#x+2),AC0`

(b) `mov @(x-x+1),AC0`

(c) `mov @(x-x+0x80),AC0`

(d) `mov *AR0+,AC0`

(e) `mov *(AR0+T0),AC0`

(f) `mov *AR0(T0),AC0`

(g) `mov *AR0+T0,AC0`

(h) `mov *AR0(#-1),AC0`

(i) `mov *AR0(#2),AC0`

(j) `mov *AR0(#0x80),AC0`

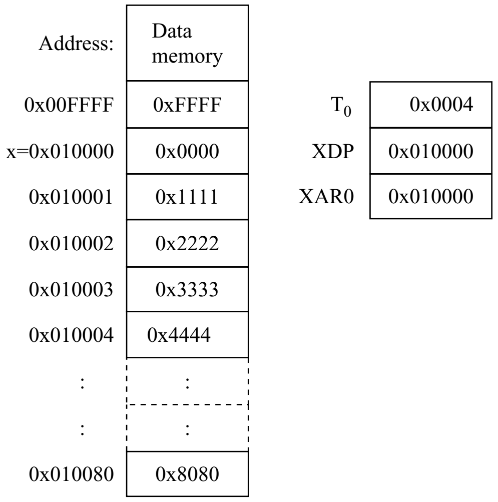


Figure 2.24 Contents of data memory and registers

3. Use Table 2.36 to show how the C compiler passes parameters for the following C functions:

(a)
```
short func_a(long, short, short, short, short, short,
                 short *, short *, short *, short *);
    var = func_a(0xD32E0E1D, 0, 1, 2, 3, 4,pa, pb, pc, pd);
```

(b)
```
short func_b(long, long, long, short, short, short,
                 short *, short *, short *, short *);
    var = func_b(0x12344321, 0, 1, 2, 3, 4,pa, pb, pc, pd);
```

(c)
```
long func_c(short, short *, short *, short, short, long,
                short *, short *, long, long);
var = func_c(0x2468ABCD, p0, p1, 1, 2, 0x1001,
                 p2, p3, 0x98765432, 0x0);
```

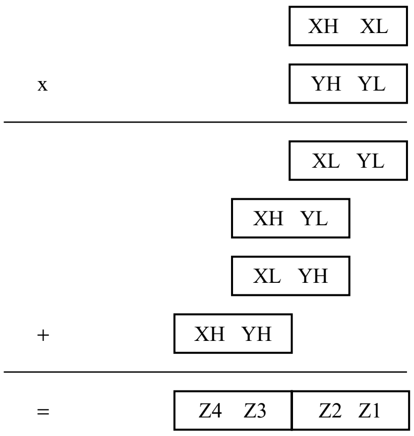
Table 2.36 List of parameters passed by the C55x C compiler

T0	T1	T2	T3	AC0	AC1	AC2	AC3
XAR0	XAR1	XAR2	XAR3	XAR4	XAR6	XAR6	XAR7
SP(−3)	SP(−2)	SP(−1)	SP(0)	SP(1)	SP(2)	SP(3)	var

4. The complex vector multiplication can be represented as $z = x \cdot y = (a + jb) \cdot (c + jd)$. The following C-callable assembly routine is written to compute the complex vector multiplication. Identify potential programming errors (bugs) in this assembly routine. Correct the errors and test it using CCS.

```
  .data
x   .word  1,2        ; Complex vector x = a+jb = 1+2j
y   .word  7,8        ; Complex vector y = c+jd = 7+8j
    .bss   z,2,1,1    ; For storing complex vector multiplication
    .global  complexVectMult

    .text
complexVectMult
    mov    #x, AR0
    mov    #Y, CDP
    mov    #Z, AR1
    mov    #1, T0
    mpy    *AR0,*CDP+,AC0       ; AC0 = a*c
::  mpy    *AR0(T0),*CDP+,AC1   ; AC1 = b*c
    mas    *AR0(T0),*CDP+,AC0   ; AC0 = a*c-b*d
::  mac    *AR0,*CDP+,AC1       ; AC1 = b*c+a*d
    mov    pair(LO(AC0)),*AR1+ ; Store the result (a+jb)(c+jd)=-9+22j
    .end
```

5. Some applications require the use of extended precision arithmetic. For example, the 32-bit by 32-bit integer multiplication will have a 64-bit result. The implementation of the double-precision multiplication can be described by the following figure:



The following assembly routine is written for computing the 32-bit by 32-bit integer multiplication. Identify potential programming errors (bugs) within this assembly routine. Correct the errors and test it using CCS.

```
    .data
x   .long  0x13579bdf          ; 32-bit data
y   .long  0x2468ace0          ; 32-bit data
    .bss   z,4,1,1             ; 64-bit result

    .global _mult32x32
    .text
_mult32x32
    amov   #x, XAR0
    amov   #y, XAR1
    amov   #z, XAR2
    amar   *AR0+
||  amar   *AR1+
    add    #3,AR2
||  mpym   *AR0-, *AR1,AC0          ; AC0 = XL*YL
    mov    AC0,*AR2-                ; Save Z1
    macm   *AR0+,*AR1-,AC0>>16,AC0  ; AC0 = (XH*YL) + (XL*YL)>>16
    macm   *AR0-,*AR1,AC0           ; AC0 += (XL*YH)
    mov    AC0,*AR2-                ; Save Z2
    macm   *AR0,*AR1,AC0>>16,AC0    ; AC0 = (XH*YH) + (AC0)>>16
    mov    AC0,*AR2-                ; Save Z3
    mov    HI(AC0),*AR2             ; Save Z4
    ret
    .end
```

6. Based on the previous experiment on interfacing C with assembly code, write a C-callable assembly function to compute $d = a \cdot b \cdot c$, where $a = 0\text{x}400$, $b = 0\text{x}600$, and $c = 0\text{x}4000$. The assembly function should pass three variables a, b, and c into the assembly routine, and return the result to the C function. Check the result and explain why.

7. Refer to the previous experiment on addressing modes using assembly programming to write an assembly routine that uses indirect addressing mode to compute an 8×8 matrix of $B = A' \cdot X$.

8. Write an assembly routine computing the following matrix operation to obtain `Y`, `Cr`, and `Cb` from given `R`, `G`, and `B` data. The values of the `R`, `G`, and `B` are 8-bit integers.

$$\begin{bmatrix} 65 & 129 & 25 \\ -38 & -74 & 112 \\ 112 & -94 & -18 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} = \begin{bmatrix} Y \\ Cr \\ Cb \end{bmatrix}.$$

9. Write a C-callable assembly function that performs the following functions:

   (a) Find the maximum and minimum values of the data file '`dtmf18005551234.dat`', which is used by previous experiment on programming flash memory devices.

   (b) Calculate the average (mean) value of the data file '`dtmf18005551234.dat`'.

10. Write a C-callable assembly function that sorts the data file '`dtmf18005551234.dat`' and write the sorting result into a memory location starting from the maximum in a decent order. Using the CCS graphic feature, plot both the '`dtmf18005551234.dat`' and the sorted result.

11. Based on the experiment program examples given in Sections 2.10.5 and 2.10.7, write a program that writes the data file '`dtmf18005551234.dat`' into flash memory and playback the data from flash memory using C5510 DSK. Add a timer that generates interrupt every 10 s. Using this timer automatically plays back the data file stored in flash memory every 10 s. In this experiment, we will learn how to set up C55x timers and create a timer interrupt for a given rate.

12. We introduced flash memory programming in Section 2.10.5. The flash erase and programming are done for whole chip. Refer to the flash memory datasheet to develop a flash program that can erase sections instead of

the whole chip and program memory in selected sections. This experiment is intended to create a flash memory update utility.

13. Based on the AIC23 experiment given in Section 2.10.7, develop an audio loopback program. The sampling rate of the AIC23 is 8 kHz in 16-bit data format. The DSK audio input is the stereo line-in and the DSK output is the stereo headphone output. Connect an audio source such as a CD player to the DSK audio input and playback the audio output through the DSK headphone output. Adjust the AIC23 control registers to set proper gain of the input and output signal levels. In this experiment, we will learn the detailed control register settings for the AIC23, and will be able to adjust the AIC23 to meet the application requirements.

14. Modify above real-time DSP program to make the audio loopback using McBSP interrupt event and ISR instead of polling the McBSP status bit. From this experiment, we will learn how the interrupt and ISR work in conjunction with McBSP.

15. Modify the audio loopback program to use DMA channels, so the audio samples will be buffered into 80 samples for the transmitter and receiver. The interrupt to the transmitter and receiver should occur every 80 samples. In this experiment, the audio is processed in blocks of 80 samples. This is a challenging experiment that requires the knowledge of DMA, McBSP, flash memory, AIC23, and the DSK system. The DMA event and McBSP interrupt along with audio sample management are all involved.

16. Create an experiment that configures the DSK for multichannel DMA data transfer. For the first DMA data transfer, the data source is in SRAM and the destination is SDRAM. The transfer data size is 8192 bytes. For the second DMA data transfer, the data source is in SDRAM and the destination is DARAM. The transfer data size is 4096 bytes. Write the program such that the data transfers will be performed at the same time for both DMA paths.

3

DSP Fundamentals and Implementation Considerations

This chapter presents fundamental DSP concepts and practical implementation considerations for the digital filters and algorithms. DSP implementations, especially using fixed-point processors, require special attention due to the quantization and arithmetic errors.

3.1 Digital Signals and Systems

In this section, we will introduce some widely used digital signals and simple DSP systems.

3.1.1 Elementary Digital Signals

Signals can be classified as deterministic or random. Deterministic signals are used for test purposes and can be described mathematically. Random signals are information-bearing signals such as speech. Some deterministic signals will be introduced in this section, while random signals will be discussed in Section 3.3.

A digital signal is a sequence of numbers $x(n)$, $-\infty < n < \infty$, where n is the time index. The `unit-impulse sequence`, with only one nonzero value at $n = 0$, is defined as

$$\delta(n) = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}, \tag{3.1}$$

where $\delta(n)$ is also called the Kronecker delta function. This unit-impulse sequence is very useful for testing and analyzing the characteristics of DSP systems.

The `unit-step` sequence is defined as

$$u(n) = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}. \tag{3.2}$$

This function is very convenient for describing causal signals, which are the most commonly encountered signals in real-time DSP systems.

Sinusoidal signals (sinusoids, tones, or sinewaves) can be expressed in a simple mathematical formula. An analog sinewave can be expressed as

$$x(t) = A\sin(\Omega t + \phi) = A\sin(2\pi f t + \phi), \tag{3.3}$$

where A is the amplitude of the sinewave.

$$\Omega = 2\pi f \tag{3.4}$$

is the frequency in radians per second (rad/s), f is the frequency in cycles per second (Hz), and ϕ is the phase in radians.

The digital signal corresponding to the analog sinewave defined in Equation (3.3) can be expressed as

$$x(n) = A\sin(\Omega nT + \phi) = A\sin(2\pi fnT + \phi), \tag{3.5}$$

where T is the sampling period in seconds. This digital sequence can also be expressed as

$$x(n) = A\sin(\omega n + \phi) = A\sin(\pi F n + \phi), \tag{3.6}$$

where

$$\omega = \Omega T = \frac{2\pi f}{f_s} \tag{3.7}$$

is the digital frequency in radians per sample, and

$$F = \frac{\omega}{\pi} = \frac{f}{(f_s/2)} \tag{3.8}$$

is the normalized digital frequency in cycles per sample.

The units, relationships, and ranges of these analog and digital frequency variables are summarized in Table 3.1. Sampling of analog signals implies a mapping of an infinite range of analog frequency variable f (or Ω) into a finite range of digital frequency variable F (or ω). The highest frequency in a digital signal is $F = 1$ (or $\omega = \pi$) based on the sampling theorem defined in Equation (1.3). Therefore, the spectrum of digital signals is restricted to a limited range as shown in Table 3.1.

Example 3.1: Generate 32 samples of a sinewave with $A = 2$, $f = 1000\,\text{Hz}$, and $f_s = 8\,\text{kHz}$ using MATLAB program.

Since $F = \frac{f}{(f_s/2)} = 0.25$, we have $\omega = \pi F = 0.25\pi$. From Equation (3.6), we can express the generated sinewave as $x(n) = 2\sin(\omega n)$, $n = 0, 1, \ldots, 31$. The generated sinewave samples are

Table 3.1 Units, relationships, and ranges of four frequency variables

Variables	Unit	Relationship	Range
Ω	Radians per second	$\Omega = 2\pi f$	$-\infty < \Omega < \infty$
f	Cycles per second (Hz)	$f = \frac{Ff_s}{2} = \frac{\omega}{2\pi T}$	$-\infty < f < \infty$
ω	Radians per sample	$\omega = \Omega T = \pi F$	$-\pi \leq \omega \leq \pi$
F	Cycles per sample	$F = \frac{f}{(f_s/2)}$	$-1 \leq F \leq 1$

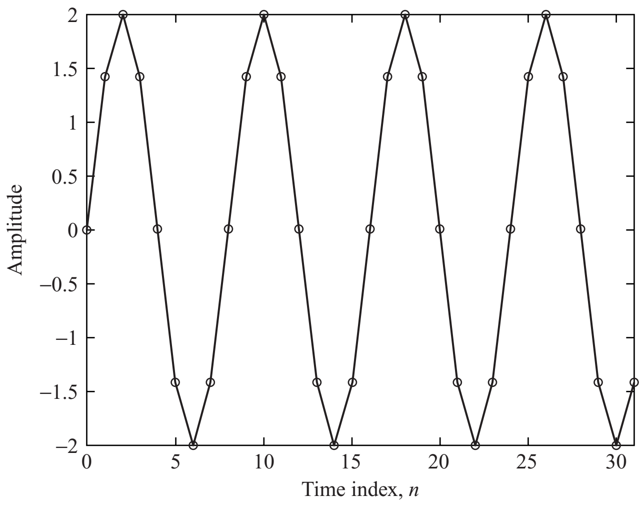


Figure 3.1 An example of sinewave with $A = 2$ and $\omega = 0.25\pi$

plotted (shown in Figure 3.1) and saved in a data file (`sine.dat`) using ASCII format using the following MATLAB script (`example3_1.m`):

```
n = [0:31];              % Time index n
omega = 0.25*pi;         % Digital frequency
xn = 2*sin(omega*n);     % Sinewave generation
plot(n, xn, '-o');       % Samples are marked by 'o'
xlabel('Time index, n');
ylabel('Amplitude');
axis([0 31 -2 2]);
save sine.dat xn -ascii ;
```

Note that $F = 0.25$ means there are four samples from 0 to π, resulting in eight samples per period of sinewave, which is clearly indicated in Figure 3.1.

3.1.2 Block Diagram Representation of Digital Systems

A DSP system performs prescribed operations on signals. The processing of digital signals can be described as combinations of certain basic operations including addition (or subtraction), multiplication, and time shift (or delay). Thus, a DSP system consists of the interconnection of three basic elements: adders, multipliers, and delay units.

Two signals, $x_1(n)$ and $x_2(n)$, can be added as illustrated in Figure 3.2, where the adder output is expressed as

$$y(n) = x_1(n) + x_2(n). \tag{3.9}$$

The adder could be drawn as a multi-input adder with more than two inputs, but the additions are typically performed with two inputs at a time. The addition operation of Equation (3.9) can be implemented as the

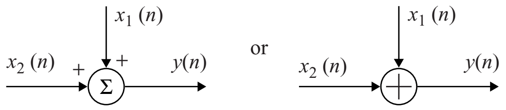


Figure 3.2 Block diagram of an adder

following C55x code using direct addressing mode:

```
mov    @x1n,AC0       ; AC0 = x1(n)
add    @x2n,AC0       ; AC0 = x1(n)+x2(n)
mov    AC0,@yn        ; y = x1(n)+x2(n)
```

A given signal can be multiplied by a scalar, α, as illustrated in Figure 3.3, where $x(n)$ is the multiplier input and the multiplier's output is

$$y(n) = \alpha x(n). \tag{3.10}$$

Multiplication of a sequence by a scalar, α, results in a sequence that is scaled by α. The output signal is amplified if $|\alpha| > 1$, or attenuated if $|\alpha| < 1$. The multiply operation of Equation (3.10) can be implemented as the following C55x code using indirect addressing mode:

```
amov   #alpha,XAR1     ; AR1 points to alpha (α)
amov   #xn,XAR2        ; AR2 points to x(n)
amov   #yn,XAR3        ; AR3 points to y(n)
mpy    *AR1,*AR2,AC0   ; AC0 = α *x(n)
mov    AC0,*AR3        ; y = α *x(n)
```

The sequence $x(n)$ can be delayed in time by one sampling period, T, as illustrated in Figure 3.4, where the box labeled z^{-1} represents the unit delay, $x(n)$ is the input signal, and the output signal

$$y(n) = x(n-1). \tag{3.11}$$

In fact, the signal $x(n-1)$ is actually the previously stored signal in memory before the current time n. Therefore, the delay unit is very easy to realize in a digital system with memory, but is difficult to implement in an analog system. A delay by more than one unit can be implemented by cascading several delay units in a row. Therefore, an L-unit delay requires L memory locations configured as a first-in first-out buffer (tapped-delay line or simply delay line) in memory.

There are several methods to implement delay operations on the TMS320C55x. The following code uses a delay instruction to move the contents of the addressed data memory location into the next higher address location:

```
amov   #xn,XAR1      ; AR1 points to x(n)
delay  *AR1          ; Contents of x(n) is copied to x(n-1)
```

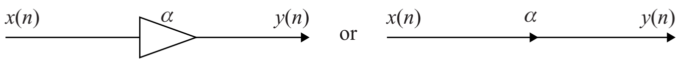


Figure 3.3 Block diagram of a multiplier

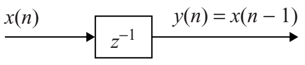


Figure 3.4 Block diagram of a unit delay

Example 3.2: Consider a simple DSP system described by the difference equation

$$y(n) = \alpha x(n) + \alpha x(n-1). \tag{3.12}$$

The block diagram of the system using the three basic building blocks is sketched in Figure 3.5(a), which shows that the output signal $y(n)$ is computed using two multiplications and one addition. A simple algebraic simplification may be used to reduce computational requirements. For example, Equation (3.12) can be rewritten as

$$y(n) = \alpha\, [x(n) + x(n-1)]\,. \tag{3.13}$$

The implementation of this difference equation is illustrated in Figure 3.5(b), where only one multiplication is required. This example shows that with careful design (or optimization), the complexity of the system (or algorithm) can be further reduced.

Example 3.3: In practice, the complexity of algorithm also depends on the architecture and instruction set of the DSP processor. For example, the C55x implementation of Equation (3.13) can be written as

```
amov   #alpha,XAR1       ; AR1 points to α
amov   #temp,XAR2        ; AR2 points to temp
amov   #yn,XAR4          ; AR4 points to yn
mov    *(x1n),AC0        ; AC0 = x1(n)
add    *(x2n),AC0        ; AC0 = x1(n)+x2(n)
```

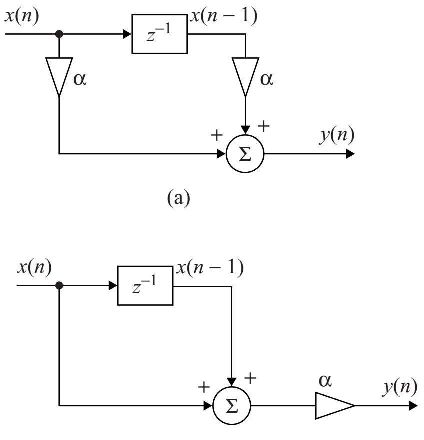


Figure 3.5 Block diagrams of DSP systems: (a) direct realization described in (3.12); (b) simplified implementation given in (3.13)

```
    mov     AC0,*AR2          ; temp = x1(n)+x2(n), pointed by AR2
    mpy     *AR1,*AR2,AC1     ; AC1 = α *[x1(n)+x2(n)]
    mov     AC1,*AR4          ; yn = α *[x1(n)+x2(n)]
```

Equation (3.12) can be implemented as

```
    amov    #x1n,XAR1         ; AR1 points to x1(n)
    amov    #x2n,XAR2         ; AR2 points to x2(n)
    amov    #alpha,XAR3       ; AR3 points to α
    amov    #yn,XAR4          ; AR4 points to yn
    mpy     *AR1,*AR3,AC1     ; AC1 = α *x1(n)
    mac     *AR2,*AR3,AC1     ; AC1 = α *x1(n)+ α *x2(n)
    mov     AC1,*AR4          ; yn = α *x1(n)+ α *x2(n)
```

This example shows Equation (3.12) is more efficient for implementation on the TMS320C55x because its architecture is optimized for the sum of products operation. Therefore, the complexity of DSP algorithm cannot be simply measured by the number of required multiplications.

When the multiplier coefficient α is a number with a base of 2 such as 0.25 (1/4), we can use shift operation instead of multiplication. The following example uses the absolute addressing mode:

```
    mov    *(x1n)<<#-2,AC0      ; AC0 = 0.25*x1(n)
    add    *(x2n)<<#-2,AC0      ; AC0 = 0.25*x1(n)+0.25*x2(n)
```

where the right-shift option, `<<#-2`, shifts the contents of `x1n` and `x2n` to the right by 2 bits. This is equivalent to dividing the number by 4.

3.2 System Concepts

In this section, we introduce several techniques for describing and analyzing the linear time-invariant (LTI) digital systems.

3.2.1 Linear Time-Invariant Systems

If the input signal to an LTI system is the unit-impulse sequence $\delta(n)$ defined in Equation (3.1), then the output signal is called the impulse response of the system, $h(n)$.

Example 3.4: Consider a digital system with the I/O equation

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2). \tag{3.14}$$

Applying the unit-impulse sequence $\delta(n)$ to the input of the system, the outputs are the impulse response coefficients and can be computed as follows:

$$\begin{aligned} h(0) &= y(0) = b_0 \cdot 1 + b_1 \cdot 0 + b_2 \cdot 0 = b_0 \\ h(1) &= y(1) = b_0 \cdot 0 + b_1 \cdot 1 + b_2 \cdot 0 = b_1 \\ h(2) &= y(2) = b_0 \cdot 0 + b_1 \cdot 0 + b_2 \cdot 1 = b_2 \\ h(3) &= y(3) = b_0 \cdot 0 + b_1 \cdot 0 + b_2 \cdot 0 = 0 \\ &\vdots \end{aligned}$$

Therefore, the impulse response of the system defined in Equation (3.14) is $\{b_0, b_1, b_2, 0, 0, \ldots\}$.

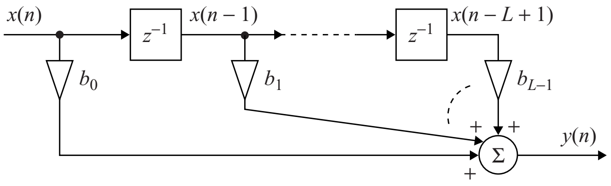


Figure 3.6 Detailed signal-flow diagram of an FIR filter

The I/O equation given in (3.14) can be generalized with L coefficients, expressed as

$$\begin{aligned} y(n) &= b_0x(n) + b_1x(n-1) + \cdots + b_{L-1}x(n-L+1) \\ &= \sum_{l=0}^{L-1} b_l x(n-l). \end{aligned} \tag{3.15}$$

Substituting $x(n) = \delta(n)$ into Equation (3.15), the output is the impulse response expressed as

$$\begin{aligned} h(n) &= \sum_{l=0}^{L-1} b_l \delta(n-l) \\ &= \begin{cases} b_n & n = 0, 1, \ldots, L-1 \\ 0 & \text{otherwise} \end{cases}. \end{aligned} \tag{3.16}$$

Therefore, the length of the impulse response is L for the system defined in Equation (3.15). Such a system is called a finite impulse response (FIR) filter. The coefficients, b_l, $l = 0, 1, \ldots, L-1$, are called filter coefficients (also called as weights or taps). For FIR filters, the filter coefficients are identical to the impulse response coefficients.

The signal-flow diagram of the system described by the I/O equation (3.15) is illustrated in Figure 3.6. The string of z^{-1} units is called a tapped-delay line. The parameter, L, is the length of the FIR filter. Note that the order of filter is $L-1$ for the FIR filter with length L since they are $L-1$ zeros. The design and implementation of FIR filters will be further discussed in Chapter 4.

The `moving (running) average` filter is a simple example of FIR filter. Consider an L-point moving-average filter defined as

$$\begin{aligned} y(n) &= \frac{1}{L}\left[x(n) + x(n-1) + \cdots + x(n-L+1)\right] \\ &= \frac{1}{L}\sum_{l=0}^{L-1} x(n-l), \end{aligned} \tag{3.17}$$

where each output signal is the average of L consecutive input samples. Implementation of Equation (3.17) requires $L-1$ additions and L memory locations for storing signal samples $x(n)$, $x(n-1), \ldots,$ $x(n-L+1)$ in a memory buffer. Note that the division by a constant L can be implemented by multiplication of constant α, where $\alpha = 1/L$.

As illustrated in Figure 3.6, the signal samples used to compute the output signal are L samples included in the window at time n. These samples are almost the same as those samples used for the previous window at time $n-1$ to compute $y(n-1)$, except that the oldest sample $x(n-L)$ of the window at time $n-1$ is replaced by the newest sample $x(n)$ of the window at time n. The concept of moving window is illustrated

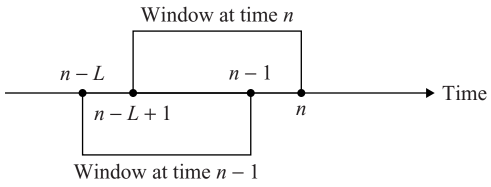


Figure 3.7 Time windows at current time n and previous time $n-1$

in Figure 3.7. Therefore, the averaged signal, $y(n)$, can be computed recursively as

$$y(n) = y(n-1) + \frac{1}{L}\left[x(n) - x(n-L)\right]. \tag{3.18}$$

This recursive equation can be realized by using only two additions. However, we still need $L+1$ memory locations for keeping $L+1$ signal samples $[x(n)x(n-1)\ldots x(n-L)]$.

Example 3.5: The following C55x assembly code illustrates the implementation of the moving-average filter of $L = 8$ based on Equation (3.18):

```
L                .set 8            ; Length of filter
xin              .usect "indata",1
xbuffer          .usect "indata",L ; Length of buffer
y                .usect "outdata",2,1,1   ; Long-word format
    amov        #xbuffer+L-1,XAR3   ; AR3 points to end of x buffer
    amov        #xbuffer+L-2,XAR2   ; AR2 points to next sample
    mov         dbl(*(y)),AC1       ; AC1 = y(n-1) in long format
    mov         *(xin),AC0          ; AC0 = x(n)
    sub         *AR3,AC0            ; AC0 = x(n)-x(n-L)
    add         AC0,#-3,AC1         ; AC0 = y(n-1)+1/L[x(n)-x(n-L)]
    mov         AC1,dbl(*(y))       ; AC1 = y(n)
    rpt         #(L-2)              ; Update the tapped-delay-line
    mov         *AR2-,*AR3-         ; x(n-1) = x(n)
    mov         *(xin),AC0          ; Update the newest sample x(n)
    mov         AC0,*AR3            ; x(n) = input xin
```

Consider an LTI system illustrated in Figure 3.8, the output of the system can be expressed as

$$\begin{aligned} y(n) &= x(n) * h(n) = h(n) * x(n) \\ &= \sum_{l=-\infty}^{\infty} x(l)h(n-l) = \sum_{l=-\infty}^{\infty} h(l)x(n-l), \end{aligned} \tag{3.19}$$

where * denotes the linear convolution. The exact internal structure of the system is either unknown or ignored. The only way to interact with the system is by using its input and output terminals as shown in Figure 3.8. This ‘black box’ representation is a very effective way to depict complicated DSP systems.

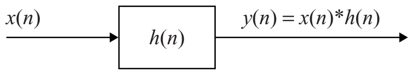


Figure 3.8 An LTI system expressed in time domain

A digital system is called the causal system if and only if

$$h(n) = 0, \qquad n < 0. \tag{3.20}$$

A causal system does not provide a zero-state response prior to input application; that is, the output depends only on the present and previous samples of the input. This is an obvious property for real-time DSP systems since we simply do not have future data. However, if the data is recorded and processed later, the algorithm operating on this data set does not need to be causal. For a causal system, the limits on the summation of Equation (3.19) can be modified to reflect this restriction as

$$y(n) = \sum_{l=0}^{\infty} h(l)x(n-l). \tag{3.21}$$

Example 3.6: Consider the I/O equation of the digital system expressed as

$$y(n) = bx(n) - ay(n-1), \tag{3.22}$$

where each output signal $y(n)$ is dependent on the current input signal $x(n)$ and the previous output signal $y(n-1)$. Assuming that the system is causal, i.e., $y(n) = 0$ for $n < 0$ and let $x(n) = \delta(n)$. The output signals are computed as

$$\begin{aligned} y(0) &= bx(0) - ay(-1) = b \\ y(1) &= bx(1) - ay(0) = -ay(0) = -ab \\ y(2) &= bx(2) - ay(1) = -ay(1) = a^2b \\ &\vdots \end{aligned}$$

In general, we have

$$y(n) = (-1)^n a^n b, \quad n = 0, 1, 2, \ldots, \infty.$$

This system has infinite impulse response $h(n)$ if the coefficients a and b are nonzero.

A digital filter can be classified as either an FIR filter or an infinite impulse response (IIR) filter, depending on whether or not the impulse response of the filter is of finite or infinite duration. The system defined in Equation (3.22) is an IIR system (or filter) since it has infinite impulse response as shown in Example 3.6. The I/O equation of the IIR system can be generalized as

$$\begin{aligned} y(n) &= b_0x(n) + b_1x(n-1) + \cdots + b_{L-1}x(n-L+1) - a_1y(n-1) - \cdots - a_My(n-M) \\ &= \sum_{l=0}^{L-1} b_lx(n-l) - \sum_{m=1}^{M} a_my(n-m). \end{aligned} \tag{3.23}$$

This IIR system is represented by a set of feedforward coefficients $\{b_l, l = 0, 1, \ldots, L-1\}$ and a set of feedback coefficients $\{a_m, m = 1, 2, \ldots, M\}$. Since the outputs are fed back and combined with the weighted inputs, IIR systems are feedback systems. Note that when all a_m are zero, Equation (3.23) is identical to Equation (3.15). Therefore, an FIR filter is a special case of an IIR filter without feedback coefficients.

Example 3.7: The IIR filters given in Equation (3.23) can be implemented using the MATLAB function `filter` as follows:

```
yn = filter(b, a, xn);
```

The vector `b` contains feedforward coefficients $\{b_l,\ l = 0, 1, \ldots, L-1\}$ and the vector `a` contains feedback coefficients $\{a_m, m = 0, 1,\ 2, \ldots, M$, where $a_0 = 1\}$. The signal vectors `xn` and `yn` are the input and output buffers of the system, respectively. The FIR filter defined in Equation (3.15) can be implemented as follows:

```
yn = filter(b, 1, xn);
```

This is because all a_m are zero except $a_0 = 1$ for an FIR filter.

Example 3.8: Assume that L is large enough so that the oldest sample $x(n-L)$ can be approximated by its average $y(n-1)$. The moving-average filter defined in Equation (3.18) can be approximated as

$$\begin{aligned} y(n) &\cong \left(1 - \frac{1}{L}\right) y(n-1) + \frac{1}{L}x(n) \\ &= (1-\alpha)\, y(n-1) + \alpha x(n), \end{aligned} \tag{3.24}$$

where $\alpha = 1/L$. This is a simple first-order IIR filter. Compared with Equation (3.18), we need two multiplications instead of one, but only need two memory locations instead of $L + 1$. Thus, Equation (3.24) is the most efficient way of approximating a moving-average filtering.

3.2.2 The *z*-Transform

Continuous-time systems are commonly analyzed using the Laplace transform. For discrete-time systems, the transform corresponding to the Laplace transform is the z-transform. The z-transform (ZT[·]) of a digital signal, $x(n)$, $-\infty < n < \infty$, is defined as the power series:

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n}, \tag{3.25}$$

where $X(z)$ represents the z-transform of $x(n)$. The variable z is a complex variable, and can be expressed in polar form as

$$z = re^{j\theta}, \tag{3.26}$$

where r is the magnitude (radius) of z and θ is the angle of z. When $r = 1$, $|z| = 1$ is called the unit circle on the z-plane. Since the z-transform involves an infinite power series, it exists only for those values of z where the power series defined in Equation (3.25) converges. The region on the complex z-plane in which the power series converges is called the region of convergence.

For causal signals, the two-sided z-transform defined in Equation (3.25) becomes a one-sided z-transform expressed as

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n}. \tag{3.27}$$

Example 3.9: Consider the exponential function

$$x(n) = a^n u(n).$$

The z-transform can be computed as

$$X(z) = \sum_{n=-\infty}^{\infty} a^n z^{-n} u(n) = \sum_{n=0}^{\infty} \left(az^{-1}\right)^n.$$

Using the infinite geometric series given in Appendix A, we have

$$X(z) = \frac{1}{1 - az^{-1}} = \frac{z}{z - a} \quad \text{if} \quad \left|az^{-1}\right| < 1.$$

The equivalent condition for convergence is

$$|z| > |a|,$$

which is the region outside the circle with radius a.

The properties of the z-transform are extremely useful for the analysis of discrete-time LTI systems. These properties are summarized as follows:

1. *Linearity (superposition)*: The z-transform of the sum of two sequences is the sum of the z-transforms of the individual sequences. That is,

$$\begin{aligned} \text{ZT}\left[a_1 x_1(n) + a_2 x_2(n)\right] &= a_1\, \text{ZT}\left[x_1(n)\right] + a_2\, \text{ZT}\left[x_2(n)\right] \\ &= a_1 X_1(z) + a_2 X_2(z), \end{aligned} \tag{3.28}$$

   where a_1 and a_2 are constants.

2. *Time shifting*: The z-transform of the shifted (delayed) signal $y(n) = x(n-k)$ is

$$Y(z) = \text{ZT}\left[x(n-k)\right] = z^{-k} X(z). \tag{3.29}$$

   Thus, the effect of delaying a signal by k samples is equivalent to multiplying its z-transform by a factor of z^{-k}. For example, $\text{ZT}\left[x(n-1)\right] = z^{-1}X(z)$. The unit delay z^{-1} corresponds to a time shift of one sample in the time domain.

3. *Convolution*: Consider the signal

$$x(n) = x_1(n) * x_2(n), \tag{3.30}$$

   we have

$$X(z) = X_1(z) X_2(z). \tag{3.31}$$

   The z-transform converts the convolution in time domain to the multiplication in z domain.

The inverse z-transform is defined as

$$x(n) = \mathrm{ZT}^{-1}[X(z)] = \frac{1}{2\pi j}\oint_C X(z)z^{n-1}dz, \tag{3.32}$$

where C denotes the closed contour of $X(z)$ taken in a counterclockwise direction. Several methods are available for finding the inverse z-transform: long division, partial-fraction expansion, and residue method. A limitation of the long-division method is that it does not lead to a closed form solution. However, it is simple and lends itself to software implementation. Both the partial-fraction-expansion and the residue methods lead to closed form solutions. The main disadvantage is the need to factorize the denominator polynomial, which is difficult if the order of $X(z)$ is high.

3.2.3 Transfer Functions

Consider the LTI system illustrated in Figure 3.8. Using the convolution property, we have

$$Y(z) = X(z)H(z), \tag{3.33}$$

where $X(z) = \mathrm{ZT}[x(n)]$, $Y(z) = \mathrm{ZT}[y(n)]$, and $H(z) = \mathrm{ZT}[h(n)]$. The combination of time- and frequency-domain representations of LTI system is illustrated in Figure 3.9. This diagram shows that we can replace the time-domain convolution by the z-domain multiplication.

The transfer function of an LTI system is defined in terms of the system's input and output. From Equation (3.33), we have

$$H(z) = \frac{Y(z)}{X(z)}. \tag{3.34}$$

The z-transform can be used in creating alternative filters that have exactly the same input–output behavior. An important example is the cascade or parallel connection of two or more systems, as illustrated in Figure 3.10. In the cascade (series) interconnection shown in Figure 3.10(a), we have

$$Y_1(z) = X(z)H_1(z) \quad \text{and} \quad Y(z) = Y_1(z)H_2(z).$$

Thus,

$$Y(z) = X(z)H_1(z)H_2(z).$$

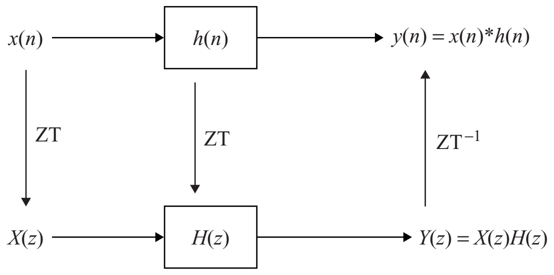
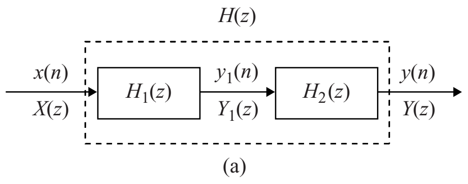

$$\begin{tikzcd}
x(n) \arrow[r] \arrow[d, "\mathrm{ZT}"] & \boxed{h(n)} \arrow[r] \arrow[d, "\mathrm{ZT}"] & y(n) = x(n)*h(n) \\
X(z) \arrow[r] & \boxed{H(z)} \arrow[r] & Y(z) = X(z)H(z) \arrow[u, "\mathrm{ZT}^{-1}"]
\end{tikzcd}$$

Figure 3.9 A block diagram of LTI system in both time domain and z domain



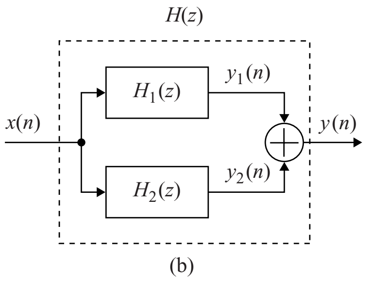


Figure 3.10 Interconnect of digital systems: (a) cascade form; (b) parallel form

Therefore, the overall transfer function of the cascade of the two systems is

$$H(z) = H_1(z)H_2(z) = H_2(z)H_1(z). \tag{3.35}$$

Since multiplication is commutative, the two systems can be cascaded in either order to obtain the same overall system. The overall impulse response of the system is

$$h(n) = h_1(n) * h_2(n) = h_2(n) * h_1(n). \tag{3.36}$$

Similarly, the overall impulse response and transfer function of the parallel connection of two LTI systems shown in Figure 3.10(b) are given by

$$h(n) = h_1(n) + h_2(n) \tag{3.37}$$

and

$$H(z) = H_1(z) + H_2(z). \tag{3.38}$$

If we can multiply several transfer functions to get a higher-order system, we can also factor polynomials to break down a large system into smaller sections. The concept of parallel and cascade implementation will be further discussed in the realization of IIR filters in Chapter 5.

Example 3.10: The LTI system with transfer function

$$H(z) = 1 - 2z^{-1} + z^{-3}$$

can be factored as

$$H(z) = \left(1 - z^{-1}\right)\left(1 - z^{-1} - z^{-2}\right) = H_1(z)H_2(z).$$

Thus, the overall system $H(z)$ can be realized as the cascade of the first-order system $H_1(z) = 1 - z^{-1}$ and the second-order system $H_2(z) = 1 - z^{-1} - z^{-2}$.

The I/O equation of an FIR filter is given in Equation (3.15). Taking the z-transform of both sides, we have

$$\begin{aligned} Y(z) &= b_0 X(z) + b_1 z^{-1} X(z) + \cdots + b_{L-1} z^{-(L-1)} X(z) \\ &= \left(b_0 + b_1 z^{-1} + \cdots + b_{L-1} z^{-(L-1)}\right) X(z). \end{aligned} \tag{3.39}$$

Therefore, the transfer function of the FIR filter is expressed as

$$H(z) = b_0 + b_1 z^{-1} + \cdots + b_{L-1} z^{-(L-1)} = \sum_{l=0}^{L-1} b_l z^{-l}. \tag{3.40}$$

Similarly, taking the z-transform of both sides of the IIR filter defined in Equation (3.23) yields

$$\begin{aligned} Y(z) &= b_0 X(z) + b_1 z^{-1} X(z) + \cdots + b_{L-1} z^{-L+1} X(z) - a_1 z^{-1} Y(z) - \cdots - a_M z^{-M} Y(z) \\ &= \left(\sum_{l=0}^{L-1} b_l z^{-l}\right) X(z) - \left(\sum_{m=1}^{M} a_m z^{-m}\right) Y(z). \end{aligned} \tag{3.41}$$

By rearranging the terms, we can derive the transfer function of the IIR filter as

$$H(z) = \frac{\sum\limits_{l=0}^{L-1} b_l z^{-l}}{1 + \sum\limits_{m=1}^{M} a_m z^{-m}} = \frac{\sum\limits_{l=0}^{L-1} b_l z^{-l}}{\sum\limits_{m=0}^{M} a_m z^{-m}}, \tag{3.42}$$

where $a_0 = 1$. A detailed block diagram of an IIR filter is illustrated in Figure 3.11 for $M = L - 1$.

Example 3.11: Consider the moving-average filter given in Equation (3.17). Taking the z-transform of both sides, we have

$$Y(z) = \frac{1}{L} \sum_{l=0}^{L-1} z^{-l} X(z).$$

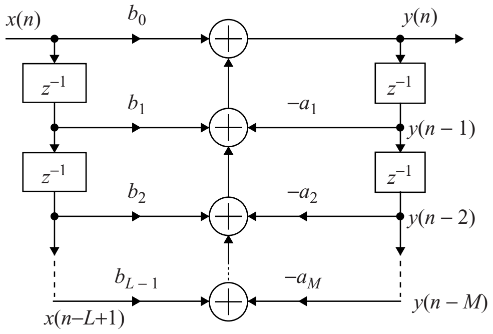


Figure 3.11 Detailed signal-flow diagram of an IIR filter

Using the geometric series defined in Appendix A, the transfer function of the filter can be expressed as

$$H(z) = \frac{1}{L}\sum_{l=0}^{L-1} z^{-l} = \frac{1}{L}\left(\frac{1-z^{-L}}{1-z^{-1}}\right) = \frac{Y(z)}{X(z)}. \tag{3.43}$$

This equation can be rearranged as

$$Y(z) = z^{-1}Y(z) + \frac{1}{L}\left[X(z) - z^{-L}X(z)\right].$$

Taking the inverse z-transform of both sides, we obtain

$$y(n) = y(n-1) + \frac{1}{L}\left[x(n) - x(n-L)\right].$$

This is an effective way of deriving Equation (3.18) from (3.17).

3.2.4 Poles and Zeros

Factoring the numerator and denominator polynomials of $H(z)$, Equation (3.42) can be expressed as the following rational function:

$$H(z) = b_0 \frac{\prod_{l=1}^{L-1}(z - z_l)}{\prod_{m=1}^{M}(z - p_m)} = \frac{b_0(z - z_1)(z - z_2)\cdots(z - z_{L-1})}{(z - p_1)(z - p_2)\cdots(z - p_M)}. \tag{3.44}$$

The roots of the numerator polynomial are the zeros of the transfer function $H(z)$ since they are the values of z for which $H(z) = 0$. Thus, $H(z)$ given in Equation (3.44) has $(L-1)$ zeros at $z = z_1, z_2, \ldots, z_{L-1}$. The roots of the denominator polynomial are the poles since they are the values of z such that $H(z) = \infty$, and there are M poles at $z = p_1, p_2, \ldots, p_M$. The LTI system described in Equation (3.44) is a pole-zero system, while the system described in Equation (3.40) is an all-zero system.

Example 3.12: The roots of the numerator polynomial defined in Equation (3.43) determine the zeros of $H(z)$, i.e., $z^L - 1 = 0$. Using the complex arithmetic given in Appendix A, we have

$$z_l = e^{j(2\pi/L)l}, \quad l = 0, 1, \ldots, L-1. \tag{3.45}$$

Therefore, there are L equally spaced zeros on the unit circle $|z| = 1$.

Similarly, the poles of $H(z)$ are determined by the roots of the denominator $z^{L-1}(z-1)$. Thus, there are $L-1$ poles at the origin $z = 0$ and one pole at $z = 1$. A pole-zero diagram of $H(z)$ for $L = 8$ on the complex z-plane is illustrated in Figure 3.12.

The pole-zero diagram provides an insight into the properties of an LTI system. To find poles and zeros of a rational function $H(z)$, we can use the MATLAB function `roots` on both the numerator and denominator polynomials. Another useful MATLAB function for analyzing transfer function is `zplane(b,a)`, which displays the pole-zero diagram of $H(z)$.

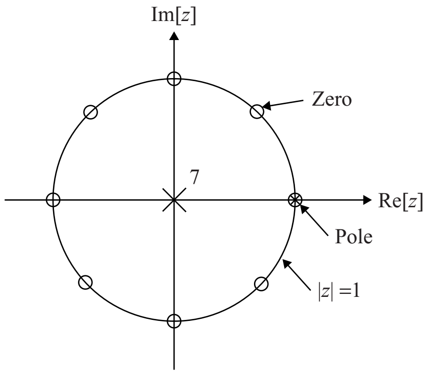


Figure 3.12 Pole-zero diagram of the moving-average filter, $L = 8$

Example 3.13: Consider the IIR filter with the transfer function

$$H(z) = \frac{1}{1 - z^{-1} + 0.9z^{-2}}.$$

We can plot the pole-zero diagrams using the following MATLAB script (`example3_13a.m`):

```
b=[1];
a=[1, -1, 0.9];
zplane(b,a);
```

Similarly, we can plot (Figure 3.13) the pole-zero diagram of moving-average filter using the following MATLAB script (`example3_13b.m`) for $L = 8$:

```
b=[1, 0, 0, 0, 0, 0, 0, 0, -1];
a=[1, -1];
zplane(b,a);
```

As shown in Figure 3.13, the moving-average filter has a single pole at $z = 1$, which is canceled by the zero at $z = 1$. In this case, the pole-zero cancellation occurs in the system transfer function itself. The portion of the output $y(n)$ that is due to the poles of $X(z)$ is called the forced response of the system. The portion of the output that is due to the poles of $H(z)$ is called the natural response. If a system has all its poles within the unit circle, its natural response decays to zero as $n \to \infty$, and this is called the transient response. If the input to such a system is a sinusoidal signal, the corresponding forced response is called the sinusoidal steady-state response.

Example 3.14: Consider the recursive moving-window filter given in Equation (3.24). Taking the z-transform of both sides and rearranging terms, we obtain the transfer function

$$H(z) = \frac{\alpha}{1 - (1 - \alpha) z^{-1}}. \tag{3.46}$$

This is a simple first-order IIR filter with a zero at the origin and a pole at $z = 1 - \alpha$. A pole-zero plot of $H(z)$ given in Equation (3.46) is illustrated in Figure 3.14. Note that $\alpha = 1/L$ results in

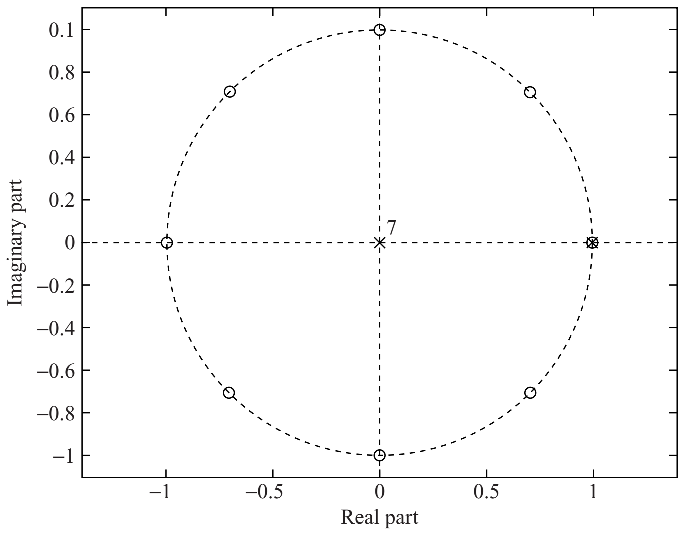


Figure 3.13 A pole-zero diagram generated by MATLAB

$1 - \alpha = (L-1)\big/L$, which is slightly less than 1. For a longer window, L is large; the value of $1-\alpha$ closes to 1, and the pole is closer to the unit circle.

An LTI system $H(z)$ is stable if and only if all the poles are inside the unit circle. That is,

$$|p_m| < 1, \quad \text{for all } m. \tag{3.47}$$

In this case, $\lim\limits_{n\to\infty}\{h(n)\} = 0$. A system is unstable if $H(z)$ has pole(s) outside the unit circle or multiple-order pole(s) on the unit circle. For example, if $H(z) = z/(z-1)^2$, then $h(n) = n$, which is unstable. A system is marginally stable, or oscillatory bounded, if $H(z)$ has first-order pole(s) that lie on the unit circle. For example, if $H(z) = z/(z+1)$, then $h(n) = (-1)^n$, $n \geq 0$.

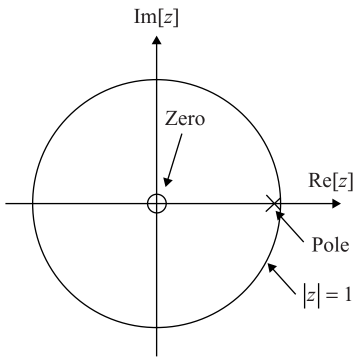


Figure 3.14 Pole-zero diagram of the recursive first-order IIR filter

Example 3.15: Given an LTI system with transfer function

$$H(z) = \frac{z}{z-a}.$$

There is a zero at the origin $z = 0$ and a pole at $z = a$. From Example 3.9, we have

$$h(n) = a^n, \ n \geq 0.$$

When $|a| > 1$, i.e., the pole at $z = a$ is outside the unit circle, we have

$$\lim_{n \to \infty} h(n) \to \infty,$$

that is an unstable system. However, when $|a| < 1$, i.e., the pole is inside the unit circle, we have

$$\lim_{n \to \infty} h(n) \to 0,$$

which is a stable system.

3.2.5 Frequency Responses

The frequency response of a digital system can be readily obtained from its transfer function $H(z)$ by setting $z = e^{j\omega}$ and obtain

$$H(\omega) = H(z)\,|_{z=e^{j\omega}} = \sum_{n=-\infty}^{\infty} h(n)z^{-n}\,|_{z=e^{j\omega}} = \sum_{n=-\infty}^{\infty} h(n)e^{-j\omega n}. \tag{3.48}$$

Thus, the frequency response $H(\omega)$ of the system is obtained by evaluating the transfer function on the unit circle $|z| = \left|e^{j\omega}\right| = 1$. As summarized in Table 3.1, the digital frequency is in the range of $-\pi \leq \omega \leq \pi$.

The characteristics of the system can be described using the frequency response. In general, $H(\omega)$ is a complex-valued function expressed in polar form as

$$H(\omega) = |H(\omega)|\, e^{j\phi(\omega)}, \tag{3.49}$$

where $|H(\omega)|$ is the magnitude (or amplitude) response and $\phi(\omega)$ is the phase response. The magnitude response $|H(\omega)|$ is an even function of ω, and the phase response $\phi(\omega)$ is an odd function. Thus, we only need to evaluate these functions in the frequency region $0 \leq \omega \leq \pi$. $|H(\omega)|^2$ is the squared-magnitude response, and $|H(\omega_0)|$ is the system gain at frequency ω_0.

Example 3.16: The moving-average filter expressed as

$$y(n) = \frac{1}{2}\left[x(n) + x(n-1)\right], \quad n \geq 0$$

is a simple first-order FIR filter. Taking the z-transform of both sides and rearranging the terms, we obtain

$$H(z) = \frac{1}{2}\left(1 + z^{-1}\right).$$

From Equation (3.48), we have

$$\begin{aligned} H(\omega) &= \frac{1}{2}\left(1 + e^{-j\omega}\right) = \frac{1}{2}\left(1 + \cos\omega - j\sin\omega\right), \\ |H(\omega)|^2 &= \{Re\,[H(\omega)]\}^2 + \{\mathrm{Im}\,[H(\omega)]\}^2 = \frac{1}{2}\left(1 + \cos\omega\right), \\ \phi(\omega) &= \tan^{-1}\left\{\frac{Im\,[H(\omega)]}{Re\,[H(\omega)]}\right\} = \tan^{-1}\left(\frac{-\sin\omega}{1 + \cos\omega}\right). \end{aligned}$$

From Appendix A,

$$\sin\omega = 2\sin\left(\frac{\omega}{2}\right)\cos\left(\frac{\omega}{2}\right) \; and \; \cos\omega = 2\cos^2\left(\frac{\omega}{2}\right) - 1.$$

Therefore, the phase response is

$$\phi(\omega) = \tan^{-1}\left[-\tan\left(\frac{\omega}{2}\right)\right] = -\frac{\omega}{2}.$$

For a given transfer function $H(z)$ expressed in Equation (3.42), the frequency response can be analyzed using the MATLAB function

```
[H,w]=freqz(b,a,N);
```

which returns the `N`-point frequency vector `w` and the complex frequency response vector `H`.

Example 3.17: Consider the IIR filter defined as

$$y(n) = x(n) + y(n-1) - 0.9y(n-2).$$

The transfer function is

$$H(z) = \frac{1}{1 - z^{-1} + 0.9z^{-2}}.$$

The MATLAB script (`example3_17a.m`) for analyzing the magnitude and phase responses of this IIR filter is listed as follows:

```
b=[1]; a=[1, -1, 0.9];
freqz(b,a);
```

Similarly, we can plot the magnitude and phase responses (shown in Figure 3.15) of the moving-average filter for $L = 8$ using the following script (`example3_17b.m`):

```
b=[1, 0, 0, 0, 0, 0, 0, 0, -1]; a=[1, -1];
freqz(b,a);
```

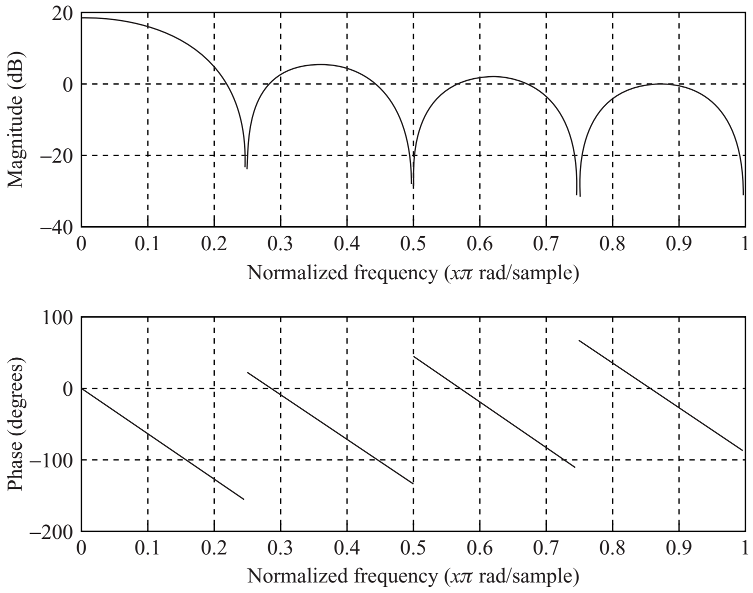


Figure 3.15 Magnitude (top) and phase responses of a moving-average filter, $L = 8$

A useful method of obtaining the brief frequency response of an LTI system is based on the geometric evaluation of its poles and zeros. For example, consider a second-order IIR filter expressed as

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \tag{3.50}$$

The roots of the characteristic equation

$$z^2 + a_1 z + a_2 = 0 \tag{3.51}$$

are the poles of the filter, which may be either real or complex. Complex poles can be expressed as

$$p_1 = re^{j\theta} \quad \text{and} \quad p_2 = re^{-j\theta}, \tag{3.52}$$

where r is radius of the pole and θ is the angle of the pole. Therefore, (3.51) becomes

$$\left(z - re^{j\theta}\right)\left(z - re^{-j\theta}\right) = z^2 - 2r\cos\theta + r^2 = 0. \tag{3.53}$$

Comparing this equation with (3.51), we have

$$r = \sqrt{a_2} \quad \text{and} \quad \theta = \cos^{-1}\left({}^{-a_1}\!/_{2r}\right). \tag{3.54}$$

The system with a pair of complex-conjugated poles as given in Equation (3.52) is illustrated in Figure 3.16. The filter behaves as a digital resonator for r close to unity. The digital resonator is a bandpass filter with its passband centered at the resonant frequency θ.

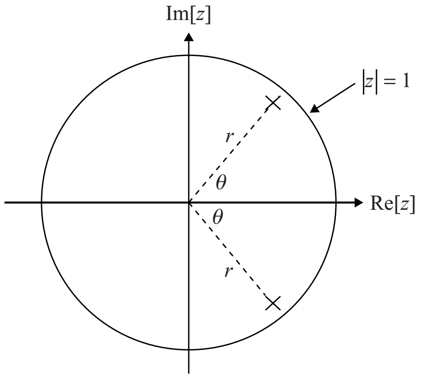


Figure 3.16 A second-order IIR filter with complex-conjugated poles

Similarly, we can obtain two zeros, z_1 and z_2, by evaluating $b_0 z^2 + b_1 z + b_2 = 0$. Thus, the transfer function defined in Equation (3.50) can be expressed as

$$H(z) = \frac{b_0 (z - z_1)(z - z_2)}{(z - p_1)(z - p_2)}. \tag{3.55}$$

In this case, the frequency response is given by

$$H(\omega) = \frac{b_0 \left(e^{j\omega} - z_1\right)\left(e^{j\omega} - z_2\right)}{\left(e^{j\omega} - p_1\right)\left(e^{j\omega} - p_2\right)}. \tag{3.56}$$

The magnitude response can be obtained by evaluating $|H(\omega)|$ as the point z moves in counterclockwise direction from $z = 0$ to $z = -1$ (π) on the unit circle. As the point z moves closer to the pole p_1, the magnitude response increases. The closer r is to the unity, the sharper the peak. On the other hand, as the point z moves closer to the zero z_1, the magnitude response decreases. The magnitude response exhibits a peak at the pole angle (or frequency), whereas the magnitude response falls to the valley at the angle of zero.

3.2.6 Discrete Fourier Transform

To perform frequency analysis of $x(n)$, we can convert the time-domain signal into frequency domain using the z-transform defined in Equation (3.27), and the frequency analysis can be performed by substituting $z = e^{j\omega}$ as shown in Equation (3.48). However, $X(\omega)$ is a continuous function of continuous frequency ω, and it also requires an infinite number of $x(n)$ samples for calculation. Therefore, it is difficult to compute $X(\omega)$ using digital hardware.

The discrete Fourier transform (DFT) of N-point signals $\{x(0), x(1), x(2), \ldots, x(N-1)\}$ can be obtained by sampling $X(\omega)$ on the unit circle at N equally-spaced samples at frequencies $\omega_k = 2\pi k/N$, $k = 0, 1, \ldots, N-1$. From Equation (3.48), we have

$$X(k) = X(\omega)|_{\omega = 2\pi k/N} = \sum_{n=0}^{N-1} x(n) e^{-j\left(\frac{2\pi k}{N}\right)n}, \quad k = 0,\ 1, \ldots, N-1, \tag{3.57}$$

where n is the time index, k is the frequency index, and $X(k)$ is the kth DFT coefficient. The DFT can be manipulated to obtain a very efficient computing algorithm called the fast Fourier transform (FFT). The derivation, implementation, and application of DFT and FFT will be further discussed in Chapter 6.

MATLAB provides the function `fft(x)` to compute the DFT of the signal vector `x`. The function `fft(x,N)` performs N-point FFT. If the length of `x` is less than `N`, then `x` is padded with zeros at the end. If the length of `x` is greater than `N`, function `fft(x,N)` truncates the sequence `x` and performs DFT of the first `N` samples only.

DFT generates N coefficients $X(k)$ for $k = 0, 1, \ldots N-1$. The frequency resolution of the N-point DFT is

$$\Delta = \frac{f_s}{N}. \tag{3.58}$$

The frequency f_k (in Hz) corresponding to the index k can be computed by

$$f_k = k\Delta = \frac{kf_s}{N}, \quad k = 0, 1, \ldots, N-1. \tag{3.59}$$

The Nyquist frequency ($f_s/2$) corresponds to the frequency index $k = N/2$. Since the magnitude $|X(k)|$ is an even function of k, we only need to display the spectrum for $0 \le k \le N/2$ (or $0 \le \omega_k \le \pi$).

Example 3.18: Similar to Example 3.1, we can generate 100 samples of sinewave with $A = 1$, $f = 1$ kHz, and sampling rate of 10 kHz. The magnitude response of signal can be computed and plotted (Figure 3.17) using the following MATLAB script (`example3_18.m`):

```
N=100; f = 1000; fs = 10000;
n=[0:N-1]; k=[0:N-1];
omega=2*pi*f/fs;
xn=sin(omega*n);
Xk=fft(xn,N);                % Perform DFT
magXk=20*log10(abs(Xk));     % Compute magnitude spectrum
plot(k, magXk);
axis([0, N/2, -inf, inf]);   % Plot from 0 to pi
xlabel('Frequency index, k');
ylabel('Magnitude in dB');
```

From Equation (3.58), frequency resolution is 100 Hz. The peak spectrum shown in Figure 3.17 is located at the frequency index $k = 10$, which corresponds to 1000 Hz as indicated by Equation (3.59).

3.3 Introduction to Random Variables

The signals encountered in practice are often random signals such as speech and music. In this section, we will briefly introduce the basic concepts of random variables.

3.3.1 Review of Random Variables

An experiment that has at least two possible outcomes is fundamental to the concept of probability. The set of all possible outcomes in any given experiment is called the sample space S. A random variable, x,

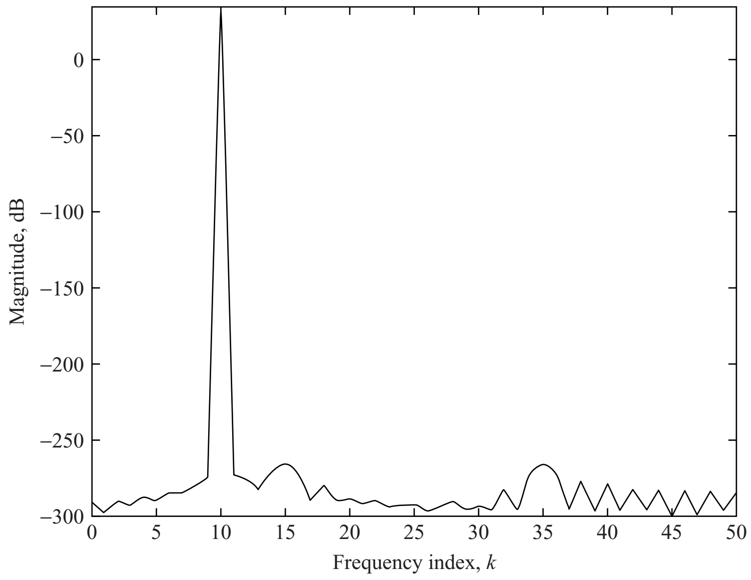


Figure 3.17 Magnitude spectrum of sinewave

is defined as a function that maps all elements from the sample space S into points on the real line. Thus, a random variable is a number whose value depends on the outcome of an experiment. For example, considering the outcomes of rolling of a fair die N times, we obtain a discrete random variable that can be any one of the discrete values from 1 through 6.

The cumulative probability distribution function of a random variable x is defined as

$$F(X) = P(x \leq X), \tag{3.60}$$

where X is a real number ranging from $-\infty$ to ∞, and $P(x \leq X)$ is the probability of $\{x \leq X\}$.

The probability density function of a random variable x is defined as

$$f(X) = \frac{\mathrm{d}F(X)}{\mathrm{d}X} \tag{3.61}$$

if the derivative exists. Two important properties of $f(X)$ are summarized as follows:

$$\int_{-\infty}^{\infty} f(X)\,\mathrm{d}X = 1 \tag{3.62}$$

$$P(X_1 < x \leq X_2) = F(X_2) - F(X_1) = \int_{X_1}^{X_2} f(X)\,\mathrm{d}X. \tag{3.63}$$

If x is a discrete random variable that can take on any one of the discrete values $X_i,\ i = 1,\ 2, \ldots$ as the result of an experiment, we define

$$p_i = P(x = X_i). \tag{3.64}$$

Example 3.19: Consider a random variable x that has a probability density function

$$f(X) = \begin{cases} 0, & x < X_1 \quad \text{or} \quad x > X_2 \\ a, & X_1 \le x \le X_2 \end{cases},$$

which is uniformly distributed between X_1 and X_2. The constant value a can be computed using Equation (3.62). That is,

$$\int_{-\infty}^{\infty} f(X)\,\mathrm{d}X = \int_{X_1}^{X_2} a \cdot \mathrm{d}X = a\,(X_2 - X_1) = 1.$$

Thus,

$$a = \frac{1}{X_2 - X_1}.$$

If a random variable x is equally likely to take on any value between the two limits X_1 and X_2, and cannot assume any value outside that range, it is uniformly distributed in the range $[X_1, X_2]$. As shown in Figure 3.18, a uniform density function is defined as

$$f\,(X) = \begin{cases} \frac{1}{X_2 - X_1}, & X_1 \le x \le X_2 \\ 0, & \text{otherwise} \end{cases}. \tag{3.65}$$

3.3.2 Operations of Random Variables

The statistics associated with random variables is often more meaningful from a physical viewpoint than the probability density function. The mean (expected value) of a random variable x is defined as

$$\begin{aligned} m_x = E\,[x] &= \int_{-\infty}^{\infty} X f(X)\,\mathrm{d}X, \quad \text{continuous-time case} \\ &= \sum_i X_i p_i, \qquad \text{discrete-time case,} \end{aligned} \tag{3.66}$$

where $E[\cdot]$ denotes the expectation operation (or ensemble averaging). The mean m_x defines the level about which the random process x fluctuates.

The expectation is a linear operation. Two useful properties of the expectation operation are $E\,[\alpha] = \alpha$ and $E\,[\alpha x] = \alpha E\,[x]$, where α is a constant. If $E[x] = 0, x$ is the zero-mean random variable. The

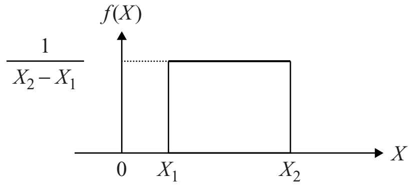


Figure 3.18 A uniform density function

MATLAB function `mean` calculates the mean value. For example, the statement `mx = mean(x)` computes the mean `mx` of the elements in the vector `x`.

Example 3.20: Considering the rolling of a fair die N times ($N \to \infty$), the probability of outcomes is listed as follows:

X_i	1	2	3	4	5	6
p_i	1/6	1/6	1/6	1/6	1/6	1/6

The mean of outcomes can be computed as

$$m_x = \sum_{i=1}^{6} p_i X_i = \frac{1}{6}(1+2+3+4+5+6) = 3.5.$$

The variance is a measure of the spread about the mean, and is defined as

$$\begin{aligned}\sigma_x^2 &= E\left[(x-m_x)^2\right] \\ &= \int_{-\infty}^{\infty} (X-m_x)^2 f(X)\,\mathrm{d}X, \quad \text{continuous-time case} \\ &= \sum_i p_i (X_i - m_x)^2, \quad \text{discrete-time case,}\end{aligned} \tag{3.67}$$

where $(x - m_x)$ is the deviation of x from the mean value m_x. The positive square root of variance is called the standard deviation σ_x. The MATLAB function `std` calculates standard deviation of the elements in the vector.

The variance defined in Equation (3.67) can be expressed as

$$\begin{aligned}\sigma_x^2 &= E\left[(x-m_x)^2\right] = E\left(x^2 - 2xm_x + m_x^2\right) = E\left(x^2\right) - 2m_x E(x) + m_x^2 \\ &= E\left(x^2\right) - m_x^2.\end{aligned} \tag{3.68}$$

We call $E\left(x^2\right)$ the mean-square value of x. Thus, the variance is the difference between the mean-square value and the square of the mean value.

If the mean value is equal to zero, then the variance is equal to the mean-square value. For a zero-mean random variable x, i.e., $m_x = 0$, we have

$$\sigma_x^2 = E\left(x^2\right) = P_x, \tag{3.69}$$

which is the power of x.

Consider the uniform density function defined in Equation (3.65). The mean of the function can be computed by

$$\begin{aligned}m_x &= E[x] = \int_{-\infty}^{\infty} Xf(X)\,\mathrm{d}X = \frac{1}{X_2 - X_1}\int_{X_1}^{X_2} X\mathrm{d}X \\ &= \frac{X_2 - X_1}{2}.\end{aligned} \tag{3.70}$$

The variance of the function is

$$\begin{aligned}\sigma_x^2 &= E\left(x^2\right) - m_x^2 = \int_{-\infty}^{\infty} X^2 f(X)\,\mathrm{d}X - m_x^2 \\ &= \frac{1}{X_2 - X_1}\int_{X_1}^{X_2} X^2\,\mathrm{d}X - m_x^2 = \frac{1}{X_2 - X_1}\cdot\frac{X_2^3 - X_1^3}{3} - m_x^2 \\ &= \frac{(X_2 - X_1)^2}{12}.\end{aligned} \tag{3.71}$$

In general, if x is a uniformly distributed random variable in the interval $(-\Delta, \Delta)$, we have

$$m_x = 0 \quad \text{and} \quad \sigma_x^2 = {}^{\Delta^2}\!/_3. \tag{3.72}$$

Example 3.21: The MATLAB function `rand` generates pseudo-random numbers uniformly distributed in the interval [0, 1]. From Equation (3.70), the mean of the generated pseudo-random numbers is 0.5. From Equation (3.71), the variance is 1/12.

To generate zero-mean random numbers, we subtract 0.5 from every generated random number. The numbers are now distributed in the interval $[-0.5, 0.5]$. To make these pseudo-random numbers with unit variance, i.e., $\sigma_x^2 = {}^{\Delta^2}\!/_3 = 1$, the generated numbers must be equally distributed in the interval $[-\sqrt{3}, \sqrt{3}]$. Therefore, we have to multiply $2\sqrt{3}$ to every generated number that was subtracted by 0.5.

The following MATLAB statement can be used to generate the uniformly distributed random numbers with mean 0 and variance 1:

```
xn = 2*sqrt(3)*(rand-0.5);
```

The waveform of zero-mean, unit-variance ($\sigma_x^2 = 1$) white noise generated by MATLAB code (`example3_21.m`) is shown in Figure 3.19.

A sinewave corrupted by white noise $v(n)$ can be expressed as

$$x(n) = A\sin(\omega n) + v(n). \tag{3.73}$$

When a signal $s(n)$ with power P_s is corrupted by a noise $v(n)$ with power P_v, the signal-to-noise ratio (SNR) in dB is defined as

$$\mathrm{SNR} = 10\log_{10}\left(\frac{P_\mathrm{s}}{P_\mathrm{v}}\right). \tag{3.74}$$

From Equation (3.69), the power of sinewave defined in Equation (3.6) can be computed as

$$P_\mathrm{s} = E\left[A^2\sin^2(\omega n)\right] = A^2/2. \tag{3.75}$$

Example 3.22: If we want to generate signal $x(n)$ expressed in Equation (3.73), where $v(n)$ is a zero-mean, unit-variance white noise. As shown in Equation (3.74), SNR is determined by the power of sinewave. As shown in Equation (3.75), when the sinewave amplitude $A = \sqrt{2}$, the power is equal to 1. From Equation (3.74), the SNR is 0 dB.

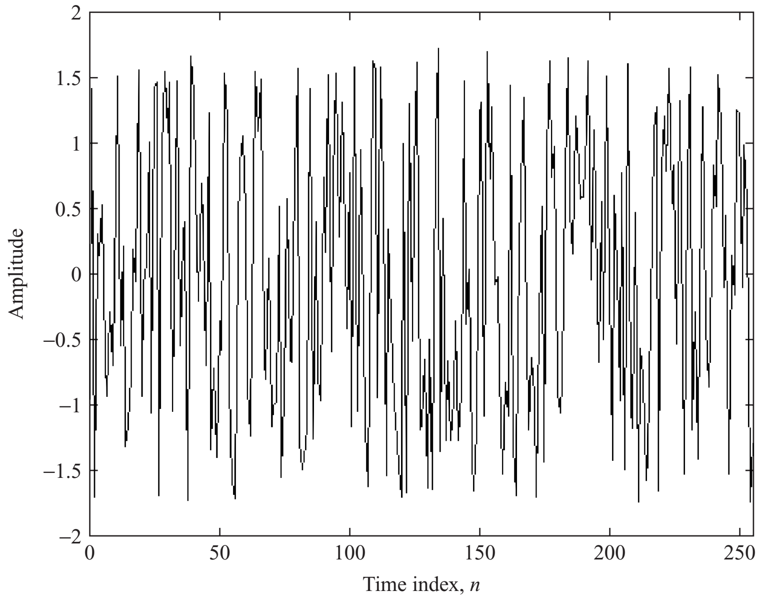


Figure 3.19 A zero-mean, unit-variance white noise

We can generate a sinewave corrupted by the zero-mean, unit-variance white noise with SNR $= 0$ dB using MATLAB script `example3_22.m`.

Example 3.23: We can compute the DFT of signal $x(n)$ to obtain $X(k)$. The magnitude spectrum in dB scale can be calculate as $20\log_{10}|X(k)|$ for $k = 0, 1, \ldots, N/2$. Using the signal $x(n)$ generated in Example 3.22, magnitude spectrum can be computed and displayed using the MATLAB code `example3_23.m`. The noisy spectrum is shown in Figure 3.20. Comparing this figure with Figure 3.17, we show that the power of white noise is uniformly distributed from 0 to π, while the power of sinewave is concentrated at its frequency 0.2π.

3.4 Fixed-Point Representations and Quantization Effects

The basic element in digital hardware is the binary device that contains one bit of information. A register (or memory unit) containing B bits of information is called a B-bit word. There are several different methods for representing numbers and carrying out arithmetic operations. In this book, we focus on widely used fixed-point implementations.

3.4.1 Fixed-Point Formats

The most commonly used fixed-point representation of a fractional number x is illustrated in Figure 3.21. The wordlength is $B(= M + 1)$ bits, i.e., M magnitude bits and one sign bit. The most significant bit (MSB) is the sign bit, which represents the sign of the number as follows:

$$b_0 = \begin{cases} 0, & x \geq 0 \quad \text{(positive number)} \\ 1, & x < 0 \quad \text{(negative number)} \end{cases}. \tag{3.76}$$

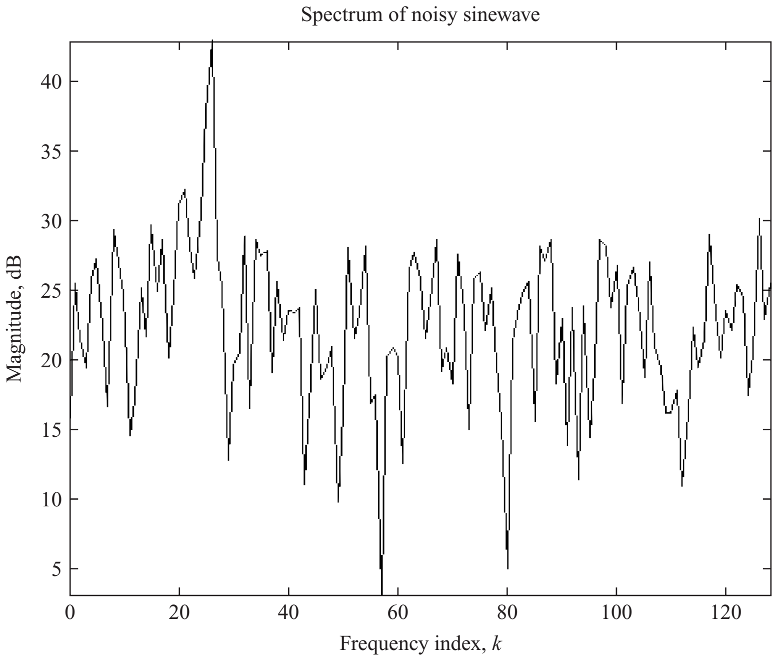


Figure 3.20 Spectrum of sinewave corrupted by white noise, SNR = 0 dB

The remaining M bits give the magnitude of the number. The rightmost bit b_M is called the least significant bit (LSB), which represents precision of the number.

As shown in Figure 3.21, the decimal value of a positive ($b_0 = 0$) binary fractional number x can be expressed as

$$(x)_{10} = b_1 \cdot 2^{-1} + b_2 \cdot 2^{-2} + \cdots + b_M \cdot 2^{-M}$$
$$= \sum_{m=1}^{M} b_m 2^{-m}. \tag{3.77}$$

Example 3.24: The largest (positive) 16-bit fractional number in binary format is $x = 0111\ 1111\ 1111\ 1111$b (the letter 'b' denotes that the number is in binary representation). The decimal value of this number can be obtained as

$$(x)_{10} = \sum_{m=1}^{15} 2^{-m} = 2^{-1} + 2^{-2} + \cdots + 2^{-15}$$
$$= 1 - 2^{-15} \approx 0.999969.$$

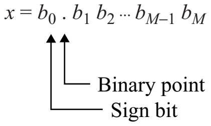


Figure 3.21 Fixed-point representation of binary fractional numbers

The smallest nonzero positive number is $x = 0000\ 0000\ 0000\ 0001\text{b}$. The decimal value of this number is

$$(x)_{10} = 2^{-15} = 0.000030518.$$

The negative numbers ($b_0 = 1$) can be represented using three different formats: the sign-magnitude, the 1's complement, and the 2's complement. Fixed-point DSP processors usually use the 2's complement format to represent negative numbers because it allows the processor to perform addition and subtraction using the same hardware. With the 2's complement format, a negative number is obtained by complementing all the bits of the positive binary number and then adding 1 to the LSB.

In general, the decimal value of a B-bit binary fractional number can be calculated as

$$(x)_{10} = -b_0 + \sum_{m=1}^{15} b_m 2^{-m}. \tag{3.78}$$

For example, the smallest (negative) 16-bit fractional number in binary format is $x = 1000\ 0000\ 0000\ 0000\text{b}$. From Equation (3.78), its decimal value is -1. Therefore, the range of fractional binary numbers is

$$-1 \leq x \leq \left(1 - 2^{-M}\right). \tag{3.79}$$

For a 16-bit fractional number x, the decimal value range is $-1 \leq x \leq 1 - 2^{-15}$.

Example 3.25: 4-bit binary numbers represent both integers and fractional numbers using the 2's complement format and their corresponding decimal values are listed in Table 3.2.

Example 3.26: If we want to initialize a 16-bit data x with the constant decimal value 0.625, we can use the binary form $x = 0101\ 0000\ 0000\ 0000\text{b}$, the hexidecimal form $x = 0\text{x}5000$, or the decimal integer $x = 2^{14} + 2^{12} = 20480$.

Table 3.2 4-bit binary numbers in 2's complement format and their corresponding decimal values

Binary numbers	Integers (*sxxx.*)	Fractions (*s.xxx*)
0000	0	0.000
0001	1	0.125
0010	2	0.250
0011	3	0.375
0100	4	0.500
0101	5	0.675
0110	6	0.750
0111	7	0.875
1000	−8	−1.000
1001	−7	−0.875
1010	−6	−0.750
1011	−5	−0.675
1100	−4	−0.500
1101	−3	−0.375
1110	−2	−0.250
1111	−1	−0.125

As shown in Figure 3.21, the easiest way to convert a normalized 16-bit fractional number into the integer that can be used by the C55x assembler is to move the binary point to the right by 15 bits (at the right of b_M). Since shifting the binary point 1 bit right is equivalent to multiply the fractional number by 2, this can be done by multiplying the decimal value by $2^{15} = 32768$. For example, $0.625 \times 32\,768 = 20\,480$.

It is important to note that we use an implied binary point to represent the binary fractional number. It will affect the accuracy (dynamic range and precision) of the number. The binary point is purely a programmer's convention and has no relationship with the hardware. The programmer needs to keep track of the binary point when manipulating fractional numbers in assembly language programming.

Different notations can be used to represent different fractional formats. Similar to Figure 3.21, a more general fractional format Qnm is illustrated in Figure 3.22 where $n + m = M = B - 1$. There are n bits at the left of binary point that represent integer portion, while m bits at the right represent fractional values. The most popular used fractional number representation shown in Figure 3.21 is called the Q0.15 format ($n = 0$ and $m = 15$), which is simply also called Q15 format since there are 15 fractional bits. Note that the Qnm format is represented in MATLAB as $[B\ m]$. For example, Q15 format is represented as [16 15].

Example 3.27: The decimal value of a 16-bit binary number $x = 0100\,1000\,0001\,1000$b depends on which Q format is used by the programmer. Some examples are given as follows:

$$\text{Q0.15}, x = 2^{-1} + 2^{-4} + 2^{-11} + 2^{-12} = 0.56323$$

$$\text{Q2.13}, x = 2^{1} + 2^{-2} + 2^{-9} + 2^{-10} = 2.25293$$

$$\text{Q5.10}, x = 2^{4} + 2^{1} + 2^{-6} + 2^{-7} = 18.02344$$

Example 3.28: As introduced in Chapter 2, the TMS320 assembly directives `.set` and `.equ` assign a value to a symbolic name. The directives `.word` and `.short` (or `.int`) initialize memory locations with particular data values represented in binary, hexidecimal, or integer format. Each data is treated as a 16-bit value and is separated by a comma. Some examples of the Q15 format data used for C55x are given as follows:

```
ONE          .set   32767        ; 1-2^-15 ≈ 0.999969 in integer
ONE_HALF     .set   0x4000       ; 0.5 in hexadecimal
ONE_EIGHTH   .equ   1000h        ; 1/8 in hexadecimal
MINUS_ONE    .equ   0xffff       ; -1.0 in hexadecimal
COEFF        .short 0ff00h       ; -2^-7 = -0.0078125 in hexadecimal
ARRAY        .word  2048,-2048   ; ARRAY[0.0625, -0.625]
```

As discussed in Chapter 1, fixed-point arithmetic is often used with DSP hardware for real-time processing because it offers fast operation and relatively economical implementation. Its drawbacks

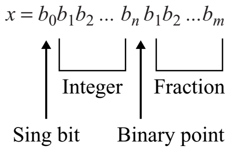


Figure 3.22 A general binary fractional numbers

include a small dynamic range and low resolution. These problems will be discussed in details in the following sections.

3.4.2 Quantization Errors

As discussed in Section 3.4.1, numbers are represented by a finite number of bits. The errors between the desired and actual values are called the finite-wordlength (finite-precision, or numerical) effects. In general, finite-precision effects can be broadly categorized into the following classes.

1. *Quantization errors*:
   (a) signal quantization
   (b) coefficient quantization

2. *Arithmetic errors*:
   (a) roundoff (or truncation)
   (b) overflow

The limit cycle oscillation is another phenomenon that may occur when implementing a feedback system such as an IIR filter with finite-precision arithmetic. The output of the system may continue to oscillate indefinitely while the input remains zero.

3.4.3 Signal Quantization

The analog-to-digital converter (ADC) converts an analog signal $x(t)$ into a digital signal $x(n)$. The input signal is first sampled to obtain the discrete-time signal $x(nT)$ with infinite precision. Each $x(nT)$ value is then encoded using B-bit wordlength to obtain the digital signal $x(n)$. We assume that the signal $x(n)$ is interpreted as the Q15 fractional number shown in Figure 3.21 such that $-1 \leq x(n) < 1$. Thus, the dynamic range of fractional numbers is 2. Since the quantizer employs B bits, the number of quantization levels available is 2^B. The spacing between two successive quantization levels is

$$\Delta = \frac{2}{2^B} = 2^{-B+1} = 2^{-M}, \tag{3.80}$$

which is called the quantization step (interval, width, or resolution). For example, the output of a 4-bit converter with quantization interval $\Delta = 2^{-3} = 0.125$ is summarized in Table 3.2.

As discussed in Chapter 1, we use rounding (instead of truncating) for quantization in this book. The input value $x(nT)$ is rounded to the nearest level as illustrated in Figure 3.23 for a 3-bit ADC. We assume there is a line exactly between two quantization levels. The signal value above this line will be assigned to the higher quantization level, while the signal value below this line is assigned to the lower level. For example, the discrete-time signal $x(T)$ in Figure 3.23 is rounded to 010b since the real value is below the middle line between 010b and 011b, while $x(2T)$ is rounded to 011b since the value is above the middle line.

The quantization error (or noise) $e(n)$ is the difference between the discrete-time signal $x(nT)$ and the quantized digital signal $x(n)$, and is be expressed as

$$e(n) = x(n) - x(nT). \tag{3.81}$$

Figure 3.23 clearly shows that

$$|e(n)| \leq \frac{\Delta}{2}. \tag{3.82}$$

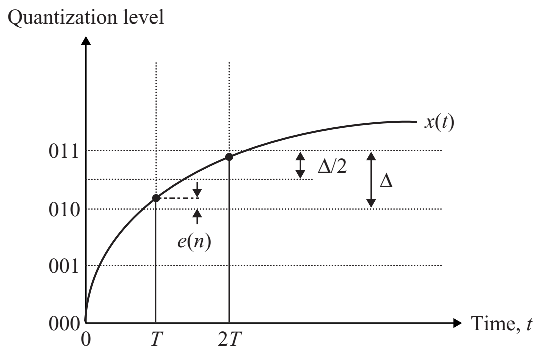


Figure 3.23 Quantization process related to a 3-bit ADC

Thus, the quantization noise generated by an ADC depends on the quantization interval. The presence of more bits results in a smaller quantization step, a lower quantization noise.

From Equation (3.81), we can express the ADC output as the sum of the quantizer input $x(nT)$ and the error $e(n)$. That is,

$$x(n) = Q\left[x(nT)\right] = x(nT) + e(n), \tag{3.83}$$

where $Q[\cdot]$ denotes the quantization operation. Therefore, the nonlinear operation of the quantizer is modeled as a linear process that introduces an additive noise $e(n)$ to the digital signal $x(n)$.

For an arbitrary signal with fine quantization (B is large), the quantization error $e(n)$ is assumed to be uncorrelated with $x(n)$, and is a random noise that is uniformly distributed in the interval $[-{}^{\Delta}\!/_2,\ {}^{\Delta}\!/_2]$. From Equation (3.70), we have

$$E[e(n)] = \frac{-\Delta/2 + \Delta/2}{2} = 0. \tag{3.84}$$

Thus, the quantization noise $e(n)$ has zero mean. From Equation (3.72), the variance

$$\sigma_e^2 = \frac{\Delta^2}{12} = \frac{2^{-2B}}{3}. \tag{3.85}$$

Therefore, the larger wordlength results in smaller input quantization error.

The SQNR can be expressed as

$$\text{SQNR} = \frac{\sigma_x^2}{\sigma_e^2} = 3 \cdot 2^{2B}\sigma_x^2, \tag{3.86}$$

where σ_x^2 denotes the variance of the signal, $x(n)$. Usually, the SQNR is expressed in dB as

$$\begin{aligned}\text{SQNR} &= 10\log_{10}\left(\frac{\sigma_x^2}{\sigma_e^2}\right) = 10\log_{10}\left(3 \cdot 2^{2B}\sigma_x^2\right)\\ &= 10\log_{10}3 + 20B\log_{10}2 + 10\log_{10}\sigma_x^2\\ &= 4.77 + 6.02B + 10\log_{10}\sigma_x^2.\end{aligned} \tag{3.87}$$

This equation indicates that for each additional bit used in the ADC, the converter provides about 6-dB gain. When using a 16-bit ADC ($B = 16$), the maximum SQNR is about 98.1 dB if the input signal is a sinewave. This is because the maximum sinewave having amplitude 1.0 in decimal makes $10\log_{10}(\sigma_x^2) = 10\log_{10}(1/2) = -3$, and Equation (3.87) becomes $4.77 + 6.02 \times 16 - 3.0 = 98.09$. Another important fact about Equation (3.87) is that the SQNR is proportional to σ_x^2. Therefore, we want to keep the power of signal as large as possible. This is an important consideration when we discuss scaling issues in Section 3.5.

Example 3.29: Effects of signal quantization may be subjectively evaluated by observing and listening to the quantized speech. The speech file `timit1.asc` was digitized with $f_s = 8$ kHz and $B = 16$. This speech file can be viewed and played using the MATLAB script (`example3_29.m`):

```
load timit1.asc;
plot(timit1);
soundsc(timit1, 8000, 16);
```

where the MATLAB function `soundsc` autoscales and plays the vector as sound. We can simulate the quantization of data with 8-bit wordlength by

```
qx = round(timit1/256);
```

where the function (`round`) rounds the real number to the nearest integer. We then evaluate the quantization effects by

```
plot(qx);
soundsc(qx, 8000, 16);
```

By comparing the graph and sound of `timit1` and `qx`, the signal quantization effects may be understood.

3.4.4 Coefficient Quantization

The filter coefficients, b_l and a_m, of the digital filter determined by a filter design package such as MATLAB are usually represented using the floating-point format. When implementing a digital filter, the filter coefficients have to be quantized for a given fixed-point processor. Therefore, the performance of the fixed-point digital filter will be different from its design specification.

The coefficient quantization effects become more significant when tighter specifications are used, especially for IIR filters. Coefficient quantization can cause serious problems if the poles of designed IIR filters are too close to the unit circle. This is because those poles may move outside the unit circle due to coefficient quantization, resulting in an unstable implementation. Such undesirable effects are far more pronounced in high-order systems.

The coefficient quantization is also affected by the structures used for the implementation of digital filters. For example, the direct-form implementation of IIR filters is more sensitive to coefficient quantization than the cascade structure consisting of sections of first- or second-order IIR filters.

3.4.5 Roundoff Noise

As shown in Equation (3.10), we may need to compute the product $y(n) = \alpha x(n)$ in a DSP system. Assuming the wordlength associated with α and $x(n)$ is B bits, the multiplication yields $2B$-bit product $y(n)$. In most applications, this product may have to be stored in memory or output as a B-bit word. The $2B$-bit product can be either truncated or rounded to B bits. Since truncation causes an undesired bias effect, we should restrict our attention to the rounding.

Example 3.30: In C programming, rounding a real number to an integer number can be implemented by adding 0.5 to the real number and then truncating the fractional part. The following C statement

```
y = (short)(x + 0.5);
```

rounds the real number `x` to the nearest integer `y`. As shown in Example 3.29, MATLAB provides the function `round` for rounding a real number.

In TMS320C55x implementation, the CPU rounds the operands enclosed by the `rnd()` expression qualifier as

```
mov rnd(HI(AC0)),*AR1
```

This instruction will round the content of the high portion of `AC0(31:16)` and the rounded 16-bit value is stored in the memory location pointed at by AR1. Another key word `R` (or `r`) also performs rounding operation on the operands. The following instruction

```
mpyr   AC0,AC1
```

multiplies and stores the rounded product in the upper portion of the accumulator `AC1(31:16)` and clears the lower portion of the accumulator `AC1(15:0)`.

The process of rounding a $2B$-bit product to B bits is similar to that of quantizing discrete-time signal using a B-bit quantizer. Similar to Equation (3.83), the nonlinear roundoff operation can be modeled as the linear process expressed as

$$y(n) = Q\left[\alpha x(n)\right] = \alpha x(n) + e(n), \tag{3.88}$$

where $\alpha x(n)$ is the $2B$-bit product and $e(n)$ is the roundoff noise due to rounding $2B$-bit product to B-bit product. The roundoff noise is a uniformly distributed random process defined in Equation (3.82). Thus, it has a zero mean and its power is defined in Equation (3.85).

It is important to note that most commercially available fixed-point DSP processors, such as the TMS320C55x, have double-precision accumulator(s). As long as the program is carefully written, it is possible to ensure that rounding occurs only at the final stage of calculation. For example, consider the computation of FIR filter output given in Equation (3.15). We can keep the sum of all temporary products, $b_l x(n-l)$, in the double-precision accumulator. Rounding is performed only when the final sum is saved to memory with B-bit wordlength.

3.4.6 Fixed-Point Toolbox

The MATLAB *Fixed-Point Toolbox* provides fixed-point data types and arithmetic for enabling fixed-point algorithm development. This toolbox has the following features:

- defining fixed-point data types, scaling, rounding, and overflow methods in the MATLAB workspace;
- bit-true real and complex simulation;

- fixed-point arithmetic;
- relational, logical, and bitwise operators; and
- conversions between binary, hex, double, and built-in integers.

This toolbox provides the function `quantizer` to construct a quantizer object. For example,

```
q = quantizer('PropertyName1',PropertyValue1,... )
```

creates a quantizer object `q` that uses property name/property value pairs that are summarized in Table 3.3. We also can use the following syntax

```
q = quantizer
```

to create a quantizer object `q` with properties set to the following default values:

```
mode = 'fixed';
roundmode = 'floor';
overflowmode = 'saturate';
format = [16 15];
```

Note that `[16 15]` is equivalent to Q15 format.

After we have constructed a quantizer object, we can apply it to data using the `quantize` function with the following syntax:

```
y = quantize(q, x)
```

The command `y = quantize(q, x)` uses the quantizer object `q` to quantize `x`. When `x` is a numeric array, each element of `x` is quantized.

Table 3.3 List of quantizer property name/property value pairs

Property name	Property value	Description
`mode`	`'double'`	Double-precision mode
	`'float'`	Custom-precision floating-point mode
	`'fixed'`	Signed fixed-point mode
	`'single'`	Single-precision mode
	`'ufixed'`	Unsigned fixed-point mode
`roundmode`	`'ceil'`	Round toward negative infinity
	`'convergent'`	Convergent rounding
	`'fix'`	Round toward zero
	`'floor'`	Round toward positive infinity
	`'round'`	Round toward nearest
`overflowmode`	`'saturate'`	Saturate on overflow
	`'wrap'`	Wrap on overflow
`format`	`[B m]`	Format for `fixed` or `ufixed` mode, `B` is wordlength, `m` is number of fractional bits

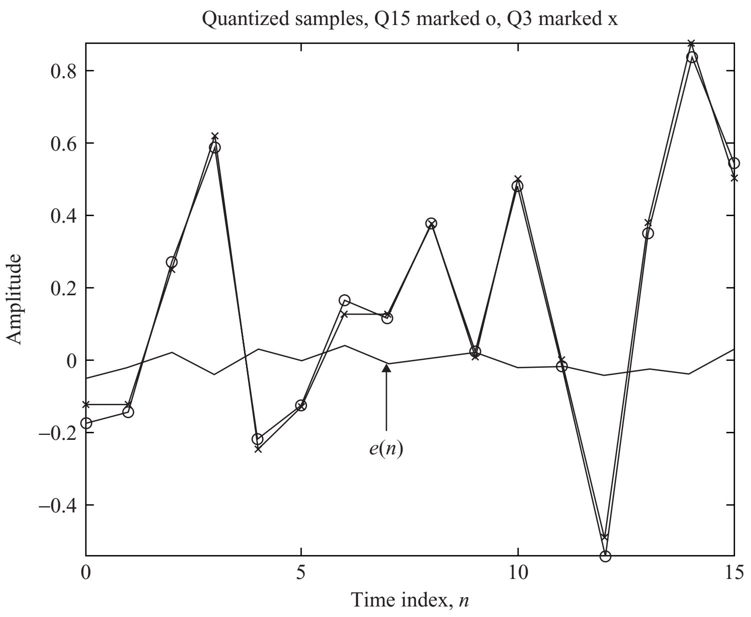


Figure 3.24 Quantization using Q15 and Q3 formats and the difference $e(n)$

Example 3.31: Similar to Example 3.21, we generate a zero-mean white noise using MATLAB function `rand`, which uses double-precision, floating-point format. We then construct two quantizer objects and quantize the white noise to Q15 and Q3 (4-bit) representations. We plot the quantized noise in Q15 and Q3 formats and the difference between these two is shown in Figure 3.24 using the following MATLAB script (`example3_31.m`):

```
N=16;
n=[0:N-1];
xn = sqrt(3)*(rand(1,N)-0.5); % Generate zero-mean white noise
q15 = quantizer('fixed', 'convergent', 'wrap', [16 15]); % Q15
q3 = quantizer('fixed', 'convergent', 'wrap', [4 3]); % Q3
y15 = quantize(q15,xn);        % Quantization using Q15 format
y3 = quantize(q3,xn);          % Quantization using Q3 format
en = y15-y3,                   % Difference between Q15 and Q3
plot(n,y15,'-o',n,y3,'-x',n,en);
```

MATLAB *Fixed-Point Toolbox* also provides several radix conversion functions which are summarized in Table 3.4. For example,

```
y = num2int(q,x)
```

uses `q.format` to convert a number `x` to an integer `y`.

Example 3.32: For testing some programs using fixed-point C programs with CCS and DSK, we may need to generate input data files for simulations. As shown in Example 3.31, we use MATLAB

Table 3.4 List of radix conversion functions using a quantizer object

Function	Description
`bin2num`	Convert a 2's complement binary string to a number
`hex2num`	Convert hexadecimal string to a number
`num2bin`	Convert a number to a binary string
`num2hex`	Convert a number to its hexadecimal equivalent
`num2int`	Convert a number to a signed integer

to generate signal and construct a quantizer object. In order to save the Q15 data in integer format, we use the function `num2int` in the following MATLAB script (`example3_32.m`):

```
N=16; n=[0:N-1];
xn = sqrt(3)*(rand(1,N)-0.5); % Generate zero-mean white noise
q15 = quantizer('fixed', 'convergent', 'wrap', [16 15]); % Q15
Q15int = num2int(q15,xn);
```

3.5 Overflow and Solutions

Assuming that the signals and filter coefficients have been properly normalized in the range of -1 to 1 for fixed-point arithmetic, the sum of two B-bit numbers may fall outside the range of -1 to 1. The term overflow is a condition in which the result of an arithmetic operation exceeds the capacity of the register used to hold that result. When using a fixed-point processor, the range of numbers must be carefully examined and adjusted in order to avoid overflow. This may be achieved by using different Q$n.m$ formats with desired dynamic ranges.

Example 3.33: Assume that a 4-bit fixed-point hardware uses the fractional 2's complement format (see Table 3.2). If $x_1 = 0.875$ (0111b) and $x_2 = 0.125$ (0001b), the binary sum of $x_1 + x_2$ is 1000b. The decimal value of this signed binary number is -1, not the correct answer $+1$. That is, when the result exceeds the dynamic range of the register, overflow occurs and unacceptable error is produced.

Similarly, if $x_3 = -0.5$ (1100b) and $x_4 = 0.625$(0101b). $x_3 - x_4 =$ 0110b, which is $+0.875$, and not the correct answer -1.125. Therefore, subtraction may also result in underflow.

For the FIR filter defined in Equation (3.15), this overflow will result in the severe distortion of the output $y(n)$. For the IIR filter defined in Equation (3.23), the overflow effect is much more serious because the errors are fed back. The problem of overflow may be eliminated using saturation arithmetic and proper scaling (or constraining) signals at each node within the filter to maintain the magnitude of the signal.

3.5.1 Saturation Arithmetic

Most commercially available DSP processors have mechanisms that protect against overflow and automatically indicate the overflow if it occurs. Saturation arithmetic prevents overflow by keeping the result at a maximum value. Saturation logic is illustrated in Figure 3.25 and can be expressed as

$$y = \begin{cases} 1 - 2^{-M}, & x \geq 1 - 2^{-M} \\ x, & -1 \leq x < 1 \\ -1, & x < -1 \end{cases}, \tag{3.89}$$

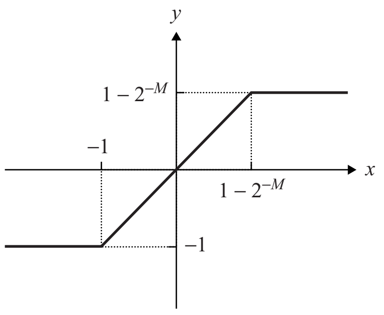


Figure 3.25 Characteristics of saturation arithmetic

where x is the original addition result and y is the saturated adder output. If the adder is under saturation mode, the undesired overflow can be avoided since the 32-bit accumulator fills to its maximum (or minimum) value, but does not roll over. Similar to Example 3.31, when 4-bit hardware with saturation arithmetic is used, the addition result of $x_1 + x_2$ is 0111b, or 0.875 in decimal value. Compared with the correct answer 1, there is an error of 0.125. This result is much better than the hardware without saturation arithmetic.

Saturation arithmetic has a similar effect of 'clipping' the desired waveform. This is a nonlinear operation that will add undesired nonlinear components into the signal. Therefore, saturation arithmetic can only be used to guarantee that overflow will not occur. It should not be the only solution for solving overflow problems.

3.5.2 Overflow Handling

As mentioned earlier, the C55x supports the saturation logic to prevent overflow. The logic is enabled when the overflow mode bit (SATD) in status register ST1 is set (SATD = 1). When this mode is set, the accumulators are loaded with either the largest positive 32-bit value (0x00 7FFF FFFF) or the smallest negative 32-bit value (0xFF 8000 0000) if the result overflows. The C55x overflow mode bit can be set with the instruction

```
bset SATD
```

and reset (disabled) with the instruction

```
bclr SATD
```

The TMS320C55x provides overflow flags that indicate whether or not an arithmetic operation has overflowed. The overflow flag ACOVn, ($n = 0$, 1, 2, or 3) is set to 1 when an overflow occurs in the corresponding accumulator ACn. This flag will remain set until a reset is performed or when a status bit clear instruction is implemented. If a conditional instruction (such as a branch, return, call, or conditional execution) that tests overflow status is executed, the overflow flag will be cleared.

3.5.3 Scaling of Signals

The most effective technique in preventing overflow is by scaling down the signal. For example, consider the simple FIR filter illustrated in Figure 3.26 without the scaling factor β (or $\beta = 1$). Let $x(n) = 0.8$

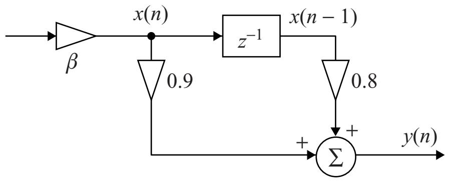


Figure 3.26 Block diagram of simple FIR filters with scaling factor β

and $x(n-1) = 0.6$, the filter output $y(n) = 1.2$. When this filter is implemented on a fixed-point DSP processor using Q15 format without saturation arithmetic, undesired overflow occurs. As illustrated in Figure 3.26, the scaling factor $\beta < 1$ can be used to scale down the input signal. For example, when $\beta = 0.5$, we have $x(n) = 0.4$ and $x(n-1) = 0.3$, and the result $y(n) = 0.6$ without overflow.

If the signal $x(n)$ is scaled by β, the corresponding signal variance changes to $\beta^2\sigma_x^2$. Thus, the SQNR in dB given in Equation (3.87) changes to

$$\begin{aligned}\text{SQNR} &= 10\log_{10}\left(\frac{\beta^2\sigma_x^2}{\sigma_e^2}\right)\\ &= 4.77 + 6.02B + 10\log_{10}\sigma_x^2 + 20\log_{10}\beta. \qquad (3.90)\end{aligned}$$

Since we perform fractional arithmetic, $\beta < 1$ is used to scale down the input signal. The last term $20\log_{10}\beta$ has negative value. Thus, scaling down the signal reduces the SQNR. For example, when $\beta = 0.5$, $20\log_{10}\beta = -6.02$ dB, thus reducing the SQNR of the input signal by about 6 dB. This is equivalent to losing 1 bit in representing the signal.

3.5.4 Guard Bits

The TMS320C55x provides four 40-bit accumulators as introduced in Chapter 2. Each accumulator is split into three parts as illustrated in Figure 3.27. These guard bits are used as a head-margin for preventing overflow in iterative computations such as the FIR filtering defined in Equation (3.15).

Because of the potential overflow in a fixed-point implementation, engineers need to be concerned with the dynamic range of numbers. This usually demands greater coding and testing efforts. In general, the optimum solution is combining of scaling factors, guard bits, and saturation arithmetic. The scaling factors (smaller than 1) are set as large as possible so that there maybe only some occasional overflows which can be avoided by using guard bits and saturation arithmetic.

3.6 Experiments and Program Examples

In this section, the first half of the experiments is used to demonstrate quantization effects, overflow and saturation arithmetic, and to determine the proper fixed-point representations. The rest of experiments emphasize on the hands-on DSP programming and implementation using the C5510 DSK.

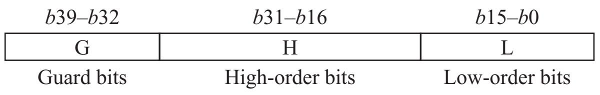


Figure 3.27 Configuration of the TMS320C55x accumulators

Table 3.5 C program for quantizing a sinusoid, `quantSine.c`

```
#define BUF_SIZE 40
const short sineTable[BUF_SIZE]= {
 0x0000,0x01E0,0x03C0,0x05A0,0x0740,0x08C0,0x0A00,0x0B20,
 0x0BE0,0x0C40,0x0C60,0x0C40,0x0BE0,0x0B20,0x0A00,0x08C0,
 0x0740,0x05A0,0x03C0,0x01E0,0x0000,0xFE20,0xFC40,0xFA60,
 0xF8C0,0xF740,0xF600,0xF4E0,0xF420,0xF3C0,0xF3A0,0xF3C0,
 0xF420,0xF4E0,0xF600,0xF740,0xF8C0,0xFA60,0xFC40,0x0000};

short out16[BUF_SIZE];     /* 16 bits output sample buffer */
short out12[BUF_SIZE];     /* 12 bits output sample buffer */
short out8[BUF_SIZE];      /* 8 bits output sample buffer  */
short out6[BUF_SIZE];      /* 6 bits output sample buffer  */

void main()
{
   short i;

   for (i = 0; i < BUF_SIZE; i++)
   {
      out16[i] = sineTable[i];          /* 16-bit data     */
      out12[i] = sineTable[i]&0xfff0;   /* Mask off 4-bit  */
      out8[i] = sineTable[i]&0xff00;    /* Mask off 8-bit  */
      out6[i] = sineTable[i]&0xfc00;    /* Mask off 10-bit */
   }
}
```

3.6.1 Quantization of Sinusoidal Signals

The C program listed in Table 3.5 simulates an ADC with different wordlengths. Instead of shifting off the bits, we mask out the least significant 4, 8, or 10 bits of each sample, resulting in the 12, 8, or 6 bits of data samples that have the comparable amplitude to the original 16-bit data. Table 3.6 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Load the project `quantSine.pjt`, rebuild, and load the program to the DSK or C55x simulator.

2. Use the CCS graphic display to plot four output buffers: `out16`, `out12`, `out8`, and `out6`, as shown in Figure 3.28. Compare and describe the graphical results of each output waveforms represented by different wordlengths.

3. Find the mean and variance of quantization noise for the 12-, 8-, and 6-bit ADCs.

Table 3.6 File listing for experiment `exp3.6.1_quantSine`

Files	Description
`quantSine.c`	C function for implementing quantization
`quantSine.pjt`	DSP project file
`quantSine.cmd`	DSP linker command file

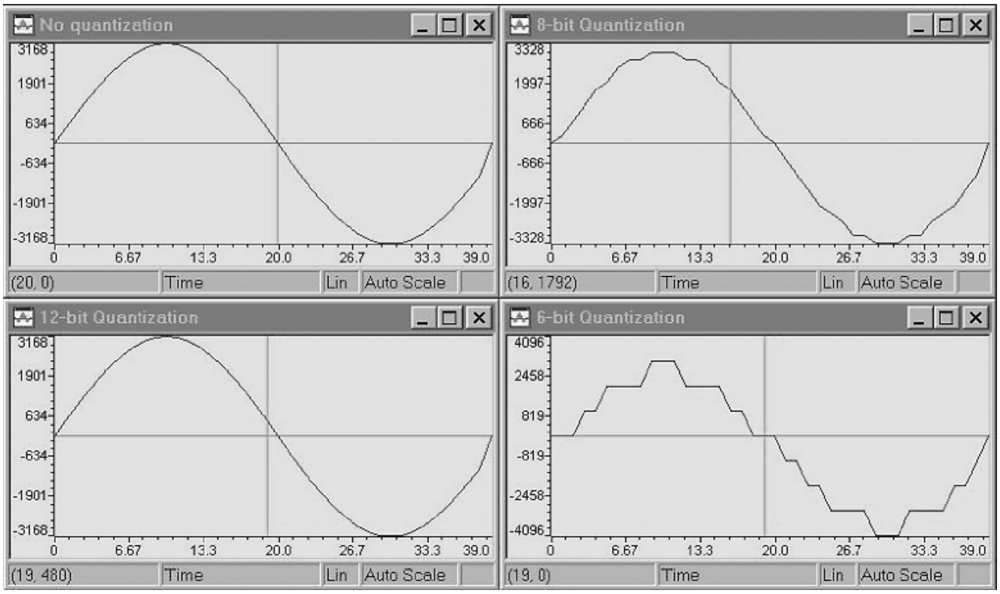


Figure 3.28 Quantizing 16-bit data (top-left) into 12-bit (bottom-left), 8-bit (top-right), and 6-bit (bottom-right)

3.6.2 Quantization of Audio Signals

To evaluate the quantization effects of audio signals, we use the DSK for real-time experiment. The program that emulates the quantizer is listed in Table 3.7. During the real-time audio playback, the masked variable `quant` will be changed to emulate the quantization effects.

Table 3.8 lists the files used for this experiment. This experiment uses the program given in Section 1.6.6, which is modified from the C5510 DSK audio example. The program reads audio samples, applies quantizer to the samples, and plays the quantized samples using DSK headphone output.

Procedures of the experiment are listed as follows:

1. Load the project `quantAudio.pjt`, rebuild, and load the program to DSK.
2. Use an audio source (CD player or radio) as the audio input to the DSK. The included wave files can be used with Windows media player as audio sources.
3. Listen to the quantization effects of representing audio samples with different wordlengths using a headphone (or loudspeaker) connected to the headphone output of the DSK.
4. Compare and describe the quantization effects of speech and music samples.

Table 3.7 Listing of audio signal quantization program, `quantAudio.c`

```
short quantAudio(short indata, short quant)
{
    return(indata&quant); /* Quantization by masking the data sample */
}
```

Table 3.8 File listing for experiment `exp3.6.2_quantAudio`

Files	Description
`quantAudioTest.c`	C function for testing experiment
`quantAudio.c`	C function for audio quantization
`quantAudio.pjt`	DSP project file
`quantAudiocfg.cmd`	DSP linker command file
`quantAudio.cdb`	DSP BIOS configuration file
`desertSun.wav`	Wave file
`fools8k.wav`	Wave file

3.6.3 Quantization of Coefficients

Since filter design and implementation will be discussed in Chapters 4 and 5, we will only briefly describe the fourth-order IIR filter used in this experiment. Table 3.9 lists an assembly program that implements a fourth-order IIR filter. This lowpass filter is designed as $f_c/f_s = 0.225$, where f_c is the cutoff frequency. The signal components with frequencies below the cutoff frequency will pass through the lowpass filter,

Table 3.9 List of IIR filtering program, `IIR4.asm`

```
   .def _IIR4
   .def _initIIR4
;
;  Original coefficients of 4th-order IIR lowpass filter
;     with fc/fs = 0.225
;
;  short b[5]={ 0.0072, 0.00287, 0.0431, 0.0287, 0.0072};
;  short a[5]={ 1.0000, -2.16860,2.0097,-0.8766, 0.1505};
;
   .data    ; Q13 formatted coefficients
iirCoeff
    .word 0x003B, 0x00EB         ; b0, b1,
    .word 0x0161, 0x00EB, 0x003B ; b2, b3, b4
    .word 0x4564, -0x404F        ; -a1, -a2,
    .word 0x1C0D, -0x04D1        ; -a3, -a4

    .bss  x,5         ; x buffer
    .bss  y,4         ; y buffer
    .bss  coeff,9     ; Filter coefficients

   .text
;
;    4th-order IIR filter initialization routine
;    Entry T0 = mask for filter coefficients
;
_initIIR4
   amov #x,XAR0       ; Zero x buffer
   rpt  #4
   mov  #0,*AR0+
   amov #y,XAR0       ; Zero y buffer
   rpt  #3
   mov  #0,*AR0+
   mov  #8,BRC0       ; Mask off bits of coefficients
```

Table 3.9 (*continued*)

```
   amov #iirCoeff,XAR0
   amov #coeff,XAR1
   rptb maskCoefLoop-1
   mov  *AR0+,AC0
   and  T0,AC0
   mov  AC0,*AR1+
maskCoefLoop
   ret
;
;    4-th-order IIR filtering
;    Entry T0 = sample
;    Exit T0 = filtered sample
;
_IIR4
   bset SATD
   bset SXM
   amov #x,XAR0
   amov #y,XAR1
   amov #coeff,XCDP
   bset FRCT
||  mov  T0,*AR0            ; x[0] = indata
;
;  Perform IIR filtering
;
   mpym *AR0+,*CDP+,AC0    ; AC0=x[0]*bn[0]
||  rpt  #3                ; i=1,2,3,4
   macm *AR0+,*CDP+,AC0    ; AC0+=x[i]*bn[i]
   rpt  #3                 ; i=0,1,2,3
   macm *AR1+,*CDP+,AC0    ; AC0+=y[i]*an[i]
   amov #y+2,XAR0
   amov #y+3,XAR1
   sfts AC0,#2             ; Scale to Q15 format
||  rpt  #2
   mov  *AR0-,*AR1-        ; Update y[]
   mov  hi(AC0),*AR1
||  mov  hi(AC0),T0         ; Return y[0] in T0
   amov #x+3,XAR0
   amov #x+4,XAR1
   bclr FRCT
||  rpt  #3
   mov  *AR0-,*AR1-        ; Update x[]
   bclr SXM
   bclr SATD
   ret
   .end
```

while the higher frequency components will be attenuated. The assembly routine, `_initIIR4`, initializes the memory locations of `x` and `y` buffers to zero. In our experiment, the coefficients are masked during the initialization to 16, 12, 8, and 4 bits. The IIR filter assembly routine, `_IIR4`, performs the filtering operation to the input data samples. The initialization is performed only once, while the IIR routine will be called to perform the filter operation for every incoming sample. The coefficient data pointer (CDP) is used to address the filter coefficients. The auxiliary registers, AR0 and AR1, are pointing to the `x` and `y`

Table 3.10 List of files used for experiment `exp3.6.3_quantFiltCoef`

Files	Description
`quantFiltCoefTest.c`	C function for testing filter quantization
`quantFiltCoef.asm`	Assembly IIR function for quantized filter
`quantFiltCoef.pjt`	DSP project file
`quantFiltCoefcfg.cmd`	DSP linker command file
`quantFiltCoef.cdb`	DSP BIOS configuration file
`desertSun.wav`	Wave file
`fools8k.wav`	Wave file

data buffers, respectively. After each sample is processed, both the `x` and `y` buffers are updated by shifting the data in the buffers, which will be further discused in Chapter 4.

The files used for this experiment are listed in Table 3.10. The experiment program reads audio samples, applies an IIR filter to the samples, and plays the filter results via DSK headphone jack.

Procedures of the experiment are listed as follows:

1. Load the project `quantFiltCoef.pjt`, rebuild, and load the program to the DSK.
2. Connect an audio source to the line-in of the DSK and connect a headphone to the headphone output of the DSK and play the audio signals. The included wave files can be used as audio sources for Windows media player.
3. Listen to the audio output and compare the left channel with the right channel. In this experiment, the left-channel audio input samples are sent directly to the output while the right-channel input samples are filtered by the IIR filter. Describe the quantization effects due to the use of limited wordlength for representing filter coefficients.

3.6.4 Overflow and Saturation Arithmetic

As discussed in Section 3.5, overflow may occur when DSP processors perform fixed-point accumulation such as FIR filtering. Overflow may occur when data is transferred to memory because the C55x accumulators (AC0–AC3) have 40 bits, while the memory space is usually defined as a 16-bit word. In this experiment, we use an assembly routine `ovf_sat.asm` to evaluate the results with and without overflow protection. Table 3.11 lists a portion of the assembly code used for this experiment.

In the assembly program, the following code repeatedly adds the constant 0x140 to AC0:

```
    rptblocal add_loop_end-1
    add       #0x140<<#16,AC0
    mov       hi(AC0),*AR2+
add_loop_end
```

The updated value is stored at the buffer pointed at by AR2. The content of AC0 will grow larger and larger and eventually this accumulator will overflow. When the overflow occurs, a positive number in AC0 suddenly turns into negative. However, when the C55x saturation mode is set, the overflowed positive number will be limited to 0x7FFFFFFF. The second half of the code stores the left-shifted sinewave values to data memory locations. Without saturation protection, this shift will cause some of the shifted values to overflow.

Table 3.11 Program for experiment of overflow and saturation

```
    .def    _ovftest
    .def    _buff,_buff1

    .bss    _buff,(0x100)
    .bss    _buff1,(0x100)
;
;   Code start
;
_ovftest
    bclr    SATD                ; Clear saturation bit if set
    xcc     start,T0!=#0        ; If T0!=0, set saturation bit
    bset    SATD
start
    mov     #0,AC0
    amov    #_buff,XAR2         ; Set buffer pointer
    rpt     #0x100-1            ; Clear buffer
    mov     AC0,*AR2+
    amov    #_buff1,XAR2        ; Set buffer pointer
    rpt     #0x100-1            ; Clear buffer1
    mov     AC0,*AR2+

    mov     #0x80-1,BRC0        ; Initialize loop counts for addition
    amov    #_buff+0x80,XAR2    ; Initialize buffer pointer
    rptblocal add_loop_end-1
    add     #0x140<<#16,AC0     ; Use upper AC0 as a ramp up counter
    mov     hi(AC0),*AR2+       ; Save the counter to buffer
add_loop_end
    mov     #0x80-1,BRC0        ; Initialize loop counts for subtraction
    mov     #0,AC0
    amov    #_buff+0x7f,XAR2    ; Initialize buffer pointer
    rptblocal sub_loop_end-1
    sub     #0x140<<#16,AC0     ; Use upper AC0 as a ramp down counter
    mov     hi(AC0),*AR2-       ; Save the counter to buffer
sub_loop_end
    mov     #0x100-1,BRC0       ; Initialize loop counts for sinewave
    amov    #_buff1,XAR2        ; Initialize buffer pointer
    mov     mmap(@AR0),BSA01    ; Initialize base register
    mov     #40,BK03            ; Set buffer as size 40
    mov     #20,AR0             ; Start with an offset of 20 samples
    bset    AR0LC               ; Active circular buffer
    rptblocal sine_loop_end-1
    mov     *ar0+<<#16,AC0      ; Get sine value into high AC0
    sfts    AC0,#9              ; Scale the sine value
    mov     hi(AC0),*AR2+       ; Save scaled value
sine_loop_end
    mov     #0,T0               ; Return 0 if no overflow
    xcc     set_ovf_flag,overflow(AC0)
    mov     #1,T0               ; Return 1 if overflow detected
set_ovf_flag
    bclr    AR0LC               ; Reset circular buffer bit
    bclr    SATD                ; Reset saturation bit
    ret
    .end
```

Table 3.12 List of files for experiment `exp3.6.4_overflow`

Files	Description
`overflowTest.c`	C function for testing overflow experiment
`ovf_sat.asm`	Assembly function showing overflow
`overflow.pjt`	DSP project file
`overflow.cmd`	DSP linker command file

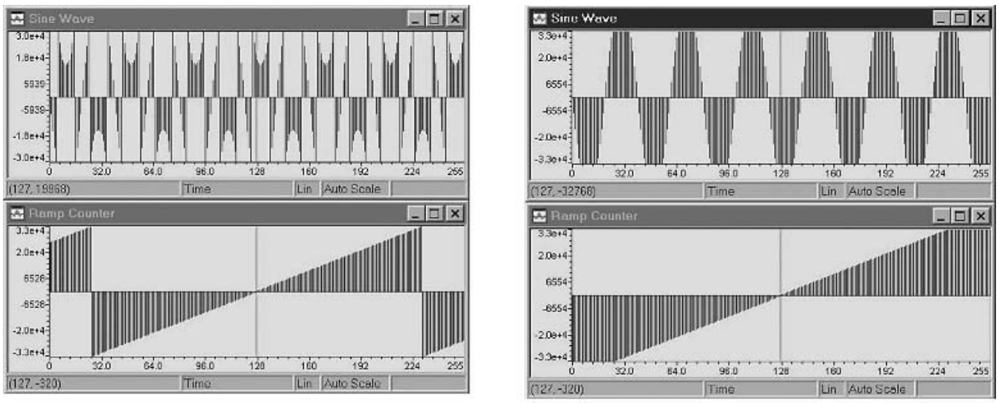
The following segment of code sets up and uses the circular addressing mode:

```
mov     #sineTable,BSA01    ; Initialize base register
mov     #40,BK03            ; Set buffer size to 40
mov     #20,AR0             ; Start with an offset of 20
bset    AR0LC               ; Activate circular buffer
```

The first instruction sets up the circular buffer base register (BSA01). The second instruction initializes the size of the circular buffer. The third instruction initializes the offset from the base as the starting point. In this case, the offset is set to 20 words from the base of `sineTable[]`. The last instruction enables AR0 as the circular pointer. Table 3.12 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Load the project `overflow.pjt`, rebuild, and load the program to DSK or CCS.
2. Use the graphic function to display the results `buff1` (top) and the `buff` (bottom) as shown in Figure 3.29.
3. Turn off overflow protection and repeat the experiment. Display and compare the results as shown in Figure 3.29(a) without saturation protection, and Figure 3.29(b) with saturation protection.



(a) (b)

Figure 3.29 Fixed-point implementation showing overflow and saturation: (a) without saturation protection; (b) with saturation protection

3.6.5 Function Approximations

This experiment uses polynomial approximation of sinusoidal functions to show a typical DSP algorithm design and implementation process. The DSP algorithm development usually starts with MATLAB or a floating-point C simulation, changes to fixed-point C, optimizes the code to improve its efficiency, and uses assembly language if necessary.

The cosine and sine functions can be expressed as the infinite power (Taylor) series expansion as follows:

$$\cos(\theta) = 1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \frac{1}{6!}\theta^6 + \cdots, \tag{3.91a}$$

$$\sin(\theta) = \theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - \frac{1}{7!}\theta^7 + \cdots, \tag{3.91b}$$

where θ is in radians and '!' represents the factorial operation. The accuracy of the approximation depends on the number of terms used in the series. Usually more terms are needed for larger values of θ. However, only a limited number of terms can be used in real-time DSP applications.

Floating-point C implementation

In this experiment, we implement the cosine function approximation in Equation (3.91a) using the C program listed in Table 3.13. In the function `fCos1( )`, 12 multiplications are required. The C55x compiler has a built-in run-time support library for floating-point arithmetic operations. These floating-point functions are very inefficient for real-time applications. For example, the program `fCos1()` needs over 2300 clock cycles to compute one sine value.

We can improve the computation efficiency by reducing the multiplication from 14 to 4. The modified program is listed in Table 3.14. This improved program reduces the clock cycles from 2300 to less than 1100. To further improve the efficiency, we will use the fixed-point C and assembly language programs.

The files used for this experiment are listed in Table 3.15. This experiment can be run on a DSK or a simulator.

Table 3.13 Floating-point C Program for cosine approximation

```
// Coefficients for cosine function approximation
double fcosCoef[4]={
   1.0, -(1.0/2.0), (1.0/(2.0*3.0*4.0)), -(1.0/(2.0*3.0*4.0*5.0*6.0))
};

// Direct implementation of function approximation
double fCos1(double x)
{
   double cosine;

   cosine  = fcosCoef[0];
   cosine += fcosCoef[1]*x*x;
   cosine += fcosCoef[2]*x*x*x*x;
   cosine += fcosCoef[3]*x*x*x*x*x*x;
   return(cosine);
}
```

Table 3.14 Improved floating-point C program for cosine approximation

```
// More efficient implementation of function approximation
double fCos2(double x)
{
   double cosine,x2;

   x2 = x * x;
   cosine = fcosCoef[3] * x2;
   cosine = (cosine+fcosCoef[2]) * x2;
   cosine = (cosine+fcosCoef[1]) * x2;
   cosine = cosine + fcosCoef[0];
   return(cosine);
}
```

Procedures of the experiment are listed as follows:

1. Load the project `floatingPointC.pjt`, rebuild, and load the program to the C5510 DSK or C55x simulator.
2. Run the program and verify the results.
3. Profile and record the cycles needed for approximating the function cos(x).

Fixed-point C implementation

Since the values of a cosine function are between +1.0 and −1.0, the fixed-point C uses Q15 format as shown in Table 3.16. This fixed-point C requires only 80 cycles, a significant improvement as compared with the floating-point C that requires 1100 cycles.

The fixed-point C implementation has effectively reduced the computation to 80 clock cycles per function call. This performance can be further improved by examining the program carefully. The CCS in mixed mode shows that the C multiplication uses the run-time support library function `I$$LMPY` and `MPYM` instruction as follows:

```
    cosine = (long)icosCoef[3] * x2;
    cosine = cosine >> 13;                   // Scale back to Q15
010013 D3B706        mpym *AR5(short(#3)),T2,AC0
010016 100533        sfts AC0,#-13,AC0
    cosine = (cosine + (long)icosCoef[2]) * x2;
    cosine = cosine >> 13;               // Scale back to Q15
010019 D6B500        add *AR5(short(#2)),AC0,AC0
01001C 6C010542      call I$$LMPY
010020 100533        sfts AC0,#-13,AC0
```

Table 3.15 List of files for experiment `exp3.6.5.1_using floating-pointC`

Files	Description
`fcos.c`	Floating-point C function approximation
`floatingPointC.pjt`	DSP project file
`funcAppro.cmd`	DSP linker command file

Table 3.16 Fixed-point C program for function approximation

```
#define UNITQ15 0x7FFF

// Coefficients for cosine function approximation
short icosCoef[4]={
   (short)(UNITQ15),
   (short)(-(UNITQ15/2.0)),
   (short)(UNITQ15/(2.0*3.0*4.0)),
   (short)(-(UNITQ15/(2.0*3.0*4.0*5.0*6.0)))
};

// Fixed-point implementation of function approximation
short iCos1(short x)
{
   long cosine,z;
   short x2;

   z = (long)x * x;
   x2 = (short)(z>>15);                   // x2 has x(Q14)*x(Q14)
   cosine = (long)icosCoef[3] * x2;
   cosine = cosine >> 13;                 // Scale back to Q15
   cosine = (cosine + (long)icosCoef[2]) * x2;
   cosine = cosine >> 13;                 // Scale back to Q15
   cosine = (cosine + (long)icosCoef[1]) * x2;
   cosine = cosine >> 13;                 // Scale back to Q15
   cosine = cosine + icosCoef[0];
   return((short)cosine);
}
```

As introduced in Chapter 2, the correct way of writing fixed-point C multiplication is

```
short b,c;
long a;
a = (long)b * (long)c;
```

The following changes ensure that the C55x compiler will generate the efficient instructions:

```
     cosine = (short)(cosine + (long)icosCoef[2]) * (long)x2;
     cosine = cosine >> 13;                          // Scale back to Q15
010015 D67590       add     *AR3(short(#2)),AC0,AR1
010018 2251         mov     T1,AC1
01001A 5290         mov     AR1,HI(AC0)
01001C 5804         mpy     T1,AC0,AC0
01001E 100533       sfts    AC0,#-13,AC0
```

The modified program (listed in Table 3.17) needs only 33 cycles. We write the fixed-point C code to mimic the instructions of DSP processor. Thus, this stage produces a ‘practical’ DSP program that can be run on the target DSP system, and used as reference for assembly programming. Table 3.18 briefly describes the files used for this experiment.

Table 3.17 Improved fixed-point C implementation

```
// Fixed-point C implementation that simulates assembly programming
short iCos(short T0)
{
    long  AC0;
    short *ptr;

    ptr = &icosCoef[3];
    AC0 = (long)T0 * T0;
    T0  = (short)(AC0>>15);                 // AC0 has T0(Q14)*T0(Q14)

    AC0 = (long)T0 * *ptr--;
    AC0 = AC0 >> 13;                        // Scale back to Q15
    AC0 = (short)(AC0 + *ptr--) * (long)T0;
    AC0 = AC0 >> 13;                        // Scale back to Q15
    AC0 = (short)(AC0 + *ptr--) * (long)T0;
    AC0 = AC0 >> 13;                        // Scale back to Q15
    AC0 = AC0 + *ptr;
    return((short)AC0);
}
```

Procedures of the experiment are listed as follows:

1. Load the project `fixedPointC.pjt`, rebuild, and load the program to the DSK.

2. Run the program and compare the results of cos(x) function with that obtained by floating-point C implementation.

3. Profile the cycles needed for running the function cos(x).

C55x assembly implementation

In many real-world applications, the DSP algorithms are written in assembly language or mixed C-and-assembly programs. The assembly implementation can be verified by comparing the output of the assembly program against the fixed-point C code. In this experiment, we write the cosine program in assembly language as shown in Table 3.19. This assembly function needs 19 cycles to compute a cosine value. With the overhead of function call setup and return in the mixed C-and-assembly environment, this program requires 30 cycles to generate a cosine value.

Table 3.18 List of files for experiment `exp3.6.5.2_using fixed-pointC`

Files	Description
`icos.c`	Fixed-point C function approximation
`fixedPointC.pjt`	DSP project file
`funcAppro.cmd`	DSP linker command file

Table 3.19 C55x assembly program for cosine function approximation

```
    .data
_icosCoef   ; [1 (-1/2!) (1/4!) (-1/6!)]
    .word   32767,-16383,1365,-45

    .sect ".text"
    .def _cosine

_cosine:
     amov #(_icosCoef+3),XAR3 ; ptr = &icosCoef[3];
     amov #AR1,AR2            ; AR1 is used as temp register
||   mov  T0,HI(AC0)
     sqr  AC0                 ; AC0 = (long)T0 * T0;
     sfts AC0,#-15            ; T0  = (short)(AC0>>15);
     mov  AC0,T0
     mpym *AR3-,T0,AC0        ; AC0 = (long)T0 * *ptr--;
     sfts AC0,#-13            ; AC0 = AC0 >> 13;
     add  *AR3-,AC0,AR1       ; AC0 = (short)(AC0 + *ptr--) * (long)T0;
     mpym *AR2,T0,AC0
     sfts AC0,#-13            ; AC0 = AC0 >> 13;
     add  *AR3-,AC0,AR1       ; AC0 = (short)(AC0 + *ptr--) * (long)T0;
     mpym *AR2,T0,AC0
     sfts AC0,#-13            ; AC0 = AC0 >> 13;
||   mov  *AR3,T0
     add  AC0,T0              ; AC0 = AC0 + *ptr;
     ret                      ; Return((short)AC0);
    .end
```

The real-time evaluation and test can be done using the hardware such as a DSK. The real-time experiments can verify system control and interrupt handling issues. To summarize the software design approach used in this experiment, we list the profile results of different implementations in Table 3.20. The files used for this experiment are listed in Table 3.21.

Procedures of the experiment are listed as follows:

1. Load the project `c55xCos.pjt`, rebuild, and load the program to the DSK.
2. Run the program and compare the results of assembly routine with those obtained by floating-point C implementation.
3. Profile the clock cycles needed for the assembly routine and compare with C implementations.

Table 3.20 Profile results of cosine approximation for different implementations

Arithmetic	Function	Implementation details	Profile (cycles/call)
Floating-point C	`fCos1( )`	Direct implementation, 12 multiplications	2315
	`fCos2( )`	Reduced multiplications, 4 multiplications	1097
Fixed-point C	`iCos1( )`	Using fixed-point arithmetic	88
	`iCos( )`	Using single multiplication instruction	33
Assembly language	`cosine( )`	Hand-code assembly routine	30

Table 3.21 List of files for experiment `exp3.6.5.3_using c55x assembly language`

Files	Description
`c55CosineTest.c`	C function for testing function approximation
`cos.asm`	Assembly routine for cosine approximation
`c55xCos.pjt`	DSP project file
`funcAppro.cmd`	DSP linker command file

Practical applications

Since the input arguments to cosine function are in the range of $-\pi$ to π, we must map the data range of $-\pi$ to π to the linear 16-bit data variables as shown in Figure 3.30. Using 16-bit wordlength, we map 0 to 0x0000, π to 0x7FFF, and $-\pi$ to 0x8000 for representing the radius arguments. Therefore, the function approximation given in Equation (3.91) is no longer the best choice, and different function approximation should be considered.

Using the Chebyshev approximation, $\cos(\theta)$ and $\sin(\theta)$ can be computed as

$$\cos(\theta) = 1 - 0.001922\theta - 4.90001474\theta^2 - 0.264892\theta^3 + 5.04541\theta^4 + 1.800293\theta^5, \tag{3.92a}$$

$$\sin(\theta) = 3.140625\theta + 0.020263670\theta^2 - 5.325196\theta^3 + 0.54467880\theta^4 + 1.800293\theta^5, \tag{3.92b}$$

where the value of θ is defined in the first quadrant, $0 \leq \theta < \pi/2$. For other quadrants, the following properties can be used to transfer it to the first quadrant:

$$\sin(180^\circ - \theta) = \sin(\theta), \qquad \cos(180^\circ - \theta) = -\cos(\theta) \tag{3.93}$$

$$\sin(-180^\circ + \theta) = -\sin(\theta), \quad \cos(-180^\circ + \theta) = -\cos(\theta) \tag{3.94}$$

and

$$\sin(-\theta) = -\sin(\theta), \quad \cos(-\theta) = \cos(\theta). \tag{3.95}$$

The C55x assembly routine (listed in Table 3.22) synthesizes the sine and cosine functions, which can be used to calculate the angle θ from -180° to 180°.

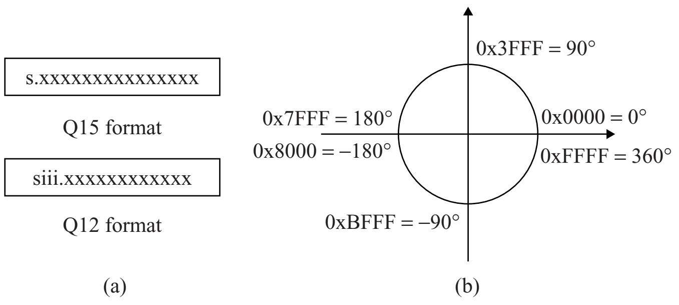


Figure 3.30 Scaled fixed-point number representation: (a) Q formats; (b) map angle value to 16-bit signed integer

Table 3.22 The C55x program for approximation of sine and cosine functions

```
    .def _sine_cos
;
;   Approximation coefficients in Q12 (4096) format
;
    .data
coeff ;   Sine approximation coefficients
    .word 0x3240 ; c1 =  3.140625
    .word 0x0053 ; c2 =  0.02026367
    .word 0xaacc ; c3 = -5.325196
    .word 0x08b7 ; c4 =  0.54467780
    .word 0x1cce ; c5 =  1.80029300
    ; Cosine approximation coefficients
    .word 0x1000 ; d0 =  1.0000
    .word 0xfff8 ; d1 = -0.001922133
    .word 0xb199 ; d2 = -4.90014738
    .word 0xfbc3 ; d3 = -0.2648921
    .word 0x50ba ; d4 =  5.0454103
    .word 0xe332 ; d5 = -1.800293
;
;   Function starts
;
    .text
_sine_cos
    amov    #14,AR2
    btstp   AR2,T0          ; Test bit 15 and 14
    nop
;
;   Start cos(x)
;
    amov    #coeff+10,XAR2 ; Pointer to the end of coefficients
    xcc     _neg_x,TC1
    neg     T0              ; Negate if bit 14 is set
_neg_x
    and     #0x7fff,T0      ; Mask out sign bit
    mov     *AR2-<<#16,AC0 ; AC0 = d5
||  bset    SATD            ; Set Saturate bit
    mov     *AR2-<<#16,AC1 ; AC1 = d4
||  bset    FRCT            ; Set up fractional bit
    mac     AC0,T0,AC1      ; AC1 = (d5*x+d4)
||  mov     *AR2-<<#16,AC0 ; AC0 = d3
    mac     AC1,T0,AC0      ; AC0 = (d5*x^2+d4*x+d3)
||  mov     *AR2-<<#16,AC1 ; AC1 = d2
    mac     AC0,T0,AC1      ; AC1 = (d5*x^3+d4*x^2+d3*x+d2)
||  mov     *AR2-<<#16,AC0 ; AC0 = d1
    mac     AC1,T0,AC0      ; AC0 = (d5*x^4+d4*x^3+d3*x^2+d2*x+d1)
||  mov     *AR2-<<#16,AC1 ; AC1 = d0
    macr    AC0,T0,AC1      ; AC1 = (d5*x^4+d4*x^3+d3*x^2+d2*x+d1)*x+d0
||  xcc     _neg_result1,TC2
    neg     AC1
_neg_result1
    mov     *AR2-<<#16,AC0 ; AC0 = c5
```

continues overleaf

Table 3.22 (*continued*)

```
||  xcc     _neg_result2,TC1
    neg     AC1
_neg_result2
    mov     hi(saturate(AC1<<#3)),*AR0+  ; Return cos(x) in Q15
;
;   Start sin(x) computation
;
    mov     *AR2-<<#16,AC1 ; AC1 = c4
    mac     AC0,T0,AC1     ; AC1 = (c5*x+c4)
||  mov     *AR2-<<#16,AC0 ; AC0 = c3
    mac     AC1,T0,AC0     ; AC0 = (c5*x^2+c4*x+c3)
||  mov     *AR2-<<#16,AC1 ; AC1 = c2
    mac     AC0,T0,AC1     ; AC1 = (c5*x^3+c4*x^2+c3*x+c2)
||  mov     *AR2-<<#16,AC0 ; AC0 = c1
    mac     AC1,T0,AC0     ; AC0 = (c5*x^4+c4*x^3+c3*x^2+c2*x+c1)
    mpyr    T0,AC0,AC1     ; AC1 = (c5*x^4+c4*x^3+c3*x^2+c2*x+c1)*x
||  xcc     _neg_result3,TC2
    neg     AC1
_neg_result3
    mov     hi(saturate(AC1<<#3)),*AR0-  ; Return sin(x) in Q15
||  bclr    FRCT           ; Reset fractional bit
    bclr    SATD           ; Reset saturate bit
    ret
    .end
```

Since the absolute value of the largest coefficient given in this experiment is 5.325196, we must scale the coefficients or use a different Q format as shown in Figure 3.22. We can achieve this by using the Q3.12 format, which has one sign bit, three integer bits, and 12 fraction bits to cover the range (−8, 8), as illustrated in Figure 3.30(a). In the example, we use Q3.12 format for all the coefficients, and map the angle $-\pi \leq \theta \leq \pi$ to a signed 16-bit number ($0x8000 \leq x \leq 0x7FFF$) as shown in Figure 3.30(b).

When the assembly subroutine sine_cos is called, a 16-bit mapped angle (function argument) is passed to the assembly routine using register T0 (see C calling conversion described in Chapter 2). The quadrant information is tested and stored in TC1 and TC2. If TC1 (bit 14) is set, the angle is located in either quadrant II or quadrant IV. We use the 2's complement to convert the angle to the first or third quadrant. We mask out the sign bit to calculate the third quadrant angle in the first quadrant, and the negation changes the fourth quadrant angle to the first quadrant. Therefore, the angle to be calculated is always located in the first quadrant. Because we use Q3.12 format for coefficients, the computed result needs to be left-shifted 3 bits to scale back to Q15 format. The files used for this experiment are listed in Table 3.23.

Table 3.23 List of files for experiment exp3.6.5.4_using assembly routine

Files	Description
sinCosTest.c	C function for testing function approximation
sine_cos.asm	Assembly routine for sine and cosine approximation
sin_cos.pjt	DSP project file
funcAppro.cmd	DSP linker command file

Procedures of the experiment are listed as follows:

1. Load the project `sine_cos.pjt`, rebuild, and load the program to the DSK or CCS simulator.

2. Calculate the angles in the following table, and run the experiment to read the approximation results and compare the differences.

θ	30°	45°	60°	90°	120°	135°	150°	180°
$\cos(\theta)$								
$\sin(\theta)$								
θ	−150°	−135°	−120°	−90°	−60°	−45°	−30°	0°
$\cos(\theta)$								
$\sin(\theta)$								

3. Explain the tasks of following C55x instructions:

   (a) `bset FRCT`, (b) `bset SATD`, (c) `bset SMUL`

4. Remove the assembly instruction `bset SATD` and rerun the experiment. Observe the difference of approximation results.

3.6.6 Real-Time Digital Signal Generation Using DSK

In this section, we will generate tones and random numbers using C5510 DSK. The generated signals will be played back in real time via AIC23 on the C5510 DSK.

Tone generation using floating-point C

In this experiment, we will generate and play a tone embedded in random noise using the C5510 DSK. Table 3.24 shows the functions that are used to generate a tone and random noise. The function `cos(x)` uses 4682 cycles per call and the function `rand()` uses only 87 cycles. Table 3.25 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Load the project `floatPointSigGen.pjt`, rebuild, and load the program to the DSK.

2. Connect a headphone to the headphone output of the DSK and start audio payback.

3. Listen to the audio output. Use a scope to observe the generated waveform.

Tone generation using fixed-point C

Refer to the experiment given above in section 3.6.5, we write a cosine function in C55x assembly language similar to Table 3.22. This assembly routine uses only 58 cycles per function call. Table 3.26 lists the files used for this experiment.

Table 3.24 Floating-point C program for tone and noise generation

```
#define UINTQ14     0x3FFF
#define PI          3.1415926

// Variable definition
static unsigned short n;
static float twoPI_f_Fs;

void initFTone(unsigned short f, unsigned short Fs)
{
    n = 0;
    twoPI_f_Fs = 2.0*PI*(float)f/(float)Fs;
}

short fTone(unsigned short Fs)
{
     n++;
     if (n >= Fs)
         n=0;

     return( (short)(cos(twoPI_f_Fs*(float)n)*UINTQ14));
}

void initRand(unsigned short seed)
{
     srand(seed);
}

short randNoise(void)
{
     return((rand()-RAND_MAX/2)>>1);
}
```

Table 3.25 List of files for experiment `exp3.6.6.1_using floating-PointC`

Files	Description
`floatSigGenTest.c`	C function for testing experiment
`ftone.c`	Floating-point C function for tone generation
`randNoise.c`	C function for generating random numbers
`floatPointSigGen.pjt`	DSP project file
`floatPointSigGencfg.cmd`	DSP linker command file
`floatPointSigGen.cdb`	DSP BIOS configuration file

Table 3.26 List of files for experiment `exp3.6.6.2_of tone generation`

Files	Description
`toneTest.c`	C function for testing experiment
`tone.c`	C function controls tone generation
`cos.asm`	Assembly routine computes cosine values
`toneGen.pjt`	DSP project file
`toneGencfg.cmd`	DSP linker command file
`toneGen.cdb`	DSP BIOS configuration file

Procedures of the experiment are listed as follows:

1. Load the project `toneGen.pjt`, rebuild, and load the program to the DSK.

2. Connect a headphone to the headphone output of the DSK and start playback of the tone.

3. Listen to the DSK output. Use a scope to observe the generated waveform.

Random number generation using fixed-point C

The linear congruential sequence method (will be further discussed in Chapter 8) is widely used because of its simplicity. The random number generation can be expressed as

$$x(n) = [ax(n-1) + b]_{\text{mod } M}\,, \tag{3.96}$$

where the modulo operation (mod) returns the remainder after division by M. For this experiment, we select $M = 2^{20} = 0\text{x}100000$, $a = 2045$, and $x(0) = 12\,357$. The C program for the random number generation is given in Table 3.27, where `seed=x(0)=12357`.

Floating-point multiplication and division are very slow on fixed-point DSP processors. We have learned in Chapter 2 that we can use a mask instead of modulo operation for a power-of-2 number. We improve the run-time efficiency by rewriting the program as listed in Table 3.28. The profile shows that the function `randNumber2( )` needs only 48 cycles while the original function `randNumber1( )` uses 427 cycles. The files used for this experiment are listed in Table 3.29.

Procedures of the experiment are listed as follows:

1. Load the project `randGenC.pjt`, rebuild, and load the program to the DSK.

2. Connect a headphone to the headphone output of the DSK and start the random signal generation.

3. Listen to the DSK output. Use a scope to observe the generated waveform.

Table 3.27 C program for random number generation

```
// Variable definition
static volatile long n;
static short a;

void initRand(long seed)
{
   n = (long)seed;
   a = 2045;
}

short randNumber1(void)
{
   short ran;

   n = a*n + 1;
   n = n - (long)((float)(n*0x100000)/(float)0x100000);
   ran = (n+1)/0x100001;
   return (ran);
}
```

Table 3.28 C program that uses mask for modulo operation

```
short randNumber2(void)
{
   short ran;

   n = a*n;
   n = n&0xFFFFF000;
   ran = (short)(n>>20);
   return (ran);
}
```

Random number generation using C55x assembly language

To further improve the performance, we use assembly language for random number generation. The assembly routine is listed in Table 3.30, which reduces the run-time clock cycles by 50%. Table 3.31 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Load the project `randGen.pjt`, rebuild, and load the program to the DSK.
2. Connect a headphone to the headphone output of the DSK and start the random signal generation.
3. Listen to the DSK output. Use a scope to observe the generated waveform.

Signal generation using C55x assembly language

This experiment combines the tone and random number generators for generating random noise, tone, and tone with additive random noise. The files used for this experiment are listed in Table 3.32.

Procedures of the experiment are listed as follows:

1. Load the project `signalGen.pjt`, rebuild, and load the program to the DSK.
2. Connect a headphone to the headphone output of the DSK and start signal generation.
3. Listen to the DSK output. Use a scope to observe the generated waveform.

Table 3.29 List of files for experiment `exp3.6.6.3_of random number generation`

Files	Description
`randTest.c`	C function for testing experiment
`rand.c`	C function generates random numbers
`randGenC.pjt`	DSP project file
`randGencfg.cmd`	DSP linker command file
`randGen.cdb`	DSP BIOS configuration file

Table 3.30 C55x assembly program of random number generator

```
  .bss  _n,2,0,2 ;    long n
  .bss  _a,1,0,0 ;    short a

  .def  _initRand
  .def  _randNumber

  .sect ".text"
_initRand:
    mov  AC0,dbl(*(#_n))      ; n = (long)seed;
    mov  #2045,*(#_a)         ; a = 2045;
    ret

_randNumber:
    amov  #_n,XAR0
    mov   *(#_a),T0
    mpym  *AR0+,T0,AC0       ; n = a*n;
    mpymu *AR0-,T0,AC1       ; This is an 32x16 integer multiply
    sfts  AC0,#16
    add   AC1,AC0
||  mov   #0xFFFF<<#16,AC2   ; n = n&0xFFFFF000;
    or    #0xF000,AC2
    and   AC0,AC2
    mov   AC2,dbl(*AR0)
||  sfts  AC2,#-20,AC0       ; ran = (short)(n>>20);
    mov   AC0,T0             ; Return (ran);
    ret
    .end
```

Table 3.31 List of files for experiment `exp3.6.6.4_using assembly routine`

Files	Description
`randTest.c`	C function for testing experiment
`rand.asm`	Assembly routine generates random numbers
`randGen.pjt`	DSP project file
`randGencfg.cmd`	DSP linker command file
`randGen.cdb`	DSP BIOS configuration file

Table 3.32 List of files for experiment `exp3.6.6.5_of signal generation`

Files	Description
`sigGenTest.c`	C function for testing experiment
`tone.c`	C function controls tone generation
`cos.asm`	Assembly routine computes cosine values
`rand.asm`	Assembly routine generates random numbers
`signalGen.pjt`	DSP project file
`signalGencfg.cmd`	DSP linker command file
`signalGen.cdb`	DSP BIOS configuration file

References

[1] N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1983.
[2] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1989.
[3] S. J. Orfanidis, *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1996.
[4] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd Ed., Englewood Cliffs, NJ: Prentice Hall, 1996.
[5] P. Peebles, *Probability, Random Variables, and Random Signal Principles*, New York, NY: McGraw-Hill, 1980.
[6] A Bateman and W. Yates, *Digital Signal Processing Design*, New York: Computer Science Press, 1989.
[7] S. M. Kuo and D. R. Morgan, *Active Noise Control Systems – Algorithms and DSP Implementations*, New York, NY: John Wiley & Sons, Inc., 1996.
[8] C. Marven and G. Ewers, *A Simple Approach to Digital Signal Processing*, New York: John Wiley & Sons, Inc., 1996.
[9] J. H. McClellan, R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach*, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1998.
[10] D. Grover and J. R. Deller, *Digital Signal Processing and the Microcontroller*, Upper Saddle River, NJ: Prentice Hall, 1999.
[11] S. M. Kuo and W. S. Gan, *Digital Signal Processors – Architectures, Implementations, and Applications*, Upper Saddle River, NJ: Prentice Hall, 2005.
[12] MathWorks, Inc.,*Using MATLAB*, Version 6, 2000.
[13] MathWorks, Inc., *Signal Processing Toolbox User's Guide*, Version 6, 2004.
[14] MathWorks, Inc., *Filter Design Toolbox User's Guide*, Version 3, 2004.
[15] MathWorks, Inc., *Fixed-Point Toolbox User's Guide*, Version 1, 2004.

Exercises

1. The all-digital touch-tone phones use the sum of two sinewaves for signaling. Frequencies of these sinewaves are defined as 697, 770, 852, 941, 1209, 1336, 1477, and 1633 Hz. The sampling rate used by the telecommunications is 8 kHz, converts those 8 frequencies in terms of radians per sample and cycles per sample.

2. Compute the impulse response $h(n)$ for $n = 0, 1, 2, 3, 4$ of the digital systems defined by the following I/O equations:

   (a) $y(n) = x(n) = 0.75y(n-1)$;

   (b) $y(n) - 0.3y(n-1) - 0.4y(n-2) = x(n) - 2x(n-1)$; and

   (c) $y(n) = 2x(n) - 2x(n-1) + 0.5x(n-2)$.

3. Construct detailed signal-flow diagrams for the digital systems defined in Problem 2.

4. Similar to the signal-flow diagram for the IIR filter as shown in Figure 3.11, construct a general signal-flow diagram for the IIR filter defined in Equation (3.42) for $M \neq L-1$.

5. Find the transfer functions for the three digital systems defined in Problem 2.

6. Find the zero(s) and/or pole(s) of the digital systems given in Problem 2. Discuss the stability of these systems.

7. For a second-order IIR filter defined in Equation (3.42) with two complex poles defined in (3.52), the radius $r = 0.9$ and the angle $\theta = 0.25\pi$. Find the transfer function and I/O equation of this filter.

8. A 2 kHz sinewave is sampled with 10-kHz sampling rate, what is the sampling period? What is the digital frequency in terms of ω and F? If we have 100 samples, how many cycles of sinewave are covered?

9. For the digital sinewave given in Problem 8, if we compute the DFT with $N = 100$, what is the frequency resolution? If we display the magnitude spectrum as shown in Figure 3.17, what is the value of k corresponding to the peak spectrum? What happens if the frequencies of sinewave are 1.5 and 1.05 kHz?

10. Similar to Table 3.2, construct a new table for 5-bit binary numbers.

11. Find the fixed-point 2's complement binary representation with $B = 6$ for the decimal numbers 0.5703125 and −0.640625. Also, find the hexadecimal representation of these two numbers. Round the binary numbers to 6 bits and compute the corresponding roundoff errors.

12. Similar to Example 3.26, represent the two fractional numbers in Problem 11 in integer format for the C55x assembly programs.

13. Represent the 16-bit number given in Example 3.27 in Q1.14, Q3.12, and Q15.0 formats.

14. If the quantization process uses truncating instead of rounding, show that the truncation error $e(n) = x(n) - x(nT)$ will be in the interval $-\Delta < e(n) < 0$. Assuming that the truncation error is uniformly distributed in the interval $(-\Delta, 0)$, compute the mean and the variance of $e(n)$.

15. Generate and plot (20 samples) the following sinusoidal signals using MATLAB:

    (a) $A = 1$, $f = 100$ Hz, and $f_s = 1$ kHz;

    (b) $A = 1$, $f = 400$ Hz, and $f_s = 1$ kHz'

    (c) discuss the difference of results between (a) and (b);

    (d) $A = 1$, $f = 600$ Hz, and $f_s = 1$ kHz; and

    (e) compare and explain the results obtained from (b) and (d).

16. Use MATLAB to show pole-zero diagram of three digital systems given in Problem 2.

17. Use MATLAB to display magnitude and phase responses of three digital systems given in Problem 2.

18. Generate 1024 samples of pseudo-random numbers with zero mean and unit variance using the MATLAB function `rand`. Then use MATLAB functions `mean`, `std`, and `hist` to verify the results.

19. Generate 1024 samples of sinusoidal signal at frequency 1000 Hz, amplitude equal to unity, and the sampling rate 8000 Hz. Mix the generated sinewave with the zero-mean pseudo-random number of variance 0.2. What is the SNR? Calculate and display the magnitude spectrum using MATLAB script.

20. Write a MATLAB or C program to implement the moving-average filter defined in Equation (3.15). Test the filter using the corrupted sinewave generated in Problem 18 as input for different L. Plot both the input and output waveforms and magnitude spectra. Discuss the results related to the filter order L.

21. Given the difference equations in Problem 2, calculate and plot the impulse response $h(n)$, $n = 0, 1, \ldots, 127$, using MATLAB.

22. Similar to Example 3.31, use MATLAB functions `quantizer` and `quantize` to convert the speech file `timit1.asc` given in Example 3.29 to 4-, 8-, and 12-bit data, and use `soundsc` to playback the quantized signals.

23. Select the proper radix conversion functions listed in Table 3.4 to convert the white noise generated in Example 3.32 to hex format.

24. Use DSK to conduct the quantization experiment in real-time.

    (a) Generate an analog signal such as a sinewave using a signal generator for DSK input signal. Both the input and output channels of the DSK are displayed on an oscilloscope. Assuming that the ADC has 16-bit resolutions, adjust the amplitude of input signal to the full scale of ADC without clipping the waveform. Vary the number of bits (by shifting out or masking) to 14, 12, 10, etc. to represent the signal and output the signal to DAC. Observe the output waveform using the oscilloscope.

    (b) Replace the input source with a microphone, radio line output, or CD player, and send the DSK output to a loudspeaker for audio playback. Vary the number of bits (16, 12, 8, 4, etc.) for the output signal, and listen to the output sound. Depending on the type of loudspeaker being used, we may need to use an amplifier to drive the loudspeaker.

25. Implement the following square-root approximation equation in C55x assembly language:

$$\sqrt{x} = 0.2075806 + 1.454895x - 1.34491x^2 + 1.106812x^3 - 0.536499x^4 + 0.1121216x^5$$

This equation approximates an input variable within the range of $0.5 \leq x \leq 1$. Based on the values in the following table, calculate $\sqrt{x}$:

x	0.5	0.6	0.7	0.8	0.9
$\sqrt{x}$					

26. Write a C55x assembly function to implement the inverse square-root approximation equation as following:

$$1/\sqrt{x} = 1.84293985 - 2.57658958x + 2.11866164x^2 - 0.67824984x^3.$$

This equation approximates an input variable in the range of $0.5 \leq x \leq 1$. Use this approximation equation to compute $1/\sqrt{x}$ in the following table:

x	0.5	0.6	0.7	0.8	0.9
$1/\sqrt{x}$					

Note that $1/\sqrt{x}$ will result a number greater than 1.0. Try to use Q14 data format. That is, use 0x3FFF for 1 and 0x2000 for 0.5, and scale back to Q15 after calculation.

27. Write a C55x assembly function to implement the arctangent function as following:

$$\tan^{-1}(x) = 0.318253x + 0.003314x^2 - 0.130908x^3 + 0.068542x^4 - 0.009195x^5.$$

This equation approximates an input variable in the range of $x < 1$. Use this approximation equation to compute $\tan^{-1}(x)$ in the following table:

x	0.1	0.3	0.5	0.7	0.9
$\tan^{-1}(x)$					

28. Write a C55x assembly function to generate a $\sin(x)$ table with following requirements:
The input variable $0^{o} \leq x \leq 45^{o}$ with resolution 0.5^{o}.
Output must over $0^{o} \leq x \leq 359^{o}$with resolution 1.0^{o}.
Hint: this program can be implemented with the following steps:

    (a) implement $\sin(x)$and$\cos(x)$ for $0^{o} \leq x \leq 45^{o}$;

    (b) use $\sin(2x) = \sin(x)\cos(x)$ for $0^{o} \leq x \leq 90^{o}$; and

    (c) use $\sin(180^\circ - x) = \sin(x)$ and $\sin(-180^\circ + x) = -\sin(x)$ to map the rest values for the table.

29. If the random variables x_i are independent of the mean m_i and variance σ_i^2, and the random variable y is defined as

$$y = x_1 + x_2 + \cdots + x_N = \sum_{i=1}^{N} x_i,$$

the probability density function becomes a Gaussian (normal) distribution function (normal curve) as $N \rightarrow \infty$. Write a DSP assembly routine to generate a Gaussian random noise with $N = 12$ in real time using the C55x DSK.

30. Write a DSP program to generate a tone with an additive Gaussian random noise generated in Problem 29 with SNR = 20 dB. Play this signal in real time at 32-kHz sampling rate by DSK. Modify the program such that SNR = 10 dB. Play this signal again using DSK.

4 Design and Implementation of FIR Filters

A digital filter is a mathematical algorithm implemented in hardware, firmware, and/or software for achieving filtering objectives. Digital filters can be classified as linear or nonlinear filters. Linear filters can be divided into FIR and IIR filters. This chapter focuses on the analysis, design, implementation, and application of the digital FIR filters.

4.1 Introduction to FIR Filters

Some advantages of FIR filters are summarized as follows:

1. The FIR filters are always stable because there is no feedback from past outputs and the absence of poles.

2. The design of linear-phase filters can be guaranteed.

3. The finite-precision errors are less severe in FIR filters than in IIR filters.

4. FIR filters can be efficiently implemented on most DSP processors with MAC units, circular buffers, zero-overhead loops, and special instructions for FIR filtering.

The process of deriving filter coefficients that satisfies a given set of specifications is called filter design. Even though a number of computer-aided design tools are widely available for designing digital filters, we still need to understand the characteristics of filters and be familiar with techniques used for implementing digital filters.

4.1.1 Filter Characteristics

Linear, time-invariant filters are characterized by magnitude response, phase response, stability, rising time, settling time, and overshoot. Magnitude and phase responses determine the steady-state response of the filter; while the rising time, settling time, and overshoot specify the transient response. For an instantaneous input change, the rising time specifies its output-changing rate. The settling time describes the amount of time for an output to settle down to a stable value, and the overshoot shows if the output exceeds the desired value.

When a signal passes through a filter, its amplitude and phase are modified by the filter. The magnitude response of the filter specifies the gains at certain frequencies, and the phase response affects phase and time delay of frequency components. The group-delay function is defined as

$$T_{\mathrm{d}}(\omega) = \frac{-\mathrm{d}\phi(\omega)}{\mathrm{d}\omega}, \tag{4.1}$$

where $\phi(\omega)$ is the phase spectrum. A linear-phase filter has phase response that satisfies

$$\phi(\omega) = -\alpha\omega \quad \text{or} \quad \phi(\omega) = \pi - \alpha\omega. \tag{4.2}$$

Therefore, for a filter with linear phase, the group delay $T_{\mathrm{d}}(\omega)$ given in Equation (4.1) is a constant α for all frequencies. This filter avoids phase distortion because all frequency components in the input are delayed by the same amount. Linear phase is important in many real-world applications where the temporal relationships between different frequency components are critical.

Example 4.1: Considering a simple moving-average filter given in Example 3.16, the frequency response is

$$H(\omega) = \frac{1}{2}\left(1 + e^{-j\omega}\right) = \frac{1}{2}e^{-j\omega/2}\left[e^{j\omega/2} + e^{-j\omega/2}\right] = e^{-j\omega/2}\cos(\omega/2).$$

Therefore, we have the squared-magnitude response

$$|H(\omega)|^2 = \cos^2\left(\frac{\omega}{2}\right) = \frac{1}{2}\left[1 + \cos(\omega)\right].$$

Since the magnitude response falls off monotonically to zero at $\omega = \pi$, this is a lowpass filter with the phase response

$$\phi(\omega) = \frac{-\omega}{2},$$

which is a linear phase as shown in Equation (4.2). Therefore, this filter has constant time delay

$$T_{\mathrm{d}}(\omega) = \frac{-\mathrm{d}\phi(\omega)}{\mathrm{d}\omega} = 0.5.$$

These characteristics can be verified using the following MATLAB script (`example4_1.m`):

```
b=[0.5, 0.5];
a = [1];         % Define filter coefficients
freqz(b,a); % Show magnitude and phase responses
```

The magnitude and phase responses of `freqz(b,a)` are illustrated in Figure 4.1. Group delay can be computed and displayed using `grpdelay(b,a)`.

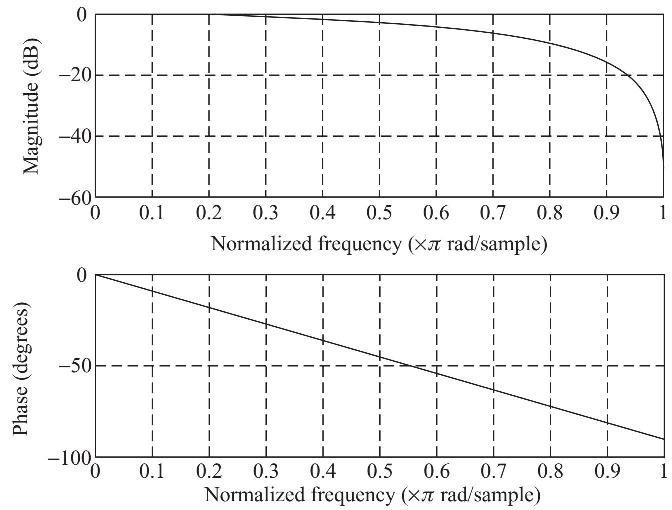


Figure 4.1 Magnitude and phase responses of simple moving-average filter

4.1.2 Filter Types

If a filter is defined in terms of its magnitude response, there are four different types of filters: lowpass, highpass, bandpass, and bandstop filters. Because the magnitude response of a digital filter with real coefficients is an even function of ω, the filter specifications are defined in the range $0 \leq \omega \leq \pi$.

The magnitude response of an ideal lowpass filter is illustrated in Figure 4.2(a). The regions $0 \leq \omega \leq \omega_c$ and $\omega > \omega_c$ are referred to as the passband and stopband, respectively, and the frequency ω_c that separates

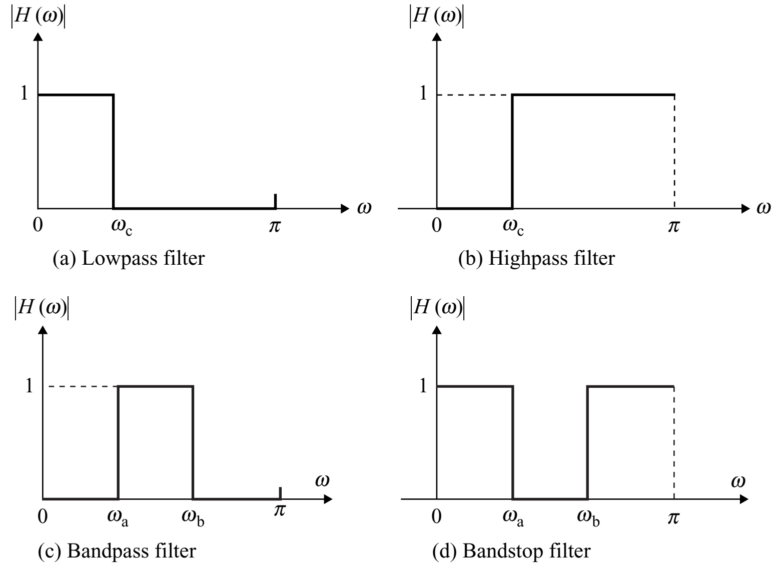


Figure 4.2 Magnitude response of ideal filters: (a) lowpass; (b) highpass; (c) bandpass; and (d) bandstop

the passband and stopband is called the cutoff frequency. An ideal lowpass filter has magnitude response $|H(\omega)| = 1$ in the frequency range $0 \le \omega \le \omega_c$, and $|H(\omega)| = 0$ for $\omega > \omega_c$. Thus, a lowpass filter passes low-frequency components below the cutoff frequency and attenuates high-frequency components above ω_c.

The magnitude response of an ideal highpass filter is illustrated in Figure 4.2(b). A highpass filter passes high-frequency components above the cutoff frequency ω_c and attenuates low-frequency components below ω_c. In practice, highpass filters can be used to eliminate DC offset, 60 Hz hum, and other low-frequency noises.

The magnitude response of an ideal bandpass filter is illustrated in Figure 4.2(c). The frequencies ω_a and ω_b are called the lower cutoff frequency and the upper cutoff frequency, respectively. A bandpass filter passes frequency components between the two cutoff frequencies ω_a and ω_b, and attenuates frequency components below the frequency ω_a and above the frequency ω_b.

The magnitude response of an ideal bandstop (or band-reject) filter is illustrated in Figure 4.2(d). A filter with a very narrow stopband is also called a notch filter. For example, a power line generates a 60 Hz sinusoidal noise called 60 Hz hum, which can be removed by a notch filter with notch frequency at 60 Hz.

In addition to these frequency-selective filters, an allpass filter provides frequency response $|H(\omega)| = 1$ for all ω. The principal use of allpass filters is to correct the phase distortion introduced by physical systems and/or other filters. A very special case of the allpass filter is the ideal Hilbert transformer, which produces a 90° phase shift to input signals.

A multiband filter has more than one passband and stopband. A special case of the multiband filter is the comb filter. A comb filter has equally spaced zeros with the shape of the magnitude response resembling a comb. The difference equation of the comb filter is given as

$$y(n) = x(n) - x(n - L), \tag{4.3}$$

where the number of delay L is an integer. The transfer function of this FIR filter is

$$H(z) = 1 - z^{-L} = \frac{z^L - 1}{z^L}. \tag{4.4}$$

Thus, the comb filter has L poles at the origin and L zeros equally spaced on the unit circle at

$$z_l = e^{j(2\pi/L)l}, \quad l = 0, 1, \ldots, L - 1. \tag{4.5}$$

Example 4.2: A comb filter with $L = 8$ has eight zeros at

$$z_l = 1, e^{\pi/4}, e^{\pi/2}, e^{3\pi/4}, e^{\pi} = -1, e^{5\pi/4}, e^{3\pi/2}, e^{7\pi/4}.$$

The frequency response can be computed and plotted in Figure 4.3 using the following MATLAB script (`example4_2.m`) for $L = 8$:

```
b=[1 0 0 0 0 0 0 0 -1]; a=[1];
freqz(b, a);
```

Figure 4.3 shows that the comb filter can be used as a crude bandstop filter to remove harmonics at frequencies

$$\omega_l = 2\pi l/L, \qquad l = 0, 1, \ldots, L/2 - 1. \tag{4.6}$$

The center of the passband lies halfway between the zeros of the response; that is, at frequencies $\frac{(2l+1)\pi}{L}$, $l = 0, 1, \ldots, L/2 - 1$.

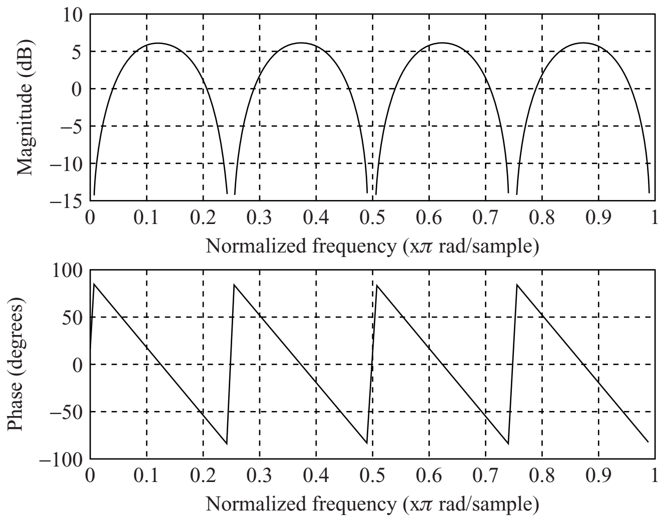


Figure 4.3 Magnitude and phase responses of a comb filter

Comb filters are useful for passing or eliminating specific frequencies and their harmonics. Using comb filters for attenuating periodic signals with harmonic related components is more efficient than having individual filters for each harmonic. For example, the humming sound produced by large transformers located in electric utility substations is composed with even-numbered harmonics (120, 240, 360 Hz, etc.) of the 60 Hz power-line frequency. When a desired signal is corrupted by the transformer noise, the comb filter with notches at the multiples of 120 Hz can be used to eliminate those undesired harmonic components.

Example 4.3: We can selectively cancel zeros in a comb filter with corresponding poles. Canceling the zero provides a passband, while the remaining zeros provide attenuation for stopband. If we add a pole at $z = 1$, the transfer function given in Equation (4.4) changes to

$$H(z) = \frac{1 - z^{-L}}{1 - z^{-1}}. \tag{4.7}$$

The pole at $z = 1$ is canceled by the zero at $z = 1$, resulting in a lowpass filter with passband centered at $z = 1$. The system defined by Equation (4.7) is still an FIR filter because the pole is canceled. Applying the scaling factor $1/L$ to Equation (4.7), the transfer function becomes

$$H(z) = \frac{1}{L}\left(\frac{1 - z^{-L}}{1 - z^{-1}}\right). \tag{4.8}$$

This is the moving-average filter introduced in Chapter 3. Note that canceling the zero at $z = 1$ produces a lowpass filter and canceling the zero at $z = -1$ produces a highpass filter. This is because that $z = 1$ is corresponding to $\omega = 0$ and $z = -1$ is corresponding to $\omega = \pi$.

4.1.3 Filter Specifications

The characteristics of digital filters are often specified in the frequency domain, and thus the design is based on magnitude-response specifications. In practice, we cannot achieve the infinitely sharp cutoff as

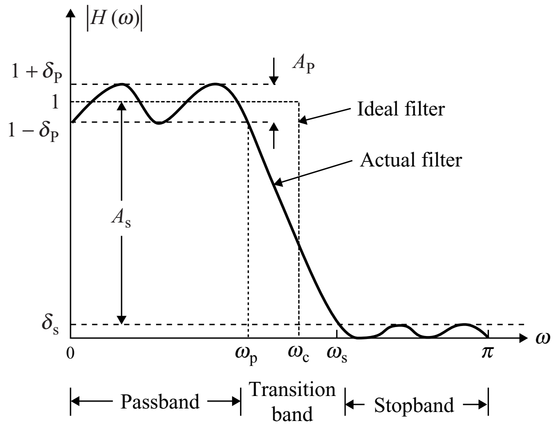


Figure 4.4 Magnitude response and performance measurement of a lowpass filter

the ideal filters given in Figure 4.2. We must accept a more gradual cutoff with a transition band between the passband and the stopband. The specifications are often given in the form of tolerance (or ripple) schemes, and a transition band is specified to permit the smooth magnitude roll-off. A typical magnitude response of lowpass filter is illustrated in Figure 4.4. The dotted horizontal lines in the figure indicate the tolerance limits. The magnitude response has a peak deviation δ_p in the passband, and a maximum deviation δ_s in the stopband. The frequencies ω_p and ω_s are the passband edge (cutoff) frequency and the stopband edge frequency, respectively.

As shown in Figure 4.4, the magnitude of passband ($0 \le \omega \le \omega_p$) approximates unity with an error of $\pm\delta_p$. That is,

$$1 - \delta_p \le |H(\omega)| \le 1 + \delta_p, \qquad 0 \le \omega \le \omega_p. \tag{4.9}$$

The passband ripple δ_p is the allowed variation in magnitude response in the passband. Note that the gain of the magnitude response is normalized to 1 (0 dB).

In the stopband, the magnitude response approximates zero with an error δ_s. That is,

$$|H(\omega)| \le \delta_s, \qquad \omega_s \le \omega \le \pi. \tag{4.10}$$

The stopband ripple (or attenuation) δ_s describes the minimum attenuation for signal components above the ω_s.

Passband and stopband deviations are usually expressed in decibels. The peak passband ripple and the minimum stopband attenuation in decibels are defined as

$$A_p = 20\log_{10}\left(\frac{1+\delta_p}{1-\delta_p}\right)\ \text{dB} \tag{4.11}$$

and

$$A_s = -20\log_{10}\delta_s\ \text{dB}. \tag{4.12}$$

Example 4.4: Consider a filter has passband ripples within ±0.01; that is, $\delta_p = 0.01$. From Equation (4.11), we have

$$A_p = 20\log_{10}\left(\frac{1.01}{0.99}\right) = 0.1737\,\text{dB}.$$

When the minimum stopband attenuation is given as $\delta_s = 0.01$, we have

$$A_s = -20\log_{10}(0.01) = 40\,\text{dB}.$$

The transition band is the area between the passband edge frequency ω_p and the stopband edge frequency ω_s. The magnitude response decreases monotonically from the passband to the stopband in this region. The width of the transition band determines how sharp the filter is. Generally, a higher order filter is needed for smaller δ_p and δ_s, and narrower transition band.

4.1.4 Linear-Phase FIR Filters

The signal-flow diagram of the FIR filter is shown in Figure 3.6, and the I/O equation is defined in Equation (3.15). If L is an odd number, we define $M = (L-1)/2$. Equation (3.15) can be written as

$$B(z) = \sum_{l=0}^{2M} b_l z^{-l} = \sum_{l=-M}^{M} b_{l+M} z^{-(l+M)} = z^{-M}\left[\sum_{l=-M}^{M} h_l z^{-l}\right] = z^{-M}H(z), \tag{4.13}$$

where

$$H(z) = \sum_{l=-M}^{M} h_l z^{-l}. \tag{4.14}$$

Let h_l have the symmetry property as

$$h_l = h_{-l}, \qquad l = 0, 1, \ldots, M. \tag{4.15}$$

From Equation (4.13), the frequency response $B(\omega)$ can be written as

$$\begin{aligned} B(\omega) &= B(z)|_{z=e^{j\omega}} = e^{-j\omega M}H(\omega) \\ &= e^{-j\omega M}\left[\sum_{l=-M}^{M} h_l e^{-j\omega l}\right] = e^{-j\omega M}\left[h_0 + \sum_{l=1}^{M} h_l\left(e^{j\omega l} + e^{-j\omega l}\right)\right] \\ &= e^{-j\omega M}\left[h_0 + 2\sum_{l=1}^{M} h_l \cos(\omega l)\right]. \end{aligned} \tag{4.16}$$

If L is an even integer and $M = L/2$, the derivation of Equation (4.16) has to be modified slightly.

If h_l is real valued, $H(\omega)$ is a real function of ω. If $H(\omega) \geq 0$, then the phase of $B(\omega)$ is equal to

$$\phi(\omega) = -\omega M, \tag{4.17}$$

which is a linear function of ω. However, if $H(\omega) < 0$, then the phase of $B(\omega)$ is equal to $\pi - \omega M$. Thus, there are sign changes in $H(\omega)$ corresponding to 180° phase shifts in $B(\omega)$, and $B(\omega)$ is only piecewise linear as shown in Figure 4.3.

If h_l has the antisymmetry property as

$$h_l = -h_{-l}, \qquad l = 0, 1, \ldots, M, \tag{4.18}$$

this implies $h(0) = 0$. Following the derivation of Equation (4.16), we can also show that the phase of $B(z)$ is a linear function of ω.

In conclusion, an FIR filter has linear phase if its coefficients satisfy the positive symmetric condition

$$b_l = b_{L-1-l}, \qquad l = 0, 1, \ldots, L-1, \tag{4.19}$$

or the antisymmetric condition (negative symmetry)

$$b_l = -b_{L-1-l}, \qquad l = 0, 1, \ldots, L-1. \tag{4.20}$$

The group delay of a symmetric (or antisymmetric) FIR filter is $T_{\text{d}}(\omega) = (L-1)/2$, which corresponds to the midpoint of the FIR filter. Depending on whether L is even or odd and whether b_l has positive or negative symmetry, there are four types of linear-phase FIR filters as illustrated in Figure 4.5.

The symmetry (or antisymmetry) property of a linear-phase FIR filter can be exploited to reduce the total number of multiplications required by filtering. Consider the FIR filter with even length L and positive symmetric as defined in Equation (4.19), Equation (3.40) can be combined as

$$H(z) = b_0\left(1 + z^{-L+1}\right) + b_1\left(z^{-1} + z^{-L+2}\right) + \cdots + b_{L/2-1}\left(z^{-L/2+1} + z^{-L/2}\right). \tag{4.21}$$

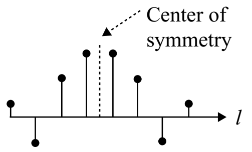
A realization of $H(z)$ defined in Equation (4.21) is illustrated in Figure 4.6 with the I/O equation expressed as

$$\begin{aligned} y(n) &= b_0\left[x(n) + x(n-L+1)\right] + b_1\left[x(n-1) + x(n-L+2)\right] \\ &\quad + \cdots + b_{L/2-1}\left[x(n-L/2+1) + x(n-L/2)\right] \\ &= \sum_{l=0}^{L/2-1} b_l\left[x(n-l) + x(n-L+1+l)\right]. \end{aligned} \tag{4.22}$$

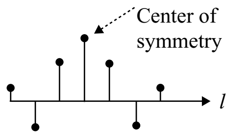
For an antisymmetric FIR filter, the addition of two signals is replaced by subtraction. That is,

$$y(n) = \sum_{l=0}^{L/2-1} b_l\left[x(n-l) - x(n-L+1+l)\right]. \tag{4.23}$$

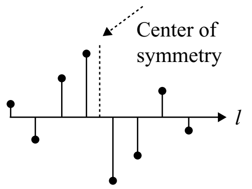
As shown in Equation (4.22) and Figure 4.6, the number of multiplications is cut in half by adding the pairs of samples, and then multiplying the sum by the corresponding coefficient. The trade-off is that we need two address pointers that point at both $x(n-l)$ and $x(n-L+1+l)$ instead of accessing data linearly through the same buffer with a single pointer. The TMS320C55x provides two special instructions `firsadd` and `firssub` for implementing the symmetric and antisymmetric FIR filters, respectively. In Section 4.5, we will demonstrate how to use the symmetric FIR instructions for experiments.



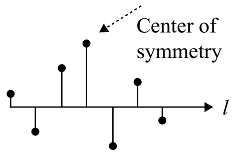
(a) Type I: L even ($L = 8$), positive symmetry.



(b) Type II: L odd ($L = 7$), positive symmetry.



(c) Type III: L even ($L = 8$), negative symmetry.



(d) Type IV: L odd ($L = 7$), negative symmetry.

Figure 4.5 Coefficients of the four types of linear-phase FIR filters: (a) type I: L even ($L = 8$), positive symmetry; (b) type II: L odd ($L = 7$), positive symmetry; (c) type III: L even ($L = 8$), negative symmetry; and (d) type IV: L odd ($L = 7$), negative symmetry

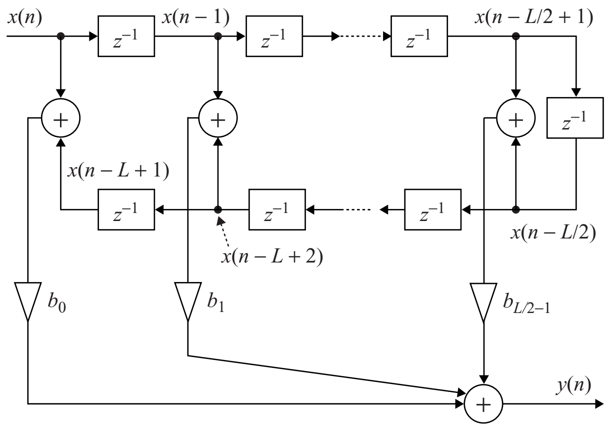


Figure 4.6 Signal-flow diagram of symmetric FIR filter, L is even

4.1.5 Realization of FIR Filters

An FIR filter can be operated on either a block basis or a sample-by-sample basis. In the block processing, the input samples are segmented into multiple data blocks. Filtering is performed one block at a time, and the resulting output blocks are recombined to form the overall output. The filtering of each data block can be implemented using the linear convolution or fast convolution, which will be introduced in Chapter 6. In the sample-by-sample processing, the input samples are processed at every sampling period after the current input $x(n)$ becomes available.

As discussed in Section 3.2.1, the output of an LTI system is the input samples convoluted with the impulse response coefficients of the system. Assuming that the filter is casual, the output at time n is given as

$$y(n) = \sum_{l=0}^{\infty} h(l)x(n-l). \tag{4.24}$$

The process of computing the linear convolution involves the following four steps:

1. *Folding*: Fold $x(l)$ about $l = 0$ to obtain $x(-l)$.

2. *Shifting*: Shift $x(-l)$ by n samples to the right to obtain $x(n-l)$.

3. *Multiplication*: Multiply $h(l)$ by $x(n-l)$ to obtain the products of $h(l) \times (n-l)$ for all l.

4. *Summation*: Sum all the products to obtain the output $y(n)$ at time n.

Repeat Steps 2 through 4 in computing the output of the system at other time instants n. Note that the convolution of the length M input signal with the length L impulse response results in length $L + M -$ 1 output signal.

Example 4.5: Considering an FIR filter that consists of four coefficients b_0, b_1, b_2, and b_3, we have

$$y(n) = \sum_{l=0}^{3} b_l x(n-l), \qquad n \geq 0.$$

The linear convolution yields

$$\begin{aligned}
n &= 0,\ y(0) = b_0x(0)\\
n &= 1,\ y(1) = b_0x(1) + b_1x(0)\\
n &= 2,\ y(2) = b_0x(2) + b_1x(1) + b_2x(0)\\
n &= 3,\ y(3) = b_0x(3) + b_1x(2) + b_2x(1) + b_3x(0)
\end{aligned}$$

In general, we have

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + b_3x(n-3), \qquad n \geq 3.$$

The graphical interpretation is illustrated in Figure 4.7.

As shown in Figure 4.7, the input sequence is flipped around (folded) and then shifted to the right to overlap with the filter coefficients. At each time instant, the output value is the sum of products of overlapped coefficients with the corresponding input data aligned below it. This flip-and-slide form of

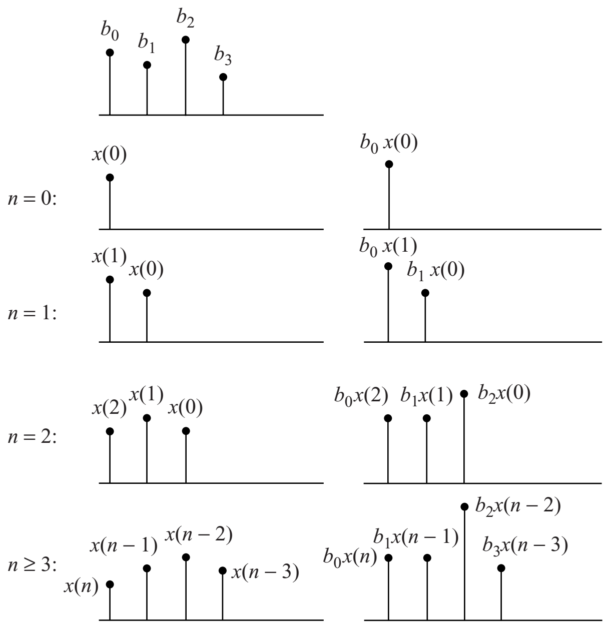


Figure 4.7 Graphical interpretation of linear convolution, $L = 4$

linear convolution can be illustrated in Figure 4.8. Note that shifting $x(-l)$ to the right is equivalent to shifting b_l to the left 1 unit at each sampling period.

At time $n = 0$, the input sequence is extended by padding $L - 1$ zeros to its right. The only nonzero product comes from b_0 multiplied with $x(0)$, which is time aligned. It takes the filter $L - 1$ iterations before it is completely overlapped with the input sequence. Therefore, the first $L - 1$ outputs correspond to the transient of the FIR filtering. After $n \geq L - 1$, the signal buffer of the FIR filter is full and the filter is in the steady state.

In FIR filtering, the coefficients are constants, but the data in the signal buffer (or tapped delay line) changes every sampling period, T. The signal buffer is refreshed in the fashion illustrated in Figure 4.9, where the oldest sample $x(n - L + 1)$ is discarded and the rest samples are shifted one location to the right in the buffer. A new sample (from ADC in real-time applications) is inserted to the memory location labeled as $x(n)$. This $x(n)$ at time n will become $x(n - 1)$ in the next sampling period, then $x(n - 2)$, etc., until it simply drops out off the end of the delay chain. The process of refreshing the signal buffer

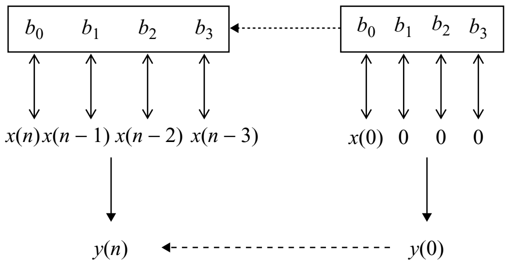


Figure 4.8 Flip-and-slide process of linear convolution

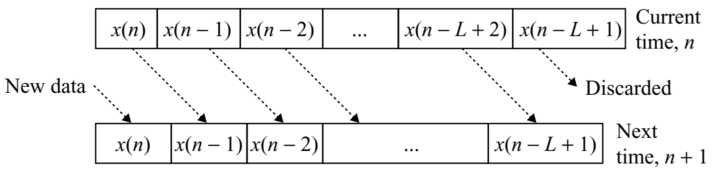


Figure 4.9 Signal buffer refreshing for FIR filtering

shown in Figure 4.9 requires intensive processing time if the data-move operations are not implemented by hardware.

The most efficient method for refreshing a signal buffer is to arrange the signal samples in a circular fashion as illustrated in Figure 4.10(a). Instead of shifting the data samples forward while holding the start address of buffer fixed as shown in Figure 4.9, the data samples in the circular buffer do not move but the buffer start address is updated backward (counterclockwise). The beginning of the signal sample, $x(n)$, is pointed by start-address pointer, and the previous samples are already loaded sequentially from that point in a clockwise direction. As we receive a new sample, it is placed at the position $x(n)$ and our filtering operation is performed. After calculating the output $y(n)$, the start pointer is moved counterclockwise one position to $x(n-L+1)$ and we wait for the next input signal. The next input at time $n+1$ is written to the $x(n-L+1)$ position and is referred as $x(n)$ for the next iteration. The circular buffer is very efficient because the update is carried out by adjusting the start-address pointer without physically shifting any data samples in memory.

Figure 4.10(b) shows a circular buffer for FIR filter coefficients. Circular buffer allows the coefficient pointer to wrap around when it reaches to the end of the coefficient buffer. That is, the pointer moves from b_{L-1} to b_0 such that the filtering will always start at the first coefficient.

4.2 Design of FIR Filters

The objective of designing FIR filter is to determine a set of filter coefficients that satisfies the given specifications. A variety of techniques have been developed for designing FIR filters. The Fourier series

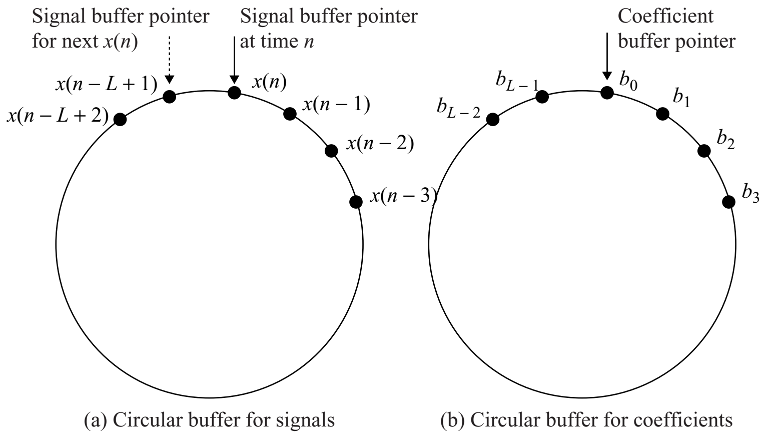

Signal buffer pointer for next x(n)
Signal buffer pointer at time n
Coefficient buffer pointer
x(n − L + 1)
x(n)
x(n − L + 2)
x(n − 1)
x(n − 2)
x(n − 3)
b_{L − 1}
b_0
b_{L − 2}
b_1
b_2
b_3
(a) Circular buffer for signals
(b) Circular buffer for coefficients

Figure 4.10 Circular buffers for FIR filter: (a) circular buffer for holding the signals. The start pointer to $x(n)$ is updated in the counterclockwise direction; (b) circular buffer for FIR filter coefficients, the pointer always pointing to b_0 at the beginning of filtering

method offers a simple way of computing FIR filter coefficients, thus can be used to explain the principles of FIR filter design.

4.2.1 Fourier Series Method

Fourier series method designs an FIR filter by calculating the impulse response of a filter that approximates the desired frequency response. Thus, it can be expanded in a Fourier series as

$$H(\omega) = \sum_{n=-\infty}^{\infty} h(n) e^{-j\omega n}, \tag{4.25}$$

where

$$h(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H(\omega) e^{j\omega n}\, d\omega, \quad -\infty \le n \le \infty. \tag{4.26}$$

Equation (4.26) shows that the impulse response $h(n)$ is double sided and has infinite length.

For a desired frequency response $H(\omega)$, the corresponding impulse response $h(n)$ (same as filter coefficients) can be calculated by evaluating the integral defined in Equation (4.26). A finite-duration impulse response $\{h'(n)\}$ can be simply obtained by truncating the ideal infinite-length impulse response defined in Equation (4.26). That is,

$$h'(n) = \begin{cases} h(n), & -M \le n \le M \\ 0, & \text{otherwise} \end{cases}. \tag{4.27}$$

Note that in this definition, we assume L to be an odd number.

A causal FIR filter can be derived by shifting the $h'(n)$ sequence to the right by M samples and reindexing the coefficients as

$$b'_l = h'(l - M), \qquad l = 0, 1, \ldots, 2M. \tag{4.28}$$

This FIR filter has $L(= 2M + 1)$ coefficients $b'_l, l = 0, 1, \ldots, L - 1$. The impulse response is symmetric about b'_M due to the fact that $h(-n) = h(n)$ is given in Equation (4.26). Therefore, the transfer function $B'(z)$ has a linear phase and a constant group delay.

Example 4.6: The ideal lowpass filter given in Figure 4.2(a) has frequency response

$$H(\omega) = \begin{cases} 1, & |\omega| \le \omega_c \\ 0, & \text{otherwise} \end{cases}. \tag{4.29}$$

The corresponding impulse response can be computed using Equation (4.26) as

$$\begin{aligned} h(n) &= \frac{1}{2\pi} \int_{-\pi}^{\pi} H(\omega) e^{j\omega n}\, d\omega = \frac{1}{2\pi} \int_{-\omega_c}^{\omega_c} e^{j\omega n}\, d\omega \\ &= \frac{1}{2\pi} \left[\frac{e^{j\omega n}}{jn} \right]_{-\omega_c}^{\omega_c} = \frac{1}{2\pi} \left[\frac{e^{j\omega_c n} - e^{-j\omega_c n}}{jn} \right] \\ &= \frac{\sin(\omega_c n)}{\pi n} = \frac{\omega_c}{\pi} \text{sinc}\left(\frac{\omega_c n}{\pi} \right), \end{aligned} \tag{4.30}$$

where the sinc function is defined as

$$\operatorname{sinc}(x) = \frac{\sin(\pi x)}{\pi x}.$$

By setting all impulse response coefficients outside the range $-M \leq n \leq M$ to zero, we obtain an FIR filter with the symmetry property. By shifting M units to the right, we obtain a causal FIR filter of finite length L with coefficients

$$b_l' = \left\{ \begin{array}{ll} \frac{\omega_c}{\pi}\operatorname{sinc}\left[\frac{\omega_c(l-M)}{\pi}\right], & 0 \leq l \leq L-1 \\ 0, & \text{otherwise} \end{array} \right\}. \tag{4.31}$$

Example 4.7: Design a lowpass FIR filter with the frequency response

$$H(f) = \begin{cases} 1, & 0 \leq f \leq 1\,\text{kHz} \\ 0, & 17\,\text{kHz} < f \leq 4\,\text{kHz} \end{cases},$$

where the sampling rate is 8 kHz. The impulse response is limited to 2.5 ms.

Since $2MT = 0.0025$ s and $T = 0.000125$ s, we need $M = 10$. Thus, the actual filter has 21 ($L = 2M + 1$) coefficients, and 1 kHz corresponds to $\omega_c = 0.25\pi$. From Equation (4.31), we have

$$b_l' = 0.25\operatorname{sinc}[0.25(l-10)], \quad l = 0, 1, \ldots, 20.$$

Example 4.8: Design a lowpass filter of cutoff frequency $\omega_c = 0.4\pi$ with filter length $L = 41$ and $L = 61$.

When $L = 41$, $M = (L-1)/2 = 20$. From Equation (4.31), the designed coefficients are

$b_l' = 0.4\operatorname{sinc}[0.4(l-20)], l = 0, 1, \ldots, 20.$

When $L = 61$, $M = (L-1)/2 = 30$. The coefficients become

$b_l' = 0.4\operatorname{sinc}[0.4(l-30)], l = 0, 1, \ldots, 30.$

These coefficients are computed and plotted in Figure 4.11 using the MATLAB script `example4_8.m`.

4.2.2 Gibbs Phenomenon

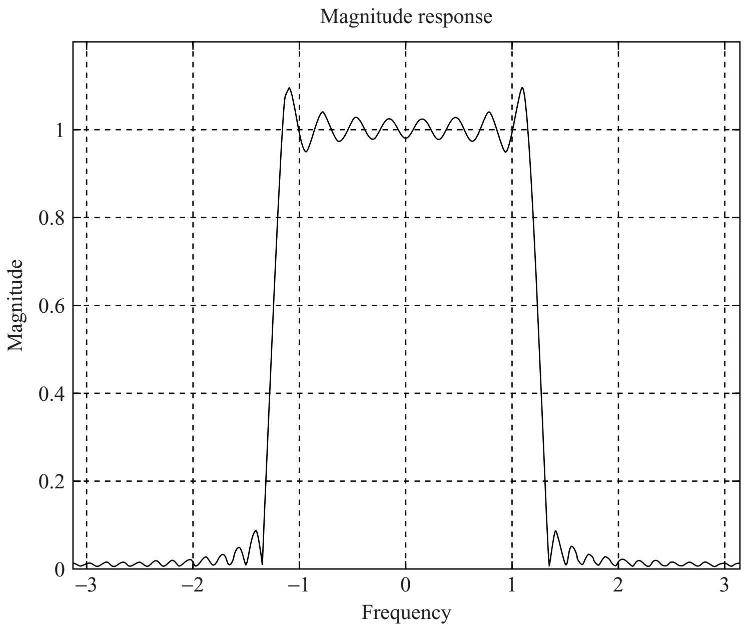
As shown in Figure 4.11, the FIR filter obtained by simply truncating the impulse response of the desired filter exhibits an oscillatory behavior (or ripples) in its magnitude response. As the length of the filter increases, the number of ripples in both passband and stopband increases, and the width of the ripple decreases. The largest ripple occurs near the transition discontinuity and their amplitude is independent of L.

The truncation operation described in Equation (4.27) can be considered as multiplication of the infinite-length sequence $h(n)$ by the rectangular window $w(n)$. That is,

$$h'(n) = h(n)w(n), \quad -\infty \leq n \leq \infty, \tag{4.32}$$

where the rectangular window $w(n)$ is defined as

$$w(n) = \begin{cases} 1, & -M \leq n \leq M \\ 0, & \text{otherwise} \end{cases}. \tag{4.33}$$



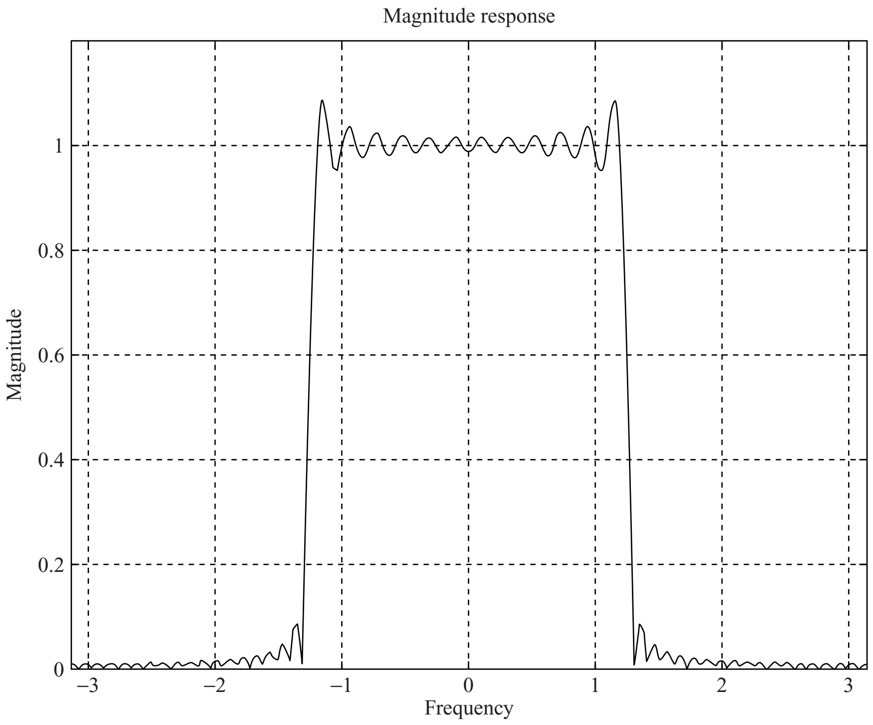


Figure 4.11 Magnitude responses of lowpass filters designed by Fourier series method: (top) $L = 41$; (bottom) $L = 61$

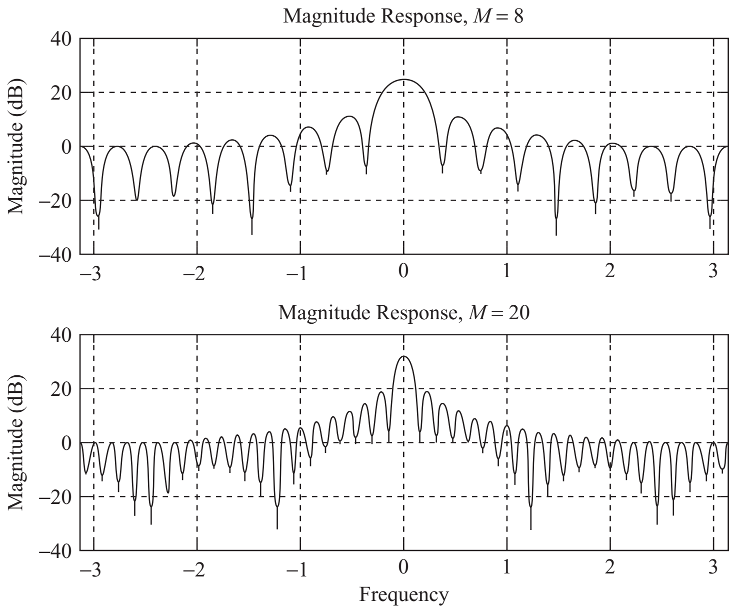


Figure 4.12 Magnitude responses of the rectangular window for $M = 8$ and 20

In order to approximate $H(\omega)$ very closely, we need the window function with infinite length.

Example 4.9: The oscillatory behavior of a truncated Fourier series representation of FIR filter, observed in Figure 4.11, can be explained by the frequency response of the rectangular window defined in Equation (4.33). It can be expressed as

$$W(\omega) = \sum_{n=-M}^{M} e^{-j\omega n} = \frac{\sin[(2M+1)\omega/2]}{\sin(\omega/2)}. \tag{4.34}$$

Magnitude responses of $W(\omega)$ for $M = 8$ and 20 are generated by MATLAB script `example4_9.m`. As illustrated in Figure 4.12, the magnitude response has a mainlobe centered at $\omega = 0$. All the other ripples are called the sidelobes. The magnitude response has the first zero at $\omega = 2\pi/(2M+1)$. Therefore, the width of the mainlobe is $4\pi/(2M+1)$. From Equation (4.34), it is easy to show that the magnitude of mainlobe is $|W(0)| = 2M + 1$. The first sidelobe is approximately located at frequency $\omega_1 = 3\pi/(2M+1)$ with the magnitude of $|W(\omega_1)| \approx 2(2M+1)/3\pi$ for $M >> 1$. The ratio of the mainlobe magnitude to the first sidelobe magnitude is

$$\left|\frac{W(0)}{W(\omega_1)}\right| \approx \frac{3\pi}{2} = 13.5\,\text{dB}.$$

As ω increases toward π, the denominator grows larger. This results in a damped function shown in Figure 4.12. As M increases, the width of the mainlobe decreases.

The rectangular window has an abrupt transition to zero outside the range $-M \le n \le M$, which causes the Gibbs phenomenon in the magnitude response. The Gibbs phenomenon can be reduced either by using a window that tapers smoothly to zero at each end or by providing a smooth transition from the passband to the stopband. A tapered window will reduce the height of the sidelobes and increase the width of the mainlobe, resulting in a wider transition at the discontinuity. This phenomenon is often referred to as leakage or smearing.

4.2.3 Window Functions

A large number of tapered windows have been developed and optimized for different applications. In this section, we restrict our discussion to four commonly used windows of length $L = 2M + 1$. That is, $w(n)$, where $n = 0, 1, \ldots, L - 1$ and is symmetric about its middle, $n = M$. Two parameters that predict the performance of the window in FIR filter design are its mainlobe width and the relative sidelobe level. To ensure a fast transition from the passband to the stopband, the window should have a small mainlobe width. On the other hand, to reduce the passband and stopband ripples, the area under the sidelobes should be small. Unfortunately, there is a trade-off between these two requirements.

The Hann (Hanning) window function is one period of the raised cosine function defined as

$$w(n) = 0.5\left[1 - \cos\left(\frac{2\pi n}{L-1}\right)\right], \qquad n = 0, 1, \ldots, L - 1. \tag{4.35}$$

The window coefficients can be generated by the MATLAB function

```
w = hanning(L);
```

which returns the L-point Hanning window function in array `w`. The MATLAB script `hanWindow.m` generates window coefficients $w(n)$, $n = 1, \ldots, L$. For a large L, the peak-to-sidelobe ratio is approximately 31 dB, an improvement of 17.5 dB over the rectangular window.

The Hamming window function is defined as

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{L-1}\right), \qquad n = 0, 1, \ldots, L - 1, \tag{4.36}$$

which also corresponds to a raised cosine, but with different weights for the constant and cosine terms. The Hamming function tapers the end values to 0.08. MATLAB provides the Hamming window function

```
w = hamming(L);
```

The Hamming window function and its magnitude response generated by MATLAB script `hamWindow.m` are shown in Figure 4.13. The mainlobe width is about the same as the Hanning window, but this window has an additional 10 dB of stopband attenuation (41 dB). The Hamming window provides low ripple over the passband and good stopband attenuation, and it is usually more appropriate for a lowpass filter design.

Example 4.10: Design a lowpass FIR filter of cutoff frequency $\omega_c = 0.4\pi$ and order $L = 61$ using the Hamming window. Using the MATLAB script (`example4_10.m`) similar to the one used in Example 4.8, we plot the magnitude responses of designed filters in Figure 4.14 using both rectangular and Hamming windows. We observe that the ripples produced by the rectangular window design are virtually eliminated from the Hamming window design. The trade-off of eliminating the ripples is increasing transition width.

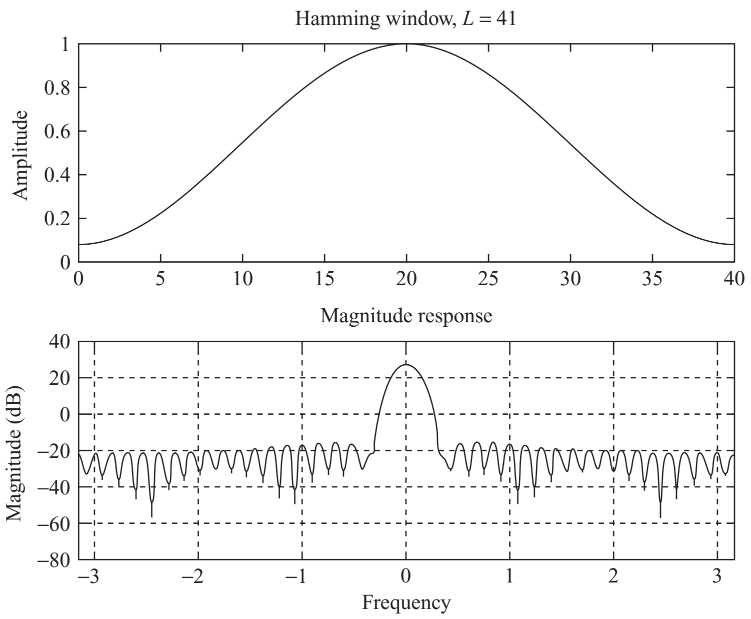


Figure 4.13 Hamming window function (top) and its magnitude response (bottom), $L = 41$

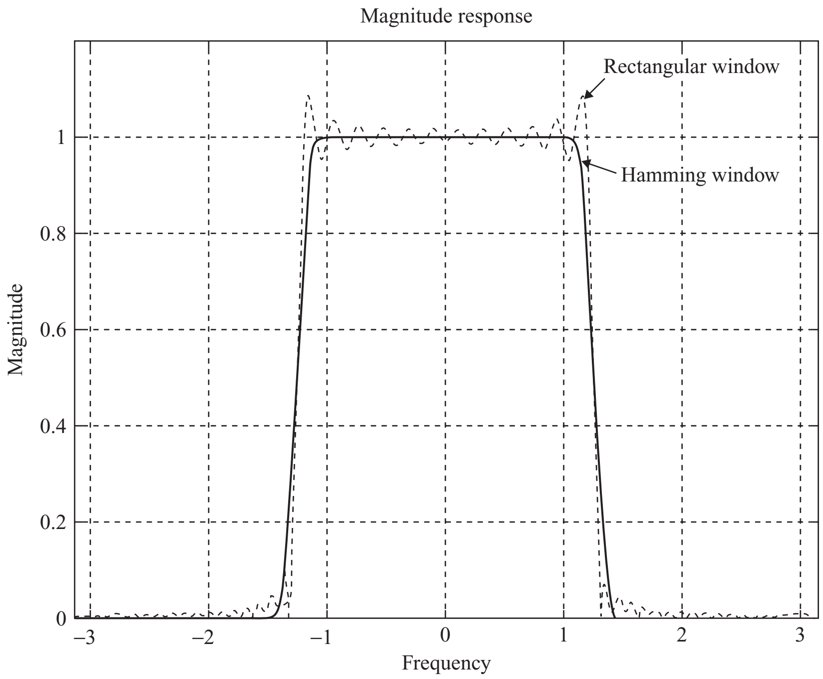


Figure 4.14 Magnitude response of lowpass filter using Hamming window, $L = 61$

The Blackman window function is defined as

$$w(n) = 0.42 - 0.5\cos\left(\frac{2\pi n}{L-1}\right) + 0.08\cos\left(\frac{4\pi n}{L-1}\right), \qquad n = 0,\ 1, \ldots, L-1. \tag{4.37}$$

This function is also supported by the MATLAB function

```
w = blackman(L);
```

This window can be generated and its magnitude response can be plotted by MATLAB script `blackmanWindow.m`. The addition of the second cosine term in Equation (4.37) has the effect of increasing the width of the mainlobe (50%), but at the same time improving the peak-to-sidelobe ratio to about 57 dB. The Blackman window provides 74 dB of stopband attenuation, but with a transition width six times that of the rectangular window.

The Kaiser window is defined as

$$w(n) = \frac{I_0\left(\beta\sqrt{1-(n-M)^2/M^2}\right)}{I_0(\beta)}, \qquad n = 0,\ 1, \ldots, L-1, \tag{4.38}$$

where β is an adjustable (shape) parameter and

$$I_0(\beta) = \sum_{k=0}^{\infty}\left[\frac{(\beta/2)^k}{k!}\right]^2 \tag{4.39}$$

is the zero-order modified Bessel function of the first kind. MATLAB provides Kaiser window

```
kaiser(L,beta);
```

The window function and its magnitude response for $L = 41$ and $\beta = 8$ can be displayed using the MATLAB script `kaiserWindow.m`. The Kaiser window is nearly optimum in the sense of having the largest energy in the mainlobe for a given peak sidelobe level. Providing a large mainlobe width for the given stopband attenuation implies the sharpness transition width. This window can provide different transition widths for the same L by choosing the parameter β to determine the trade-off between the mainlobe width and the peak sidelobe level.

The procedure of designing FIR filters using Fourier series and windows is summarized as follows:

1. Determine the window type that will satisfy the stopband attenuation requirements.
2. Determine the window size L based on the given transition width.
3. Calculate the window coefficients $w(l)$, $l = 0, 1, \ldots, L-1$.
4. Generate the ideal impulse response $h(n)$ using Equation (4.26) for the desired filter.
5. Truncate the ideal impulse response of infinite length using Equation (4.27) to obtain $h'(n)$, $-M \le n \le M$.
6. Make the filter causal by shifting the result M units to the right using Equation (4.28) to obtain b'_l, $l = 0,\ 1, \ldots, L-1$.

7. Multiply the window coefficients obtained in Step 3 and the impulse response coefficients obtained in Step 6. That is,

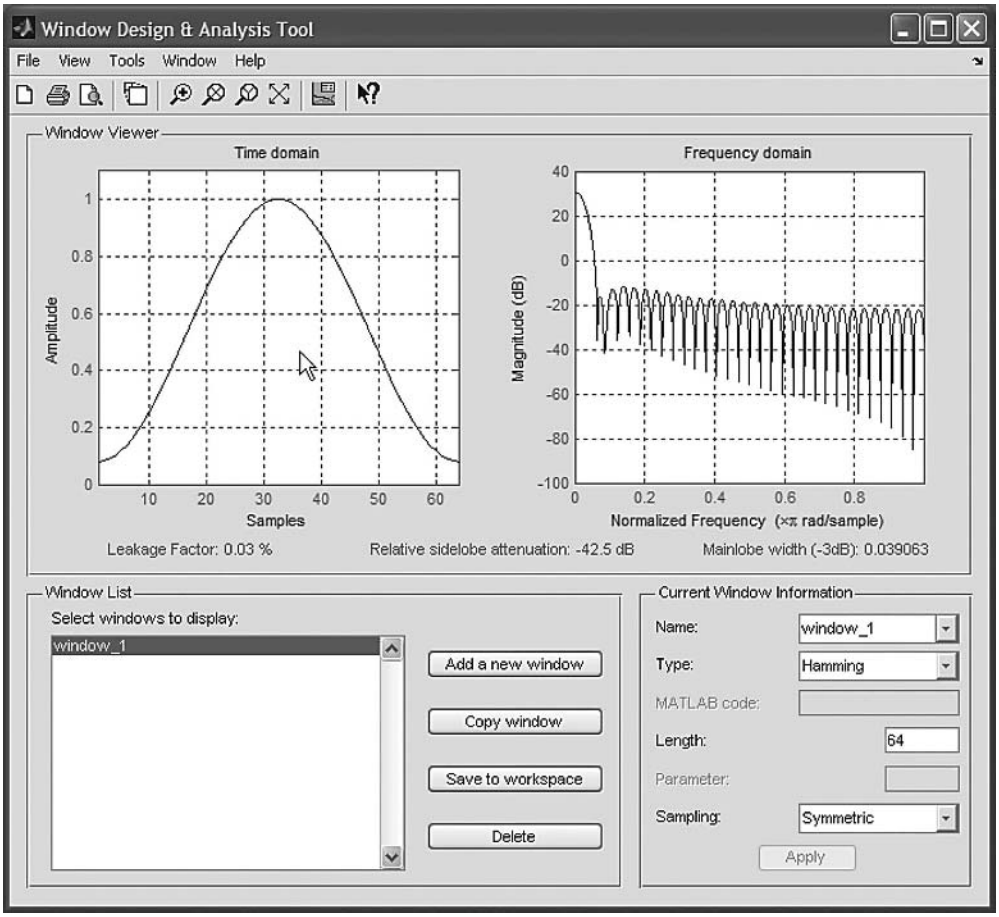
$$b_l = b'_l w(l), \qquad l = 0,\ 1, \ldots, L-1. \tag{4.40}$$

Applying a window to an FIR filter's impulse response has the effect of smoothing the resulting filter's magnitude response. A symmetric window will preserve a symmetric FIR filter's linear-phase response.

MATLAB provides a GUI tool called Window Design & Analysis Tool (WinTool) that allows users to design and analyze windows. It can be activated by entering the following command in MATLAB command window:

```
wintool
```

It opens with a default 64-point Hamming window as shown in Figure 4.15. WinTool has three panels: **Window Viewer**, **Window List**, and **Current Window Information**. **Window Viewer** displays the time-domain (left) and frequency-domain (right) representations of the selected window(s). Three


Window Design & Analysis Tool
File View Tools Window Help
Window Viewer
Time domain
Amplitude
Samples
Frequency domain
Magnitude (dB)
Normalized Frequency (×π rad/sample)
Window List
Select windows to display:
window_1
Add a new window
Copy window
Save to workspace
Delete
Current Window Information
Apply

Leakage Factor: 0.03 %
Relative sidelobe attenuation: -42.5 dB
Mainlobe width (-3dB): 0.039063

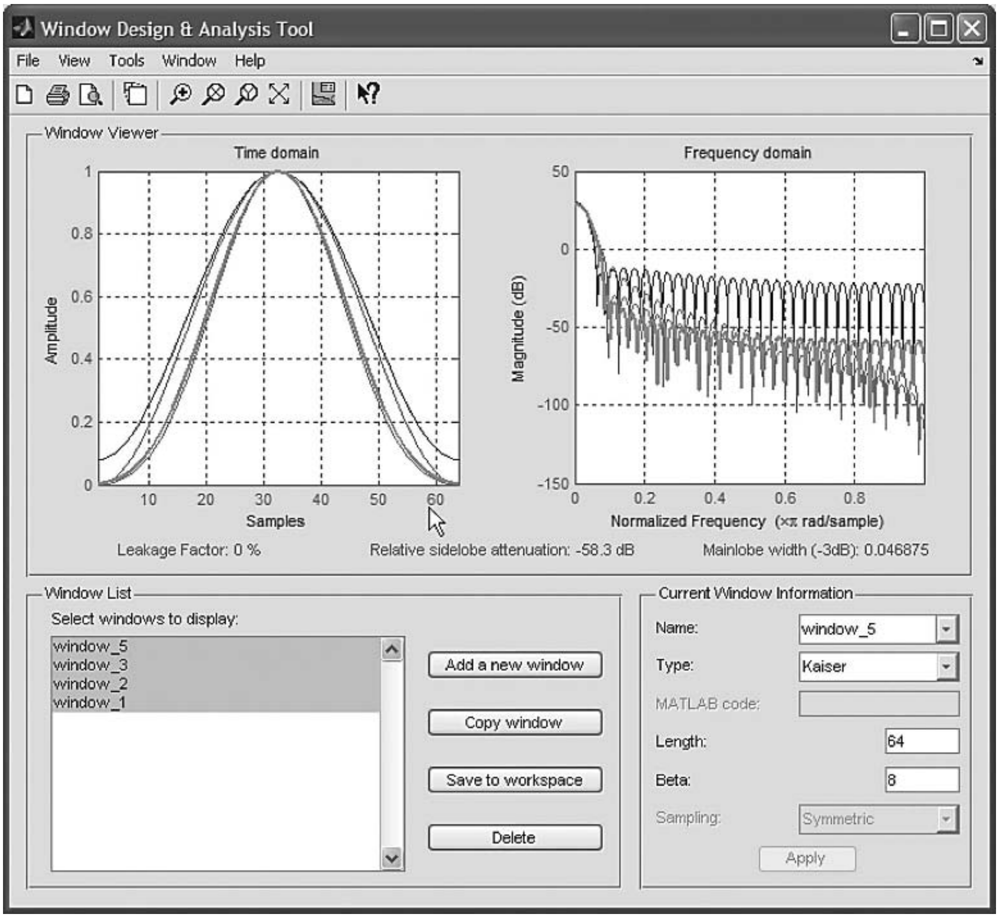
Name: window_1
Type: Hamming
MATLAB code:
Length: 64
Parameter:
Sampling: Symmetric

Figure 4.15 Default window for WinTool

measurements are displayed under the time-domain and frequency-domain plots: (1) *Leakage factor* indicates the ratio of power in the sidelobes to the total window power. (2) *Relative sidelobe attenuation* shows the difference in height from the mainlobe peak to the highest sidelobe peak. (3) *Mainlobe width* (−3 dB) shows the width of the mainlobe at 3 dB below the mainlobe peak.

Window List panel lists the windows available for display in the **Window Viewer**. Highlight one or more windows to display them. There are four **Window List** buttons: (1) **Add a New Window**, (2) **Copy Window**, (3) **Save to Workspace**, and (4) **Delete**. Each window is defined by the parameters in the **Current Window Information** panel. We can change the current window's characteristics by changing its parameters. From the **Type** pull-down menu, we can choose different windows available in the *Signal Processing Toolbox*. From the **Length** box, we can specify number of samples.

With this tool, we can evaluate different windows. For example, we can click **Add a New Window** button and then select a new window Hann in the **Type** pull-down menu. We repeat this process for Blackman and Kaiser windows. We then highlight all four (including the default Hamming) windows in the **Select Windows to Display** box. As shown in Figure 4.16, we have four window functions and magnitude responses displayed in the same graph for comparison.



Leakage Factor: 0 %
Relative sidelobe attenuation: -58.3 dB
Mainlobe width (-3dB): 0.046875

Select windows to display:
window_5
window_3
window_2
window_1

Name: window_5
Type: Kaiser
MATLAB code:
Length: 64
Beta: 8
Sampling: Symmetric

Figure 4.16 Comparison of Hamming, Hann, Blackman, and Kaiser windows

4.2.4 Design of FIR Filters Using MATLAB

Filter design algorithms use iterative optimization techniques to minimize the error between the desired and actual frequency responses. The most widely used algorithm is the Parks–McClellan algorithm for designing the optimum linear-phase FIR filter. This algorithm spreads out the error to produce equal-magnitude ripples. In this section, we consider only the design methods and filter functions available in MATLAB *Signal Processing Toolbox*, which are summarized in Table 4.1, and the MATLAB *Filter Design Toolbox* provides more advanced FIR filter design methods.

As an example, `fir1` and `fir2` functions design FIR filters using windowed Fourier series method. The function `fir1` designs FIR filters using the Hamming window as

```
    b = fir1(L, Wn);
```

where `Wn` is the normalized cutoff frequency between 0 and 1. The function `fir2` designs an FIR filter with arbitrarily shaped magnitude response as

```
    b = fir2(L, f, m);
```

where the frequency response is specified by vectors `f` and `m` that contain the frequency and magnitude, respectively. The frequencies in `f` must be between 0 < `f` <1 in increasing order.

A more efficient Remez algorithm designs the optimum linear-phase FIR filters based on the Parks–McClellan algorithm. This algorithm uses the Remez exchange and Chebyshev approximation theory to design a filter with an optimum fit between the desired and actual frequency responses. This `remez` function has syntax as follows:

```
    b = remez(L, f, m);
```

Example 4.11: Design a linear-phase FIR bandpass filter of length 18 with a passband from normalized frequency 0.4–0.6. This filter can be designed and displayed using the following MATLAB script (`example4_11.m`):

```
    f = [0  0.3  0.4  0.6  0.7  1];
    m = [0  0    1    1    0    0];
    b = remez(17, f, m);
    [h, omega] = freqz(b, 1, 512);
    plot(f, m, omega/pi, abs(h));
```

The desired and obtained magnitude responses are shown in Figure 4.17.

Table 4.1 List of FIR filter design methods and functions available in MATLAB

Design method	Filter function	Description
Windowing	`fir1`, `fir2`, `kaiserord`	Truncated Fourier series with windowing methods
Multiband with transition bands	`firls`, `firpm`, `firpmord`	Equiripple or least squares approach
Constrained least squares	`fircls`, `fircls1`	Minimize squared integral error over entire frequency range
Arbitrary response	`cfirpm`	Arbitrary responses

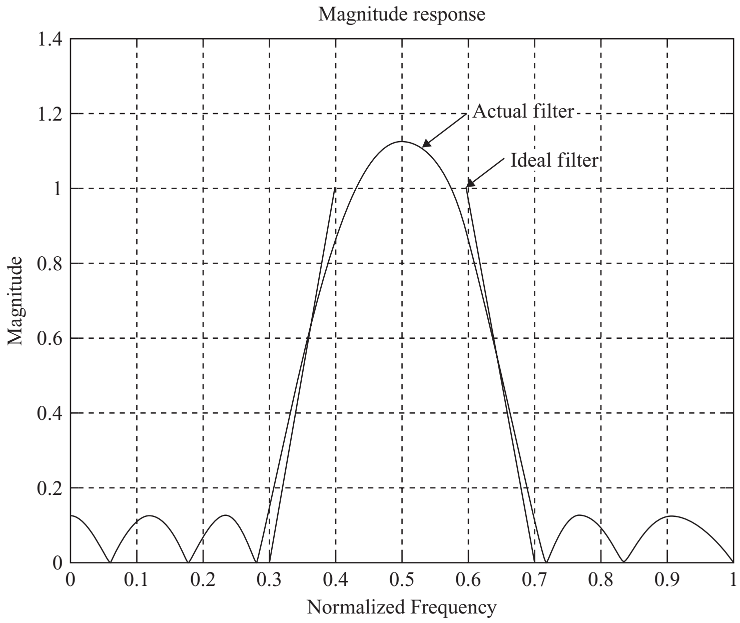


Figure 4.17 Magnitude responses of the desired and actual FIR filters

4.2.5 Design of FIR Filters Using FDATool

The Filter Design and Analysis Tool (FDATool) is a graphical user interface (GUI) for designing, quantizing, and analyzing digital filters. It includes a number of advanced filter design techniques and supports all the filter design methods in the *Signal Processing Toolbox*. This tool has the following functions:

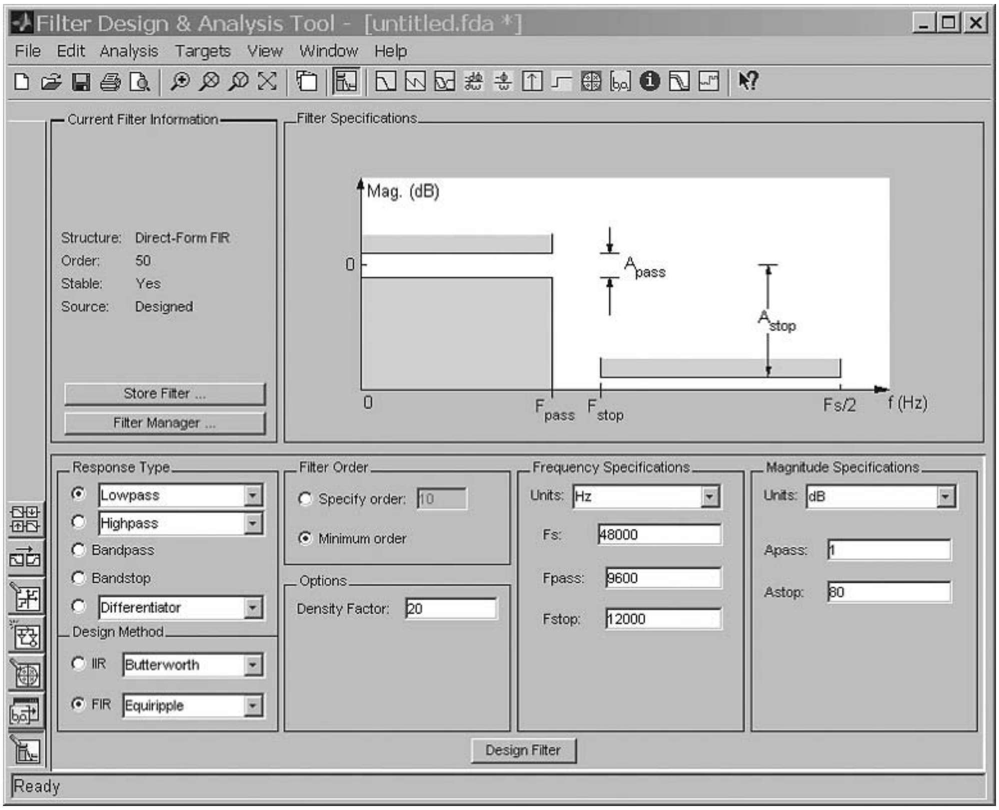
1. designing filters by setting filter specifications;
2. analyzing designed filters;
3. converting filters to different structures; and
4. quantizing and analyzing quantized filters.

Note that the last feature is available only with the *Filter Design Toolbox*. In this section, we introduce the FDATool for designing and quantizing FIR filters.

We can open the FDATool by typing

```
fdatool
```

at the MATLAB command window. The **Filter Design & Analysis Tool** window is shown in Figure 4.18. We can choose from several response types: **Lowpass**, **Highpass**, **Bandpass**, **Bandstop**, and **Differentiator**. For example, to design a bandpass filter, select the **Radio** button next to **Bandpass** in the **Response Type** region on the GUI. It has multiple options for **Lowpass**, **Highpass**, and **Differentiator** types. More response types are available with the *Filter Design Toolbox*.



Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Response Type: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator

Design Method: IIR Butterworth [✓ FIR Equiripple]

Filter Order: Specify order: 10 [✓ Minimum order]

Options: Density Factor: 20

Frequency Specifications: Units: Hz; Fs: 48000; Fpass: 9600; Fstop: 12000

Magnitude Specifications: Units: dB; Apass: 1; Astop: 80

Ready

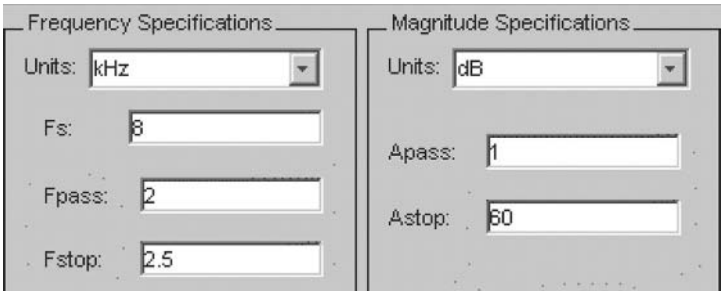
Figure 4.18 FDATool window

It is important to compare the **Filter Specifications** region in Figure 4.18 with Figure 4.4. The parameters F_{pass}, F_{stop}, A_{pass}, and A_{stop} are corresponding to ω_{p}, ω_{s}, A_{p}, and A_{s}, respectively. These parameters can be entered in the **Frequency Specifications** and **Magnitude Specifications** regions. The frequency units are Hz (default), kHz, MHz, or GHz, and the magnitude options are **dB** (default) or **Linear**.

Example 4.12: Design a lowpass FIR filter with the following specifications:

sampling frequency (f_{s}) = 8 kHz;
passband cutoff frequency (ω_{p}) = 2 kHz;
stopband cutoff frequency (ω_{s}) = 2.5 kHz;
passband ripple (A_{p}) = 1 dB; and
stopband attenuation (A_{s}) = 60 dB.

We can easily design this filter by entering parameters in **Frequency Specifications** and **Magnitude Specifications** regions as shown in Figure 4.19. Pressing **Design Filter** button computes the filter coefficients. The **Filter Specifications** region will show the **Magnitude Response (dB)** (see Figure 4.20). We can analyze different characteristics of the designed filter by clicking the **Analysis** menu. For example, selecting the **Impulse Response** available in the menu opens a new **Impulse Response** window to display the designed FIR filter coefficients as shown in Figure 4.21.



Frequency Specifications

Units: kHz

Fs: 8

Fpass: 2

Fstop: 2.5

Magnitude Specifications

Units: dB

Apass: 1

Astop: 60

Figure 4.19 Frequency and magnitude specifications for a lowpass filter

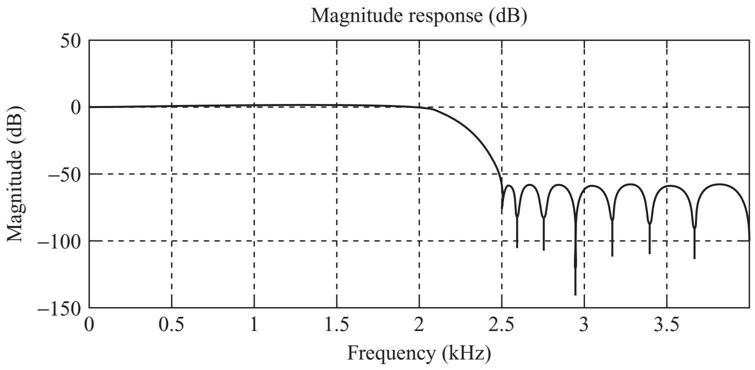


Figure 4.20 Magnitude response of the designed lowpass filter

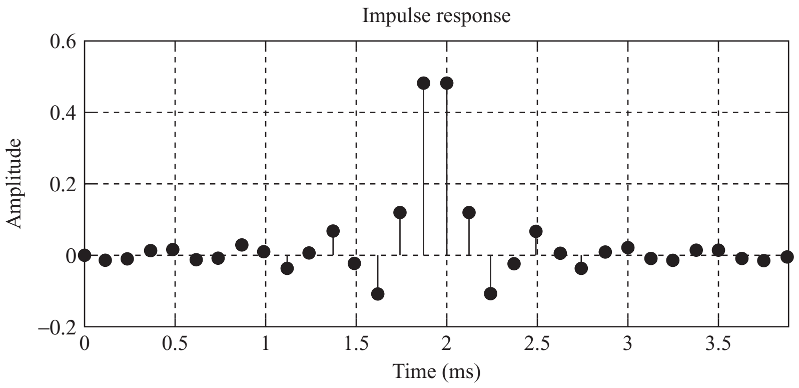
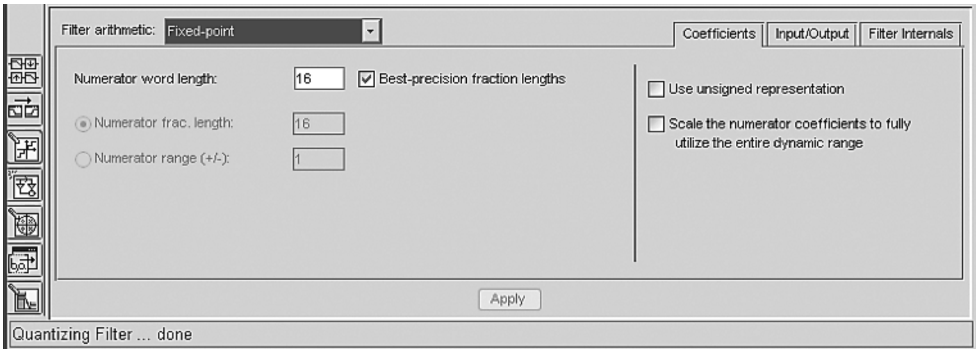


Figure 4.21 Impulse responses (filter coefficients) of the designed filter



Filter arithmetic: Fixed-point

Numerator word length: 16

[✓] Best-precision fraction lengths

(●) Numerator frac. length: 16

( ) Numerator range (+/-): 1

[ ] Use unsigned representation

[ ] Scale the numerator coefficients to fully utilize the entire dynamic range

Quantizing Filter ... done

Figure 4.22 Setting fixed-point quantization parameters in the FDATool

We have two options for determining the filter order: we can specify the filter order by **Specify Order**, or use the default **Minimum Order**. In Example 4.12, we use the default minimum order, and the order (31) is shown in the **Current Filter Information** region. Note that order = 31 means the length of FIR filter is $L = 32$, which is shown in Figure 4.21 with 32 coefficients.

Once the filter has been designed (using 64-bit double-precision floating-point arithmetic and representation) and verified, we can turn on the quantization mode by clicking the **Set Quantization Parameters** button on the side bar shown in Figure 4.18. The bottom-half of the FDATool window will change to a new pane with the default **Double-Precision Floating-Point** as shown in the **Filter Arithmetic** menu. The **Filter Arithmetic** option allows users to quantize the designed filter and analyze the effects with different quantization settings. When the user has chosen an arithmetic setting (single-precision floating-point or fixed-point), FDATool quantizes the current filter according to the selection and updates the information displayed in the analysis area. For example, to enable the fixed-point quantization settings in the FDATool, select **Fixed-Point** from the pull-down menu. The quantization options appear in the lower pane of the FDATool window as shown in Figure 4.22.

As shown in Figure 4.22, there are three tabs in the dialog window for user to select quantization tasks from the FDATool:

1. **Coefficients** tab defines the coefficient quantization.

2. **Input/Output** tab quantizes the input and output signals for the filter.

3. **Filter Internals** tab sets a variety of options for the arithmetic.

After setting the proper options for the desired filter, click **Apply** to start the quantization processes.

The **Coefficients** tab is the default active pane. The filter type and structure determine the available options. **Numerator Wordlength** sets the wordlength used to represent coefficients of FIR filters. Note that the **Best-Precision Fraction Lengths** box is also checked and the **Numerator Wordlength** box is set to 16 by default. We can uncheck the **Best-Precision Fraction Lengths** box to specify **Numerator Frac. Length** or **Numerator Range** (+/−).

The **Filter Internals** tab as shown in Figure 4.23 specifies how the quantized filter performs arithmetic operations. **Round towards** options,**Ceiling** (round up), **Nearest**, **Nearest (convergent)**, **Zero**, or **Floor** (round down), set a rounding mode that the filter will be used to quantize the numeric values. **Overflow Mode** options, **Wrap** and **Saturate**, set to overflow conditions in fixed-point arithmetic. **Product mode**

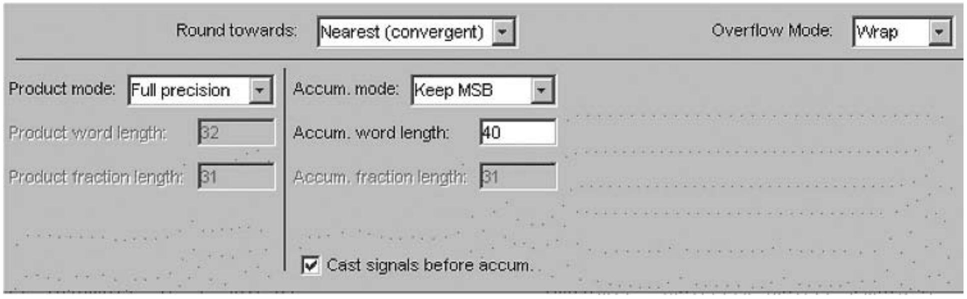

Round towards: Nearest (convergent)
Overflow Mode: Wrap
Product mode: Full precision
Product word length: 32
Product fraction length: 31
Accum. mode: Keep MSB
Accum. word length: 40
Accum. fraction length: 31
[✓] Cast signals before accum.

Figure 4.23 Setting filter arithmetic operations in the FDATool

options, **Full precision**, **Keep MSB**, **Keep LSB**, or **Specify all** (set the fraction length), determine how the filter handles the output of the multiplication operations. The **Accum. mode** option determines how the accumulator stores its output values.

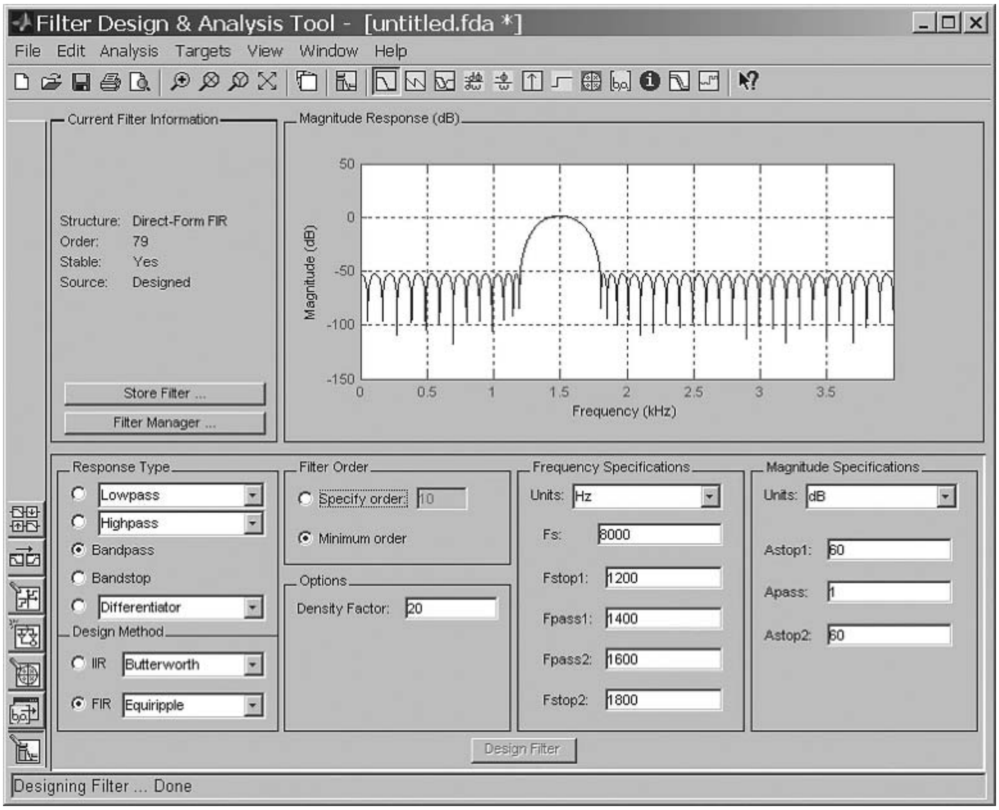
Example 4.13: Design a bandpass FIR filter for a 16-bit fixed-point DSP processor with the following specifications:

Sampling frequency = 8000 Hz.
Lower stopband cutoff frequency (F_{stop1}) = 1200 Hz.
Lower passband cutoff frequency (F_{pass1}) = 1400 Hz.
Upper passband cutoff frequency (F_{pass2}) = 1600 Hz.
Upper stopband cutoff frequency (F_{stop2}) = 1800 Hz.
Passband ripple = 1 dB.
Stopband (both lower and upper) attenuation = 60 dB.

After entering these parameters in the **Frequency Specifications** and **Magnitude Specifications** regions and clicking **Design Filter**, Figure 4.24 will be displayed. Click the **Set Quantization Parameters** button to switch to quantization mode and open the quantization panel. Selecting the **Fixed-point** option from the **Filter arithmetic** pull-down menu, the analysis areas will show the magnitude responses for both the designed filter and the fixed-point quantized filter. The default settings in **Coefficients**, **Input/Output**, and **Filter Internals Taps** are used.

We can export filter coefficients to MATLAB workspace to a coefficient file or MAT-file. To save the quantized filter coefficients as a text file, select **Export** from the **File** menu on the toolbar. When the **Export** dialog box appears, select **Coefficient File (ASCII)** from the **Export to** menu and choose **Decimal**, **Hexadecimal**, or **Binary** from the **Format** options. After clicking the **OK** button, the **Export Filter Coefficients to .FCF File** dialog box will appear. Enter a filename and click the **Save** button.

To create a C header file containing filter coefficients, select **Generate C header** from the **Targets** menu. For an FIR filter, variable used in C header file are for numerator name and length. We can use the default variable names `B` and `BL` as shown in Figure 4.25 in the C program, or change them to match the variable names defined in the C program that will include this header file. As shown in the figure, we can use the default **Signed 16-Bit Integer with 16-Bit Fractional Length**, or select **Export as** and choose the desired data type. Clicking **Generate** button opens **Generate C Header** dialog box. Enter the filename and click **Save** to save the file.


Filter Design & Analysis Tool - [untitled.fda *]
File Edit Analysis Targets View Window Help
Current Filter Information
Store Filter ...
Filter Manager ...
Magnitude Response (dB)
Magnitude (dB)
Frequency (kHz)
Design Filter

Structure: Direct-Form FIR
Order: 79
Stable: Yes
Source: Designed

Response Type: Lowpass, Highpass, [✓ Bandpass], Bandstop, Differentiator

Design Method: IIR Butterworth, [✓ FIR Equiripple]

Filter Order: Specify order: 10, [✓ Minimum order]

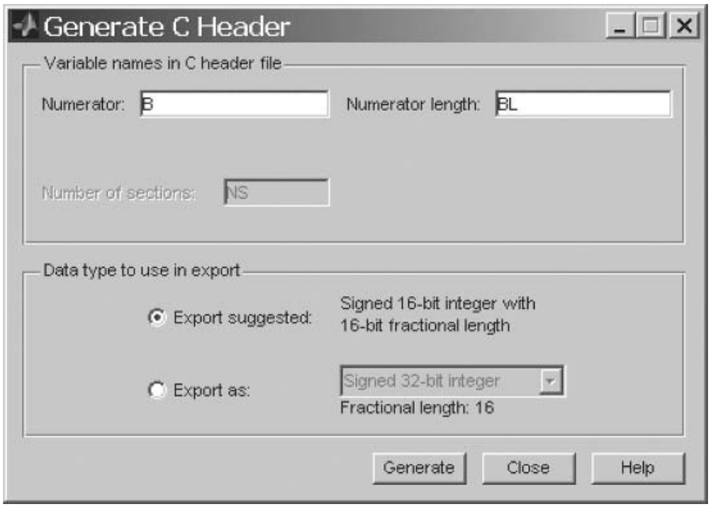
Options: Density Factor: 20

Frequency Specifications: Units: Hz; Fs: 8000; Fstop1: 1200; Fpass1: 1400; Fpass2: 1600; Fstop2: 1800

Magnitude Specifications: Units: dB; Astop1: 60; Apass: 1; Astop2: 60

Designing Filter ... Done

Figure 4.24 FDATool window for designing a bandpass filter



Variable names in C header file

Numerator: B
Numerator length: BL
Number of sections: NS

Data type to use in export

[✓ Export suggested:] Signed 16-bit integer with 16-bit fractional length

Export as: Signed 32-bit integer
Fractional length: 16

Figure 4.25 Generate C header dialog box

Example 4.14: We continue the Example 4.13 by saving the quantized 16-bit FIR filter coefficients in a file named as `Bandpass1500FIR.h`. The parameters and filter coefficients saved in the header file are shown as follows:

```
const int BL = 80;
const int 16_T B[80] = {
   79,    -48,   -126,    -86,     71,    155,     34,   -148,   -149,
   28,    135,     59,    -24,     23,     20,   -188,   -296,    101,
  674,    492,   -614,  -1321,   -315,   1563,   1806,   -480,  -2725,
 1719,   1886,   3635,    784,  -3559,  -3782,    931,   4906,   2884,
-2965,  -5350,  -1080,   4634,   4634,  -1080,  -5350,  -2965,   2884,
 4906,    931,  -3782,  -3559,    784,   3635,   1886,  -1719,  -2725,
 -480,   1806,   1563,   -315,  -1321,   -614,    492,    674,    101,
 -296,   -188,     20,     23,    -24,     59,    135,     28,   -149,
 -148,     34,    155,     71,    -86,   -126,    -48,     79
};
```

If the TMS320C5000 CCS is also installed on the computer, the **Targets** pull-down menu has additional option called `Composer Studio (tm) IDE`. Selecting this option, the **Export to Code Composer Studio (R) IDE** dialog box appears as shown in Figure 4.26. Comparing with Figure 4.25, we have additional options in **Export mode**: **C Header File** or **Write Directly to Memory**. In addition, we can select target DSP board such as the C5510 DSK. The MATLAB connects with DSK via MATLAB Link for CCS. This useful feature can simplify the DSP development and testing procedures by combining MATLAB functions with DSP processors. The MATLAB Link for CCS will be introduced in Chapter 9.

4.3 Implementation Considerations

In this section, we will consider finite-wordlength effects of digital FIR filters, and discuss the software implementation using MATLAB and C to illustrate some important issues.

4.3.1 Quantization Effects in FIR Filters

Consider the FIR filter given in Equation (3.22). The filter coefficients, b_l, are determined by a filter design package such as MATLAB. These coefficients are usually represented by double-precision floating-point numbers and have to be quantized for implementation on a fixed-point processor. The filter coefficients are quantized and analyzed during the design process. If it no longer meets the given specifications, we shall optimize, redesign, restructure, and/or use more bits to satisfy the specifications.

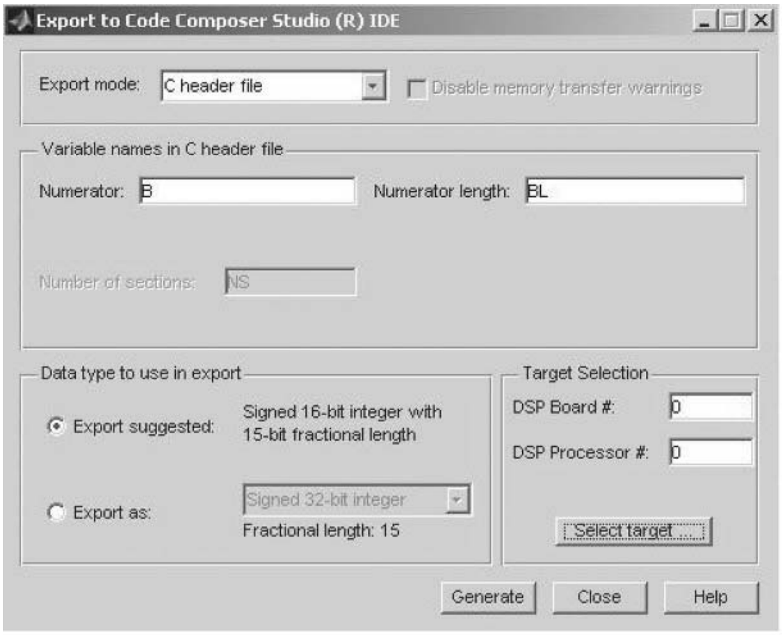
Let b'_l denote the quantized values corresponding to b_l. As discussed in Chapter 3, the nonlinear quantization can be modeled as a linear operation expressed as

$$b'_l = Q[b_l] = b_l + e(l), \tag{4.41}$$

where $e(l)$ is the quantization error and can be assumed as a uniformly distributed random noise of zero mean.

The frequency response of the actual FIR filter with quantized coefficients b'_l can be expressed as

$$B'(\omega) = B(\omega) + E(\omega), \tag{4.42}$$


Export to Code Composer Studio (R) IDE

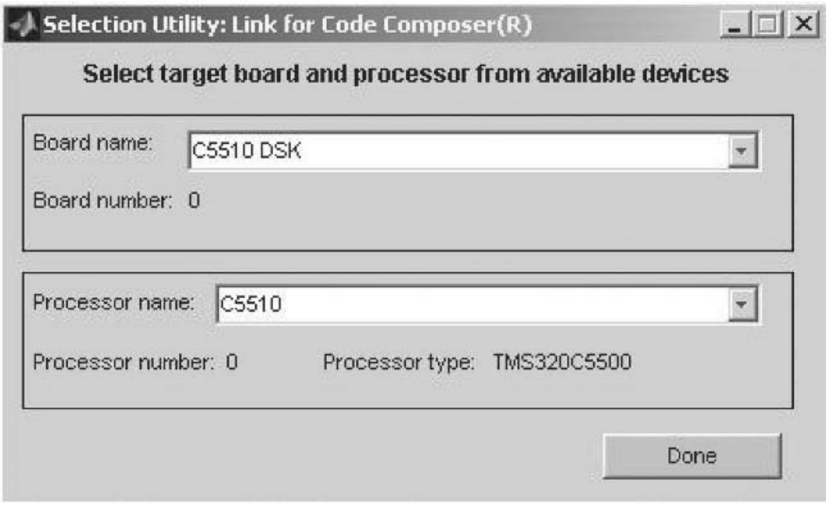
Export mode: C header file
Disable memory transfer warnings

Variable names in C header file
Numerator: B
Numerator length: BL
Number of sections: NS

Data type to use in export
Export suggested: Signed 16-bit integer with 15-bit fractional length
Export as: Signed 32-bit integer
Fractional length: 15

Target Selection
DSP Board #: 0
DSP Processor #: 0

(a) FDA Tool exports to CCS dialog box.


Selection Utility: Link for Code Composer(R)

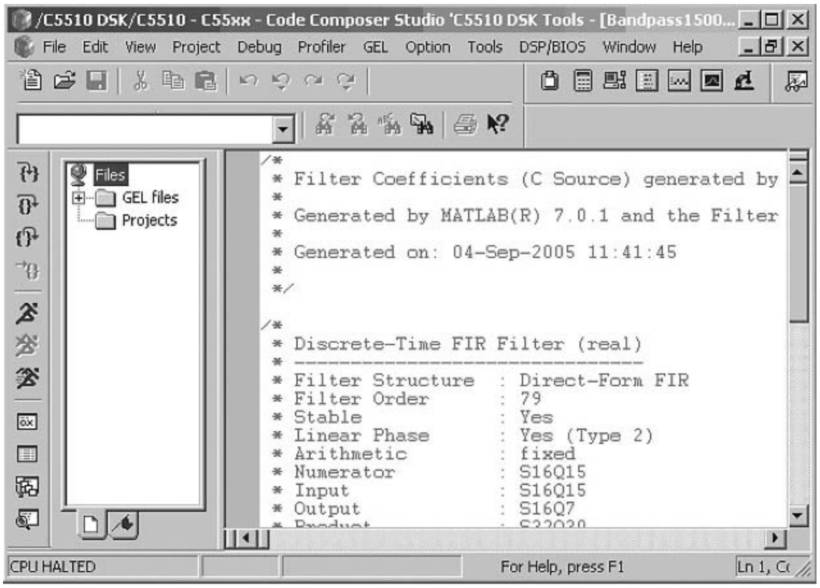
Select target board and processor from available devices

Board name: C5510 DSK
Board number: 0

Processor name: C5510
Processor number: 0
Processor type: TMS320C5500

(b) Link for CCS target selection dialog box.

Figure 4.26 FDATool exports to CCS: (a) FDATool exports to CCS dialog box; (b) link for CCS target selection dialog box; (c) C5510 DSK linked with MATLAB


/C5510 DSK/C5510 - C55xx - Code Composer Studio 'C5510 DSK Tools - [Bandpass1500...
File Edit View Project Debug Profiler GEL Option Tools DSP/BIOS Window Help
Files
GEL files
Projects
CPU HALTED
For Help, press F1
Ln 1, C

```
/*
 * Filter Coefficients (C Source) generated by
 *
 * Generated by MATLAB(R) 7.0.1 and the Filter
 *
 * Generated on: 04-Sep-2005 11:41:45
 *
 */

/*
 * Discrete-Time FIR Filter (real)
 * -------------------------------
 * Filter Structure  : Direct-Form FIR
 * Filter Order      : 79
 * Stable            : Yes
 * Linear Phase      : Yes (Type 2)
 * Arithmetic        : fixed
 * Numerator         : S16Q15
 * Input             : S16Q15
 * Output            : S16Q7
```

(c) C5510 DSK linked with MATLAB

Figure 4.26 (*Continued*)

where

$$E(\omega) = \sum_{l=0}^{L-1} e(l)e^{-j\omega l} \tag{4.43}$$

represents the error in the desired frequency response $B(\omega)$. The error spectrum is bounded by

$$|E(\omega)| = \left|\sum_{l=0}^{L-1} e(l)e^{-j\omega l}\right| \le \sum_{l=0}^{L-1} |e(l)| \left|e^{-j\omega l}\right| \le \sum_{l=0}^{L-1} |e(l)|. \tag{4.44}$$

As shown in Equation (3.82),

$$|e(l)| \le \frac{\Delta}{2} = 2^{-B}. \tag{4.45}$$

Thus, Equation (4.44) becomes

$$|E(\omega)| \le L \cdot 2^{-B}. \tag{4.46}$$

This bound is too conservative because it can only be reached if all errors, $e(l)$, are of the same sign and have the maximum value in the range. A more realistic bound can be derived assuming $e(l)$ is statistically independent random variable.

Example 4.15: We first use a least-square method to design the FIR filter coefficients. To convert it to the fixed-point FIR filter, we use the filter construction function `dfilt` and change the arithmetic

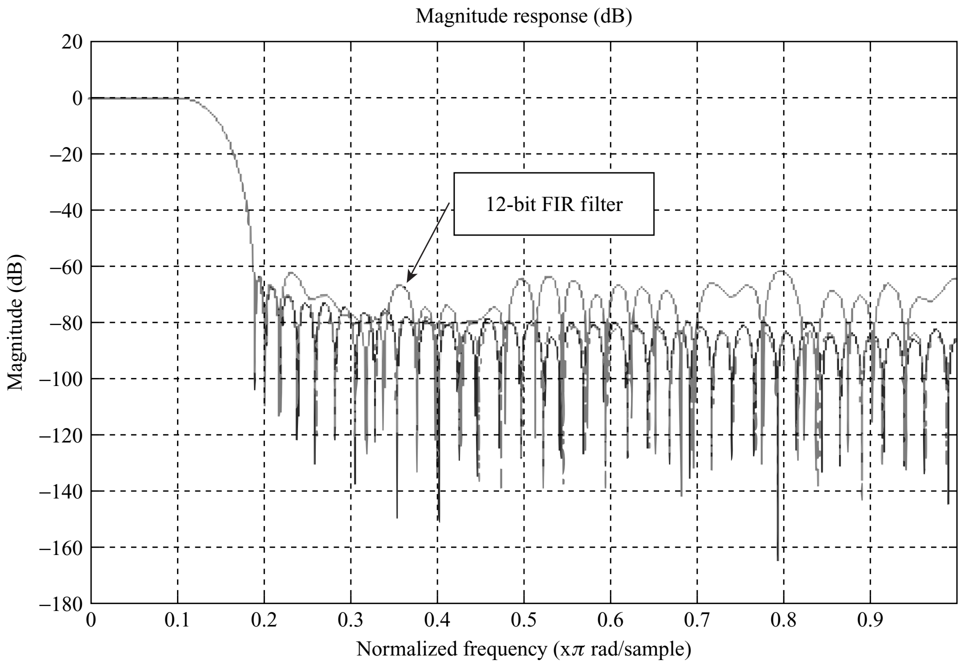


Figure 4.27 Magnitude responses of 12-bit and 16-bit FIR filters

setting for the filter to fixed-point arithmetic as follows:

```
hd = dfilt.dffir(b); % Create the direct-form FIR filter
set(hd,'Arithmetic','fixed');
```

The first function returns a digital filter object `hd` of type `dffir` (direct-form FIR filter). The second function `set(hd,'PropertyName',PropertyValue)` sets the value of the specified property for the graphics object with handle `hd`. We can use FVTool to plot the magnitude responses for both the quantized filter and the corresponding reference filter.

The fixed-point filter object `hd` uses 16 bits to represent the coefficients. We can make several copies of the filter for different wordlengths. For example, we can use

```
h1 = copy(hd);                    % Copy hd to h1
set(h1,'CoeffWordLength',12);     % Use 12 bits for coefficients
```

The MATLAB script is given in `example4_15.m`, and the magnitude responses of FIR filters with 16-bit and 12-bit coefficients are shown in Figure 4.27.

4.3.2 MATLAB Implementations

For simulation purposes, it is convenient to use a powerful software package such as MATLAB for software implementation of digital filter. MATLAB provides the function `filter` for FIR and IIR

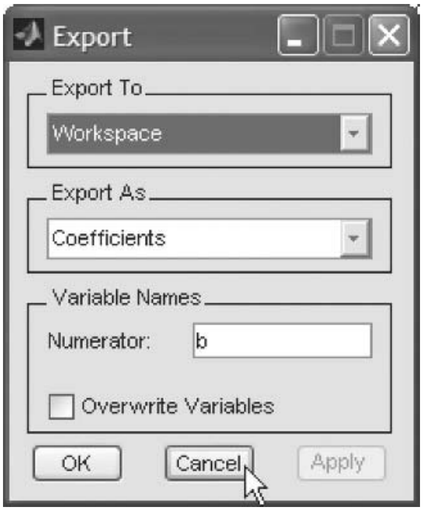


Figure 4.28 Export window from FDATool

filtering. The basic form of this function is

```
y = filter(b, a, x)
```

For FIR filtering, a $= 1$ and filter coefficients b_l are contained in the vector b. The input vector is x while the output vector generated by the filter is y.

Example 4.16: A 1.5 kHz sinewave with sampling rate 8 kHz is corrupted by white noise. This noisy signal can be generated, saved in file xn_int.dat, and plotted by MATLAB script example4_16.m. Note that we normalized the floating-point numbers and saved them in Q15 integer format using the following MATLAB commands:

```
xn_int = round(32767*in./max(abs(in)));% Normalize to 16-bit integer
fid = fopen('xn_int.dat','w');          % Save signal to xn_int.dat
fprintf(fid,'%4.0f\ n',xn_ int);        % Save in integer format
```

Using the bandpass filter designed in Example 4.13, we export FIR filter coefficients to current MATLAB workplace by selecting File→Export. From the pop-up dialog box **Export** shown in Figure 4.28, type b in the **Numerator** box, and click **OK**. This saves the filter coefficients in vector b, which is available for use in current MATLAB directory.

Now, we can perform FIR filtering using the MATLAB function filter by the command:

```
y = filter(b, 1, xn_int);
```

The filter output is saved in y vector of workspace, which can be plotted to compare with the input waveform.

Example 4.17: This example evaluates the accuracy of the fixed-point filter when compared to a double-precision floating-point version using random data as input signal. We create a quantizer to generate uniformly distributed white-noise data using 16-bit wordlength as

```
rand('state',0); % Initializing the random number generator
q = quantizer([16,15],'RoundMode','round');
xq = randquant(q,256,1);  % 256 samples
xin = fi(xq,true,16,15);
```

Now `xin` is an array of integers with 256 members, represented as a fixed-point object (a `fi` object). Now we perform the actual fixed-point filtering as follows:

```
y = filter(hd,xin);
```

The complete MATLAB program is given in `example4_17.m`.

4.3.3 Floating-Point C Implementations

The FIR filtering implementation usually begins with floating-point C, migrates to the fixed-point C, and then to assembly language programs.

Example 4.18: The input data is denoted as `x` and the filter output as `y`. The filter coefficients are stored in the coefficient array `h[ ]`. The filter delay line (signal vector) `w[ ]` keeps the past data samples. The sample-by-sample floating-point C program is listed as follows:

```
void floatPointFir(float *x, float *h, short order, float *y, float *w)
{
   short i;
   float sum;

   w[0] = *x++;                     // Get current data to delay line
   for (sum=0, i=0; i<order; i++) // FIR filter processing
   {
     sum += h[i] * w[i];
   }
   *y++ = sum;                      // Save filter output

   for (i=order-1; i>0; i--)        // Update signal buffer
   {
     w[i] = w[i-1];
   }
}
```

The signal buffer `w[ ]` is updated every sampling period as shown in Figure 4.9. For each update process, the oldest sample at the end of the signal buffer is discarded and the remaining samples are shifted one location down in the buffer. The most recent data sample $x(n)$ is inserted to the top location at `w[0]`.

It is more efficient to implement DSP algorithms using block-processing technique. For many practical applications such as wireless communications, speech processing, and video compression, the signal samples are usually grouped into packets or frames. An FIR filter that processes data by frames instead of sample by sample is called block-FIR filter. With the circular addressing mode available on most DSP processors, the shifting of data in the signal buffer can be replaced by circular buffer.

Example 4.19: The block-FIR filtering function processes one block of data samples for each function call. The input samples are stored in the array `x[ ]` and the filtered output samples are stored in the array `y[ ]`. In the following C program, the block size is denoted as `blkSize`:

```
void floatPointBlockFir(float *x, short blkSize, float *h, short order,
                        float *y, float *w, short *index)
{
   short i,j,k;
   float sum;
   float *c;

   k = *index;
   for (j=0; j<blkSize; j++)         // Block processing
   {
      w[k] = *x++;                   // Get current data to delay line
      c = h;
      for (sum=0, i=0; i<order; i++)// FIR filter processing
      {
         sum += *c++ * w[k++];
         k %= order                  // Simulate circular buffer
      }
      *y++ = sum;                    // Save filter output
      k = (order + k - 1)%order;     // Update index for next time
   }
   *index = k;                       // Update circular buffer index
}
```

4.3.4 Fixed-Point C Implementations

The fixed-point implementation using fractional representation is introduced in Chapter 3. The commonly used Q15 format is often used by fixed-point DSP processors.

Example 4.20: For fixed-point implementation, we use Q15 format to represent data samples in the range of -1 to $1 - 2^{-15}$. The ANSI C compiler requires the data type to be `long` to ensure that the product is saved as left aligned 32-bit data. When saving the filter output, the 32-bit temporary variable `sum` is shifted 15 bits to the right to compensate for the conversion from 32 bits to 16 bits after multiplication. The fixed-point C code is listed as follows:

```
void fixedPointBlockFir(short *x, short blkSize, short *h, short order,
                        short *y, short *w, short *index)
{
   short i,j,k;
   long sum;
   short *c;

   k = *index;
   for (j=0; j<blkSize; j++)      // Block processing
   {
      w[k] = *x++;                // Get current data to delay line
      c = h;
      for (sum=0, i=0; i<order; i++) // FIR filter processing
```

```
            {
               sum += *c++ * (long)w[k++];
               if (k == order)         // Simulate circular buffer
                   k = 0;
            }
            *y++ = (short)(sum>>15);   // Save filter output
            if (k-- <=0)               // Update index for next time
                k = order-1;
        }
        *index = k;                    // Update circular buffer index
    }
```

4.4 Applications: Interpolation and Decimation Filters

In many applications such as interconnecting DSP systems operating at different sampling rates, sampling frequency changes are necessary. The process of converting a digital signal to a different sampling rate is called sampling-rate conversion. The key processing for sampling-rate conversion is lowpass FIR filtering.

Sampling rate increased by an integer factor U is called interpolation, while decreased by an integer factor D is called decimation. Combination of interpolation and decimation allows the digital system to change the sampling rate with any ratio. One of the main applications of decimation is to eliminate the need for high-quality analog antialiasing filters. In an audio system that uses oversampling and decimation, the analog input is first filtered by a simple analog antialiasing filter and then sampled at a higher rate. The decimation filter then reduces the bandwidth of the sampled digital signal. The digital decimation filter provides high-quality lowpass filtering and reduces the cost of using expensive analog filters.

4.4.1 Interpolation

Interpolation is the process of inserting additional samples between successive samples of the original low-rate signal, and filtering the interpolated samples with an interpolation filter. For an interpolator of 1:U, the process inserts $(U-1)$ zeros in between the successive samples of the original signal $x(n)$ of sampling rate f_s, thus the sampling rate is increased to Uf_s, or the sampling period is reduced to T/U. This intermediate signal, $x(n')$, is then filtered by a lowpass filter to produce the final interpolated signal $y(n')$.

The simplest lowpass filter is a linear-phase FIR filter. The FDA Tool introduced in Section 4.2.5 can be used for designing this interpolation filter. The interpolating filter $B(z)$ operates at the high rate of $f_s' = Uf_s$ with the frequency response

$$B(\omega) = \begin{cases} U, & 0 \le \omega \le \omega_c \\ 0, & \omega_c < \omega \le \pi \end{cases}, \tag{4.47}$$

where the cutoff frequency is determined as

$$\omega_c = \frac{\pi}{U} \text{ or } f_c = f_s'/2U = f_s/2. \tag{4.48}$$

Since the insertion of $(U-1)$ zeros spreads the energy of each signal sample over U output samples, the gain U compensates for the energy loss of the up-sampling process. The interpolation increases the sampling rate while the bandwidth $(f_s/2)$ of the interpolated signal is still the same as the original signal.

Because the interpolation introduces $(U - 1)$ zeros between successive samples of the input signal, only one out of every U input samples sent to the interpolation filter is nonzero. To efficiently implement this filter, the required filtering operations may be rearranged to operate only on the nonzero samples. Suppose at time n, these nonzero samples are multiplied by the corresponding FIR filter coefficients b_0, $b_U, b_{2U}, \ldots, b_{L-U}$. At the following time $n+1$, the nonzero samples are multiplied by the coinciding filter coefficients $b_1, b_{U+1}, b_{2U+1}, \ldots, b_{L-U+1}$. This can be accomplished by replacing the high-rate FIR filter of length L with U, shorter polyphase filters $B_m(z)$, $m = 0, 1, \ldots, U-1$ of length $I = L/U$ at the low-rate f_s. The computational efficiency of the polyphase filter structure comes from dividing the single L-point FIR filter into a set of smaller filters of length L/U, each of which operates at the lower sampling rate f_s. Furthermore, these U polyphase filters share a single signal buffer of size L/U.

Example 4.21: Given the signal file `wn20db.dat`, which is sampled at 8 kHz. We can use the MATLAB script (`example4_21.m`) to interpolate it to 48 kHz. Figure 4.29 shows the spectra of the original signal, interpolated by 6 before and after lowpass filtering. This example shows that the lowpass filtering defined by Equation (4.47) removes all folded image spectra. Some useful MATLAB functions used in this example are presented in Section 4.4.4.

4.4.2 Decimation

Decimation of a high-rate signal with sampling rate f_s' by a factor D results in the lower rate $f_s'' = f_s'/D$. The down sample process by a factor of D may be simply done by discarding the $(D-1)$ samples that are between the low-rate ones. However, decreasing the sampling rate by a factor D reduces the bandwidth by the same factor D. Thus, if the original high-rate signal has frequency components outside the new bandwidth, aliasing would occur. Lowpass filtering the original signal $x(n')$ prior to the decimation process can solve the aliasing problem. The cutoff frequency of the lowpass filter is given as

$$f_c = f_s'/2D = f_s''/2. \tag{4.49}$$

This lowpass filter is called the decimation filter. The high-rate filter output $y(n')$ is down sampled to obtain the desired low-rate decimated signal $y(n'')$ by discarding $(D-1)$ samples for every D sample of the filtered signal $y(n')$.

The decimation filter operates at the high-rate f_s'. Because only every Dth output of the filter is needed, it is unnecessary to compute output samples that will be discarded. Therefore, the overall computation is reduced by a factor of D.

Example 4.22: Given the signal file `wn20dba.dat`, which is sampled at 48 kHz. We can use the MATLAB script (`example4_22.m`) to decimate it to 8 kHz. Figure 4.30 shows the spectra of the original signal, decimated by 6 with and without lowpass filtering. This example shows the lowpass filtering before decimation reduces the aliasing.

The spectrum in Figure 4.30(c) basically is part of the spectrum from 0 to 4000 Hz of Figure 4.30(a). The spectrum in Figure 4.30(b) is distorted especially in the low-magnitude segments.

4.4.3 Sampling-Rate Conversion

The sampling-rate conversion by a rational factor U/D can be done entirely in the digital domain with proper interpolation and decimation factors. We can achieve this digital sampling-rate conversion by first performing interpolation of a factor U, and then decimating the signal by a factor D. For example, we can convert digital audio signals for broadcasting (32 kHz) to professional audio (48 kHz) using a factor

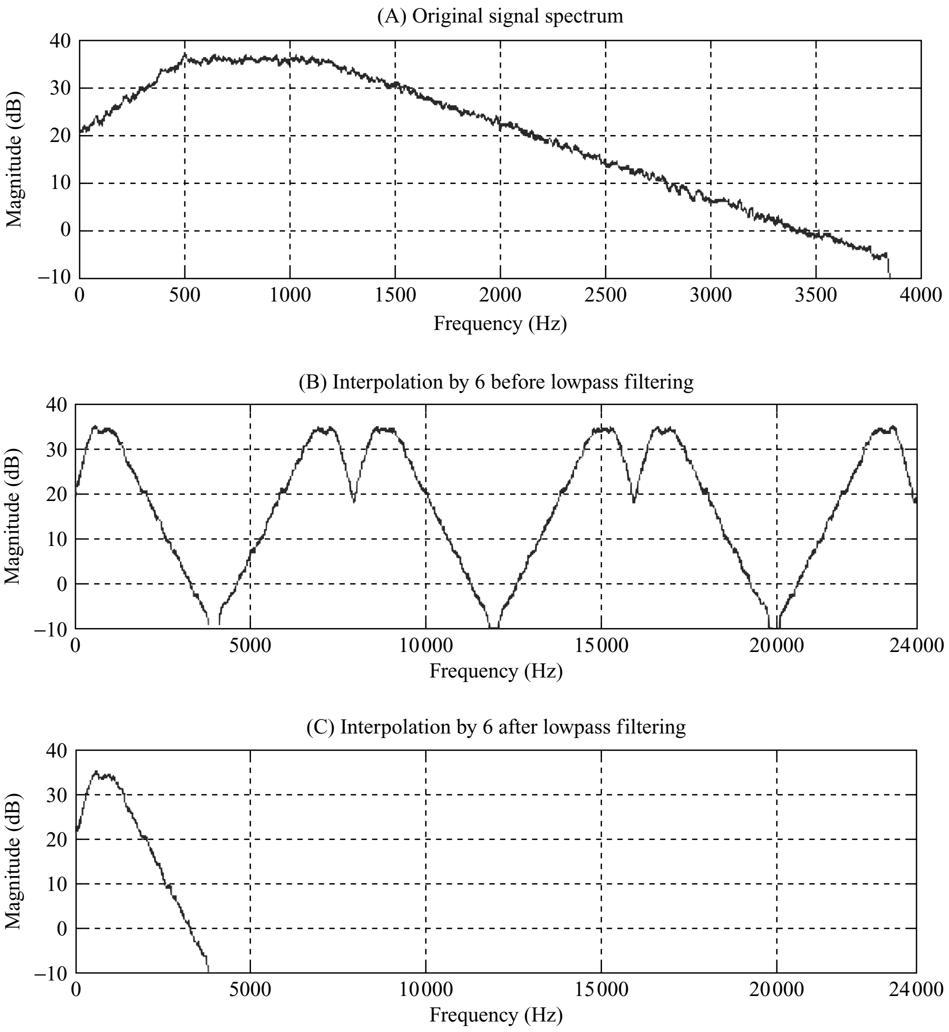


Figure 4.29 Interpolation by an integer operation: (a) original signal spectrum; (b) interpolation by 6 before lowpass filtering; and (c) interpolation after lowpass filtering

of $U/D = 3/2$. That is, we interpolate the 32 kHz signal with $U = 3$, then decimate the resulting 96 kHz signal with $D = 2$ to obtain the desired 48 kHz. It is very important to note that we have to perform interpolation before the decimation in order to preserve the desired spectral characteristics. Otherwise, the decimation may remove some of the desired frequency components that cannot be recovered by interpolation.

The interpolation filter must have the cutoff frequency given in Equation (4.48), and the cutoff frequency of the decimation filter is given in Equation (4.49). The frequency response of the combined filter must incorporate the filtering operations for both interpolation and decimation, and hence it should ideally have the cutoff frequency

$$f_c = \frac{1}{2} \min\left(f_s, f_s''\right). \tag{4.50}$$

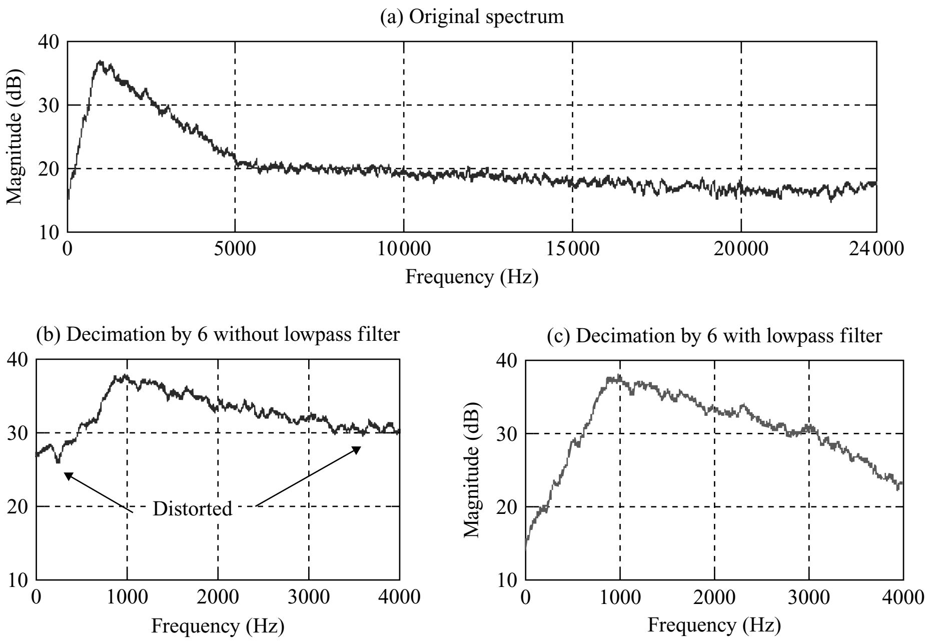


Figure 4.30 Decimation operation: (a) original signal spectrum; (b) decimation by 6 without lowpass filter; and (c) decimation by 6 with lowpass filter

Example 4.23: Convert a sinewave from 48 to 44.1 kHz using the following MATLAB script `example4_23.m` (adapted from the MATLAB **Help** menu for `upfirdn`). Some useful MATLAB functions used in this example are presented in Section 4.4.4.

```
g = gcd(48000, 44100);          % Greatest common divisor, g = 300
U = 44100/g;                    % Up sample factor, U=147
D = 48000/g;                    % Down sample factor, D = 160
N = 24*D;
b = fir1(N,1/D,kaiser(N+1,7.8562)); % Design FIR filter in b
b = U*b;                        % Passband gain = U
Fs = 48000;                     % Original sampling frequency: 48 kHz
n = 0:10239;                    % 10240 samples, 0.213 seconds long
x = sin(2*pi*1000/Fs*n);        % Original signal, sinusoid at 1 kHz
y = upfirdn(x,b,U,D);           % 9408 samples, still 0.213 seconds

% Overlay original (48 kHz) with resampled signal (44.1 kHz) in red
stem(n(1:49)/Fs,x(1:49));
hold on
stem(n(1:45)/(Fs*U/D),y(13:57),'r','filled');
xlabel('Time (seconds)');
ylabel('Signal value');
```

The original 48 kHz sinewave and the converted 44.1 kHz signal are shown in Figure 4.31.

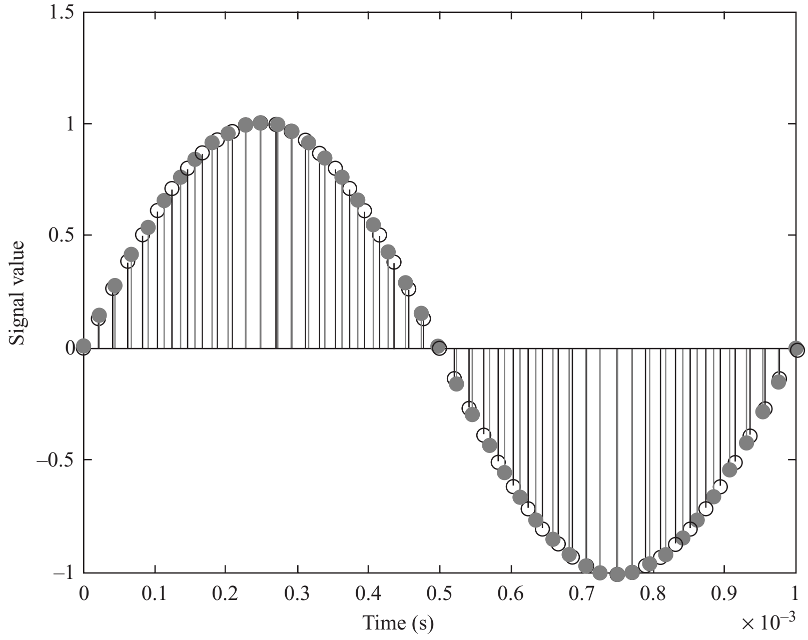


Figure 4.31 Sampling-rate conversion from 48 to 44.1 kHz

4.4.4 MATLAB Implementations

The interpolation introduced in Section 4.4.1 can be implemented by the MATLAB function `interp` with the following syntax:

```
y = interp(x, U);
```

The interpolated vector `y` is `U` times longer than the original input vector `x`.

The decimation for decreasing the sampling rate of a given sequence can be implemented by the MATLAB function `decimate` with the following syntax:

```
y = decimate(x, D);
```

This function uses an eighth-order lowpass Chebyshev type-I filter by default. We can employ FIR filter by using the following syntax:

```
y = decimate(x, D, 'fir');
```

This command uses a 30-order FIR filter generated by `fir1(30, 1/D)` to filter the data. We can also specify the FIR filter order L by using `y = decimate(x, D, L, 'fir')`.

Example 4.24: Given the speech file `timit_4.asc`, which is sampled by a 16-bit ADC with sampling rate 16 kHz. We can use the following MATLAB script (`example4_24.m`) to decimate it to 4 kHz:

```
load timit_4.asc -ascii;                % Load speech file
soundsc(timit_4, 16000)                 % Play at 16 kHz

timit4 = decimate(timit_4,4,60,'fir'); % Decimation by 4
soundsc(timit4, 4000)                   % Play the decimated speech
```

We can tell the sound quality (bandwidth) difference by listening to `timit_4` with 16 kHz bandwidth and `timit4` with 2 kHz bandwidth.

For sampling-rate conversion, we can use the MATLAB function `gcd` to find the conversion factor U/D. For example, to convert an audio signal from CD (44.1 kHz) for transmission using telecommunication channels (8 kHz), we can use the following commands:

```
g = gcd(8000, 44100);     % Find the greatest common divisor
U = 8000/g;               % Up sample factor
D = 44100/g;              % Down sample factor
```

In this example, we obtain $U = 80$ and $D = 441$ since `g` $= 100$.

The sampling-rate conversion algorithm is supported by the function `upfirdn` in the *Signal Processing Toolbox*. This function implements the efficient polyphase filtering technique. For example, we can use the following command for sampling-rate conversion:

```
y = upfindn(x, b, U, D);
```

This function first interpolates the signal in vector `x` with factor `U`, filters the intermediate resulting signal by the FIR filter given in coefficient vector `b`, and finally decimates the intermediate result using the factor `D` to obtain the final output vector `y`. The quality of the sampling-rate conversion result depends on the quality of the FIR filter.

Another function that performs sampling-rate conversion is `resample`. For example,

```
y = resample(x, U, D);
```

This function converts the sequence in vector `x` to the sequence in vector `y` with the sampling ratio U/D. It designs the FIR lowpass filter using `firls` with a Kaiser window.

MATLAB provides the function `intfilt` for designing interpolation (and decimation) FIR filters. For example,

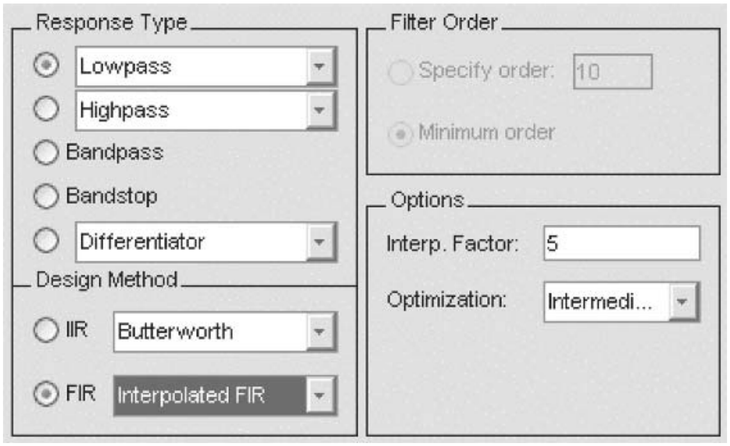
```
b = intfilt(U, L, alpha);
```

designs a linear-phase FIR filter with the interpolation ratio 1:U and saves the coefficients in vector `b`. The bandwidth of filter is `alpha` times the Nyquist frequency.

Finally, we can use FDATool designing an interpolation filter by selecting **Lowpass** and **Interpolated FIR** as shown in Figure 4.32. For the **Options**, we can enter U in **Interp. Factor** box. We can also specify other parameters as introduced in Section 4.2.5.

4.5 Experiments and Program Examples

In this section, we will present FIR filter implementation using fixed-point C, assembly programming, and use the C55x DSK for real-time application.


Response Type: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator

Design Method: IIR Butterworth [✓ FIR Interpolated FIR]

Filter Order: Specify order: 10 [✓ Minimum order]

Options:
Interp. Factor: 5
Optimization: Intermedi...

Figure 4.32 Design an interpolation filter using FDATool

4.5.1 Implementation of FIR Filters Using Fixed-Point C

This experiment uses the block-FIR filtering example presented in Section 4.3.4. The 16-bit test data is sampled at 8000 Hz and has three sinusoidal components at frequencies 800, 1800, and 3300 Hz. The 48-tap bandpass filter is designed using the following MATLAB script:

```
f = [0 0.3 0.4 0.5 0.6 1];
m = [0 0 1 1 0 0 ];
b = remez(47, f, m);
```

This bandpass filter will attenuate the input sinusoids of frequencies 800 and 3300 Hz. Figure 4.33 shows the CCS plots of the input and output waveforms along with their spectra. We use the file I/O method (introduced in Section 1.6.4) for reading and storing data files. The files used for this experiment are listed in Table 4.2.

Procedures of the experiment are listed as follows:

1. Open the project `fixedPoint_BlockFIR.pjt` and rebuild the project.
2. Load and run the program to filter the input data file `input.pcm`.
3. Validate the output result using CCS plots to show that 800 and 3300 Hz components are removed.
4. Profile the FIR filter performance.

4.5.2 Implementation of FIR Filter Using C55x Assembly Language

The TMS320C55x has MAC instructions, circular addressing modes, and zero-overhead nested loops to efficiently support FIR filtering. In this experiment, we use the same filter and input data as the previous

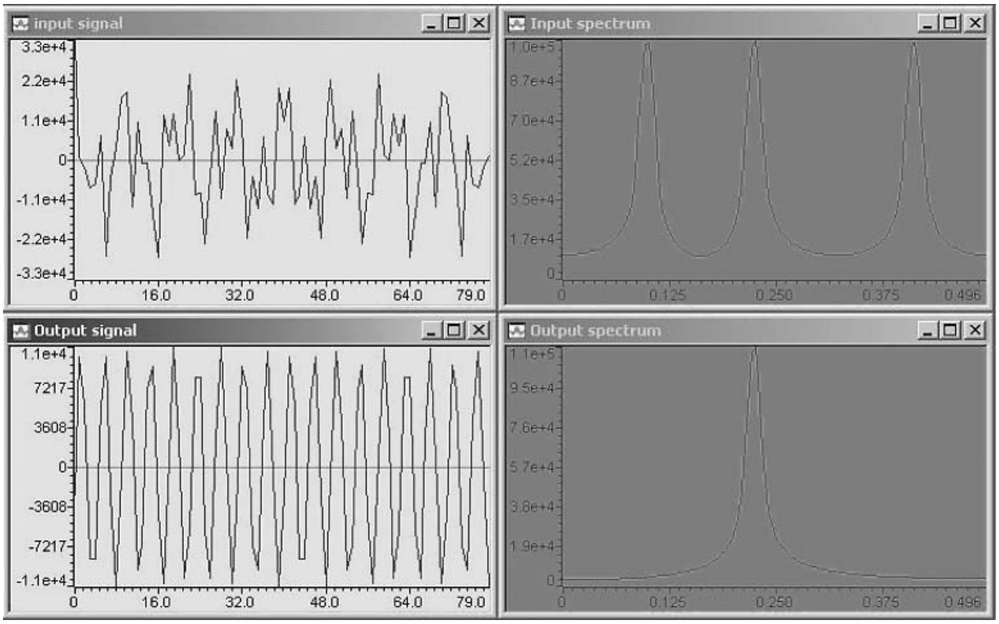


Figure 4.33 Input and output of the FIR filter. Input waveform (top left) and its spectrum (top right), and output waveform (bottom left) and its spectrum (bottom right)

experiment to realize the FIR filter using the following C55x assembly language:

```
      rptblocal sample_loop-1   ; Start the outer loop
      mov *AR0+,*AR3            ; Put the new sample to signal buffer
      mpym *AR3+,*AR1+,AC0      ; Do the 1st operation
   || rpt CSR                   ; Start the inner loop
      macm *AR3+,*AR1+,AC0
      macmr *AR3,*AR1+,AC0      ; Do the last operation with rounding
      mov hi(AC0),*AR2+         ; Save Q15 filtered value
   sample_loop
```

The filtering loop counter is CSR and the block FIR loop counter is BRC0. `AR0` points to the input buffer `x[ ]`. The signal buffer `w[ ]` is pointed by `AR3`. The coefficient array `h[ ]` is pointed by `AR1`. A new

Table 4.2 File listing for experiment `exp4.5.1_fixedPoint_BlockFIR`

Files	Description
`fixedPointBlockFirTest.c`	C function for testing block FIR filter
`fixedPointBlockFir.c`	C function for fixed-point block FIR filter
`fixedPointFir.h`	C header file for block FIR experiment
`firCoef.h`	FIR filter coefficients file
`fixedPoint_BlockFIR.pjt`	DSP project file
`fixedPoint_BlockFIR.cmd`	DSP linker command file
`input.pcm`	Data file

Table 4.3 File listing for experiment `exp4.5.2_asm_BlockFIR`

Files	Description
`blockFirTest.c`	C function for testing block FIR filter
`blockFir.asm`	Assembly implementation of block FIR filter
`blockFir.h`	C header file for block FIR experiment
`blockFirCoef.h`	FIR filter coefficients file
`asm_BlockFIR.pjt`	DSP project file
`asm_BlockFIR.cmd`	DSP linker command file
`input.pcm`	Data file

sample is placed in the signal buffer, and the inner loop repeats the MAC instructions. The intermediate results are kept in `AC0`. When the filtering operation is completed, the output $y(n)$ is rounded in Q15 format and stored in the output buffer `out[ ]`, which is pointed at by `AR2`. Both `AR1` and `AR3` are configured as circular pointers. The circular addressing mode is set as follows:

```
    mov   mmap(AR1),BSA01      ; AR1=base address for coefficients
    mov   mmap(T1),BK03        ; Set coefficient array size (order)
    mov   mmap(AR3),BSA23      ; AR3=base address for signal buffer
    or    #0xA,mmap(ST2_55)    ; AR1 & AR3 as circular pointers
    mov   #0,AR1               ; Coefficient start from h[0]
    mov   *AR4,AR3             ; Signal buffer start from w[index]
```

The circular addressing mode for signal and coefficient buffers is configured by setting the base address register `BSA01` for `AR1` and `BSA23` for `AR3`. The length of the circular buffers is determined by `BK03`. The starting address of the circular buffer for the coefficient vector `h` is always `h[0]`. For the signal buffer, the circular buffer starting address depends upon the previous iteration, which is passed by AR4. At the end of computation, the signal buffer pointer `AR3` will point at the oldest sample, $w(n - L + 1)$. This offset is kept as shown in Figure 4.10.

In this experiment, we set C55x `FRCT` bit to automatically compensate for the Q15 multiplication. The `SMUL` and `SATD` bits are set to handle the saturation of the fractional integer operation. The `SXMD` bit sets the sign-extension mode. The C55x assembly language implementation of FIR filtering takes `order+3` clock cycles to process each input sample. Thus, the 48-tap FIR filter needs 51 cycles, excluding the overhead. Table 4.3 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Open the project `asm_BlockFIR.pjt` and rebuild it.

2. Load the FIR filter project and run the program to filter the input data.

3. Validate the output data to ensure that the 800 and 3300 Hz components are attenuated.

4. Profile the FIR filter performance and compare the result with the fixed-point C implementation.

4.5.3 Optimization for Symmetric FIR Filters

The TMS320C55x has two special instructions `firsadd` and `firssub` to implement the symmetric and antisymmetric FIR filters, respectively. The syntax of instructions is

```
firsadd Xmem,Ymem,Cmem,ACx,ACy
```

where `Xmem` and `Ymem` are the signal buffers of $\{x(n), x(n-1), \ldots x(n-L/2+1)\}$ and $\{x(n-L/2), \ldots x(n-L+1)\}$, respectively, and `Cmem` is the coefficient buffer.

The `firsadd` instruction is equivalent to the following parallel instructions:

```
     macm *CDP+,ACx,ACy    ; b_l[x(n − l)+x(n+l − L+1)]
||   add  *ARx+,*ARy+,ACx ; x(n − l+1)+x(n+l − L+2)
```

The `macm` instruction carries out the multiply–accumulate portion of the symmetric filter operation, and the `add` instruction adds a pair of samples for the next iteration. The implementation of symmetric FIR filter using the C55x assembly program is listed as follows:

```
     rptblocal sample_loop-1       ; To prevent overflow in addition,
     mov    #0,AC0                 ; input is scaled to Q14 format
||   mov    AC1<<#-1,*AR3          ; Put input to signal buffer in Q14
     add    *AR3+,*AR1-,AC1        ; AC1=[x(n)+x(n-L+1)]<<16
||   rpt    CSR                    ; Do order/2-2 iterations
     firsadd *AR3+,*AR1-,*CDP+,AC1,AC0
     firsadd *(AR3-T0),*(AR1+T1),*CDP+,AC1,AC0
     macm   *CDP+,AC1,AC0          ; Finish the last macm instruction
     mov   rnd(hi(AC0<<#1)),*AR2+  ; Store the rounded & scaled result
||   mov   *AR0+,AC1               ; Get next sample
sample_loop
```

We need to store only the first half of the symmetric FIR filter coefficients. The inner-repeat loop is set to $L/2 - 2$ since each multiply–accumulate operation accounts for a pair of samples. In order to use `firsadd` instruction inside a repeat loop, we add the first pair of filter samples using the dual memory add instruction

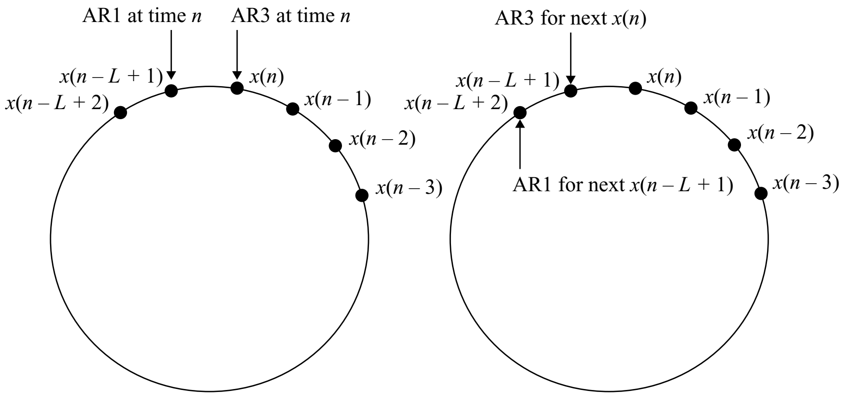
```
add *AR3+,*AR1-,AC1
```

We also place the following instructions outside the repeat loop for the final calculation:

```
firsadd *(AR3-T0),*(AR1+T1),*CDP+,AC1,AC0
macm    *CDP+,AC1,AC0
```

We use two data pointers AR1 and AR3 to address the signal buffer. AR3 points at the newest sample in the buffer, and AR1 points at the oldest sample in the buffer. Temporary registers, T1 and T0, are used as the offsets for updating circular buffer pointers. The offsets are initialized to $T0 = L/2$ and $T1 = L/2 - 2$. Figure 4.34 illustrates these two circular buffer pointers for a symmetric FIR filtering. The `firsadd` instruction accesses three data buses simultaneously.

Two implementation issues should be considered. First, the instruction `firsadd` adds two corresponding samples, which may cause an undesired overflow. Second, the `firsadd` instruction accesses three read operations in the same cycle, which may cause data bus contention. The first problem can be resolved by scaling the input to Q14 format, and scaling the filter output back to Q15. The second problem can be resolved by placing the coefficient buffer and the signal buffer in different memory blocks. The C55x assembly language implementation of symmetric FIR filter takes (`order`/2) + 4 clock cycles to process each input data. Thus, this 48-tap FIR filter needs 28 cycles for each sample, excluding the overhead. Table 4.4 lists the files used for this experiment.



(a) Circular buffer for a symmetric FIR filter at time n

(b) Circular buffer for a symmetric FIR filter at time $n+1$

Figure 4.34 Circular buffer for a symmetric FIR filtering. The pointers to $x(n)$ and $x(n-L+1)$ are updated at the counterclockwise direction: (a) circular buffer for a symmetric FIR filter at time n; (b) circular buffer for a symmetric FIR filter at time $n+1$

Procedures of the experiment are listed as follows:

1. Open the `symmetric_BlockFIR.pjt` and rebuild the project.
2. Load and run the program to filter the input data.
3. Validate the output data to ensure that the 800 and 3300 Hz components are removed.
4. Profile the FIR filter performance and compare the result with previous C55x assembly language implementation.

4.5.4 Optimization Using Dual MAC Architecture

Dual MAC improves the processing speed by generating two outputs, $y(n)$ and $y(n+1)$, in parallel. For example, the following parallel instructions use dual MAC architecture:

```
   rpt CSR
   mac *ARx+,*CDP+,ACx ; ACx=bl*x(n)
:: mac *ARy+,*CDP+,ACy ; ACy=bl*x(n+1)
```

Table 4.4 File listing for experiment `exp4.5.3_symmetric_BlockFIR`

Files	Description
`symFirTest.c`	C function for testing symmetric FIR filter
`symFir.asm`	Assembly routine of symmetric FIR filter
`symFir.h`	C header file for symmetric FIR experiment
`symFirCoef.h`	FIR filter coefficients file
`symmetric_BlockFIR.pjt`	DSP project file
`symmetric_BlockFIR.cmd`	DSP linker command file
`input.pcm`	Data file

In this example, ARx and ARy are data pointers to $x(n)$ and $x(n+1)$, and CDP is the coefficient pointer. The repeat loop produces two filter outputs $y(n)$ and $y(n+1)$. After execution, pointers CDP, ARx, and ARy are increased by 1. The following example shows the C55x assembly implementation using the dual MAC and circular buffer for a block-FIR filter:

```
   rptblocal sample_loop-1
   mov   *AR0+,*AR1       ; Put new sample to signal buffer x[n]
   mov   *AR0+,*AR3       ; Put next new sample to location x[n+1]
   mpy   *AR1+,*CDP+,AC0  ; The first operation
:: mpy   *AR3+,*CDP+,AC1
|| rpt   CSR
   mac   *AR1+,*CDP+,AC0  ; The rest MAC iterations
:: mac   *AR3+,*CDP+,AC1
   macr  *AR1,*CDP+,AC0
:: macr  *AR3,*CDP+,AC1   ; The last MAC operation
   mov    pair(hi(AC0)),dbl(*AR2+); Store two output data
sample_loop
```

There are three implementation issues to be considered when using the dual MAC architecture: (1) We must increase the length of the signal buffer by 1 to accommodate an extra memory location required for computing two signals in parallel. With an additional space in the buffer, we can form two sequences in the signal buffer, one pointed by AR1 and the other by AR3. (2) Dual MAC implementation of the FIR filtering needs three memory reads (two data samples and one filter coefficient) simultaneously. To avoid memory bus contention, we shall place the signal buffer and the coefficient buffer in different memory blocks. (3) The results are kept in two accumulators, thus requires two store instructions to save two output samples. It is more efficient to use the following dual-memory-store instruction

```
mov pair(hi(AC0)),dbl(*AR2+)
```

to save both outputs to the data memory in 1 cycle. However, the dual-memory-store instruction requires the data to be aligned on an even word (32-bit) boundary. This alignment can be set using the key word `align 4` in the linker command file as

```
output : {} > RAM0 align 4 /* word boundary alignment */
```

and using the `DATA_SECTION pragma` directive to tell the linker where to place the output sequence. Another method to set data alignment is to use `DATA_ALIGN pragma` directive as

```
#pragma DATA_ALIGN(y,2); /* Alignment for dual accumulator store */
```

The C55x implementation of FIR filter using dual MAC needs (`order`+3)/2 clock cycles to process each input data. Thus, it needs 26 cycles for each sample excluding the overhead. The files used for this experiment are listed in Table 4.5.

Table 4.5 File listing for experiment `exp4.5.4_dualMAC_BlockFIR`

Files	Description
`dualMacFirTest.c`	C function for testing dual MAC FIR filter
`dualMacFir.asm`	Assembly routine of dual MAC FIR filter
`dualMacFir.h`	C header file for dual MAC FIR experiment
`dualMacFirCoef.h`	FIR filter coefficients file
`dualMAC_BlockFIR.pjt`	DSP project file
`dualMAC_BlockFIR.cmd`	DSP linker command file
`input.pcm`	Data file

Procedures of the experiment are listed as follows:

1. Open the `dual_BlockFIR.pjt` and rebuild the project.

2. Load the FIR filter project and run the program to filter the input data.

3. Validate the output data to ensure that the 800 and 3300 Hz components are removed.

4. Profile the FIR filter performance and compare the result with previous C55x assembly language implementations.

4.5.5 Implementation of Decimation

The implementation of a decimator must consider multistage filter if the decimation factor can be formed by common multiply factors. In this experiment, we will implement the 6:1 decimator using two FIR filters of 2:1 and 3:1 decimation ratios.

The two-stage decimator uses the input, output, and temporary buffers. The input buffer size is equal to the frame size multiplied by the decimation factor. For example, when the frame size is chosen as 80, the 48 to 8 kHz decimation will require the input buffer size of 480 (80 * 6). The temporary buffer size (240) is determined as the input buffer size (480) divided by the first decimation factor 2.

The offset, $D - 1$, is preloaded to the temporary register T0. After reading two input data samples to the signal buffer, the address pointers AR1 and AR3 are incremented by $D - 1$. The decimation FIR filter uses the dual MAC instruction with loop unrolling. The last instruction

```
mov pair(hi(AC0)),dbl(*AR2+)
```

requires the output address pointer to be aligned with even-word boundary. Table 4.6 lists the files used for this experiment.

```
||    rptblocal sample_loop-1
      mov  *(AR0+T0),*AR3    ; Put new sample to signal buffer x[n]
      mov  *(AR0+T0),*AR1    ; Put next new sample to location x[n+1]
      mpy  *AR1+,*CDP+,AC0   ; The first operation
::    mpy  *AR3+,*CDP+,AC1
||    rpt  CSR
      mac  *AR1+,*CDP+,AC0   ; The rest MAC iterations
::    mac  *AR3+,*CDP+,AC1
      macr *AR1,*CDP+,AC0
::    macr *AR3,*CDP+,AC1    ; The last MAC operation
```

Table 4.6 File listing for experiment `exp4.5.5_decimation`

Files	Description
`decimationTest.c`	C function for testing decimation experiment
`decimate.asm`	Assembly routine of decimation filter
`decimation.h`	C header file for decimation experiment
`coef48to24.h`	FIR filter coefficients for 2:1 decimation
`coef24to8.h`	FIR filter coefficients for 3:1 decimation
`decimation.pjt`	DSP project file
`decimation.cmd`	DSP linker command file
`tone1k_48000.pcm`	Data file 1 kHz tone at 48 kHz sampling rate

```
        mov pair(hi(AC0)),dbl(*AR2+); Store two output data
    sample_loop
```

Procedures of the experiment are listed as follows:

1. Open the `decimation.pjt` and rebuild the project.

2. Load and run the program to obtain the output data using the input data given in the data folder.

3. The 1000 Hz sinewave at 48 kHz sampling rate will have 48 samples per cycle. Validate the output data of the 1000 Hz sinewave. At 8 kHz sampling rate, each cycle should have eight samples, see Figure 4.35.

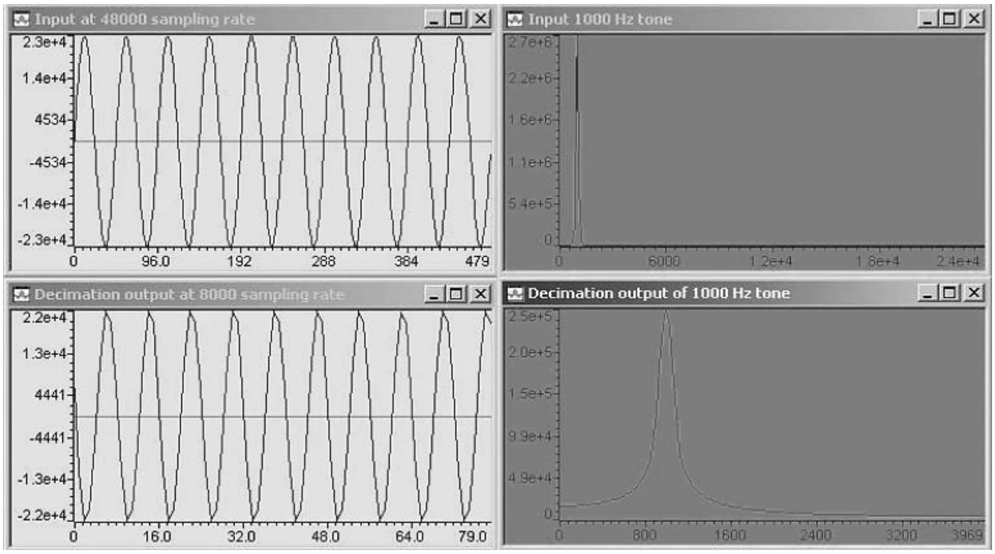


Figure 4.35 Decimation of the 48 kHz sampling-rate signal to 8 kHz. 1 kHz tone sampled at 48 kHz (top left) and its spectrum (top right). Decimation output of 8 kHz sampling rate (bottoom left) and its spectrum (bottom right)

4. Use MATLAB to plot the spectrum of decimation output to verify that it is 1000 Hz sinewave.

4.5.6 Implementation of Interpolation

In this experiment, we interpolate the 8 kHz sampling data to 48 kHz. We will use two interpolation filters with interpolation factors of 2 and 3. The interpolation filter is implemented using fixed-point C program that mimics circular addressing mode. The circular buffer index is kept by the variable `index`. The coefficient array is `h[ ]`, and the signal buffer is `w[ ]`. Since we do not have to filter the data samples with zero values, the coefficient array pointer is offset with interpolation factor. Table 4.7 lists the files used for this experiment. The C code is listed as follows:

```
    k = *index;
    for (j=0; j<blkSize; j++)      // Block processing
```

```
{
  c = h;
  w[k] = *x++;                // Get the current data to delay line
  m = k;
  for (n=0; n<intp; n++)
  {
     sum = 0;
     for (i=0; i<order; i++)  // FIR filtering
     {
        sum += c[i*intp] * (long)w[k++];
        if (k == order)       // Simulate circular buffer
           k = 0;
     }
     *y++ = (short)(sum>>14); // Save filter output
     c++;
     k = m;
  }
  k--;
  if (k<0)                    // Update index for next time
     k += order;
}
```

Table 4.7 File listing for experiment `exp4.5.6_interpolation`

Files	Description
`interpolateTest.c`	C function for testing interpolation experiment
`interpolate.c`	C function for interpolation filter
`interpolation.h`	C header file for interpolation experiment
`coef8to16.h`	FIR filter coefficients for 1:2 interpolation
`coef16to48.h`	FIR filter coefficients for 1:3 interpolation
`interpolation.pjt`	DSP project file
`interpolation.cmd`	DSP linker command file
`tone1k_8000.pcm`	Data file – 1 kHz tone at 8 kHz sampling rate

Procedures of the experiment are listed as follows:

1. Open the `interpolation.pjt` and rebuild the project.
2. Load and run the program to obtain the output data using the input data given in the data folder.
3. The 1000 Hz sinewave input data sampled at 8 kHz will have eight samples in each cycle. Validate the output 1000 Hz sinewave data at 48 kHz sampling rate that each cycle should have 48 samples.
4. Use MATLAB to plot the spectrum of interpolator output to verify that it is a 1000 Hz tone.

4.5.7 Sample Rate Conversion

In this experiment, we will convert the sampling rate from 48 to 32 kHz. We first interpolate the signal sampled at 48 kHz to 96 kHz, and then decimate it to 32 kHz. The files used for this experiment are listed in Table 4.8. Figure 4.36 illustrates the procedures of sampling-rate conversion from 48 to 32 kHz.

Table 4.8 File listing for experiment `exp4.5.7_SRC`

Files	Description
`srcTest.c`	C function for testing sample rate conversion
`interpolate.c`	C function for interpolation filter
`decimate.asm`	Assembly routine for decimation filter
`interpolation.h`	C header file for interpolation
`decimation.h`	C header file for decimation
`coef96to32.h`	FIR filter coefficients for 3:1 decimation
`SRC.pjt`	DSP project file
`SRC.cmd`	DSP linker command file
`tone1k_48000.pcm`	Data file – 1 kHz tone at 48 kHz sampling rate

The first lowpass filter with cutoff frequency 48 kHz ($\pi/2$) may not be necessary in this case since a decimation lowpass filter with narrower cutoff frequency is immediately followed.

Procedures of the experiment are listed as follows:

1. Open the `SRC.pjt` and rebuild the project.
2. Load and run the program to obtain the output data using the input data given in the folder.
3. The input signal sampled at 48 kHz will be converted to 32 kHz at the output. For each period, the output should have 32 samples.
4. Use MATLAB to plot the output spectrum to verify that it is 1000 Hz.

4.5.8 Real-Time Sample Rate Conversion Using DSP/BIOS and DSK

In this experiment, we create a DSP/BIOS application that uses C5510 DSK to capture and play back audio samples for real-time sample rate conversion using C5510 DSK. The DSP/BIOS is a small kernel included in the CCS for real-time synchronization, host-target communication, and scheduling. It provides multithreading, real-time analysis, and configuration capabilities to greatly reduce the development effort when hardware and other processor resources are involved.

Step 1: *Create a DSP/BIOS configuration file*

A DSP/BIOS program needs a configuration file, which is a window interface that determines application parameters and sets up modules including interrupts and I/Os. To create configuration file for the C55x DSK, we start from CCS menu File→New→DSP Configuration to select `dsk5510.cdb` and click **OK**. When the new configuration file is opened, save it as `dspbios.cdb`. Similar to the previous experiments, create and save the DSP/BIOS project, `DSPBIOS.pjt`, and add the configuration file to the project.

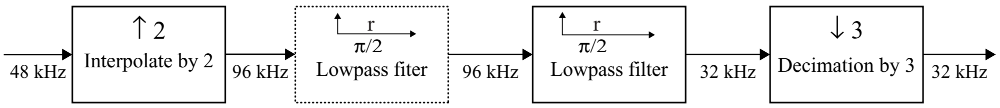
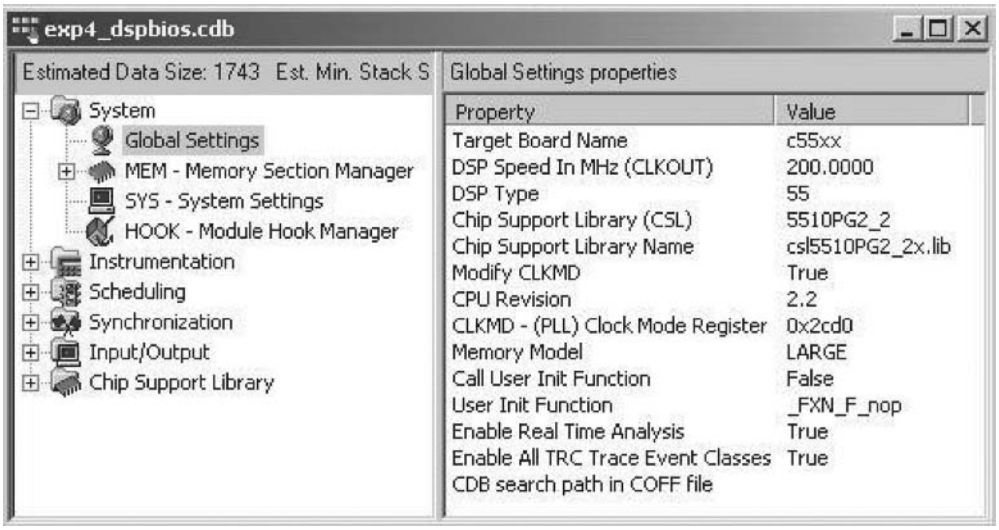


Figure 4.36 Sampling-rate conversion



Property	Value
Target Board Name	c55xx
DSP Speed In MHz (CLKOUT)	200.0000
DSP Type	55
Chip Support Library (CSL)	5510PG2_2
Chip Support Library Name	csl5510PG2_2x.lib
Modify CLKMD	True
CPU Revision	2.2
CLKMD - (PLL) Clock Mode Register	0x2cd0
Memory Model	LARGE
Call User Init Function	False
User Init Function	_FXN_F_nop
Enable Real Time Analysis	True
Enable All TRC Trace Event Classes	True
CDB search path in COFF file	

Figure 4.37 The DSP/BIOS configuration file

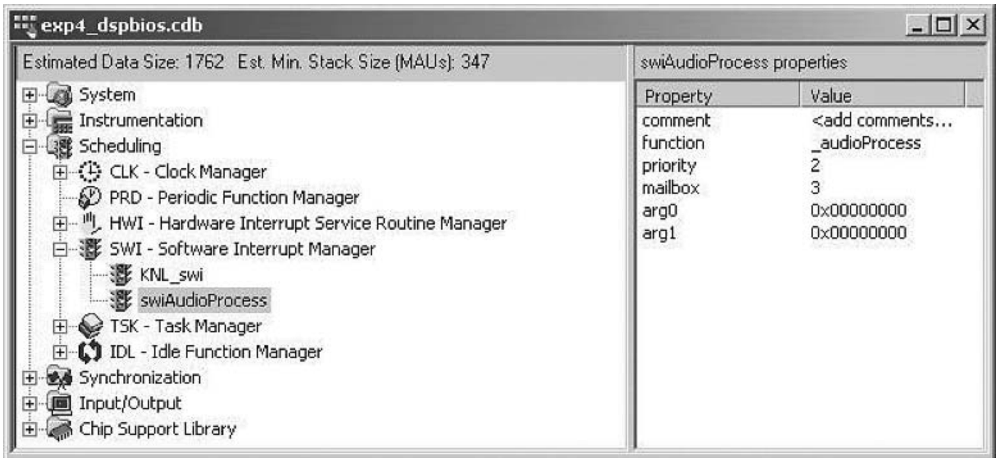
Double click the configuration file to open it as shown in Figure 4.37. Left click the + sign in front of an item on the left window will open the property of that item on the right window. To change the parameters listed by the configuration file, right click the item and select **Properties**. The configuration has six items: **System**, **Instrumentation**, **Scheduling**, **Synchronization**, **Input/Output**, and **Chip Support Library**. Under the **System** item, users can change and modify the processor global settings and adjust memory blocks size and allocation. To make changes, select the item and right click to bring up the **Properties** of that item. In this experiment, we will use the default global settings.

Step 2: *Create a software interrupt object*

Open the **Scheduling** and click the + sign in front of **SWI** to open the submenu, right click **SWI** and select **Insert SWI** to insert a new software interrupt object. Rename the newly inserted `SWI0` to `swiAudioProcess`. Right click `swiAudioProcess` and select **Properties** again to open the dialog box, enter new function name `_audioProcess` to the **Function** box, and set the **Priority** to 2 and **Mailbox** to 3 as shown in Figure 4.38.

Step 3: *Set up pipe input and output*

We now connect the input and output of the DSK with DSP/BIOS through the configuration file. Click the + sign in front of the **Input/Output** to open the submenu, right click the **PIP Buffer Pipe Manager** and select **Insert PIP** to insert two new PIPs. Rename one to `pipRx` and the other to `pipTx`. This adds two ping-pong data buffers through the DMA for connecting input and output. Right click `pipRx` and select **Properties** to open the dialog box. For the experiment, we configure `pipRx` to receive audio samples and `pipTx` to transmit audio samples. First, we align the buffer in even-word boundary by setting the **bufalign** to 2, and we change the buffer size to 480 by modifying the **frame-size**. We then move to **Notify Functions** window and change the **notifyWriter** from `_FXN_F_nop` to `_PLIO_rxPrime`, and change the **notifyReader** from `_FXN_F_nop` to `_SWI_andnHook`. In the function `_PLIO_rxPrime`, we enter `_plioRx` to the **nwarg0** field, and 0 to the **nwarg1** field. In the function `_SWI_andnHook`, we enter `_swiAudioProcess` to the **nrarg0** field, and 1 to the **nrarg1** field. We configure `pipTx` to work with `pipRx` in a similar way. Note that the notification monitor for


exp4_dspbios.cdb
Estimated Data Size: 1762 Est. Min. Stack Size (MAUs): 347
System
Instrumentation
Scheduling
CLK - Clock Manager
PRD - Periodic Function Manager
HWI - Hardware Interrupt Service Routine Manager
SWI - Software Interrupt Manager
KNL_swi
swiAudioProcess
TSK - Task Manager
IDL - Idle Function Manager
Synchronization
Input/Output
Chip Support Library
swiAudioProcess properties

Property	Value
comment	<add comments...
function	_audioProcess
priority	2
mailbox	3
arg0	0x00000000
arg1	0x00000000

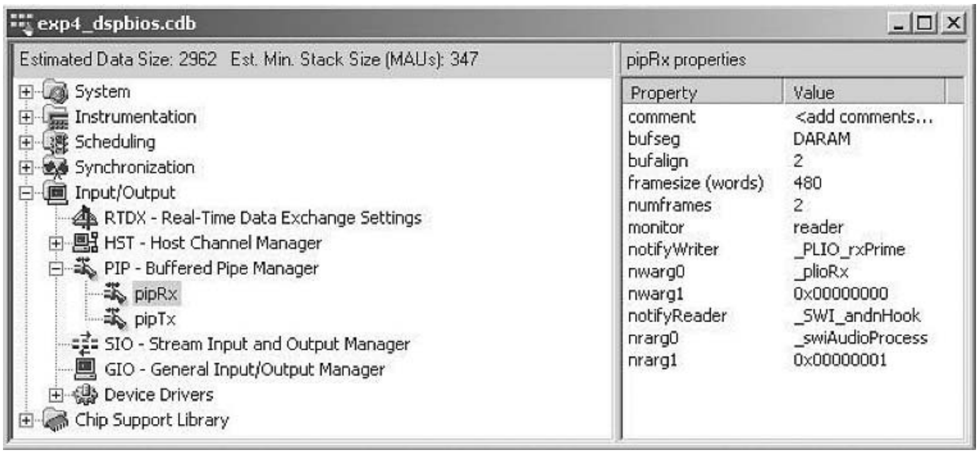
Figure 4.38 Setting up SWI

`pipRx` is `reader`, while `pipTx` uses `writer`. The settings of `pipRx` and `pipTx` are shown in Figures 4.39 and 4.40, respectively.

Step 4: *Configure the DMA*

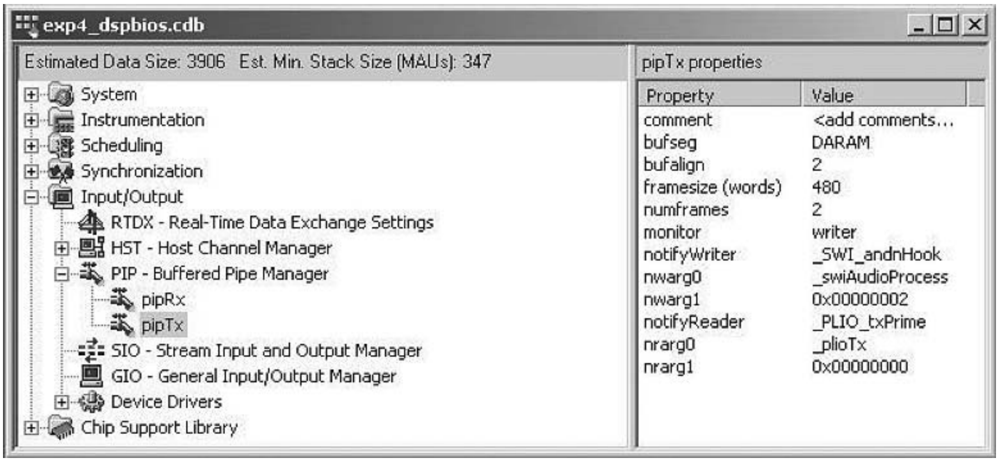
This step connects the input/output of the DSK using the DMA controller. From the DSP/BIOS configuration file dialog window, select **Chip Support Library** and open DMA→Direct Memory Access Controller. From the DMA **Configuration Manager**, insert two new **dmaCfg** objects and rename them as `dmaCfgReceive` and `dmaCfgTransmit`. Open `dmaCfgReceive` and from the **Frame** tab configure the frame as follows:

Data Type = `16-bit`.
Number of Element (CEN) = `256`.


exp4_dspbios.cdb
Estimated Data Size: 2962 Est. Min. Stack Size (MAUs): 347
System
Instrumentation
Scheduling
Synchronization
Input/Output
RTDX - Real-Time Data Exchange Settings
HST - Host Channel Manager
PIP - Buffered Pipe Manager
pipRx
pipTx
SIO - Stream Input and Output Manager
GIO - General Input/Output Manager
Device Drivers
Chip Support Library
pipRx properties

Property	Value
comment	<add comments...
bufseg	DARAM
bufalign	2
framesize (words)	480
numframes	2
monitor	reader
notifyWriter	_PLIO_rxPrime
nwarg0	_plioRx
nwarg1	0x00000000
notifyReader	_SWI_andnHook
nrarg0	_swiAudioProcess
nrarg1	0x00000001

Figure 4.39 The settings of `pipRx` in DSP/BIOS buffered pipe manager


exp4_dspbios.cdb
Estimated Data Size: 3906 Est. Min. Stack Size (MAUs): 347
System
Instrumentation
Scheduling
Synchronization
Input/Output
RTDX - Real-Time Data Exchange Settings
HST - Host Channel Manager
PIP - Buffered Pipe Manager
pipRx
pipTx
SIO - Stream Input and Output Manager
GIO - General Input/Output Manager
Device Drivers
Chip Support Library
pipTx properties

Property	Value
comment	<add comments...
bufseg	DARAM
bufalign	2
framesize (words)	480
numframes	2
monitor	writer
notifyWriter	_SWI_andnHook
nwarg0	_swiAudioProcess
nwarg1	0x00000002
notifyReader	_PLIO_txPrime
nrarg0	_plioTx
nrarg1	0x00000000

Figure 4.40 The settings of `pipTx` in DSP/BIOS buffered pipe manager

Number of Frames (CFN) = `1`.
Frame Index (CFI) = `0`.
Element Index (CEI) = `0`.

In the **Source** tab of the `dmaCfgReceive`, set the source configuration as follows:
Burst Enable (SRC BEN) = `Single Access (No Burst)`.
Packing (SRC PACK) = `No Packing Access`.
Source Space = `Data Space`.
Source Address Format = `Numeric`.
Start Address (CSSA) = `0x006002`.
Address Mode (SRC AMODE) = `Constant`.
Transfer Source (SRC) = `Peripheral Bus`.

In the **Destination** tab of the `dmaCfgReceive`, set the destination configuration as follows:
Burst Enable (DST BEN) = `Single Access (No Burst)`.
Packing (DST PACK) = `No Packing Access`.
Destination Space = `Data Space`.
Destination Address Format = `Numeric`.
Start Address (CDSA) = `0x000000`.
Address Mode (DST AMODE) = `Post-incremented`.
Transfer Destination (DST) = `DARAM`.

In the **Control** tab of the `dmaCfgReceive`, set the control configuration as follows:
Sync Event (SYNC) = `McBSP 2 Receive Event (REVT2)`.
Repetitive Operations (REPEAT) = `Only if END PROG = 1`.
End of Programmation (END PROG) = `Delay re-initialization`.
Frame Synchronization (FS) = `Disabled`.
Channel Priority (PRIO) = `High`.
Channel Enable (EN) = `Disabled`.
Auto-initialization (Auto INIT) = `Disabled`.

In the **Interrupts** tab of the `dmaCfgReceive`, set the interrupt configuration as follows:
Timeout (TIMEOUT IE) = `Disabled`.
Synchronization Event drop (DROP IE) = `Disabled`.

Half Frame (FALF IE) = `Disabled`.
Frame Complete (FRAME IE) = `Enabled`.
Last Frame (LAST IE) = `Disabled`.
End Block (BLOCK IE) = `Disabled`.

The DMA configuration management also needs to be configured for transmit. Open `dmaCfgTransmit` and from the **Frame** tab configure the frame to:

Data Type = `16-bit`.
Number of Element (CEN) = `256`.
Number of Frames (CFN) = `1`.
Frame Index (CFI) = `0`.
Element Index (CEI) = `0`.

In the **Source** tab of the `dmaCfgTransmit`, set the source configuration as follows:

Burst Enable (SRC BEN) = `Single Access (No Burst)`.
Packing (SRC PACK) = `No Packing Access`.
Source Space = `Data Space`.
Source Address Format = `Numeric`.
Start Address (CSSA) = `0x000000`.
Address Mode (SRC AMODE) = `Post-increment`.
Transfer Source (SRC) = `DARAM`.

In the **Destination** tab of the `dmaCfgTransmit`, set the destination configuration as follows:

Burst Enable (DST BEN) = `Single Access (No Burst)`.
Packing (DST PACK) = `No Packing Access`.
Destination Space = `Data Space`.
Destination Address Format = `Numeric`.
Start Address (CDSA) = `0x006006`.
Address Mode (DST AMODE) = `Constant`.
Transfer Destination (DST) = `Peripheral Bus`.

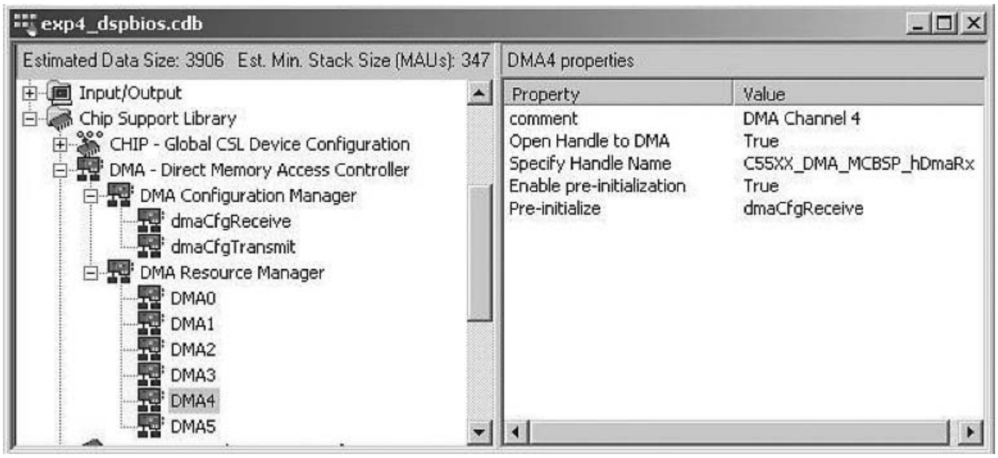
In the **Control** tab of the `dmaCfgTransmit`, set the control configurations as follows:

Sync Event (SYNC) = `McBSP 2 Transmit Event (XEVT2)`.
Repetitive Operations (REPEAT) = `Only if END PROG = 1`.
End of Programmation (END PROG) = `Delay re-initialization`.
Frame Synchronization (FS) = `Disabled`.
Channel Priority (PRIO) = `High`.
Channel Enable (EN) = `Disabled`.
Auto-initialization (Auto INIT) = `Disabled`.

In the **Interrupts** tab of the `dmaCfgTransmit`, set the interrupt configuration as follows:

Timeout (TIMEOUT IE) = `Disabled`.
Synchronization Event Drop (DROP IE) = `Disabled`.
Half Frame (FALF IE) = `Disabled`.
Frame Complete (FRAME IE) = `Enabled`.
Last Frame (LAST IE) = `Disabled`.
End Block (BLOCK IE) = `Disabled`.

From the DSP/BIOS configuration file dialog window, select **Chip Support Library** and open DMA→Direct Memory Access Controller. There are six DMA channels in the **DMA Resource Manager**. We set DMA4 for receiving and DMA5 for transmitting for C5510 DSK. Open the DMA4 dialog box by right clicking it and selecting its **Properties**. Enable the **Open Handle to DMA** box and specify the DMA handle name as `C55XX_DMA_MCBSP_hDmaRx` and select `dmaCfgReceive` as shown in Figure 4.41. Open the DMA5 dialog box, enable **Open Handle to DMA** box and specify the handle name as `C55XX_DMA_MCBSP_hDmaTx` and select `dmaCfgTransmit` as shown in Figure 4.42.


exp4_dspbios.cdb

Estimated Data Size: 3906 Est. Min. Stack Size (MAUs): 347

- Input/Output
- Chip Support Library
  - CHIP - Global CSL Device Configuration
  - DMA - Direct Memory Access Controller
    - DMA Configuration Manager
      - dmaCfgReceive
      - dmaCfgTransmit
    - DMA Resource Manager
      - DMA0
      - DMA1
      - DMA2
      - DMA3
      - DMA4
      - DMA5

DMA4 properties

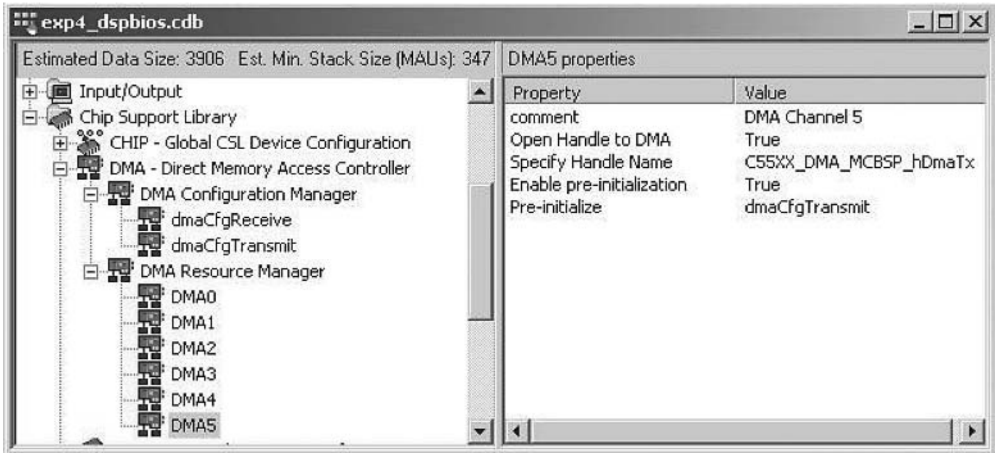
Property	Value
comment	DMA Channel 4
Open Handle to DMA	True
Specify Handle Name	C55XX_DMA_MCBSP_hDmaRx
Enable pre-initialization	True
Pre-initialize	dmaCfgReceive

Figure 4.41 The settings of DMA4 in DSP/BIOS DMA manager

Step 5: *McBSP configuration*

The command and data transfer control between the processor and the AIC23 is via the serial ports as discussed in Chapter 2. The C55x chip support library also provides the McBSP functions through the DSP/BIOS configuration file. From the DSP/BIOS configuration file dialog window, select **Chip Support Library** and open McBSP→Multichannel Buffered Serial Port. Add two new objects to **McBSP Configuration Manager** and rename them as `mcbspCfg1` and `mcbspCfg2`. Open `mcbspCfg1` from the **General** tab to:

Only check the box of Configure DX, PSX, and CLKX as Serial Pins.
Uncheck the box of Configure DR, FSR, and CLKX as Serial Pins if checked.
Breakpoint Emulation = `Stop After Current Word`.
SPI Mode (CLKSTP) = `Falling Edge w/o Delay`.
Digital Loop Back (DLB) = `Disabled`.


exp4_dspbios.cdb

Estimated Data Size: 3906 Est. Min. Stack Size (MAUs): 347

- Input/Output
- Chip Support Library
  - CHIP - Global CSL Device Configuration
  - DMA - Direct Memory Access Controller
    - DMA Configuration Manager
      - dmaCfgReceive
      - dmaCfgTransmit
    - DMA Resource Manager
      - DMA0
      - DMA1
      - DMA2
      - DMA3
      - DMA4
      - DMA5

DMA5 properties

Property	Value
comment	DMA Channel 5
Open Handle to DMA	True
Specify Handle Name	C55XX_DMA_MCBSP_hDmaTx
Enable pre-initialization	True
Pre-initialize	dmaCfgTransmit

Figure 4.42 The settings of DMA5 in DSP/BIOS DMA manager

In the **Transmit Modes** tab of the `mcbspCfg1`, set the configurations as follows:
SPI Clock Mode (CLKXM) = `Master`.
Frame-Sync Polarity (FSXP) = `Active Low`.
DX Pin Delay (DXENA) = `Disabled`.
Transmit Delay (XDATDLY) = `0-bit`.
Detect Sync Error (XSYNCERR) = `Disabled`.
Interrupt Mode (XINTM) = `XRDY`.
Early Frame Sync Response (XFIG) = `Restart Transfer`.
Companding (XCOMPAND) = `No Companding-MSB First`.
Transmit Frame-Sync Source = `DXR(1/2)-to-XSR(1/2) Copy`.
In the **Transmit Lengths** tab of the `mcbspCfg1`, set the configurations as follows:
Phase (XPHASE) = `Single-phase`.
Word Length Phase1 (XWDLEN1) = `16-bit`.
Words/Frame Phase1 (XFRLEN1) = `1`.
In the **Transmit Multichannel** tab of the `mcbspCfg1`, set the configuration as:
TX Channel Enable = `All 128 Channels`.
In the **Sample-Rate Gen** tab of the `mcbspCfg1`, set the configurations as follows:
SRG Clock Source (CLKSM) = `CPU Clock`.
Transmit Frame-Sync Mode (FSXM=1)(FSGM) = `Disabled`.
Frame Width (1-256)(FWID) = `1`.
Clock Divider (1-256)(CLKGDV) = `100`.
Frame Period (1-4096)(FRER) = `20`.
In the **GPIO** tab of the `mcbspCfg1`, set the configurations as follows:
Select CLKR Pin as = `Input`.
Select FSR Pin as = `Input`.
The McBSP 1 is used for command control and McBSP 2 is used for data transfer. McBSP 2 is configured as bidirectional. Open `mcbspCfg2` from the **General** tab to:
Check the box of Configure DX, PSX, and CLKX as Serial Pins.
Check the Configure DR, FSR, and CLKX as Serial Pins if checked.
Breakpoint Emulation = `Stop After Current Word`.
SPI Mode (CLKSTP) = `Disabled`.
Digital Loop Back (DLB) = `Disabled`.
In the **Transmit Modes** tab of the `mcbspCfg2`, set the configurations as follows:
Clock Mode (CLKXM) = `External`.
Clock Polarity (CLKXP) = `Falling Edge`.
Frame-Sync Polarity (FSXP) = `Active High`.
DX Pin Delay (DXENA) = `Disabled`.
Transmit Delay (XDATDLY) = `0-bit`.
Detect Sync Error (XSYNCERR) = `Disabled`.
Interrupt Mode (XINTM) = `XRDY`.
Early Frame Sync Response (XFIG) = `Restart Transfer`.
Companding (XCOMPAND) = `No Companding-MSB First`.
Transmit Frame-Sync Source = `External`.
In the **Transmit Lengths** tab of the `mcbspCfg2`, set the configurations as follows:
Phase (XPHASE) = `Single-phase`.
Word Length Phase1 (XWDLEN1) = `16-bit`.
Words/Frame Phase1 (XFRLEN1) = `2`.
In the **Transmit Multichannel** tab of the `mcbspCfg1`, set the configuration as follows:
TX Channel Enable = `All 128 Channels`.
In the **Receive Modes** tab of the `mcbspCfg2`, set the configurations as follows:

Clock Mode (CLKXM) = `External`.
Clock Polarity (CLKXP) = `Rising Edge`.
Frame-Sync Polarity (FSXP) = `Active High`.
Receive Delay (RDATDLY) = `0-bit`.
Detect Sync Error (RSYNCERR) = `Disabled`.
Interrupt Mode (RINTM) = `RRDY`.
Frame-Sync Mode (FSRM) = `External`.
Early Frame Sync Response (RFIG) = `Restart Transfer`.
Sign-Ext and Justification (RJUST) = `Right-justify/zero-fill`.
Companding (XCOMPAND) = `No Companding-MSB First`.

In the **Receive Lengths** tab of the `mcbspCfg2`, set the configurations as follows:
Phase (RPHASE) = `Single-phase`.
Word Length Phase1 (RWDLEN1) = `16-bit`.
Words/Frame Phase1 (RFRLEN1) = `2`.

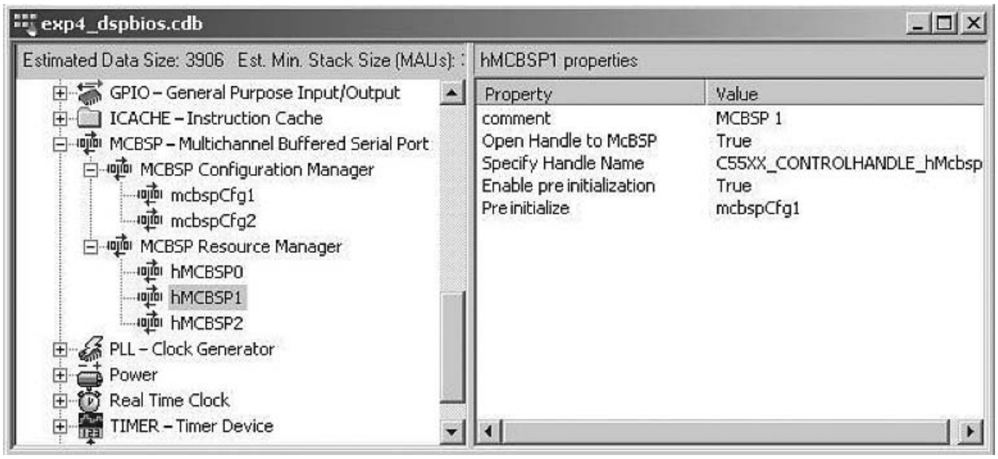
In the **Receive Multichannel** tab of the `mcbspCfg2`, set the configuration as follows:
RX Channel Enable = `All 128 Channels`.

In the **Sample-Rate Gen** tab of the `mcbspCfg2`, set the configurations as follows:
SRG Clock Source (CLKSM) = `CLKS Pin`.
Clock Synchronization with CLKS Pin (GSYNC) = `Disabled`.
CLKS Polarity Clock Edge (From CLKS Pin) (CLKSP) = `Rising Edge of CLKS`.
Frame Width (1–256)(FWID) = `1`.
Clock Divider (1–256)(CLKGDV) = `1`.
Frame Period (1–4096) (FRER) = `1`.

From the DSP/BIOS configuration file dialog window, select **Chip Support Library** and open McBSP→Multichannel Buffered Serial Port. From **McBSP Resource Manager** modify `hMCBSP1` and `hMCBSP2` as shown in Figures 4.43 and 4.44, respectively.

Step 6: *Configuration of hardware interrupts of the DSP/BIOS*

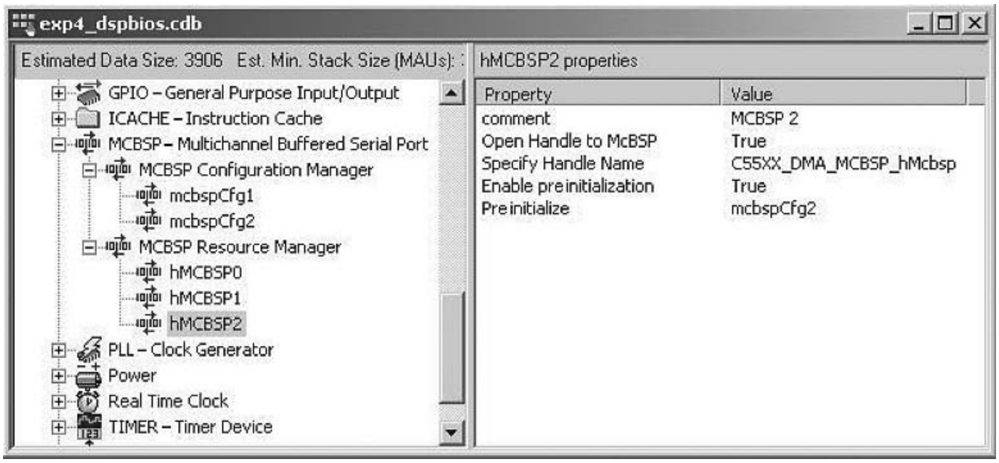
Open the **HWI** under the **Scheduling** from the DSP/BIOS configuration file to connect the interrupts to DSK. Hardware interrupts 14 and 15 are used by the DSK as receive and transmit interrupts. Modify **HWI_INT14** by adding the receive function `_C55XX_DMA_MCBSP_rxIsr`


exp4_dspbios.cdb
Estimated Data Size: 3906 Est. Min. Stack Size (MAUs):
GPIO – General Purpose Input/Output
ICACHE – Instruction Cache
MCBSP – Multichannel Buffered Serial Port
MCBSP Configuration Manager
mcbspCfg1
mcbspCfg2
MCBSP Resource Manager
hMCBSP0
hMCBSP1
hMCBSP2
PLL – Clock Generator
Power
Real Time Clock
TIMER – Timer Device

hMCBSP1 properties

Property	Value
comment	MCBSP 1
Open Handle to McBSP	True
Specify Handle Name	C55XX_CONTROLHANDLE_hMcbsp
Enable pre initialization	True
Pre initialize	mcbspCfg1

Figure 4.43 The settings of McBSP 1 in DSP/BIOS McBSP resource manager

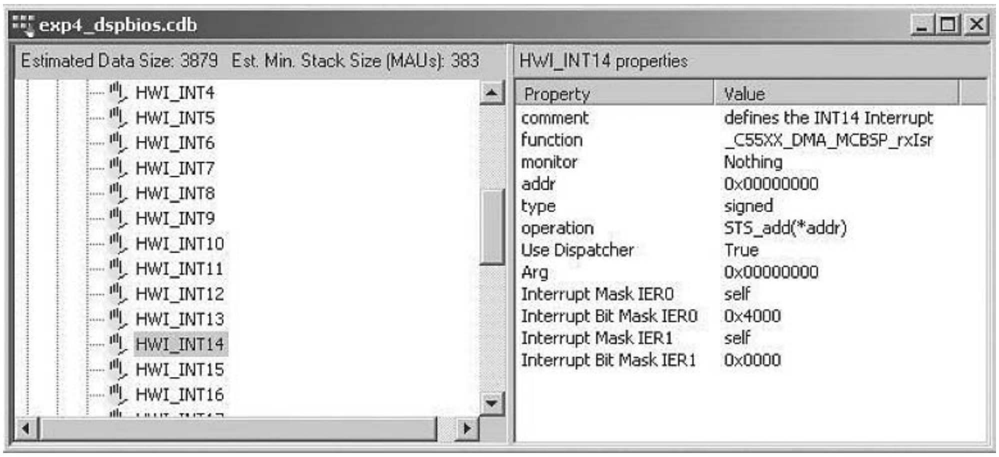

exp4_dspbios.cdb
Estimated Data Size: 3906 Est. Min. Stack Size (MAUs):
GPIO – General Purpose Input/Output
ICACHE – Instruction Cache
MCBSP – Multichannel Buffered Serial Port
MCBSP Configuration Manager
mcbspCfg1
mcbspCfg2
MCBSP Resource Manager
hMCBSP0
hMCBSP1
hMCBSP2
PLL – Clock Generator
Power
Real Time Clock
TIMER – Timer Device
hMCBSP2 properties

Property	Value
comment	MCBSP 2
Open Handle to McBSP	True
Specify Handle Name	C55XX_DMA_MCBSP_hMcbsp
Enable preinitialization	True
Preinitialize	mcbspCfg2

Figure 4.44 The settings of McBSP 2 in DSP/BIOS McBSP resource manager

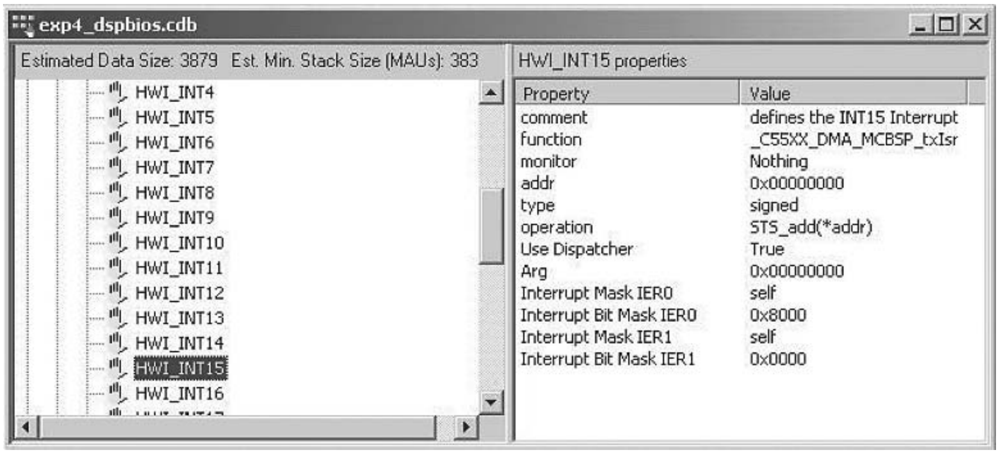
into the **Function** box and check **Use Dispatcher** box under the **Dispatch** tab. Also, modify **HWI_INT15** by adding the transmit function `_C55XX_DMA_MCBSP_txIsr` into the **Function** box and check **Use Dispatcher** box under the **Dispatch** tab as shown in Figures 4.45 and 4.46.

Step 7: *Build and run the real-time DSP/BIOS experiment*
Open the CCS project, set the project to use large memory (-ml option) and add CHIP_5510PG2_2 in **Compiler-Preprocessor-**defined symbol field. We also add the DSK board support library `dsk5510bslx.lib` to the **Linker Include Libraries** search path. When we create the configuration file, the CCS will generate a command file, `dspbioscfg.cmd`. We must use this linker command file. The files used for this experiment are listed in Table 4.9 with brief descriptions. Add the command file `dspbioscfg.cmd` and C and assembly source files listed in Table 4.9, and build this DSP/BIOS project.



Property	Value
comment	defines the INT14 Interrupt
function	_C55XX_DMA_MCBSP_rxIsr
monitor	Nothing
addr	0x00000000
type	signed
operation	STS_add(*addr)
Use Dispatcher	True
Arg	0x00000000
Interrupt Mask IER0	self
Interrupt Bit Mask IER0	0x4000
Interrupt Mask IER1	self
Interrupt Bit Mask IER1	0x0000

Figure 4.45 The settings of receive interrupt in HWI INT14 in DSP/BIOS


exp4_dspbios.cdb

Estimated Data Size: 3879 Est. Min. Stack Size (MAUs): 383

HWI_INT4
HWI_INT5
HWI_INT6
HWI_INT7
HWI_INT8
HWI_INT9
HWI_INT10
HWI_INT11
HWI_INT12
HWI_INT13
HWI_INT14
HWI_INT15
HWI_INT16

HWI_INT15 properties

Property	Value
comment	defines the INT15 Interrupt
function	_C55XX_DMA_MCBSP_txIsr
monitor	Nothing
addr	0x00000000
type	signed
operation	STS_add(*addr)
Use Dispatcher	True
Arg	0x00000000
Interrupt Mask IER0	self
Interrupt Bit Mask IER0	0x8000
Interrupt Mask IER1	self
Interrupt Bit Mask IER1	0x0000

Figure 4.46 The settings of transmit interrupt in HWI INT15 in DSP/BIOS

Procedures of the experiment are listed as follows:

1. Create a DSP/BIOS configuration file and configure the DSK for real-time audio processing application.
2. Create the DSP project and rebuild the project.
3. Connect input and output audio cables to audio source and headphone (or loudspeaker).
4. Load the project and run the program to validate the DSP/BIOS project.

Table 4.9 File listing for experiment `exp4.5.8_realtime_SRC`

Files	Description
`realtime_SRCTest.c`	C function for testing sample rate conversion
`plio.c`	Interface for PIP functions with low level I/O
`interpolate.c`	C function for interpolation filter
`decimate.asm`	Assembly routine for decimation filter
`interpolation.h`	C header file for interpolation
`decimation.h`	C header file for decimation
`coef8to16.h`	FIR filter coefficients for 1:2 interpolation
`coef16to48.h`	FIR filter coefficients for 1:3 interpolation
`coef48to24.h`	FIR filter coefficients for 2:1 decimation
`coef24to8.h`	FIR filter coefficients for 3:1 decimation
`lio.h`	Header file for low level I/O
`plio.h`	Header file for PIP to connect with low level I/O
`DSPBIOS.pjt`	DSP project file
`dspbios.cdb`	DSP/BIOS configuration file
`dspbioscfg.cmd`	DSP/BIOS linker command file

References

[1] N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Englewood Cliffs, NJ: Prentice Hall, 1983.
[2] V. K. Ingle and J. G. Proakis, *Digital Signal Processing Using MATLAB V.4*, Boston: PWS Publishing, 1997.
[3] S. M. Kuo and W. S. Gan, *Digital Signal Processors*, Upper Saddle River, NJ: Prentice Hall, 2005.
[4] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1989.
[5] S. J. Orfanidis, *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1996.
[6] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd Ed., Englewood Cliffs, NJ: Prentice Hall, 1996.
[7] S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, 2nd Ed., New York, NY: McGraw Hill, 1998.
[8] D. Grover and J. R. Deller, *Digital Signal Processing and the Microcontroller*, Englewood Cliffs, NJ: Prentice Hall, 1999.
[9] F. Taylor and J. Mellott, *Hands-On Digital Signal Processing*, New York, NY: McGraw Hill, 1998.
[10] S. D. Stearns and D. R. Hush, *Digital Signal Analysis*, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1990.
[11] The Math Works, Inc., *Signal Processing Toolbox User's Guide*, Version 6, June 2004.
[12] The Math Works, Inc., *Filter Design Toolbox User's Guide*, Version 3, Oct. 2004.
[13] Texas Instruments, Inc., *TMS320C55x Optimizing C Compiler User's Guide*, Literature no. SPRU281E, Mar. 2003.
[14] Texas Instruments, Inc., *TMS320C55x Chip Support Library API Reference Guide*, Literature no. SPRU433J, Sep. 2004.
[15] Texas Instruments, Inc., *TMS320 DSP/BIOS User's Guide*, Literature no. SPRU423, Nov. 2002.
[16] Texas Instruments, Inc., *TMS320C5000 DSP/BIOS Application Programming Interface (API) Reference Guide*, Literature no. SPRU404E, Oct. 2002.

Exercises

1. Consider the moving-average filter given in Example 4.1. What is the 3-dB bandwidth of this filter if the sampling rate is 8 kHz?

2. Consider the FIR filter with the impulse response $h(n) = \{1, 1, 1\}$. Calculate the magnitude and phase responses, and verify that the filter has linear phase.

3. Consider the comb filter designed in Example 4.2 with sampling rate 8 kHz. If a periodic signal with fundamental frequency 500 Hz, and all its harmonics at 1, 1.5, ..., 4 kHz, is filtered by this comb filter, then find out which harmonics will be attenuated and why?

4. Using the graphical interpretation of linear convolution given in Figure 4.7, compute the linear convolution of $h(n) = \{1, 2, 1\}$ and $x(n)$, $n = 0, 1, 2$ defined as follows:

   (a) $x(n) = \{1, -1, 2\}$

   (b) $x(n) = \{1, 2, -1\}$

   (c) $x(n) = \{1, 3, 1\}$

5. The comb filter can also be described as

$$y(n) = x(n) + x(n - L).$$

   Find the transfer function, zeros, and the magnitude response of this filter using MATLAB and compare the results with Figure 4.3 (assume $L = 8$).

6. Assuming $h(n)$ has the symmetry property $h(n) = h(-n)$ for $n = 0, 1, \ldots, M$, verify that $H(\omega)$ can be expressed as

$$H(\omega) = h(0) + \sum_{n=1}^{M} 2h(n)\cos(\omega n).$$

7. The simplest digital approximation to a continuous-time differentiator is the first-order operation defined as

$$y(n) = \frac{1}{T}\left[x(n) - x(n-1)\right].$$

   Find the transfer function $H(z)$, the frequency response $H(\omega)$, and the phase response of the differentiator.

8. Redraw the signal-flow diagram shown in Figure 4.6 and modify Equations (4.22) and (4.23) in the case that L is an odd number.

9. Design a lowpass FIR filter of length $L = 5$ with a linear phase to approximate the ideal lowpass filter of cutoff frequency 1.5 kHz with the sampling rate 8 kHz.

10. Consider the FIR filters with the following impulse responses:

    (a) $h(n) = \{-4, 1, -1, -2, 5, 0, -5, 2, 1, -1, 4\}$

    (b) $h(n) = \{-4, 1, -1, -2, 5, 6, 5, -2, -1, 1, -4\}$

    Use MATLAB to plot magnitude responses, phase responses, and locations of zeros for both filters.

11. Show the frequency response of the lowpass filter given in Equation (4.8) for $L = 8$ and compare the result with Figure 4.3.

12. Use Examples 4.6 and 4.7 to design and plot the magnitude response of a linear-phase FIR highpass filter of cutoff frequency $\omega_c = 0.6\pi$ by truncating the impulse response of the ideal highpass filter to length $L = 2M + 1$ for $M = 32$ and 64.

13. Repeat Problem 12 using Hamming and Blackman window functions. Show that oscillatory behavior is reduced using the windowed Fourier series method.

14. Design a bandpass filter

$$H(f) = \begin{cases} 1, & 1.6\,\text{kHz} \le f \le 2\,\text{kHz} \\ 0, & \text{otherwise} \end{cases}$$

    with the sampling rate 8 kHz and the duration of impulse response 50 ms using Fourier series method; that is, using MATLAB functions `fir1`. Plot the magnitude and phase responses.

15. Repeat Problem 14 using the FDATool using different design methods, and compare results with Problem 14.

16. Redo Example 4.15, quantize the designed coefficients using Q15 format, and save in C header file. Write a floating-point C program to implement this FIR filter and test the result by comparing both input and output signals in terms of time-domain waveforms and frequency-domain spectra.

17. Redo Problem 16 using a fixed-point C, and also use circular buffer.

18. Redo Example 4.12 with different cutoff frequencies and ripples, and summarize their relationship with the required filter order.

19. List the window functions supported by the MATLAB WinTool. Also, use this tool to study the Kaiser window with different L and β.

20. Write a C (or MATLAB) program that implements a comb filter of $L = 8$. The program must have the input/output capability. Test the filter using the sinusoidal signals of frequencies $\omega_1 = \pi/4$ and $\omega_2 = 3\pi/8$. Explain the results based on the distribution of the zeros of the filter.

21. Rewrite above program using a circular buffer.

22. Design a 24th-order bandpass FIR filter using MATLAB. The filter has passband frequencies of 1300–2100 Hz. Implement this filter using the C55x assembly routines `blockFir.asm`, `symFir.asm`, and `dualMacFir.asm`. The test data, `input.pcm`, is sampled at 8 kHz. Plot the filter results in both the time domain and the frequency domain using the CCS graphics.

23. When designing highpass or bandstop FIR filter using MATLAB, the number of filter coefficients is an odd number. This ensures the unit gain at the half-sampling frequency. Design a highpass FIR filter, such that its cutoff frequency is 3000 Hz. Implement this filter using the dual MAC block-FIR filter. Plot the results in both the time domain and the frequency domain. (*Hint*: Modify the assembly routine `dualMacFir.asm` to handle the odd numbered coefficients.)

24. Design an antisymmetric bandpass FIR filter to allow only the middle frequency of the tri-frequency input signal (`input.pcm`) to pass. Use `firssub` instruction to implement the FIR filter and plot the filter results in both the time domain and the frequency domain using the CCS graphics.

25. Use symmetric instruction to implement the decimation function of the experiment `exp4.5.5_decimation`. Compare the run-time efficiency of the function using symmetric instruction implementation and using dual MAC implementation.

26. The assembly routine, `asmIntpFir.asm`, is written for implementing signal interpolation function. However, there are some bugs in the code so it does not work, yet. Debug this assembly program and fix the problems. Test the routine using `exp4.5.6_interpolation`.

27. Implementing a dual MAC assembly routine interpolation function for the experiment `exp4.5.7_SRC`, measure the performance improvement over C function in number of clock cycles.

28. Design a converter to change the 32 kHz sampling rate to 48 kHz.

29. For an experiment given in Section 4.5.7, the approach is to interpolate the 48 kHz signal to 96 kHz and then decimate the 96 kHz signal to 32 kHz. Another approach is to decimate the 48 kHz signal to 16 kHz first and then interpolate the 16 kHz signal to 32 kHz. Will these approaches provide the same result or performance, why? Design an experiment to support your claim.

30. Design an interpolator that converts the 44.1 kHz sampling rate to 48 kHz.

31. Use the TMS320C5510 DSK for the following real-time tasks:
    - Set the TMS320C5510 DSK to 8 kHz sampling rate.
    - Connect the signal source to the audio input of the DSK.
    - Write an interrupt service routine to handle input samples or use DSP/BIOS.
    - Process signal in blocks with 128 samples per block, and apply lowpass filter, highpass filter, and bandpass filter to input signals.

32. Use the TMS320C5510 DSK for the following real-time SRC:
    - Set the TMS320C55x DSK to 16 kHz sampling rate.
    - Connect the signal source to the audio input of the DSK.
    - Write an interrupt service routine to handle input samples or use DSP/BIOS.
    - Process signal in blocks with 160 samples per block.
    - Verify the result using an oscilloscope or spectrum analyzer.

5 Design and Implementation of IIR Filters

In this chapter, we focus on the design, realization, implementation, and applications of digital IIR filters. We will use experiments to demonstrate the implementation of IIR filters in different forms using fixed-point processors.

5.1 Introduction

Designing a digital IIR filter usually begins with the designing of an analog filter, and applies a mapping technique to transform it from the s-plane into the z-plane. Therefore, we will briefly review the Laplace transform, analog filters, mapping properties, and frequency transformation.

5.1.1 Analog Systems

Given a positive time function $x(t) = 0$ for $t < 0$, the one-sided Laplace transform is defined as

$$X(s) = \int_{0}^{\infty} x(t)e^{-st}\,\mathrm{d}t, \tag{5.1}$$

where s is a complex variable defined as

$$s = \sigma + j\Omega, \tag{5.2}$$

and σ is a real number. The inverse Laplace transform is expressed as

$$x(t) = \frac{1}{2\pi j}\int_{\sigma-j\infty}^{\sigma+j\infty} X(s)e^{st}\,\mathrm{d}s. \tag{5.3}$$

The integral is evaluated along the straight line $\sigma + j\Omega$ in the complex plane from $\Omega = -\infty$ to $\Omega = \infty$, which is parallel to the imaginary axis $j\Omega$ at a distance σ from it.

Example 5.1: Find the Laplace transform of function $x(t) = e^{-at}u(t)$, where a is a real number. From Equation (5.1), we have

$$\begin{aligned} X(s) &= \int_0^\infty e^{-at}e^{-st}\,\mathrm{d}t = \int_0^\infty e^{-(s+a)t}\,\mathrm{d}t \\ &= -\frac{1}{s+a}e^{-(s+a)t}\bigg|_0^\infty = \frac{1}{s+a},\ \mathrm{Re}[s] > -a. \end{aligned}$$

Equation (5.2) clearly shows a complex s-plane with a real axis σ and an imaginary axis $j\Omega$. For values of s along the $j\Omega$-axis, i.e., $\sigma = 0$, we have

$$X(s)|_{s=j\Omega} = \int_0^\infty x(t)e^{-j\Omega t}\,\mathrm{d}t, \tag{5.4}$$

which is the Fourier transform of the causal signal $x(t)$. Therefore, given a function $X(s)$, we can find its frequency characteristics by substituting $s = j\Omega$.

If $Y(s)$, $X(s)$, and $H(s)$ are the one-sided Laplace transforms of $y(t)$, $x(t)$, and $h(t)$, respectively, and

$$\begin{aligned} y(t) &= x(t) * h(t) \\ &= \int_0^\infty x(\tau)h(t-\tau)\,\mathrm{d}\tau = \int_0^\infty h(\tau)x(t-\tau)\,\mathrm{d}\tau, \end{aligned} \tag{5.5}$$

we have

$$Y(s) = H(s)X(s). \tag{5.6}$$

Thus, linear convolution in the time domain is equivalent to multiplication in the Laplace (or frequency) domain.

In Equation (5.6), the transfer function of a casual system is defined as

$$H(s) = \frac{Y(s)}{X(s)} = \int_0^\infty h(t)e^{-st}\,\mathrm{d}t, \tag{5.7}$$

where $h(t)$ is the impulse response of the system. The general form of a system transfer function can be expressed as

$$H(s) = \frac{b_0 + b_1 s + \cdots + b_{L-1}s^{L-1}}{a_0 + a_1 s + \cdots + a_M s^M} = \frac{N(s)}{D(s)}. \tag{5.8}$$

The roots of $N(s)$ are the zeros of $H(s)$, while the roots of $D(s)$ are the poles of the system. MATLAB provides the function `freqs` to compute the frequency response $H(\Omega)$ of an analog system $H(s)$.

Example 5.2: The input signal $x(t) = e^{-2t}u(t)$ is applied to an LTI system, and the output of the system is given as $y(t) = \left(e^{-t} + e^{-2t} - e^{-3t}\right) u(t)$. Find the system's transfer function $H(s)$ and the impulse response $h(t)$.

From Example 5.1 for different values of a, we have

$$X(s) = \frac{1}{s+2} \quad \text{and} \quad Y(s) = \frac{1}{s+1} + \frac{1}{s+2} - \frac{1}{s+3}.$$

From Equation (5.7), we obtain

$$H(s) = \frac{Y(s)}{X(s)} = \frac{s^2 + 6s + 7}{(s+1)(s+3)} = 1 + \frac{1}{s+1} + \frac{1}{s+3}.$$

Taking the inverse Laplace transform, we have

$$h(t) = \delta(t) + \left(e^{-t} + e^{-3t}\right) u(t).$$

The stability condition of an analog system can be represented in terms of its impulse response $h(t)$ or its transfer function $H(s)$. A system is stable if

$$\lim_{t\to\infty} h(t) = 0. \tag{5.9}$$

This condition requires that all the poles of $H(s)$ must lie in the left-half of the s-plane, i.e., $\sigma < 0$. If $\lim_{t\to\infty} h(t) \to \infty$, the system is unstable. This condition is equivalent to the system that has one or more poles in the right-half of the s-plane, or has multiple-order pole(s) on the $j\Omega$-axis.

Example 5.3: Consider the system with impulse response $h(t) = e^{-at}u(t)$. This function satisfies Equation (5.9), thus the system is stable for $a > 0$. From Example 5.1, the transfer function of this system is

$$H(s) = \frac{1}{s+a}, \quad a > 0,$$

which has the pole at $s = -a$. Thus, the system is stable since the pole is located at the left-hand side of s-plane. This example shows we can evaluate the stability of system from the impulse response $h(t)$, or from the transfer function $H(s)$.

5.1.2 Mapping Properties

The z-transform can be viewed as the Laplace transform of the sampled function $x(nT)$ by changing of variable

$$z = e^{sT}. \tag{5.10}$$

This relationship represents the mapping of a region in the s-plane to the z-plane because both s and z are complex variables. Since $s = \sigma + j\Omega$, we have

$$z = e^{\sigma T} e^{j\Omega T} = |z|e^{j\omega}, \tag{5.11}$$

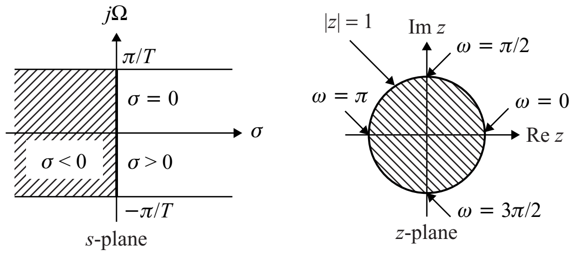


Figure 5.1 Mapping between the s-plane and the z-plane

where the magnitude

$$|z| = e^{\sigma T} \tag{5.12}$$

and the angle

$$\omega = \Omega T. \tag{5.13}$$

When $\sigma = 0$ (the $j\Omega$-axis on the s-plane), the amplitude given in Equation (5.12) is $|z| = 1$ (the unit circle on the z-plane), and Equation (5.11) is simplified to $z = e^{j\Omega T}$. It is apparent that the portion of the $j\Omega$-axis between $\Omega = -\pi/T$ and $\Omega = \pi/T$ in the s-plane is mapped onto the unit circle in the z-plane from $-\pi$ to π as illustrated in Figure 5.1. As Ω increases from π/T to $3\pi/T$, it results in another counterclockwise encirclement of the unit circle. Thus, as Ω varies from 0 to ∞, there are infinite numbers of encirclements of the unit circle in the counterclockwise direction. Similarly, there are infinite numbers of encirclements of the unit circle in the clockwise direction as Ω varies from 0 to $-\infty$.

From Equation (5.12), $|z| < 1$ when $\sigma < 0$. Thus, each strip of width $2\pi/T$ in the left-half of the s-plane is mapped inside the unit circle. This mapping occurs in the form of concentric circles in the z-plane as σ varies from 0 to $-\infty$. Equation (5.12) also implies that $|z| > 1$ if $\sigma > 0$. Thus, each strip of width $2\pi/T$ in the right-half of the s-plane is mapped outside of the unit circle. This mapping also occurs in concentric circles in the z-plane as σ varies from 0 to ∞.

In conclusion, the mapping from the s-plane to the z-plane is not one to one since there are many points in the s-plane that correspond to a single point in the z-plane. This issue will be discussed later when we design a digital filter $H(z)$ from a given analog filter $H(s)$.

5.1.3 Characteristics of Analog Filters

The ideal lowpass filter prototype is obtained by finding a polynomial approximation to the squared magnitude $|H(\Omega)|^2$, and then converting this polynomial into a rational function. The approximations of the ideal prototype will be discussed briefly based on Butterworth filters, Chebyshev type I and type II filters, elliptic filters, and Bessel filters.

The Butterworth lowpass filter is an all-pole approximation to the ideal filter, which is characterized by the squared-magnitude response

$$|H(\Omega)|^2 = \frac{1}{1 + \left(\Omega/\Omega_{\mathrm{p}}\right)^{2L}}, \tag{5.14}$$

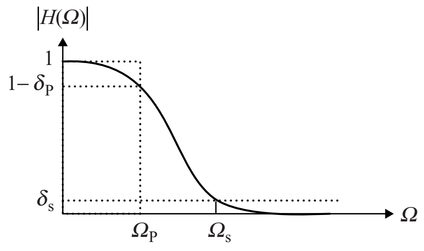


Figure 5.2 Magnitude response of Butterworth lowpass filter

where L is the order of the filter, which determines how closely the Butterworth approximates the ideal filter. Equation (5.14) shows that $|H(0)| = 1$ and $|H(\Omega_\mathrm{p})| = 1/\sqrt{2}$ (or $20\log_{10}|H(\Omega_\mathrm{p})| = -3$ dB) for all values of L. Thus, Ω_p is called the 3-dB cutoff frequency. The magnitude response of a typical Butterworth lowpass filter is monotonically decreasing in both the passband and the stopband as illustrated in Figure 5.2. The Butterworth filter has a flat magnitude response over the passband and stopband, and thus is often referred to as the 'maximally flat' filter. This flat passband is achieved at the expense of slow roll-off in the transition region from Ω_p to Ω_s.

Although the Butterworth filter is easy to design, the rate at which its magnitude decreases in the frequency range $\Omega \geq \Omega_\mathrm{p}$ is rather slow for a small L. Therefore, for a given transition band, the order of the Butterworth filter is often higher than that of other types of filters. We can improve the roll-off by increasing the filter order L.

Chebyshev filters permit a certain amount of ripples, but have a steeper roll-off near the cutoff frequency than the Butterworth filters. There are two types of Chebyshev filters. Type I Chebyshev filters are all-pole filters that exhibit equiripple behavior in the passband and a monotonic characteristic in the stopband (see the top plot of Figure 5.3). Type II Chebyshev filters contain both poles and zeros, and exhibit a monotonic behavior in the passband and an equiripple behavior in the stopband as shown in bottom plot

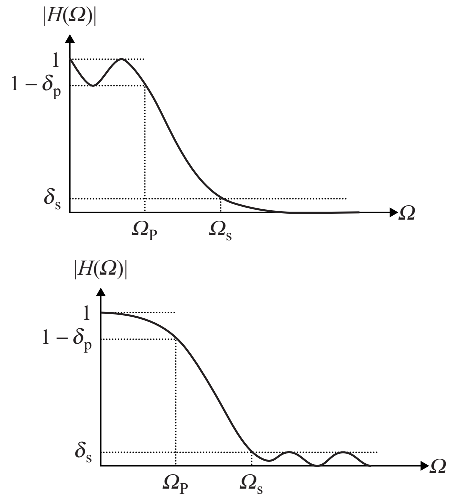


Figure 5.3 Magnitude responses of type I (top) and type II Chebyshev lowpass filters

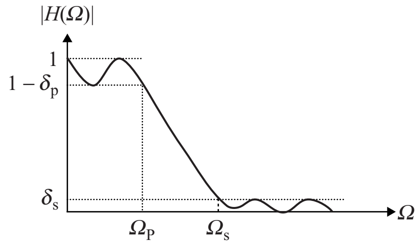


Figure 5.4 Magnitude response of elliptic lowpass filter

of Figure 5.3. In general, a Chebyshev filter meets the specifications with a fewer number of poles than the corresponding Butterworth filter and improves the roll-off; however, it has a poorer phase response.

The sharpest transition from passband to stopband for any given δ_p, δ_s, and L can be achieved using the elliptic filter design. As shown in Figure 5.4, elliptic filters exhibit equiripple behavior in both the passband and the stopband. In addition, the phase response of elliptic filter is extremely nonlinear in the passband, especially near the cutoff frequency. Therefore, we can only use the elliptic design where the phase is not an important design parameter.

In summary, the Butterworth filter has a monotonic magnitude response at both passband and stopband with slow roll-off. By allowing ripples in the passband for type I and in the stopband for type II, the Chebyshev filter can achieve sharper cutoff with the same number of poles. An elliptic filter has even sharper cutoffs than the Chebyshev filter for the same order, but it results in both passband and stopband ripples. The design of these filters strives to achieve the ideal magnitude response with trade-offs in phase response. Bessel filters are all-pole filters that approximate linear phase in the sense of maximally flat group delay in the passband. However, we must sacrifice steepness in the transition region.

5.1.4 Frequency Transforms

We have discussed the design of prototype lowpass filters with cutoff frequency Ω_p. Although the same procedure can be applied to design highpass, bandpass, or bandstop filters, it is easier to obtain these filters from the lowpass filter using frequency transformations. In addition, most classical filter design techniques generate lowpass filters only.

A highpass filter $H_{hp}(s)$ can be obtained from the lowpass filter $H(s)$ by

$$H_{hp}(s) = H(s)|_{s=\frac{1}{s}} = H\left(\frac{1}{s}\right). \tag{5.15}$$

For example, we have Butterworth $H(s) = 1/(s+1)$ for $L = 1$. From Equation (5.15), we obtain

$$H_{hp}(s) = \left.\frac{1}{s+1}\right|_{s=\frac{1}{s}} = \frac{s}{s+1}. \tag{5.16}$$

This shows that $H_{hp}(s)$ has identical pole as the lowpass prototype, but with an additional zero at the origin.

Bandpass filters can be obtained from the lowpass prototypes by replacing s with $(s^2 + \Omega_\mathrm{m}^2)/BW$. That is,

$$H_\mathrm{bp}(s) = H(s)|_{s=\frac{s^2+\Omega_\mathrm{m}^2}{BW}}, \tag{5.17}$$

where Ω_m is the center frequency of the bandpass filter defined as

$$\Omega_\mathrm{m} = \sqrt{\Omega_\mathrm{a}\Omega_\mathrm{b}}, \tag{5.18}$$

where Ω_a and Ω_b are the lower and upper cutoff frequencies, respectively. The filter bandwidth BW is defined as

$$BW = \Omega_\mathrm{b} - \Omega_\mathrm{a}. \tag{5.19}$$

For example, considering $L = 1$, we have

$$H_\mathrm{bp}(s) = \left.\frac{1}{s+1}\right|_{s=\frac{s^2+\Omega_\mathrm{m}^2}{BW}} = \frac{BWs}{s^2 + BWs + \Omega_\mathrm{m}^2}. \tag{5.20}$$

For an Lth-order lowpass filter, we obtain a bandpass filter of order $2L$.

Bandstop filter transfer functions can be obtained from the corresponding highpass filters by

$$H_\mathrm{bs}(s) = H_\mathrm{hp}(s)\Big|_{s=\frac{s^2+\Omega_\mathrm{m}^2}{BW}}. \tag{5.21}$$

5.2 Design of IIR Filters

The transfer function of the IIR filter is defined in Equation (3.42) as

$$H(z) = \frac{\sum_{l=0}^{L-1} b_l z^{-l}}{1 + \sum_{m=1}^{M} a_m z^{-m}}. \tag{5.22}$$

The design problem is to find the coefficients b_l and a_m so that $H(z)$ satisfies the given specifications. The IIR filter can be realized by the I/O equation

$$y(n) = \sum_{l=0}^{L-1} b_l x(n-l) - \sum_{m=1}^{M} a_m y(n-m). \tag{5.23}$$

The problem of designing IIR filters is to determine a digital filter $H(z)$ which approximates the prototype filter $H(s)$ designed by one of the analog filter design methods. There are two methods that can map the analog filter into an equivalent digital filter: the impulse-invariant and the bilinear transform. The impulse-invariant method preserves the impulse response of the original analog filter by digitizing its impulse response, but has inherent aliasing problem. The bilinear transform will preserve the magnitude response characteristics of the analog filters, and thus is better for designing frequency-selective IIR filters.

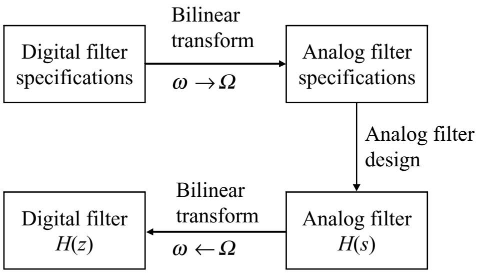

Digital filter specifications
Bilinear transform
$\omega \rightarrow \Omega$
Analog filter specifications
Analog filter design
Analog filter $H(s)$
Bilinear transform
$\omega \leftarrow \Omega$
Digital filter $H(z)$

Figure 5.5 Digital IIR filter design using the bilinear transform

5.2.1 Bilinear Transform

The procedure of digital filter design using bilinear transform is illustrated in Figure 5.5. This method maps the digital filter specifications to an equivalent analog filter. The designed analog filter is then mapped back to obtain the desired digital filter using the bilinear transform.

The bilinear transform is defined as

$$s = \frac{2}{T}\left(\frac{z-1}{z+1}\right) = \frac{2}{T}\left(\frac{1-z^{-1}}{1+z^{-1}}\right). \tag{5.24}$$

This is called the bilinear transform due to the linear functions of z in both the numerator and the denominator. Because the $j\Omega$-axis maps onto the unit circle ($z = e^{j\omega}$), there is a direct relationship between the s-plane frequency Ω and the z-plane frequency ω.

Substituting $s = j\Omega$ and $z = e^{j\omega}$ into Equation (5.24), we have

$$j\Omega = \frac{2}{T}\left(\frac{e^{j\omega}-1}{e^{j\omega}+1}\right). \tag{5.25}$$

It can be easily shown that the corresponding mapping of frequencies is obtained as

$$\Omega = \frac{2}{T}\tan\left(\frac{\omega}{2}\right), \tag{5.26}$$

or equivalently,

$$\omega = 2\tan^{-1}\left(\frac{\Omega T}{2}\right). \tag{5.27}$$

Thus, the entire $j\Omega$-axis is compressed into the interval $[-\pi/T, \pi/T]$ for ω in a one-to-one manner. The portion of $0 \rightarrow \infty$ in the s-plane is mapped onto the $0 \rightarrow \pi$ portion of the unit circle, while the $0 \rightarrow -\infty$ portion in the s-plane is mapped onto the $0 \rightarrow -\pi$ portion of the unit circle. Each point in the s-plane is uniquely mapped onto the z-plane.

The relationship between the frequency variables Ω and ω is illustrated in Figure 5.6. The bilinear transform provides a one-to-one mapping of the points along the $j\Omega$-axis onto the unit circle, or onto the Nyquist band $|\omega| \leq \pi$. However, the mapping is highly nonlinear. The point $\Omega = 0$ is mapped to $\omega = 0$ (or $z = 1$), and the point $\Omega = \infty$ is mapped to $\omega = \pi$ (or $z = -1$). The entire band $\Omega T \geq 1$ is compressed onto $\pi/2 \leq \omega \leq \pi$. This frequency compression effect is known as frequency warping, and

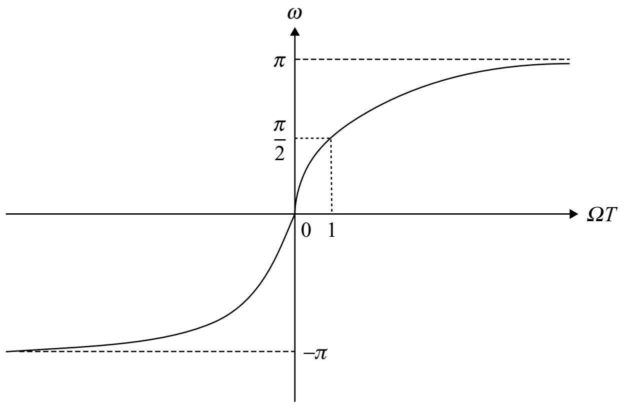


Figure 5.6 Plot of transformation given in Equation (5.27)

must be taken into consideration for digital filter design using the bilinear transform. The solution is to prewarp the critical frequencies according to Equation (5.26).

5.2.2 Filter Design Using Bilinear Transform

The bilinear transform of an analog filter $H(s)$ is obtained by simply replacing s with z using Equation (5.24). The filter specifications will be in terms of the critical frequencies of the digital filter. For example, the critical frequency ω for a lowpass filter is the bandwidth of the filter.

Three steps involved in the IIR filter design using bilinear transform are summarized as follows:

1. Prewarp the critical frequency ω_c of the digital filter using Equation (5.26) to obtain the corresponding analog filter's frequency Ω_c.

2. Scale the analog filter $H(s)$ with Ω_c to obtain the scaled transfer function

$$\hat{H}(s) = H(s)|_{s=s/\Omega_c} = H\left(\frac{s}{\Omega_c}\right). \tag{5.28}$$

3. Replace s using Equation (5.24) to obtain desired digital filter $H(z)$. That is

$$H(z) = \hat{H}(s)|_{s=2(z-1)/(z+1)T}. \tag{5.29}$$

Example 5.4: Using the simple lowpass filter $H(s) = 1/(s+1)$ and the bilinear transform method to design a digital lowpass filter with the bandwidth 1000 Hz and the sampling frequency 8000 Hz.

The critical frequency for the lowpass filter is the bandwidth $\omega_c = 2\pi(1000/8000) = 0.25\pi$, and $T = 1/8000$ s.

Step 1: *Prewarp the critical frequency as*

$$\Omega_c = \frac{2}{T}\tan\left(\frac{\omega_c}{2}\right) = \frac{2}{T}\tan(0.125\pi) = \frac{0.8284}{T}.$$

Step 2: *Use frequency scaling to obtain*

$$\hat{H}(s) = H(s)\big|_{s=s/(0.8284/T)} = \frac{0.8284}{sT + 0.8284}.$$

Step 3: *Using bilinear transform in Equation (5.29) yields the desired transfer function*

$$H(z) = \hat{H}(s)\big|_{s=2(z-1)/(z+1)T} = 0.2929\frac{1+z^{-1}}{1-0.4142z^{-1}}.$$

MATLAB *Signal Processing Toolbox* provides `impinvar` and `bilinear` functions to support impulse-invariant and bilinear transform methods, respectively. For example, we can use numerator and denominator polynomials as follows:

```
[NUMd,DENd] = bilinear(NUM,DEN,Fs, Fp);
```

where `NUMd` and `DENd` are digital filter coefficients obtained from the bilinear function. `NUM` and `DEN` are row vectors containing numerator and denominator coefficients in descending powers of s, respectively, `Fs` is the sampling frequency in Hz, and `Fp` is prewarping frequency.

Example 5.5: In order to design a digital IIR filtering using the bilinear transform, the transfer function of the analog prototype filter is first determined. The numerator and denominator polynomials of the prototype filter are then mapped to the polynomials for the digital filter using the bilinear transform. The following MATLAB script (`example5_5.m`) designs a lowpass filter:

```
Fs = 2000;                             % Sampling frequency
Wn = 300;                              % Edge frequency
Fc = 2*pi*Wn                           % Edge frequency in rad/s
n = 4;                                 % Order of analog filter
[b, a] = butter(n, Fc, 's');           % Design an analog filter
[bz, az] = bilinear(b, a, Fs, Wn);     % Determine digital filter
[Hz,Wz] = freqz(bz,az,512,Fs);         % Display magnitude & phase
```

5.3 Realization of IIR Filters

An IIR filter can be realized in different forms or structures. In this section, we will discuss direct-form I, direct-form II, cascade, and parallel realizations of IIR filters. These realizations are equivalent mathematically, but may have different performance in practical implementation due to the finite wordlength effects.

5.3.1 Direct Forms

The direct-form I realization is defined by the I/O equation (5.23). This filter has $(L + M)$ coefficients and needs $(L + M + 1)$ memory locations to store $\{x(n-l), l = 0, 1, \ldots, L-1\}$ and $\{y(n-m), m = 0, 1, \ldots, M\}$. It also requires $(L + M)$ multiplications and $(L + M - 1)$ additions. The detailed signal-flow diagram for $L = M + 1$ is illustrated in Figure 3.11.

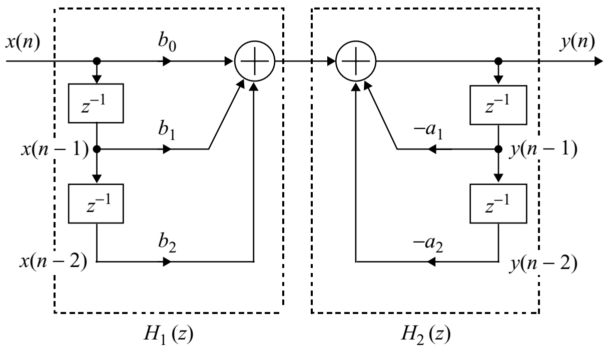


Figure 5.7 Direct-form I realization of second-order IIR filter

Example 5.6: Consider a second-order IIR filter

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \tag{5.30}$$

The I/O equation of the direct-form I realization is described as

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) - a_1 y(n-1) - a_2 y(n-2). \tag{5.31}$$

The signal-flow diagram is illustrated in Figure 5.7.

As shown in Figure 5.7, the IIR filter $H(z)$ can be interpreted as the cascade of two transfer functions $H_1(z)$ and $H_2(z)$. That is,

$$H(z) = H_1(z)H_2(z), \tag{5.32}$$

where $H_1(z) = b_0 + b_1 z^{-1} + b_2 z^{-2}$ and $H_2(z) = 1/\left(1 + a_1 z^{-1} + a_2 z^{-2}\right)$. Since multiplication is commutative, we have $H(z) = H_2(z)H_1(z)$. Therefore, Figure 5.7 can be redrawn by exchanging the order of $H_1(z)$ and $H_2(z)$, and combining two signal buffers into one as illustrated in Figure 5.8. This efficient realization of a second-order IIR filter is called direct-form II (or biquad), which requires three memory

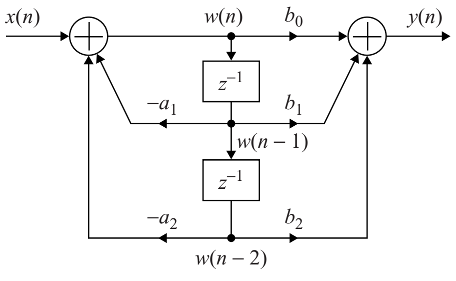


Figure 5.8 Direct-form II realization of second-order IIR filter

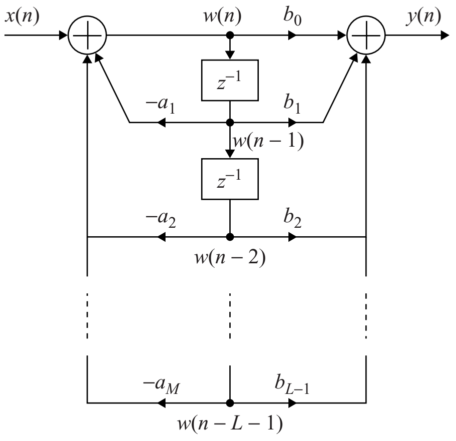


Figure 5.9 Direct-form II realization of general IIR filter, $L = M + 1$

locations as opposed to six memory locations required for the direct-form I given in Figure 5.7. Therefore, the direct-form II is called the canonical form since it needs the minimum numbers of memory.

The direct-form II second-order IIR filter can be implemented as

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2), \tag{5.33}$$

where

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2). \tag{5.34}$$

This realization can be expanded as Figure 5.9 to realize the IIR filter defined in Equation (5.23) with $M = L - 1$ using the direct-form II structure.

5.3.2 Cascade Forms

By factoring the numerator and the denominator polynomials of the transfer function $H(z)$, an IIR filter can be realized as a cascade of second-order IIR filter sections. Consider the transfer function $H(z)$ given in Equation (5.22), it can be expressed as

$$H(z) = b_0 H_1(z) H_2(z) \cdots H_K(z) = b_0 \prod_{k=1}^{K} H_k(z), \tag{5.35}$$

where K is the total number of sections, and $H_k(z)$ is a second-order filter expressed as

$$H_k(z) = \frac{(z - z_{1k})(z - z_{2k})}{(z - p_{1k})(z - p_{2k})} = \frac{1 + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}}. \tag{5.36}$$

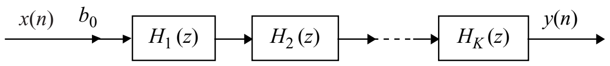


Figure 5.10 Cascade realization of digital filter

If the order is an odd number, one of the $H_k(z)$ is a first-order IIR filter expressed as

$$H_k(z) = \frac{z - z_{1k}}{z - p_{1k}} = \frac{1 + b_{1k}z^{-1}}{1 + a_{1k}z^{-1}}. \tag{5.37}$$

The realization of Equation (5.35) in cascade form is illustrated in Figure 5.10. In this form, any complex-conjugated roots must be grouped into the same section to guarantee that the coefficients of $H_k(z)$ are all real-valued numbers. Assuming that every $H_k(z)$ is a second-order IIR filter described by Equation (5.36), the I/O equations describing the cascade realization are

$$w_k(n) = x_k(n) - a_{1k}w_k(n-1) - a_{2k}w_k(n-2), \tag{5.38}$$

$$y_k(n) = w_k(n) + b_{1k}w_k(n-1) + b_{2k}w_k(n-2), \tag{5.39}$$

$$x_{k+1}(n) = y_k(n), \tag{5.40}$$

for $k = 1, 2, \ldots, K$ where $x_1(n) = b_0x(n)$ and $y(n) = y_K(n)$.

It is possible to obtain many different cascade realizations for the same transfer function $H(z)$ by different ordering and pairing. Ordering means the order of connecting $H_k(z)$, and pairing means the grouping of poles and zeros of $H(z)$ to form $H_k(z)$. In theory, these different cascade realizations are equivalent; however, they may be different due to the finite-wordlength effects. In DSP implementation, each section will generate a certain amount of roundoff error, which is propagated to the next section. The total roundoff noise at the final output will depend on the particular pairing/ordering.

In the direct-form realization shown in Figure 5.9, the variation of one parameter will affect all the poles of $H(z)$. In the cascade realization, the variation of one parameter will only affect pole(s) in that section. Therefore, the cascade realization is preferred in practical implementation because it is less sensitive to parameter variation due to quantization effects.

Example 5.7: Consider the second-order IIR filter

$$H(z) = \frac{0.5(z^2 - 0.36)}{z^2 + 0.1z - 0.72}.$$

By factoring the numerator and denominator polynomials of $H(z)$, we obtain

$$H(z) = \frac{0.5(1 + 0.6z^{-1})(1 - 0.6z^{-1})}{(1 + 0.9z^{-1})(1 - 0.8z^{-1})}.$$

By different pairings of poles and zeros, there are four possible realizations of $H(z)$ in terms of first-order sections. For example, we may choose

$$H_1(z) = \frac{1 + 0.6z^{-1}}{1 + 0.9z^{-1}} \quad \text{and} \quad H_2(z) = \frac{1 - 0.6z^{-1}}{1 - 0.8z^{-1}}.$$

The IIR filter can be realized by the cascade form expressed as

$$H(z) = 0.5H_1(z)H_2(z).$$

5.3.3 Parallel Forms

The expression of $H(z)$ in a partial-fraction expansion leads to another canonical structure called the parallel form expressed as

$$H(z) = c + H_1(z) + H_2(z) + \cdots + H_K(z), \tag{5.41}$$

where c is a constant, and $H_k(z)$ is a second-order IIR filter expressed as

$$H_k(z) = \frac{b_{0k} + b_{1k}z^{-1}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}, \tag{5.42}$$

or a first-order filter expressed as

$$H_k(z) = \frac{b_{0k}}{1 + a_{1k}z^{-1}}. \tag{5.43}$$

The realization of Equation (5.41) in parallel form is illustrated in Figure 5.11. Each second-order section can be implemented as direct-form II shown in Figure 5.8.

Example 5.8: Considering the transfer function $H(z)$ given in Example 5.7, we can express it as

$$H'(z) = \frac{H(z)}{z} = \frac{0.5\left(1 + 0.6z^{-1}\right)\left(1 - 0.6z^{-1}\right)}{z\left(1 + 0.9z^{-1}\right)\left(1 - 0.8z^{-1}\right)} = \frac{A}{z} + \frac{B}{z + 0.9} + \frac{C}{z - 0.8},$$

where
$A = zH'(z)|_{z=0} = 0.25$
$B = (z + 0.9)H'(z)|_{z=-0.9} = 0.147$
$C = (z - 0.8)H'(z)|_{z=0.8} = 0.103.$

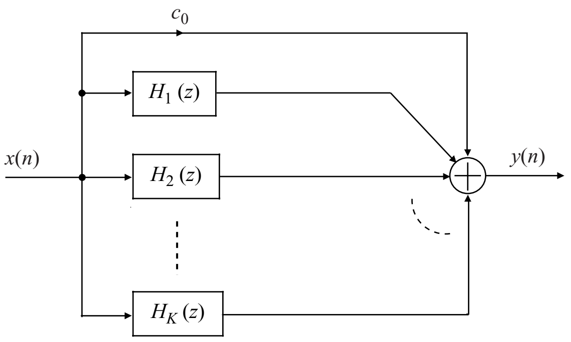


Figure 5.11 A parallel realization of digital IIR filter

Therefore, we obtain

$$H(z) = 0.25 + \frac{0.147}{1 + 0.9z^{-1}} + \frac{0.103}{1 - 0.8z^{-1}}.$$

5.3.4 Realization of IIR Filters Using MATLAB

The cascade realization of an IIR filter involves its factorization. This can be done in MATLAB using the function `roots`. For example, the statement

```
r = roots(b);
```

returns the roots of the numerator vector `b` in the output vector `r`. Similarly, we can obtain the roots of the denominator vector `a`. The coefficients of each section can be determined by pole-zero pairings.

The function `tf2zp` available in the *Signal Processing Toolbox* finds the zeros, poles, and gain of systems. For example, the statement

```
[z, p, c] = tf2zp(b, a);
```

will return the zero locations in `z`, the pole locations in `p`, and the gain in `c`. Similarly, the function

```
[b, a] = zp2tf(z,p,k);
```

forms the transfer function $H(z)$ given a set of zero locations in vector `z`, a set of pole locations in vector `p`, and a gain in scalar `k`.

Example 5.9: The zeros, poles, and gain of the system defined in Example 5.7 can be obtained using the MATLAB script (`example5_9.m`) as follows:

```
b = [0.5, 0, -0.18];
a = [1, 0.1, -0.72];
[z, p, c] = tf2zp(b,a)
```

Runing the program, we obtain `z` $= 0.6, -0.6$, `p` $= -0.9, 0.8$, and `c` $= 0.5$. These results verify the derivation obtained in Example 5.7.

Signal Processing Toolbox also provides a useful function `zp2sos` to convert a zero-pole-gain representation to an equivalent representation of second-order sections. The function

```
[sos, G] = zp2sos(z, p, c);
```

finds the overall gain `G` and a matrix `sos` containing the coefficients of each second-order section determined from its zero-pole form. The matrix `sos` is a $K \times 6$ matrix as

$$sos = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0K} & b_{1K} & b_{2K} & 1 & a_{1K} & a_{2K} \end{bmatrix}, \tag{5.44}$$

where each row contains the numerator and denominator coefficients, b_{ik} and a_{ik}, of the kth second-order section $H_k(z)$. The overall transfer function is expressed as

$$H(z) = G\prod_{k=1}^{K} H_k(z) = G\prod_{k=1}^{K} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}, \tag{5.45}$$

where G is a scalar which accounts for the overall gain of the system.

Similarly, the function `[sos, G] = tf2sos(b, a)` finds a matrix `sos` and a gain `G`. In addition, we can use

```
[sos, G] = tf2sos(b, a, dir_flag, scale);
```

to specify the ordering of the second-order sections. If `dir_flag` is `UP`, the first row will contain the poles closest to the origin, and the last row will contain the poles closest to the unit circle. If `dir_flag` is `DOWN`, the sections are ordered in the opposite direction. The input parameter `scale` specifies the desired scaling of the gain and the numerator coefficients of all second-order sections.

The parallel realizations discussed in Section 5.3.3 can be developed in MATLAB using the function `residuez` in the *Signal Processing Toolbox*. This function converts the transfer function expressed as Equation (5.22) to the partial-fraction-expansion (or residue) form as Equation (5.41). The function

```
[r, p, c] = residuez(b, a);
```

returns the column vector `r` that contains the residues, `p` contains the pole locations, and `c` contains the direct terms.

5.4 Design of IIR Filters Using MATLAB

MATLAB can be used to evaluate the IIR filter design methods, realize and analyze the designed filters, and quantize filter coefficients for fixed-point implementations.

5.4.1 Filter Design Using MATLAB

The *Signal Processing Toolbox* provides a variety of functions for designing IIR filters. This toolbox supports design of Butterworth, Chebyshev type I, Chebyshev type II, elliptic, and Bessel IIR filters in four different types: lowpass, highpass, bandpass, and bandstop. The direct filter design function `yulewalk` finds a filter with magnitude response approximating a desired function, which supports the design of a bandpass filter with multiple passbands. The filter design methods and functions available in the *Signal Processing Toolbox* are summarized in Table 5.1.

Table 5.1 List of IIR filter design methods and functions

Design method	Functions	Description
Order estimation	`buttord`, `cheb1ord`, `cheb2ord`, `ellipord`	Design a digital filter through frequency transformation and bilinear transform using an analog lowpass prototype filter
Design function	`besself`, `butter`, `cheby1`, `cheby2`, `ellip`	
Direct design	`yulewalk`	Design directly by approximating a magnitude response
Generalized design	`maxflat`	Design lowpass Butterworth filters with more zeros than poles

Additional IIR filter design methods are supported by MATLAB *Filter Design Toolbox*, which are summarized as follows:

`iircomb` – IIR comb notching or peaking digital filter design;

`iirgrpdelay` – allpass filter design given a group delay;

`iirlpnorm` – least P-norm optimal IIR filter design;

`iirlpnormc` – constrained least P-norm IIR filter design;

`iirnotch` – second-order IIR notch digital filter design; and

`iirpeak` – second-order IIR peaking (resonator) digital filter design.

We will use `iirpeak` in Section 5.6 for practical application.

As indicated in Table 5.1, the IIR filter design requires two processes. First, compute the minimum filter order `N` and the frequency-scaling factor `Wn` from the given specifications. Second, calculate the filter coefficients using these two parameters. In the first step, the following MATLAB functions are used for estimating filter order:

```
[N, Wn] = buttord(Wp, Ws, Rp, Rs);  % Butterworth filter
[N, Wn] = cheb1ord(Wp, Ws, Rp, Rs); % Chebyshev type I filter
[N, Wn] = cheb2ord(Wp, Ws, Rp, Rs); % Chebyshev type II filter
[N, Wn] = ellip(Wp, Ws, Rp, Rs);    % Elliptic filter
```

The parameters `Wp` and `Ws` are the normalized passband and stopband edge frequencies, respectively. The ranges of `Wp` and `Ws` are between 0 and 1, where 1 corresponds to the Nyquist frequency ($f_s/2$). The parameters `Rp` and `Rs` are the passband ripple and the minimum stopband attenuation specified in dB, respectively. These four functions return the order `N` and the frequency-scaling factor `Wn`, which are needed in the second step of IIR filter design.

In the second step, the *Signal Processing Toolbox* provides the following functions:

```
[b, a] = butter(N, Wn);
[b, a] = cheby1(N, Rp, Wn);
[b, a] = cheby2(N, Rs, Wn);
[b, a] = ellip(N, Rp, Rs, Wn);
[b, a] = besself(N, Wn);
```

These functions return the filter coefficients in row vectors `b` and `a`. We can use `butter(N,Wn,'high')` to design a highpass filter. If `Wn` is a two-element vector, `Wn = [W1 W2]`, `butter` returns an order 2`N` bandpass filter with passband in between `W1` and `W2`, and `butter(N,Wn,'stop')` designs a bandstop filter.

Example 5.10: Design a lowpass Butterworth filter with less than 1.0 dB of ripple from 0 to 800 Hz, and at least 20 dB of stopband attenuation from 1600 Hz to the Nyquist frequency 4000 Hz.

The MATLAB script (`example5_10.m`) for designing the filter is listed as follows:

```
Wp = 800/4000;
Ws= 1600/4000;
Rp = 1.0;
Rs = 20.0;
[N, Wn] = buttord(Wp, Ws, Rp, Rs); % First stage
[b, a] = butter(N, Wn);            % Second stage
freqz(b, a, 512, 8000);            % Display frequency responses
```

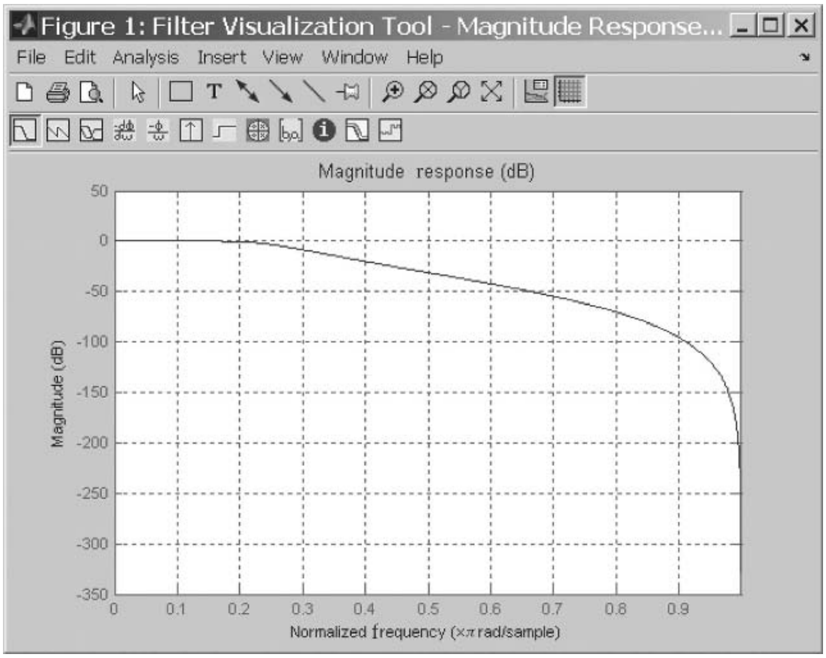


Figure 1: Filter Visualization Tool - Magnitude Response...
File Edit Analysis Insert View Window Help
Magnitude response (dB)
Magnitude (dB)
50
0
-50
-100
-150
-200
-250
-300
-350
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized frequency (×π rad/sample)

Figure 5.12 Filter visualization tool window

Instead of using `freqz` for display magnitude and phase responses, we can use a graphical user interface (GUI) tool called the Filter Visualization Tool (FVTool) to analyze digital filters. The following command

```
fvtool(b,a)
```

launches the FVTool and computes the magnitude response for the filter defined by numerator and denominator coefficients in vectors `b` and `a`, respectively. For example, after execution of `example5_10.m`, when you type in `fvtool(b,a)` in the MATLAB command window, Figure 5.12 is displayed. From the **Analysis** menu, we can further analyze the designed filter.

Example 5.11: Design a bandpass filter with passband of 100–200 Hz, and the sampling rate is 1 kHz. The passband ripple is less than 3 dB and the stopband attenuation is at least 30 dB by 50 Hz out on both sides of the passband.

The MATLAB script (`example5_11.m`) for designing and evaluating filter is listed as follows:

```
Wp = [100  200]/500;
Ws = [50  250]/500;
Rp = 3;
Rs = 30;
[N, Wn] = buttord(Wp, Ws, Rp, Rs);
[b, a] = butter(N, Wn); % Design a Butterworth filter
fvtool(b, a);           % Analyze the designed IIR filter
```

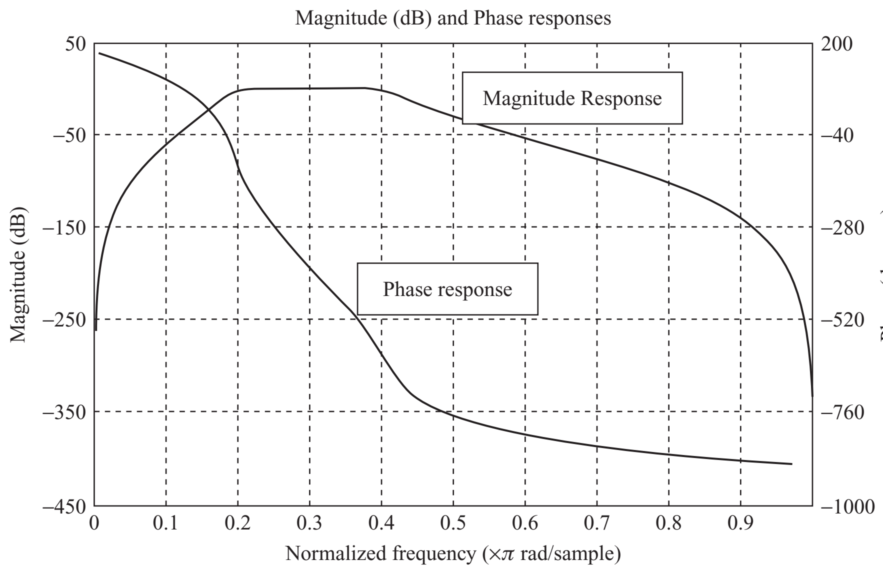


Figure 5.13 Magnitude and phase responses of the bandpass filter

From the **Analysis** menu in the FVTool window, we select the **Magnitude and Phase Responses**. The magnitude and phase responses of the designed bandpass filter are shown in Figure 5.13.

5.4.2 Frequency Transforms Using MATLAB

The *Signal Processing Toolbox* provides functions `lp2hp`, `lp2bp`, and `lp2bs` for converting the prototype lowpass filters to highpass, bandpass, and bandstop filters, respectively. For example, the following command

```
    [numt,dent] = lp2hp(num,den,wo);
```

transforms the lowpass filter prototype `num/den` with unity cutoff frequency to a highpass filter with cutoff frequency `wo`.

The *Filter Design Toolbox* provides additional frequency transformations via numerator and denominator functions that are listed as follows:

`iirbpc2bpc` – complex bandpass to complex bandpass;

`iirlp2bp` – real lowpass to real bandpass;

`iirlp2bpc` – real lowpass to complex bandpass;

`iirlp2bs` – real lowpass to real bandstop;

`iirlp2bsc` – real lowpass to complex bandstop;

`iirlp2hp` – real lowpass to real highpass;

`iirlp2lp` – real lowpass to real lowpass;

`iirlp2mb` – real lowpass to real multiband; and

`iirlp2mbc` – real lowpass to complex multiband.

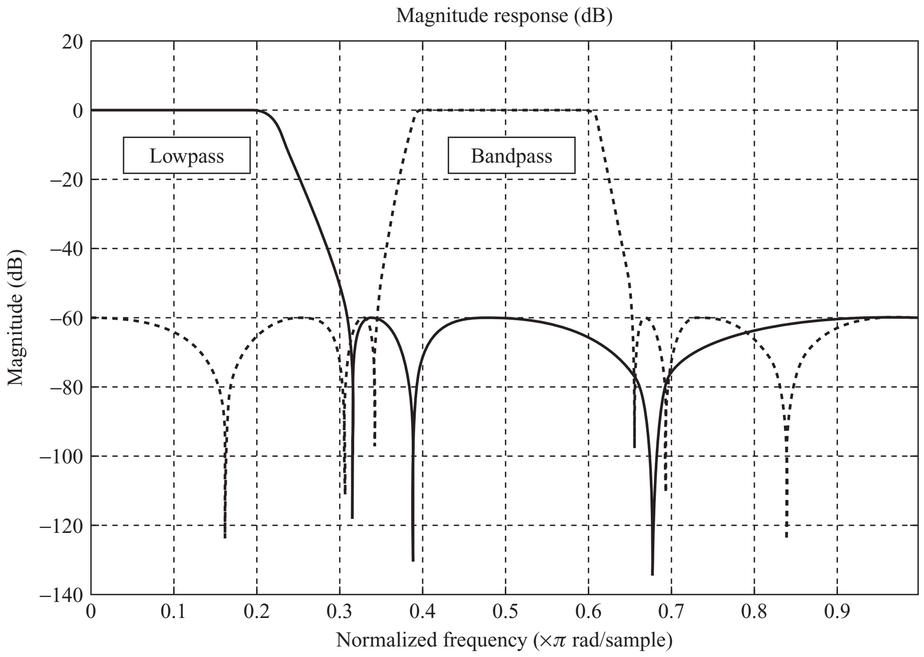


Figure 5.14 Magnitude responses of the lowpass and bandpass filters

Example 5.12: The function `iirlp2bp` converts an IIR lowpass to an bandpass filter with the following syntax:

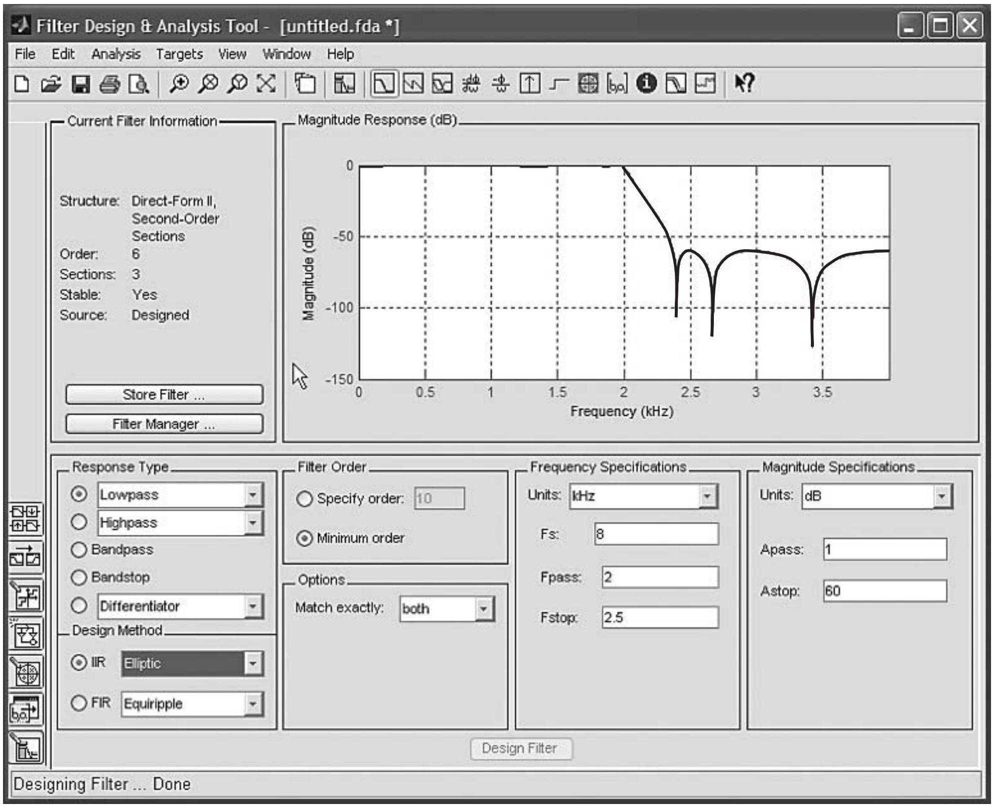
```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(b,a,Wo,Wt);
```

This functions returns numerator and denominator vectors, `Num` and `Den` of the transformed lowpass digital filter. It also returns the numerator `AllpassNum` and the denominator `AllpassDen` of the allpass mapping filter. The prototype lowpass filter is specified by numerator `b` and denominator `a`, `Wo` is the center frequency value to be transformed from the prototype filter, and `Wt` is the desired frequency in the transformed filter. Frequencies must be normalized to be between 0 and 1. The following MATLAB script (`example5_12.m`, adapted from the **Help** menu) converts a lowpass filter to a bandpass filter and analyzes it using FVTool as shown in Figure 5.14:

```
[b,a] = ellip(6,0.1,60,0.209);             % Lowpass filter
[num,den] = iirlp2bp(b,a,0.5,[0.25,0.75]); % Convert to bandpass
fvtool(b,a,num,den); % Display both lowpass & bandpass filters
```

5.4.3 Design and Realization Using FDATool

In this section, we use the FDATool shown in Figure 4.18 for designing, realizing, and quantizing IIR filters. To design an IIR filter, select the radio button next to **IIR** in the **Design Method** region on the GUI. There are seven options (from the pull-down menu) for **Lowpass** types, and several different filter design methods are available for different response types.



Filter Design & Analysis Tool - [untitled.fda *]

Current Filter Information

Structure: Direct-Form II, Second-Order Sections
Order: 6
Sections: 3
Stable: Yes
Source: Designed

Response Type: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator

Design Method: [✓ IIR Elliptic] FIR Equiripple

Filter Order: Specify order: 10 [✓ Minimum order]

Options: Match exactly: both

Frequency Specifications: Units: kHz; Fs: 8; Fpass: 2; Fstop: 2.5

Magnitude Specifications: Units: dB; Apass: 1; Astop: 60

Designing Filter ... Done

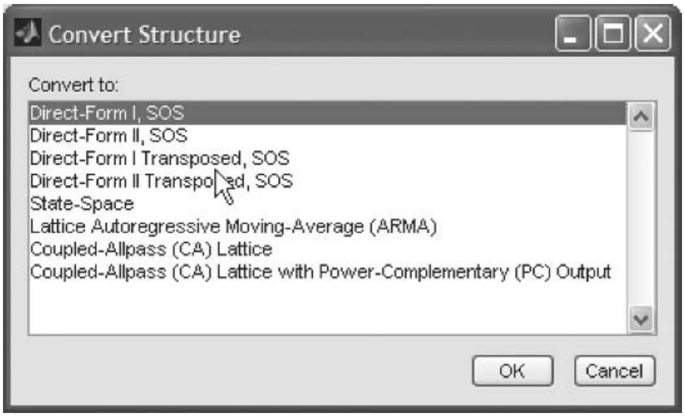
Figure 5.15 GUI of designing an elliptic IIR lowpass filter

Example 5.13: Similar to Example 4.12, design a lowpass IIR filter with the following specifications: sampling frequency $f_s = 8\,\text{kHz}$, passband cutoff frequency $\omega_p = 2\,\text{kHz}$, stopband cutoff frequency $\omega_s = 2.5\,\text{kHz}$, passband ripple $A_p = 1\,\text{dB}$, and stopband attenuation $A_s = 60\,\text{dB}$.

We can design an elliptic filter by clicking the radio button next to **IIR** in the **Design Method** region and selecting **Elliptic** from the pull-down menu. We then enter parameters in **Frequency Specifications** and **Magnitude Specifications** regions as shown in Figure 5.15. After pressing **Design Filter** button to compute the filter coefficients, the **Filter Specifications** region changed to a **Magnitude Response (dB)** as shown in Figure 5.15.

We can specify filter order by clicking the radio button **Specify Order** and entering the filter order in a text box, or choose the default **Minimum Order**. The order of the designed filter is 6, which is stated in **Current Filter Information** region (top-left) as shown in Figure 5.15. By default, the designed IIR filter was realized by cascading of second-order IIR sections using the direct-form II biquads shown in Figure 5.8. We can change this default setting from Edit→Convert Structure, the dialog window shown in Figure 5.16 displayed for selecting different structures. We can reorder and scale second-order sections by selecting Edit→Reorder and Scale Second-Order Sections.

Once the filter has been designed and verified as shown in Figure 5.15, we can turn on the quantization mode by clicking the **Set Quantization Parameters** button. The bottom-half of the FDATool window


Convert Structure

Convert to:

- Direct-Form I, SOS
- Direct-Form II, SOS
- Direct-Form I Transposed, SOS
- Direct-Form II Transposed, SOS
- State-Space
- Lattice Autoregressive Moving-Average (ARMA)
- Coupled-Allpass (CA) Lattice
- Coupled-Allpass (CA) Lattice with Power-Complementary (PC) Output

Figure 5.16 Convert filter structure window

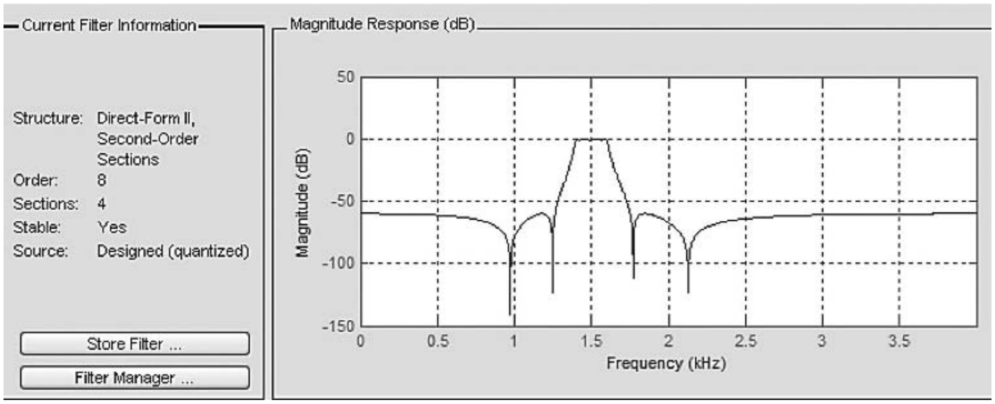
will change to a new pane with the **Filter Arithmetic** option allowing the user to quantize the designed filter and analyzing the effects of changing quantization settings. To enable the fixed-point quantization, select **Fixed-Point** from the **Filter Arithmetic** pull-down menu. See Section 4.2.5 for details of those options and settings.

Example 5.14: Design a quantized bandpass IIR filter for a 16-bit fixed-point DSP processor with the following specifications: sampling frequency = 8000 Hz, lower stopband cutoff frequency $F_{\text{stop1}} = 1200$ Hz, lower passband cutoff frequency $F_{\text{pass1}} = 1400$ Hz, upper passband cutoff frequency $F_{\text{pass2}} = 1600$ Hz, upper stopband cutoff frequency $F_{\text{stop2}} = 1800$ Hz, passband ripple = 1 dB, and stopband (both lower and upper) attenuation = 60 dB.

Start FDATool and enter the appropriate parameters in the **Frequency Specifications** and **Magnitude Specifications** regions, select elliptic IIR filter type, and click **Design Filter**. The order of designed filter is 16 with eight second-order sections. Click the **Set Quantization Parameters** button, and select the **Fixed-Point** option from the pull-down menu of **Filter Arithmetic** and use default settings. After designing and quantizing the filter, select the **Magnitude Response Estimate** option on the **Analysis** menu for estimating the frequency response for quantized filter. The magnitude response of the quantized filter is displayed in the analysis area as shown in Figure 5.17. We observe that quantizing the coefficients has satisfactory filter magnitude response, primarily because FDATool implements the filter in cascade second-order sections, which is more resistant to the effects of coefficient quantization.

We also select **Filter Coefficients** from the **Analysis** menu, and display it in Figure 5.18. It shows both the quantized coefficients (top) with Q15 format and the original coefficients (bottom) with double-precision floating-point format.

We can save the designed filter coefficients in a C header file by selecting **Generate C header** from the **Targets** menu. The **Generate C Header** dialog box appears as shown in Figure 5.19. For an IIR filter, variable names in the C header file are numerator (`NUM`), numerator length (`NL`), denominator (`DEN`), denominator length (`DL`), and number of sections (`NS`). We can use the default variable names as shown in Figure 5.19, or change them to match the names used in the C program that will include this header file. Click **Generate**, and the **Generate C Header** dialog box appears. Enter the filename and click **Save** to save the file.



Structure: Direct-Form II, Second-Order Sections
Order: 8
Sections: 4
Stable: Yes
Source: Designed (quantized)

Figure 5.17 FDATool window for a quantized 16-bit bandpass filter

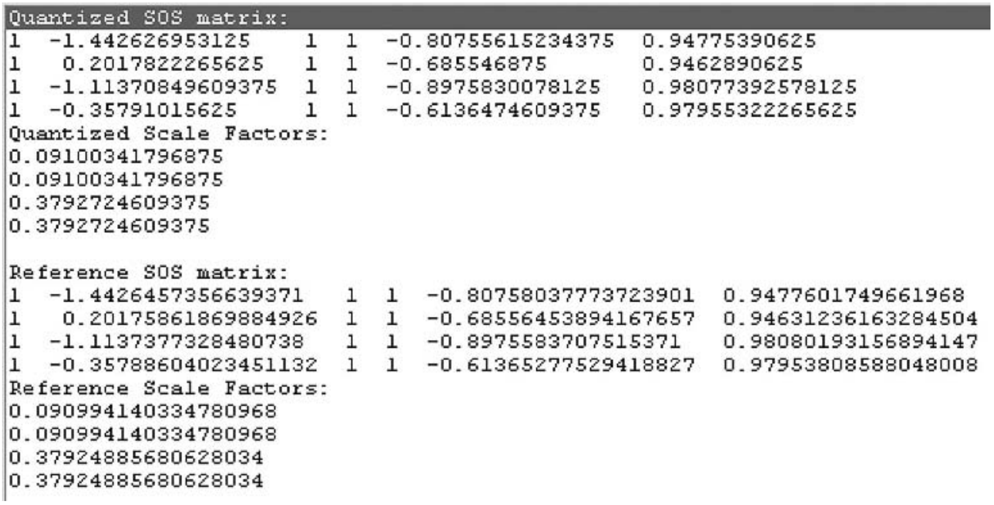
5.5 Implementation Considerations

This section discusses important considerations for implementing IIR filters, including stability and finite wordlength effects.

5.5.1 Stability

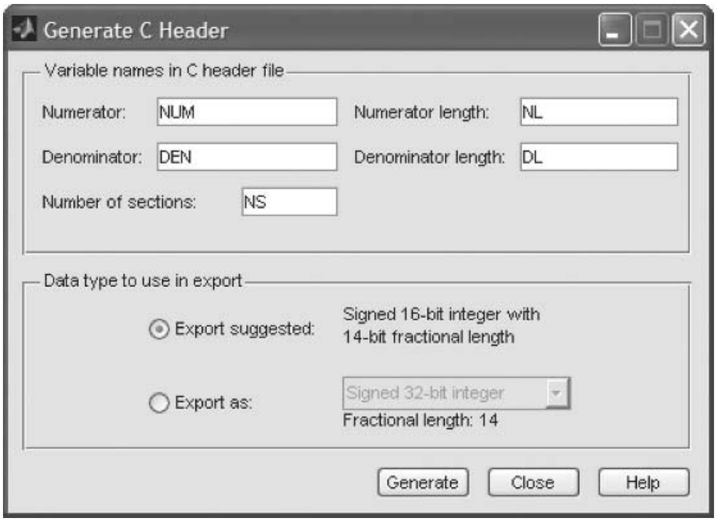
The IIR filter defined by the transfer function given in Equation (3.44) is stable if all the poles lie within the unit circle. That is,

$$|p_m| < 1, m = 1, 2, \dots, M. \tag{5.46}$$



```
Quantized SOS matrix:
1  -1.442626953125    1  1  -0.80755615234375   0.94775390625
1   0.2017822265625   1  1  -0.685546875        0.9462890625
1  -1.11370849609375  1  1  -0.8975830078125    0.98077392578125
1  -0.35791015625     1  1  -0.6136474609375    0.97955322265625
Quantized Scale Factors:
0.09100341796875
0.09100341796875
0.3792724609375
0.3792724609375

Reference SOS matrix:
1  -1.4426457356639371   1  1  -0.80758037773723901  0.9477601749661968
1   0.20175861869884926  1  1  -0.68556453894167657  0.94631236163284504
1  -1.1137377328480738   1  1  -0.8975583707515371   0.98080193156894147
1  -0.35788604023451132  1  1  -0.61365277529418827  0.97953808588048008
Reference Scale Factors:
0.090994140334780968
0.090994140334780968
0.3792488568062803 4
0.3792488568062803 4
```

Figure 5.18 Filter coefficients


Generate C Header

Variable names in C header file

Numerator: NUM   Numerator length: NL

Denominator: DEN   Denominator length: DL

Number of sections: NS

Data type to use in export

[✓ Export suggested:] Signed 16-bit integer with 14-bit fractional length

Export as: Signed 32-bit integer
Fractional length: 14

Figure 5.19 Generate C header dialog box

In this case, we can show that $\lim_{n\to\infty} h(n) = 0$. If $|p_m| > 1$ for any m, then the IIR filter is unstable since $\lim_{n\to\infty} h(n) \to \infty$. In addition, an IIR filter is unstable if $H(z)$ has multiple-order pole(s) on the unit circle.

Example 5.15: Considering the IIR filter with transfer function

$$H(z) = \frac{1}{1 - az^{-1}},$$

the impulse response of the system is $h(n) = a^n, n \geq 0$. If the pole is inside the unit circle, i.e., $|a| < 1$, the impulse response $\lim_{n\to\infty} h(n) = \lim_{n\to\infty} a^n \to 0$. Thus, the IIR filter is stable. However, the IIR filter is unstable for $|a| > 1$ since the pole is outside the unit circle and

$$\lim_{n\to\infty} h(n) = \lim_{n\to\infty} a^n \to \infty \text{ if } |a| > 1.$$

Example 5.16: Considering the system with transfer function

$$H(z) = \frac{z}{(z-1)^2},$$

there is a second-order pole at $z = 1$. The impulse response of the system is $h(n) = n$, which is an unstable system.

An IIR filter is marginally stable (or oscillatory bounded) if

$$\lim_{n\to\infty} h(n) = c, \tag{5.47}$$

where c is a nonzero constant. For example, if $H(z) = 1/1 + z^{-1}$, there is a first-order pole on the unit circle. It is easy to show that the impulse response oscillates between ± 1 since $h(n) = (-1)^n, n \geq 0$.

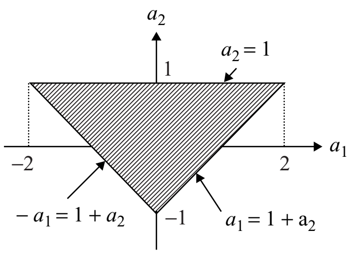


Figure 5.20 Region of coefficient values for a stable second-order IIR filter

Consider the second-order IIR filter defined by Equation (5.30). The denominator can be factored as

$$1 + a_1 z^{-1} + a_2 z^{-2} = \left(1 - p_1 z^{-1}\right)\left(1 - p_2 z^{-1}\right), \tag{5.48}$$

where

$$a_1 = -\left(p_1 + p_2\right) \quad \text{and} \quad a_2 = p_1 p_2. \tag{5.49}$$

The poles must lie inside the unit circle for stability; that is, $|p_1| < 1$ and $|p_2| < 1$.

From Equation (5.49), we need

$$|a_2| = |p_1 p_2| < 1 \tag{5.50}$$

for a stable system. The corresponding condition on a_1 can be derived from the Schur–Cohn stability test as

$$|a_1| < 1 + a_2. \tag{5.51}$$

Stability conditions in Equations (5.50) and (5.51) are illustrated in Figure 5.20, which shows the resulting stability triangle in the $a_1 - a_2$ plane. That is, the second-order IIR filter is stable if and only if the coefficients define a point (a_1, a_2) that lies inside the stability triangle.

5.5.2 Finite-Precision Effects and Solutions

In practical applications, the coefficients obtained from filter design are quantized to a finite number of bits for implementation. The filter coefficients, b_l and a_m, obtained by MATLAB are represented using double-precision floating-point format. Let b'_l and a'_m denote the quantized values corresponding to b_l and a_m, respectively. The transfer function of quantized IIR filter is expressed as

$$H'(z) = \frac{\sum_{l=0}^{L-1} b'_l z^{-l}}{1 + \sum_{m=1}^{M} a'_m z^{-m}}. \tag{5.52}$$

If the wordlength is not sufficiently large, some undesirable effects will occur. For example, the magnitude and phase responses of $H'(z)$ may be different from those of $H(z)$. If the poles of $H(z)$ are close to the unit circle, the pole(s) of $H'(z)$ may move outside the unit circle after coefficient quantization, resulting in an unstable implementation. These undesired effects are more serious when higher order IIR filters are implemented using the direct-form realization. Therefore, the cascade and parallel realizations are preferred in practical DSP implementations with each $H_k(z)$ be a first- or second-order section. The cascade form is recommended for the implementation of high-order narrowband IIR filters that have closely clustered poles.

Example 5.17: Consider the IIR filter with transfer function

$$H(z) = \frac{1}{1 - 0.85z^{-1} + 0.18z^{-2}},$$

with the poles located at $z = 0.4$ and $z = 0.45$. This filter can be realized in the cascade form as $H(z) = H_1(z)H_2(z)$, where $H_1(z) = \frac{1}{(1-0.4z^{-1})}$ and $H_2(z) = \frac{1}{(1-0.45z^{-1})}$.

If this IIR filter is implemented on a 4-bit (a sign bit plus three data bits; see Table 3.2) fixed-point hardware, 0.85 and 0.18 are quantized to 0.875 and 0.125, respectively. Therefore, the direct-form realization is described as

$$H'(z) = \frac{1}{1 - 0.875z^{-1} + 0.125z^{-2}}.$$

The poles of the direct-form $H'(z)$ become $z = 0.1798$ and $z = 0.6952$, which are significantly different than the original 0.4 and 0.45.

For cascade realization, the poles 0.4 and 0.45 are quantized to 0.375 and 0.5, respectively. The quantized cascade filter is expressed as

$$H''(z) = \frac{1}{1 - 0.375z^{-1}} \cdot \frac{1}{1 - 0.5z^{-1}}.$$

The poles of $H''(z)$ are $z = 0.375$ and $z = 0.5$. Therefore, the poles of cascade realization are closer to the desired $H(z)$ at $z = 0.4$ and $z = 0.45$.

Rounding of $2B$-bit product to B bits introduces the roundoff noise. The order of cascade sections influences the output noise power due to roundoff. In addition, when digital filters are implemented using fixed-point processors, we have to optimize the ratio of signal power to the power of the quantization noise. This involves a trade-off with the probability of arithmetic overflow. The most effective technique in preventing overflow is to use scaling factors at various nodes within the filter sections. The optimization is achieved by keeping the signal level as high as possible at each section without getting overflown.

Example 5.18: Consider the first-order IIR filter with scaling factor α described by

$$H(z) = \frac{\alpha}{1 - az^{-1}},$$

where stability requires that $|a| < 1$. The goal of including the scaling factor α is to ensure that the values of $y(n)$ will not exceed 1 in magnitude. Suppose that $x(n)$ is a sinusoidal signal of frequency

ω_0, the amplitude of the output is a factor of $|H(\omega_0)|$. For such signals, the gain of $H(z)$ is

$$\max_{\omega} |H(\omega)| = \frac{\alpha}{1 - |a|}.$$

Thus, if the signals being considered are sinusoidal, a suitable scaling factor is given by $\alpha < 1 - |a|$.

5.5.3 MATLAB Implementations

The MATLAB function `filter` implements the IIR filter defined by Equation (5.23). The basic forms of this function are

```
y = filter(b, a, x);
y = filter(b, a, x, zi);
```

The first element of vector `a`, the first coefficient `a(1)`, is assumed to be 1. The input vector is `x`, and the filter output vector is `y`. At the beginning, the initial conditions (data in the signal buffers) are set to zero. However, they can be specified in the vector `zi` to reduce transients.

Example 5.19: Given a signal consists of sinewave (150 Hz) corrupted by white noise with SNR = 0 dB, and the sampling rate is 1000 Hz. To enhance the sinewave, we need a bandpass filter with passband centered at 150 Hz. Similar to Example 5.11, we design a bandpass filter with the following MATLAB functions:

```
Wp = [130  170]/500; % Passband edge frequencies
Ws = [100  200]/500; % Stopband edge frequencies
Rp = 3;              % Passband ripple
Rs = 40;             % Stopband ripple
[N, Wn] = buttord(Wp, Ws, Rp, Rs); % Find the filter order
[b, a] = butter(N, Wn); % Design an IIR filter
```

We implement the designed filter using the following function:

```
y = filter(b, a, xn);  % IIR filtering
```

We then plot the input and output signals in the `xn` and `y` vectors, which are displayed in Figure 5.21. The complete MATLAB script for this example is given in `example5_19.m`.

MATLAB *Signal Processing Toolbox* also provides the second-order (biquad) IIR filtering function with the following syntax:

```
y = sosfilt(sos,x)
```

This function applies the IIR filter $H(z)$ with second-order sections `sos` as defined in Equation (5.44) to the vector `x`.

Example 5.20: In Example 5.19, we design a bandpass filter and implement the direct-form IIR filter using the function `filter`. In this example, we convert the direct-form filter to cascade of second-order sections using the following function:

```
sos = tf2sos(b,a);
```

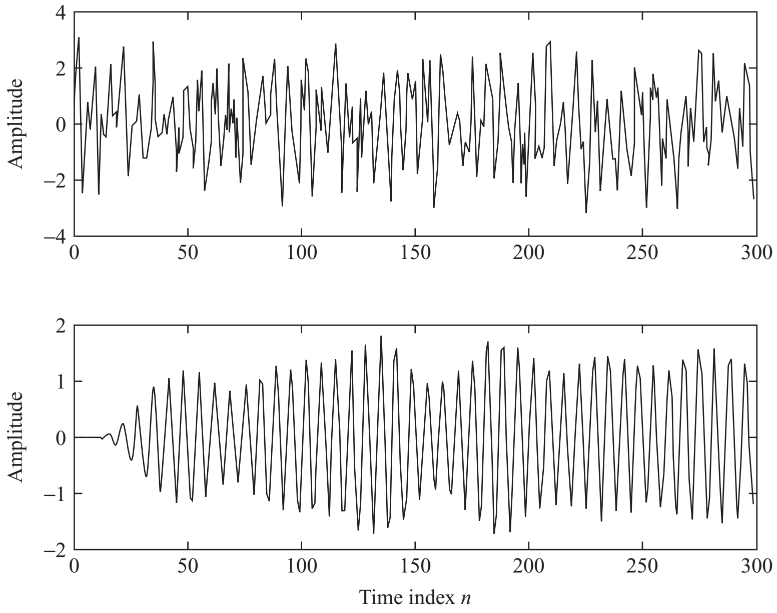


Figure 5.21 Input (top) and output (bottom) signals of bandpass filter

The `sos` matrix is shown as follows:

```
sos =
    0.0000    0.0000    0.0000    1.0000   -0.9893    0.7590
    1.0000    2.0000    1.0000    1.0000   -1.0991    0.7701
    1.0000    1.9965    0.9965    1.0000   -0.9196    0.8119
    1.0000   -2.0032    1.0032    1.0000   -1.2221    0.8350
    1.0000   -1.9968    0.9968    1.0000   -0.9142    0.9257
    1.0000   -2.0000    1.0000    1.0000   -1.3363    0.9384
```

We then perform the IIR filtering using the following function:

```
y = sosfilt(sos,xn);
```

The complete MATLAB program for this example is `example5_20.m`.

The Signal Processing Tool (SPTool) supports the user to analyze signals, design and analyze filters, perform filtering, and analyze the spectra of signals. We can open this tool by typing

```
sptool
```

in the MATLAB command window. The SPTool main window is shown in Figure 5.22.

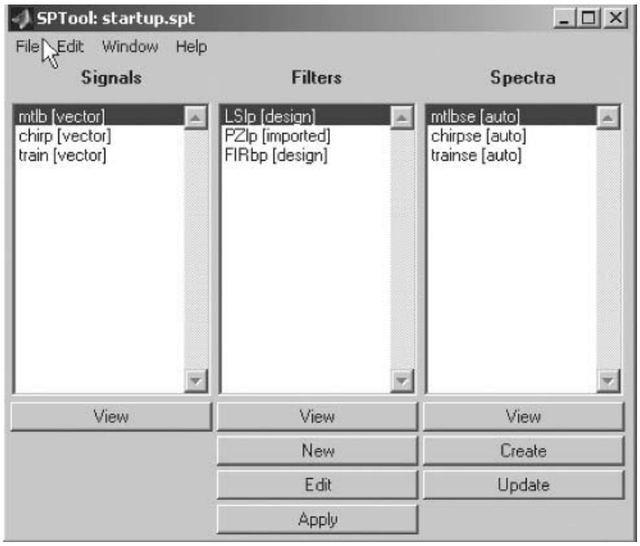


Figure 5.22 SPTool window

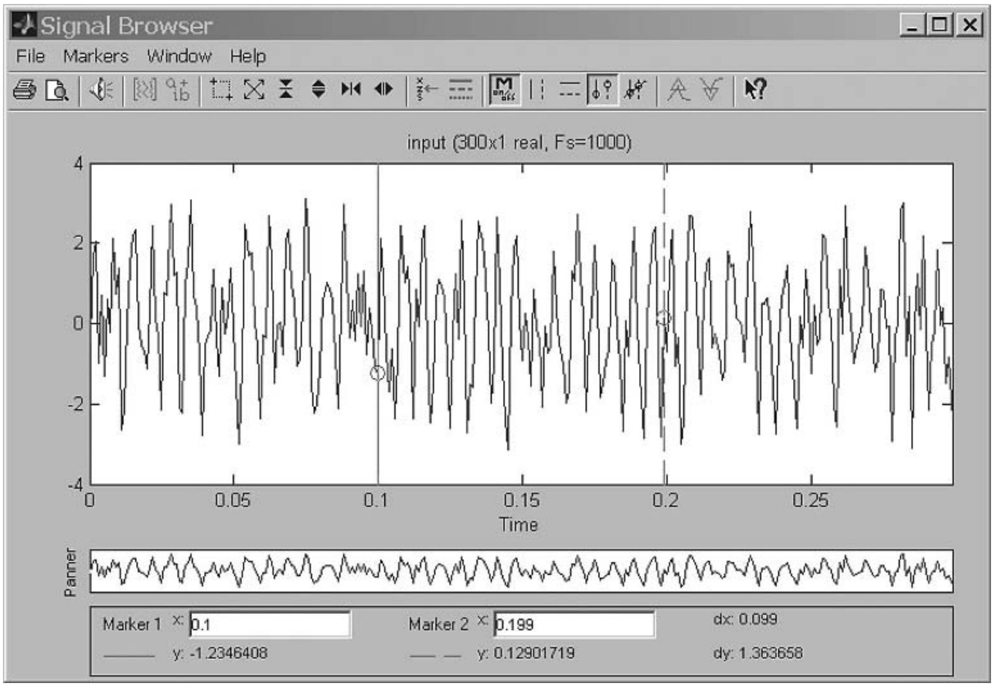
There are four windows that can be accessed within the SPTool:

1. The **Signal Browser** is used to view the input signals. Signals from the workspace or a file can be loaded into the SPTool by clicking **File → Import**. The **Import to SPTool** window allows users to select the data from either a file or a workspace. For example, after we execute `example5_19.m`, our workspace contains noisy sinewave in vector `xn` with sampling rate 1000 Hz. We import it by entering appropriate parameters in the dialog box. To view the signal, simply highlight the signal, and click **View**. The **Signal Browser** window is shown in Figure 5.23, which allows the user to zoom-in the signal, read the data values via markers, display format, and even play the selected signal using the computer's speakers.

2. The **Filter Designer** is used to design filters. Users can click the **New** icon to start a new filter, or the **Edit** icon to open an existing filter. We can design filters using different filter design algorithms. For example, we design an IIR filter displayed in Figure 5.24 that uses the same specifications as Example 5.19. In addition, we can also design a filter using the **Pole/Zero Editor** to graphically place the poles and zeros in the z-plane.

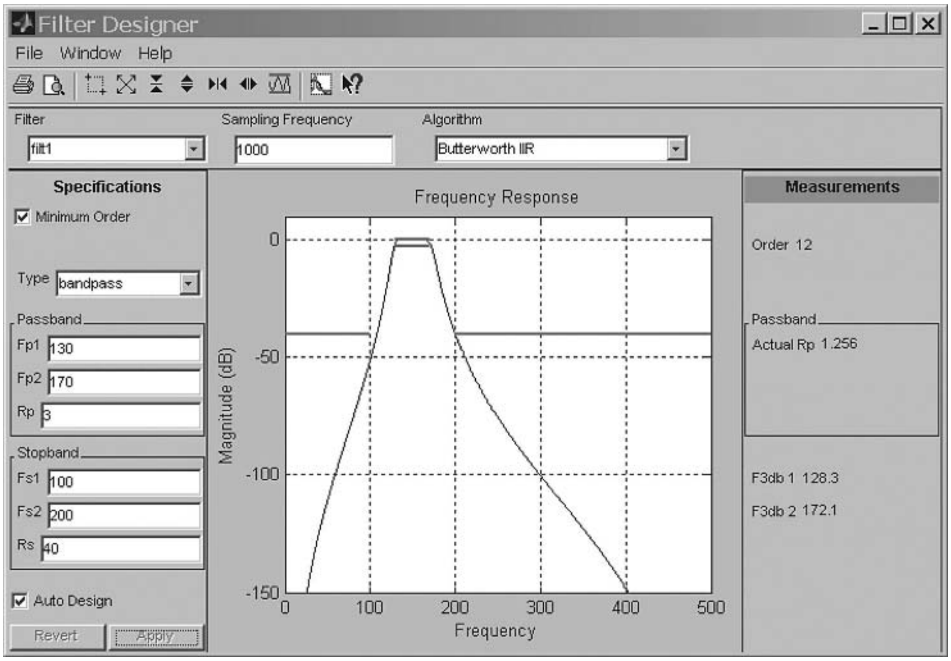
3. Once the filter has been designed, the frequency specification and other filter characteristics can be verified using the **Filter Viewer**. Selecting the name of the designed filter, and clicking the **View** icon under the **Filter** column will open the **Filter Viewer** window. We can analyze the filter in terms of its magnitude response, phase response, group delay, zero-pole plot, impulse response, and step response.

   After the filter characteristics have been verified, we can perform the filtering operation of the selected input signal. Click the **Apply** button, the **Apply Filter** window will be displayed, which allows the user to specify the file name of the output signal.



Marker 1 x: 0.1 y: -1.2346408
Marker 2 x: 0.199 y: 0.12901719
dx: 0.099
dy: 1.363658

Figure 5.23 Signal browser window



Filter: filt1
Sampling Frequency: 1000
Algorithm: Butterworth IIR

Specifications
Minimum Order ✓
Type: bandpass
Passband
Fp1: 130
Fp2: 170
Rp: 3
Stopband
Fs1: 100
Fs2: 200
Rs: 40
Auto Design ✓

Measurements
Order 12
Passband
Actual Rp 1.256
F3db 1 128.3
F3db 2 172.1

Figure 5.24 Design of bandpass filter

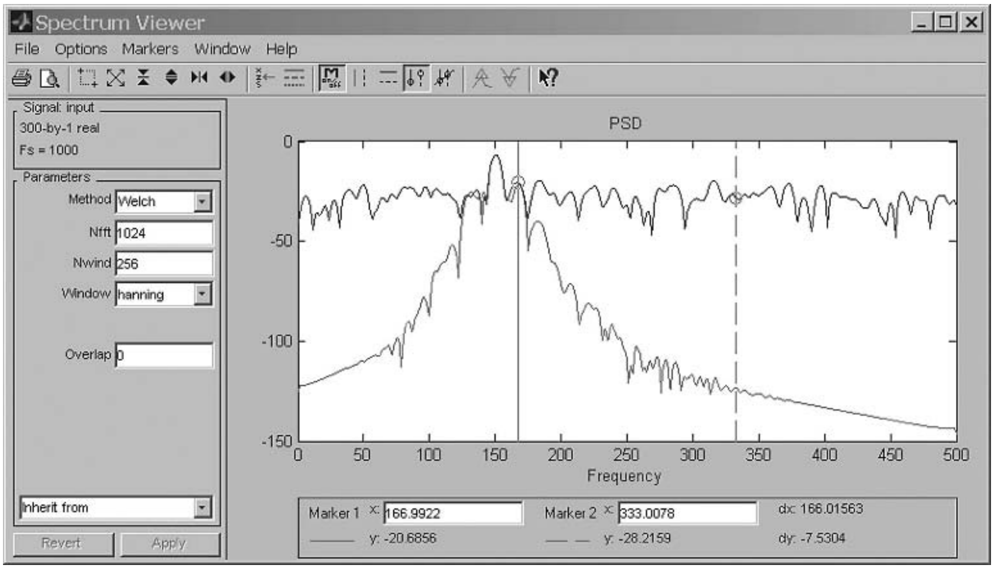

Signal: input
300-by-1 real
Fs = 1000
Parameters
Method: Welch
Nfft: 1024
Nwind: 256
Window: hanning
Overlap: 0
Marker 1 x: 166.9922 y: -20.6856
Marker 2 x: 333.0078 y: -28.2159
dx: 166.01563
dy: -7.5304

Figure 5.25 Spectrum viewer window for both input and output signals

4. We can compute the spectrum by selecting the signal, and then clicking the **Create** button in the **Spectra** column. Figure 5.25 is the display of the **Spectrum Viewer**. At the left-bottom corner of the window, click **Apply** to generate the spectrum of the selected signal. We repeat this process for both input and output signals. To view the spectra of input and output signals, select both `spect1` (spectrum of input) and `spect2` (spectrum of output), and click the **View** button in the **Spectra** column to display them (see Figure 5.25).

5.6 Practical Applications

In this section, we briefly introduce the application of IIR filtering for signal generation and audio equalization.

5.6.1 Recursive Resonators

Consider a simple second-order filter whose frequency response is dominated by a single peak at frequency ω_0. To make a peak at frequency $\omega = \omega_0$, we place a pair of complex-conjugated poles at

$$p_i = r_{\mathrm{p}} e^{\pm j\omega_0}, \tag{5.53}$$

where the radius $0 < r_{\mathrm{p}} < 1$. The transfer function of this IIR filter can be expressed as

$$\begin{aligned} H(z) &= \frac{A}{\left(1 - r_{\mathrm{p}} e^{j\omega_0} z^{-1}\right)\left(1 - r_{\mathrm{p}} e^{-j\omega_0} z^{-1}\right)} = \frac{A}{1 - 2r_{\mathrm{p}}\cos(\omega_0) z^{-1} + r_{\mathrm{p}}^2 z^{-2}} \\ &= \frac{A}{1 + a_1 z^{-1} + a_2 z^{-2}}, \end{aligned} \tag{5.54}$$

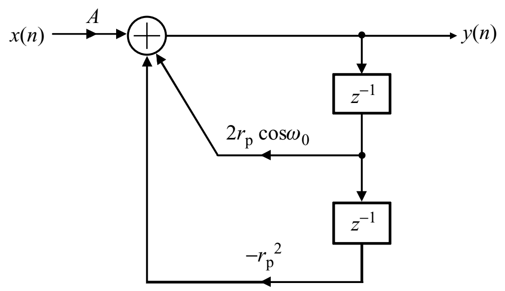


Figure 5.26 Signal-flow graph of second-order resonator filter

where A is a fixed gain used to normalize the filter to unity at ω_0 such that $|H(\omega_0)| = 1$. The direct-form realization is shown in Figure 5.26.

The magnitude response of this normalized filter is given by

$$|H(\omega_0)|_{z=e^{-j\omega_0}} = \frac{A}{|(1 - r_p e^{j\omega_0} e^{-j\omega_0})(1 - r_p e^{-j\omega_0} e^{-j\omega_0})|} = 1. \tag{5.55}$$

This condition can be used to obtain the gain

$$A = |(1 - r_p)(1 - r_p e^{-2j\omega_0})| = (1 - r_p)\sqrt{1 - 2r_p \cos(2\omega_0) + r_p^2}. \tag{5.56}$$

The 3-dB bandwidth of the filter is equivalent to

$$|H(\omega)|^2 = \frac{1}{2}|H(\omega_0)|^2 = \frac{1}{2}. \tag{5.57}$$

There are two solutions on both sides of ω_0, and the bandwidth is the difference between these two frequencies. When the poles are close to the unit circle, the BW is approximated as

$$BW \cong 2(1 - r_p). \tag{5.58}$$

This design criterion determines the value of r_p for a given BW. The closer r_p is to 1, the sharper the peak.

From Equation (5.54), the I/O equation of resonator is given by

$$y(n) = Ax(n) - a_1 y(n-1) - a_2 y(n-2), \tag{5.59}$$

where

$$a_1 = -2r_p \cos\omega_0 \quad \text{and} \quad a_2 = r_p^2. \tag{5.60}$$

This recursive oscillator is very useful for generating sinusoidal waveforms. This method uses a marginally stable two-pole resonator where the complex-conjugated poles lie on the unit circle ($r_p = 1$). This recursive oscillator is the most efficient way for generating a sinusoidal waveform, particularly if the quadrature signals (sine and cosine signals) are required.

The *Filter Design Toolbox* provides the function `iirpeak` for designing IIR peaking filter with the following syntax:

```
[NUM, DEN] = iirpeak(Wo, BW);
```

This function designs a second-order resonator with the peak at frequency `Wo` and a 3-dB bandwidth `BW`. In addition, we can use `[NUM,DEN] = iirpeak(Wo,BW,Ab)` to design a peaking filter with a bandwidth of `BW` at a level `Ab` in decibels.

Example 5.21: Design resonators operating at a sampling rate of 10 kHz having peaks at 1 and 2.5 kHz, and a 3-dB bandwidth of 500 and 200 Hz, respectively. These filters can be designed using the following MATLAB script (`example5_21.m`, adapted from the **Help** menu):

```
Fs = 10000;                 % Sampling rate
Wo = 1000/(Fs/2);           % First filter peak frequency
BW = 500/(Fs/2);            % First filter bandwidth
W1 = 2500/(Fs/2);           % Second filter peak frequency
BW1 = 200/(Fs/2);           % Second filter bandwidth
[b,a] = iirpeak(Wo,BW);     % Design first filter
[b1,a1] = iirpeak(W1,BW1); % Design second filter
fvtool(b,a,b1,a1);          % Analyze both filters
```

The magnitude responses of both filters are shown in Figure 5.27. In the FVTool window, we select Analysis→Pole/Zero Plot to display poles and zeros of both filters, which are shown in Figure 5.28. It is clearly shown that the second filter (peak at 2500 Hz) has a narrower bandwidth (200 Hz), and thus its poles are closer to the unit circle.

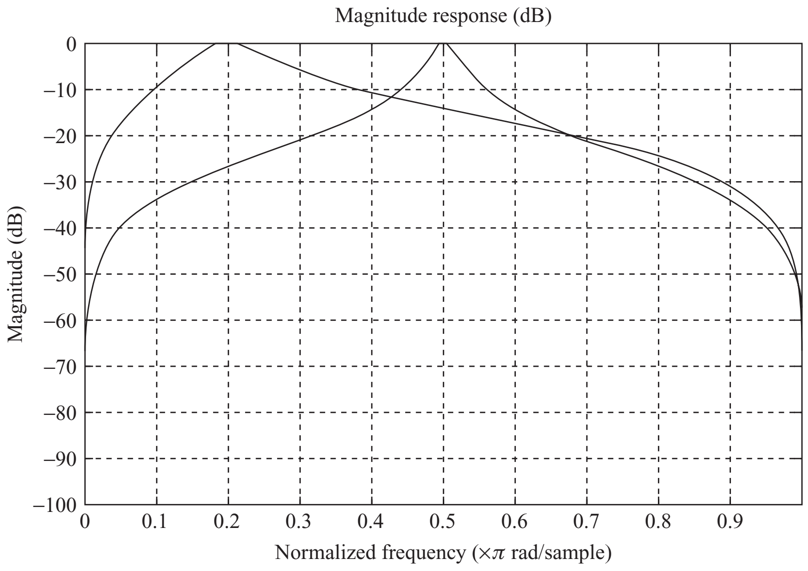


Figure 5.27 Magnitude responses of resonators

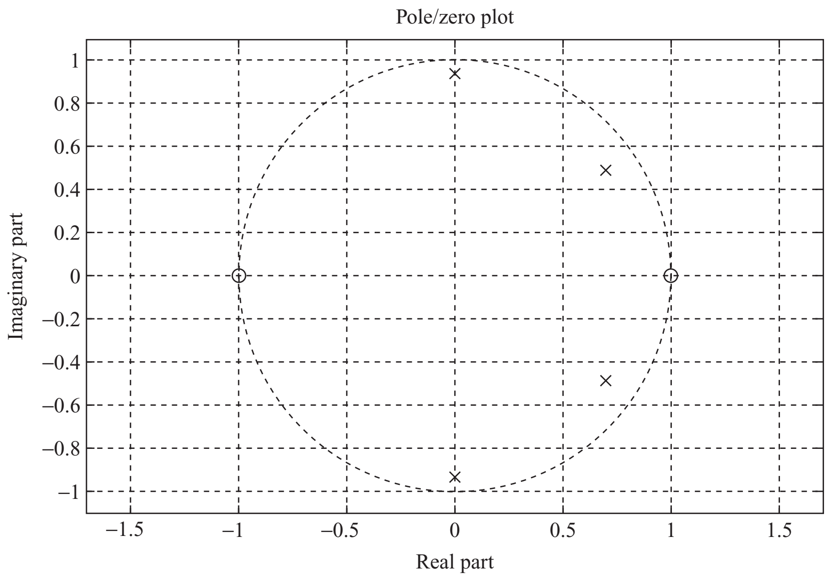


Figure 5.28 Pole/zero plot of resonators

5.6.2 Recursive Quadrature Oscillators

Consider two causal impulse responses

$$h_c(n) = \cos(\omega_0 n)\, u(n) \tag{5.61a}$$

and

$$h_s(n) = \sin(\omega_0 n)\, u(n), \tag{5.61b}$$

where $u(n)$ is the unit step function. The corresponding system transfer functions are

$$H_c(z) = \frac{1 - \cos(\omega_0)z^{-1}}{1 - 2\cos(\omega_0)z^{-1} + z^{-2}} \tag{5.62a}$$

and

$$H_s(z) = \frac{\sin(\omega_0)z^{-1}}{1 - 2\cos(\omega_0)z^{-1} + z^{-2}}. \tag{5.62b}$$

A two-output recursive structure with these system transfer functions is illustrated in Figure 5.29. The implementation requires just two data memory locations and two multiplications per sample. The output equations are

$$y_c(n) = w(n) - \cos(\omega_0)w(n-1) \tag{5.63a}$$

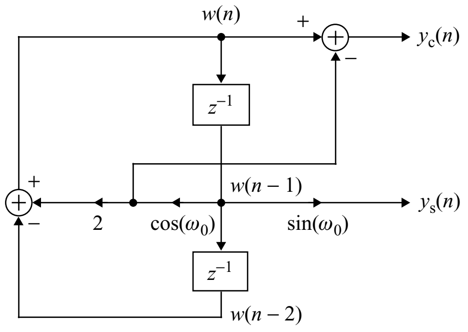


Figure 5.29 Recursive quadrature oscillators

and

$$y_s(n) = \sin(\omega_0)w(n-1), \tag{5.63b}$$

where $w(n)$ is an internal state variable that is updated as

$$w(n) = 2\cos(\omega_0)w(n-1) - w(n-2). \tag{5.64}$$

An impulse signal $A\delta(n)$ is applied to excite the oscillator, which is equivalent to presetting the following initial conditions:

$$w(-2) = -A \quad \text{and} \quad w(-1) = 0. \tag{5.65}$$

The waveform accuracy is limited primarily by the DSP processor wordlength. The quantization of the coefficient cos (ω_0) will cause the actual output frequency to differ slightly from the ideal frequency ω_0.

For some applications, only a sinewave is required. From Equations (5.59) and (5.60) using the conditions that $x(n) = A\delta(n)$ and $r_p = 1$, we can obtain the sinusoidal function

$$\begin{aligned} y_s(n) &= Ax(n) - a_1 y_s(n-1) - a_2 y_s(n-2) \\ &= 2\cos(\omega_0)y_s(n-1) - y_s(n-2) \end{aligned} \tag{5.66}$$

with the initial conditions

$$y_s(-2) = -A\sin(\omega_0) \quad \text{and} \quad y_s(-1) = 0. \tag{5.67}$$

The oscillating frequency defined by Equation (5.66) is determined from its coefficient a_1 and its sampling frequency f_s, and can be expressed as

$$f = \cos^{-1}\left(\frac{|a_1|}{2}\right)\frac{f_s}{2\pi}\ \text{Hz}, \tag{5.68}$$

where the coefficient $|a_1| \leq 2$.

Example 5.22: The sinewave generator using resonator can be realized from the recursive computation given in Equation (5.66). The implementation using the TMS320C55x assembly language is listed as follows:

```
     mov    cos_w,T1
     mpym   *AR1+,T1,AC0        ; AC0=cos(w)*y[n-1]
     sub    *AR1-<<#16,AC0,AC1 ; AC1=cos(w)*y[n-1]-y[n-2]
     add    AC0,AC1            ; AC1=2*cos(w)*y[n-1]-y[n-2]
||   delay *AR1                ; y[n-2]=y[n-1]
     mov    rnd(hi(AC1)),*AR1  ; y[n-1]=y[n]
     mov    rnd(hi(AC1)),*AR0+ ; y[n]=2*cos(w)*y[n-1]-y[n-2]
||   mpym   *AR1+,T1,AC0       ; AC0=cos(w)*y[n-1]
```

In the program, AR1 is the pointer for the signal buffer. The output sinewave samples are stored in the output buffer pointed by AR0. Due to the limited wordlength, the quantization error of fixed-point DSP processors such as the TMSC320C55x could be severe for the recursive computation.

5.6.3 Parametric Equalizers

A simple parametric equalizer filter can be designed from a resonator given in Equation (5.54) by adding a pair of zeros near the poles at the same angles as the poles; that is, placing the complex-conjugated poles at

$$z_i = r_z e^{\pm j\omega_0}, \tag{5.69}$$

where $0 < r_z < 1$. Thus, the transfer function given in Equation (5.54) becomes

$$\begin{aligned} H(z) &= \frac{\left(1 - r_z e^{j\omega_0} z^{-1}\right)\left(1 - r_z e^{-j\omega_0} z^{-1}\right)}{\left(1 - r_p e^{j\omega_0} z^{-1}\right)\left(1 - r_p e^{-j\omega_0} z^{-1}\right)} \\ &= \frac{1 - 2r_z \cos(\omega_0) z^{-1} + r_z^2 z^{-2}}{1 - 2r_p \cos(\omega_0) z^{-1} + r_p^2 z^{-2}} \\ &= \frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \end{aligned} \tag{5.70}$$

When $r_z < r_p$, the pole dominates over the zero because it is closer to the unit circle than the zero does. Thus, it generates a peak in the frequency response at $\omega = \omega_0$. When $r_z > r_p$, the zero dominates over the pole, thus providing a dip in the frequency response. When the pole and zero are very close to each other, the effects of the poles and zeros are reduced, resulting in a flat response. Therefore, Equation (5.70) provides a boost if $r_z < r_p$, or a cut if $r_z > r_p$. The amount of gain and attenuation is controlled by the difference between r_p and r_z. The distance from r_p to the unit circle will determine the bandwidth of the equalizer.

Example 5.23: Design a parametric equalizer with a peak at frequency 1500 Hz, and the sampling rate is 10 kHz. The parameters $r_z = 0.8$ and $r_p = 0.9$. The MATLAB script (`example5_23.m`) is listed as follows:

```
rz=0.8; rp=0.9;
b=[1, -2*rz*cos(w0), rz*rz];
a=[1, -2*rp*cos(w0), rp*rp];
```

Since $r_z < r_p$, this filter provides a boost.

Table 5.2 List of C function for implementing a floating-point, direct-form I IIR filter

```
void floatPoint_IIR(double in, double *x, double *y,
                    double *b, short nb, double *a, short na)
{
   double z1,z2;
   short i;

   for(i=nb-1; i>0; i--)          // Update the buffer x[]
      x[i] = x[i-1];
   x[0] = in;                     // Insert new data to x[0]
   for(z1=0, i=0; i<nb; i++)      // Filter x[] with coefficients in b[]
     z1 += x[i] * b[i];
   for(i=na-1; i>0; i--)          // Update y buffer
     y[i] = y[i-1];
   for(z2=0, i=1; i<na; i++)      // Filter y[] with coefficients in a[]
      z2 += y[i] * a[i];
   y[0] = z1 - z2;                // Place the result into y[0]
}
```

5.7 Experiments and Program Examples

This section will demonstrate the IIR filter design and implementation in MATLAB, C, and the TMS320C55x assembly programs using the simulator and DSK.

5.7.1 Floating-Point Direct-Form I IIR Filter

The direct-form I realization of IIR filter given by Equation (5.31) can be implemented by the C function listed in Table 5.2. The input and output signal buffers are `x` and `y`, respectively. The current input data is passed to the function via the variable `in`, and the filter output is saved on the top of `y` buffer as `y[0]`. The IIR filter coefficients are stored in the arrays `a` and `b` with lengths `na` and `nb`, respectively. This IIR filter function uses a sample-by-sample processing with 8 kHz sampling rate. The input signal contains three sinusoids with frequencies 800, 1800, and 3300 Hz. The IIR bandpass filter is designed with center frequency at 1800 Hz, passband bandwidth of 836 Hz, and stopband attenuation of 60 dB, thus the filter attenuates the 800 and 3300 Hz sinusoidal components. Table 5.3 lists the files used for this experiment.

Table 5.3 File listing for experiment `exp5.7.1_floatPoint_directIIR`

Files	Description
`floatPoint_directIIRTest.c`	C function for testing floating-point IIR filter
`floatPoint_directIIR.c`	C function for floating-point IIR filter
`floatPointIIR.h`	C header file for IIR experiment
`floatPoint_direcIIR.pjt`	DSP project file
`floatPoint_direcIIR.cmd`	DSP linker command file
`input.pcm`	Data file

Procedures of the experiment are listed as follows:

1. Open the project `floatPoint_directIIR.pjt`, and rebuild the floating-point IIR filter project.
2. Run the project to filter the input data located in the data directory.
3. Validate the output signal to ensure that the 800 and 3300 Hz frequency components are reduced by the 60 dB.

5.7.2 Fixed-Point Direct-Form I IIR Filter

The fixed-point implementation can be obtained by modifying the floating-point IIR filter from the previous experiment. Data type `long` must be used for integer multiplication. For fractional integer implementation, the product of the multiplication resides in the upper portion of the `long` variables. The fixed-point IIR filter function is listed in Table 5.4. Table 5.5 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Open the project file `fixedPoint_direcIIR.pjt` and rebuild the project.
2. Run the project using the input signal in the data directory.

Table 5.4 Fixed-point implementation of direct-form I IIR filter

```
void fixPoint_IIR(short in, short *x, short *y,
                  short *b, short nb, short *a, short na)
{
   long  z1,z2,temp;
   short i;

   for(i=nb-1; i>0; i--)          // Update the buffer x[]
     x[i] = x[i-1];
   x[0] = in;                     // Insert new data to x[0]
   for(z1=0, i=0; i<nb; i++)      // Filter x[] with coefficients in b[]
   {
     temp = (long)x[i] * b[i];
     temp += 0x400;
     z1 += (short)(temp>>11);
   }
   for(i=na-1; i>0; i--)          // Update y[] buffer
     y[i] = y[i-1];
   for(z2=0, i=1; i<na; i++)      // Filter y[] with coefficients in a[]
   {
     temp = (long)y[i] * a[i];
     temp += 0x400;
     z2 += (short)(temp>>11);
   }
   y[0] = (short)(z1 - z2);       // Place the result into y[0]
}
```

Table 5.5 File listing for experiment `exp5.7.2_fixedPoint_directIIR`

Files	Description
`fixPoint_directIIRTest.c`	C function for testing fixed-point IIR filter experiment
`fixPoint_directIIR.c`	C function for fixed-point IIR filter
`fixPointIIR.h`	C header file for IIR experiment
`fixedPoint_direcIIR.pjt`	DSP project file
`floatPoint_direcIIR.cmd`	DSP linker command file
`input.pcm`	Data file

3. Validate the output signal to ensure that the 800 and 3300 Hz sinusoidal components are reduced by 60 dB.

4. Compare the output signal with previous floating-point output signal to check the performance difference.

5. Profile the fixed-point IIR filter performance.

5.7.3 Fixed-Point Direct-Form II Cascade IIR Filter

The cascade structure shown in Figure 5.8 can be expressed as

$$\begin{aligned} w(n) &= x(n) - a_1 w(n-1) - a_2 w(n-2). \\ y(n) &= b_0 w(n) + b_1 w(n-1) + b_2 w(n-2) \end{aligned} \tag{5.71}$$

The C implementation of cascading K second-order sections is given as follows:

```
temp = input[n];
for (k=0; k<K; k++)
{
  w[k][0] = temp-a[k][1]*w[k][1]-a[k][2]*w[k][2];
  temp = b[k][0]*w[k][0]+b[k][1]*w[k][1]+b[k][2]*w[k][2];
  w[k][2] = w[k][1];       /* w(n-2) <- w(n-1) */
  w[k][1] = w[k][0];       /* w(n-1) <- w(n)   */
}
output[n] = temp;
```

In the code, `a[ ][ ]` and `b[ ][ ]` are filter coefficient matrices, and `w[ ][ ]` is the signal buffer for $w_k(n-m)$, $m = 0, 1, 2$. The row index `k` represents the kth second-order IIR filter section, and the column index points at the filter coefficients or signal samples in the buffers.

As mentioned earlier, the zero-overhead repeat loops, multiply–accumulate instructions, and circular addressing modes are three important features of modern DSP processors. To better understand these features, we write the function of second-order IIR filter with cascade structure in fixed-point C using the data pointers to simulate the circular addressing modes. We also arrange the C statements to mimic the DSP multiply–accumulate operations. The fixed-point C program listed in Table 5.6 implements an IIR filter with `Ns` second-order sections in cascade form.

Table 5.6 Fixed-point implementation of direct-form II IIR filter

```
void cascadeIIR(short *x, short Nx, short *y, short *coef, short Ns,
                short *w)
{
    short i,j,n,m,k,l;
    short temp16;
    long  w_0,temp32;

    m=Ns*5;                             // Setup circular buffer coef[]
    k=Ns*2;                             // Setup circular buffer w[]

    for (j=0,l=0,n=0; n<Nx; n++)        // IIR filtering
    {
        w_0 = (long)x[n]<<12;           // Scale input to prevent overflow
        for (i=0; i<Ns; i++)
        {
            temp32 = (long)(*(w+l)) * *(coef+j); j++; l=(l+Ns)%k;
            w_0 -= temp32<<1;
            temp32 = (long)(*(w+l)) * *(coef+j); j++;
            w_0 -= temp32<<1;
            w_0 += 0x4000;              // Rounding
            temp16 = *(w+l);
            *(w+l) = (short)(w_0>>15); // Save in Q15 format
            w_0 = (long)temp16 * *(coef+j); j++;
            w_0  <<= 1;
            temp32 = (long)*(w+l) * *(coef+j); j++; l=(l+Ns)%k;
            w_0 += temp32<<1;
            temp32 = (long)*(w+l) * *(coef+j); j=(j+1)%m; l=(l+1)%k;
            w_0 += temp32<<1;
            w_0   += 0x800;             // Rounding
      }
      y[n] = (short)(w_0>>12);          // Output in Q15 format
    }
}
```

The coefficient and signal buffers are configured as circular buffers shown in Figure 5.30. The signal buffer contains two elements, $w_k(n-1)$ and $w_k(n-2)$, for each second-order section. The pointer address is initialized pointing at the first sample $w_1(n-1)$ in the buffer. The coefficient vector is arranged with five coefficients (a_{1k}, a_{2k}, b_{2k}, b_{0k}, and b_{1k}) per section with the coefficient pointer initialized to point at the first coefficient, a_{11}. The circular pointers are updated by `j=(j+1)%m` and `l=(l+1)%k`, where `m` and `k` are the sizes of the coefficient and signal buffers, respectively.

The test function reads in the filter coefficients header file, `fdacoefsMATLAB.h`, generated by the FDATool directly. Table 5.7 lists the files used for this experiment, where the test input data file `in.pcm` consists of three frequencies, 800, 1500, and 3300 Hz with the 8 kHz sampling rate.

Procedures of the experiment are listed as follows:

1. Open the project file `fixedPoint_cascadeIIR.pjt` and rebuild the project.

2. Run the cascade filter experiment to filter the input signal in the data directory.

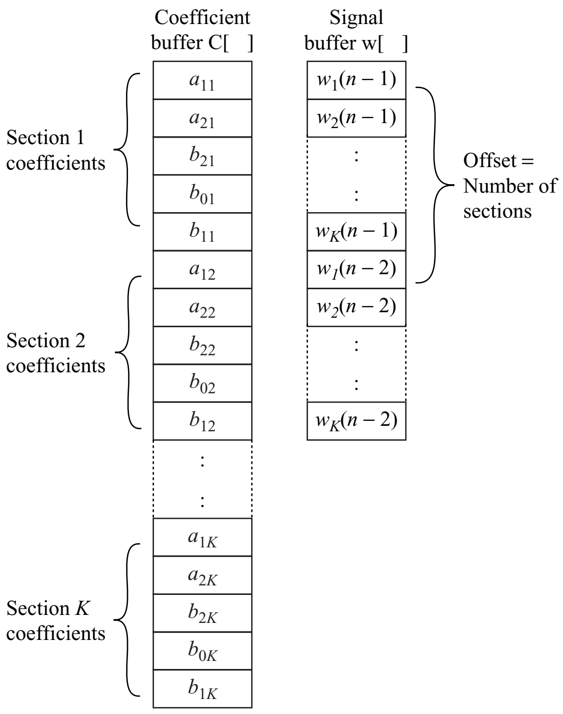


Figure 5.30 IIR filter coefficient and signal buffers configuration

3. Validate the output data to ensure that the 800 and 3300 Hz sinusoidal components are reduced by the 60 dB.

4. Profile the fixed-point direct-form II cascade IIR filter performance.

5.7.4 Implementation Using DSP Intrinsics

The C55x C intrinsics can be used as any C function and they produce assembly language statements directly in compile time. The intrinsics are specified with a leading underscore and can be accessed

Table 5.7 File listing for experiment `exp5.7.3_fixedPoint_cascadeIIR`

Files	Description
`fixPoint_cascadeIIRTest.c`	C function for testing cascade IIR filter experiment
`fixPoint_cascadetIIR.c`	C function for fixed-point second-order IIR filter
`cascadeIIR.h`	C header file for cascade IIR experiment
`fdacoefsMATLAB.h`	FDATool generated C header file
`tmwtypes.h`	Data type definition file for MATLAB C header file
`fixedPoint_cascadeIIR.pjt`	DSP project file
`fixedPoint_cascadeIIR.cmd`	DSP linker command file
`in.pcm`	Data file

by calling them as C functions. For example, the multiply–accumulation operation, `z+=x*y`, can be implemented by the following intrinsic:

```
short x,y;
long z;
z = _smac(z,x,y);    // Perform signed z=z+x*y
```

This intrinsic performs the following assembly instruction:

```
macm Xmem,Ymem,Acx;   Perform signed z=z+x*y
```

Table 5.8 lists the intrinsics supported by the TMS320C55x C compiler.

We will modify the previous fixed-point C function for cascade IIR filter using C intrinsics. Table 5.9 lists the implementation of the fixed-point IIR filter with coefficients in Q14 format. For the modulo operation, we replaced the sections, `k`, with an `and`(&) operation since the number `k` is a power-of-2 number.

The test function reads in the filter coefficients from the C header file `fdacoefsMATLAB.h` generated by the FDATool. Table 5.10 lists the files used for this experiment, where the input data file `in.pcm` consists of three frequencies, 800, 1500, and 3300 Hz with the 8 kHz sampling rate.

Procedures of the experiment are listed as follows:

1. Open the project file `intrisics_implementation.pjt` and rebuild the project.

2. Run the experiment to filter the test signal in the data directory.

3. Validate the output signal to ensure that the 800 and 3300 Hz sinusoidal components are attenuated by 60 dB.

4. Profile the code and compare the result with the performance obtained in previous experiment.

5.7.5 Implementation Using Assembly Language

The fixed-point C implementation of an IIR filter can be more efficient using the C55x multiply–accumulator instruction with circular buffers. The C55x assembly implementation of the second-order, direct-form II IIR filter given in Equation (5.71) can be written as

```
mov *AR0+<<#12,AC0       ; AC0 = x(n) with scale down
masm *AR3+,*AR7+,AC0     ; AC0=AC0-a1*wi(n-1)
masm T3=*AR3,*AR7+,AC0   ; AC0=AC0-a2*wi(n-2)
mov hi(AC0),*AR3-        ; wi(n-2)=wi(n)
mpym *AR7+,T3,AC0        ; AC0=b2*wi(n-2)
macm *AR3+,*AR7+,AC0     ; AC0=AC0+bi0*wi(n-1)
macm *AR3,*AR7+,AC0      ; AC0=AC0+bi1*wi(n)
mov hi(AC0),*AR1+        ; Store filter result
```

The assembly program contains three data pointers and a coefficient pointer. The auxiliary register `AR0` is the input buffer pointer pointing to the input sample. The filtered sample is rounded and stored in the output buffer pointed by `AR1`. The signal buffer $w_i(n)$ is pointed by `AR3`. The filter coefficients pointer `AR7` and signal pointer `AR3` can be efficiently implemented using circular addressing mode.

Table 5.8 Intrinsics supported by the TMS320C55x C compiler

C compiler intrinsics	Description
`short _sadd(short src1, short src2);`	Adds two 16-bit integers with SATA set, producing a saturated 16-bit result
`long _lsadd(long src1, long src2);`	Adds two 32-bit integers with SATD set, producing a saturated 32-bit result
`short _ssub(short src1, short src2);`	Subtracts `src2` from `src1` with SATA set, producing a saturated 16-bit result
`long _lssub(long src1, long src2);`	Subtracts `src2` from `src1` with SATD set, producing a saturated 32-bit result
`short _smpy(short src1, short src2);`	Multiplies `src1` and `src2` and shifts the result left by 1. Produces a saturated 16-bit result. (SATD and FRCT are set.)
`long _lsmpy(short src1, short src2);`	Multiplies `src1` and `src2` and shifts the result left by 1. Produces a saturated 32-bit result. (SATD and FRCT are set.)
`long _smac(long src, short op1, short op2);`	Multiplies `op1` and `op2`, shifts the result left by 1, and adds it to `src`. Produces a saturated 32-bit result. (SATD, SMUL, and FRCT are set.)
`long _smas(long src, short op1, short op2);`	Multiplies `op1` and `op2`, shifts the result left by 1, and subtracts it from `src`. Produces a 32-bit result. (SATD, SMUL, and FRCT are set.)
`short _abss(short src);`	Creates a saturated 16-bit absolute value. `_abss`(0x8000) => 0x7FFF (SATA set)
`long _labss(long src);`	Creates a saturated 32-bit absolute value. `_labss`(0x8000000) => 0x7FFFFFFF (SATD set)
`short _sneg(short src);`	Negates the 16-bit value with saturation `_sneg`(0xffff8000) => 0x00007FFF
`long _lsneg(long src);`	Negates the 32-bit value with saturation. `_lsneg`(0x80000000) => 0x7FFFFFFF
`short _smpyr(short src1, short src2);`	Multiplies `src1` and `src2`, shifts the result left by 1, and rounds by adding 2^{15} to the result. (SATD and FRCT are set.)
`short _smacr(long src, short op1, short op2);`	Multiplies `op1` and `op2`, shifts the result left by 1, adds the result to `src`, and then rounds the result by adding 2^{15}. (SATD, SMUL, and FRCT are set)
`short _smasr(long src, short op1, short op2);`	Multiplies `op1` and `op2`, shifts the result left by 1, subtracts the result from `src`, and then rounds the result by adding 2^{15}. (SATD, SMUL, and FRCT set.)
`short _norm(short src);`	Produces the number of left shifts needed to normalize `src.`
`short _lnorm(long src);`	Produces the number of left shifts needed to normalize `src.`
`short _rnd(long src);`	Rounds `src` by adding 2^{15}. Produces a 16-bit saturated result. (SATD set)

continues overleaf

Table 5.8 (*continued*)

C compiler intrinsics	Description
`short _sshl(short src1, short src2);`	Shifts `src1` left by `src2` and produces a 16-bit result. The result is saturated if `src2` is less than or equal to 8. (SATD set)
`long _lsshl(long src1, short src2);`	Shifts `src1` left by `src2` and produces a 32-bit result. The result is saturated if `src2` is less than or equal to 8. (SATD set)
`short _shrs(short src1, short src2);`	Shifts `src1` right by `src2` and produces a 16-bit result. Produces a saturated 16-bit result. (SATD set)
`long _lshrs(long src1, short src2);`	Shifts `src1` right by `src2` and produces a 32-bit result. Produces a saturated 32-bit result. (SATD set)
`short _addc(short src1, short src2);`	Adds `src1`, `src2`, and carry bit and produces a 16-bit result.
`long _laddc(long src1, short src2);`	Adds `src1`, `src2`, and carry bit and produces a 32-bit result.

This IIR filtering code can be easily modified for performing either a sample-by-sample or block processing. When the IIR filter function is called, the temporary register `T0` contains the number of input samples to be filtered, and `T1` contains the number of second-order sections. The IIR filter sections are implemented by the inner loop, and the outer loop is used for processing samples in blocks. The

Table 5.9 Fixed-point implementation of direct-form II IIR filter using intrinsics

```
void intrinsics_IIR(short *x, short Nx, short *y,
                    short *coef, short Ns, short *w)
{
    short i,j,n,m,k,l;
    short temp16;
    long  w_0;

    m=Ns*5;                            // Setup circular buffer coef[]
    k=Ns*2-1;                          // Setup circular buffer w[]

    for (j=0,l=0,n=0; n<Nx; n++)       // IIR filtering
    {
        w_0 = (long)x[n]<<12;          // Scale input to prevent overflow
        for (i=0; i<Ns; i++)
        {
           w_0 = _smas(w_0,*(w+l),*(coef+j)); j++; l=(l+Ns)&k;
           w_0 = _smas(w_0,*(w+l),*(coef+j)); j++;
           temp16 = *(w+l);
           *(w+l) = (short)(w_0>>15); // Save in Q15 format
           w_0 = _lsmpy(temp16,*(coef+j)); j++;
           w_0 = _smac(w_0,*(w+l),*(coef+j)); j++; l=(l+Ns)&k;
           w_0 = _smac(w_0,*(w+l),*(coef+j)); j=(j+1)%m; l=(l+1)&k;
        }
        y[n] = (short)(w_0>>12);       // Output in Q15 format
    }
}
```

Table 5.10 File listing for experiment `exp5.7.4_intrisics_implementation`

Files	Description
`intrinsics_IIRTest.c`	C function for testing IIR filter intrinsics experiment
`intrinsics_IIR.c`	Intrinsics implementation of second-order IIR filter
`intrinsics_IIR.h`	C header file for intrinsics IIR experiment
`fdacoefsMATLAB.h`	FDATool generated C header file
`tmwtypes.h`	Data type definition file for MATLAB C header file
`intrisics_implementation.pjt`	DSP project file
`intrisics_implementation.cmd`	DSP linker command file
`in.pcm`	Data file

IIR filter coefficients are represented using Q14 format. To prevent the overflow, the input sample is scaled down as well. To compensate the Q14 formatted coefficients and scaled down input samples, the filter result $y(n)$ is scaled up to form the Q15 format and stored with rounding. Temporary register `T3` is used to hold the second element $w_i(n-2)$ of the signal buffer when the buffer update is taking place.

For a K-section IIR filter, the signal buffer elements are arranged in such a way that two elements of each section are separated by $K-1$ elements. The filter coefficients and the signal samples are arranged for circular buffer as shown in Figure 5.30. The complete assembly language implementation of the IIR filter in cascade second-order sections is listed in Table 5.11.

Table 5.12 lists the files used for this experiment. The test function reads in the filter coefficients generated by the FDATool, which are saved in C header file `fdacoefsMATLAB.h`.

Procedures of the experiment are listed as follows:

1. Open the project file `asm_implementation.pjt` and rebuild the project.
2. Run the experiment to filter the input signal in data directory.
3. Validate the output signal to ensure that the 800 and 3300 Hz sinusoidal components are attenuated by 60 dB.
4. Profile this experiment and compare the result with previous experiments.

5.7.6 Real-Time Experiments Using DSP/BIOS

In Chapter 4, we have used DSP/BIOS for real-time FIR filtering. In this experiment, we will apply the same process to create a new DSP/BIOS project for IIR filtering. The DSP/BIOS provides additional tools for code development and debug. One of the useful tools is the CPU load graph. The CPU loading can be plotted in real time to monitor the real-time performance of DSP system. This feature is especially useful for multithread system when multiple threads sharing the CPU concurrently. Another useful graphical tool is the DSP execution graph. The DSP execution graph shows several DSP/BIOS tasks, including hardware interrupts (HWI), software interrupts (SWI), tasks (TSK), and semaphores (SEM). The IIR filtering experiment execution graph is shown in Figure 5.31. Our experiment has one task – `swiAudioProcess`. This task is software interrupt based and has the highest priority.

Table 5.11 Assembly language implementation of direct-form II IIR filter

```
      .global _asmIIR
      .sect   ".text:iir_code"

_asmIIR
      pshm  ST1_55                  ; Save ST1, ST2, ST3
      pshm  ST2_55
      pshm  ST3_55
      psh   T3                      ; Save T3
      pshboth XAR7                  ; Save AR7
      or    #0x340,mmap(ST1_55)     ; Set FRCT, SXMD, SATD
      bset  SMUL                    ; Set SMUL
      sub   #1,T0                   ; Number of samples - 1
      mov   T0,BRC0                 ; Set up outer loop counter
      sub   #1,T1,T0                ; Number of sections -1
      mov   T0,BRC1                 ; Set up inner loop counter
      mov   T1,T0                   ; Set up circular buffer sizes
      sfts  T0,#1
      mov   mmap(T0),BK03           ; BK03=2*number of sections
      sfts  T0,#1
      add   T1,T0
      mov   mmap(T0),BK47           ; BK47=5*number of sections
      mov   mmap(AR3),BSA23         ; Initial signal buffer base
      mov   mmap(AR2),BSA67         ; Initial coefficient base
      amov  #0,AR3                  ; Initial signal buffer entry
      amov  #0,AR7                  ; Initial coefficient entry
      or    #0x88,mmap(ST2_55)
      mov   #1,T0                   ; Used for shift left
||    rptblocal sample_loop-1       ; Start IIR filtering loop
      mov   *AR0+ <<#12,AC0         ; AC0 = x(n)/8 (i.e. Q12)
||    rptblocal filter_loop-1       ; Loop for each section
      masm  *(AR3+T1),*AR7+,AC0     ; AC0-=ai1*wi(n-1)
      masm  T3=*AR3,*AR7+,AC0       ; AC0-=ai2*di(n-2)
      mov   rnd(hi(AC0<<T0)),*AR3   ; wi(n-2)=wi(n)
||    mpym  *AR7+,T3,AC0            ; AC0+=bi2*wi(n-2)
      macm  *(AR3+T1),*AR7+,AC0     ; AC0+=bi0*wi(n-1)
      macm  *AR3+,*AR7+,AC0         ; AC0+=bi1*wi(n)
filter_loop
      mov   rnd(hi(AC0<<#4)),*AR1+ ; Store result in Q15 format
sample_loop
      popboth XAR7                  ; Restore AR7
      pop   T3                      ; Restore T3
      popm  ST3_55                  ; Restore ST1, ST2, ST3
      popm  ST2_55
      popm  ST1_55
      ret
.end
```

The DSP/BIOS example code uses buffered pipe manager (PIP) for data exchange between threads. The PIP operation is frame based. The important parameters for PIP are:

`bufseg` – memory segment for the PIP data;

`bufalign` – data alignment in memory;

Table 5.12 File listing for experiment `exp5.7.5_asm_implementation`

Files	Description
`asmIIRTest.c`	C function for testing IIR filter assembly experiment
`asmIIR.asm`	Assembly implementation of second-order IIR filter
`asmIIR.h`	Header file for assembly IIR experiment
`fdacoefsMATLAB.h`	FDATool generated C header file
`tmwtypes.h`	Data type definition file for MATLAB C header file
`asm_implementation.pjt`	DSP project file
`asm_implementation.cmd`	DSP linker command file
`in.pcm`	Data file

`numframes` – number of frames;

`framesize` – Length of the frame.

`monitor` – Monitoring the PIP reader and writer for status module.

Each data transfer should have its own pipe. Each pipe should have its own reader for receiving data and writer for transmit data. Usually, one end of the pipe will be connected and controlled by HWI, while the other end is connected to processor and controlled by SWI. We use the following code to show an example of using PIP for data exchange:

```
// Get the full rx buffer from the receive PIP
PIP_get(&pipRx);
src = PIP_getReaderAddr(&pipRx);
size = PIP_getReaderSize(&pipRx) * sizeof(short);

// Get the empty tx buffer from the transmit PIP
PIP_alloc(&pipTx);
dst = PIP_getWriterAddr(&pipTx);

// Record the amount of actual data being sent
PIP_setWriterSize(&pipTx, PIP_getReaderSize(&pipRx));

// Free the receive buffer, put the transmit buffer
PIP_free(&pipRx);
PIP_put(&pipTx);
```

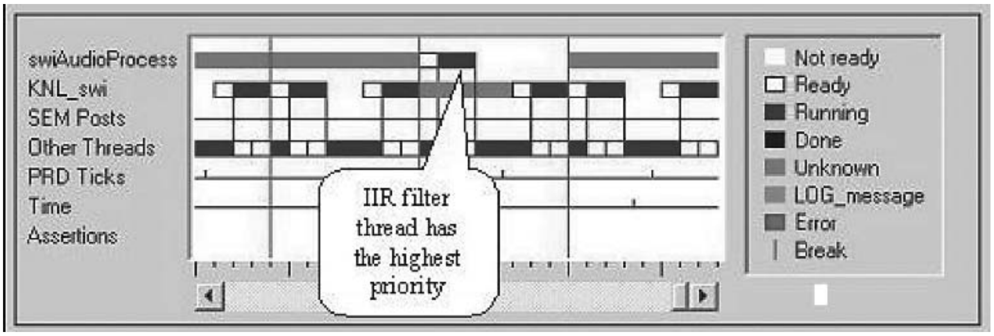


Figure 5.31 DSP/BIOS execution graph of the IIR filter experiment

Table 5.13 File listing for experiment `exp5.7.6_realtime_DSPBIOS`

Files	Description
`realtime_DSPBIOS.c`	C function for testing DSP/BIOS IIR filter experiment
`asmIIR.asm`	Assembly implementation of second-order IIR filter
`plio.c`	PIP function in C
`asmIIR.h`	Header file for assembly IIR experiment
`fdacoefsMATLAB.h`	FDATool generated C header file
`tmwtypes.h`	Data type definition file for MATLAB C header file
`lio.h`	Low level I/O header file
`plio.h`	PIP to low level I/O interface header file
`dspbios.cdb`	DSP/BIOS configuration file
`dSPBIOS.pjt`	DSP/BIOS project file
`dspbioscfg.cmd`	DSP/BIOS linker command file
`in.pcm`	Data file
`in.wav`	Data file

To read data, we first call `PIP_get( )` to get a frame of received data, and then call `PIP_getReaderAddr( )` and `PIP_getReaderSize( )` to set the data frame buffer address and the length of the frame. Before writing data to the PIP, we call `PIP_alloc( )` to allocate an empty frame memory. The data pointer is obtained by the function `PIP_getWriterAddr( )`. The number of the data to be sent is passed using the function `PIP_setWriterSize( )`. Finally, we free the receive frame memory and notify the writer that the transmit frame data is ready to be sent.

The DSP/BIOS project with assembly implementation of the direct-form II cascade IIR filter is included in the companion CD. We encourage readers to create their own DSP/BIOS project and configure the DSP/BIOS project properly. Table 5.13 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Create the DSP/BIOS project for using DSK with PIP module.

2. Configure the AIC23 of the DSK for real-time I/O at 8 kHz sampling rate for 16-bit audio data.

3. Add IIR filter program and build the project. Run the project to filter the input signal in real time.

4. Validate the output signal using a scope.

5.7.7 Implementation of Parametric Equalizer

This experiment follows Equation (5.54) to design an equalizer that has the resonators at 200 and 1000 Hz when the sampling rate is 8 kHz. The equalizer has a dynamic range of $\pm$ 6 dB with 1 dB step. A direct-form II IIR filter is used as the resonator. The equalizer coefficients are generated by the function `coefGen( )` listed in Table 5.14. This function uses the parameter `gain` to select $\texttt{r}_\texttt{z}$ and $\texttt{r}_\texttt{p}$, and then computes the coefficients.

The experiment is implemented in fixed-point C to show the logic flow. Table 5.15 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Open the project file `parametric_equalizer.pjt` and rebuild the project.

2. Run the equalizer project using the input signal file, `input.pcm`.

Table 5.14 C function for generating parametric equalizer coefficients

```
void coefGen(double (*gainTbl)[2], short gain, short *c, float freq)
{
  double rz,rp,temp,omega;

  rz = gainTbl[gain][0];              // Get rz from the lookup table
  rp = gainTbl[gain][1];              // Get rp from the lookup table

  omega = 2.0*3.1415926*freq/8000.0;  // 8kHz sampling rate
  c[3] = 0x4000;                      // b[0] in Q14 format
  temp = -2.0*rz*cos(omega);
  c[4] = (short)(temp*16384.0+0.5);   // b[1]
  c[2] = (short)(rz*rz*16384.0+0.5);  // b[2]
  temp = -2.0*rp*cos(omega);
  c[0] = (short)(temp*16384.0+0.5);   // a[1]
  c[1] = (short)(rp*rp*16384.0+0.5);  // a[2]
}
```

3. Examine the output signal file and measure the signal level at frequencies 200 and 1000 Hz.

4. Vary the equalizer gains and repeat the experiment to see the response of the output signal under the different equalizer settings.

5. Replace the test data file `input.pcm` with the white noise file `wn.pcm`, and repeat the experiment.

5.7.8 Real-Time Two-Band Equalizer Using DSP/BIOS

The DSP/BIOS project using assembly program to implement a two-band parametric equalizer. Table 5.16 lists the files used for this experiment. We encourage readers to create their own DSP/BIOS project and configure the DSP/BIOS project properly.

Procedures of the experiment are listed as follows:

1. Create the DSP/BIOS project for using DSK with PIP module.

2. Configure the AIC23 of the DSK for real-time I/O at 8 kHz sampling rate for 16-bit audio data.

Table 5.15 File listing for experiment `exp5.7.7_parametric_equalizer`

Files	Description
`parametric_equalizerTest.c`	C function for testing equalizer experiment
`fixPoint_cascadetIIR.c`	C function of parametric equalizer
`cascadeIIR.h`	Header file for the experiment
`parametric_equalizer.pjt`	Experiment project file
`parametric_equalizer.cmd`	DSP linker command file
`input.pcm`	Data file consists of two tones
`wn.wav`	Data file consists of white noise

Table 5.16 File listing for experiment `exp5.7.8_realtime_2Band_EQ`

Files	Description
`rt_2band_eq.c`	C function for testing DSP/BIOS EQ experiment
`asmIIR.asm`	Assembly implementation of second-order IIR filter
`plio.c`	PIP function in C
`asmIIR.h`	Header file for assembly IIR experiment
`lio.h`	Low level I/O header file
`plio.h`	PIP to low level I/O interface header file
`DSPBIOS_2band_eq.cdb`	DSP/BIOS configuration file
`DSPBIOS_2band_eq.pjt`	DSP/BIOS project file
`DSPBIOS_2band_eqcfg.cmd`	DSP/BIOS linker command file

3. Test the project in audio loopback only mode.

4. Refer to Experiment 5.7.6 and add function `coefGen( )` that generates equalizer coefficients.

5. Insert the IIR filter program to the loopback project.

6. Build and run the project using real-time audio signal such as from a CD player.

Table 5.17 Equalizer coefficients switching in real time

```
void switchBand()
{
    short i;

    // Initialize IIR filter signal buffer
    for (i=0; i<SECTIONS*2;i++)
    {
        w[i] = 0;
    }
    if (bandFlag)
    {
       // Generate filter coefficients for high band
       coefGen(gain1000, NEG_6dB, &C[0], 1000);

       // Generate filter coefficients for low band
       coefGen(gain200, POS_6dB, &C[5], 200);
       bandFlag = 0;
    }
else
    {
       // Generate filter coefficients for high band
    coefGen(gain1000, POS_6dB, &C[0], 1000);

       // Generate filter coefficients for low band
    coefGen(gain200, NEG_6dB, &C[5], 200);
       bandFlag = 1;
    }
}
```

7. Listen to the equalizer output and alter the equalizer settings and listen again.

8. Add a function that will be called periodically to switch the upper and lower bands. The function `switchBand( )` that swaps upper and lower bands is listed in Table 5.17.

9. In order for DSP/BIOS to call `switchBand( )` periodically, we need to add a new DSP/BIOS object. Open the DSP/BIOS CDB file from the project, go to the **Scheduling** and add a new object under the **PRD**. Rename the new object to `changeBand`. Open the new PRD object `changeBnad` by right clicking it and selecting **Properties**. Set the ticks to 20 000 for running it at the 20-s rate. Change the function name to `_switchBand`. The underscore in front of the function name is necessary for DSP/BIOS call. Leave the mode as **Continuous**. Click **OK** when finish.

10. Rebuild the project and run the experiment again. This time, the equalizer will automatically switch upper and lower band settings every 20 s.

References

[1] N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1983.
[2] V. K. Ingle and J. G. Proakis, *Digital Signal Processing Using MATLAB V.4*, Boston: PWS Publishing, 1997.
[3] Math Works, Inc., *Signal Processing Toolbox for Use with MATLAB*, Math Works, Inc., 1994.
[4] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989.
[5] S. J. Orfanidis, *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1996.
[6] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1996.
[7] S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, New York: McGraw-Hill, 1998.
[8] D. Grover and J. R. Deller, *Digital Signal Processing and the Microcontroller*, Englewood Cliffs, NJ: Prentice-Hall, 1999.
[9] F. Taylor and J. Mellott, *Hands-On Digital Signal Processing*, New York: McGraw-Hill, 1998.
[10] S. D. Stearns and D. R. Hush, *Digital Signal Analysis*, 2nd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1990.
[11] S. S. Soliman and M. D. Srinath, *Continuous and Discrete Signals and Systems*, 2nd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1998.
[12] L. B. Jackson, *Digital Filters and Signal Processing*, 2nd Ed., Boston, MA: Kluwer Academic Publishers, 1989.
[13] Math Works, Inc.,*Using MATLAB*, Version 6, 2000.
[14] Math Works, Inc., *Signal Processing Toolbox User's Guide*, Version 6, 2004.
[15] Math Works, Inc., *Filter Design Toolbox User's Guide*, Version 3, 2004.
[16] Math Works, Inc., *Fixed-Point Toolbox User's Guide*, Version 1, 2004.
[17] Texas Instruments, *TMS320 DSP/BIOS User's Guide*, Literature no. SPRU423B, Nov. 2002.
[18] Texas Instruments, *TMS3205000 DSP/BIOS Application Programming Interface (API) Reference Guide*, Literature no. SPRU404E, Oct. 2002.

Exercises

1. Compute the Laplace transform of unit impulse function $\delta(t)$ and unit step function $u(t)$.

2. Given the analog system

$$H(s) = \frac{2s + 3}{s^2 + 3s + 2},$$

find the poles and zeros of the system and discuss its stability.

3. Given the transfer function

$$H(z) = \frac{0.5\left(z^2 + 0.55z - 0.2\right)}{z^3 - 0.7z^2 - 0.84z + 0.544},$$

realize the system using the direct-form II and cascade of first-order sections.

4. Draw the direct-form I and II realizations of the transfer function

$$H(z) = \frac{\left(z^2 + 2z + 2\right)(z + 0.6)}{(z - 0.8)(z + 0.8)\left(z^2 + 0.1z + 0.8\right)}.$$

5. Given an IIR filter with the transfer function

$$H(z) = \frac{\left(1 + 1.414z^{-1} + z^{-2}\right)\left(1 + 2z^{-1} + z^{-2}\right)}{\left(1 - 0.8z^{-1} + 0.64z^{-2}\right)\left(1 - 1.0833z^{-1} + 0.25z^{-2}\right)},$$

find the poles and zeros of the filter, and using the stability triangle, check if H(z) is a stable filter.

6. Considering the second-order IIR filter with the I / O equation

$$y(n) = x(n) + a_1 y(n-1) + a_2 y(n-2), \qquad n \geq 0,$$

find the transfer function H(z), and discuss the stability conditions related to the cases:

(a) $a_1^2/4 + a_2 < 0$;

(b) $a_1^2/4 + a_2 > 0$; and

(c) $a_1^2/4 + a_2 = 0$.

7. A first-order allpass filter has the transfer function

$$H(z) = \frac{z^{-1} - a}{1 - az^{-1}}.$$

(a) Draw the direct-form I and II realizations.

(b) Show that $|H(\omega)| = 1$ for all ω.

(c) Sketch the phase response of this filter.

8. Given a six-order IIR transfer function

$$H(z) = \frac{6 + 17z^{-1} + 33z^{-2} + 25z^{-3} + 20z^{-4} - 5z^{-5} + 8z^{-6}}{1 + 2z^{-1} + 3z^{-2} + z^{-3} + 0.2z^{-4} - 0.3z^{-5} - 0.2z^{-6}},$$

find the factored form of the IIR transfer function in terms of second-order sections using MATLAB.

9. Given a fourth-order IIR transfer function

$$H(z) = \frac{12 - 2z^{-1} + 3z^{-2} + 20z^{-4}}{6 - 12z^{-1} + 11z^{-2} - 5z^{-3} + z^{-4}}.$$

(a) Use MATLAB to express $H(z)$ in factored form.

(b) Develop two different cascade realizations.

(c) Develop two different parallel realizations.

10. Design and plot the magnitude response of an elliptic IIR lowpass filter with the following specifications using MATLAB: passband edge at 1600 Hz, stopband edge at 2000 Hz, passband ripple of 0.5 dB, and minimum stopband attenuation of 40 dB with sampling rate of 8 kHz. Analyze the design filter using the FVTool.

11. Use FDATool to design an IIR filter specified in Problem 10 using:

    (a) Butterworth;

    (b) Chebyshev type-I; and

    (c) Chebyshev type-II and Bessel methods.

    Show both magnitude and phase responses of the designed filters and indicate the required filter order.

12. Redo Problem 10 using the FDATool, compare the results with Problem 10, and design a quantized filter for 16-bit fixed-point DSP processors.

13. Redo Problem 12 for designing an 8-bit fixed-point filter. Show the differences with the 16-bit filter designed in Problem 12.

14. Design an IIR Butterworth bandpass filter with the following specifications: passband edges at 450 and 650 Hz, stopband edges at 300 and 750 Hz, passband ripple of 1 dB, minimum stopband attenuation of 60 dB, and sampling rate of 8 kHz. Analyze the design filter using the FVTool.

15. Redo Problem 14 using the FDATool, compare the results with Problem 14, and design a quantized filter for 16-bit fixed-point DSP processors.

16. Design a type-I Chebyshev IIR highpass filter with passband edge at 700 Hz, stopband edge at 500 Hz, passband ripple of 1 dB, and minimum stopband attenuation of 32 dB. The sampling frequency is 2 kHz. Analyze the design filter using the FVTool.

17. Redo Problem 16 using FDATool, compare the results with Problem 16, and design a quantized filter for 16-bit fixed-point DSP processors.

18. Given an IIR lowpass filter with transfer function

    $$H(z) = \frac{0.0662\left(1 + 3z^{-1} + 3z^{-2} + z^{-3}\right)}{1 - 0.9356z^{-1} + 0.5671z^{-2} - 0.1016z^{-3}},$$

    plot the impulse response using an appropriate MATLAB function and compare the result using the FVTool.

19. It is interesting to examine the frequency response of the second-order resonator filter as the radius r_p and the pole angle ω_0 are varied. Use the MATLAB to compute and plot the magnitude response for $\omega_0 = \pi/2$ and various values of r_p. Also, plot the magnitude response for $r_\mathrm{p} = 0.95$ and various values of ω_0.

20. Use MATLAB FDATool to design an 8 kHz sampling rate highpass filter with the passband starting at 3000 Hz and at least 45-dB attenuation in the stopband. Write a direct-form I IIR filter function in fixed-point C. The test data file is given in the companion CD.

21. Rewrite the highpass filter implementation in Problem 20 using C intrinsics.

22. Use MATLAB FDATool to design an 8 kHz sampling rate lowpass filter with the stopband beginning at 1000 Hz with at least 60-dB attenuation. Write the direct-form I IIR filter in C55x assembly language. The test data file is given in the companion CD.

23. Create a real-time experiment using DSP/BIOS for the direct-form I IIR filter designed by Problem 22.

24. The cascade IIR filter implementation in this problem has some issues that need to be corrected. Identify the problems and make corrections. Run the test and compare the result with the experiment given in Section 5.7.3. The software for this exercise is included in the companion CD.

25. Write the IIR filter function using intrinsics for Problem 24. Compare the profile result against the experiment given in Section 5.7.4.

26. Write the program for IIR filter function defined in Problem 24 using C55x assembly language. Compare the profile result against the experiment given in Section 5.7.5.

27. Can the experiment given in Section 5.7.5 be optimized even further? Try to improve the run-time efficiency of the experiment given in Section 5.7.5.

28. Use MATLAB FDATool to design two 16-kHz sampling rate filters. A lowpass filter with stopband at 800 Hz and attenuation at least 40 dB in the stopband and a highpass filter with the passband starting from 1200 Hz and at least 40-dB attenuation in its stopband. Modify the experiment given in Section 5.7.6 such that the DSK uses stereo line-in at 16 kHz sampling rate and PIP frame size set to 40. Place the lowpass filter on the left channel of the audio path and the highpass filter on the right channel of the audio path. Use a high-fidelity CD as audio input and listen to the filter output of the left and right channels.

29. Using Equation (5.70), design a three-band equalizer with normalized resonate frequencies of 0.05, 0.25, and 0.5. The equalizer must have a dynamic range of at least $\pm$ 9 dB at 1 dB step with:

    (a) 8 kHz sampling rate;

    (b) 48 kHz sampling rate.

30. Write a fixed-point C program to verify the three-band equalizer performance from Problem 29. Implement this real-time three-band equalizer using DSK with:

    (a) 8 kHz sampling rate;

    (b) 48 kHz sampling rate.

6

Frequency Analysis and Fast Fourier Transform

This chapter introduces the properties, applications, and implementations of the discrete Fourier transform (DFT). Because of the development of the fast Fourier transform algorithms, the DFT is now widely used for spectral analysis and fast convolution.

6.1 Fourier Series and Transform

In this section, we will introduce the representation of analog periodic signals using Fourier series and the analysis of finite-energy signals using Fourier transform.

6.1.1 Fourier Series

A periodic signal can be represented as the sum of an infinite number of harmonic-related sinusoids and complex exponentials. The representation of periodic signal $x(t)$ with period T_0 is the Fourier series defined as

$$x(t) = \sum_{k=-\infty}^{\infty} c_k e^{jk\Omega_0 t}, \tag{6.1}$$

where c_k is the Fourier series coefficient, $\Omega_0 = 2\pi/T_0$ is the fundamental frequency, and $k\Omega_0$ is the frequency of the kth harmonic.

The kth Fourier coefficient c_k is expressed as

$$c_k = \frac{1}{T_0}\int_{T_0} x(t)e^{-jk\Omega_0 t}\,\mathrm{d}t. \tag{6.2}$$

For an odd function $x(t)$, it is easier to calculate the interval from 0 to T_0. For an even function, integration from $-T_0/2$ to $T_0/2$ is commonly used. The term $c_0 = \frac{1}{T_0}\int_{T_0} x(t)\,\mathrm{d}t$ is called the DC component because it equals the average value of $x(t)$ over one period.

Example 6.1: A rectangular pulse train is a periodic signal with period T_0 and can be expressed as

$$x(t) = \begin{cases} A, & kT_0 - \tau/2 \le t \le kT_0 + \tau/2 \\ 0, & \text{otherwise} \end{cases}, \tag{6.3}$$

where $k = 0, \pm 1, \pm 2, \ldots$, and $\tau < T_0$ is the width of rectangular pulse with amplitude A. Since $x(t)$ is an even function, its Fourier coefficients can be calculated as

$$c_k = \frac{1}{T_0} \int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} A e^{-jk\Omega_0 t}\, \mathrm{d}t = \frac{A}{T_0} \left[\left. \frac{e^{-jk\Omega_0 t}}{-jk\Omega_0} \right|_{-\frac{\tau}{2}}^{\frac{\tau}{2}} \right] = \frac{A\tau}{T_0} \frac{\sin(k\Omega_0 \tau/2)}{k\Omega_0 \tau/2}. \tag{6.4}$$

This equation shows that c_k has a maximum value of $A\tau/T_0$ at the DC frequency $\Omega_0 = 0$, decays to zero as $\Omega_0 \to \pm\infty$, and equals zero at frequencies that are multiples of π.

The plot of $|c_k|^2$ shows that the power of the periodic signal is distributed among the frequency components. Since the power of a periodic signal exists only at discrete frequencies $k\Omega_0$, the signal has a line spectrum. The spacing between two consecutive spectral lines is equal to the fundamental frequency Ω_0. For the rectangular pulse train with a fixed period T_0, the effect of decreasing τ is to spread the signal power over the entire frequency range. On the other hand, when τ is fixed but the period T_0 increases, the spacing between adjacent spectral lines decreases.

Example 6.2: Consider a perfect sinewave expressed as

$$x(t) = \sin(2\pi f_0 t).$$

Using Euler's formula (see Appendix A) and Equation (6.1), we obtain

$$\sin(2\pi f_0 t) = \frac{1}{2j}\left(e^{j2\pi f_0 t} - e^{-j2\pi f_0 t}\right) = \sum_{k=-\infty}^{\infty} c_k e^{jk2\pi f_0 t}.$$

Therefore, the Fourier series coefficients can be calculated as

$$c_k = \begin{cases} 1/2j, & k = 1 \\ -1/2j, & k = -1 \\ 0, & \text{otherwise} \end{cases}. \tag{6.5}$$

This equation indicates that the power of a pure sinewave is distributed only at the harmonics $k = \pm 1$, a perfect line spectrum.

6.1.2 Fourier Transform

We have shown that a periodic signal has a line spectrum and the space between two adjacent spectral lines is equal to the fundamental frequency $\Omega_0 = 2\pi/T_0$. As T_0 increases, the line space decreases and the number of frequency components increases. If we increase the period without limit (i.e., $T_0 \to \infty$), the line spacing tends toward zero with infinite frequency components. Therefore, the discrete line components converge into a continuum of frequency spectrum.

In practice, most real-world signals, such as speech, are not periodic. They can be approximated by periodic signals with infinite period, i.e., $T_0 \to \infty$ (or $\Omega_0 \to 0$). Therefore, the number of exponential components in Equation (6.1) tends toward infinity, and the summation becomes integration over the range $(-\infty, \infty)$. Thus, Equation (6.1) becomes

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\Omega) e^{j\Omega t}\, \mathrm{d}\Omega. \tag{6.6}$$

This is the inverse Fourier transform. Similarly, Equation (6.2) becomes

$$X(\Omega) = \int_{-\infty}^{\infty} x(t)e^{-j\Omega t}\,\mathrm{d}t, \tag{6.7}$$

or

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi f t}\,\mathrm{d}t. \tag{6.8}$$

This is the Fourier transform of the analog $x(t)$.

Example 6.3: Calculate the Fourier transform of function $x(t) = e^{-at}u(t)$, where $a > 0$ and $u(t)$ is the unit-step function. From Equation (6.7), we have

$$\begin{aligned} X(\Omega) &= \int_{-\infty}^{\infty} e^{-at}u(t)e^{-j\Omega t}\,\mathrm{d}t = \int_{0}^{\infty} e^{-(a+j\Omega)t}\,\mathrm{d}t \\ &= \frac{1}{a+j\Omega}. \end{aligned}$$

For a function $x(t)$ defined over a finite interval T_0, i.e., $x(t) = 0$ for $|t| > T_0/2$, the Fourier series coefficients c_k can be expressed in terms of $X(\Omega)$ using Equations (6.2) and (6.7) as

$$c_k = \frac{1}{T_0}X(k\Omega_0). \tag{6.9}$$

Therefore, the Fourier transform $X(\Omega)$ of a finite interval function at a set of equally spaced points on the Ω-axis is specified exactly by the Fourier series coefficients c_k.

6.2 Discrete Fourier Transform

In this section, we introduce the discrete-time Fourier transform and discrete Fourier transform of digital signals.

6.2.1 Discrete-Time Fourier Transform

The discrete-time Fourier transform (DTFT) of a discrete-time signal $x(nT)$ is defined as

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(nT)e^{-j\omega nT}. \tag{6.10}$$

It shows that $X(\omega)$ is a periodic function with period 2π. Thus, the frequency range of a discrete-time signal is unique over the range $(-\pi, \pi)$ or $(0, 2\pi)$.

The DTFT of $x(nT)$ can also be defined in terms of normalized frequency as

$$X(F) = \sum_{n=-\infty}^{\infty} x(nT)e^{-j2\pi Fn}. \tag{6.11}$$

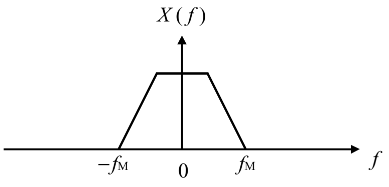
Comparing this equation with Equation (6.8), the periodic sampling imposes a relationship between the independent variables t and n as $t = nT = n/f_s$. It can be shown that

$$X(F) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X(f - kf_s). \tag{6.12}$$

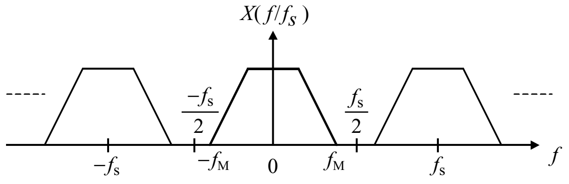
This equation states that $X(F)$ is the sum of an infinite number of $X(f)$, scaled by $1/T$, and then frequency shifted to kf_s. It also states that $X(F)$ is a periodic function with period $T = 1/f_s$.

Example 6.4: Assume that a continuous-time signal $x(t)$ is bandlimited to f_M, i.e., $|X(f)| = 0$ for $|f| \geq f_M$, where f_M is the bandwidth of signal $x(t)$. The spectrum is zero for $|f| \geq f_M$ as shown in Figure 6.1(a).

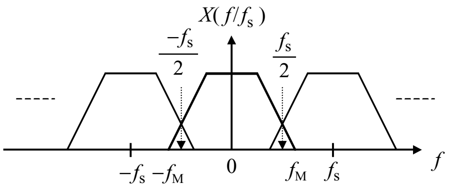
As shown in Equation (6.12), sampling extends the spectrum $X(f)$ repeatedly on both sides of the f-axis. When the sampling rate f_s is greater than $2f_M$, i.e., $f_M \leq f_s/2$, the spectrum $X(f)$ is preserved in $X(F)$ as shown in Figure 6.1(b). In this case, there is no aliasing because the spectrum



(a) Spectrum of an analog signal.



(b) Spectrum of discrete-time signal when the sampling theorem is satisfied.



(c) Spectrum of discrete-time signal when the sampling theorem is violated.

Figure 6.1 Spectrum replication caused by sampling: (a) spectrum of analog bandlimited signal $x(t)$; (b) sampling theorem is satisfied; and (c) overlap of spectral components

of the discrete-time signal is identical (except the scaling factor $1/T$) to the spectrum of the analog signal within the frequency range $|f| \leq f_s/2$ or $|F| \leq 1$. The analog signal $x(t)$ can be recovered from the discrete-time signal $x(nT)$ by passing it through an ideal lowpass filter with bandwidth f_M and gain T. This verifies the sampling theorem defined in Equation (1.3).

However, if the sampling rate $f_s < 2f_M$, the shifted replicas of $X(f)$ will overlap as shown in Figure 6.1(c). This phenomenon is called aliasing since the frequency components in the overlapped region are corrupted.

The DTFT $X(\omega)$ is a continuous function of frequency ω and the computation requires an infinite-length sequence $x(n)$. We have defined DFT in Section 3.2.6 for N samples of $x(n)$ at N discrete frequencies. Therefore, DFT is a numerically computable transform.

6.2.2 Discrete Fourier Transform

The DFT of a finite-duration sequence $x(n)$ of length N is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j(2\pi/N)kn}, \qquad k = 0, 1, \ldots, N-1, \tag{6.13}$$

where $X(k)$ is the kth DFT coefficient and the upper and lower indices in the summation reflect the fact that $x(n) = 0$ outside the range $0 \leq n \leq N-1$. The DFT is equivalent to taking N samples of DTFT $X(\omega)$ over the interval $0 \leq \omega < 2\pi$ at N discrete frequencies $\omega_k = 2\pi k/N$, where $k = 0, 1, \ldots, N-1$. The spacing between two successive $X(k)$ is $2\pi/N$ rad (or f_s/N Hz).

Example 6.5: If the signal $\{x(n)\}$ is real valued and N is an even number, we can show that

$$X(0) = \sum_{n=0}^{N-1} x(n)$$

and

$$X(N/2) = \sum_{n=0}^{N-1} e^{-j\pi n}x(n) = \sum_{n=0}^{N-1} (-1)^n x(n).$$

Therefore, the DFT coefficients $X(0)$ and $X(N/2)$ are real values.

The DFT defined in Equation (6.13) can also be written as

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn}, \qquad k = 0, 1, \ldots, N-1, \tag{6.14}$$

where

$$W_N^{kn} = e^{-j\left(\frac{2\pi}{N}\right)kn} = \cos\left(\frac{2\pi kn}{N}\right) - j\sin\left(\frac{2\pi kn}{N}\right), \qquad 0 \leq k, n \leq N-1. \tag{6.15}$$

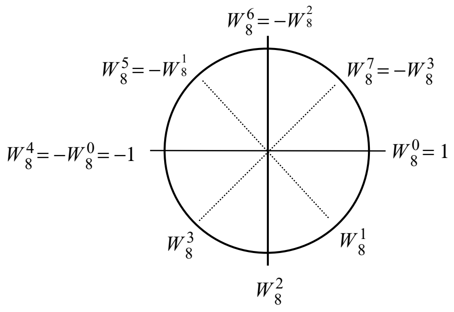


Figure 6.2 Twiddle factors for DFT, $N = 8$

The parameter W_N^{kn} is called the twiddle factors of the DFT. Because $W_N^N = e^{-j2\pi} = 1 = W_N^0$, W_N^k, $k = 0, 1, \ldots, N-1$ are the N roots of unity in clockwise direction on the unit circle. It can be shown that $W_N^{N/2} = e^{-j\pi} = -1$.

The twiddle factors have the symmetry property

$$W_N^{k+N/2} = -W_N^k, \qquad 0 \le k \le N/2 - 1, \tag{6.16}$$

and the periodicity property

$$W_N^{k+N} = W_N^k. \tag{6.17}$$

Figure 6.2 illustrates the cyclic property of the twiddle factors for an 8-point DFT.

Example 6.6: Consider the finite-length signal

$$x(n) = a^n, \qquad n = 0, 1, \ldots, N-1,$$

where $0 < a < 1$. The DFT of $x(n)$ is computed as

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} a^n e^{-j(2\pi k/N)n} = \sum_{n=0}^{N-1} \left(ae^{-j2\pi k/N}\right)^n \\ &= \frac{1 - \left(ae^{-j2\pi k/N}\right)^N}{1 - ae^{-j2\pi k/N}} = \frac{1 - a^N}{1 - ae^{-j2\pi k/N}}, \quad k = 0, 1, \ldots, N-1. \end{aligned}$$

The inverse discrete Fourier transform (IDFT) is used to transform the frequency domain $X(k)$ back into the time-domain signal $x(n)$. The IDFT is defined as

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j(2\pi/N)kn} = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-kn}, \qquad n = 0, 1, \ldots, N-1. \tag{6.18}$$

This is identical to the DFT with the exception of the normalizing factor $1/N$ and the opposite sign of the exponent of the twiddle factors.

The DFT and IDFT defined in Equations (6.14) and (6.18), respectively, can be expressed in matrix-vector form as

$$\mathbf{X} = \mathbf{W}\mathbf{x} \tag{6.19}$$

and

$$\mathbf{x} = \frac{1}{N}\mathbf{W}^*\mathbf{X}, \tag{6.20}$$

where $\mathbf{x} = [x(0)\,x(1)\ldots x(N-1)]^T$ is the signal vector, the complex vector $\mathbf{X} = [X(0)X(1)\ldots X(N-1)]^T$ contains the DFT coefficients, and the NxN twiddle-factor matrix (or DFT matrix) $\mathbf{W}$ is defined by

$$\mathbf{W} = \left[W_N^{kn}\right]_{0\le k,n\le N-1} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & W_N^1 & \cdots & W_N^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{N-1} & \cdots & W_N^{(N-1)^2} \end{bmatrix}, \tag{6.21}$$

and $\mathbf{W}^*$ is the complex conjugate of the matrix $\mathbf{W}$. Since $\mathbf{W}$ is a symmetric matrix, the inverse matrix $\mathbf{W}^{-1} = \frac{1}{N}\mathbf{W}^*$ was used to derive Equation (6.20).

Example 6.7: Given $x(n) = \{1, 1, 0, 0\}$, the DFT of this 4-point sequence can be computed using the matrix formulation as

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W_4^1 & W_4^2 & W_4^3 \\ 1 & W_4^2 & W_4^4 & W_4^6 \\ 1 & W_4^3 & W_4^6 & W_4^9 \end{bmatrix}\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1-j \\ 0 \\ 1+j \end{bmatrix},$$

where we used symmetry and periodicity properties given in Equations (6.16) and (6.17) to obtain $W_4^0 = W_4^4 = 1$, $W_4^1 = W_4^9 = -j$, $W_4^2 = W_4^6 = -1$, and $W_4^3 = j$. The IDFT can be computed as

$$\mathbf{x} = \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W_4^{-1} & W_4^{-2} & W_4^{-3} \\ 1 & W_4^{-2} & W_4^{-4} & W_4^{-6} \\ 1 & W_4^{-3} & W_4^{-6} & W_4^{-9} \end{bmatrix}\mathbf{X} = \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & -j & -1 & j \end{bmatrix}\begin{bmatrix} 2 \\ 1-j \\ 0 \\ 1+j \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}.$$

The DFT coefficients are equally spaced on the unit circle with frequency intervals of f_s/N (or $2\pi/N$). Therefore, the frequency resolution of the DFT is $\Delta = f_s/N$. The frequency sample $X(k)$ represents discrete frequency

$$f_k = k\frac{f_s}{N}, \qquad \text{for } k = 0, 1, \ldots, N-1. \tag{6.22}$$

Since the DFT coefficient $X(k)$ is a complex variable, it can be expressed in polar form as

$$X(k) = |X(k)|e^{j\phi(k)}, \tag{6.23}$$

where the DFT magnitude spectrum is defined as

$$|X(k)| = \sqrt{\{\mathrm{Re}[X(k)]\}^2 + \{\mathrm{Im}[X(k)]\}^2} \tag{6.24}$$

and the phase spectrum is defined as

$$\phi(k) = \tan^{-1}\left\{\frac{\mathrm{Im}[X(k)]}{\mathrm{Re}[X(k)]}\right\}. \tag{6.25}$$

Example 6.8: Consider a finite-length DC signal $x(n) = c$, where $n = 0, 1, \ldots, N-1$. From Equation (6.14), we obtain

$$X(k) = c\sum_{n=0}^{N-1} W_N^{kn} = c\frac{1 - W_N^{kN}}{1 - W_N^k}.$$

Since $W_N^{kN} = e^{-j\left(\frac{2\pi}{N}\right)kN} = 1$ for all k, and $W_N^k \neq 1$ for $k \neq iN$, we have $X(k) = 0$ for $k = 1, 2, \ldots, N-1$. For $k = 0$, $\sum_{n=0}^{N-1} W_N^{kn} = N$. Therefore, we obtain

$$X(k) = cN\delta(k), \qquad k = 0, 1, \ldots, N-1.$$

6.2.3 Important Properties

This section introduces several important properties of DFT that are useful for analyzing digital signals and systems.

Linearity: If $\{x(n)\}$ and $\{y(n)\}$ are digital sequences of the same length,

$$\begin{aligned}\mathrm{DFT}[ax(n) + by(n)] &= a\mathrm{DFT}[x(n)] + b\mathrm{DFT}[y(n)] \\ &= aX(k) + bY(k),\end{aligned} \tag{6.26}$$

where a and b are arbitrary constants. Linearity allows us to analyze complex signals and systems by evaluating their individual components. The overall response is the combination of individual results evaluated at every frequency component.

Complex conjugate: If the sequence $\{x(n), 0 \leq n \leq N-1\}$ is real valued, then

$$X(-k) = X^*(k) = X(N-k), \qquad 0 \leq k \leq N-1, \tag{6.27}$$

where $X^*(k)$ is the complex conjugate of $X(k)$. Or equivalently,

$$X(M+k) = X^*(M-k), \qquad 0 \leq k \leq M, \tag{6.28}$$

where $M = N/2$ if N is even, or $M = (N-1)/2$ if N is odd. This property shows that only the first $(M+1)$ DFT coefficients from $k = 0$ to M are independent as illustrated in Figure 6.3. For complex signals, however, all N complex outputs carry useful information.

From the symmetry property, we obtain

$$|X(k)| = |X(N-k)|, \qquad k = 1, 2, \ldots, M-1 \tag{6.29}$$

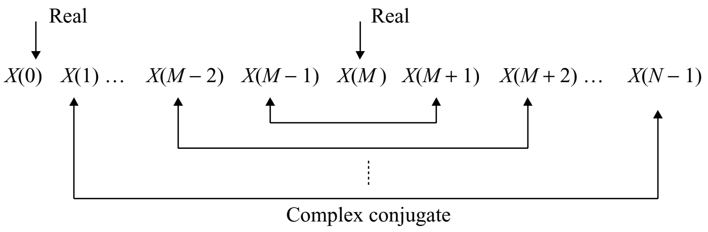

Real
Real
X(0) X(1) ... X(M − 2) X(M − 1) X(M) X(M + 1) X(M + 2) ... X(N − 1)
Complex conjugate

Figure 6.3 Complex-conjugate property, N is an even number

and

$$\phi(k) = -\phi(N-k), \qquad k = 1, 2, \ldots, M-1. \tag{6.30}$$

Circular shifts: Let $y(n)$ be a circular-shifted sequence defined as

$$y(n) = x(n-m)_{\bmod N}, \tag{6.31}$$

where m is the number of samples by which $x(n)$ is shifted to the right and the modulo operation

$$0 \le (n-m)_{\bmod N} = (n - m \pm iN) < N. \tag{6.32}$$

For example, if $m = 1$, $x(0)$ shifts to $x(1)$, $x(1)$ shifts to $x(2)$, ..., $x(N-2)$ shifts to $x(N-1)$, and $x(N-1)$ shifts back to $x(0)$. Thus, a circular shift of an N-point sequence is equivalent to a linear shift of its periodic extension. Considering the $y(n)$ defined in Equation (6.31), we have

$$Y(k) = e^{-j(2\pi k/N)m}X(k) = W_N^{mk}X(k). \tag{6.33}$$

DFT and z-transform: DFT is equal to the z-transform of a sequence $x(n)$ of length N, evaluated on the unit circle at N equally spaced frequencies $\omega_k = 2\pi k/N$, where $k = 0, 1, \ldots, N-1$. That is,

$$X(k) = X(z)\big|_{z=e^{j\left(\frac{2\pi}{N}\right)k}}, \quad k = 0, 1, \ldots, N-1. \tag{6.34}$$

Circular convolution: If $x(n)$ and $h(n)$ are real-valued N-periodic sequences, $y(n)$ is the circular convolution of $x(n)$ and $h(n)$ defined as

$$y(n) = x(n) \otimes h(n), \qquad n = 0, 1, \ldots, N-1, \tag{6.35}$$

where $\otimes$ denotes circular convolution. The circular convolution in time domain is equivalent to multiplication in the frequency domain expressed as

$$Y(k) = X(k)H(k), \qquad k = 0, 1, \ldots, N-1. \tag{6.36}$$

Note that the shorter sequence must be padded with zeros in order to have the same length for computing circular convolution.

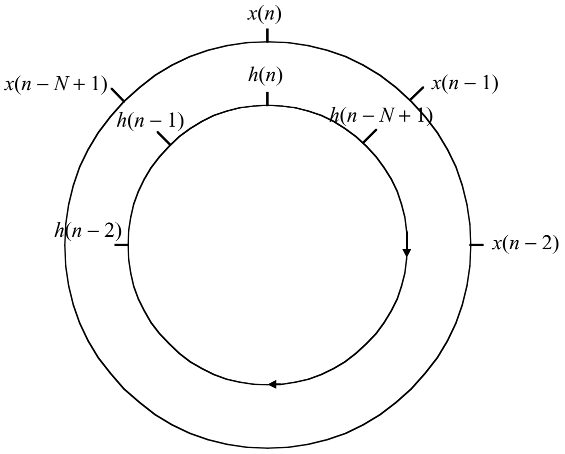


Figure 6.4 Circular convolution of two sequences using the concentric circle approach

Figure 6.4 illustrates the cyclic property of circular convolution using two concentric circles. To perform circular convolution, N samples of $x(n)$ are equally spaced around the outer circle in the clockwise direction, and N samples of $h(n)$ are displayed on the inner circle in the counterclockwise direction starting at the same point. Corresponding samples on the two circles are multiplied, and the products are summed to form an output. The successive value of the circular convolution is obtained by rotating the inner circle of one sample in the clockwise direction, and repeating the operation of computing the sum of corresponding products. This process is repeated until the first sample of inner circle lines up with the first sample of the exterior circle again.

Example 6.9: Given two 4-point sequences $x(n) = \{1, 2, 3, 4\}$ and $h(n) = \{1, 0, 1, 1\}$. Using the circular convolution method illustrated in Figure 6.4, we can obtain

$$n = 0, y(0) = 1 \times 1 + 1 \times 2 + 1 \times 3 + 0 \times 4 = 6$$

$$n = 1, y(1) = 0 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 4 = 9$$

$$n = 2, y(2) = 1 \times 1 + 0 \times 2 + 1 \times 3 + 1 \times 4 = 8$$

$$n = 3, y(3) = 1 \times 1 + 1 \times 2 + 0 \times 3 + 1 \times 4 = 7$$

Therefore, we obtain

$$y(n) = x(n) \otimes h(n) = \{6, 9, 8, 7\}.$$

Note that the linear convolution of sequences $x(n)$ and $h(n)$ results in

$$y(n) = x(n) * h(n) = \{1, 2, 4, 7, 5, 7, 4\},$$

which is also implemented in MATLAB script `example6_9.m`.

To eliminate the circular effect and ensure that the DFT method results in a linear convolution, the signals must be zero-padded. Since the linear convolution of two sequences of lengths L and M will result in a sequence of length $L + M - 1$, the two sequences must be extended to the length of $L + M - 1$ or greater by zero-padding. That is, append the sequence of length L with at least $M - 1$ zeros, and pad the sequence of length M with at least $L - 1$ zeros.

Example 6.10: Consider the same sequences $h(n)$ and $x(n)$ given in Example 6.9. If those 4-point sequences are zero-padded to 8 points as $x(n) = \{1, 2, 3, 4, 0, 0, 0, 0\}$ and $h(n) = \{1, 0, 1, 1, 0, 0, 0, 0\}$, the resulting circular convolution is

$$n = 0, y(0) = 1 \times 1 + 0 \times 2 + 0 \times 3 + 0 \times 4 + 0 \times 0 + 1 \times 0 + 1 \times 0 + 0 \times 0 = 1$$

$$n = 1, y(1) = 0 \times 1 + 1 \times 2 + 0 \times 3 + 0 \times 4 + 0 \times 0 + 0 \times 0 + 1 \times 0 + 1 \times 0 = 2$$

$$n = 2, y(2) = 1 \times 1 + 0 \times 2 + 1 \times 3 + 0 \times 4 + 0 \times 0 + 0 \times 0 + 0 \times 0 + 1 \times 0 = 4$$

$$\vdots$$

We finally have

$$y(n) = x(n) \otimes h(n) = \{1, 2, 4, 7, 5, 7, 4, 0\}.$$

This result is identical to the linear convolution of the two sequences as given in Example 6.9. Thus, the linear convolution can be realized by the circular convolution with proper zero-padding. MATLAB script `example6_10.m` implements the circular convolution of zero-padded sequences using DFT.

Zero-padding can be implemented using the MATLAB function `zeros`. For example, the 4-point sequence $x(n)$ given in Example 6.9 can be zero-padded to 8 points with the following command,

```
x = [1, 2, 3, 4, zeros(1, 4)];
```

where the MATLAB function `zeros(1, N)` generates a row vector of `N` zeros.

6.3 Fast Fourier Transforms

The drawback of using DFT for practical applications is its intensive computational requirement. To compute each $X(k)$ defined in Equation (6.14), we need approximately N complex multiplications and additions. For computing N samples of $X(k)$ for $k = 0, 1, \ldots, N - 1$, approximately N^2 complex multiplications and $(N^2 - N)$ complex additions are required. Since a complex multiplication requires four real multiplications and two real additions, the total number of arithmetic operations required for computing N-point DFT is proportional to $4N^2$, which becomes huge for large N.

The twiddle factor W_N^{kn} is a periodic function with a limited number of distinct values since

$$W_N^{kn} = W_N^{(kn) \bmod N}, \qquad \text{for } kn > N \tag{6.37}$$

and $W_N^N = 1$. Therefore, different powers of W_N^{kn} have the same value as shown in Equation (6.37). In addition, some twiddle factors have either real or imaginary parts equal to 1 or 0. By reducing these redundancies, a very efficient algorithm called the fast Fourier transform (FFT) can be derived, which requires only $N \log_2 N$ operations instead of N^2 operations. If $N = 1024$, FFT requires about 10^4 operations instead of 10^6 operations for DFT.

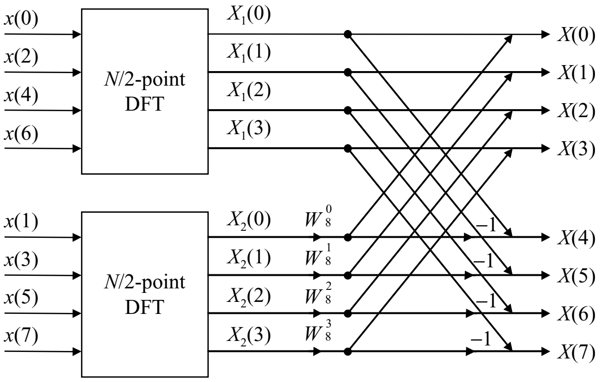


Figure 6.5 Decomposition of an N-point DFT into two $N/2$ DFTs, $N = 8$

The generic term FFT covers many different algorithms with different features, advantages, and disadvantages. Each FFT algorithm has different strengths and makes different trade-offs in terms of code complexity, memory usage, and computation requirements. In this section, we introduce two classes of FFT algorithms: decimation-in-time and decimation-in-frequency.

6.3.1 Decimation-in-Time

For the decimation-in-time algorithms, the sequence $\{x(n), n = 0, 1, \ldots, N - 1\}$ is first divided into two shorter interwoven sequences: the even numbered sequence

$$x_1(m) = x(2m), \quad m = 0, 1, \ldots, (N/2) - 1 \tag{6.38}$$

and the odd numbered sequence

$$x_2(m) = x(2m + 1), \quad m = 0, 1, \ldots, (N/2) - 1. \tag{6.39}$$

Apply the DFT defined in Equation (6.14) to these two sequences of length $N/2$, and combine the resulting $N/2$-point $X_1(k)$ and $X_2(k)$ to produce the final N-point DFT. This procedure is illustrated in Figure 6.5 for $N = 8$.

The structure shown on the right side of Figure 6.5 is called the butterfly network because of its crisscross appearance, which can be generalized in Figure 6.6. Each butterfly involves just a single complex multiplication by a twiddle factor W_N^k, one addition, and one subtraction.

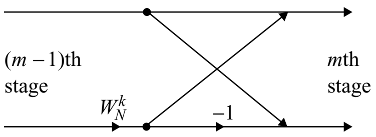


Figure 6.6 Flow graph for a butterfly computation

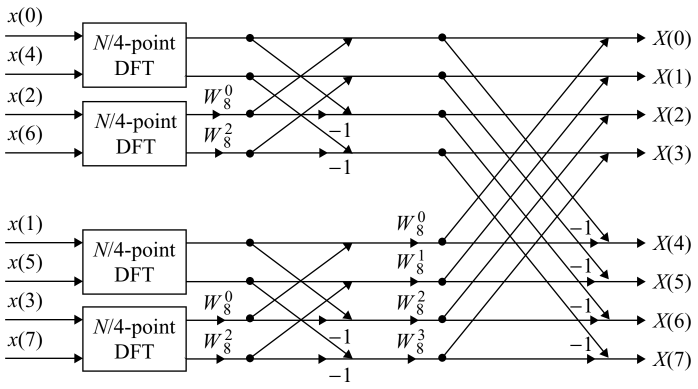


Figure 6.7 Flow graph illustrating second step of N-point DFT, $N = 8$

Since N is a power of 2, $N/2$ is an even number. Each of these $N/2$-point DFTs can be computed by two smaller $N/4$-point DFTs. This second step process is illustrated in Figure 6.7.

By repeating the same process, we will finally obtain a set of 2-point DFTs since N is a power of 2. For example, the $N/4$-point DFT becomes a 2-point DFT in Figure 6.7 for $N = 8$. Since the first stage uses the twiddle factor $W_N^0 = 1$, the 2-point butterfly network illustrated in Figure 6.8 requires only one addition and one subtraction.

Example 6.11: Consider the 2-point FFT algorithm which has two input samples $x(0)$ and $x(1)$. The DFT output samples $X(0)$ and $X(1)$ can be computed as

$$X(k) = \sum_{n=0}^{1} x(n)W_2^{nk}, \qquad k = 0, 1.$$

Since $W_2^0 = 1$ and $W_2^1 = e^{-\pi} = -1$, we have

$$X(0) = x(0) + x(1) \quad \text{and} \quad X(1) = x(0) - x(1).$$

The computation is identical to the signal-flow graph shown in Figure 6.8.

As shown in Figure 6.7, the input sequence is arranged as if each index was written in binary form and then the order of binary digits was reversed. This bit-reversal process is illustrated in Table 6.1 for the case of $N = 8$. The input sample indices in decimal are first converted to their binary representations, the binary bit streams are reversed, and then the reversed binary numbers are converted back to decimal

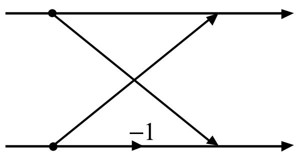


Figure 6.8 Flow graph of 2-point DFT

Table 6.1 Example of bit-reversal process, $N = 8$ (3 bits)

Input sample index		Bit-reversed sample index	
Decimal	Binary	Binary	Decimal
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

values to give the reordered time indices. Most modern DSP processors (such as the TMS320C55x) provide the bit-reversal addressing mode to efficiently support this process.

For the FFT algorithm shown in Figure 6.7, the input values are no longer needed after the computation of output values at a particular stage. Thus, the memory locations used for the FFT outputs can be the same locations used for storing the input data. This observation supports the in-place FFT algorithms that use the same memory locations for both the input and output numbers.

6.3.2 Decimation-in-Frequency

The development of the decimation-in-frequency FFT algorithm is similar to the decimation-in-time algorithm presented in the previous section. The first step is to divide the data sequence into two halves, each of $N/2$ samples. The next step is to separate the frequency terms $X(k)$ into even and odd samples of k. Figure 6.9 illustrates the first decomposition of an N-point DFT into two $N/2$-point DFTs.

Continue the process of decomposition until the last stage consists of 2-point DFTs. The decomposition and symmetry relationships are reversed from the decimation-in-time algorithm. The bit reversal occurs at the output instead of the input and the order of the output samples $X(k)$ will be rearranged as Table 6.1. Figure 6.10 illustrates the butterfly representation for the decimation-in-frequency FFT algorithm.

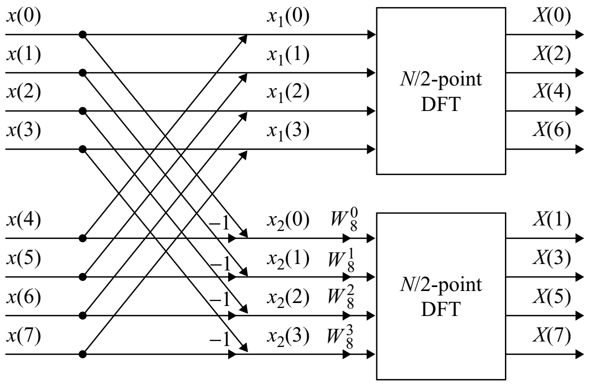


Figure 6.9 Decomposition of an N-point DFT into two $N/2$ DFTs

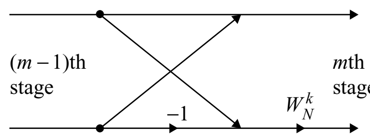


Figure 6.10 Butterfly network for decimation-in-frequency FFT algorithm

The FFT algorithms introduced in this chapter are based on two-input, two-output butterfly computations, and are classified as radix-2 FFT algorithms. It is possible to use other radix values to develop FFT algorithms. However, these algorithms only work well for some specific FFT lengths. In addition, these algorithms are more complicated than the radix-2 FFT algorithms and the programs for real-time implementation are not widely available for DSP processors.

6.3.3 Inverse Fast Fourier Transform

The FFT algorithms introduced in the previous sections can be modified to efficiently compute the inverse FFT (IFFT). By complex conjugating both sides of Equation (6.18), we have

$$x^*(n) = \frac{1}{N}\sum_{k=0}^{N-1} X^*(k)W_N^{kn}, \qquad n = 0, 1, \ldots, N-1. \tag{6.40}$$

This equation shows we can use an FFT algorithm to compute the IFFT by first conjugating the DFT coefficients $X(k)$ to obtain $X^*(k)$, computing the DFT of $X^*(k)$ using an FFT algorithm, scaling the results by $1/N$ to obtain $x^*(n)$, and then complex conjugating $x^*(n)$ to obtain the output sequence $x(n)$. If the signal is real valued, the final conjugation operation is not required.

6.4 Implementation Considerations

Many FFT routines are available in C and assembly programs for some specific DSP processors; however, it is important to understand the implementation issues in order to use FFT properly.

6.4.1 Computational Issues

The FFT routines accept complex-valued inputs; therefore, the number of memory locations required is $2N$ for N-point FFT. To use the available complex FFT program for real-valued signals, we have to set the imaginary parts to zero. The complex multiplication has the form

$$(a + jb)(c + jd) = (ac + bd) + j(bc + ad),$$

which requires four real multiplications and two real additions. The number of multiplication and the storage requirements can be reduced if the signal has special properties. For example, if $x(n)$ is real, only $N/2$ samples from $X(0)$ to $X(N/2)$ need to be computed as shown by complex-conjugate property.

In most FFT programs developed for general-purpose computers, the computation of twiddle factors W_N^{kn} defined in Equation (6.15) is embedded in the program. However, the twiddle factors only need to be computed once during the program initialization stage. In the implementation of FFT algorithm on DSP processors, it is preferable to tabulate the values of twiddle factors so that they can be looked up during the computation of FFT.

The complexity of FFT algorithms is usually measured by the required number of arithmetic operations (multiplications and additions). In practical real-time implementations with DSP processors, the architecture, instruction set, data structures, and memory organizations of the processors are critical factors. For example, modern DSP processors usually provide bit-reversal addressing and a high degree of instruction parallelism to implement FFT algorithms.

6.4.2 Finite-Precision Effects

From the signal-flow graph of the FFT algorithm shown in Figure 6.7, $X(k)$ will be computed by a series of butterfly computations with a single complex multiplication per butterfly network. Note that some butterfly networks with coefficients ± 1 (such as 2-point FFT in the first stage) do not require multiplication. Figure 6.7 also shows that the computation of N-point FFT requires $M = \log_2 N$ stages. There are $N/2$ butterflies in the first stage, $N/4$ in the second stage, and so on. Thus, the total number of butterflies required is

$$\frac{N}{2} + \frac{N}{4} + \cdots + 2 + 1 = N - 1. \tag{6.41}$$

The quantization errors introduced at the mth stage are multiplied by the twiddle factors at each subsequent stage. Since the magnitude of the twiddle factor is always unity, the variances of the quantization errors do not change while propagating to the output.

The definition of DFT given in Equation (6.14) shows that we can scale the input sequence with the condition

$$|x(n)| < \frac{1}{N} \tag{6.42}$$

to prevent the overflow at the output because $|e^{-j(2\pi/N)kn}| = 1$. For example, in a 1024-point FFT, the input data must be shifted right by 10 bits ($1024 = 2^{10}$). If the original data is 16 bits, the effective wordlength of the input data is reduced to only 6 bits after scaling. This worst-case scaling substantially reduces the resolution of the FFT results.

Instead of scaling the input samples by $1/N$ at the beginning of the FFT, we can scale the signals at each stage. Figure 6.6 shows that we can avoid overflow within the FFT by scaling the input at each stage by 0.5 because the outputs of each butterfly involve the addition of two numbers. This scaling process provides a better accuracy than the scaling of input by $1/N$.

An alternative conditional scaling method examines the results of each FFT stage to determine whether to scale the inputs of that stage. If all of the results in a particular stage have magnitude less than 1, no scaling is necessary at that stage. Otherwise, scale the inputs to that stage by 0.5. This conditional scaling technique achieves much better accuracy, however, at the cost of increasing software complexity.

6.4.3 MATLAB Implementations

As introduced in Section 3.2.6, MATLAB provides the function `fft` with syntax

```
y = fft(x);
```

to compute the DFT of $x(n)$ in the vector `x`. If the length of `x` is a power of 2, the `fft` function employs an efficient radix-2 FFT algorithm. Otherwise, it uses a slower mixed-radix FFT algorithm or even a DFT.

An alternative way of using `fft` function is

```
y = fft(x, N);
```

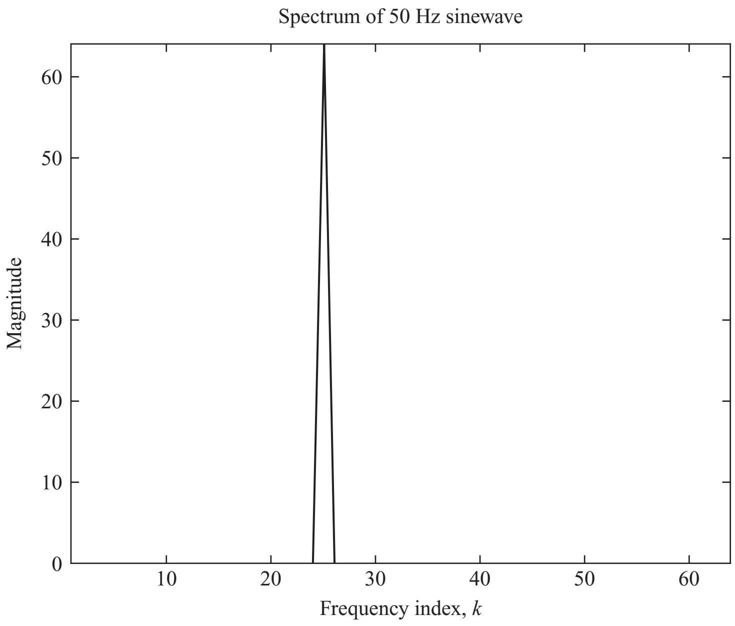


Figure 6.11 Spectrum of 50 Hz sinewave

to specify N-point FFT. If the length of `x` is less than `N`, the vector `x` is padded with trailing zeros to length `N`. If the length of `x` is greater than `N`, the `fft` function only performs the FFT of the first `N` samples.

The execution time of the `fft` function depends on the input data type and the sequence length. If the input data is real valued, it computes a real power-of-2 FFT algorithm that is faster than a complex FFT of the same length. The execution is fastest if the sequence length is exactly a power of 2. For example, if the length of `x` is 511, the function `y = fft(x, 512)` will be computed faster than `fft(x)` which performs 511-point DFT. It is important to note that the vectors in MATLAB are indexed from 1 to N instead of from 0 to $N - 1$ as given in the DFT and IDFT definitions.

Example 6.12: Consider a sinewave of frequency $f = 50\,\text{Hz}$ expressed as

$$x(n) = \sin(2\pi f n/f_s), n = 0, 1, \ldots, 127,$$

where the sampling rate $f_s = 256\,\text{Hz}$. We analyze this sinewave using a 128-point FFT given in the MATLAB script (`example6_12.m`), and display the magnitude spectrum in Figure 6.11. It shows the frequency index $k = 25$ corresponding to the spectrum peak. Substituting the associated parameters into Equation (6.22), we verified that the line spectrum is corresponding to 50 Hz.

The MATLAB function `ifft` implements the IFFT algorithm as

```
    y = ifft(x);
```

or

```
    y = ifft(x,N);
```

The characteristics and usage of `ifft` are the same as those for `fft`.

6.4.4 Fixed-Point Implementation Using MATLAB

MATLAB provides a function `qfft` for quantizing an FFT object to support fixed-point implementation. For example, the following command,

```
F = qfft
```

constructs a quantized FFT object `F` with default values. We can change the default settings by

```
F = qfft('Property1',Value1, 'Property2',Value2, ...)
```

to create a quantized FFT object with specific property/value pairs.

Example 6.13: We can change the default 16-point FFT to 128-point FFT using the following command:

```
F = qfft('length',128)
```

We then obtain the following quantized FFT object in the command window:

```
F =
             Radix = 2
            Length = 128
 CoefficientFormat = quantizer('fixed', 'round', 'saturate', [16 15])
       InputFormat = quantizer('fixed', 'floor', 'saturate', [16 15])
      OutputFormat = quantizer('fixed', 'floor', 'saturate', [16 15])
MultiplicandFormat = quantizer('fixed', 'floor', 'saturate', [16 15])
     ProductFormat = quantizer('fixed', 'floor', 'saturate', [32 30])
         SumFormat = quantizer('fixed', 'floor', 'saturate', [32 30])
  NumberOfSections = 7
       ScaleValues = [1]
```

This shows that the quantized FFT is a 128-point radix-2 FFT for the fixed-point data and arithmetic. The coefficients, input, output, and multiplicands are represented using Q15 format `[16 15]`, while the product and sum use Q30 format `[32 30]`. There are seven stages for $N = 128$, and no scaling is applied to the input at each stage by the default setting `ScaleValues = [1]`. We can set a scaling factor 0.5 at the input of each stage as follows:

```
F.ScaleValues = [0.5 0.5 0.5 0.5 0.5 0.5 0.5];
```

Or, set different values at specific stages using different scaling factors.

Example 6.14: Similar to Example 6.12, we used a quantized FFT to analyze the spectrum of sinewave. In `example6_14a.m`, we first generate the same sinewave as in Example 6.12, then use the following functions to compute the fixed-point FFT with Q15 format:

```
FXk = qfft('length',128);  % Create quantized FFT object
qXk = fft(FXk, xn);        % Compute Q15 FFT in xn vector
```

When we run the MATLAB script, we receive the following warning messages reported in MATLAB command window:

```
Warning: 1135 overflows in quantized fft.
                  Max       Min  NOverflows  NUnderflows  NOperations
 Coefficient        1        -1           7            6          254
       Input   0.9999   -0.9999           0            0          128
      Output        2        -2          16           32          256
Multiplicand        2        -2        1063           91         3584
     Product        1        -1           0            0         3584
         Sum    2.414    -2.414          56            0         4480
```

Without proper scaling, the FFT has 1135 overflows, and thus the FFT results are wrong.

We can modify the code by setting the scaling factor 0.5 at each stage as follows (see `example6_14b.m`):

```
FXk = qfft('length',128);  % Create quantized FFT object
FXk.ScaleValues = [0.5 0.5 0.5 0.5 0.5 0.5 0.5]; % Set scaling
factors
qXk = fft(FXk, xn);        % Compute Q15 FFT of xn vector
```

When we run the modified program (`example6_14b.m`), there are no warnings or errors. The spectrum plot displayed in Figure 6.12 shows that we can perform FFT properly using 16-bit processors with adequate scaling factor at each stage.

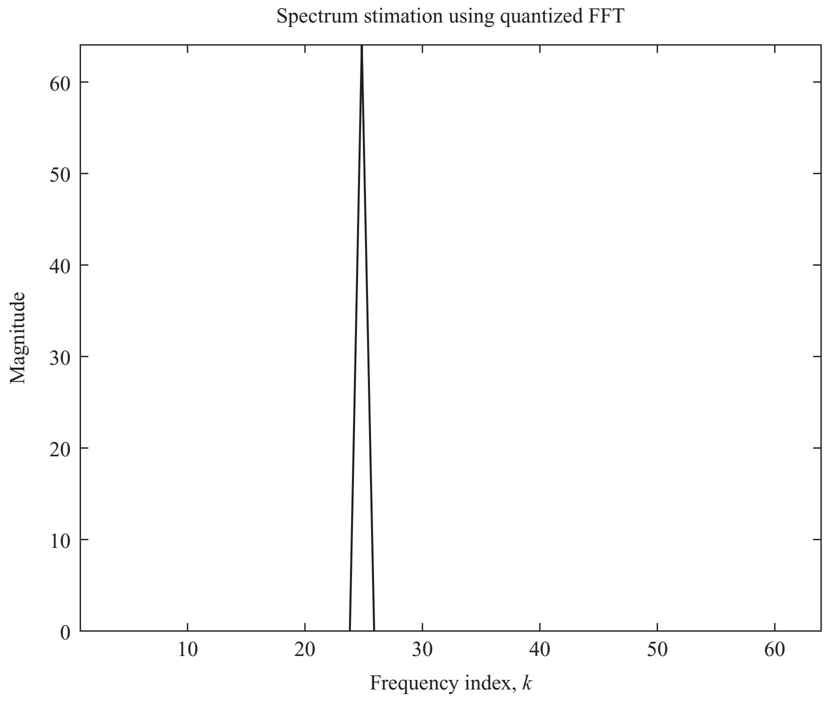


Figure 6.12 Sinewave spectrum computed using the quantized 16-bit FFT

6.5 Practical Applications

In this section, we will introduce two important FFT applications: spectral analysis and fast convolution.

6.5.1 Spectral Analysis

The inherent properties of the DFT directly affect its performance on spectral analysis. The spectrum estimated from a finite number of samples is correct only if the signal is periodic and the sample set exactly covers one or multiple period of signal. In practice, we may have to break up a long sequence into smaller segments and analyze each segment individually using the DFT.

As discussed in Section 6.2, the frequency resolution of N-point DFT is f_s/N. The DFT coefficients $X(k)$ represent frequency components that are equally spaced at frequencies f_k as defined in Equation (6.22). One cannot properly represent a signal component that falls between two adjacent samples in the spectrum, because its energy will spread to neighboring bins and distort their spectral amplitude.

Example 6.15: In Example 6.12, the frequency resolution (f_s/N) is 2 Hz using a 128-point FFT and sampling rate 256 Hz. The line component at 50 Hz can be represented by $X(k)$ at $k = 25$ as shown in Figure 6.11.

Consider the case of adding another sinewave at frequency 61 Hz (see `example6_15.m`). Figure 6.13 shows both spectral components at 50 and 61 Hz. However, the frequency component at 61 Hz (between $k = 30$ and $k = 31$) does not show a line component because its energy spreads into adjacent frequency bins.

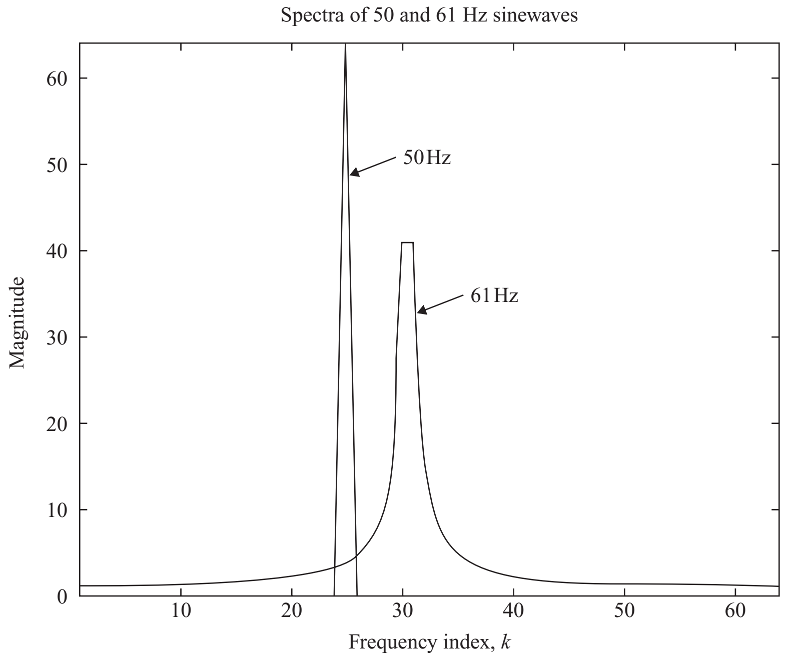


Figure 6.13 Spectra of sinewaves at 50 and 61 Hz

A solution to this spectral leakage problem is to have a finer resolution f_s/N by using a larger FFT size N. If the number of data samples is not sufficiently large, the sequence may be expanded to length N by adding zeros to the tail of true data. This process is simply the interpolation of the spectral curve between adjacent frequency components.

Other problems relate to the FFT-based spectral analysis including aliasing, finite data length, spectral leakage, and spectral smearing. These issues will be discussed in the following section.

6.5.2 Spectral Leakage and Resolution

The data set that represents the signal of finite length N can be obtained by multiplying the signal with a rectangular window expressed as

$$x_N(n) = w(n)x(n) = \begin{cases} x(n), & 0 \le n \le N-1 \\ 0, & \text{otherwise} \end{cases}, \tag{6.43}$$

where the rectangular function $w(n)$ is defined in Equation (4.33). As the length of the window increases, the windowed signal $x_N(n)$ becomes a better approximation of $x(n)$, and thus $X(k)$ becomes a better approximation of the DTFT $X(\omega)$.

The time-domain multiplication given in Equation (6.43) is equivalent to the convolution in the frequency domain. Thus, the DFT of $x_N(n)$ can be expressed as

$$X_N(k) = W(k) * X(k) = \sum_{l=-N}^{N} W(k-l)X(k), \tag{6.44}$$

where $W(k)$ is the DFT of the window function $w(n)$, and $X(k)$ is the true DFT of the signal $x(n)$. Equation (6.44) shows that the computed spectrum $X_N(k)$ consists of the true spectrum $X(k)$ convoluted with the window function's spectrum $W(k)$. Therefore, the computed spectrum of the finite-length signal is corrupted by the rectangular window's spectrum.

As discussed in Section 4.2, the magnitude response of the rectangular window consists of a mainlobe and several smaller sidelobes. The frequency components that lie under the sidelobes represent the sharp transition of $w(n)$ at the endpoints. The sidelobes introduce spurious peaks into the computed spectrum, or to cancel true peaks in the original spectrum. This phenomenon is known as spectral leakage. To avoid spectral leakage, it is necessary to use different windows as introduced in Section 4.2.3 to reduce the sidelobe effects.

Example 6.16: If the signal $x(n)$ consists of a single sinusoid $\cos(\omega_0 n)$, the spectrum of the infinite-length sampled signal is

$$X(\omega) = 2\pi\delta(\omega \pm \omega_0), \qquad -\pi \le \omega \le \pi, \tag{6.45}$$

which consists of two line components at frequencies $\pm\omega_0$. However, the spectrum of the windowed sinusoid can be obtained as

$$X_N(\omega) = \frac{1}{2}[W(\omega-\omega_0) + W(\omega+\omega_0)], \tag{6.46}$$

where $W(\omega)$ is the spectrum of the window function.

Equation (6.46) shows that the windowing process has the effect of smearing the original sharp spectral line $\delta(\omega - \omega_0)$ at frequency ω_0 and replacing it with $W(\omega - \omega_0)$. Thus, the power has been spread into the entire frequency range by the windowing operation. This undesired effect is called spectral smearing. Thus, windowing not only distorted the spectrum due to leakage effects, but also reduced spectral resolution.

Example 6.17: Consider a signal consisting of two sinusoidal components expressed as $x(n) = \cos(\omega_1 n) + \cos(\omega_2 n)$. The spectrum of the windowed signal is

$$X_N(\omega) = \frac{1}{2}[W(\omega - \omega_1) + W(\omega + \omega_1) + W(\omega - \omega_2) + W(\omega + \omega_2)], \tag{6.47}$$

which shows that the sharp spectral lines are replaced with their smeared versions. If the frequency separation, $\Delta\omega = |\omega_1 - \omega_2|$, of the two sinusoids is

$$\Delta\omega \leq \frac{2\pi}{N} \tag{6.48}$$

or

$$\Delta f \leq \frac{f_s}{N}, \tag{6.49}$$

the mainlobe of the two window functions $W(\omega - \omega_1)$ and $W(\omega - \omega_2)$ overlap. Thus, the two spectral lines in $X_N(\omega)$ are not distinguishable. MATLAB script `example6_17.m` uses 128-point FFT for signal with sampling rate 256 Hz. This example shows that two sinewaves of frequencies 60 and 61 Hz are mixed. From Equation (6.49), the frequency separation 1 Hz is less than the frequency resolution 2 Hz, thus these two sinewaves are overlapped as shown in Figure 6.14.

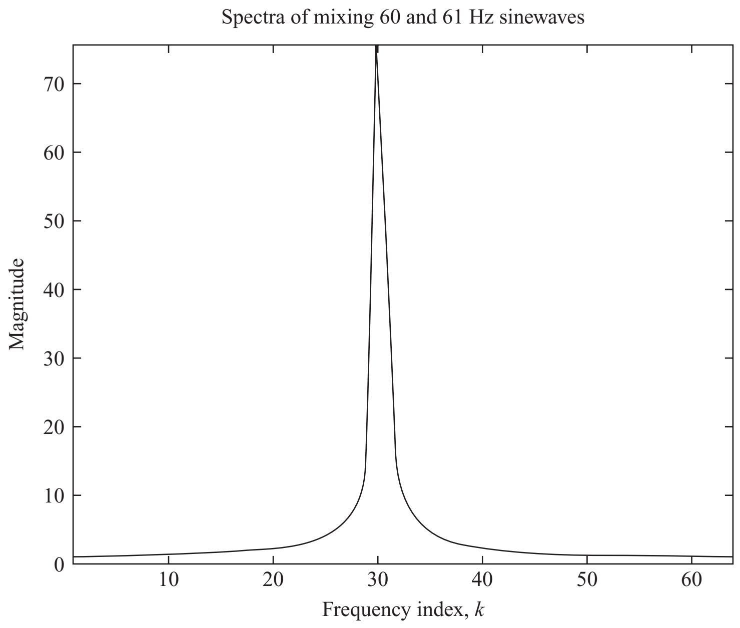


Figure 6.14 Spectra of mixing sinewaves at 60 and 61 Hz

To guarantee that two sinusoids appear as two distinct ones, their frequency separation must satisfy the condition

$$\Delta\omega > \frac{2\pi}{N} \quad \text{or} \quad \Delta f > \frac{f_s}{N}. \tag{6.50}$$

Thus, the minimum DFT length to achieve a desired frequency resolution is given as

$$N > \frac{f_s}{\Delta f} = \frac{2\pi}{\Delta\omega}. \tag{6.51}$$

In summary, the mainlobe width determines the frequency resolution of the windowed spectrum. The sidelobes determine the amount of undesired frequency leakage. The optimum window used for spectral analysis must have narrow mainlobe and small sidelobes. The amount of leakage can be substantially reduced using nonrectangular window functions introduced in Section 4.2.3 at the cost of decreased spectral resolution. For a given window length N, windows such as rectangular, Hanning, and Hamming have relatively narrow mainlobe compared with Blackman and Kaiser windows. Unfortunately, the first three windows have relatively high sidelobes, thus having more leakage. There is a trade-off between frequency resolution and spectral leakage in choosing windows for a given application.

Example 6.18: Consider the 61 Hz sinewave in Example 6.15. We can apply the Kaiser window with $N = 128$ and $\beta = 8.96$ to the signal using the following commands:

```
beta = 8.96;
wn = (kaiser(N,beta))';     % Kaiser window
x1n = xn.*wn;               % Generate windowed sinewave
```

The magnitude spectra of sinewaves with the rectangular and Kaiser windows are shown in Figure 6.15 by the MATLAB script `example6_18.m`. This shows that the Kaiser window can effectively reduce the spectral leakage. Note that the gain for using Kaiser window has been scaled up by 2.4431 in order to compensate the energy loss compared with using rectangular window. The time- and frequency-domain plots of the Kaiser window with length $N = 128$ and $\beta = 8.96$ are shown in Figure 6.16 using WinTool.

For a given window, increasing the length of the window reduces the width of the mainlobe, which leads to better frequency resolution. However, if the signal changes frequency content over time, the window cannot be too long in order to provide a meaningful spectrum.

6.5.3 Power Spectrum Density

Consider a sequence $x(n)$ of length N whose DFT is $X(k)$, the Parseval's theorem can be expressed as

$$E = \sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N}\sum_{k=0}^{N-1} |X(k)|^2. \tag{6.52}$$

The term $|X(k)|^2$ is called the power spectrum that measures the power of signal at frequency f_k. Therefore, squaring the DFT magnitude spectrum $|X(k)|$ produces a power spectrum, which is also called the periodogram.

The power spectrum density (PSD) (power density spectrum or simply power spectrum) characterizes stationary random processes. The PSD is very useful in the analysis of random signals since it provides

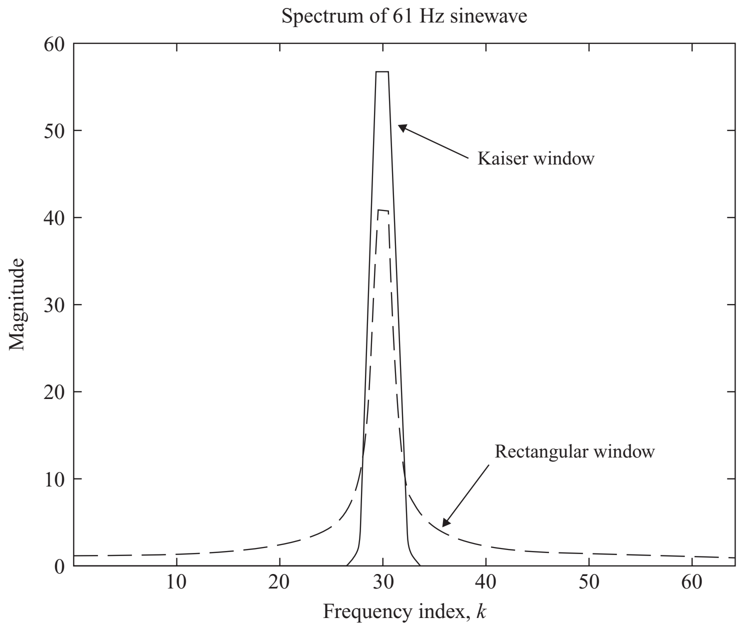


Figure 6.15 Spectra obtained using rectangular and Kaiser windows

a meaningful measure for the distribution of the average power in such signals. There are different techniques for estimating the PSD. Since the periodogram is not a consistent estimate of the true PSD, the averaging method can reduce statistical variation of the computed spectra.

One way of computing the PSD is to decompose $x(n)$ into M segments, $x_m(n)$, of N samples each. These signal segments are spaced $N/2$ samples apart; i.e., there is 50 % overlap between successive segments. In order to reduce spectral leakage, each $x_m(n)$ is multiplied by a nonrectangular window

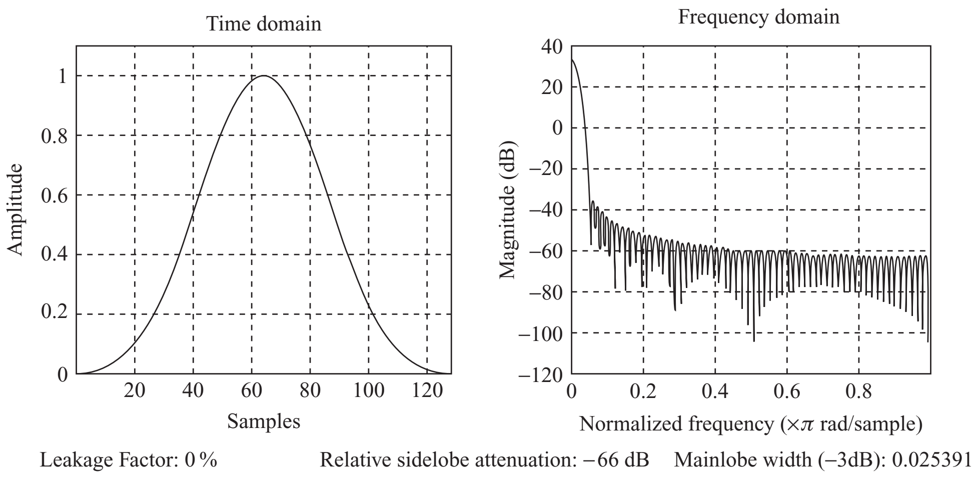


Figure 6.16 Kaiser window of $N = 128$ and $\beta = 8.96$

(such as Hamming) function $w(n)$ of length N. The PSD is a weighted sum of the periodograms of the individual overlapped segment.

The MATLAB *Signal Processing Toolbox* provides the function `psd` to estimate the PSD of the signal given in the vector `x` using the following statement:

```
h = spectrum.periodogram;  % Create a periodogram object
psd(h,x,'Fs',Fs);          % Plots the two-sided PSD by default
```

where Fs is the sampling frequency.

Example 6.19: Consider the signal $x(n)$ which consists of two sinusoids (140 and 150 Hz) and noise. This noisy signal is generated by `example6_19.m` adapted from the MATLAB **Help** menu. The PSD can be computed by creating the following periodogram object:

```
Hs=spectrum.periodogram;
```

The `psd` function can also display the PSD (Figure 6.17) as follows:

```
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```

Note that we can specify the window used for computing PSD. For example, we can use Hamming window as follows:

```
Hs = spectrum.periodogram('Hamming');
```

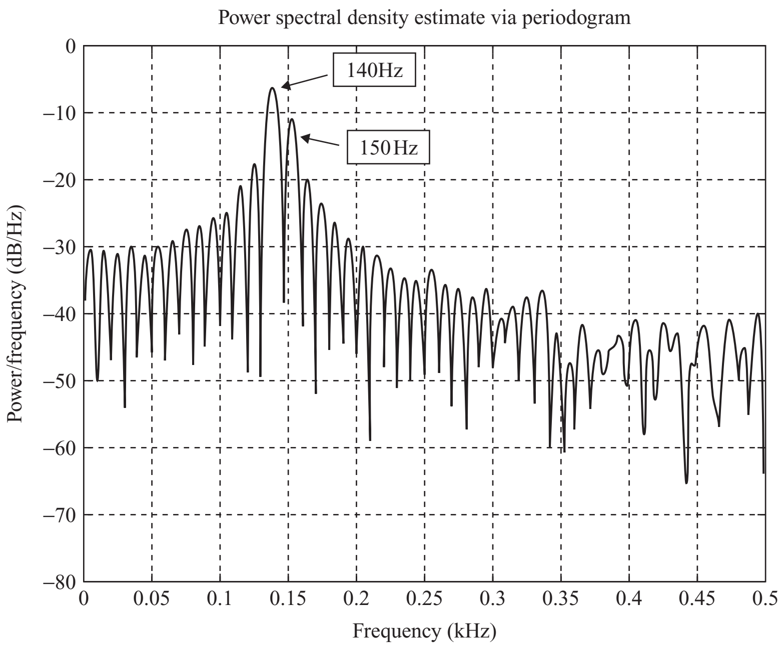


Figure 6.17 PSD of two sinewaves embedded in noise

For a time-varying signal, it is more meaningful to compute a local spectrum that measures spectral contents over a short-time interval. We use a sliding window to break up a long sequence into several short blocks $x'_m(n)$ of N samples, and then perform the FFT to obtain the time-dependent frequency spectrum at each segment m as follows:

$$X_m(k) = \sum_{n=0}^{N-1} x'_m(n) W_N^{kn}, \qquad k = 0, 1, \ldots, N-1. \tag{6.53}$$

This process is repeated for the next block of N samples. This technique is called the short-term Fourier transform, since $X_m(k)$ is just the spectrum of the short segment of $x_m(n)$ that lies inside the sliding window $w(n)$. This form of time-dependent Fourier transform has several applications in speech, sonar, and radar signal processing.

Equation (6.53) shows that $X_m(k)$ is a two-dimensional sequence. The index k represents frequency, and the block index m represents segment (or time). Since the result is a function of both time and frequency, a three-dimensional graphical display is needed. This is done by plotting $|X_m(k)|$ using gray-scale (or color) images as a function of both k and m. The resulting three-dimensional graphic is called the spectrogram. The spectrogram uses the x-axis to represent time and the y-axis to represent frequency. The gray level (or color) at point (m, k) is proportional to $|X_m(k)|$.

The *Signal Processing Toolbox* provides a function `spectrogram` to compute spectrogram. This MATLAB function has the form

```
B = spectrogram(a,window,noverlap,nfft,Fs);
```

where `B` is a matrix containing the complex spectrogram values $|X_m(k)|$, and other arguments are defined in the function `psd`. More overlapped samples make the spectrum move smoother from block to block. It is common to pick the overlap to be around 25 %. The `spectrogram` function with no output arguments displays the scaled logarithm of the spectrogram in the current graphic window.

Example 6.20: The MATLAB program `example6_20.m` loads the speech file `timit2.asc`, plays it using the function `soundsc`, and displays the spectrogram as shown in Figure 6.18. The color bar on the right side indicates the signal strength in dB. The color corresponding to the lower power in the figure represents the silence and the color corresponding to the higher power represents the speech signals.

6.5.4 Fast Convolution

As discussed in Chapter 4, FIR filtering is a linear convolution of filter impulse response $h(n)$ with the input sequence $x(n)$. If the FIR filter has L coefficients, we need L real multiplications and $L-1$ real additions to compute each output $y(n)$. To obtain L output samples, the number of operations (multiplication and addition) needed is proportional to L^2. To take advantage of efficient FFT and IFFT algorithms, we can use the fast convolution algorithm illustrated in Figure 6.19 for FIR filtering. Fast convolution provides a significant reduction in computational requirements for higher order FIR filters, thus it is often used to implement FIR filtering in applications having a large number of data samples.

It is important to note that the fast convolution shown in Figure 6.19 produces the circular convolution discussed in Section 6.2.3. In order to produce a linear convolution, it is necessary to append zeros to the signals as shown in Example 6.10. If the data sequence $x(n)$ has finite duration M, the first step is to pad data sequence and coefficients with zeros to a length corresponding to an allowable FFT size N

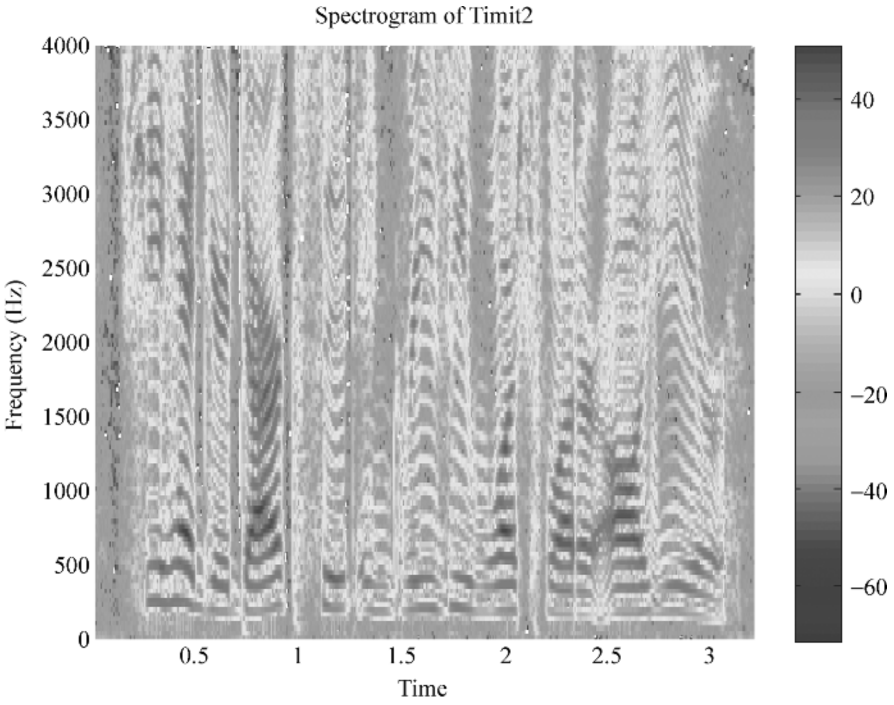


Figure 6.18 Spectrogram of speech signal

($\geq L + M - 1$), where L is the length of $h(n)$. The FFT is computed for both sequences to obtain $X(k)$ and $H(k)$, the corresponding complex products $Y(k) = X(k)H(k)$ are calculated, and the IFFT of $Y(k)$ is used to obtain $y(n)$. The desired linear convolution is contained in the first $L + M - 1$ terms of these results. Since the filter impulse response $h(n)$ is known as *a priori*, the FFT of $h(n)$ can be precalculated and stored as fixed coefficients.

For many applications, the input sequence is very long as compared to the FIR filter length L. This is especially true in real-time applications, such as in audio signal processing where the FIR filter order is extremely high due to high-sampling rate and input data is very long. In order to use the efficient FFT and IFFT algorithms, the input sequence must be partitioned into segments of N ($N > L$ and N is a size supported by the FFT algorithm) samples, process each segment using the FFT, and finally assemble the output sequence from the outputs of each segment. This procedure is called the block-processing operation. The cost of using this efficient block processing is the buffering delay. More complicated

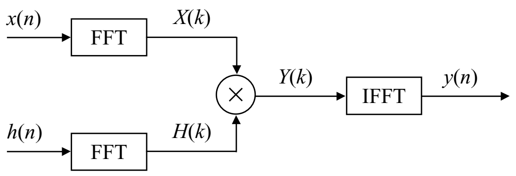


Figure 6.19 Fast convolution algorithm

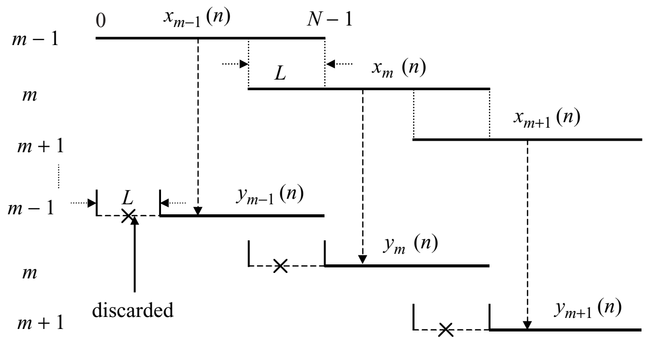


Figure 6.20 Overlap data segments for the overlap-save technique

algorithms have been devised to have both zero latency as direct FIR filtering and computational efficiency [10].

There are two techniques for the segmentation and recombination of the data: the overlap-save and overlap-add algorithms.

Overlap-Save Technique

The overlap-save process overlaps Linput samples on each segment. The output segments are truncated to be nonoverlapping and then concatenated. The following steps describe the process illustrated in Figure 6.20:

1. Apply N-point FFT to the expanded (zero-padded) impulse response sequence to obtain $H'(k)$, where $k = 0, 1, \ldots, N - 1$. This process can be precalculated off-line and stored in memory.

2. Select N signal samples $x_m(n)$ (where m is the segment index) from the input sequence $x(n)$ based on the overlap illustrated in Figure 6.20, and then use N-point FFT to obtain $X_m(k)$.

3. Multiply the stored $H'(k)$ (obtained in Step 1) by the $X_m(k)$ (obtained in Step 2) to get

$$Y_m(k) = H'(k)X_m(k), \qquad k = 0, 1, \ldots, N - 1. \tag{6.54}$$

4. Perform N-point IFFT of $Y_m(k)$ to obtain $y_m(n)$ for $n = 0, 1, \ldots, N - 1$.

5. Discard the first L samples from each IFFT output. The resulting segments of $(N - L)$ samples are concatenated to produce $y(n)$.

Overlap-Add Technique

The overlap-add process divides the input sequence $x(n)$ into nonoverlapping segments of length $(N - L)$. Each segment is zero-padded to produce $x_m(n)$ of length N. Follow the Steps 2–4 of the overlap-save method to obtain N-point segment $y_m(n)$. Since the convolution is the linear operation, the output sequence $y(n)$ is the summation of all segments.

MATLAB implements this efficient FIR filtering using the overlap-add technique as

```
y = fftfilt(b, x);
```

The `fftfilt` function filters the input signal in the vector `x` with the FIR filter described by the coefficient vector `b`. The function chooses an FFT and a data block length that automatically guarantees efficient execution time. However, we can specify the FFT length `N` by using

```
y = fftfilt(b, x, N)
```

Example 6.21: The speech data `timit2.asc` (used in Example 6.20) is corrupted by a tonal noise at frequency 1000 Hz. We design a bandstop FIR filter with edge frequencies of 900 and 1100 Hz, and filter the noisy speech using the following MATLAB script (`example6_21.m`):

```
Wn = [900  1100]/4000;      % Edge frequencies
b = fir1(128, Wn, 'stop'); % Design bandstop filter
yn = fftfilt(b, xn);        % FIR filtering using fast convolution
soundsc(yn, fs, 16);        % Listen to the filtered signal
spectrogram(yn,kaiser(256,5),200,256,fs,'yaxis') % Spectrogram
```

MATLAB program `example6_21.m` plays the original speech first, plays the noisy speech that is corrupted by the 1 kHz tone, and then shows the spectrogram with the noise component (in red) at 1000 Hz. In order to attenuate that tonal noise, a bandstop FIR filter is designed (using the function `fir1`) to filter the noisy speech using the function `fftfilt`. Finally, the filter output is played and its spectrogram is shown in Figure 6.21 with the 1000 Hz tonal noise being attenuated.

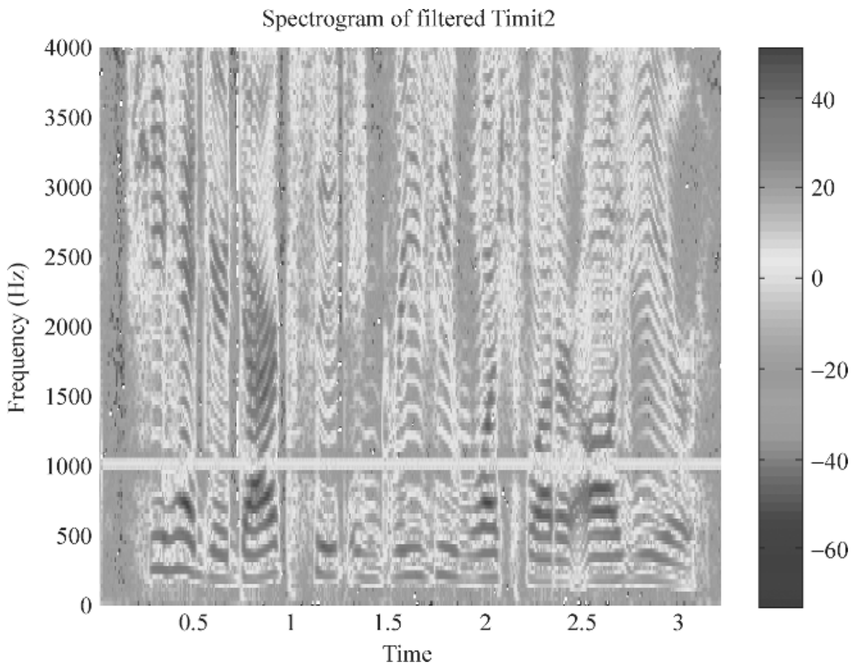


Figure 6.21 Spectrogram of bandstop filter output

6.6 Experiments and Program Examples

In the section, we will implement the DFT and FFT algorithms for DSP applications. The computation of the DFT and FFT involves nested loops, complex multiplication, and complex twiddle-factor generation.

6.6.1 Floating-Point C Implementation of DFT

For multiplying a complex data sample $x(n) = x_r(n) + jx_i(n)$ and a complex twiddle factor $W_N^{kn} = \cos(2\pi kn/N) - j\sin(2\pi kn/N) = W_r - jW_i$ defined in Equation (6.15), the product can be expressed as

$$x(n)W_N^{kn} = x_r(n)W_r + x_i(n)W_i + j[x_i(n)W_r - x_r(n)W_i], \tag{6.55}$$

where the subscripts r and i denote the real and imaginary parts of complex variable. Equation (6.55) can be rewritten as

$$X(n) = X_r(n) + jX_i(n), \tag{6.56}$$

where

$$X_r(n) = x_r(n)W_r + x_i(n)W_i \tag{6.57a}$$

$$X_i(n) = x_i(n)W_r - x_r(n)W_i. \tag{6.57b}$$

The C program listed in Table 6.2 uses two arrays, `Xin[2*N]` and `Xout[2*N]`, to hold the complex input and output samples. The twiddle factors are computed at run time. Since most of real-world applications contain only real data, it is necessary to compose a complex data set from the given real data. The simplest way is to zero out the imaginary part before calling the DFT function.

This experiment computes 128-point DFT of signal given in file `input.dat`, and displays the spectrum using the CCS graphics. Table 6.3 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Open the project file, `float_dft128.pjt`, and rebuild the project.

2. Run the DFT experiment using the input data file `input.dat`.

3. Examine the results saved in the data array `spectrum[ ]` using CCS graphics as shown in Figure 6.22. The magnitude spectrum shows the normalized frequencies at 0.125, 0.25, and 0.5.

4. Profile the code and record the required cycles per data sample for the floating-point implementation of DFT.

6.6.2 C55x Assembly Implementation of DFT

We write assembly routines based on the C program listed in Table 6.4 to implement DFT on TMS320C55x. The sine and cosine generators for experiments given in Chapter 3 can be used to generate the twiddle factors. The assembly function `sine_cos.asm` (see section 'Practical Applications' in Chapter 3) is a C-callable function that follows the C55x C-calling convention. This function has two arguments: `angle` and `Wn`. The first argument contains the input `angle` in radians and is passed to the

Table 6.2 List of floating-point C function for DFT

```
#include <math.h>
#define PI 3.1415926536
void floating_point_dft(float Xin[], float Xout[])
{
  short i,n,k,j;
  float angle;
  float Xr[N],Xi[N];
  float W[2];
  for (i=0,k=0;k<N;k++)
  {
      Xr[k]=0;
      Xi[k]=0;
      for(j=0,n=0;n<N;n++)
      {
        angle =(2.0*PI*k*n)/N;
        W[0]=cos(angle);
        W[1]=sin(angle);
        Xr[k]=Xr[k]+Xin[j]*W[0]+Xin[j+1]*W[1];
        Xi[k]=Xi[k]+Xin[j+1]*W[0]-Xin[j]*W[1];
        j+=2;
      }
      Xout[i++] = Xr[k];
      Xout[i++] = Xi[k];
   }
}
```

C55x assembly routine via the temporary register T0. The second argument is the pointer to `Wn` passed by the auxiliary register AR0, for which the computed results will be stored upon return.

The calculation of the angle depends on two variables, k and n, as follows:

$$\text{Angle} = (2\pi/N)kn. \tag{6.58}$$

As shown in Figure 3.30, the fixed-point representation of value π for sine–cosine generator is 0x7FFF. Thus, the angle used to generate the twiddle factors can be expressed as

$$\text{Angle} = (2 \cdot 0x7\text{FFF}/N)kn \tag{6.59}$$

Table 6.3 File listing for experiment `exp6.6.1_floatingPoint_DFT`

Files	Description
`float_dft128Test.c`	C function for testing floating-point DFT experiment
`float_dft128.c`	C function for 128-point floating-point DFT algorithm
`float_mag128.c`	C function computes magnitude of 128 DFT results
`float_dft128.h`	C header file for DFT experiment
`float_dft128.pjt`	DSP project file
`float_dft128.cmd`	DSP linker command file
`input.dat`	Data file

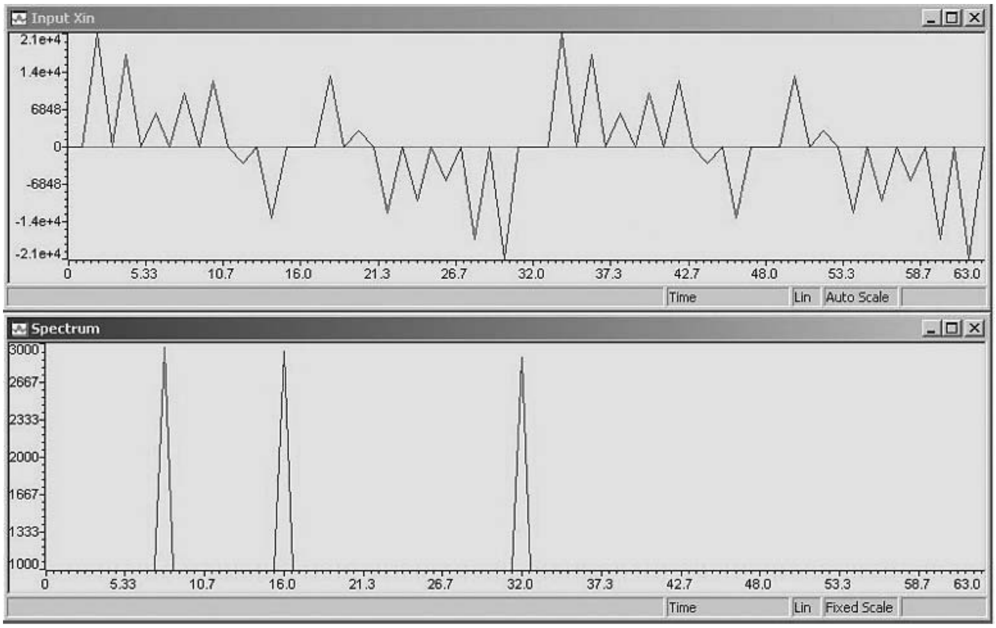


Figure 6.22 Input signal (top) and the magnitude spectrum (bottom)

Table 6.4 List of C55x assembly implementation of DFT

```
;  DFT_128 - 128-point DFT routine
;
;  Entry T0: AR0: pointer to complex input buffer
;            AR1: pointer to complex output buffer
;  Return: None
;
        .def  dft_128
        .ref  sine_cos
N       .set  128
TWOPIN  .set  0x7fff>>6       ; 2*PI/N, N=128
        .bss  Wn,2            ; Wn[0]=Wr, Wn[1]=Wi
        .bss  angle,1         ; Angle for sine-cosine function
        .text
_dft_128
    pshboth XAR5              ; Save AR5
    bset    SATD
    mov     #N-1,BRC0         ; Repeat counter for outer loop
    mov     #N-1,BRC1         ; Repeat counter for inner loop
    mov     XAR0,XAR5         ; AR5 pointer to sample buffer
    mov     XAR0,XAR3
    mov     #0,T2             ; k = T2 = 0
    rptb    outer_loop-1      ; for(k=0;k<N;k++) {
    mov     XAR3,XAR5         ; Reset x[] pointer
    mov     #TWOPIN<<#16,AC0  ; hi(AC0) = 2*PI/N
    mpy     T2,AC0
```

Table 6.4 (*continued*)

```
    mov     #0,AC2            ; Xr[k] = 0
    mov     #0,AC3            ; Xi[k] = 0
    mov     #0,*(angle)
    mov     AC0,T3            ; angle=2*PI*k/N
    rptb    inner_loop-1      ; for(n=0;n<N;n++) {
    mov     *(angle),T0       ; T0=2*PI*k*n/N
    mov     *(angle),AC0
    add     T3,AC0
    mov     AC0,*(angle)      ; Update angle
    amov    #Wn,XAR0          ; AR0 is the pointer to Wn
    call    _sine_cos         ; sine_cos(angle, Wn)
    bset    SATD              ; sine_cos turn off FRCT & SATD
    macm40  *AR5+,*AR0,AC2    ; XR[k] + Xin[n]*Wr
    macm40  *AR5-,*AR0+,AC3   ; XI[k] + Xin[n+1]*Wr
    masm40  *AR5+,*AR0,AC3    ; XI[k] + Xin[n+1]*Wr - Xin[n]*Wi
    macm40  *AR5+,*AR0-,AC2   ; XR[k] + Xin[n]*Wr + Xin[n+1]*Wi
inner_loop                    ; End of inner loop
    mov     hi(AC2<<#-5),*AR1+
    mov     hi(AC3<<#-5),*AR1+
    add     #1,T2
outer_loop                    ; End of outer loop
    popboth XAR5
    bclr    SATD
    ret
```

for $n = 0, 1, \ldots, N-1$ and $k = 0, 1, \ldots, N-1$. The C55x assembly routine listed in Table 6.4 computes 128-point DFT. This assembly program calls the assembly routine `sine_cos` to compute the twiddle factors.

Table 6.5 lists the files used for the C55x assembly DFT experiment. This experiment uses C55x assembly program to compute 128-point DFT and magnitude spectrum. The zero-overhead loops allow program to initialize the loop counters `BRC0` and `BRC1` outside the nested loops. When implementing nested loops, the inner loop uses `BRC1` while the outer loop uses `BRC0`.

Procedures of the experiments are listed as follows:

1. Open the project file, `asm_dft128.pjt`, and rebuild the project.

2. Run the DFT project using the input data file `input.dat`.

Table 6.5 File listing for experiment `exp6.6.2_asm_DFT`

Files	Description
`asm_dft128Test.c`	C function for testing DFT experiment
`dft_128.asm`	Assembly function for 128-point DFT
`mag_128.asm`	Assembly function computes magnitude spectrum
`sine_cos.asm`	Assembly function computes twiddle factors
`asm_dft128.h`	C header file for DFT experiment
`asm_dft128.pjt`	DSP project file
`asm_dft128.cmd`	DSP linker command file
`input.dat`	Data file

3. Examine the results stored in `spectrum[ ]` using CCS graphics. It should have the same peaks in Figure 6.22 with the normalized frequencies at 0.125, 0.25, and 0.5.

4. Profile the C55x assembly language implementation of the DFT and compare the required cycles per data sample with the floating-point C experiment result obtained in previous experiment.

6.6.3 Floating-Point C Implementation of FFT

This experiment implements the complex, radix-2, decimation-in-time FFT algorithm using floating-point C. The floating-point FFT function is listed in Table 6.6. The first argument is a pointer to the complex data array. The second argument passes the number of exponential values of the radix-2 FFT. The third argument is the pointer to the twiddle factors. The last argument is a flag used to determine if a scale is needed during the computation of the FFT algorithm. As explained in Section 6.4.2, the input data is scaled by 0.5 at each FFT stage. The twiddle factors are computed at the end of the function to reduce computational requirement. This FFT routine can be used for inverse FFT with the scale factors set to 1.0.

Table 6.6 List of floating-point C FFT function

```c
void fft(complex *X, unsigned short EXP, complex *W, unsigned short SCALE)
{
    complex  temp;          /* Temporary storage of complex variable */
    complex  U;             /* Twiddle factor W^k */
    unsigned short i,j;
    unsigned short id;      /* Index for lower point in butterfly */
    unsigned short N=1<<EXP;/* Number of points for FFT */
    unsigned short L;       /* FFT stage */
    unsigned short LE;      /* Number of points in sub DFT at stage L
                                and offset to next DFT in stage */
    unsigned short LE1;     /* Number of butterflies in one DFT at
                                stage L.  Also is offset to lower point
                                in butterfly at stage L */
    float scale;
    scale = 0.5;
    if (SCALE == 0)
        scale = 1.0;
    for (L=1; L<=EXP; L++)  /* FFT butterfly */
    {
        LE=1<<L;            /* LE=2^L=points of sub DFT */
        LE1=LE>>1;          /* Number of butterflies in sub-DFT */
        U.re = 1.0;
        U.im = 0.;
        for (j=0; j<LE1;j++)
        {
            for(i=j; i<N; i+=LE) /* Do the butterflies */
            {
                id=i+LE1;
                temp.re = (X[id].re*U.re - X[id].im*U.im)*scale;
                temp.im = (X[id].im*U.re + X[id].re*U.im)*scale;
```

Table 6.6 *(continued)*

```
                X[id].re = X[i].re*scale - temp.re;
                X[id].im = X[i].im*scale - temp.im;
                X[i].re = X[i].re*scale + temp.re;
                X[i].im = X[i].im*scale + temp.im;
            }
            /* Recursive compute W^k as U*W^(k-1) */
            temp.re = U.re*W[L-1].re - U.im*W[L-1].im;
            U.im = U.re*W[L-1].im + U.im*W[L-1].re;
            U.re = temp.re;
        }
    }
}
```

This experiment also uses the bit-reversal function listed in Table 6.7. This function rearranges the order of data samples according to the bit-reversal definition in Table 6.1 before the data sample is passed to the FFT function.

This experiment computes 128-point FFT using floating-point C and displays the magnitude spectrum. The files used for this experiment are listed in Table 6.8.

Procedures of the experiment are listed as follows:

1. Open the project file, `float_fft.pjt`, and rebuild the project.

2. Run the FFT experiment using the input data file `input_f.dat`.

Table 6.7 List of bit-reversal function in C

```
void bit_rev(complex *X, short EXP)
{
    unsigned short i,j,k;
    unsigned short N=1<<EXP; /* Number of points for FFT */
    unsigned short N2=N>>1;
    complex  temp; /* Temporary storage of the complex variable */

    for (j=0,i=1;i<N-1;i++)
    {
        k=N2;
        while(k<=j)
        {
            j-=k;
            k>>=1;
        }
        j+=k;
        if (i<j)
        {
          temp = X[j];
          X[j] = X[i];
          X[i] = temp;
        }
    }
}
```

Table 6.8 File listing for experiment `exp6.6.3_floatingPoint_FFT`

Files	Description
`float_fftTest.c`	C function for testing floating-point FFT experiment
`fft_float.c`	C function for floating-point FFT
`fbit_rev.c`	C function performs bit reversal
`float_fft.h`	C header file for floating-point FFT experiment
`fcomplex.h`	C header file defines floating-point complex data type
`float_fft.pjt`	DSP project file
`float_fft.cmd`	DSP linker command file
`input_f.dat`	Data file

3. Examine the results saved in `spectrum[ ]` using CCS graphics. The spectrum plot shows the normalized line frequency at 0.25.

4. Profile the FFT function and record the required cycles per data sample using the floating-point implementation.

6.6.4 C55x Intrinsics Implementation of FFT

In this experiment, we modify the floating-point C implementation of the FFT with the C55x intrinsics (see Table 5.4). We will replace the arithmetic operations with intrinsics `_lsmpy`, `_smas`, `_smac`, `_sadd`, and `_ssub` to implement the fixed-point FFT.

The FFT program that uses intrinsics is listed in Table 6.9. This experiment computes 128-point FFT and displays the magnitude spectrum. The program is a mix of fixed-point C and intrinsics. Its run-time performance is greatly improved over the floating-point C implementation. Table 6.10 lists the files used for this experiment.

Table 6.9 Code segment of intrinsics implementation of FFT

```
for (L=1; L<=EXP; L++)        /* FFT butterfly */
{
    LE=1<<L;                  /* LE=2^L=points of sub DFT */
    LE1=LE>>1;                /* Number of butterflies in sub DFT */
    U.re = 0x7fff;
    U.im = 0;

    for (j=0; j<LE1;j++)
    {
        for(i=j; i<N; i+=LE) /* Do the butterflies */
        {
            id=i+LE1;
            ltemp.re = _lsmpy(X[id].re, U.re);
            temp.re = (_smas(ltemp.re, X[id].im, U.im)>>SFT16);
            temp.re = _sadd(temp.re, 1)>>scale; /* Rounding & scale */
            ltemp.im = _lsmpy(X[id].im, U.re);
            temp.im = (_smac(ltemp.im, X[id].re, U.im)>>SFT16);
```

Table 6.9 (*continued*)

```
                temp.im = _sadd(temp.im, 1)>>scale; /* Rounding & scale */
                X[id].re = _ssub(X[i].re>>scale, temp.re);
                X[id].im = _ssub(X[i].im>>scale, temp.im);
                X[i].re = _sadd(X[i].re>>scale, temp.re);
                X[i].im = _sadd(X[i].im>>scale, temp.im);
            }
            /* Recursive compute W^k as W*W^(k-1) */
            ltemp.re = _lsmpy(U.re, W[L-1].re);
            ltemp.re = _smas(ltemp.re, U.im, W[L-1].im);
            ltemp.im = _lsmpy(U.re, W[L-1].im);
            ltemp.im = _smac(ltemp.im, U.im, W[L-1].re);
            U.re = ltemp.re>>SFT16;
            U.im = ltemp.im>>SFT16;
        }
    }
```

Procedures of the experiment are listed as follows:

1. Open the project file, `intrinsic_fft.pjt`, and rebuild the project.
2. Run the FFT experiment using the data file `input_i.dat`.
3. Examine the results saved in `spectrum[ ]` using CCS graphics. The spectrum plot shows the normalized line frequency at 0.25.
4. Profile the intrinsics implementation of the FFT and compare the required cycles per data sample with the floating-point C FFT experiment result obtained in previous section.

6.6.5 Assembly Implementation of FFT and Inverse FFT

In this experiment, we use the C55x assembly routines for computing the same radix-2 FFT algorithm implemented by the fixed-point C with intrinsics given in the previous experiment. The C55x FFT assembly routine listed in Table 6.11 follows the C55x C-calling convention. For readability, the assembly code mimics the C function closely. It optimizes the memory usage but not the run-time efficiency. The execution speed can be further improved by unrolling the loop and taking advantage of the FFT butterfly characteristics, but with the expense of the memory.

Table 6.10 File listing for experiment `exp6.6.4_intrinsics_FFT`

Files	Description
`intrinsic_fftTest.c`	C function for testing intrinsics FFT experiment
`intrinsic_fft.c`	C function for intrinsics FFT
`ibit_rev.c`	C function performs fixed-point bit reversal
`intrinsic_fft.h`	C header file for fixed-point FFT experiment
`icomplex.h`	C header file defines fixed-point complex data type
`intrinsic_fft.pjt`	DSP project file
`intrinsic_fft.cmd`	DSP linker command file
`input_i.dat`	Data file

The assembly routine defines local variables as a structure using the stack-relative addressing mode. The last memory location contains the return address of the caller function. Since the status registers ST1 and ST3 will be modified by the assembly routine, we use two stack locations to store the contents of these registers at entry, and they will be restored upon returning to the caller function. The complex temporary variable is stored in two consecutive memory locations by using a bracket with the numerical number to indicate the number of memory locations for the integer data type.

Table 6.11 List of C55x assembly implementation of FFT algorithm

```
    .global   _fft
ARGS    .set    0                ; Number of variables passed via stack

FFT_var .struct                  ; Define local variable structure
d_temp      .short (2)           ; Temporary variables (Re, Im)
d_L         .short
d_N         .short
d_T2        .short               ; Used to save content of T2
d_ST1       .short               ; Used to save content of ST1
d_ST3       .short               ; Used to save content of ST3
d_AR5       .short               ; Used to save content of ar5
dummy      .short                ; Used to align stack pointer
return_addr .short               ; Space for routine return address
Size        .endstruct
fft         .set 0
fft         .tag FFT_var

    .sect ".text:fft_code"
_fft:
    aadd #(ARGS-Size+1),SP       ; Adjust stack for local variables
    mov  mmap(ST1_55),AR2        ; Save ST1,ST3
    mov  mmap(ST3_55),AR3
    mov  AR2,fft.d_ST1
    mov  AR3,fft.d_ST3
    mov  AR5,(fft.d_AR5)         ; Protect AR5
    btst @#0,T1,TC1              ; Check SCALE flag set
    mov  #0x6340,mmap(ST1_55)    ; Set CPL,XF,SATD,SXAM,FRCT (SCALE=1)
    mov  #0x1f22,mmap(ST3_55)    ; Set: HINT,SATA,SMUL
    xcc  do_scale,TC1
    mov  #0x6300,mmap(ST1_55)    ; Set CPL,XF,SATD,SXAM (SCALE=2)
do_scale
    mov  T2,fft.d_T2             ; Save T2
||  mov  #1,AC0
    mov  AC0,fft.d_L             ; Initialize L=1
||  sfts AC0,T0                  ; T0=EXP
    mov  AC0,fft.d_N             ; N=1<<EXP
    mov  XAR1,XCDP               ; CDP = pointer to U[]
    mov  XSP,XAR4
    add  #fft.d_temp,AR4         ; AR4 = pointer to temp
    mov  XAR0,XAR1               ; AR1 points to sample buffer
    mov  T0,T1
    mov  XAR0,XAR5               ; Copy extended bits to XAR5
outer_loop                       ; for (L=1; L<=EXP; L++)
    mov  fft.d_L,T0              ; Note: Since the buffer is
||  mov  #2,AC0                  ;       arranged in re,im pairs
```

Table 6.11 (*continued*)

```
    sfts AC0,T0                 ;      the index to the buffer
    neg  T0                     ;      is doubled
||  mov  fft.d_N,AC1            ;      But the repeat counters
    sftl AC1,T0                 ;      are not doubled
    mov  AC0,T0                 ; LE=2<<L
||  sfts AC0,#-1
    mov  AC0,AR0                ; LE1=LE>>1
||  sfts AC0,#-1
    sub  #1,AC0                 ; Init mid_loop counter
    mov  mmap(AC0L),BRC0        ;   BRC0=LE1-1
    sub  #1,AC1                 ; Initialize inner loop counter
    mov  mmap(AC1L),BRC1        ;   BRC1=(N>>L)-1
    add  AR1,AR0
    mov  #0,T2                  ; j=0
||  rptblocal mid_loop-1        ; for (j=0; j<LE1;j++)
    mov  T2,AR5                 ; AR5=id=i+LE1
    mov  T2,AR3
    add  AR0,AR5                ; AR5 = pointer to X[id].re
    add  #1,AR5,AR2             ; AR2 = pointer to X[id].im
    add  AR1,AR3                ; AR3 = pointer to X[i].re
||  rptblocal inner_loop-1      ; for(i=j; i<N; i+=LE)
    mpy  *AR5+,*CDP+,AC0        ; AC0=(X[id].re*U.re
::  mpy  *AR2-,*CDP+,AC1        ;     -X[id].im*U.im)/SCALE
    masr *AR5-,*CDP-,AC0        ; AC1=(X[id].im*U.re
::  macr *AR2+,*CDP-,AC1        ;     +X[id].re*U.im)/SCALE
    mov  pair(hi(AC0)),dbl(*AR4); AC0H=temp.re AC1H=temp.im
||  mov  dbl(*AR3),AC2
    xcc  scale,TC1
||  mov  AC2>>#1,dual(*AR3)     ; Scale X[i] by 1/SCALE
    mov  dbl(*AR3),AC2
scale
    add  T0,AR2
||  sub  dual(*AR4),AC2,AC1     ; X[id].re=X[i].re/SCALE-temp.re
    mov  AC1,dbl(*(AR5+T0))     ; X[id].im=X[i].im/SCALE-temp.im
||  add  dual(*AR4),AC2         ; X[i].re=X[i].re/SCALE+temp.re
    mov  AC2,dbl(*(AR3+T0))     ; X[i].im=X[i].im/SCALE+temp.im
inner_loop                      ; End of inner loop
    amar *CDP+
    amar *CDP+                  ; Update k for pointer to U[k]
||  add  #2,T2                  ; Update j
mid_loop                        ; End of mid-loop
    sub  #1,T1
    add  #1,fft.d_L             ; Update L
    bcc  outer_loop,T1>0        ; End of outer loop
    mov  fft.d_ST1,AR2          ; Restore ST1,ST3,T2
    mov  fft.d_ST3,AR3
    mov  AR2,mmap(ST1_55)
    mov  AR3,mmap(ST3_55)
    mov  (fft.d_AR5),AR5
    mov  fft.d_T2,T2
    aadd #(Size-ARGS-1),SP      ; Reset SP
    ret
```

We also write the bit-reversal function using C55x assembly language for improving run-time efficiency. Table 6.12 lists the assembly implementation of bit-reversal function. To reduce the computation of the FFT algorithm, we precalculate the twiddle factors using C function `w_table.c` during the setup process. In order to use the same FFT routine for the IFFT calculation, two simple changes are made. First, the conjugating twiddle factors imply the sign change of the imaginary portion of the complex samples; that is, `X[i].im = -X[i].im`. Second, the normalization of $1/N$ is handled in the FFT routine by setting the scale flag to zero.

Table 6.12 List of assembly implementation of bit-reversal function

```
.global   _bit_rev
     .sect    ".text:fft_code"
_bit_rev
    psh  mmap(ST2_55)         ; Save ST2
    bclr ARMS                 ; Reset ARMS bit
    mov  #1,AC0
    sfts AC0,T0               ; T0=EXP, AC0=N=2EXP
    mov  AC0,T0               ; T0=N
    mov  T0,T1
    add  T0,T1
    mov  mmap(T1),BK03        ; Circular buffer size=2N
    mov  mmap(AR0),BSA01      ; Init circular buffer base
    sub  #2,AC0
    mov  mmap(AC0L),BRC0      ; Initialize repeat counter to N-1
    mov  #0,AR0               ; Set buffer start address
    mov  #0,AR1               ;   as offset = 0
    bset AR0LC                ; Enable AR0 and AR1 as
    bset AR1LC                ;   circular pointers
||  rptblocal loop_end-1      ; Start bit reversal loop
    mov  dbl(*AR0),AC0        ; Get a pair of sample
||  amov AR1,T1
    mov  dbl(*AR1),AC1        ; Get another pair
||  asub AR0,T1
    xccpart swap1,T1>=#0
||  mov  AC1,dbl(*AR0+)       ; Swap samples if j>=i
swap1
    xccpart loop_end,T1>=#0
||  mov  AC0,dbl(*(AR1+T0B))
loop_end                      ; End bit reversal loop
    pop  mmap(ST2_55)         ; Restore ST2
    ret
```

The experiment computes 128-point FFT, inverse FFT, and the error between the input and the output of inverse FFT. The files used for this experiment are listed in Table 6.13.

Procedures of the experiment are listed as follows:

1. Open the project file, `asm_fft.pjt`, and rebuild the project.

2. Run the FFT experiment using the input data file `input.dat`.

3. Examine the FFT and IFFT input and output, and check the input and output differences stored in the array `error[ ]`.

Table 6.13 File listing for experiment `exp6.6.5_asm_FFT`

Files	Description
`asm_fftTest.c`	C function for testing assembly FFT experiment
`fft.asm`	Assembly function for FFT
`bit_rev.asm`	Assembly function performs bit reversal
`w_table.c`	C function generates twiddle factors
`asm_fft.h`	C header file for fixed-point FFT experiment
`icomplex.h`	C header file defines fixed-point complex data type
`asm_fft.pjt`	DSP project file
`asm_fft.cmd`	DSP linker command file
`input.dat`	Data file

6.6.6 Implementation of Fast Convolution

This experiment uses the overlap-add technique with the following steps:

- Pad M $(N - L)$ zeros to the FIR filter impulse response of length L where $N > L$, and process the sequence using an N-point FFT. Store the results in the complex buffer `H[N]`.
- Segment the input sequence of length M with $L - 1$ zeros padded at the end.
- Process each segment of data samples with an N-point FFT to obtain the complex array `X[N]`.
- Multiply `H` and `X` in frequency domain to obtain `Y`.
- Perform N-point IFFT to get the time-domain filtered sequence.
- Add the first L samples overlapped with the previous segment to form the output. Combine all the resulting segments to obtain$y(n)$.

The C program implementation of fast convolution using FFT and IFFT is listed in Table 6.14. The files used for this experiment are listed in Table 6.15.

Table 6.14 C program section for fast convolution

```
    for (i=0; i<L; i++)      /* Copy filter coefficient to work buffer */
    {
        X[i].re = LP_h[i];
        X[i].im = 0;
    }
    w_table(U,EXP);          /* Create Twiddle lookup table for FFT */
    bit_rev(X,EXP);          /* Bit reversal arrange the coefficient */
    fft(X,EXP,U,1);          /* FFT to the filter coefficients */

    for (i=0; i<N; i++)      /* Save frequency domain coefficients */
    {
       H[i].re = X[i].re <<EXP;
       H[i].im = X[i].im <<EXP;
    }
    /* Start FFT Convolution test */
```

continues overleaf

Table 6.14 (*continued*)

```
    j=0;
    for (;;)
    {
        for (i=0; i<M; i++)
        {
            X[i].re = input[j++];/* Generate input samples */
            X[i].im = 0;
            re1[i] = X[i].re;    /* Display re1[] shows all 3 freq. */
            if (j==DATA_LEN)
            {
                j=0;
            }
        }
        for (i=i; i<N; i++)      /* Fill zeros to data buffer */
        {
            X[i].re = 0;
            X[i].im = 0;
        }
        /* Start FFT convolution*/
        bit_rev(X,EXP);         /* Arrange sample in bit reversal order */
        fft(X,EXP,U,1);         /* Perform FFT */
        freqflt(X,H,N);         /* Perform frequency domain filtering */
        bit_rev(X,EXP);         /* Arrange sample in bit reversal order */
        fft(X,EXP,U,0);         /* Perform IFFT */
        olap_add(X,OVRLAP,L,M,N);/* Overlap and add algorithm */
    }
```

Procedures of the experiment are listed as follows:

1. Open the project file, `fast_convolution.pjt`, and rebuild the project.
2. Run the fast convolution experiment using the data file `input.dat`.
3. Replace different FIR filter coefficients provided in this project and profile the filter run time cycles for different filter lengths.

Table 6.15 File listing for experiment `exp6.6.6_fastconvolution`

Files	Description
`fast_convolution.c`	C function for testing fast convolution experiment
`fft.asm`	Assembly function for FFT
`bit_rev.asm`	Assembly function performs bit reversal
`freqflt.asm`	Assembly function performs fast convolution
`olap_add.asm`	Assembly function performs overlap-add
`w_table.c`	C function generates twiddle factors
`fast_convolution.h`	C header file for fast convolution filter experiment
`icomplex.h`	C header file defines fixed-point complex data type
`fast_convolution.pjt`	DSP project file
`fast_convolution.cmd`	DSP linker command file
`input.dat`	Data file
`firlp8.dat - firlp512.dat`	Lowpass filter coefficients from 8th to 512th

Table 6.16 File listing for experiment `exp6.6.7_realtime_FFT_DSPBIOS`

Files	Description
`realtime_DSPBIOS.c`	C function for testing real-time FFT experiment
`fft.asm`	Assembly function for FFT
`bit_rev.asm`	Assembly function performs bit reversal
`w_table.c`	C function generates twiddle factors
`plio.c`	C function for interface PIP with low-level I/O functions
`realtime_fft.h`	C header file for FFT experiment
`icomplex.h`	C header file defines fixed-point complex data type
`plio.h`	C header file for PIP with low-level I/O functions
`lio.h`	C header file for low-level I/O functions
`rt_FFT_DSPBIOS.pjt`	DSP project file
`rt_FFT_dspbioscfg.cmd`	DSP linker command file
`rt_FFT_dspbios.cdb`	DSP/BIOS configuration file
`timit2.wav`	Speech file
`tone.wav`	Tone file
`unControl.wav`	Data file

6.6.7 Real-Time FFT Using DSP/BIOS

This experiment integrates the FFT and IFFT using DSP/BIOS for real-time demonstrations. The C5510 DSK takes input signal from an audio source, applies FFT, and uses the IFFT to convert it back to time domain for real-time playback. The input signals used for this experiment include speech, pure tone, and modulated tone. These signals are sampled at 8000 Hz and stored in wave file format. The experiment uses 128-point FFT. Table 6.16 lists the files used for this experiment.

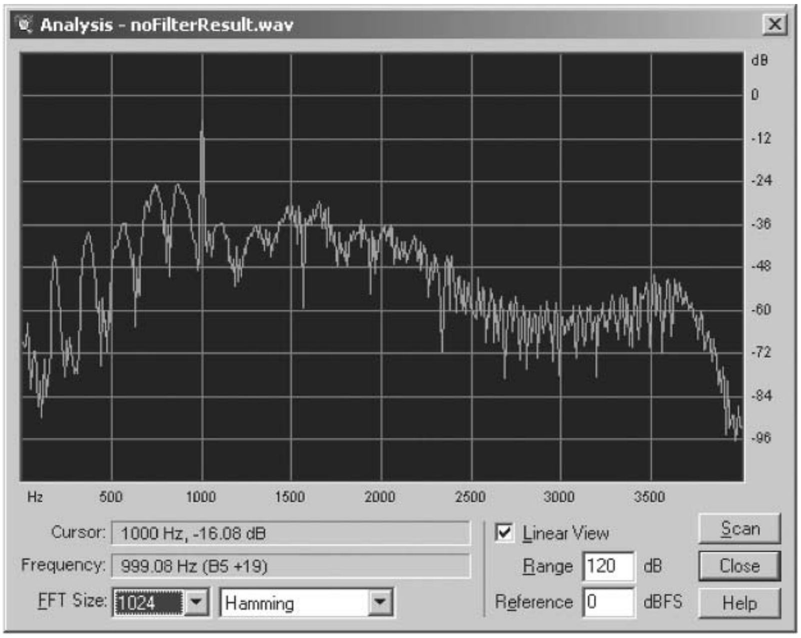
Procedures of the experiment are listed as follows:

1. Open the project file, `rt_FFT_DSPBIOS.pjt`, and rebuild the project.

2. Connect the line-in of the DSK with the source audio player and headphone-out to a headphone or loudspeaker.

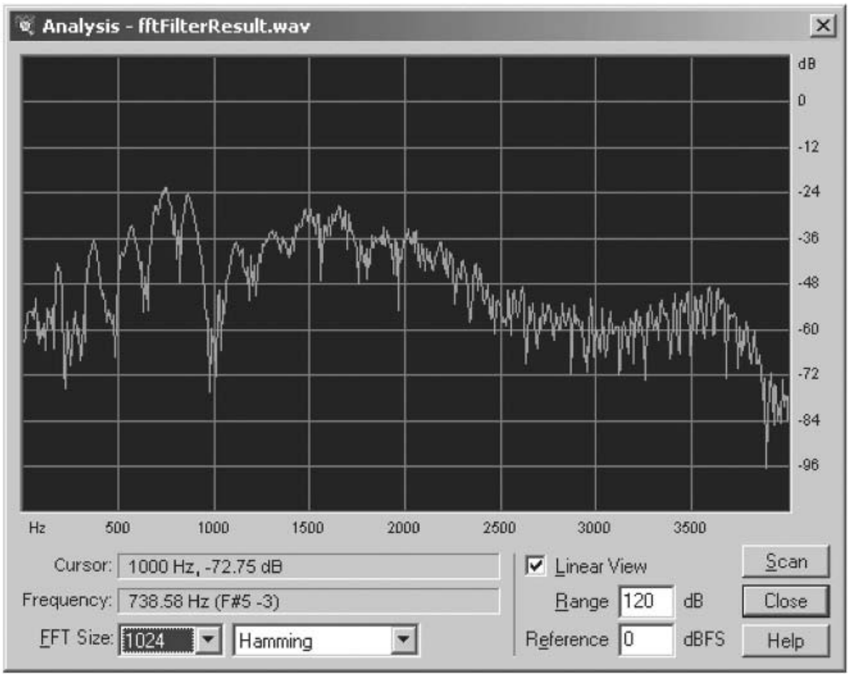
Table 6.17 File listing for experiment `exp66.8_realtime_fftFilter`

Files	Description
`rt_fftFilter.c`	C function for testing real-time FFT experiment
`fft.asm`	Assembly function for FFT
`bit_rev.asm`	Assembly function performs bit reversal
`freqflt.asm`	Assembly function performs fast convolution
`olap_add.asm`	Assembly function performs overlap-add
`olap_add.c`	C function controls overlap-add process
`plio.c`	C function for interface PIP with low-level I/O functions
`rt_fftFilter.h`	C header file for fast convolution experiment
`icomplex.h`	C header file defines fixed-point complex data type
`plio.h`	C header file for PIP with low-level I/O functions
`lio.h`	C header file for low-level I/O functions
`bandstop_128tap.h`	FIR filter coefficients
`rt_fftFilter.pjt`	DSP project file
`rt_fftFiltercfg.cmd`	DSP linker command file
`rt_fftFilter.cdb`	DSP/BIOS configuration file
`timit2.wav`	Speech file
`timit2_with_tone.wav`	Data file of speech + tone



Cursor: 1000 Hz, -16.08 dB
Frequency: 999.08 Hz (B5 +19)
FFT Size: 1024 Hamming
☑ Linear View
Range 120 dB
Reference 0 dBFS

(a) Speech signal with 1000 Hz tone interference, 1000 Hz tone at −16.08 dB.



Cursor: 1000 Hz, -72.75 dB
Frequency: 738.58 Hz (F#5 -3)
FFT Size: 1024 Hamming
☑ Linear View
Range 120 dB
Reference 0 dBFS

(b) After FIR filtering, the 1000 Hz tone has been reduced to −72.75 dB.

Figure 6.23 Fast convolution for removing 1000 Hz interference from speech: (a) speech signal with the 1000 Hz tone interference, 1000 Hz tone at −16.08 dB; (b) after FIR filtering, the 1000 Hz tone has been reduced to −72.75 dB

3. Run the experiment and listen to the playback of audio samples.

4. Repeat the experiment using different audio data files.

6.6.8 Real-Time Fast Convolution

This experiment uses the fast convolution with the DSP/BIOS for real-time FIR filtering. The files used for this experiment are listed in Table 6.17. The input signals used for this experiment include the speech files `timit2.wav` and `timit2_with_tone.wav`, which adds 1000 Hz tone to `timit2.wav`. Both wave files have sampling rate of 8000 Hz. The experiment uses 512-point FFT. The FIR filter was designed by MATLAB in Example 6.21.

Procedures of the experiment are listed as follows:

1. Open the project file, `exp6_rt_fftFilter.pjt`.

2. Connect the line-in of the DSK with the audio source and headphone-out to a headphone or loudspeaker.

3. Open the source file, `rt_fftFilter.c`, turn off the conditional compile switch FAST_CONVOLUTION, and recompile the project. This will disable the FIR filtering using fast convolution.

4. Run the project and listen to the audio playback. The 1000 Hz tone interference can be heard clearly because the filter is set in bypass mode. The magnitude spectrum shown in Figure 6.23(a) shows the strong presence of 1000 Hz tone.

5. Open the source file `rt_fftFilter.c`, set the conditional compile switch FAST_CONVOLUTION on, and recompile the project. This will perform the fast convolution as FIR filtering in the project.

6. Rerun the project and listen to the real-time playback of audio samples. The 1000 Hz tone interference will be reduced as shown in Figure 6.23(b).

References

[1] D. J. DeFatta, J. G. Lucas, and W. S. Hodgkiss, *Digital Signal Processing*: *A System Design Approach*, New York: John Wiley & Sons, Inc., 1988.
[2] N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Englewood Cliffs, NJ: Prentice Hall, 1983.
[3] S. M. Kuo and W. S. Gan, *Digital Signal Processors*, Upper Saddle River, NJ: Prentice Hall, 2005.
[4] L. B. Jackson, *Digital Filters and Signal Processing*, 2nd Ed., Boston, MA: Kluwer Academic, 1989.
[5] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1989.
[6] S. J. Orfanidis, *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1996.
[7] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd Ed., Englewood Cliffs, NJ: Prentice Hall, 1996.
[8] A. Bateman and W. Yates, *Digital Signal Processing Design*, New York: Computer Science Press, 1989.
[9] S. D. Stearns and D. R. Hush, *Digital Signal Analysis*, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1990.
[10] W. Tian, *Method for Efficient and Zero Latency Filtering in a Long-Impulse-Response System*, European patent, WO0217486A1, Feb. 2002.

[11] Math Works, Inc.,*Using MATLAB*, Version 6, 2000.
[12] Math Works, Inc., *Signal Processing Toolbox User's Guide*, Version 6, 2004.
[13] Math Works, Inc., *Filter Design Toolbox User's Guide*, Version 3, 2004.
[14] Math Works, Inc., *Fixed-Point Toolbox User's Guide*, Version 1, 2004.

Exercises

1. Compute the Fourier series coefficients of cosine function $x(t) = \cos(2\pi f_0 t)$.

2. Compute the 4-point DFT of the sequence $\{1, 1, 1, 1\}$ using the matrix equation given in Equation (6.19).

3. Compute $X(0)$ and $X(4)$ of 8-point DFT of sequence $\{1, 1, 1, 1, 2, 3, 4, 5\}$.

4. Prove the symmetry and periodicity properties of the twiddle factors defined as

   (a) $W_N^{k+N/2} = -W_N^k$;

   (b) $W_N^{k+N} = W_N^k$.

5. Consider the following two sequences:
   $x_1(n) = 1$ and $x_2(n) = n, \qquad 0 \le n \le 3$.

   (a) Compute the linear convolution of these two sequences.

   (b) Compute the circular convolution of these two sequences.

   (c) Show how to pad zeros for these two sequences such that the circular convolution results are the same as linear convolution in (a).

6. Construct the signal-flow diagram of FFT for $N = 16$ using the decimation-in-time method.

7. Construct the signal-flow diagram of FFT for $N = 8$ using the decimation-in-frequency method.

8. Similar to Table 6.1, show the bit-reversal process for 16-point FFT.

9. Consider 1 s of digitized signal with sampling rate 20 kHz. It is desired to have the spectrum with a frequency resolution of 100 Hz or less. Is this possible? If impossible, what FFT size N should be used?

10. A 1 kHz sinusoid is sampled at 8 kHz. The 128-point FFT is performed to compute $X(k)$. What is the computational resolution? At what frequency indices k we expect to observe peaks in $|X(k)|$? Can we observe the line spectrum?

11. A touch-tone phone with a dual-tone multifrequency (DTMF) transmitter encodes each keypress as a sum of two sinusoids, with two frequencies taken from each of the following groups:

    Vertical group: 697, 770, 852, 941 Hz;

    Horizontal group: 1209, 1336, 1477, 1633 Hz.

    What is the smallest DFT size N that we can distinguish these two sinusoids from the computed spectrum? The sampling rate used in telecommunications is 8 kHz.

12. Write a MATLAB script to verify the DFT and IDFT results obtained in Example 6.7.

13. Similar to Example 6.9 and given $x(n) = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $h(n) = \{1, 0, 1, 0, 1, 0, 1, 0\}$. Write MATLAB scripts that implement the following tasks:

    (a) linear convolution;

    (b) circular convolution using FFT and IFFT; and

    (c) fast convolution with zero padding of two sequences $x(n)$ and $h(n)$.

14. Similar to Example 6.14, compute fixed-point FFT with Q15 format using the scaling factor 1/128 at stage 1. Does overflow occur? Compare the results with scaling factor 0.5 at the input of each stage.

15. Similar to Example 6.14 but using a zero-mean, variance = 0.5, white noise instead of sinewave, compute Q15 FFT without scaling. How many overflows occur in the quantized FFT? Try different scaling vectors, `F.ScaleValues`, and discuss the difference by using sinewave as input.

16. Similar to Example 6.17, use different techniques to distinguish two sinewaves at 60 and 61 Hz.

17. Write a C or MATLAB program to compute the fast convolution of a long data sequence with a short coefficient sequence employing the overlap-save method introduced in Section 6.5.4. Compare the results with the MATLAB function `fftfilt` that uses overlap-add method.

18. The radix-2 FFT code used in the experiments is written in consideration of minimizing the code size. An alternative FFT implementation can be more efficient in terms of the execution speed with the expense of using more program memory. For example, the twiddle factor used by the first stage and the first group of other stages is constant $W_N^0 = 1$. Therefore, the multiplication operations in these stages can be simplified. Modify the assembly FFT routine given in Table 6.11 to incorporate this observation. Profile the run-time clock cycles and record the memory usage. Compare the results with those obtained by the experiment given in Section 6.6.5.

19. The radix-2 FFT is the most widely used algorithm for FFT computation. When the number of data samples is a power of $2m$ (i.e., $N = 2^{2m} = 4^m$), we can further improve the run-time efficiency by employing the radix-4 FFT algorithm. Modify the assembly FFT routine given in Table 6.11 to the radix-4 FFT algorithm. Profile the run-time clock cycles, and record the memory space usage for a 1024-point radix-4 FFT ($2^{10} = 4^5 = 1024$). Compare the radix-4 FFT results with the results of 1024-point radix-2 FFT computed by the assembly routine.

20. Take advantage of twiddle factor $W_N^0 = 1$ to further improve the radix-4 FFT algorithm run-time efficiency. Compare the results of 1024-point FFT implementation using different approaches.

21. Most of the DSP applications use real input samples, our complex FFT implementation zeros out the imaginary components of the complex buffer (see experiment given in Section 6.6.5). This approach is simple and easy, but it is not efficient in terms of the execution speed. For real input, we can split the even and odd samples into two sequences, and compute both even and odd sequences in parallel. This approach will reduce the execution time by approximately 50 %. Given a real-value input $x(n)$ of $2N$ samples, we can define $c(n) = a(n) + jb(n)$, where two inputs $a(n) = x(n)$ and $b(n) = x(n + 1)$ are real sequences. we can represent these sequences as $a(n) = [c(n) + c^*(n)]/2$ and $b(n) = -j[c(n) - c^*(n)]/2$, then they can be written in terms of DFTs as $A_k(k) = [C(k) + C^*(N - k)]/2$ and $B_k(k) = -j[C(k) - C^*(N - k)]/2$. Finally, the real input FFT can be obtained by $X(k) = A_k(k) + W_{2N}^k B_k(k)$ and $X(k + N) = A_k(k) - W_{2N}^k B_k(k)$, where $k = 0, 1, \ldots, N - 1$. Modify the complex radix-2 FFT assembly routine to efficiently compute $2N$ real input samples.

22. Write a 128-point, decimation-in-frequency FFT function in fixed-point C or intrinsics and verify it using the experiment given in Section 6.6.4. Then write the FFT function using C55x assembly language and verify it using the experiment given in Section 6.6.5.

23. The TMS320C55x supports bit-reversal addressing mode. Replace the bit-reversal function with the C55x bit-reversal addressing mode for the experiment given in Section 6.6.5.

24. Develop an experiment to compute the PSD using the C55x simulator or DSK. Add Hamming window to compute PSD.

25. Develop an experiment for fast convolution using the overlap-save technique. Verify the fast convolution result using the data from the experiment given in Section 6.6.6.

26. Develop an experiment and compare the computational load between the experiment given in Section 6.6.6 using fast convolution method and using direct FIR filtering when FIR filter order is 512.

7 Adaptive Filtering

We have introduced techniques for design and implementation of time-invariant FIR and IIR filters with fixed coefficients in Chapters 4 and 5, respectively. In this chapter, we will introduce time-varying adaptive filters with changing characteristics.

7.1 Introduction to Random Processes

As discussed in Section 3.3, the real-world signals are often random in nature. Some common examples of random signals are speech, music, and noises. In this section, we will briefly review the important properties of the random processes and introduce fundamental processing techniques.

The autocorrelation function of the random process $x(n)$ is defined as

$$r_{xx}(n, k) = E\left[x(n)x(k)\right]. \tag{7.1}$$

This function specifies the statistical relation at different time indices n and k, and gives the degree of dependence between two random variables of $(n - k)$ units apart.

Example 7.1: Consider a digital white noise $x(n)$ as uncorrelated random variables with zero-mean and variance σ_x^2. The autocorrelation function is

$$\begin{aligned} r_{xx}(n,\ k) &= E\left[x(n)x(k)\right] = E\left[x(n)\right] E\left[x(k)\right] \\ &= \begin{cases} 0, & n \neq k \\ \sigma_x^2 & n = k \end{cases}. \end{aligned}$$

Correlation is a very useful tool for detecting signals that are corrupted by additive random noises, measuring the time delay between two signals, determining the impulse response of a system, and many others. Correlation is often used in radar, sonar, digital communications, and other engineering areas. For example, in radar and sonar applications, the received signal reflected from the target is the delayed version of the transmitted signal. By measuring the round-trip delay using an appropriate correlation function, the radar and sonar can determine the distance of the target.

A random process is stationary if its statistics do not change with time. The most useful and relaxed form of stationary is the wide-sense stationary (WSS) process that satisfies the following two conditions:

1. The mean of the process is independent of time. That is

$$E[x(n)] = m_x, \tag{7.2}$$

   where the mean m_x is a constant.

2. The autocorrelation function depends only on the time difference. That is

$$r_{xx}(k) = E\ [x(n+k)x(n)], \tag{7.3}$$

   where k is the time lag.

Example 7.2: Given the WSS sequence $x(n) = a^n u(n),\ 0 < a < 1$, the autocorrelation function can be computed as

$$r_{xx}(k) = \sum_{n=-\infty}^{\infty} x(n+k)x(n) = \sum_{n=0}^{\infty} a^{n+k}a^n = a^k \sum_{n=0}^{\infty} (a^2)^n.$$

Since $a < 1$, we obtain

$$r_{xx}(k) = \frac{a^k}{1-a^2}.$$

The autocorrelation function $r_{xx}(k)$ of a WSS process has the following important properties:

1. The autocorrelation function is an even function. That is,

$$r_{xx}(-k) = r_{xx}(k). \tag{7.4}$$

2. The autocorrelation function is bounded by

$$|r_{xx}(k)| \le r_{xx}(0), \tag{7.5}$$

   where $r_{xx}(0) = E[x^2(n)]$ is the mean-square value, or the power of random process $x(n)$. In addition, if $x(n)$ is a zero-mean random process, we have

$$r_{xx}(0) = E[x^2(n)] = \sigma_x^2. \tag{7.6}$$

Example 7.3: Considering the sinusoidal signal expressed as $x(n) = \cos(\omega n)$, find the mean and the autocorrelation function of $x(n)$:

(a) $m_x = E\,[\cos(\omega n)] = 0,$

(b) $$\begin{aligned} r_{xx}(k) &= E[x(n+k)x(n)] = E[\cos(\omega n + \omega k)\cos(\omega n)] \\ &= \frac{1}{2}E[\cos(2\omega n + \omega k)] + \frac{1}{2}\cos(\omega k) = \frac{1}{2}\cos(\omega k). \end{aligned}$$

The crosscorrelation function between two WSS processes $x(n)$ and $y(n)$ is defined as

$$r_{xy}(k) = E\left[x(n+k)y(n)\right]. \tag{7.7}$$

This crosscorrelation function has the property

$$r_{xy}(k) = r_{yx}(-k). \tag{7.8}$$

Therefore, $r_{yx}(k)$ is simply the folded version of $r_{xy}(k)$.

In practice, we may only have one sample sequence $\{x(n)\}$ available for analysis. In dealing with finite-duration sequence, the sample mean of $x(n)$ is defined as

$$\bar{m}_x = \frac{1}{N}\sum_{n=0}^{N-1} x(n), \tag{7.9}$$

where N is the number of samples in the short-time analysis interval. The sample autocorrelation function is defined as

$$\bar{r}_{xx}(k) = \frac{1}{N-k}\sum_{n=0}^{N-k-1} x(n+k)x(n), \qquad k = 0, 1, \ldots, N-1. \tag{7.10}$$

In practice, we can only expect good results for lags of no more than 5–10 % of the length of the signals.

An important random signal is called white noise which has zero mean. Its autocorrelation function is expressed as

$$r_{xx}(k) = \sigma_x^2 \delta(k), \tag{7.11}$$

and the power spectrum is given by

$$P_{xx}(\omega) = \sigma_x^2, \qquad |\omega| < \pi, \tag{7.12}$$

which is of constant value for all frequencies ω.

Example 7.4: Consider a second-order FIR filter expressed as

$$y(n) = x(n) + 3x(n-1) + 2x(n-2).$$

The input $x(n)$ is a zero-mean white noise. Find the mean m_y and the autocorrelation function $r_{yy}(k)$ of the output $y(n)$:

(a) $m_y = E[y(n)] = E[x(n)] + 3E[x(n-1)] + 2E[x(n-2)] = 0,$

(b)
$$\begin{aligned} r_{yy}(k) &= E[y(n+k)y(n)] \\ &= 14r_{xx}(k) + 9r_{xx}(k-1) + 9r_{xx}(k+1) + 2r_{xx}(k-2) + 2r_{xx}(k+2) \\ &= \begin{cases} 14\sigma_x^2, & \text{if } k = 0 \\ 9\sigma_x^2, & \text{if } k = \pm 1 \\ 2\sigma_x^2, & \text{if } k = \pm 2 \\ 0, & \text{otherwise} \end{cases}. \end{aligned}$$

MATLAB *Signal Processing Toolbox* provides the function `xcorr` for estimating crosscorrelation function as

```
    c = xcorr(x,y)
```

where `x` and `y` are length N vectors. This function returns the length $2N-1$ crosscorrelation sequence `c`. This function also estimates autocorrelation function as

```
    a = xcorr(x)
```

7.2 Adaptive Filters

The signal degradation in some physical systems is time varying, unknown, or possibly both. For example, consider a high-speed modem for transmitting and receiving data over telephone channels. It employs a filter called a channel equalizer to compensate for the channel distortion. Since the dial-up communication channels have different and time-varying characteristics on each connection, the equalizer must be an adaptive filter.

Adaptive filters modify their characteristics to achieve certain objectives by automatically updating their coefficients. Many adaptive filter structures and adaptation algorithms have been developed for different applications. This chapter presents the most widely used adaptive filters based on the FIR filter with the least-mean-square (LMS) algorithm. These adaptive filters are relatively simple to design and implement. They are well understood with regard to stability, convergence speed, steady-state performance, and finite-precision effects.

7.2.1 Introduction to Adaptive Filtering

An adaptive filter consists of two distinct parts – a digital filter to perform the desired filtering, and an adaptive algorithm to adjust the coefficients (or weights) of the filter. A general form of adaptive filter is illustrated in Figure 7.1, where $d(n)$ is a desired (or primary input) signal, $y(n)$ is the output of a digital filter driven by a reference input signal $x(n)$, and an error signal $e(n)$ is the difference between $d(n)$ and $y(n)$. The adaptive algorithm adjusts the filter coefficients to minimize the mean-square value of $e(n)$. Therefore, the filter weights are updated so that the error is progressively minimized on a sample-by-sample basis.

In general, there are two types of digital filters that can be used for adaptive filtering: FIR and IIR filters. The FIR filter is always stable and can provide a linear-phase response. On the other hand, the IIR

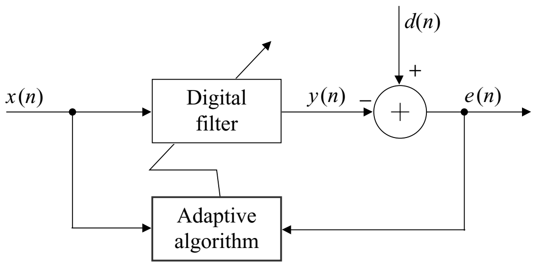


Figure 7.1 Block diagram of adaptive filter

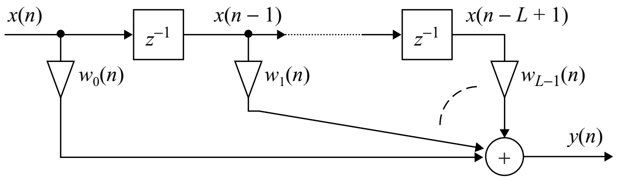


Figure 7.2 Block diagram of FIR filter for adaptive filtering

filter involves both zeros and poles. Unless they are properly controlled, the poles in the filter may move outside the unit circle and result in an unstable system during the adaptation of coefficients. Thus, the adaptive FIR filter is widely used for practical real-time applications. This chapter focuses on the class of adaptive FIR filters.

The most widely used adaptive FIR filter is depicted in Figure 7.2. The filter output signal is computed as

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l), \tag{7.13}$$

where the filter coefficients $w_l(n)$ are time varying and updated by the adaptive algorithms that will be discussed next.

We define the input vector at time n as

$$\mathbf{x}(n) \equiv [x(n)x(n-1)\ \ldots\ x(n-L+1)]^T\ , \tag{7.14}$$

and the weight vector at time n as

$$\mathbf{w}(n) \equiv [w_0(n)w_1(n)\ \ldots\ w_{L-1}(n)]^T\ . \tag{7.15}$$

Equation (7.13) can be expressed in vector form as

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n) = \mathbf{x}^T(n)\mathbf{w}(n). \tag{7.16}$$

The filter output $y(n)$ is compared with the desired $d(n)$ to obtain the error signal

$$e(n) = d(n) - y(n) = d(n) - \mathbf{w}^T(n)\mathbf{x}(n). \tag{7.17}$$

Our objective is to determine the weight vector $\mathbf{w}(n)$ to minimize the predetermined performance (or cost) function.

7.2.2 Performance Function

The adaptive filter shown in Figure 7.1 updates the coefficients of the digital filter to optimize some predetermined performance criterion. The most commonly used performance function is based on the mean-square error (MSE) defined as

$$\xi(n) \equiv E[e^2(n)]. \tag{7.18}$$

The MSE function determined by substituting Equation (7.17) into (7.18) can be expressed as

$$\xi(n) = E[d^2(n)] - 2\mathbf{p}^T\mathbf{w}(n) + \mathbf{w}^T(n)\mathbf{R}\mathbf{w}(n), \tag{7.19}$$

where **p** is the crosscorrelation vector defined as

$$\begin{aligned}\mathbf{p} &\equiv E[d(n)\mathbf{x}(n)] \\ &= [r_{dx}(0) r_{dx}(1) \ldots r_{dx}(L-1)]^T,\end{aligned} \tag{7.20}$$

and

$$r_{dx}(k) \equiv E[d(n+k)x(n)] \tag{7.21}$$

is the crosscorrelation function between $d(n)$ and $x(n)$. In Equation (7.19), **R** is the input autocorrelation matrix defined as

$$\begin{aligned}\boldsymbol{R} &\equiv E[\mathbf{x}(n)\mathbf{x}^T(n)] \\ &= \begin{bmatrix} r_{xx}(0) & r_{xx}(1) & \cdots & r_{xx}(L-1) \\ r_{xx}(1) & r_{xx}(0) & \cdots & r_{xx}(L-2) \\ \vdots & \cdots & \ddots & \vdots \\ r_{xx}(L-1) & r_{xx}(L-2) & \cdots & r_{xx}(0) \end{bmatrix},\end{aligned} \tag{7.22}$$

where $r_{xx}(k)$ is the autocorrelation function of $x(n)$ defined in Equation (7.3).

Example 7.5: Consider an optimum filter illustrated in Figure 7.3. If $E[x^2(n)] = 1$, $E[x(n)x(n-1)] = 0.5$, $E\lfloor d^2(n)\rfloor = 4$, $E[d(n)x(n)] = -1$, and $E[d(n)x(n-1)] = 1$, find ξ.

From Equation (7.22), we have $\mathbf{R} = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$, and from Equation (7.20), we have $\mathbf{p} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$. Therefore from Equation (7.19), we obtain

$$\begin{aligned}\xi &= E[d^2(n)] - 2\mathbf{p}^T\mathbf{w} + \mathbf{w}^T\mathbf{R}\mathbf{w} \\ &= 4 - 2\begin{bmatrix} -1 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ w_1 \end{bmatrix} + \begin{bmatrix} 1 & w_1 \end{bmatrix}\begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ w_1 \end{bmatrix} \\ &= w_1^2 - w_1 + 7.\end{aligned}$$

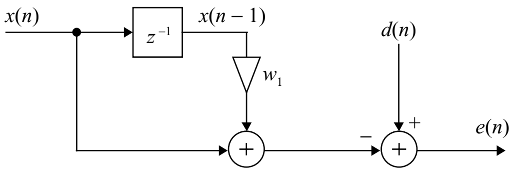


Figure 7.3 A simple optimum filter configuration

The optimum filter $\mathbf{w}^\circ$ minimizes the MSE function $\xi(n)$. Differentiation of Equation (7.19) with respect to $\mathbf{w}$ and setting the result to $\mathbf{0}$, we have

$$\mathbf{R}\mathbf{w}^\circ = \mathbf{p}. \tag{7.23}$$

This equation provides a solution to the adaptive filtering problem in principle. In many applications, the computation of solution

$$\mathbf{w}^\circ = \mathbf{R}^{-1}\mathbf{p} \tag{7.24}$$

requires continuous estimation of $\mathbf{R}$ and $\mathbf{p}$ since the signal may be nonstationary. In addition, when the dimension of the autocorrelation matrix is large, the calculation of $\mathbf{R}^{-1}$ may present a significant computational burden. Therefore, a more useful algorithm using a recursive method for computing $\mathbf{w}^\circ$ has been developed, which will be discussed in the next section.

By substituting the optimum weight vector in Equation (7.24) for $\mathbf{w}(n)$ in Equation (7.19), we obtain the minimum MSE:

$$\xi_{\min} = E[d^2(n)] - \mathbf{p}^T\mathbf{w}^\circ. \tag{7.25}$$

Since $\mathbf{R}$ is positive semidefinite, the quadratic form on the right-hand side of Equation (7.19) indicates that any departure of the weight vector $\mathbf{w}(n)$ from the optimum $\mathbf{w}^\circ$ would increase the error above its minimum value. This feature is very useful when we utilize search techniques in seeking the optimum weight vector. In such cases, our objective is to develop an algorithm that can automatically search the error surface to find the optimum weights that minimize $\xi(n)$ using the input signal $x(n)$ and the error signal $e(n)$.

Example 7.6: Consider an FIR filter with two coefficients w_0 and w_1, the desired signal $d(n) = \sqrt{2}\sin(n\omega_0)$, where $n \geq 0$, and the reference signal $x(n) = d(n-1)$. Find $\mathbf{w}^\circ$ and $\xi_{\min}$.

Similar to Example 7.5, we can obtain $r_{xx}(0) = E[x^2(n)] = E[d^2(n)] = 1$, $r_{xx}(1) = \cos(\omega_0)$, $r_{xx}(2) = \cos(2\omega_0)$, $r_{dx}(0) = r_{xx}(1)$, and $r_{dx}(1) = r_{xx}(2)$. From Equation (7.24), we have

$$\mathbf{w}^\circ = \mathbf{R}^{-1}\mathbf{p} = \begin{bmatrix} 1 & \cos(\omega_0) \\ \cos(\omega_0) & 1 \end{bmatrix}^{-1} \begin{bmatrix} \cos(\omega_0) \\ \cos(2\omega_0) \end{bmatrix} = \begin{bmatrix} 2\cos(\omega_0) \\ -1 \end{bmatrix}.$$

From Equation (7.25), we obtain

$$\xi_{\min} = 1 - \begin{bmatrix} \cos(\omega_0) & \cos(2\omega_0) \end{bmatrix} \begin{bmatrix} 2\cos(\omega_0) \\ -1 \end{bmatrix} = 0.$$

Equation (7.19) is the general expression of the performance function for an adaptive FIR filter with filter coefficient vector $\mathbf{w}(n)$. The MSE is a quadratic function because the weights appear only to the first and second degrees in Equation (7.19). For each coefficient vector $\mathbf{w}(n)$, there is a corresponding value of MSE. Therefore, the MSE values associated with $\mathbf{w}(n)$ form an $(L+1)$-dimensional space, which is commonly called the MSE surface, or the performance surface.

For $L = 2$, this corresponds to an error surface in a three-dimensional space. The height of $\xi(n)$ corresponds to the power of the error signal $e(n)$. If the filter coefficients change, the power in the error signal will also change. Since the error surface is quadratic, a unique filter setting $\mathbf{w}(n) = \mathbf{w}^\circ$ will produce the minimum MSE, $\xi_{\min}$. In the two-weight case, the error surface is an elliptic paraboloid. If we cut the paraboloid with planes parallel to the $w_0 - w_1$ plane, we obtain concentric ellipses of constant MSEs. These ellipses are called the error contours.

Example 7.7: Consider an FIR filter with two coefficients w_0 and w_1. The reference signal $x(n)$ is a zero-mean white noise with unit variance. The desired signal is given as

$$d(n) = b_0 x(n) + b_1 x(n-1).$$

Plot the error surface and error contours.

From Equation (7.22), we obtain $\mathbf{R} = \begin{bmatrix} r_{xx}(0) & r_{xx}(1) \\ r_{xx}(1) & r_{xx}(0) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. From Equation (7.20), we have $\mathbf{p} = \begin{bmatrix} r_{dx}(0) \\ r_{dx}(1) \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$. From Equation (7.19), we get

$$\begin{aligned} \xi &= E[d^2(n)] - 2\mathbf{p}^T\mathbf{w} + \mathbf{w}^T\mathbf{R}\mathbf{w} \\ &= \left(b_0^2 + b_1^2\right) - 2b_0w_0 - 2b_1w_1 + w_0^2 + w_1^2. \end{aligned}$$

Let $b_0 = 0.3$ and $b_1 = 0.5$, we have

$$\xi = 0.34 - 0.6w_0 - w_1 + w_0^2 + w_1^2.$$

The MATLAB script (`example7_7a.m`) plots the error surface shown in Figure 7.4 (top) and the script `example7_7b.m` plots the error contours shown in bottom of Figure 7.4.

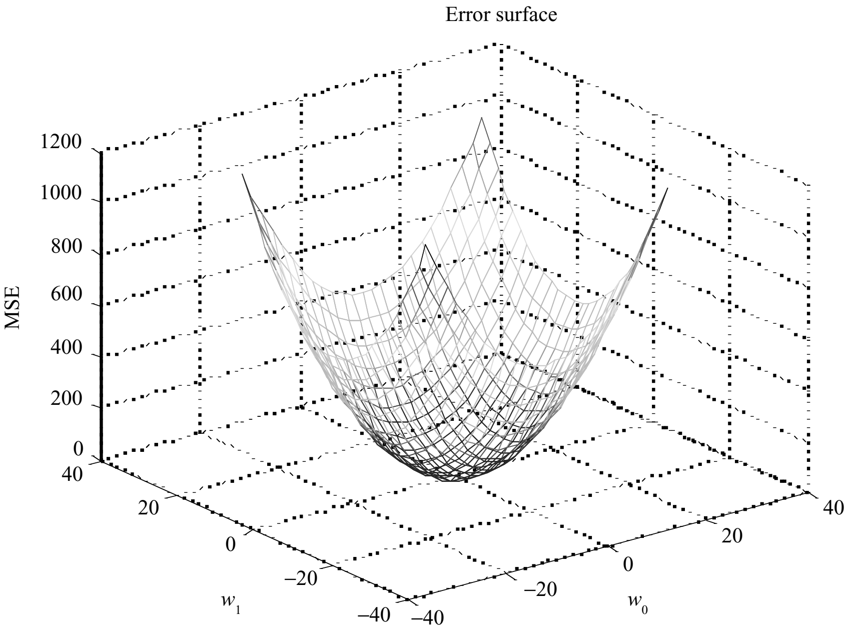
7.2.3 Method of Steepest Descent

As shown in Figure 7.4, the MSE defined by Equation (7.19) is a quadratic function of the weights that can be pictured as a positive-concave hyperparabolic surface with only one global minimum point. Adjusting the weights to minimize the error signal involves descending along this surface until reaching the 'bottom of the bowl.' Gradient-based algorithms are based on making local estimates of the gradient and moving toward the bottom of the bowl. The steepest-descent method reaches the minimum by following the negative-gradient direction in which the performance surface has the greatest rate of decrease.

The steepest-descent method is an iterative (recursive) technique that starts from some arbitrary initial weight vector $\mathbf{w}(0)$. This technique descends to the bottom of the bowl, $\mathbf{w}^\circ$, by moving on the error surface in the direction of the tangent at that point. The mathematical development of the method of steepest descent can be obtained from a geometric approach using the MSE surface. A specific orientation to the surface is obtained using the directional derivatives of the surface at that point. The gradient of the error surface $\nabla\xi(n)$ is defined as the vector of these directional derivatives. The concept of steepest descent can be implemented in the following algorithm:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \frac{\mu}{2}\nabla\xi(n), \tag{7.26}$$

where μ is a convergence factor (or step size) that controls stability and the rate of convergence. The larger the value of μ, the faster the convergence speed will be. The vector $\nabla\xi(n)$ denotes the gradient of the error function with respect to $\mathbf{w}(n)$, and the negative sign updates the weight vector in the negative-gradient direction. The successive corrections to the weight vector in the direction of the steepest descent of the performance surface should eventually lead to the optimum value $\mathbf{w}^\circ$, corresponding to the minimum MSE $\xi_{\min}$.



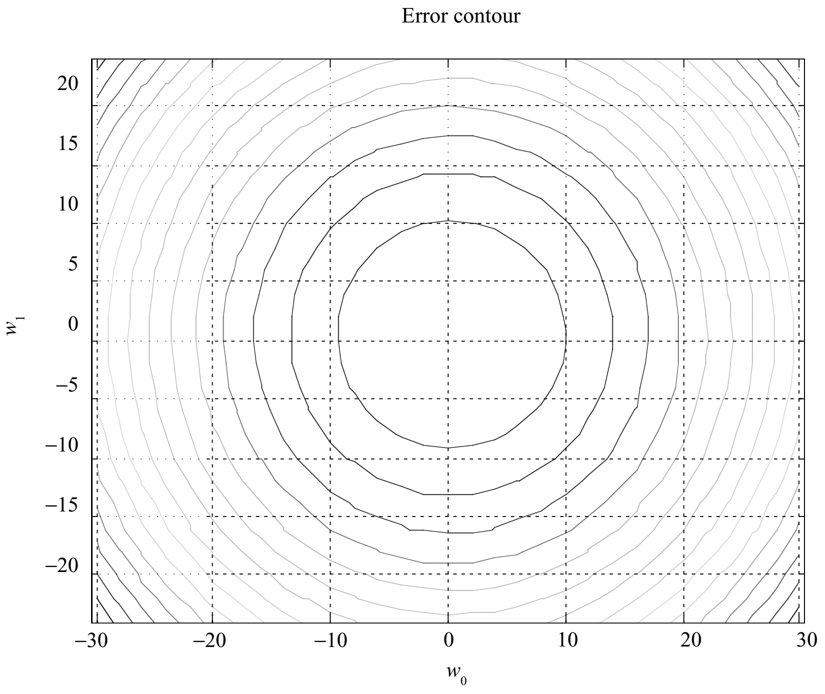


Figure 7.4 Performance surface (top) and error contours (bottom), $L = 2$

When $\mathbf{w}(n)$ has converged to $\mathbf{w}^{\circ}$, it reaches the minimum point of the performance surface. At this time when the gradient $\nabla\xi(n) = 0$, the adaptation process defined by Equation (7.26) is stopped and the weight vector stays at the optimum solution. The convergence can be viewed as a ball placed on the 'bowl-shaped' MSE surface at the point $[\mathbf{w}(0), \xi(0)]$. When the ball is released, it would roll toward the minimum of the surface, the bottom of the bowl, $[\mathbf{w}^{\circ}, \xi_{\min}]$.

7.2.4 The LMS Algorithm

In many practical applications, the statistics of $d(n)$ and $x(n)$ are unknown. Therefore, the method of steepest descent cannot be used directly since it assumes exact knowledge of the gradient vector. The LMS algorithm uses the instantaneous squared error, $e^2(n)$, to estimate the MSE. That is,

$$\hat{\xi}(n) = e^2(n). \tag{7.27}$$

Therefore, the gradient estimate used by the LMS algorithm can be written as

$$\nabla\hat{\xi}(n) = 2\,[\nabla e(n)]\,e(n). \tag{7.28}$$

Since $e(n) = d(n) - \mathbf{w}^T(n)\mathbf{x}(n)$, $\nabla e(n) = -\mathbf{x}(n)$, the gradient estimate becomes

$$\nabla\hat{\xi}(n) = -2\mathbf{x}(n)e(n). \tag{7.29}$$

Substituting this gradient estimate into the steepest-descent algorithm of Equation (7.26), we have

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\mathbf{x}(n)e(n). \tag{7.30}$$

This is the well-known LMS algorithm, or stochastic gradient algorithm. This algorithm is simple and does not require squaring, averaging, or differentiating.

The LMS algorithm is illustrated in Figure 7.5 and is summarized as follows:

1. Determine L, μ, and $\mathbf{w}(0)$, where L is the length of the filter, μ is the step size, and $\mathbf{w}(0)$ is the initial weight vector at time $n = 0$.

2. Compute the adaptive filter output

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l). \tag{7.31}$$

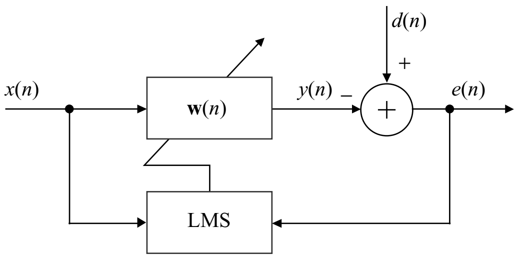


Figure 7.5 Block diagram of an adaptive filter with the LMS algorithm

3. Compute the error signal

$$e(n) = d(n) - y(n). \tag{7.32}$$

4. Update the adaptive weight vector using the LMS algorithm:

$$w_l(n+1) = w_l(n) + \mu\, x(n-l)e(n), \qquad l = 0, 1, \ldots, L-1. \tag{7.33}$$

7.2.5 Modified LMS Algorithms

There are three simplified versions of the LMS algorithm that further reduce the number of multiplications. However, the convergence rates of these LMS algorithms are slower than the LMS algorithm. The first algorithm called sign-error LMS algorithm can be expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\, \mathbf{x}(\mathrm{n})\mathrm{sgn}[e(n)], \tag{7.34}$$

where

$$\mathrm{sgn}[e(n)] \equiv \begin{cases} 1, & e(n) > 0 \\ 0, & e(n) = 0\,. \\ -1, & e(n) < 0 \end{cases} \tag{7.35}$$

This sign operation of error signal is equivalent to a very harsh quantization of $e(n)$. If μ is a negative power of 2, $\mu\,\mathbf{x}(n)$ can be computed with a right shift of $x(n)$. In DSP implementations, however, the conditional tests require more instruction cycles than the multiplications needed by the LMS algorithm.

The sign operation can be performed on data $x(n)$ instead of error $e(n)$, and it results in the sign-data LMS algorithm expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\, e(n)\mathrm{sgn}\,[\mathbf{x}(n)]\,. \tag{7.36}$$

Since L branch (IF-ELSE) instructions are required inside the adaptation loop to determine the signs of $x(n-i)$, $i = 0, 1, \ldots, L-1$, slower throughput than the sign-error LMS algorithm is expected.

Finally, the sign operation can be applied to both $e(n)$ and $x(n)$, and it results in the sign-sign LMS algorithm expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\ \mathrm{sgn}\,[e(n)]\,\mathrm{sgn}\,[\mathbf{x}(n)]\,. \tag{7.37}$$

This algorithm requires no multiplication, and is designed for VLSI or ASIC implementation to save multiplications. It is used in the adaptive differential pulse code modulation (ADPCM) for speech compression.

Some practical applications such as modems and frequency-domain adaptive filtering require complex operations for maintaining their phase relationships. The complex adaptive filter uses the complex input vector $\mathbf{x}(n)$ and complex coefficient vector $\mathbf{w}(n)$ expressed as

$$\mathbf{x}(n) = \mathbf{x}_\mathrm{r}(n) + j\mathbf{x}_\mathrm{i}(n) \tag{7.38}$$

and

$$\mathbf{w}(n) = \mathbf{w}_\mathrm{r}(n) + j\mathbf{w}_\mathrm{i}(n), \tag{7.39}$$

where the subscripts r and i denote the real and imaginary, respectively.

The complex output $y(n)$ is computed as

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n), \tag{7.40}$$

where all multiplications and additions are complex operations. The complex LMS algorithm adapts the real and imaginary parts of $\mathbf{w}(n)$ simultaneously, and is expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n)\mathbf{x}^*(n), \tag{7.41}$$

where * denotes a complex conjugate such that $\mathbf{x}^*(n) = \mathbf{x}_\mathrm{r}(n) - j\ \mathbf{x}_\mathrm{i}(n)$. An example of decomposing complex calculations into real-number operations can be found in Section 7.6.7 for adaptive channel equalizer.

7.3 Performance Analysis

In this section, we briefly discuss important properties of the LMS algorithm such as stability, convergence rate, and excess MSE due to gradient estimation.

7.3.1 Stability Constraint

As shown in Figure 7.5, the LMS algorithm involves the presence of feedback. Thus, the algorithm is subject to the possibility of becoming unstable. From Equation (7.30), we observe that the parameter μ determines the step size of correction applied to the weight vector. The convergence of the LMS algorithm must satisfy

$$0 < \mu < \frac{2}{\lambda_{\max}}, \tag{7.42}$$

where $\lambda_{\max}$ is the largest eigenvalue of the autocorrelation matrix $\mathbf{R}$ defined in Equation (7.22).

The computation of $\lambda_{\max}$ is difficult when L is large. In practical applications, it is desirable to estimate $\lambda_{\max}$ using a simple method. From Equation (7.22), we have

$$\lambda_{\max} \le \sum_{l=0}^{L-1} \lambda_l = Lr_{xx}(0) = LP_x, \tag{7.43}$$

where

$$P_x \equiv r_{xx}(0) = E\left[x^2(n)\right] \tag{7.44}$$

denotes the power of $x(n)$. Therefore, setting

$$0 < \mu < \frac{2}{LP_x} \tag{7.45}$$

assures that Equation (7.42) is satisfied.

Equation (7.45) provides important information on selecting μ:

1. The upper bound on μ is inversely proportional to filter length L, thus a small μ is used for a higher order filter.

2. Since μ is inversely proportional to the input signal power, low-power signals can use larger μ. We can normalize μ with respect to P_x for choosing step size that is independent of signal power. The resulting algorithm is called the normalized LMS (NLMS) algorithm, which will be discussed later.

7.3.2 Convergence Speed

Convergence of the weight vector $\mathbf{w}(n)$ from $\mathbf{w}(0)$ to $\mathbf{w}^\circ$ corresponds to the convergence of the MSE from $\xi(0)$ to ξ_{min}. Therefore, convergence of the MSE toward its minimum value is a commonly used performance measurement in adaptive systems because of its simplicity. A plot of the MSE versus time n is referred to as the learning curve. Since the MSE is the performance criterion of the LMS algorithms, the learning curve is a natural way to describe the transient behavior.

Each adaptive mode has its own time constant, which is determined by μ and the eigenvalue λ_l associated with that mode. Thus, the overall convergence is clearly limited by the slowest mode, and can be approximated as

$$\tau_{\text{mse}} \cong \frac{1}{\mu\lambda_{\text{min}}}, \tag{7.46}$$

where λ_{min} is the minimum eigenvalue of the $\mathbf{R}$ matrix. Because τ_{mse} is inversely proportional to μ, we have a large τ_{mse} (slow convergence) when μ is small. The maximum time constant $\tau_{\text{mse}} = 1/\mu\lambda_{\text{min}}$ is a conservative estimate in practical applications since only large eigenvalues will exert significant influence on the convergence time.

If λ_{max} is very large, only a small μ can satisfy the stability constraint. If λ_{min} is very small, the time constant can be very large, resulting in very slow convergence. The slowest convergence occurs for $\mu = 1/\lambda_{\text{max}}$. Substituting this smallest step size into Equation (7.46) results in

$$\tau_{\text{mse}} \le \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}}. \tag{7.47}$$

Therefore, the speed of convergence is dependent on the ratio of the maximum to minimum eigenvalues of the matrix $\mathbf{R}$.

The eigenvalues λ_{max} and λ_{min} are very difficult to compute if the order of filter is high. An efficient way is to approximate the eigenvalue spread by the spectral dynamic range expressed as

$$\frac{\lambda_{\text{max}}}{\lambda_{\text{min}}} \le \frac{\max |X(\omega)|^2}{\min |X(\omega)|^2}, \tag{7.48}$$

where $X(\omega)$ is DTFT of $x(n)$. Therefore, an input signal with a flat spectrum such as a white noise will have the fast convergence speed.

7.3.3 Excess Mean-Square Error

The steepest-descent algorithm defined in Equation (7.26) requires knowledge of the true gradient $\nabla\xi(n)$, which must be estimated for each iteration. After the algorithm converges, the gradient $\nabla\xi(n) = \mathbf{0}$;

however, the gradient estimator $\nabla\hat{\xi}(n) \neq \mathbf{0}$. As indicated by Equation (7.26), this will cause $\mathbf{w}(n)$ to vary randomly around $\mathbf{w}^{\circ}$, thus producing excess noise at the filter output. The excess MSE, which is caused by random noise in the weight vector after convergence, can be approximated as

$$\xi_{\text{excess}} \approx \frac{\mu}{2} L P_x \xi_{\min}. \tag{7.49}$$

This approximation shows that the excess MSE is directly proportional to μ. The larger step size μ results in faster convergence at the cost of steady-state performance. Therefore, there is a design trade-off between the excess MSE and the speed of convergence for determining μ.

The optimal step size μ is difficult to determine. Improper selection of μ might make the convergence speed unnecessarily slow or introduce more excess MSE in steady state. If the signal is nonstationary and real-time tracking capability is crucial for a given application, we may choose a larger μ. If the signal is stationary and convergence speed is not important, we can use a smaller μ to achieve better steady-state performance. In some practical applications, we can use a larger μ at the beginning of the operation for faster convergence, and then change to smaller μ to achieve better steady-state performance.

The excess MSE, ξ_{excess}, expressed in Equation (7.49) is also proportional to the filter length L, which means that a larger L results in higher algorithm noise. From Equation (7.45), a larger L implies that a smaller μ is required, thus resulting in slower convergence. On the other hand, a large L also implies better filter characteristics. Again, there exists an optimum filter length L for a given application.

7.3.4 Normalized LMS Algorithm

The stability, convergence speed, and fluctuation of the LMS algorithm are governed by the step size μ and the input signal power. As shown in Equation (7.45), the maximum stable step size μ is inversely proportional to the filter length L and the signal power. One important technique to optimize the speed of convergence while maintaining the desired steady-state performance is the NLMS:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu(n)\mathbf{x}(n)e(n), \tag{7.50}$$

where $\mu(n)$ is a normalized step size that is computed as

$$\mu(n) = \frac{\alpha}{L\hat{P}_x(n)}, \tag{7.51}$$

where $\hat{P}_x(n)$ is an estimate of the power of $x(n)$ at time n, and $0 < \alpha < 2$ is a constant.

Some useful implementation considerations are given as follows:

1. Choose $\hat{P}_x(0)$ as the best *a priori* estimate of the input signal power.
2. A software constraint is required to ensure that $\mu(n)$ is bounded if $\hat{P}_x(n)$ is very small when the signal is absent.

7.4 Implementation Considerations

In many real-world applications, adaptive filters are implemented on fixed-point processors. It is important to understand the finite wordlength effects of adaptive filters in meeting design specifications.

7.4.1 Computational Issues

The coefficient update defined in Equation (7.33) requires $L+1$ multiplications and L additions if we multiply $\mu^*e(n)$ outside the loop. Given the input vector $\mathbf{x}(n)$ stored in the array `x[ ]`, the error signal `en`, the weight vector `w[ ]`, the step size `mu`, and the filter length `L`, Equation (7.33) can be implemented in C language as follows:

```
uen=mu*en;                    // u*e(n) outside the loop
for (l=0; l<L; l++)           // l=0, 1,..., L-1
{
    w[l] += uen*x[l];         // LMS update
}
```

The architecture of most DSP processors has been optimized for convolution operations to compute filter output $y(n)$ given in Equation (7.31). However, the weight update operations in Equation (7.33) cannot take the advantage of this special architecture because each update involves loading the weight value into the accumulator, performing a multiply–add operation, and storing the result back into memory. We can reduce the computational complexity by skipping part of the weight update. In this case, the update is performed only for a portion of filter coefficients in one sampling period. The update for remaining portion may be at the following sampling periods. The computation complexity reduction is traded with the cost of somewhat slower convergence.

In some practical applications, the desired signal $d(n)$, and thus the error signal $e(n)$, is not available until several sampling intervals later. In addition, in the implementation of adaptive filters using DSP processors with pipeline architecture, the computational delay is an inherent problem. The delayed LMS algorithm can be expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n-\Delta)\mathbf{x}(n-\Delta). \tag{7.52}$$

The delay in the coefficient adaptation has only a slight influence on the steady-state behavior of the LMS algorithm. The delayed LMS algorithm with delay $\Delta = 1$ is widely used in implementing the adaptive FIR filtering on the DSP processors with pipeline architecture.

7.4.2 Finite-Precision Effects

This section analyzes finite-precision effects in adaptive filters using fixed-point arithmetic and presents methods for confining these effects to the acceptable levels. We assume that the input data samples are properly scaled so that their values lie between -1 and 1. As introduced in Chapter 3, the techniques used to inhibit the probability of overflow are scaling, saturation arithmetic, and guard bits. For adaptive filters, the feedback path makes scaling far more complicated. The dynamic range of the filter output is determined by the time-varying filter coefficients, which are unknown at the design stage.

For the adaptive FIR filter with the LMS algorithm, the scaling of the filter output and coefficients can be achieved by scaling the 'desired' signal, $d(n)$. The scale factor α, where $0 < \alpha \leq 1$, is used to prevent overflow of the filter coefficients during the weight update. Reducing the magnitude of $d(n)$ reduces the gain demand on the filter, thereby reducing the magnitude of the weight values. Since α only scales the desired signal, it does not affect the rate of convergence, which depends on the input signal $x(n)$.

With rounding operations, the finite-precision LMS algorithm can be described as follows:

$$y(n) = R\left[\sum_{l=0}^{L-1} w_l(n)x(n-l)\right] \tag{7.53}$$

$$e(n) = R[\alpha d(n) - y(n)] \tag{7.54}$$

$$w_l(n+1) = R\left[w_l(n) + \mu\, x(n-l)e(n)\right], \quad l = 0, 1, \ldots, L-1, \tag{7.55}$$

where $R[x]$ denotes the fixed-point rounding of the quantity x.

When updating weights according to Equation (7.55), the product $\mu x(n-l)e(n)$ produces a double-precision number, which is added to the original stored weight value, $w_l(n)$, then is rounded to form the updated value, $w_l(n+1)$.

The power of the roundoff noise is dominated by the error in quantizing filter coefficients, which is inversely proportional to the step size μ. Although a small value of μ reduces the excess MSE discussed in Section 7.3.3, it may result in a large quantization error. There is still another factor to consider in the selection of step size μ. As mentioned in Section 7.2, the adaptive algorithm is aimed at minimizing the error signal, $e(n)$. As the weight vector converges, the error signal decreases. The LMS algorithm modifies the current parameter settings by adding a correction term, $R\left[\mu\, x(n-l)e(n)\right]$. Adaptation will stop because this update term will be rounded to zero when the correction term is smaller in magnitude than the LSB. This phenomenon is known as 'stalling' or 'lockup'. This problem can be solved by using sufficient number of bits, and/or using a large step size μ, which still guarantees convergence of the algorithm. However, this will increase excess MSE.

We may use the leaky LMS algorithm to reduce numeric errors accumulated in the filter coefficients. The leaky LMS algorithm prevents overflow in finite-precision implementation by providing a compromise between minimizing the MSE and constraining the energy of the adaptive filter. The leaky LMS algorithm can be expressed as

$$\mathbf{w}(n+1) = \nu\mathbf{w}(n) + \mu\,\mathbf{x}(n)e(n), \tag{7.56}$$

where ν is the leakage factor with $0 < \nu \leq 1$. The leaky LMS algorithm not only prevents unconstrained weight overflow, but also limits the output power in order to avoid nonlinear distortion.

It can be shown that leakage is equivalent to adding low-level white noise. Therefore, this approach results in some degradation in adaptive filter performance. The value of the leakage factor is determined as a compromise between robustness and loss of performance. The excess error power due to the leakage is proportional to $[(1-\nu)/\mu]^2$. Therefore, $(1-\nu)$ should be kept smaller than in order to maintain an acceptable level of performance.

7.4.3 MATLAB Implementations

MATLAB *Filter Design Toolbox* provides a function `adaptfilt` to support adaptive filtering. The syntax of this function is

```
h = adaptfilt.algorithm(input1,input2,...)
```

This function returns an adaptive filter object `h` that uses the adaptive filtering technique specified by `algorithm`. The algorithm string determines which adaptive filter algorithm the `adaptfilt` object implements. The LMS-type algorithms are summarized in Table 7.1. The adaptive FIR filter objects use different LMS algorithms to determine filter coefficients. For example,

```
h = adaptfilt.lms(l,stepsize,leakage,coeffs,states)
```

constructs an adaptive FIR filter `h` with the LMS algorithm. The input parameter `l` is the filter length L; `stepsize` is the step size μ, a nonnegative scalar (defaults to 0.1); `leakage` is the leakage factor, which must be a scalar between 0 and 1 (defaults to 1 providing no leakage). If the leakage factor is less than 1, the leaky LMS algorithm is implemented. The input vector `coeffs` is the initial filter coefficient

Table 7.1 Adaptive FIR filter objects with various LMS algorithms

Object.Algorithm	Description
`adaptfilt.lms`	Direct-form, LMS algorithm
`adaptfilt.sd`	Direct-form, sign-data LMS algorithm
`adaptfilt.se`	Direct-form, sign-error LMS algorithm
`adaptfilt.ss`	Direct-form, sign-sign LMS algorithm
`adaptfilt.nlms`	Direct-form, normalized LMS algorithm
`adaptfilt.dlms`	Direct-form, delayed LMS algorithm
`adaptfilt.blms`	Block-form, LMS algorithm

with default to all zeros, and `states` vector consists of initial filter states. It defaults to a vector of all zeros. Some default parameters can be changed with

```
set(h,paramname,paramval)
```

Example 7.8: Given the primary signal $x(n)$ as normally distributed random numbers. This signal is filtered by an FIR filter with coefficient vector $\mathbf{b} = \{0.1, 0.2, 0.4, 0.2, 0.1\}$ to generate the desired signal $d(n)$. The following MATLAB script (`example7_8.m`, adapted from the MATLAB **Help** menu) implements an adaptive FIR filter with the LMS algorithm shown in Figure 7.5:

```
x  = randn(1,128);             % Primary signal x(n)
b  = [0.1,0.2,0.4,0.2,0.1];    % A system to be identified
d  = filter(b,1,x);            % Desired signal d(n)
mu = 0.05;                     % Step size mu
h  = adaptfilt.lms(5,mu);      % LMS algorithm
[y,e] = filter(h,x,d);         % Adaptive filtering
plot(1:128,[d;y;e]);           % Plot d(n), y(n), and e(n)
```

In the code, the filter length is $L = 5$ and the step size is $\mu = 0.05$. The desired signal $d(n)$, output signal $y(n)$, and error signal $e(n)$ are plotted in Figure 7.6. It shows that the filter output $y(n)$ is gradually approximated to $d(n)$, thus the difference (error) signal $e(n)$ is converged to zero in about 80 iterations.

In `example7_8.m`, we use the adaptive filtering syntax

```
[y,e] = filter(h,x,d);
```

which filters the input vector `x` through an adaptive filter object `h`, uses `d` for the desired signal, produces the output vector `y` and the error vector `e`. The vectors `x`, `d`, and `y` must have the same length.

We can use the function `maxstep` (defaults to 0) to determine a reasonable range of step size values for the signals being processed. The syntax

```
mumax = maxstep(h,x);
```

predicts a bound on the step size to provide convergence of the mean values of the adaptive filter coefficients.

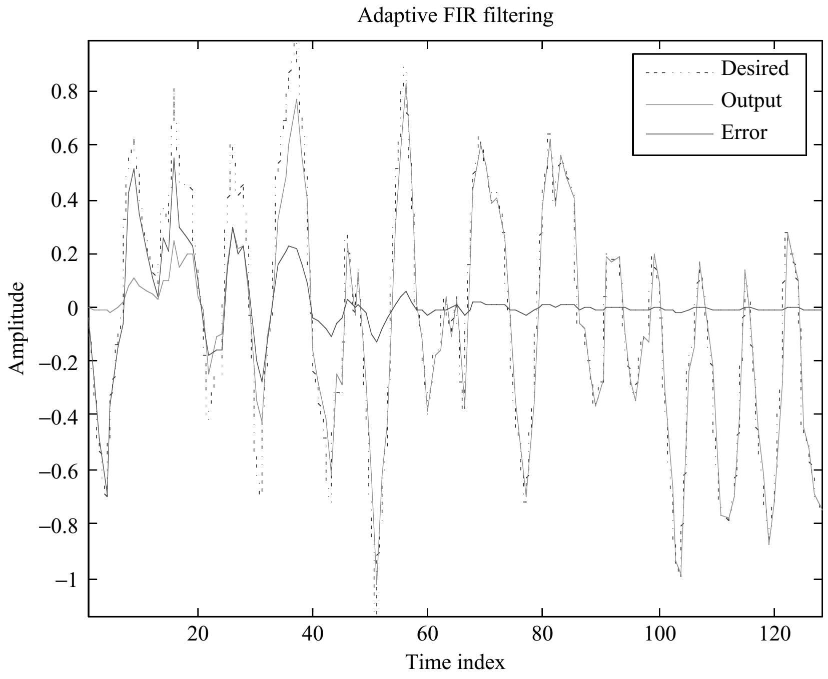


Figure 7.6 Performance of adaptive FIR filter with the LMS algorithm

7.5 Practical Applications

There are four classes of adaptive filtering applications: system identification, inverse modeling, prediction, and interference canceling. The essential difference among these applications is the configuration of signals $x(n)$, $d(n)$, $y(n)$, and $e(n)$.

7.5.1 Adaptive System Identification

System identification is an approach to model an unknown system. The paradigm of adaptive system identification is illustrated in Figure 7.7, where $P(z)$ is an unknown system to be identified by an adaptive filter $W(z)$. By exciting both $P(z)$ and $W(z)$ with the same excitation signal $x(n)$ and minimizing the difference of output signals $y(n)$ and $d(n)$, we can determine the characteristics of $P(z)$.

As shown in Figure 7.7, the estimation error is given as

$$\begin{aligned} e(n) &= d(n) - y(n) \\ &= \sum_{l=0}^{L-1} [p(l) - w_l(n)]\, x(n-l), \end{aligned} \tag{7.57}$$

where $p(l)$ is the impulse response of the unknown plant. By choosing each $w_l(n)$ close to each $p(l)$, the error will be minimized. For using white noise as the excitation signal, minimizing $e(n)$ will force the

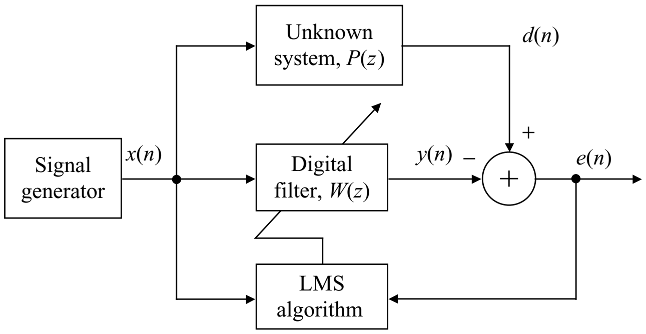


Figure 7.7 Block diagram of adaptive system identification using the LMS algorithm

$w_l(n)$ to approach $p(l)$, that is,

$$w_l(n) \approx p(l), \qquad l = 0, 1, \ldots, L-1. \tag{7.58}$$

When the difference between the physical system response $d(n)$ and the adaptive model response $y(n)$ has been minimized, the adaptive model approximates $P(z)$ from the input/output viewpoint. When the plant is time varying, the adaptive algorithm has the task of keeping the modeling error small by continually tracking time variations of the plant dynamics.

Example 7.9: Assume that the excitation signal $x(n)$ shown in Figure 7.7 is normally distributed random signal. This signal is applied to an unknown system $P(z)$ that is simulated by an FIR filter with coefficient vector $\mathbf{b} = \{0.05, –0.1, 0.15, –0.2, 0.25, –0.2, 0.15, –0.1, 0.05\}$. The MATLAB script (`example7_9.m`, adapted from the MATLAB **Help** menu) implements the adaptive system identification using the FIR filter of length $L = 9$ with the LMS algorithm. As shown in Figure 7.8, the adaptive filter coefficients are equal to the unknown FIR system's coefficients after the convergence of the algorithm. In this case, the adaptive model $W(z)$ exactly identifies the unknown system $P(z)$.

7.5.2 Adaptive Linear Prediction

Linear prediction estimates the values of signal at a future time. This technique has been successfully applied to a wide range of applications such as speech coding and separating signals from noise. As illustrated in Figure 7.9, the adaptive predictor consists of an adaptive filter in which the coefficients $w_l(n)$ are updated by the LMS algorithm. The predictor output $y(n)$ is expressed as

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n - \Delta - l), \tag{7.59}$$

where Δ is the number of delay samples. The coefficients are updated as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\, \mathbf{x}(n - \Delta)e(n), \tag{7.60}$$

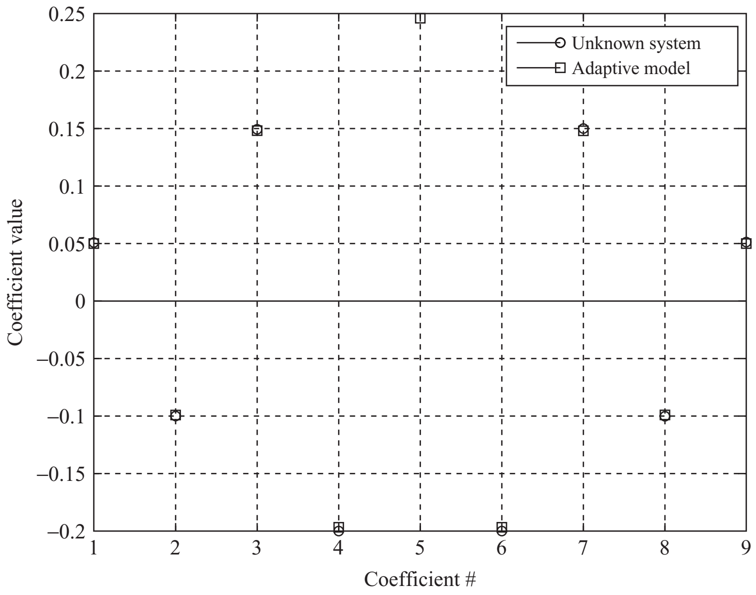


Figure 7.8 Converged adaptive filter coefficients

where $\mathbf{x}(n-\Delta) = [x(n-\Delta)\; x(n-\Delta-1) \ldots x(n-\Delta-L+1)]^T$ is the delayed reference signal vector, and $e(n) = x(n) - y(n)$ is the prediction error. Proper selection of the prediction delay Δ allows improved frequency estimation performance for multiple sinusoids in white noise.

Now consider the adaptive predictor for enhancing an input of M sinusoids embedded in white noise, which is of the form

$$\begin{aligned} x(n) &= s(n) + v(n) \\ &= \sum_{m=0}^{M-1} A_m \sin(\omega_m n + \phi_m) + v(n), \end{aligned} \tag{7.61}$$

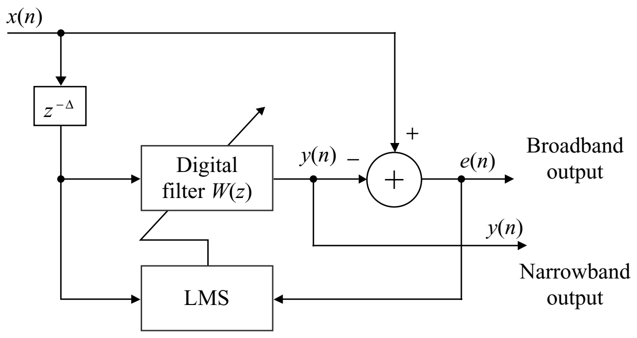


Figure 7.9 Block diagram of an adaptive predictor

where $v(n)$ is white noise with uniform noise power σ_v^2 . In this application, the structure shown in Figure 7.9 is called the adaptive line enhancer, which efficiently tracks the sinusoidal components in the received signal $x(n)$ and separates these narrowband signals $s(n)$ from broadband noise $v(n)$. This technique is very effective in practical applications when the signal and noise parameters are unknown and/or time varying.

As shown in Figure 7.9, we want the highly correlated components of $x(n)$ to appear in $y(n)$. This is accomplished by adjusting the weights to minimize the mean-square value of the error signal $e(n)$. This causes an adaptive filter $W(z)$ to form multiple bandpass filters centered at the frequencies of the sinusoidal components. The wideband noise component in the input is rejected, while the phase difference (caused by Δ) of the narrowband signals is readjusted so that they can cancel correlated components in $d(n)$ to minimize the error signal $e(n)$. In this case, the output $y(n)$ is the enhanced signal, which contains multiple sinusoids as expressed in Equation (7.61).

Example 7.10: Assume that the signal $x(n)$ shown in Figure 7.9 consists of desired sinewave that is corrupted by white noise. The adaptive line enhancer with $\Delta = 1$ can be used to decorrelate the white noise component, thus enhancing the sinewave. This example is implemented in MATLAB script `example7_10.m`. The enhanced output $y(n)$ and the error signal $e(n)$ are plotted in Figure 7.10. As shown in the figure, the error signal is gradually reduced to the broadband white noise, while the enhanced signal is converged to the desired sinewave.

In many digital communications and signal detection applications, the desired broadband signal $v(n)$ is corrupted by an additive narrowband interference $s(n)$. From a filtering viewpoint, the objective of an

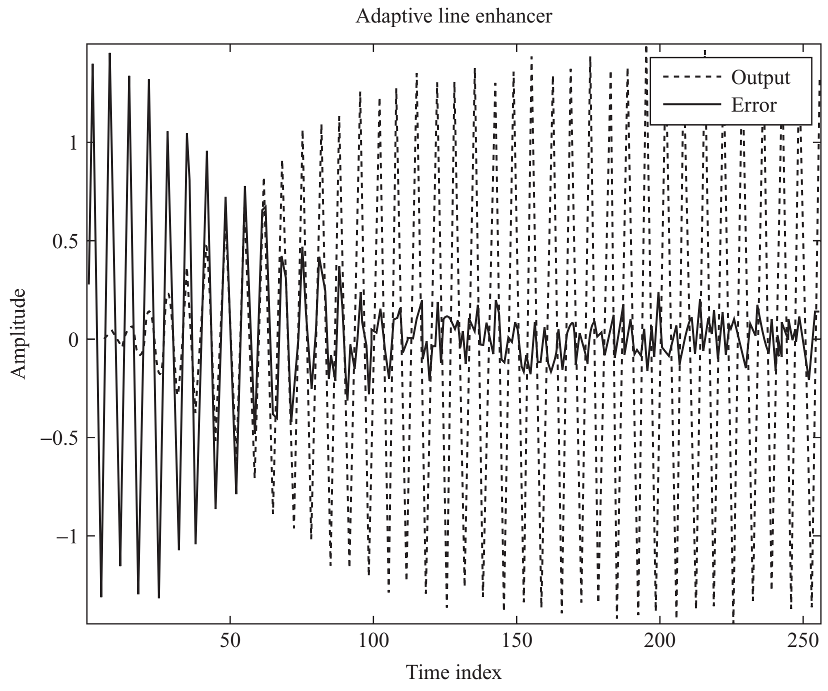


Figure 7.10 Performance of adaptive line enhancer

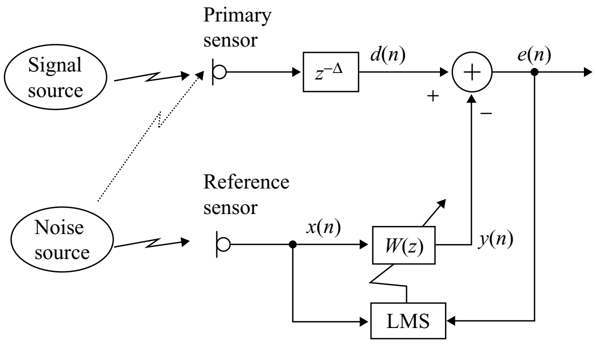


Figure 7.11 Basic concept of adaptive noise canceling

adaptive filter is to form a notch filter at the frequency of interference, thus suppressing the narrowband noise. The error signal $e(n)$ in Figure 7.9 consists of desired broadband signals. In this application, the desired output from the overall interference suppression filter is $e(n)$.

7.5.3 Adaptive Noise Cancelation

The widespread use of cellular phones has significantly increased the use of communication devices in high-noise environments. Intense background noise, however, often corrupts speech and degrades the performance of many communication systems. The widely used adaptive noise canceler employs an adaptive filter with the LMS algorithm to cancel the noise component embedded in the primary signal. As illustrated in Figure 7.11, the primary sensor is placed close to the signal source to pick up the desired signal. The reference sensor is placed close to the noise source to sense only the noise.

A block diagram of the adaptive noise cancelation system is illustrated in Figure 7.12, where $P(z)$ represents the transfer function between the noise source and the primary sensor. The canceler has two inputs: the primary input $d(n)$ and the reference input $x(n)$. The reference input $x(n)$ contains noise only. The primary input $d(n)$ consists of signal $s(n)$ plus noise $x'(n)$; i.e., $d(n) = s(n) + x'(n)$. The noise $x'(n)$ is highly correlated with $x(n)$ since they are derived from the same noise source. The objective of the adaptive filter is to use the reference input $x(n)$ to estimate the noise $x'(n)$. The filter output $y(n)$, which

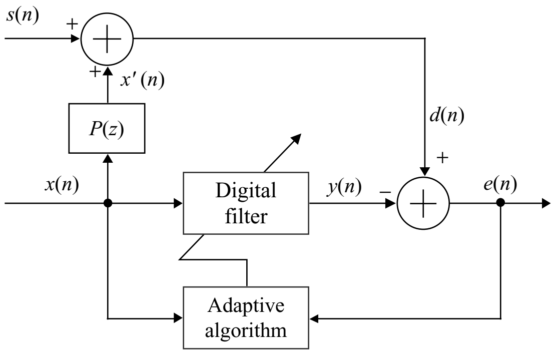


Figure 7.12 Block diagram of adaptive noise canceler

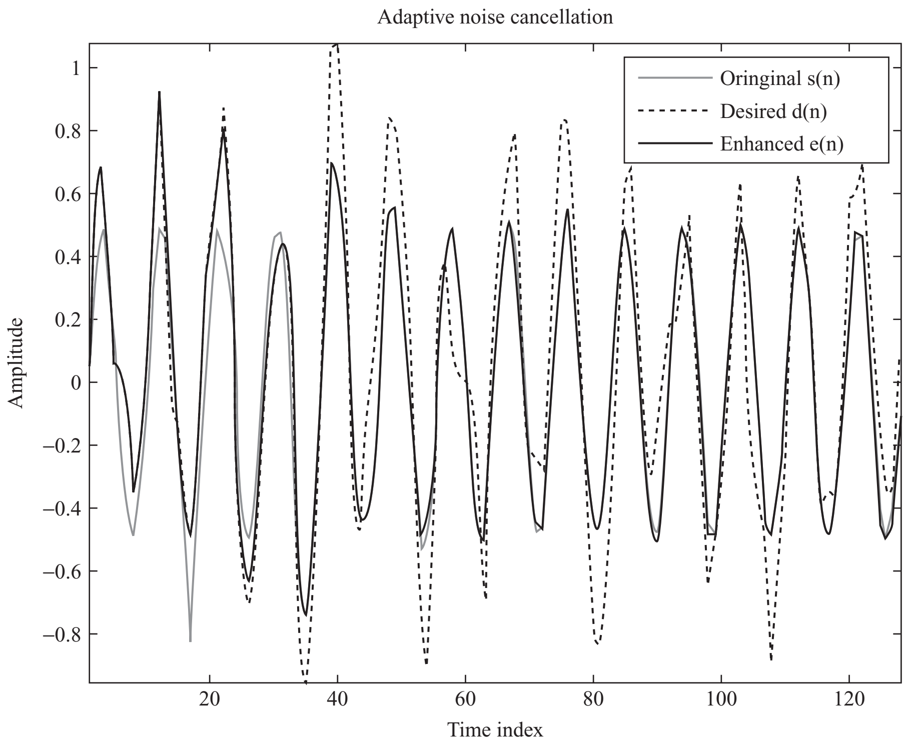


Figure 7.13 The enhanced sinewave given in $e(n)$ approaches to the original $s(n)$

is an estimate of noise $x'(n)$, is then subtracted from the primary channel signal $d(n)$, producing $e(n)$ as the desired signal plus reduced noise.

Example 7.11: As shown in Figure 7.12, assume $s(n)$ is a sinewave, $x(n)$ is a white noise, and $P(z)$ is a simple FIR system. We use the adaptive FIR filter with the LMS algorithm for noise cancelation, which is implemented in the MATLAB script `example7_11.m`. The adaptive filter will approximate $P(z)$, and thus its output $y(n)$ will converge to x′(n) in order to cancel it. Therefore, the error signal $e(n)$ will gradually approach the desired sinewave $s(n)$, as shown in Figure 7.13.

To apply the adaptive noise cancelation effectively, the reference noise picked up by the reference sensor must be highly correlated with the noise components in the primary signal. This condition requires a close spacing between the primary and reference sensors. Unfortunately, it is also critical to avoid the signal components from the signal source being picked up by the reference sensor. This ‘crosstalk’ effect will degrade the performance of adaptive noise cancelation because the presence of the signal components in reference signal will cause the adaptive noise cancelation to cancel the desired signal along with the undesired noise.

Crosstalk problem may be eliminated by placing the primary sensor far away from the reference sensor. Unfortunately, this arrangement requires a large-order filter in order to obtain adequate noise reduction. Furthermore, it is not always feasible to place the reference sensor far away from the signal source. The second method for reducing crosstalk is to place an acoustic barrier (oxygen masks used by pilots in aircraft cockpit, for example) between the primary and reference sensors. However, many applications do not allow an acoustic barrier between sensors, and a barrier may reduce the correlation of the noise component in the primary and reference signals. The third technique is to control the adaptive algorithm to update filter coefficients only during the silent intervals in the speech. Unfortunately, this method depends on a reliable speech activity detector that is very application dependent. This technique also fails to track the environment changes during the speech periods. In recent years, microphone array techniques are used to improve the performance of the noise cancelation.

7.5.4 Adaptive Notch Filters

In certain situations, the primary input is a broadband signal corrupted by undesired narrowband (sinusoidal) interference. The conventional method of eliminating such sinusoidal interference is using a notch filter that is tuned to the frequency of the interference. To design the filter, we need the precise frequency of the interference. The adaptive notch filter has the capability to track the frequency of the interference, and thus is especially useful when the interfering sinusoid drifts in frequency.

A single-frequency adaptive notch filter with two adaptive weights is illustrated in Figure 7.14. The input signal is a cosine signal

$$x(n) = x_0(n) = A\cos(\omega_0 n). \tag{7.62}$$

A 90° phase shifter is used to produce the quadrature signal

$$x_1(n) = A\sin(\omega_0 n). \tag{7.63}$$

For a sinusoidal signal, two filter coefficients are needed.

The LMS algorithm employed in Figure 7.14 is summarized as

$$y(n) = w_0(n)x_0(n) + w_1(n)x_1(n). \tag{7.64}$$

The reference input is used to estimate the composite sinusoidal interfering signal contained in the primary input $d(n)$. The center frequency of the notch filter is equal to the frequency of the primary

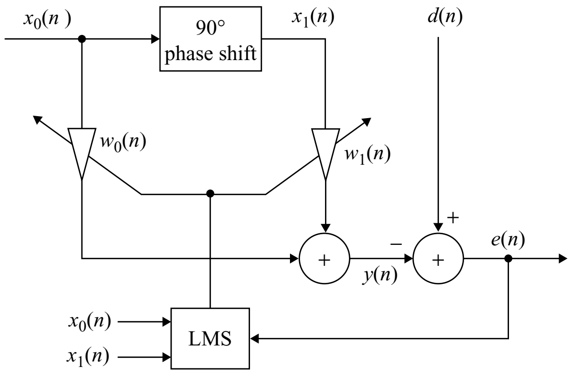


Figure 7.14 Single-frequency adaptive notch filter

sinusoidal noise. Therefore, the noise at that frequency is attenuated. This adaptive notch filter provides a simple method for eliminating sinusoidal interference.

Example 7.12: For a stationary input and sufficiently small μ, the convergence speed of the LMS algorithm is dependent on the eigenvalue spread of the input autocorrelation matrix. For $L = 2$ and the reference input given in Equation (7.62), the autocorrelation matrix can be expressed as

$$\begin{aligned}\mathbf{R} &= E\begin{bmatrix} x_0(n)x_0(n) & x_0(n)x_1(n) \\ x_1(n)x_0(n) & x_1(n)x_1(n) \end{bmatrix} \\ &= E\begin{bmatrix} A^2\cos^2(\omega_0 n) & A^2\cos(\omega_0 n)\sin(\omega_0 n) \\ A^2\sin(\omega_0 n)\cos(\omega_0 n) & A^2\sin^2(\omega_0 n) \end{bmatrix} \\ &= \begin{bmatrix} A^2/2 & 0 \\ 0 & A^2/2 \end{bmatrix}.\end{aligned}$$

This equation shows that because of the 90° phase shift, $x_0(n)$ is orthogonal to $x_1(n)$ and the off-diagonal terms in the $\mathbf{R}$ matrix is zero. The eigenvalues λ_1 and λ_2 of the $\mathbf{R}$ matrix are identical and equal to $A^2/2$. Therefore, the system has very fast convergence since the eigenvalue spread equals 1. The time constant of the adaptation is approximated as

$$\tau_{\text{mse}} \leq \frac{1}{\mu\lambda} = \frac{2}{\mu A^2},$$

which is determined by the power of the reference sinewave and the step size μ.

7.5.5 Adaptive Channel Equalization

In digital communications, the transmission of high-speed data through a channel is limited by intersymbol interference caused by distortion in the transmission channel. High-speed data transmission through channels with severe distortion can be achieved in several ways, such as (1) by designing the transmit and receive filters so that the combination of filters and channel results in an acceptable error from the combination of intersymbol interference and noise; and (2) by designing an equalizer in the receiver that counteracts the channel distortion. The second method is the most commonly used technology for data transmission applications.

As illustrated in Figure 7.15, the received signal $y(n)$ is different from the original signal $x(n)$ because it was distorted by the overall channel transfer function $C(z)$, which includes the transmit filter, the transmission medium, and the receive filter. To recover the original signal $x(n)$, we need to process $y(n)$ using the equalizer $W(z)$, which is the inverse of the channel's transfer function $C(z)$ in order to compensate for the channel distortion. That is, we have to design the equalizer

$$W(z) = \frac{1}{C(z)}, \tag{7.65}$$

i.e., $C(z)W(z) = 1$ such that $\hat{x}(n) = x(n)$.

In practice, the telephone channel is time varying and is unknown in the design stage due to variations in the transmission medium. Thus, we need an adaptive equalizer that provides precise compensation over the time-varying channel. As shown in Figure 7.15, an adaptive filter requires the desired signal $d(n)$ for computing the error signal $e(n)$ for the LMS algorithm. In theory, the delayed version of the transmitted signal $x(n - \Delta)$ is the desired response for the adaptive equalizer $W(z)$. However, since the adaptive filter

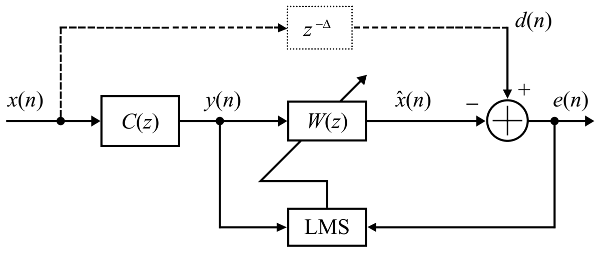


Figure 7.15 Cascade of channel with an ideal adaptive channel equalizer

is located in the receiver, the desired signal generated by the transmitter is not available at the receiver. The desired signal may be generated locally in the receiver using two methods. During the training stage, the adaptive equalizer coefficients are adjusted by transmitting a short training sequence. This known transmitted sequence is also generated in the receiver and is used as the desired signal $d(n)$ for the LMS algorithm. After the short training period, the transmitter begins to transmit the data sequence. In the data mode, the output of the equalizer $\hat{x}(n)$ is used by a decision device (slicer) to produce binary data. Assuming that the output of the decision device is correct, the binary sequence can be used as the desired signal $d(n)$ to generate the error signal for the LMS algorithm.

Example 7.13: The adaptive channel equalizer shown in Figure 7.15 is implemented using MATLAB script `example7_13.m`. We used a simple FIR filter to simulate the channel $C(z)$, and the adaptive FIR filter with the LMS algorithm as equalizer. The delay used to generate $d(n)$ is half of the filter length of $W(z)$, that is, $L/2$. As shown in Figure 7.16, the error signal $e(n)$ is minimized such that the adaptive filter approximates the inverse of channel.

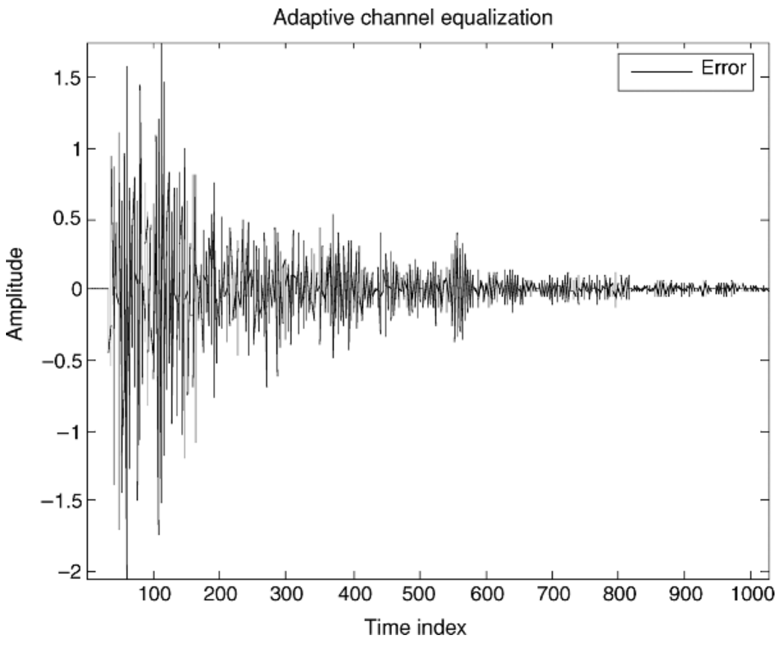


Figure 7.16 The error signal $e(n)$ is minimized after convergence of adaptive channel equalizer

MATLAB *Communications Toolbox* provides many functions to support equalizers: `dfe` constructs a decision feedback equalizer object; `equalize` equalizes signal using an equalizer object; `lineareq` constructs a linear equalizer object; `mlseeq` equalizes linearly modulated signal using Viterbi algorithm; and `reset(equalizer)` resets equalizer object.

7.6 Experiments and Program Examples

We will conduct adaptive filtering experiments in this section using adaptive FIR filters based on the LMS-type algorithms.

7.6.1 Floating-Point C Implementation

The block diagram of adaptive filter with the LMS is shown in Figure 7.5. The floating-point C implementation of the adaptive filter with the LMS algorithm is listed in Table 7.2.

Table 7.2 C implementation of adaptive filter with the LMS algorithm

```
void float_lms(LMS *lmsObj)
{
    LMS *lms=(LMS *)lmsObj;
    double *w,*x,y,ue;
    short j,n;

    n = lms->order;
    w = &lms->w[0];
    x = &lms->x[0];

    // Update signal buffer
    for(j=n-1; j>0; j--)
    {
        x[j] = x[j-1];
    }
    x[0] = lms->in;
    // Compute filter output - Equation (7.31)
    y = 0.0;
    for(j=0; j<n; j++)
    {
        y += w[j] * x[j];
    }
    lms->out = v;
    // Compute error signal - Equation (7.32)
    lms->err = lms->des - y;
    // Coefficients update - Equation (7.33)
    ue = lms->err * lms->mu;
    for(j=0; j<n ; j++)
    {
        w[j] += ue * x[j];
    }
 }
```

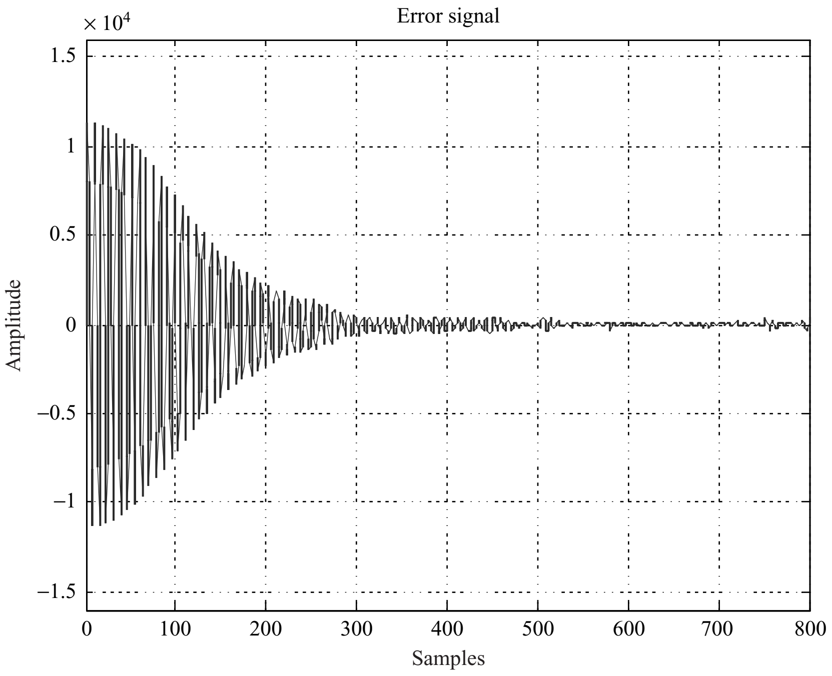


Figure 7.17 Error signal of the adaptive filter in floating-point C implementation

The input signal $x(n)$ is a sinewave corrupted by a white noise. The desired signal $d(n)$ is also a sinewave. The noise is removed from the input data file. This experiment uses an adaptive FIR filter with length 128 and step size 0.005. The adaptive filter reaches the steady state after 500 iterations as shown in Figure 7.17. Table 7.3 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Open the project file, `float_lms.pjt`, and rebuild the project.

2. Run the experiment using the data files `input.pcm` and `desired.pcm`.

3. Play both the input and output data files and compare the results.

4. For a given filter length, change the step size and plot the error signal to evaluate the convergence speed with different step sizes.

Table 7.3 File listing for experiment `exp7.6.1_floatingPoint_LMS`

Files	Description
`float_lmsTest.c`	C function for testing LMS adaptive filter
`float_lms.c`	C function for floating-point LMS algorithm
`float_lms.h`	C header file
`float_lms.pjt`	DSP project file
`float_lms.cmd`	DSP linker command file
`input.pcm`	Input signal file
`desired.pcm`	Desired signal file

5. Fix the step size that achieves the fastest convergence speed, change the filter length to observe the excess MSE in steady state and find the proper filter length for the chosen step size that generates the lowest MSE.

6. Verify that the selected step size and filter length are the optimum by plotting the error signals similar to Figure 7.17.

7.6.2 Fixed-Point C Implementation of Leaky LMS Algorithm

In this experiment, we use Q15 format for fixed-point C implementation of the leaky LMS algorithm defined in Equation (7.56). The $x(n)$ and $d(n)$ used for this experiment are the same as the previous experiment. When the rounding is not considered in the implementation, the adaptive filter may diverge as shown in Figure 7.18. The use of the leaky LMS algorithm can improve the stability, but a smaller leaky factor can also result in higher steady-state error level. Figure 7.19 shows the result of using leaky factor of 0.99. When we choose the leaky factor of 0.999, the MSE is close to the floating-point C implementation as shown in Figure 7.17. The fixed-point C implementation is listed in Table 7.4. The files used for this experiment are listed in Table 7.5.

Procedures of the experiment are listed as follows:

1. Open the project file, `fixPoint_leaky_lms.pjt`, and rebuild the project.

2. Run the leaky LMS algorithm experiment using the data files `input.pcm` and `desired.pcm`.

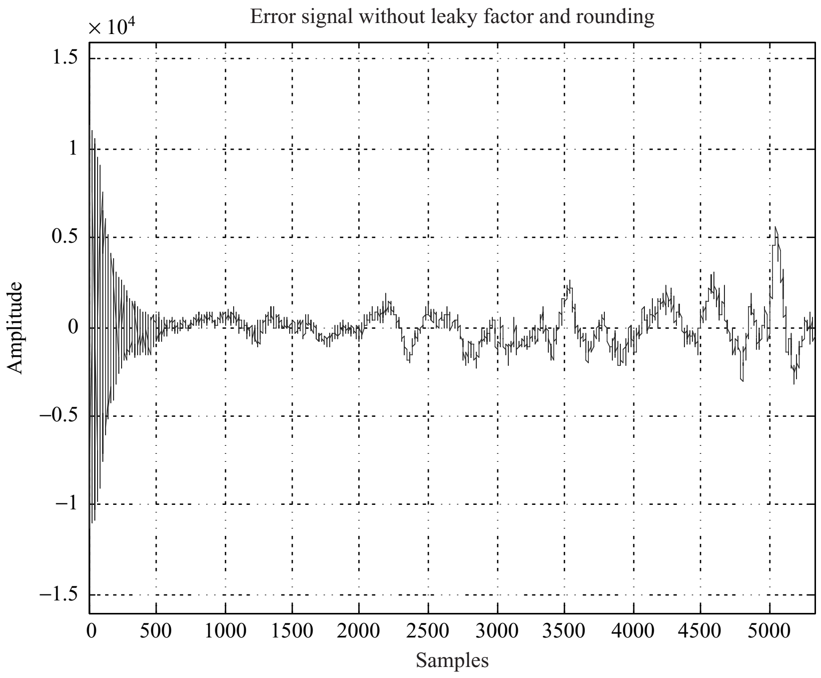


Figure 7.18 Error signal of fixed-point LMS algorithm without rounding and leaky factor

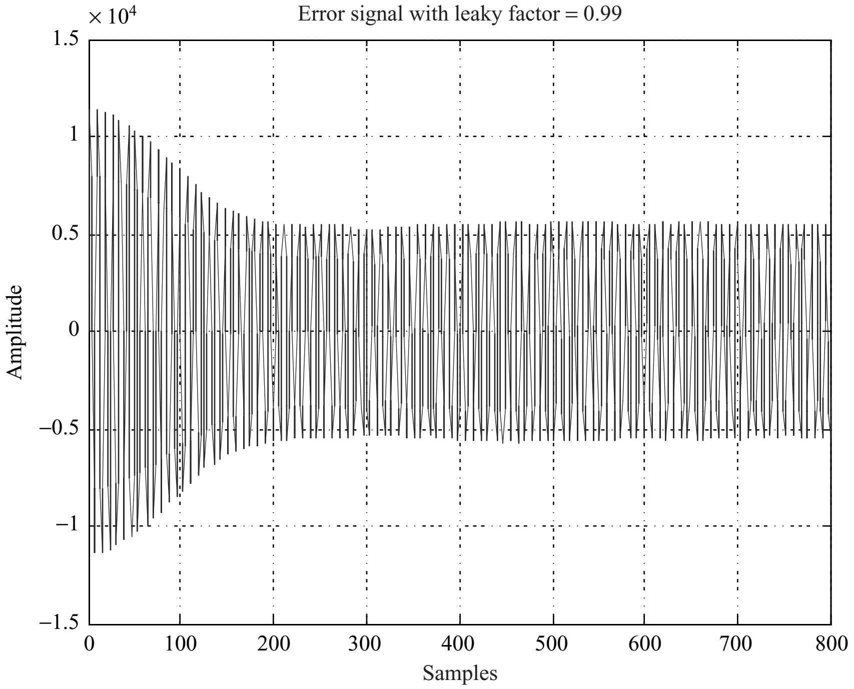


Figure 7.19 Fixed-point leaky LMS algorithm with leaky factor = 0.99

3. Compare the fixed-point C results with the floating-point C results obtained in previous experiment in terms of convergence speed and steady-state MSE.

4. To evaluate the finite wordlength effects of the fixed-point implementation, remove rounding by setting the defined constant, `ROUND`, to 0 in the header file and rerun the project. Display the error signal to see if the adaptive filter has diverged. If the LMS algorithm is diverged, identify the key positions where rounding is necessary to stabilize the fixed-point LMS algorithm.

5. With rounding enabled, adjust the leaky factor, `LEAKY`, in the header file to find the largest possible step size that provides the fastest convergence and lowest excess MSE. Will this experiment reach the similar performance as the floating-point C implementation in previous experiment?

6. Profile the fixed-point C implementation of the leaky LMS algorithm. How many cycles per data sample are required?

7.6.3 ETSI Implementation of NLMS Algorithm

In this experiment, we will introduce the ETSI (European Telecommunications Standard Institute) operators (functions) provided by the C55x compiler. These ETSI operators are very useful for developing DSP applications such as the GSM (global system for mobile communications) standards including speech coders. The original ETSI operators are fixed-point C functions. The C55x compiler supports

Table 7.4 C code for fixed-point leaky LMS algorithm

```
void fixPoint_leaky_lms(LMS *lmsObj)
{
   LMS   *lms=(LMS *)lmsObj;
   long  ue,temp32;
   short j,n;
   short *x,*w;

   n = lms->order;
   w = &lms->w[0];
   x = &lms->x[0];

   // Update data delay line
   for(j=n-1; j>0; j--)
   {
      x[j] = x[j-1];
   }
   x[0] = lms->in;
   // Get adaptive filter output - Equation (7.31)
   temp32 = (long)w[0] * x[0];
   for(j=1; j<n; j++)
   {
      temp32 += (long)w[j] * x[j];
   }
   lms->out = (short)((temp32+ROUND)>>15);
   // Compute error signal - Equation (7.32)
   lms->err = lms->des - lms->out;
   // Coefficients update - Equation (7.56)
   ue = (long)(((lms->err * (long)lms->mu)+ROUND)>>15);
   for(j=0; j<n ; j++)
   {
      temp32 = (((long)lms->leaky * w[j])+ROUND)>>15;
      w[j] = (short)temp32 + (short)(((ue * x[j])+ROUND)>>15);
   }
}
```

ETSI functions by mapping them directly to its intrinsics. Table 7.6 lists the ETSI operators and their corresponding intrinsics for the C55x.

The C55x implementation of the NLMS algorithm using ETSI operators is listed in Table 7.7. Table 7.8 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

Table 7.5 File listing for experiment `exp7.6.2_fixPoint_LeakyLMS`

Files	Description
`fixPoint_leaky_lmsTest.c`	C function for testing leaky LMS experiment
`fixPoint_leaky_lms.c`	C function for fixed-point leaky LMS algorithm
`fixPoint_leaky_lms.h`	C header file
`fixPoint_leaky_lms.pjt`	DSP project file
`fixPoint_leaky_lms.cmd`	DSP linker command file
`input.pcm`	Input signal file
`desired.pcm`	Desired signal file

Table 7.6 C55x ETSI functions and corresponding intrinsic functions

ETSI function	Intrinsics representation	Description
`L_add(a,b)`	`_lsadd((a),(b))`	Add two 32-bit integers with SATD set, producing a saturated 32-bit result.
`L_sub(a,b)`	`_lssub((a),(b))`	Subtract `b` from `a` with SATD set, producing a saturated 32-bit result.
`L_negate(a)`	`_lsneg(a)`	Negate the 32-bit value with saturation.`_lsneg` (0x80000000)=> 0x7FFFFFFF
`L_deposite_h(a)`	`(long)(a<<16)`	Deposit the 16-bit `a` into the 16 MSB of a 32-bit output and the 16 LSB of the output are zeros.
`L_deposite_l(a)`	`(long)a`	Deposit the 16-bit `a` into the 16 LSB of a 32-bit output and the 16 MSB of the output are sign extended.
`L_abs(a)`	`_labss((a))`	Create a saturated 32-bit absolute value.`_labss`(0x8000000)=> 0x7FFFFFFF (SATD is set.)
`L_mult(a,b)`	`_lsmpy((a),(b))`	Multiply `a` and `b` and shift the result left by 1. Produce a saturated 32-bit result. (SATD and FRCT are set.)
`L_mac(a,b,c)`	`_smac((a),(b),(c))`	Multiply `b` and `c`, shift the result left by 1, and add it to `a`. Produce a saturated 32-bit result. (SATD, SMUL, and FRCT are set.)
`L_macNs(a,b,c)`	`L_add_c((a),L_mult((b),(c)))`	Multiply `b` and `c`, shift the result left by 1, add the 32 bit result to `a` without saturation
`L_msu(a,b,c)`	`_smas((a),(b),(c))`	Multiply `b` and `c`, shift the result left by 1, and subtract it from `a`. Produce a 32-bit result. (SATD, SMUL, and FRCT are set.)
`L_msuNs(a,b,c)`	`L_sub_c((a),L_mult((b),(c)))`	Multiply `b` and `c`, shift the result left by 1, and subtract it from `a` without saturation.
`L_shl(a,b)`	`_lsshl((a),(b))`	Shift `a` to left by `b` and produce a 32-bit result. The result is saturated if `b` is less than or equal to 8. (SATD is set.)
`L_shr(a,b)`	`_lshrs((a),(b))`	Shift `a` to right by `b` and produce a 32-bit result. Produce a saturated 32-bit result. (SATD is set.)
`L_shr_r(a,b)`	`L_crshft_r((a),(b))`	Same as `L_shr(a,b)` but with rounding.
`abs_s(a)`	`_abss((a))`	Create a saturated 16-bit absolute value.`_abss` (0x8000)=> 0x7 FFF (SATA is set.)
`add(a,b)`	`_sadd((a),(b))`	Add two 16-bit integers with SATA set, producing a saturated 16-bit result.
`sub(a,b)`	`_ssub((a),(b))`	Subtract `b` from `a` with SATA set, producing a saturated 16-bit result.
`extract_h(a)`	`(unsigned short)((a)>>16)`	Extract the upper 16-bit of the 32-bit `a`.
`extract_l(a)`	`(short)a`	Extract the lower 16-bit of the 32-bit `a`.
`round(a)`	`(short)_rnd(a)>>16`	Round `a` by adding 2^{15}. Produce a 16-bit saturated result. (SATD is set.)
`mac_r(a,b,c)`	`(short)(_smacr((a), (b),(c))>>16)`	Multiply `b` and `c`, shift the result left by 1, add the result to `a`, and then round the result by adding 2^{15}. (SATD, SMUL, and FRCT are set.)

Table 7.6 (*continued*)

ETSI function	Intrinsics representation	Description
`msu_r(a,b,c)`	`(short)(_smasr((a),(b),(c))>>16)`	Multiply `b` and `c`, shift the result left by 1, subtract the result from `a`, and then round the result by adding 2^{15}. (SATD, SMUL, and FRCT are set.)
`mult_r(a,b)`	`(short)(_smpyr((a),(b))>>16)`	Multiply `a` and `b`, shift the result left by 1, and round by adding 2^{15} to the result. (SATD and FRCT are set.)
`mult(a,b)`	`_smpy((a),(b))`	Multiply `a` and `b` and shift the result left by 1. Produce a saturated 16-bit result. (SATD and FRCT are set.)
`norm_l(a)`	`_lnorm(a)`	Produce the number of left shifts needed to normalize `a`.
`norm_s(a)`	`_norm(a)`	Produce the number of left shifts needed to normalize `a`.
`negate(a)`	`_sneg(a)`	Negate the 16-bit value with saturation. `_sneg` (0xffff8000)=> 0x00007FFF
`shl(a,b)`	`_sshl((a),(b))`	Shift `a` to left by `b` and produce a 16-bit result. The result is saturated if `b` is less than or equal to 8. (SATD is set.)
`shr(a,b)`	`_shrs((a),(b))`	Shift `a` to right by `b` and produce a 16-bit result. Produce a saturated 16-bit result. (SATD is set.)
`shr_r(a,b)`	`crshft((a),(b))`	Same as `shr(a,b)` but with rounding.
`shift_r(a,b)`	`shr_r((a),-(b))`	Same as `shl(a,b)` but with rounding.
`div_s(a,b)`	`divs((a),(b))`	Produces a truncated positive 16-bit result which is the fractional integer division of `a` by `b`, `a` and `b` must be positive and `b` $\geq$ `a`.

1. Open the project file, `ETSI_nlms.pjt`, and rebuild the project.

2. Run the experiment using data files `input.pcm` and `desired.pcm`.

3. Compare the results of ETSI (intrinsics) implementation with the fixed-point C implementation in terms of convergence speed and steady-state MSE.

4. Profile the ETSI (intrinsics) implementation of the NLMS algorithm. How many cycles per data sample are needed using the intrinsics?

7.6.4 Assembly Language Implementation of Delayed LMS Algorithm

The TMS320C55x has a powerful assembly instruction, `LMS`, for implementing the delayed LMS algorithm. This instruction utilizes the high parallelism of the C55x architecture to perform the following

Table 7.7 Implementation of NLMS algorithm using C55x intrinsics

```
void intrinsic_nlms(LMS *lmsObj)
{
   LMS   *lms=(LMS *)lmsObj;
   long  temp32;
   short j,n,mu,ue,*x,*w;

   n = lms->order;
   w = &lms->w[0];
   x = &lms->x[0];

   // Update signal buffer
   for(j=n-1; j>0; j--)
   {
       x[j] = x[j-1];
   }
   x[0] = lms->in;
   // Compute normalized mu
   temp32 = mult_r(lms->x[0],lms->x[0]);
   temp32 = mult_r((short)temp32, ONE_MINUS_BETA);
   lms->power = mult_r(lms->power, BETA);
   temp32 = add(lms->power, (short)temp32);
   temp32 = add(lms->c, (short)temp32);
   mu = lms->alpha / (short)temp32;
   // Compute filter output - Equation (7.31)
   temp32 = L_mult(w[0], x[0]);
   for(j=1; j<n; j++)
   {
       temp32 = L_mac(temp32, w[j], x[j]);
   }
   lms->out = round(temp32);
   // Compute error signal - Equation (7.32)
   lms->err = sub(lms->des, lms->out);
   // Coefficients update - Equation (7.50)
   ue = mult_r(lms->err, mu);
   for(j=0; j<n ; j++)
   {
       w[j] = add(w[j], mult_r(ue, x[j]));
   }
}
```

Table 7.8 File listing for experiment `exp7.6.3_ETSI_NLMS`

Files	Description
`ETSI_nlmsTest.c`	C function for testing NLMS experiment
`ETSI_nlms.c`	C function for NLMS algorithm using ETSI operators
`ETSI_nlms.h`	C header file for experiment
`ETSI_nlms.pjt`	DSP project file
`ETSI_nlms.cmd`	DSP linker command file
`input.pcm`	Input signal file
`desired.pcm`	Desired signal file

two equations in one cycle:

$$e(n) = d(n-1) - y(n-1)$$
$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n-1)\mathbf{x}(n-1).$$

This LMS instruction effectively improves the run-time efficiency of the delayed LMS algorithm. The LMS instruction uses the previous error $e(n-1)$ and the previous signal vector $\mathbf{x}(n-1)$ to update the coefficient vector. Table 7.9 shows the C55x assembly language implementation of the delayed LMS algorithm using the LMS instruction.

Table 7.9 Implementation of the delayed LMS algorithm

```
_asm_dlms:
    pshboth XAR5                ; AR5 will be used as index into x[]
    pshboth XAR7                ; AR7 will be used pointer to e[]
    psh   T3                    ; T3 is needed for LMS instruction
;
; Set up C55x processor for the Q15 format with overflow
;
    mov   #0,mmap(ST0_55)       ; Clear all fields (OVx, C, TCx)
    or    #4140h,mmap(ST1_55)   ; Set CPL, FRCT, SXMD
    and   #07940h,mmap(ST1_55)  ; Clear BRAF, M40, SATD, C16, 54CM, ASM
    or    #0022h,mmap(ST2_55)   ; Set AR1 and AR5 in circular mode
    bclr  ARMS                  ; Disable ARMS bit in ST2
    bclr  SST                   ; Saturate-on-store is disabled
;
; Set up parameters and pointers for the LMS algorithm
;
    mov dbl(*AR0(#2)),XAR3      ; AR3 pointer to des[], large memory model
    mov dbl(*AR0(#4)),XAR2      ; AR2 pointer to out[], large memory model
    mov dbl(*AR0(#6)),XAR1      ; AR1 pointer to w[], large memory model
    mov dbl(*AR0(#8)),XAR4      ; AR4 pointer to x[], large memory model
    mov dbl(*AR0(#10)),XAR7     ; AR7 pointer to err[], large memory model
    mov *AR0(#12),T0            ; T0 = step
    mov *AR0(#13),T1            ; T1 = order
    mov *AR0(#14),AC0           ; AC0 = size of data block
    mov *AR0(#15),AR5           ; AR5 is index in data array
    mov dbl(*AR0),XAR0          ; AR0 point to in[], large memory model
    mov mmap(AR4),BSA45         ; BSA45 as start of circular data buffer
    mov mmap(AR1),BSA01         ; BSA01 as start of coefficients buffer
    mov #0,AR1                  ; AR0 to the 1st coefficient in buffer
    sub #1,AC0                  ; Set block repeat counter
    mov mmap(AC0L),BRC0
    mov mmap(T1),BK03           ; BK03 with order used with AR2
    aadd #1,T1
    mov mmap(T1),BK47           ; BK47 = number of data samples (order+1)
    asub #3,T1
    mov mmap(T1),BRC1           ; Inner loop to number of coefficients-2
;
; Process block data using adaptive filter
;
```

continues overleaf

Table 7.9 Implementation of the delayed LMS algorithm (*continued*)

```
    mov #0,AC1                  ; Clear AC1 for initial error term
||  rptblocal outer_loop-1
    mov hi(AC1),T3              ; Put error in T3
    mov *AR0+,*AR5+             ; Get input
    mpym *AR5+,T3,AC0           ; Put the 1st update term in AC0
||  mov #0,AC1                  ; Clearing FIR value
    lms *AR1,*AR5,AC0,AC1       ; AC0 has the update coefficient w[0]
                                ; AC1 is the 1st FIR output out[0]
||  rptblocal inner_loop-1
    mov hi(AC0),*AR1+           ; Save the updated coefficient
||  mpym *AR5+,T3,AC0           ; Computing the next update coefficient
    lms *AR1,*AR5,AC0,AC1       ; AC0 has the update coefficient w[i]
inner_loop:                     ; AC1 is update of FIR output out[i]
    mov hi(AC0),*AR1+           ; Save the updated coefficient
||  mov rnd(hi(AC1)),*AR2+      ; Save the FIR filter output
    sub AC1,*AR3+<<#16,AC2      ; AC2 is error amount
||  amar *AR5+                  ; Point to oldest data sample
    mpyr T0,AC2,AC1             ; Update mu_error term and place in AC1
||  mov hi(AC2),*AR7+           ; Save error term
outer_loop:
;
; Restore registers and DSP processor modes
;
    mov AR5,T0                  ; Return data x[] index of oldest data
    popboth XAR7                ; Restore AR7
    popboth XAR5                ; Restore AR5
    pop T3
||  bset ARMS                   ; Set ARMS bit for C-caller
    bclr FRCT                   ; Clear FRCT bit in ST1 return to C-caller
    and #0F800h,mmap(ST2_55)    ; Reset pointers in linear mode
    ret
```

This experiment is written in block-processing fashion. The nested repeat loops are placed in the instruction buffer using the `repeatlocal` instruction, which further improves the real-time performance of LMS algorithm. As introduced in Chapter 2, the C55x assembly programming supports different representations for hexadecimal constants. For example, the hex constants #0F800h in Table 7.9 is the same as C representation using #0xF8000. The files used for this experiment are listed in Table 7.10.

Table 7.10 File listing for `exp7.6.4_asm_DLMS`

Files	Description
`asm_dlmsTest.c`	C function for testing delayed LMS experiment
`asm_dlms.asm`	Assembly function for delayed LMS algorithm
`asm_dlms.h`	C header file
`asm_dlms.pjt`	DSP project file
`asm_dlms.cmd`	DSP linker command file
`input.pcm`	Input signal file
`desired.pcm`	Desired signal file

Procedures of the experiment are listed as follows:

1. Open the project file, `asm_dlms.pjt`, and rebuild the project.
2. Run the experiment using data files `input.pcm` and `desired.pcm`.
3. Compare the delayed LMS algorithm results with those obtained from the fixed-point C experiment.
4. Adjust block size `N`, filter length `L`, and step size `STEP` in the header file `asm_dlms.h` to evaluate convergence speed and steady-state MSE.
5. Profile the delayed LMS algorithm and compare the run-time efficiency with the fixed-point C implementation and intrinsics implementation in terms of number of cycles per data sample.

7.6.5 Adaptive System Identification

In this section, we will introduce the system identification experiment using the LMS algorithm. The block diagram of adaptive system identification is given in Figure 7.7. The adaptive system identification operations can be expressed as:

1. Place the current sample $x(n)$ generated by the signal generator into `x[0]` of the signal buffer.
2. Compute the adaptive FIR filter output

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l). \tag{7.66}$$

3. Calculate the error signal

$$e(n) = d(n) - y(n). \tag{7.67}$$

4. Update the filter coefficients

$$w_l(n+1) = w_l(n) + \mu e(n)x(n-l), \qquad l = 0,\ 1, \ldots, L-1. \tag{7.68}$$

5. Update the signal buffer

$$x(n-l-1) = x(n-l), \qquad l = L-2,\ L-1, \ldots, 1,\ 0. \tag{7.69}$$

The adaptive system identification shown in Figure 7.7 can be implemented in C as follows:

```
// Simulate an unknown system
    x1[0]=input;               // Get input signal x(n)
    d = 0.0;
    for (i=0; i<N1; i++)       // Compute d(n)
       d += (coef[i]*x1[i]);
    for (i=N1-1; i>0; i--)     // Update signal buffer
       x1[i] = x1[i-1];        // of unknown system
// Adaptive system identification operation
    x[0]=input;                // Get input signal x(n)
    y = 0.0;
    for (i=0; i<N0; i++)       // Compute output y(n)
       y += (w[i]*x[i]);
```

```
        e = d - y2;                  // Calculate error e(n)
        uen = twomu*e;               // uen = mu*e(n)
        for (i=0; i<N0; i++)         // Update coefficients
           w[i]+= (uen*x[i]);
        for (i=N0-1; i>0; i--)       // Update signal buffer
           x[i] = x[i-1];            //   of adaptive filter
```

The unknown system for this experiment is an FIR filter with the filter coefficients given in `plant[ ]`. The input $x(n)$ is a zero-mean white noise. The unknown system's output $d(n)$ is used as the desired signal for the adaptive filter, and the adaptive filter coefficients in `w[i]` will match closely to the unknown system response after the convergence of adaptive filter. The adaptive LMS algorithm used for system identification is listed in Table 7.11.

First, the signal and coefficient buffers are initialized to zero. The random signal used for the experiment is generated in `Ns` samples per block. The adaptive filter of the system identification program uses the unknown plant output $d(n)$ as the desired signal to calculate the error signal. The adaptive filter with N coefficients in `w[ ]` after convergence models the unknown system in the form of an FIR filter. The files used for this experiment are listed in Table 7.12. This experiment is implemented using block processing.

Procedures of the experiment are listed as follows:

1. Open the project file, `sysIdentify.pjt`, and rebuild the project.

2. Run the system identification experiment using the data file `x.pcm`. The experiment will write the result in the text file `result.txt` in the data directory.

3. Compare the system identification result in `result.txt` with the unknown plant given by `unknow_plant.dat`.

4. Use MATLAB to design a bandpass FIR filter and rerun the system identification experiment using this bandpass FIR filter as the unknown plant.

5. Increase the adaptive filter length L to $L = 2N_1$, where N_1 is the length of the unknown system. Build the project and run the program again. Check the experiment results. Will the adaptive model identify the unknown plant?

6. Reduce the adaptive filter length to $L = N_1/2$, where N_1 is the length of the unknown system. Build the project and run the program again. Check the experiment results. Will the adaptive model identify the unknown plant?

7. Use MATLAB to design a second-order bandpass IIR filter and rerun the system identification experiment using this bandpass IIR filter as an unknown plant. What is system identification result for the IIR unknown plant?

7.6.6 Adaptive Prediction and Noise Cancelation

As shown in Figure 7.9, the primary signal $x(n)$ consists of the broadband components $v(n)$ and the narrowband components $s(n)$. The output of adaptive filter is the narrowband signal $y(n) \approx s(n)$. For applications such as spread spectrum communications, the narrowband interference can be tracked and removed by the adaptive filter. The error signal $e(n) \approx v(n)$ contains the desired broadband signal. We

Table 7.11 List of C55x assembly code for adaptive system identification

```
_sysIdentification:
    pshm ST1_55                ; Save ST1, ST2, and ST3
    pshm ST2_55
    pshm ST3_55
    mov  dbl(*AR0(#2)),XAR1    ; AR1 is desired signal pointer
    mov  dbl(*AR0(#4)),XAR2    ; AR2 is signal buffer pointer
    mov  dbl(*AR0(#6)),XAR3    ; AR3 is coefficient buffer pointer
    mov  *AR0(#8),T0           ; T0 number of samples in input buffer
    mov  *AR0(#9),T1           ; T1 adaptive filter length
    mov  mmap(AR3),BSA45
    mov  mmap(T1),BK47
    mov  mmap(AR2),BSA23
    mov  mmap(T1),BK03
    mov  *AR0(#10),AR3         ; AR3 -> x[] as circular buffer
    mov  #0,AR4                ; AR4 -> w[] as circular buffer
    mov  dbl(*AR0),XAR0        ; AR0 is input pointer
    or   #0x340,mmap(ST1_55    ; Set FRCT,SXMD,SATD
    or   #0x18,mmap(ST2_55)    ; Enable circular addressing mode
    bset SATA                  ; Set SATA
    sub  #1,T0
    mov  mmap(T0),BRC0         ; Set sample block loop counter
    sub  #2,T1
    mov  mmap(T1),BRC1         ; Counter for LMS update loop
    mov  mmap(T1),CSR          ; Counter for FIR filter loop
    rptblocal loop-1           ; for (n=0; n<Ns; n++)
    mov  *AR0+,*AR3            ;   x[n]=in[n]
    mpym *AR3+,*AR4+,AC0       ;   temp = w[0]*d[0]
||  rpt  CSR                   ; for (i=0; i<N-1; i++)
    macm *AR3+,*AR4+,AC0       ;   y += w[i]*x[i]
    sub  *AR1+ <<#16,AC0       ; AC0=-e=y-d[n], AR1 points to d[n]
    mpyk #-TWOMU,AC0
    mov  rnd(hi(AC0)),mmap(T1); T1=mu*e[n]
    rptblocal lms_loop-1       ; for(j=0; i<N-2; i++)
    mpym *AR3+,T1,AC0          ;   AC0=2*mu*e*x[i]
    add  *AR4<<#16,AC0         ;   w[i]+=2*mu*e*x[i]
    mov  rnd(hi(AC0)),*AR4+
lms_loop
    mpym *AR3,T1,AC0           ; w[N-1]+=mu*e*x[N-1]
    add  *AR4<<#16,AC0
    mov  rnd(hi(AC0)),*AR4+    ; Store the last w[N-1]
loop
    popm ST3_55                ; Restore ST1, ST2, and ST2
    popm ST2_55
    popm ST1_55
    mov  AR3,T0                ; Return T0=index
    ret
```

use a fixed delay Δ as shown in Figure 7.9. The C55x assembly language implementaiton of the adaptive predictor is listed in Table 7.13.

In this experiment, we use the leaky LMS algorithm and white noise as the broadband signal. Since the white noise is uncorrelated, the delay $\Delta = 1$ is chosen. Table 7.14 lists the files used for this experiment. The experiment is written using block processing.

Table 7.12 File listing for `exp7.6.5_system_identificaiton`

Files	Description
`system_identificaitonTest.c`	C function for testing system identification experiment
`sysIdentification.asm`	Assembly function for LMS adaptive filter
`unknowFirFilter.asm`	Assembly function for an FIR unknown plant
`system_identify.h`	C header file
`unknow_plant.dat`	Include file for unknown FIR system coefficients
`sysIdentify.pjt`	DSP project file
`sysIdentify.cmd`	DSP linker command file
`x.pcm`	Input signal file

Table 7.13 C55x assembly implementation of adaptive linear predictor

```
_adaptivePredictor
    aadd #(ARGS-Size+1),SP           ; Adjust SP for local variables
    mov  dbl(*AR0(#2)),XAR1          ; AR1 pointer to y[]
    mov  dbl(*AR0(#4)),XAR2          ; AR2 pointer to e[]
    mov  dbl(*AR0(#6)),XAR3          ; AR3 pointer to x[]
    mov  dbl(*AR0(#8)),XAR4          ; AR4 pointer to w[]
    mov  *AR0(#10),T0                ; T0 = size of data block
    mov  *AR0(#11),T1                ; T1 = order
    mov  mmap(AR4),BSA45             ; Configure for circular buffers
    mov  mmap(T1),BK47
    mov  mmap(AR3),BSA23
    mov  mmap(T1),BK03
    mov  *AR0(#12),AR3               ; AR3 -> x[] as circular buffer
    mov  #0,AR4                      ; AR4 -> w[] as circular buffer
    mov  dbl(*AR0),XAR0              ; AR0 point to in[]
    mov  mmap(ST1_55),AC0            ; Save ST1, ST2, and ST3
    mov  AC0,ale.d_ST1
    mov  mmap(ST2_55),AC0
    mov  AC0,ale.d_ST2
    mov  mmap(ST3_55),AC0
    mov  AC0,ale.d_ST3
    or   #0x340,mmap(ST1_55)         ; Set FRCT,SXMD,SATD
    or   #0x18,mmap(ST2_55)          ; Enable circular addressing mode
    bset SATA                        ; Set SATA
    sub  #1,T0
    mov  mmap(T0),BRC0               ; Set sample block loop counter
    sub  #2,T1
    mov  mmap(T1),BRC1               ; Counter for LMS update loop
    mov  mmap(T1),CSR                ; Counter for FIR filter loop
    mov  #ALPHA,T0                   ; T0=leaky alpha
||  rptblocal loop-1                 ; for (n=0; n<Ns; n++)
    mpym *AR3+,*AR4+,AC0             ; temp = w[0]*x[0]
||  rpt  CSR                         ; for (i=1; i<N; i++)
    macm *AR3+,*AR4+,AC0             ;  temp += w[i]*x[i]
    mov  rnd(hi(AC0)),*AR1           ; y[n] = temp;
    sub  *AR0,*AR1+,AC0              ; e[n]=in[n]-y[n]
    mov  rnd(hi(AC0)),*AR2+          ; Save y[n]
    mpyk #TWOMU,AC0
    mov  rnd(hi(AC0)),mmap(T1)       ; T1=mu*e[n]
    mpym *AR4,T0,AC0
```

Table 7.13 (*continued*)

```
||   rptblocal lms_loop-1          ; for(j=0; i<N-2; i++)
     macm *AR3+,T1,AC0             ;   w[i]=alpha*w[i]+mu*e*x[i]
     mov  rnd(hi(AC0)),*AR4+
     mpym *AR4,T0,AC0
lms_loop
     macm *AR3,T1,AC0              ; w[N-1]=alpha*w[N-1]+mu*e[n]*x[N-1]
     mov  rnd(hi(AC0)),*AR4+       ; Store the last w[i]
     mov  *AR0+,*AR3               ; x[n]=in[n]
loop
     mov  ale.d_ST1,AR4            ; Restore ST1, ST2, and ST3
     mov  ar4,mmap(ST1_55)
     mov  ale.d_ST2,AR4
     mov  ar4,mmap(ST2_55)
     mov  ale.d_ST3,AR4
     mov  AR4,mmap(ST3_55)
     aadd #(Size-ARGS-1),SP        ; Reset SP
     mov  AR3,T0                   ; Return T0=index
     ret
```

Procedures of the experiment are listed as follows:

1. Open the project file, `adaptive_predictor.pjt`, and rebuild the project.
2. Run the adaptive predictor experiment using data files (`sine_1000hz_8khz.pcm` and `noise.pcm`) from data directory. The experiment will output the narrowband and broadband signals at files `output.pcm` and `error.pcm`.
3. Verify the adaptive predictor results using MATLAB by plotting the waveform, spectrum, and also by sound play back.
4. Change the length of the adaptive filter and observe the system performance.
5. Adjust the step size and observe the system performance.
6. Change the leaky factor value and observe the system performance.
7. Can we obtain a similar result without using the leaky LMS algorithm by setting the leaky factor to 0x7fff)?

Table 7.14 File listing for `exp7.6.6_adaptive_predictor`

Files	Description
`adaptive_predictorTest.c`	C function for testing system identification experiment
`adaptivePredictor.asm`	Assembly function for adaptive predictor
`adaptive_predictor.h`	C header file
`adaptive_predictor.pjt`	DSP project file
`adaptive_predictor.cmd`	DSP linker command file
`sine_1000hz_8khz.pcm`	1 kHz sine data file
`noise.pcm`	Noise data file

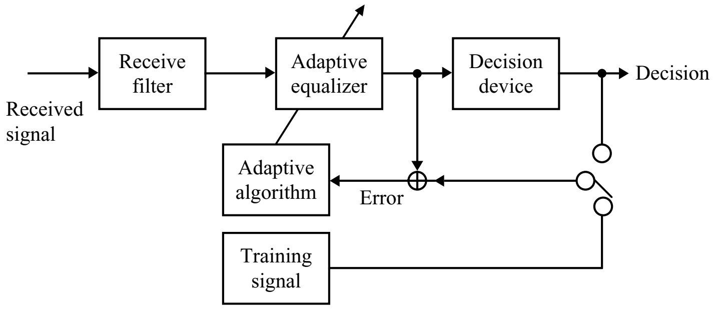


Figure 7.20 Simplified block diagram of ITU V.29 modem with adaptive channel equalizer

7.6.7 Adaptive Channel Equalizer

In this experiment, we implement a simplified complex adaptive equalizer for a simplified ITU V.29 modem. According to V.29 recommendation, the V.29 modem operates on the general switched telephone network lines. The speed is up to 9600 bits/s.

The equalizer for modems can be realized as an adaptive FIR filter. In the absence of noise and intersymbol interference, the modem receiving decision logic would be precisely matched to the transmitted symbols and the error signal will be zero. Figure 7.20 shows the block diagram of a simplified V.29 adaptive channel equalizer.

The decision-directed equalizer is effective only in tracking slow variation in channel response. For this reason, V.29 recommendation calls for force training using given sequences. The V.29 modem uses a complex equalizer for passband processing. The complex LMS algorithm defined in Equation (7.40) and (7.41) can be implemented as follows:

$$y_{\mathrm{r}}(n) = \sum_{l=0}^{L-1} \left[w_{\mathrm{r},l}(n) x_{\mathrm{r}}(n-l) - w_{\mathrm{i},l}(n) x_{\mathrm{i}}(n-l) \right] \tag{7.70}$$

$$y_{\mathrm{i}}(n) = \sum_{l=0}^{L-1} \left[w_{\mathrm{r},l}(n) x_{\mathrm{i}}(n-l) + w_{\mathrm{i},l}(n) x_{\mathrm{r}}(n-l) \right] \tag{7.71}$$

$$e_{\mathrm{r}}(n) = d_{\mathrm{r}}(n) - x_{\mathrm{r}}(n) \tag{7.72}$$

$$e_{\mathrm{i}}(n) = d_{\mathrm{i}}(n) - x_{\mathrm{i}}(n) \tag{7.73}$$

$$w_{\mathrm{r},l}(n+1) = w_{\mathrm{r},l}(n) - \mu_{\mathrm{r}}[e_{\mathrm{r}}(n) x_{\mathrm{r}}(n-l) - e_{\mathrm{i}}(n) x_{\mathrm{i}}(n-l)] \tag{7.74}$$

$$w_{\mathrm{i},l}(n+1) = w_{\mathrm{i},l}(n) - \mu_{\mathrm{i}}[e_{\mathrm{r}}(n) x_{\mathrm{i}}(n-l) + e_{\mathrm{i}}(n) x_{\mathrm{r}}(n-l)]. \tag{7.75}$$

Force training sequence defined by V.29 recommendation includes two symbols. These symbols are ordered according to the following random number generator:

$$1 \oplus x^{-6} \oplus x^{-7}, \tag{7.76}$$

where $\oplus$ represents the exclusive-OR operation. When the random number generated is 0, the point (3, 0) will be transmitted. When the random number is 1, the point (−3, 3) will be transmitted. In the receiver, a local generator will recreate the identical sequence and use it as the desired signal $d(n)$ for computing the error signal. The V.29 force training sequence consists of a total of 384 symbols. The fixed-point C implementation of complex channel equalizer is listed in Table 7.15. The files used for this experiment are listed in Table 7.16.

Table 7.15 Fixed-point implementation of complex channel equalizer for V.29 modem

```
void equalizer(COMPLEX *rx, COMPLEX *out, COMPLEX *error)
{
   COMPLEX y,err;
   long temp32,urer,urei,uier,uiei;
   short j;
   // Update data delay line
   for(j=EQ_ORDER-1; j>0; j--)
   {
       x[j] = x[j-1];
   }
   x[0] = *rx;
   // Compute normalized mu from I-symbol
   temp32 = (((long)x[0].re * x[0].re)+0x4000)>>15;
   temp32 = ((temp32 * ONE_MINUS_BETA)+0x4000)>>15;
   power.re = (short)(((power.re * (long)BETA)+0x4000)>>15);
   temp32 += (power.re+C);
   temp32 >>= 5;
   mu.re = ALPHA / (short)temp32;
   // Compute normalized mu from Q-symbol
   temp32 = (((long)x[0].im * x[0].im)+0x4000)>>15;
   temp32 = ((temp32 * ONE_MINUS_BETA)+0x4000)>>15;
   power.im = (short)(((power.im * (long)BETA)+0x4000)>>15);
   temp32 += (power.im+C);
   temp32 >>= 5;
   mu.im = ALPHA / (short)temp32;
   // Get the real adaptive filter output from complex symbols
   temp32  = (long)w[0].re * x[0].re;
   temp32 -= (long)w[0].im * x[0].im;
   for(j=1; j<EQ_ORDER; j++)
   {
       temp32 += (long)w[j].re * x[j].re;
       temp32 -= (long)w[j].im * x[j].im;
   }
   y.re = (short)((temp32+ROUND)>>15);
   // Get the image adaptive filter output from complex symbols
   temp32  = (long)w[0].im * x[0].re;
   temp32 += (long)w[0].re * x[0].im;
   for(j=1; j<EQ_ORDER; j++)
   {
       temp32 += (long)w[j].im * x[j].re;
       temp32 += (long)w[j].re * x[j].im;
   }
   y.im = (short)((temp32+ROUND)>>15);
   // Compute error term from complex data
   err.re = rxDesire[txCnt].re - y.re;
   err.im = rxDesire[txCnt++].im - y.im;
   // Coefficients update - using complex error and data
   urer = (long)(((err.re * (long)mu.re)+ROUND)>>15);
   urei = (long)(((err.im * (long)mu.re)+ROUND)>>15);
   uier = (long)(((err.re * (long)mu.im)+ROUND)>>15);
   uiei = (long)(((err.im * (long)mu.im)+ROUND)>>15);
   for(j=0; j<EQ_ORDER ; j++)
```

continues overleaf

Table 7.15 Fixed-point implementation of complex channel equalizer for V.29 modem (*continued*)

```
    {
        temp32   = (long)urer * x[j].re;
        temp32  -= (long)urei * x[j].im;
        temp32   = (short)((temp32+ROUND)>>15);
        w[j].re -= (short)temp32;
    }
    for(j=0; j<EQ_ORDER ; j++)
    {
        temp32   = (long)uiei * x[j].re;
        temp32  += (long)uier * x[j].im;
        temp32   = (short)((temp32+ROUND)>>15);
        w[j].im -= (short)temp32;
    }
    // Return the output and error
    *error = err;
    *out = y;
}
```

Procedures of the experiment are listed as follows:

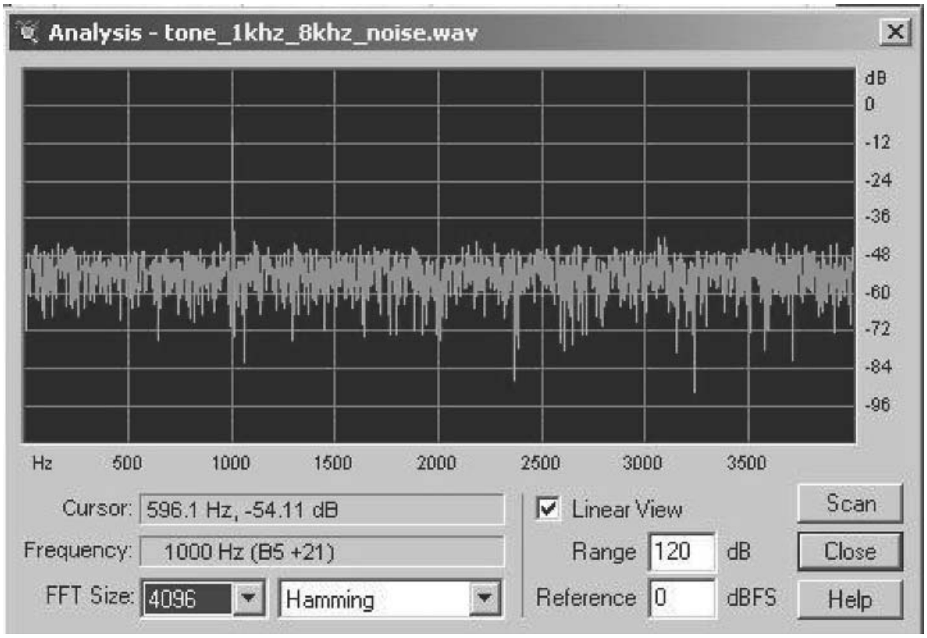
1. Open the project file, `channel_equalizer.pjt`, and rebuild the project.
2. Run the adaptive channel equalizer experiment. This experiment will output the error signal to file `error.bin` in the data directory.
3. Plot the error signal to verify the convergence of the equalizer.
4. Change the filter length and observe the behavior of adaptive equalizer.

7.6.8 Real-Time Adaptive Line Enhancer Using DSK

In this experiment, we will port the adaptive predictor experiment in Section 7.6.6 to the C5510 DSK to examine the real-time behavior. There are two signal files: one is a single tone corrupted by white noise, and the other consists of repeated telephone digits corrupted by white noise. The input data files can be played back via an audio player that supports WAV file format. The DSK takes the input, processes it, and sends the output to a headphone or loudspeaker for play back. Figure 7.21 shows the spectrum of the input signal, and Figure 7.22 is the output captured in real time by an audio sound card. It can be seen

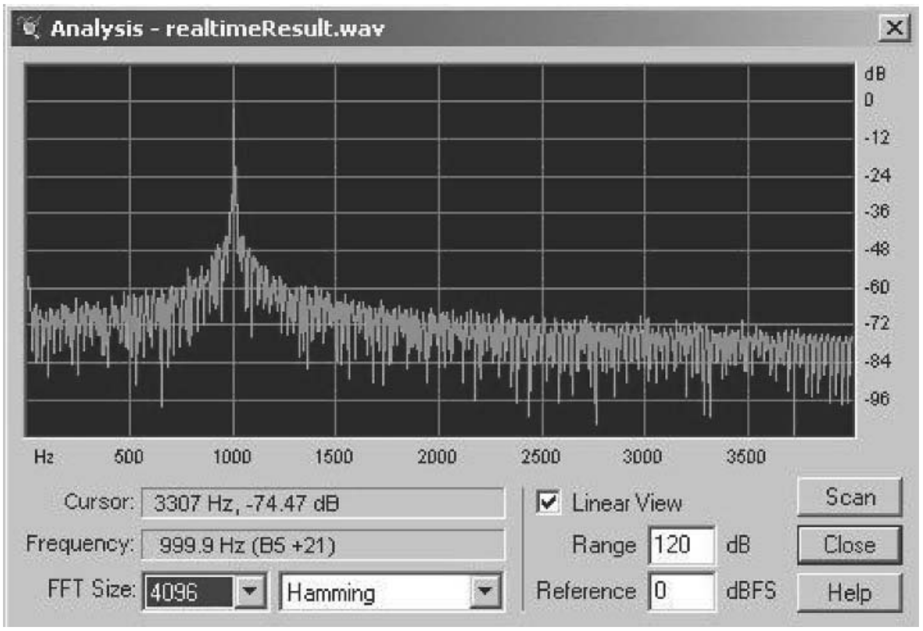
Table 7.16 File listing for `exp7.6.7_channel_equalizer`

Files	Description
`eqTest.c`	C function for testing adaptive equalizer experiment
`adaptiveEQ.c`	C function for implementing adaptive equalizer
`channel.c`	C function simulates communication channel
`signalGen.c`	C function generates training sequence
`complexEQ.h`	C header file
`channel_equalizer.pjt`	DSP project file
`channel_equalizer.cmd`	DSP linker command file



Cursor: 596.1 Hz, -54.11 dB
Frequency: 1000 Hz (B5 +21)
FFT Size: 4096 Hamming
✓ Linear View
Range 120 dB
Reference 0 dBFS

Figure 7.21 Spectrum of the signal corrupted by broadband noise



Cursor: 3307 Hz, -74.47 dB
Frequency: 999.9 Hz (B5 +21)
FFT Size: 4096 Hamming
✓ Linear View
Range 120 dB
Reference 0 dBFS

Figure 7.22 Spectrum of the adaptive line predictor output. The broadband noise has been reduced

Table 7.17 File listing for `exp7.6.8_realtime_predictor`

Files	Description
`rt_realtime_predictor.c`	C function for testing line enhancer experiment
`adaptivePredictor.asm`	Assembly function for adaptive line enhancer
`plio.c`	C function interfaces PIP with low-level I/O functions
`adaptive_predictor.h`	C header file for experiment
`plio.h`	C header file for PIP driver
`lio.h`	C header file for interfacing PIP with low-level drivers
`rt_adaptivePredictor.pjt`	DSP project file
`rt_adaptivePredictorcfg.cmd`	DSP linker command file
`rt_adaptivePredictor.cdb`	DSP/BIOS configuration file
`tone_1khz_8khz_noise.wav`	Data file – tone with noise
`multitone_noise_8khz.wav`	Data file – multitone with noise

from Figure 7.22 that the wideband noise has been greatly reduced by the 128-tap adaptive line enhancer. The files used for this experiment are listed in Table 7.17.

Procedures of the experiment are listed as follows:

1. Open the project file, `rt_adaptivePredictor.pjt`, and rebuild the project.
2. Run the adaptive line enhancer using the C5510 DSK. Connect the audio player output to the DSK line-in. Use a headphone to listen to the result from the DSK headphone output.
3. Change the adaptive filter length, step size, and evaluate the behavior changes of the adaptive line enhancer.
4. Capture the input and output signals using a digital scope and evaluate the adaptive filter performance by examining the time-domain waveform and frequency-domain noise level before and after applying the adaptive line enhancer.

References

[1] S. T. Alexander, *Adaptive Signal Processing,* New York: Springer-Verlag, 1986.
[2] M. Bellanger, *Adaptive Digital Filters and Signal Analysis,* New York: Marcel Dekker, 1987.
[3] P. M. Clarkson, *Optimal and Adaptive Signal Processing,* Boca Raton, FL: CRC Press, 1993.
[4] C. F. N. Cowan and P. M. Grant, *Adaptive Filters,* Englewood Cliffs, NJ: Prentice Hall, 1985.
[5] J. R. Glover, Jr., 'Adaptive noise canceling applied to sinusoidal interferences,' *IEEE Trans. Acoust.*, ASSP-25, pp. 484–491, Dec. 1977.
[6] S. Haykin, *Adaptive Filter Theory,* 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1991.
[7] S. M. Kuo and C. Chen, 'Implementation of adaptive filters with the TMS320C25 or the TMS320C30,' in *Digital Signal Processing Applications with the TMS320 Family*, vol. 3, P. Papamichalis, Ed., Englewood Cliffs, NJ: Prentice Hall, 1990, pp. 191–271, Chap. 7.
[8] S. M. Kuo and D. R. Morgan, *Active Noise Control Systems – Algorithms and DSP Implementations*, New York: John Wiley & Sons, Inc., 1996.
[9] L. Ljung, *System Identification: Theory for the User*, Englewood Cliffs, NJ: Prentice Hall, 1987.
[10] J. Makhoul, 'Linear prediction: A tutorial review,' *Proc. IEEE*, vol. 63, pp. 561–580, Apr. 1975.

[11] J. R. Treichler, C. R. Johnson, Jr., and M. G. Larimore, *Theory and Design of Adaptive Filters*, New York: John Wiley & Sons, Inc., 1987.
[12] B. Widrow, J. R. Glover, J. M. McCool, J. Kaunitz, C. S. Williams, R. H. Hern, J. R. Zeidler, E. Dong, and R. C. Goodlin, 'Adaptive noise canceling: Principles and applications,' *Proc. IEEE*, vol. 63, pp. 1692–1716, Dec. 1975.
[13] B. Widrow and S. D. Stearns, *Adaptive Signal Processing,* Englewood Cliffs, NJ: Prentice-Hall, 1985.
[14] M. L. Honig and D. G. Messerschmitt, *Adaptive Filters: Structures, Algorithms, and Applications*, Boston, MA: Kluwer Academic Publishers, 1986.
[15] MathWorks, Inc., *Using MATLAB*, Version 6, 2000.
[16] MathWorks, Inc., *Signal Processing Toolbox User's Guide*, Version 6, 2004.
[17] MathWorks, Inc., *Filter Design Toolbox User's Guide*, Version 3, 2004.
[18] MathWorks, Inc., *Fixed-Point Toolbox User's Guide*, Version 1, 2004.
[19] MathWorks, Inc., *Communications Toolbox User's Guide*, Version 3, 2005.
[20] ITU Recommendation V.29, *9600 Bits Per Second Modem Standardized for Use on Point-to-Point 4-Wire Leased Telephone-Type Circuits*, Nov. 1988.

Exercises

1. Determine the autocorrelation function of the following signals:

   (a) $x(n) = A\sin(2\pi n/N)$,

   (b) $y(n) = A\cos(2\pi n/N)$.

2. Find the crosscorrelation functions $r_{xy}(k)$ and $r_{yx}(k)$, where $x(n)$ and $y(n)$ are defined in the Problem 1.

3. Let $x(n)$ and $y(n)$ be two independent zero-mean WSS random signals. The random signal $w(n)$ is obtained by using

$$w(n) = ax(n) + by(n),$$

   where a and b are constants. Express $r_{ww}(k)$, $r_{wx}(k)$, and $r_{wy}(k)$ in terms of $r_{xx}(k)$ and $r_{yy}(k)$.

4. Similar to Example 7.7, the desired signal $d(n)$ is the output of the FIR filter with coefficients 0.2, 0.5, and 0.3 when the input $x(n)$ is zero-mean, unit-variance white noise. This white noise is also used as the input signal for the adaptive FIR filter with $L = 3$ using the LMS algorithm. Compute $\mathbf{R}$, $\mathbf{p}$, $\mathbf{w}^{\mathrm{o}}$, and minimum MSE.

5. Consider a second-order autoregressive (AR) process defined by

$$d(n) = v(n) - a_1 d(n-1) - a_2 d(n-2),$$

   where $v(n)$ is a white noise of zero mean and variance σ_v^2. This AR process is generated by filtering $v(n)$ using the second-order IIR filter $H(z)$.

   (a) Derive the IIR filter transfer function $H(z)$.

   (b) Consider a second-order optimum FIR filter shown in Figure 7.3. If the desired signal is $d(n)$, the primary input $x(n) = d(n-1)$. Find the optimum weight vector $\mathbf{w}^{\circ}$ and the minimum MSE $\xi_{\min}$.

6. Given the two finite-length sequences:
   $x(n) = \{1 \quad 3 \quad -2 \quad 1 \quad 2 \quad -1 \quad 4 \quad 4 \quad 2\}$,
   $y(n) = \{2 \quad -1 \quad 4 \quad 1 \quad -2 \quad 3\}$.

Using MATLAB function `xcorr`, compute and plot the crosscorrelation function $r_{xy}(k)$ and the autocorrelation function $r_{xx}(k)$.

7. Write a MATLAB script to generate the length 1024 signal defined as

$$x(n) = 0.8\sin(\omega_0 n) + v(n),$$

where $\omega_0 = 0.1\pi$, $v(n)$ is a zero-mean random noise with variance $\sigma_v^2 = 1$ (see Section 3.3 for details). Compute and plot $r_{xx}(k)$, where $k = 0, 1, \ldots, 127$, using MATLAB. Explain this simulation result using theoretical derivations given in Examples 7.1 and 7.3.

8. Redo Example 7.7 by using $x(n)$ as input to the adaptive FIR filter ($L = 2$) with the LMS algorithm. Implement this adaptive filter using MATLAB or C. Plot the error signal $e(n)$, and show the adaptive weights converged to the derived optimum values.

9. Implement the adaptive system identification technique illustrated in Figure 7.7 using MATLAB or C program. The input signal is a zero-mean, unit-variance white noise. The unknown system is an IIR filter defined in Problem 5. Evaluate different filter lengths L and step size μ, and plot $e(n)$ for these parameters. Find the optimum values that result in fast convergence and low excess MSE.

10. Implement the adaptive line enhancer illustrated in Figure 7.9 using MATLAB or C program. The desired signal is given by

$$x(n) = \sqrt{2}\sin(\omega n) + v(n),$$

where frequency $\omega = 0.2\pi$ and v(n) is the zero-mean white noise with unit variance. The decorrelation delay $\Delta = 1$. Plot both $e(n)$ and $y(n)$. Evaluate the convergence speed and steady-state MSE for different parameters L and μ.

11. Implement the adaptive noise cancelation illustrated in Figure 7.11 using MATLAB or C program. The primary signal is given by

$$d(n) = \sin(\omega n) + 0.8v(n) + 1.2v(n-1) + 0.25v(n-2)$$

where $v(n)$ is defined by Problem 5. The reference signal is $v(n)$. Plot $e(n)$ for different values of L and μ.

12. Implement the single-frequency adaptive notch filter illustrated in Figure 7.14 using MATLAB or C program. The desired signal $d(n)$ is given in Problem 11, and $x(n)$ is given by

$$x(n) = \sqrt{2}\sin(\omega n).$$

Plot $e(n)$ and the magnitude response of second-order FIR filter after convergence.

13. Use MATLAB to generate primary input signal $x(n) = 0.25\cos(2\pi n f_1/f_s) + 0.25\sin(2\pi n f_2/f_s)$ and the reference signal $d(n) = 0.125\cos(2\pi n f_2/f_s)$, where f_s is sampling frequency, f_1 and f_2 are the frequencies of the desired signal and interference, respectively. Implement the adaptive noise canceler that removed the interference signal.

14. Port the functions developed in Problem 13 to DSK. Create a real-time experiment by connecting the primary input and reference input signals to the DSK stereo-line input. Left channel is the primary input with interference and the right channel contains only the interference signal. Test the adaptive noise canceler in real time with DSK.

15. Create a real-time adaptive notch filter experiment using DSK.

16. The system identification experiment is implemented for large memory model. Modify the program given by Table 7.11 such that this assembly program can be used by both large memory model and small memory model.

8 Digital Signal Generators

Signal generations are useful for algorithm design, analysis, and real-world DSP applications. In this chapter, we will introduce different methods for the generation of digital signals and their applications.

8.1 Sinewave Generators

There are several characteristics that should be considered when designing algorithms for generating sinewaves. These issues include total harmonic distortion, frequency and phase control, memory usage, computational cost, and accuracy.

Some trigonometric functions can be approximated by polynomials, for example, the cosine and sine approximation given by Equations (3.90a) and (3.90b). Because polynomial approximations are realized with multiplications and additions, they can be efficiently implemented on DSP processors. Sinewave generation using polynomial approximation is presented in Section 3.6.5, and using resonator is introduced in Chapter 5. Therefore, this section discusses only the lookup-table method for sinewave generation.

8.1.1 Lookup-Table Method

The lookup-table (or table-lookup) method is probably the most flexible technique for generating periodic waveforms. This technique involves reading a series of stored data values that represent the waveform. These values can be obtained either by sampling analog signals or by computing the mathematical algorithms. Usually only one period of the waveform is stored in the table.

A sinewave table containing one period of waveform can be obtained by computing the following function:

$$x(n) = \sin\left(\frac{2\pi n}{N}\right), \qquad n = 0, 1, \ldots N-1. \tag{8.1}$$

These samples are represented in binary form; thus, the accuracy is determined by the wordlength. The desired sinewave can be generated by reading these stored values from the table at a constant step Δ. The data pointer wraps around at the end of the table. The frequency of the generated sinewave depends on the sampling period T, table length N, and the table address increment Δ as

$$f = \frac{\Delta}{NT}\ \text{Hz}. \tag{8.2}$$

For a given sinewave table of length N, a sinewave with frequency f and sampling rate f_s can be generated using the pointer address increment

$$\Delta = \frac{Nf}{f_s} \tag{8.3}$$

with the following constraint to avoid aliasing:

$$\Delta \leq \frac{N}{2}. \tag{8.4}$$

To generate an L-sample sinewave $x(l)$, where $l = 0, 1, \ldots, L-1$, we use a circular pointer k such that

$$k = (m + l\Delta)_{\mathrm{mod}\,N}, \tag{8.5}$$

where m determines the initial phase of sinewave. It is important to note that the step Δ given in Equation (8.3) may not be an integer; thus, $(m + l\Delta)$ in Equation (8.5) makes k a real number. The values between neighboring entries can be estimated using the existing table values. An easy solution is to round the noninteger index k to the nearest integer. A better but more complex method is to interpolate the value based on the adjacent samples.

The following two errors will cause harmonic distortion:

1. Amplitude quantization errors due to the use of finite wordlength to represent values in the table.

2. Time-quantization errors from synthesizing data values between table entries.

Increasing table size can reduce the time-quantization errors. To reduce the memory requirement, we can take the advantage of symmetry property since the absolute values of a sinewave repeat four times in each period. Thus, only one-fourth of the period is required. However, a more complex algorithm will be needed to track which quadrant of the waveform is generated.

To decrease the harmonic distortion for a given table size, an interpolation technique can be used to compute the values between table entries. The simple linear interpolation that assumes a value between two consecutive table entries lies on a straight line between these two values. Suppose the integer part of the pointer k is i $(0 \leq i < N)$ and the fractional part is $f(0 < f < 1)$, the interpolated value will be computed as

$$x(n) = s(i) + f\,[s(i+1) - s(i)], \tag{8.6}$$

where $[s(i+1) - s(i)]$ is the slope of the line between successive table entries $s(i)$ and $s(i+1)$.

Example 8.1: We use the MATLAB program `example8_1.m` for generating one period of 200 Hz sinewave with sampling rate 4000 Hz as shown in Figure 8.1. These 20 samples are stored in a table for generating sinewave with $f_s = 4$ kHz. From Equation (8.3), $\Delta = 1$ will be used for generating 200 Hz sinewave and $\Delta = 2$ for 400 Hz. But, $\Delta = 1.5$ should be needed for generating 300 Hz.

From Figure 8.1, when we access the lookup table with $\Delta = 1.5$, we get the first value which is the first entry in the table as shown by arrow. However, the second value is not available in the table since it is in between the second and third entries. Therefore, the linear interpolation results in the

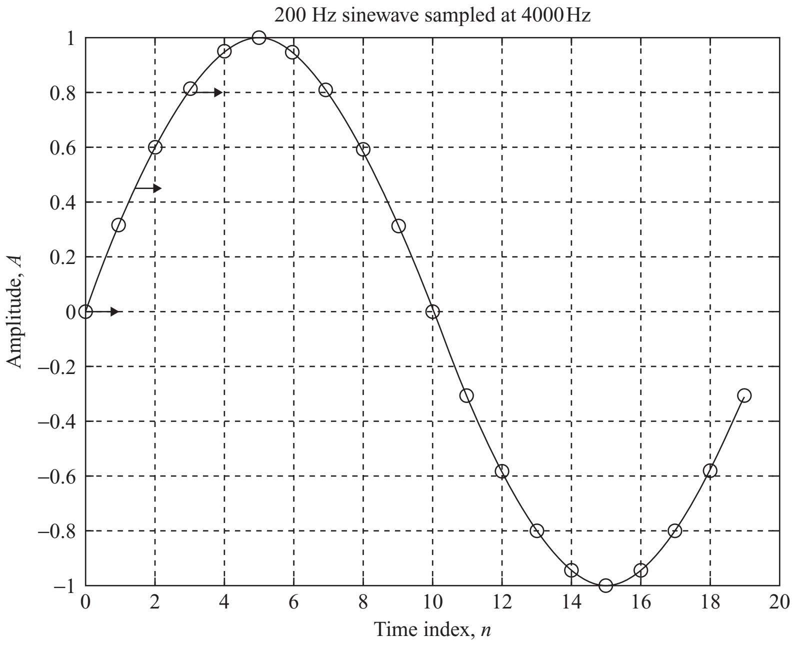


Figure 8.1 One period of sinewave, where sinewave samples are marked by 'o'

average of these two entries. To generate 250 Hz sinewave, $\Delta = 1.25$, and we can use Equation (8.6) for computing sample values with noninteger index.

Example 8.2: A cosine/sine function generator using table-lookup method with 1024-point cosine table can be implemented using the following TMS320C55x assembly code (`cos_sin.asm`):

```
;  cos_sin.asm - Table lookup sinewave generator with
;                   1024-point cosine table range (0, π)
;

;  Prototype: void cos_sin(short, short *, short *)
;  Entry:     arg0: T0 - alpha
;             arg1: AR0 - pointer to cosine
;             arg2: AR1 - pointer to sine

    .def  _cos_sin
    .ref  tab_0_PI
    .sect "cos_sin"

_cos_sin
    mov T0,AC0          ; T0=a
    sfts AC0,#11        ; Size of lookup table
    mov #tab_0_PI, T0   ; Table based address
||  mov hi(AC0),AR2
    mov AR2,AR3
```

```
        abs AR2              ; cos(-a) = cos(a)
        add #0x200,AR3       ; 90 degree offset for sine
        and #0x7ff,AR3       ; Modulo 0x800 for 11-bit
        sub #0x400,AR3       ; Offset 180 degree for sine
        abs AR3              ; sin(-a) = sin(a)
   ||   mov *AR2(T0),*AR0    ; *AR0=cos(a)
        mov *AR3(T0),*AR1    ; *AR1=sin(a)
        ret
        .end
```

In this example, we use a one-half period table $(0 - \pi)$ to reduce memory usage. Obviously, a sine function generator using a full table $(0 - 2\pi)$ can be easily implemented with only a few lines of assembly code, while a function generator using a one-fourth table $(0 - \pi/2)$ will be more complicated. The implementation of sinewave generator for the C5510 DSK using the table-lookup technique will be presented in Section 8.4.

8.1.2 Linear Chirp Signal

A linear chirp signal is a waveform whose instantaneous frequency changes linearly with time between two specified frequencies. It is a waveform with the lowest possible peak to root-mean-square amplitude ratio in the desired frequency band. The digital chirp waveform is expressed as

$$c(n) = A\sin[\phi(n)], \tag{8.7}$$

where A is a constant amplitude and $\phi(n)$ is a quadratic phase in the form of

$$\phi(n) = 2\pi\left[f_{\mathrm{L}}n + \left(\frac{f_{\mathrm{U}} - f_{\mathrm{L}}}{2(N-1)}\right)n^2\right] + \alpha, \qquad 0 \le n \le N-1, \tag{8.8}$$

where N is the total number of points in a single chirp. In Equation (8.8), α is an arbitrary constant phase factor, and f_{L} and f_{U} are the normalized lower and upper frequency limits, respectively. The waveform periodically repeats with

$$\phi(n + kN) = \phi(n), \qquad k = 1, 2, \ldots. \tag{8.9}$$

The instantaneous normalized frequency is defined as

$$f(n) = f_{\mathrm{L}} + \left(\frac{f_{\mathrm{U}} - f_{\mathrm{L}}}{N-1}\right)n, \qquad 0 \le n \le N-1. \tag{8.10}$$

This expression shows that the instantaneous frequency goes from $f(0) = f_{\mathrm{L}}$ at time $n = 0$ to $f(N-1) = f_{\mathrm{U}}$ at time $n = N-1$.

Because of the complexity of the linear chirp signal generator, it is more convenient to generate a chirp sequence by computer and store it in a lookup table for real-time applications. An alternative solution is to generate the table during DSP system initialization process. The lookup-table method introduced in Section 8.1.1 can be used to generate the desired signal using the stored table.

MATLAB *Signal Processing Toolbox* provides the function `y = chirp(t, f0, t1, f1)` for generating linear chirp signal at the time instances defined in array `t`, where `f0` is the frequency at time 0 and `f1` is the frequency at time `t1`. Variables `f0` and `f1` are in Hz.

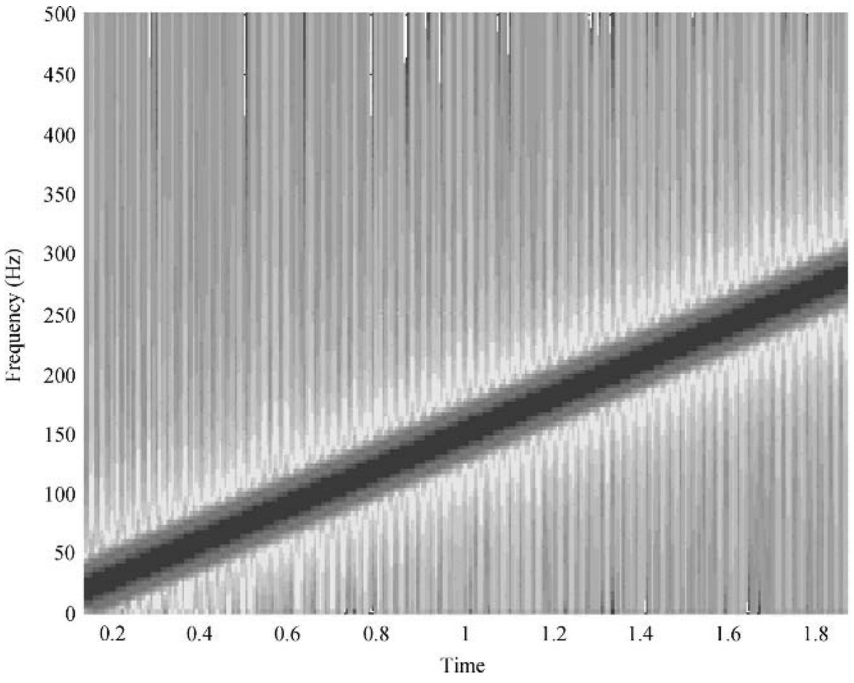


Figure 8.2 Spectrogram of chirp signal from 0 to 300 Hz

Example 8.3: Compute the spectrogram of a chirp signal with the sampling rate 1000 Hz. The signal sweeps from 0 to 150 Hz in 1 s. The MATLAB code is listed as follows (`example8_3.m`, adapted from MATLAB **Help** menu):

```
Fs = 1000;             % Define variables
T = 1/Fs;
t = 0:T:2;             % 2 seconds at 1 kHz sample rate
y = chirp(t,0,1,150);  % Start at DC, cross 150 Hz at t=1 second
spectrogram(y,256,250,256,1E3,'yaxis')
```

The spectrogram of generated chirp signal is illustrated in Figure 8.2.

8.2 Noise Generators

Random numbers are used in many practical applications for simulating noises. Although we cannot produce perfect random numbers by using digital hardware, it is possible to generate a sequence of numbers that are unrelated to each other. Such numbers are called pseudo-random numbers. In this section, we will introduce random number generation algorithms.

8.2.1 Linear Congruential Sequence Generator

The linear congruential method is widely used by random number generators, and can be expressed as

$$x(n) = [ax(n-1) + b]_{\mathrm{mod}\,M}, \qquad (8.11)$$

Table 8.1 C program for generating linear congruential sequence

```
/*
* URAN - Generation of floating-point pseudo-random numbers
*/
static long n=(long)12357; // Seed x(0) = 12357
float uran()
{
  float ran;                // Random noise r(n)
  n=(long)2045*n+1L;        // x(n)=2045*x(n-1)+1
  n-=(n/1048576L)*1048576L;//x(n)=x(n)-INT[x(n)/1048576]*1048576
  ran=(float)(n+1L)/(float)1048577; //r(n)=FLOAT[x(n)+1]/1048577
  return(ran);              // Return r(n) to the main function
}
```

where the modulo operation (mod) returns the remainder after division by M. The constants a, b, and M can be chosen as

$$a = 4K + 1, \tag{8.12}$$

where K is an odd number such that a is less than M, and

$$M = 2^L \tag{8.13}$$

is a power of 2, and b can be any odd number. Equations (8.12) and (8.13) guarantee that the period of the sequence given by Equation (8.11) has full-length M.

A good choice of these parameters are $M = 2^{20} = 1\,048\,576$, $a = 4(511) + 1 = 2045$, and $x(0) = 12\,357$. Since a random number generator usually produces samples between 0 and 1, we can normalize the nth random sample as

$$r(n) = \frac{x(n) + 1}{M + 1} \tag{8.14}$$

so that the random samples are greater than 0 and less than 1. A floating-point C function (uran.c) that implements the random number generator defined by Equations (8.11) and (8.14) is listed in Table 8.1. A fixed-point C function (rand.c) that is more efficient for a fixed-point DSP processor was provided in Section 3.6.6.

Example 8.4: The following TMS320C55x assembly code (rand_gen.asm) implements an $M = 2^{16}$ (65 536) random number generator:

```
;  rand16_gen.asm - 16-bit zero-mean random number generator
;
;  Prototype: int rand16_gen(int *)
;
;  Entry:  arg0 - AR0 pointer to seed value
;  Return: T0 - Random number
```

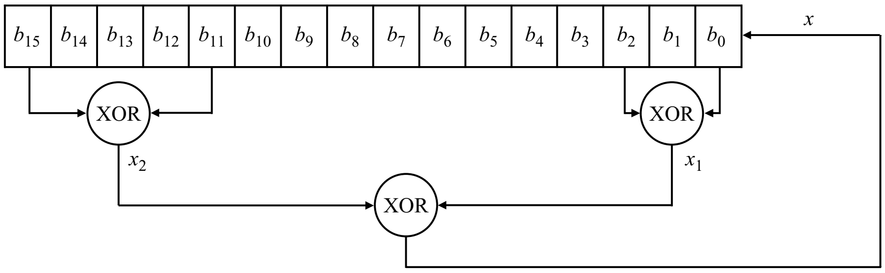

b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
x
XOR
XOR
x_2
x_1
XOR

Figure 8.3 16-bit pseudo-random number generator

```
C1 .equ   0x6255
C2 .equ   0x3619
   .def   _rand16_gen
   .sect  "rand_gen"
_rand16_gen
    mov   #C1,T0
    mpym  *AR0,T0,AC0      ; Seed=(C1*seed+C2)
    add   #C2,AC0
    and   #0xffff,AC0      ; Seed%=0x10000
    mov   AC0,*AR0
    sub   #0x4000,AC0      ; Zero-mean random number
    mov   AC0,T0
    ret
    .end
```

8.2.2 Pseudo-Random Binary Sequence Generator

A shift register with feedback from specific bit locations can also generate a repetitive pseudo-random sequence. The schematic of a 16-bit generator is shown in Figure 8.3, where the functional operator labeled 'XOR' performs the exclusive-OR operation of its two binary inputs. The sequence itself is determined by the position of the feedback bits on the shift register. In Figure 8.3, x_1 is the output of b_0 XOR with b_2, x_2 is the output of b_{11} XOR with b_{15}, and x is the output of x_1 XOR with x_2.

Each output from the sequence generator is the entire 16-bit of the register. After the random number is generated, every bit in the register is shifted left by 1 bit (b_{15} is lost), and then x is shifted into b_0 position. A shift register of length 16 bits can readily be accommodated by a single word on 16-bit DSP processors. It is important to recognize, however, that sequential words formed by this process will be correlated. The maximum sequence length before repetition is

$$L = 2^M - 1, \tag{8.15}$$

where M is the number of bits of the shift register.

Example 8.5: The pseudo-random number generator given in Table 8.2 (`pn_sequence.c`) requires at least 11 C statements to complete the computation. The following TMS320C55x assembly

Table 8.2 C program for generating pseudo-random sequence

```
//
// Pseudo-random sequence generator
//
static short shift_reg;
short pn_sequence(short *sreg)
{
    short b2,b11,b15;
    short x1,x2;            /* x2 also used for x        */

    b15 = *sreg >>15;
    b11 = *sreg >>11;
    x2 = b15^b11;           /* First b15 XOR b11 */
    b2 = *sreg >>2;
    x1 = *sreg ^b2;         /* Second b2 XOR b0 */
    x2 = x1^x2;             /* Final x1 XOR x2 */
    x2 &= 1;
    *sreg = *sreg <<1;
    *sreg = *sreg | x2;     /* Update the shift register */
    x2 = *sreg-0x4000;      /* Zero-mean random number */
    return x2;
}
```

program (`pn_gen.asm`) computes the same sequence in 11 cycles:

```
; pn_gen.asm - 16-bit pseudo-random sequence generator
;
; Prototype: int pn_gen(int *)
;
; Entry: arg0 - AR0 pointer to the shift register
; Return: T0 - Random number

BIT15 .equ 0x8000             ; b15
BIT11 .equ 0x0800             ; b11
BIT2  .equ 0x0004             ; b2
BIT0  .equ 0x0001             ; b0
   .def _pn_gen
   .sect "rand_gen"
_pn_gen
   mov   *AR0,AC0             ; Get register value
   bfxtr #(BIT15| BIT2),AC0,T0 ; Get b15 and b2
   bfxtr #(BIT11| BIT0),AC0,T1 ; Get b11 and b0
   sfts  AC0,#1
|| xor   T0,T1                 ; XOR all 4 bits
   mov   T1,T0
   sfts  T1,#-1
   xor   T0,T1                 ; Final XOR
   and   #1,T1
   or    T1,AC0
   mov   AC0,*AR0              ; Update register
   sub   #0x4000,AC0,T0        ; Zero-mean random number
|| ret
   .end
```

8.3 Practical Applications

In this section, we will introduce some real-world applications that are related to the sinewave and random number generators.

8.3.1 Siren Generators

An interesting application of chirp signal generator is to generate sirens. The electronic sirens are often produced by a generator inside the vehicle compartment. This generator drives either a 60- or 100-W loudspeaker in a light bar mounted on the vehicle roof. The actual siren characteristics (bandwidth and duration) vary slightly from manufacturers. The wail type of siren sweeps between 800 and 1700 Hz with a sweep period of approximately 4.92 s. The yelp siren has similar characteristics to the wail but with a period of 0.32 s.

Example 8.6: We modify the chirp signal generator given in Example 8.3 for generating sirens. The MATLAB code `example8_6.m` generates wail type of siren and plays it using `soundsc` function.

8.3.2 White Gaussian Noise

The MATLAB *Communication Toolbox* provides `wgn` function for generating white Gaussian noise (WGN) that is widely used for modeling communication channels. We can specify the power of the noise in dBW (decibels relative to 1-watt), dBm, or linear units. We can generate either real or complex noise. For example, the command below generates a vector of length 50 containing real-valued WGN whose power is 2 dBW:

```
y1 = wgn(50,1,2);
```

The function assumes that the load impedance is 1 Ω.

Example 8.7: A WGN channel adds white Gaussian noise to the signal that passes through it. To model a WGN channel, use the `awgn` function as follows:

```
y = awgn(x,snr)
```

This command adds white Gaussian noise to the vector signal `x`. The scalar `snr` specifies the signal-to-noise ratio in dB. If `x` is complex, then `awgn` adds complex noise. This syntax assumes that the power of x is 0 dBW.

The following MATLAB script (`example8_7.m`, adapted from MATAB **Help** menu) adds white Gaussian noise to a square wave signal. It then plots the original and noisy signals in Figure 8.4:

```
t = 0:.1:20;
x = square(t);              % Create square signal
y = awgn(x,10,'measured'); % Add white Gaussian noise
plot(t,x,t,y)               % Plot both signals
legend('Original signal','Signal with AWGN');
```

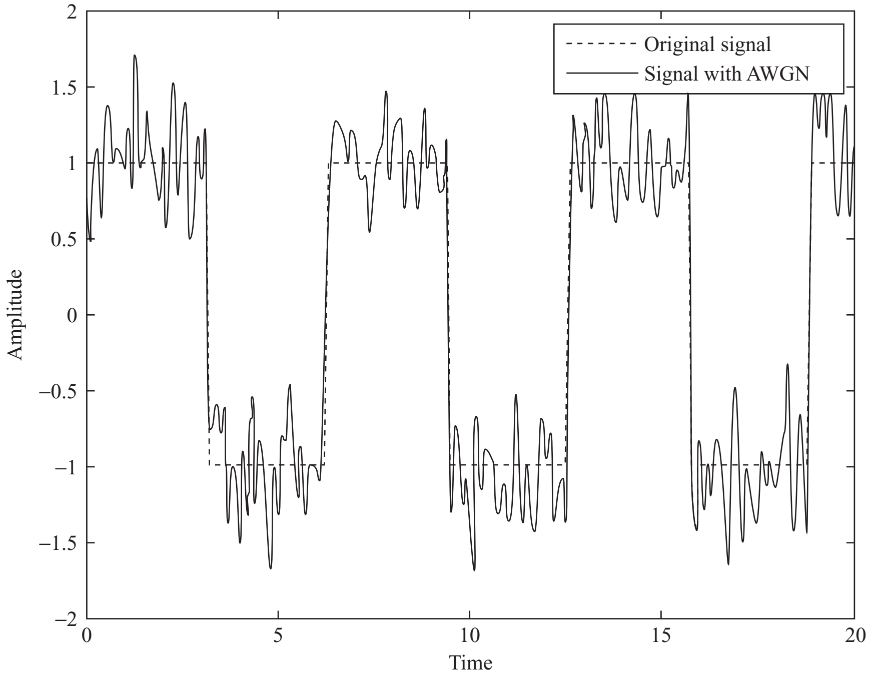


Figure 8.4 A square wave corrupted by white Gaussian noise

Note that in the code, `square(t)` generates a square wave with period 2π for the elements of time vector `t` with peaks of $+1$ to -1 instead of a sinewave.

8.3.3 Dual-Tone Multifrequency Tone Generator

A common application of sinewave generator is the touch-tone telephones and cellular phones that use the dual-tone multifrequency (DTMF) transmitter and receiver. DTMF also finds widespread uses in electronic mail systems and automated telephone servicing systems in which the user can select options from a menu by sending DTMF signals from a telephone.

Each key-press on the telephone keypad generates the sum of two tones expressed as

$$x(n) = \cos(2\pi f_L nT) + \cos(2\pi f_H nT), \tag{8.16}$$

where T is the sampling period, and the two frequencies f_L and f_H uniquely define the key that was pressed. Figure 8.5 shows the matrix of the frequencies used to encode the 16 DTMF symbols defined by ITU Recommendation Q.23. The values of these eight frequencies have been chosen carefully so that they do not interfere with speech.

The low-frequency group (697, 770, 852, and 941 Hz) selects the row frequencies of the 4×4 keypad, and the high-frequency group (1209, 1336, 1477, and 1633 Hz) selects the column frequencies. A pair of sinusoidal signals with f_L from the low-frequency group and f_H from the high-frequency group will

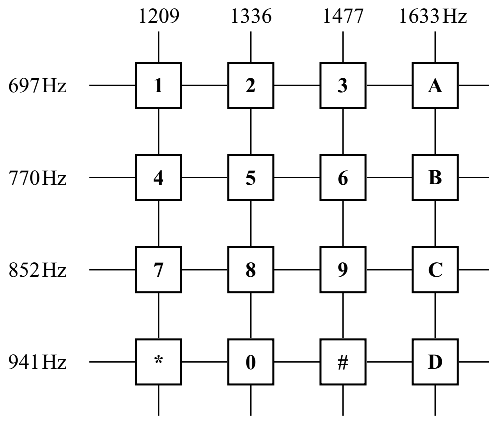


Figure 8.5 Telephone keypad matrix

represent a particular key. For example, the digit '3' is represented by two sinewaves at frequencies 697 and 1477 Hz.

The generation of dual tones can be implemented using two sinewave generators connected in parallel. The DTMF signal must meet timing requirements for duration and spacing of digit tones. Digits are required to be transmitted at a rate of less than 10 per second. A minimum spacing of 50 ms between tones is required, and the tones must be presented for a minimum of 40 ms. A tone-detection scheme used as a DTMF receiver must have sufficient time resolution to verify correct digit timing. The issues of tone detection will be discussed later in Chapter 9.

8.3.4 Comfort Noise in Voice Communication Systems

In voice communication systems, the complete suppression of a signal using residual echo suppressor (will be discussed later in Section 10.5) has an adverse subjective effect. This problem can be solved by adding a low-level comfort noise. As illustrated in Figure 8.6, the output of residual echo suppressor is expressed as

$$y(n) = \begin{cases} \alpha v(n), & |x(n)| \le \beta \\ x(n), & |x(n)| > \beta \end{cases}, \tag{8.17}$$

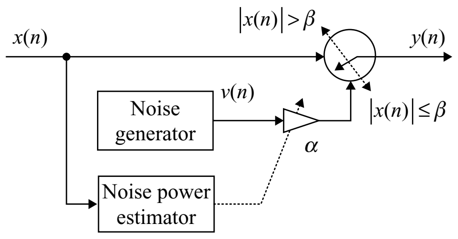


Figure 8.6 Injection of comfort noise with active center clipper

Table 8.3 File listing for experiment `exp8.4.1_signalGenerator`

Files	Description
`tone.c`	C function for testing experiment
`tone.cdb`	CCS configuration file for experiment
`tonecfg.cmd`	DSP linker command file
`signalGenerator.pjt`	DSP project file
`55xdspx.lib`	Large memory mode DSK library
`dsk5510bslx.lib`	Large memory mode DSK board support library

where $v(n)$ is an internally generated zero-mean pseudo-random noise, $x(n)$ is the input applied to the center clipper, and β is the clipping threshold.

In echo cancelation applications, the characteristics of the comfort noise should match the background noise when neither talker is active. In speech coding applications, the characteristics of the comfort noise should match the background noise during the silence. In both cases, the algorithm shown in Figure 8.6 is a process of estimating the power of the background noise in $x(n)$ and generating the comfort noise with same power to replace signals suppressed by the center clipper. Detailed information on residual echo suppressor and comfort noise generation will be presented in Chapter 10.

8.4 Experiments and Program Examples

This section presents several hands-on experiments including real-time signal generation using the C5510 DSK and DTMF generation using MATLAB.

8.4.1 Sinewave Generator Using C5510 DSK

The objective of this experiment is to use the C5510 DSK with its associated CCS, BSL (board support library), and AIC23 for generating sinusoidal signals. We will develop our programs based on the tone example project that is available in the C5510 DSK folder `..\examples\dsk5510\bsl\tone`. In this experiment, we will modify the C program and build the project using CCS for real-time execution on the C5510 DSK.

Table 8.3 lists the files used for this experiment. Procedures of the experiment are listed as follows:

1. Create a working folder and copy the following files from the DSK folder `..\examples\dsk5510\bsl\tone` into the new folder. In addition, also copy the DSPLIB `55xdspx.lib` from the DSK folder `..\c5500\dsplib` and `dsk5510bslx.lib` from the DSK folder `..\c5500\dsk5510\lib` into the new folder.

2. Start CCS and create a new project in the new folder. Add `tone.c, tone.cdb` and `tonecfg.cmd` into the project. In addition, also add the `55xdspx.lib` and `dsk5510bslx.lib` into the project. We will need DSPLIB functions to generate sine and random signals. Choose the large memory model and build the project.

3. Connect a headphone (or a loudspeaker) to the headphone output of the C5510 DSK and run the program.

Table 8.4 Code section to generate random signal

```
#define SINE_TABLE_SIZE    8         // No. of samples
short sinetable[SINE_TABLE_SIZE];    // Vector for random samples
...
for (msec = 0; msec < 5000; msec++)
{
  rand16(sinetable, SINE_TABLE_SIZE);

  for (sample = 0; sample < SINE_TABLE_SIZE; sample++)
  {
    /* Send a sample to the left channel */
    while (!DSK5510_AIC23_write16(hCodec, sinetable[sample]));
    /* Send a sample to the right channel */
    while (!DSK5510_AIC23_write16(hCodec, sinetable[sample]));
  }
}
```

In the C source code `tone.c`, the array `sinetable` contains 48 samples (which cover exactly one period) of a precalculated sinewave in Q15 data format shown below:

```
Int16 sinetable[SINE_TABLE_SIZE] = {
    0x0000, 0x10b4, 0x2120, 0x30fb, 0x3fff, 0x4dea, 0x5a81, 0x658b,
    0x6ed8, 0x763f, 0x7ba1, 0x7ee5, 0x7ffd, 0x7ee5, 0x7ba1, 0x76ef,
    0x6ed8, 0x658b, 0x5a81, 0x4dea, 0x3fff, 0x30fb, 0x2120, 0x10b4,
    0x0000, 0xef4c, 0xdee0, 0xcf06, 0xc002, 0xb216, 0xa57f, 0x9a75,
    0x9128, 0x89c1, 0x845f, 0x811b, 0x8002, 0x811b, 0x845f, 0x89c1,
    0x9128, 0x9a76, 0xa57f, 0xb216, 0xc002, 0xcf06, 0xdee0, 0xef4c
};
```

The sampling rate of CODEC is default at 48 kHz, thus the CODEC outputs 48 000 samples per second. Since the time interval between two consecutive samples is $T = 1/48\,000$ s, each period of sinewave contains 48 samples, and the period of sinewave is $48/48\,000 = 1/1000$ s $= 1$ ms. Therefore, the frequency of the generated sinewave is 1000 Hz. Since each period of sinewave is 1/1000 s, the program generates 5000 periods, and it lasts for 5 s.

8.4.2 White Noise Generator Using C5510 DSK

In this experiment, we use the C55x DSPLIB function `rand16` to generate eight samples of random signal for 8 kHz sampling rate (or 48 samples if the sampling rate is 48 kHz). Instead of writing a new program, we will modify `tone.c` from the previous experiment and rename it as `noise.c`. Partial of the modified C code that uses the array `sinetable[ ]` for storing random numbers is listed in Table 8.4. The files used for this experiment are listed in Table 8.5.

Procedures of the experiment are listed as follows:

1. Create a DSP project for the noise experiment.
2. Run the experiment and listen to the noise generated by the C5510 DSK.

Table 8.5 File listing for experiment `exp8.4.2_noiseGenerator`

Files	Description
`noise.c`	C function for testing experiment
`tone.cdb`	CCS configuration file for experiment
`tonecfg.cmd`	DSP linker command file
`noiseGeneration.pjt`	DSP project file
`55xdspx.lib`	Large memory mode DSK library
`dsk5510bslx.lib`	Large memory mode DSK board support library

3. Modify the experiment such that the noise generated will be sampled at 8 kHz.

4. Modify the experiment to generate 2 s of random noise at 8 kHz sampling rate.

8.4.3 Wail Siren Generator Using C5510 DSK

In this experiment, we will use the table-lookup method to implement a wail siren using the C5510 DSK. The modified C code using the array `sirentable[ ]` for storing siren data values is listed in Table 8.6.

There is a limitation for this experiment running on the C5510 DSK. In Example 8.6, the sweeping of data numbers from 800 to 1700 Hz at 8 kHz sampling rate requires a table of 39 360 entries. We will not be able to access the complete table because the addressing range of 16-bit C55x is limited to 32 767. To

Table 8.6 Code section for siren generator

```
#define SIREN_TABLE_SIZE      19680 /* Length of siren table */

Int16 sirentable[SIREN_TABLE_SIZE]={
    #include "wailSiren.h"
};
/* Generate 10-sweep of siren wave */
for (i=0; i<10; i++)
{
  for (sample = 0; sample < SIREN_TABLE_SIZE; sample++)
  {
    data = sirentable[sample]; // Get two samples each time
    /* Send first sample to the left channel */
    while (!DSK5510_AIC23_write16(hCodec, (data&0xff)<<8));

    /* Send first sample to the right channel */
    while (!DSK5510_AIC23_write16(hCodec, (data&0xff)<<8));

    /* Send second sample to the left channel */
    while (!DSK5510_AIC23_write16(hCodec, data&0xff00));

    /* Send second sample to the right channel */
    while (!DSK5510_AIC23_write16(hCodec, data&0xff00));
  }
}
```

Table 8.7 File listing for experiment `exp8.4.3_sirenGenerator`

Files	Description
`siren.c`	C function for testing experiment
`tone.cdb`	CCS configuration file for experiment
`tonecfg.cmd`	DSP linker command file
`sirenGenerator.pjt`	DSP project file
`55xdspx.lib`	Large memory mode DSK library
`dsk5510bslx.lib`	Large memory mode DSK board support library

resolve this problem, we generate 8-bit siren data and pack two 8-bit data into one 16-bit word. In this way, we can use a table of 19 680 entries for the 4.92 s of wail siren. The demo program is modified, so each data read will take two 8-bit data and they are unpacked and played by the DSK. Table 8.7 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Create a DSP project for the siren experiment.
2. Write a C or MATLAB program to generate siren lookup table in 8-bit data and two 8-bit data packed formats.
3. Set the AIC23 sampling rate to 8 kHz.
4. Run the experiment and listen to the siren signal generated.

8.4.4 DTMF Generator Using C5510 DSK

In this experiment, we will implement DTMF signal generation using the C5510 DSK. We modify the previous table-lookup experiment to create a DTMF generator with 8-kHz sampling frequency. The ITU Q23 recommendation defines the DTMF signaling with eight frequencies, four lower frequencies for the rows and four high frequencies for the columns as shown in Figure 8.5. The ITU Q.24 recommendation specifies the duration of the DTMF signal and silence interval between DTMF signals.

We generate eight tables for eight DTMF frequencies. Each table has 800 entries for 100-ms duration. The following C code can be used to generate the sinewave tables:

```
w = 2.0*PI*f/Fs;
for(n=0; n<800; n++)
{
  cosine[n] = (short)(cos(w*n)*16383); // Q14 format
}
```

In the code, f is the DTMF frequency and Fs is the sampling frequency. Table 8.8 lists the partial code for DTMF tone generation.

This experiment can generate a series of DTMF signals from a given digit string. The DTMF tones are separated by 60 ms of silence. The files used for this experiment are listed in Table 8.9.

Table 8.8 Code section of DTMF signal generation

```
for (sample = 0; sample <DTMF_TABLE_SIZE; sample++)
{
    data = ptrFh[sample] + ptrFl[sample];
    /* Send data to the left channel */
    while (!DSK5510_AIC23_write16(hCodec, data));
    /* Send data to the right channel */
    while (!DSK5510_AIC23_write16(hCodec, data));
}
for (sample = 0; sample <600; sample++)
{
    /* Send data to the left channel */
    while (!DSK5510_AIC23_write16(hCodec, 0));
    /* Send data to the right channel */
    while (!DSK5510_AIC23_write16(hCodec, 0));
}
```

Procedures of the experiment are listed as follows:

1. Create a DSP project for the DTMF experiment. Configure the DSK to 8 kHz sampling rate.
2. Write a C or MATLAB program to generate eight sinewave lookup tables according to the frequencies given in Figure 8.5.
3. Build and run the experiment. Listen to the generated DTMF tones. Change the DTMF string in different order or combination, rerun the experiment, and evaluate the DTMF generator.

8.4.5 DTMF Generator Using MATLAB Graphical User Interface

In this experiment, we will use MATLAB graphical user interface (GUI) and its callback functions to develop a DTMF generator. The files used for this experiment are listed in Table 8.10. Use the following

Table 8.9 File listing for experiment `exp8.4.4_DTMFGenerator`

Files	Description
`dtmfGenerator.c`	C function for testing experiment
`dtmfGenerator.cdb`	CCS configuration file for experiment
`dtmfGeneratorcfg.cmd`	DSP linker command file
`dtmfGenerator.pjt`	DSP project file
`tone697.h`	DTMF tone lookup table
`tone770.h`	DTMF tone lookup table
`tone852.h`	DTMF tone lookup table
`tone941.h`	DTMF tone lookup table
`tone1209.h`	DTMF tone lookup table
`tone1336.h`	DTMF tone lookup table
`tone1477.h`	DTMF tone lookup table
`tone1633.h`	DTMF tone lookup table
`55xdspx.lib`	Large memory mode DSK library
`dsk5510bslx.lib`	Large memory mode DSK board support library

Table 8.10 File listing for experiment `exp8.4.5_MatlabDTMFGen`

Files	Description
`DTMFGenerator.m`	MATLAB GUI controls experiment
`DTMFGenerator.fig`	MATLAB GUI graphic file for experiment

procedures to create the MATLAB DTMF generator:

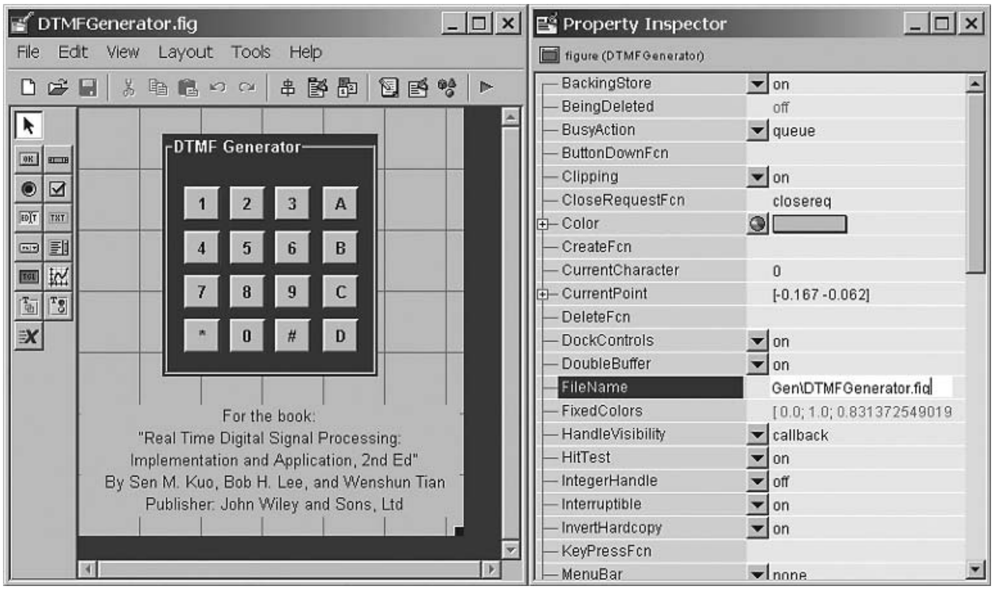
1. To start MATLAB GUI design, enter the command `guide` from MATLAB command window and choose **Blank GUI** from the MATLAB **Create New GUI** menu. The GUI design tool will be shown as in Figure 8.7.

2. Select the **Push** button from the tool menu and enter it to create 16 buttons as shown. Rename the push buttons to 1, 2, . . . C, D as shown in Figure 8.7 from **Property Inspector**. We can also change the letter font and color.

3. When the design of GUI looks satisfied, save it as `DTMFGenerator.fig`. A MATLAB file `DTM-FGenerator.m` will be saved automatically. The file `DTMFGenerator.m` consists of 16 callback functions that represent to the 16 buttons of the DTMF generator.

4. Edit the `DTMFGenerator.m` to add code for each button. For example, the following MATLAB code is added to the callback function of button '5'.

```
disp('5 is pushed.')
fl=770;fh=1336;     % "5"
key=dtmfGen(fl,fh); % Call DTMF generator
```



For the book:
"Real Time Digital Signal Processing:
Implementation and Application, 2nd Ed"
By Sen M. Kuo, Bob H. Lee, and Wenshun Tian
Publisher: John Wiley and Sons, Ltd

figure (DTMFGenerator)

BackingStore	on
BeingDeleted	off
BusyAction	queue
ButtonDownFcn	
Clipping	on
CloseRequestFcn	closereq
Color	
CreateFcn	
CurrentCharacter	0
CurrentPoint	[-0.167 -0.062]
DeleteFcn	
DockControls	on
DoubleBuffer	on
FileName	Gen\DTMFGenerator.fig
FixedColors	[0.0; 1.0; 0.831372549019
HandleVisibility	callback
HitTest	on
IntegerHandle	off
Interruptible	on
InvertHardcopy	on
KeyPressFcn	
MenuBar	none

Figure 8.7 MATLAB GUI design

5. Add the following function to `DTMFGenerator.m`. This function takes low and high frequencies and generates a DTMF tone of duration 100 ms. The `sound` function is used to play back the DTMF tone:

```
function x=dtmfGen(fl, fh)
fs = 8000;
N = [0:1/fs:0.1];
x = 0.5*(cos(2*pi*fl*N)+cos(2*pi*fh*N));
sound(x,fs)
```

6. Enter `DTMFGenerator` from the MATLAB command window to start DTMF generator. Push the buttons and listen to the generated audio signal.

References

[1] S. M. Kuo and D. R. Morgan, *Active Noise Control Systems – Algorithms and DSP Implementations*, New York: John Wiley & Sons, Inc., 1996.

[2] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd Ed., Reading, MA: Addison-Wesley, 1981.

[3] N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Englewood Cliffs, NJ: Prentice Hall, 1983.

[4] Math Works, Inc., *Using MATLAB*, Version 6, 2000.

[5] Math Works, Inc., *Signal Processing Toolbox User's Guide*, Version 6, 2004.

[6] S. J. Orfanidis, *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1996.

[7] A Bateman and W. Yates, *Digital Signal Processing Design*, New York: Computer Science Press, 1989.

[8] J. Hartung, S. L. Gay, and G. L. Smith, *Dual-tone Multifrequency Receiver Using the WE DSP16 Digital Signal Processor*, Application Note, AT&T, 1988.

[9] Analog Devices, *Digital Signal Processing Applications Using the ADSP-2100 Family*, Englewood Cliffs, NJ: Prentice Hall, 1990.

[10] P. Mock, 'Add DTMF generation and decoding to DSP-uP designs,' *Digital Signal Processing Applications with the TMS320 Family*, Texas Instruments, 1986, Chap. 19.

[11] Texas Instruments, Inc., *DTMF Tone Generation and Detection on the TMS320C54x*, Literature no. SPRA096A, 1999.

[12] ITU-T Recommendation Q.23, *Technical Features of Push-Button Telephone Sets*, 1993.

[13] ITU-T Recommendation Q.24, *Multifrequency Push-Button Signal Reception*, 1993.

Exercises

1. For the tone generation experiment presented in Section 8.4.1, the default sampling rate for CODEC is 48 kHz. We can set different sampling frequencies using the function `DSK5510_AIC23_setFreq( )` available in the BSL `dsk5510bslx.lib`. For example, we can use the following command to set sampling rate to 8 kHz:

```
DSK5510_AIC23_setFreq(hCodec, DSK5510_AIC23_FREQ_8KHZ);
```

Now, modify the C program tone.c (used for experiment in Section 8.4.1) by inserting this line of code to set the sampling rate to 8 kHz. Build a new project and perform a real-time testing. After the program is run, compare the sound effects for the sampling rates at 8 and 48 kHz. Where in the file `tone.c` should you add the code to change AIC23 CODEC frequency? What is the frequency of sinewave that was generated with 8 kHz sampling rate? Why? Also, how many seconds the tone last? Why?

2. Modify the `tone.c` to generate 1 kHz tone with 8 kHz sampling rate. *Hints*: There are many ways and we briefly introduce the following two methods:

   (a) We can recalculate one period of sinewave with eight samples (using MATLAB or hand calculation) to replace the original 48 samples in `sinetable`. In this case, be sure to change `SINE_TABLE_SIZE` from 48 entries to eight elements.

   (b) We can use the same `sinetable` with 48 samples, but step through the table every six samples by modifying the outer loop as follows:

   ```
   for (sample = 0; sample < SINE_TABLE_SIZE; sample=sample+6)
   ```

   Try both methods and perform real-time testing. Make sure that we generate 1 kHz tone with 8 kHz sampling rate.

3. We have already learned how to generate a single tone using the pregenerated table:

   (a) Try to generate multiple sinewaves at different frequencies using the same table such as the `sinetable` in `tone.c` by stepping through the same table using different steps.

   (b) Try to use the DSPLIB function `sine(x, r, Nx)` to generate an array of sinewave.

4. We can combine both the sinewave and noise generators to generate a sinewave that is embedded in white noise for future experiments. Pay special attention to overflow problem when we add two Q.15 numbers. How can we prevent overflow? Build a new project and perform a real-time testing. Try different signal-to-noise ratio and compare the differences.

5. This is a challenging and practical problem: How to generate a tone (or multiple tones) at any frequency with any predetermined sampling rate using a table-lookup technique? We may find out that we have to step through the table with a noninteger step; thus, we have to interpolate a value between two consecutive samples in the table. We will also find that it is easier and better to design a new `sinetable` that have more samples (>48) to cover one period of sinewave.

6. The yelp siren has similar characteristics as the wail siren but its period is 0.32 s. Use the wail siren experiment as reference to create a yelp siren generator using the table-lookup method.

7. ITU Q.24 recommendation specifies that the DTMF frequency offsets for North America must be no more than 1.5 %. Develop a method to examine all 16 waveform tables used for DTMF generation given in Section 8.4.4. Are these DTMF tones all within the specified tolerance? If not, how to correct the problem?

8. The DTMF signal generation uses eight tables of 800 entries each. By packing two 8-bit bytes in one 16-bit word can save half of the data memory used for tables. Compress these eight tables into byte format and rerun the DTMF experiments.

9. The ITU Q24 allows the high-frequency component of the DTMF tone level to be higher than the low-frequency component. Redesign the experiment given in Section 8.4.4 such that the level of the high-frequency component of the DTMF tone generated is 3 dB higher than the low frequency.

10. Add two graph windows to the DTMF GUI in Section 8.4.5. One of these windows is used to display the time-domain DTMF signal waveform, and the other is used to plot the spectrum of the generated DTMF signal.

9 Dual-Tone Multifrequency Detection

Dual-tone multifrequency (DTMF) generation and detection are widely used in telephone signaling and interactive control applications through telephone and cellular networks. In this chapter, we will focus on the DTMF detection and applications.

9.1 Introduction

DTMF signaling was developed initially for telephony signaling such as dialing and automatic redial. Modems use DTMF for dialing stored numbers to connect with network service providers. DTMF has also been used in interactive remote access control with computerized automatic response systems such as airline's information systems, remote voice mailboxes, electronic banking systems, as well as many semiautomatic services via telephone networks. DTMF signaling scheme, reception, testing, and implementation requirements are defined in ITU Recommendations Q.23 and Q.24.

DTMF generation is based on a 4×4 grid matrix shown in Figure 8.5. This matrix represents 16 DTMF signals including numbers 0–9, special keys * and #, and four letters A–D. The letters A–D are assigned to unique functions for special communication systems such as the military telephony systems. As discussed in Chapter 8, the DTMF signals are based on eight specific frequencies defined by two mutually exclusive groups. Each DTMF signal consists of two tones that must be generated simultaneously. One is chosen from the low-frequency group to represent the row index, and the other is chosen from the high-frequency group for the column index.

A DTMF decoder must able to accurately detect the presence of these tones specified by ITU Q.23. The decoder must detect the DTMF signals under various conditions such as frequency offsets, power level variations, DTMF reception timing inconsistencies, etc. DTMF decoder implementation requirements are detailed in ITU-T Q.24 recommendation.

An application of using DTMF signaling for remote access control between individual users and bank automated electronic database is illustrated in Figure 9.1. In this example, user follows the prerecorded voice commands to key-in the corresponding information, such as the account number and user authentication, using a touch-tone telephone keypad. User's inputs are converted to a series of DTMF signals. The reception end processes these DTMF tones to reconstruct the digits for the remote access control. The banking system sends the queries, responses, and confirmation messages via voice channel to the user during the remote access process.

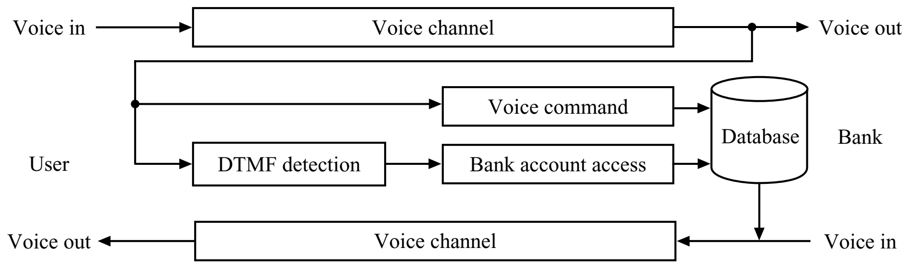


Figure 9.1 A simplified DTMF application used for remote access control

For voice over IP (VoIP) applications, a challenge for DTMF signaling is to pass through the VoIP networks via speech coders and decoders. When DTMF signaling is used with VoIP networks, the DTMF signaling events can be sent in data packet types. The procedure of how to carry the DTMF signaling and other telephony events in real-time transport protocol (RTP) packet is defined by Internet engineering task force RFC2833 specification.

Besides DTMF tones, there are many other multifrequency tones used in communications. For example, the call progress tones include dial tone, busy tone, ringing-back tone, and modem and fax tones. The basic tone detection algorithm and implementation techniques are similar. In this chapter, we will concentrate on the DTMF detection.

9.2 DTMF Tone Detection

This section introduces methods for detecting DTMF tones used in communication networks. The correct detection of a DTMF digit requires both a valid tone pair and the correct timing intervals. Since the DTMF signaling may be used to set up a call and to control functions such as call forwarding, it is necessary to detect DTMF signaling in the presence of speech.

9.2.1 DTMF Decode Specifications

The implementation of DTMF decoder involves the detection of the DTMF tones, and determination of the correct silence between the tones. In addition, it is necessary to perform additional assessments to ensure that the decoder can accurately distinguish DTMF signals in the presence of speech.

For North America, DTMF decoders are required to detect frequencies with a tolerance of $\pm 1.5\,\%$. The frequencies that are offset by $\pm 3.5\,\%$ or greater must not be recognized as DTMF signals. For Japan, the detection of frequencies has a tolerance of $\pm 1.8\,\%$, and the tone offset is limited to $\pm 3.0\,\%$. This requirement prevents the detector from falsely detecting speech and other signals as valid DTMF signals. The receiver must work under the worst-case signal-to-noise ratio of 15 dB with a dynamic range of 25 dB for North America (or 24 dB for Japan). The ITU-T requirements for North America are listed in the Table 9.1.

Another requirement is the ability to detect DTMF signals when two tones are received at different levels. This level difference is called twist. The high-frequency tone may be received at a lower level than the low-frequency tone due to the magnitude response of the communication channel, and this situation is described as a forward (or standard) twist. Reverse twist occurs when the received low-frequency tone has lower level than the high-frequency tone. The receiver must operate with a maximum 8 dB of forward twist and 4 dB of reverse twist. The final requirement is that the receiver must avoid incorrectly identifying the speech signal as valid DTMF tones. This is referred as talk-off performance.

Table 9.1 Requirements of DTMF specified in ITU-T Q.24

Signal frequencies	Low group	697, 770, 852, 941 Hz
	High group	1209, 1336, 1477, 1633 Hz
Frequency tolerance	Operation	$\leq 1.5\,\%$
	Nonoperation	$\geq 3.5\,\%$
Signal duration	Operation	40 ms min
	Nonoperation	23 ms max
Twist	Forward	8 dB max
	Reverse	4 dB max
Signal power	Operation	0 to −25 dBm
	Nonoperation	−55 dBm max
Interference by echoes	Echoes	Should tolerate echoes delayed up to 20 ms and at least 10 dB down

9.2.2 Goertzel Algorithm

The basic principle of DTMF detection is to examine the energy of the received signal and determine whether a valid DTMF tone pair has been received. The detection algorithm can be implemented using a DFT or a filterbank. For example, an FFT can calculate the energies of N evenly spaced frequencies. To achieve the required frequency resolution to detect the DTMF frequencies within $\pm 1.5\,\%$, a 256-point FFT is needed for 8 kHz sample rate. Since the DTMF detection considers only eight frequencies, it is more efficient to use a filterbank that consists of eight IIR bandpass filters. In this chapter, we will introduce the modified Goertzel algorithm as filterbank for DTMF detection.

The DFT can be used to compute eight different $X(k)$ that correspond to the DTMF frequencies as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}. \tag{9.1}$$

Using the modified Goertzel algorithm, the DTMF decoder can be implemented as a matched filter for each frequency index k as illustrated in Figure 9.2, where $x(n)$ is the input signal, $H_k(z)$ is the transfer function of the kth filter, and $X(k)$ is the corresponding filter output.

From Equation (7.4), we have

$$W_N^{-kN} = e^{j(2\pi/N)kN} = e^{j2\pi k} = 1. \tag{9.2}$$

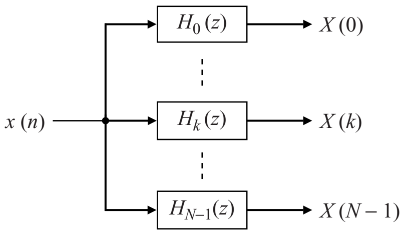


Figure 9.2 Block diagram of Goertzel filterbank

Multiplying the right-hand side of Equation (9.1) by W_N^{-kN}, we have

$$X(k) = W_N^{-kN} \sum_{n=0}^{N-1} x(n) W_N^{kn} = \sum_{n=0}^{N-1} x(n) W_N^{-k(N-n)}. \tag{9.3}$$

Define the sequence

$$y_k(n) = \sum_{m=0}^{N-1} x(m) W_N^{-k(n-m)}. \tag{9.4}$$

This equation can be interpreted as a convolution between the finite-duration sequence $x(n)$ and the sequence $W_N^{-kn}u(n)$ for $0 \le n \le N-1$. Consequently, $y_k(n)$ can be viewed as the output of a filter with impulse response $W_N^{-kn}u(n)$. That is, a filter with impulse response

$$h_k(n) = W_N^{-kn}u(n). \tag{9.5}$$

Thus, Equation (9.4) can be expressed as

$$y_k(n) = x(n) * W_N^{-kn}u(n). \tag{9.6}$$

From Equations (9.3) and (9.4), and the fact that $x(n) = 0$ for $n < 0$ and $n \ge N$, we can show that

$$X(k) = y_k(n)|_{n=N-1}. \tag{9.7}$$

That is, $X(k)$ is the output of filter $H_k(z)$ at time $n = N-1$.

Taking the z-transform of Equation (9.6), we obtain

$$Y_k(z) = X(z)\frac{1}{1 - W_N^{-k}z^{-1}}. \tag{9.8}$$

The transfer function of the kth Goertzel filter is defined as

$$H_k(z) = \frac{Y_k(z)}{X(z)} = \frac{1}{1 - W_N^{-k}z^{-1}}, k = 0, 1, \ldots, N-1. \tag{9.9}$$

This filter has a pole on the unit circle at the frequency $\omega_k = 2\pi k/N$. Thus, the DFT can be computed by filtering a block of input data using N filters in parallel as defined by Equation (9.9). Each filter has a pole at the corresponding frequency of the DFT.

The parameter *N* must be chosen to ensure that *X(k)* is the result representing to the DTMF at frequency f_k that meets the requirement of frequency tolerance given by Table 9.1. The DTMF detection accuracy can be ensured only if we choose the N such that the following approximation is satisfied:

$$\frac{2f_k}{f_s} \cong \frac{k}{N}. \tag{9.10}$$

A block diagram of the transfer function $H_k(z)$ for recursive computation of $X(k)$ is depicted in Figure 9.3. Since the coefficients W_N^{-k} are complex valued, the computation of each new value of $y_k(n)$ requires four multiplications and additions. All the intermediate values, $y_k(0)$, $y_k(1), \ldots,$ and $y_k(N-1)$,

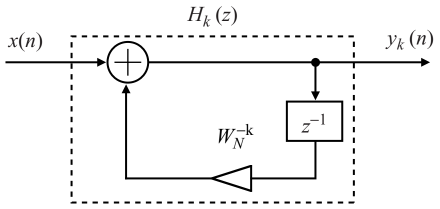


Figure 9.3 Block diagram of recursive computation of $X(k)$

must be computed in order to obtain the final output $y_k(N-1) = X(k)$. Therefore, the computation of $X(k)$ given in Figure 9.3 requires $4N$ complex multiplications and additions for each frequency index k.

We can avoid the complex multiplications and additions by combining the pairs of filters that have complex-conjugated poles. By multiplying both the numerator and the denominator of $H_k(z)$ given in Equation (9.9) by the factor $(1 - W_N^k z^{-1})$, we have

$$\begin{aligned} H_k(z) &= \frac{1 - W_N^k z^{-1}}{(1 - W_N^{-k} z^{-1})(1 - W_N^k z^{-1})} \\ &= \frac{1 - e^{j2\pi k/N} z^{-1}}{1 - 2\cos(2\pi k/N) z^{-1} + z^{-2}}. \end{aligned} \tag{9.11}$$

This transfer function can be represented as signal-flow diagram shown in Figure 9.4 using the direct-form II IIR filter. The recursive part of the filter is on the left side, and the nonrecursive part is on the right side. Since the output $y_k(n)$ is required only at time $N-1$, we just need to compute the nonrecursive part of the filter at the $(N-1)$th iteration. The recursive part of algorithm can be expressed as

$$w_k(n) = x(n) + 2\cos(2\pi f_k/f_s) w_k(n-1) - w_k(n-2), \tag{9.12}$$

while the nonrecursive calculation of $y_k(N-1)$ is expressed as

$$X(k) = y_k(N-1) = w_k(N-1) - e^{-j2\pi f_k/f_s} w_k(N-2). \tag{9.13}$$

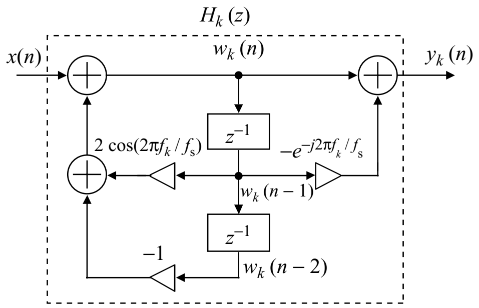


Figure 9.4 Detailed signal-flow diagram of Goertzel algorithm

The algorithm can be further simplified by realizing that only the squared $X(k)$ (magnitude) is needed for tone detections. From Equation (9.13), the squared magnitude of $X(k)$ is computed as

$$|X(k)|^2 = w_k^2(N-1) - 2\cos(2\pi f_k/f_s)w_k(N-1)w_k(N-2) + w_k^2(N-2). \tag{9.14}$$

This avoids the complex arithmetic given in Equation (9.13), and requires only one coefficient, $2\cos(2\pi f_k/f_s)$, for computing each $|X(k)|^2$. Since there are eight possible tones, the detector needs eight filters as described by Equations (9.12) and (9.14). Each filter is tuned to one of the eight frequencies. Note that Equation (9.12) is computed for $n = 0, 1, \ldots, N-1$, but Equation (9.14) is computed only once at time $n = N-1$.

9.2.3 Other DTMF Detection Methods

Goertzel algorithm is very efficient for DTMF signal detection. However, some real-world applications may already have other DSP modules that can be used for DTMF detection. For example, some noise reduction applications use FFT algorithm to analyze the spectrum of noise, and some speech-coding algorithms use the linear prediction coding (LPC). In these cases, the FFT or the LPC coefficients can be used for DTMF detection.

DTMF detection embedded in noise cancelation

In noise reduction systems that use spectrum subtraction method (will be introduced in Chapter 12), the time-domain signal is transformed to frequency domain using the FFT algorithm. Therefore, the FFT results can be used for DTMF detection as shown in Figure 9.5.

The system shown in Figure 9.5 shares the FFT results for noise cancelation and DTMF detection. Since frequency information is available from the FFT block, the DTMF detection can be simplified.

All-pole modeling using LPC coefficients

Chapter 11 will introduce many speech-coding algorithms using an all-pole LPC synthesis filter. The synthesis filter is defined as

$$\frac{1}{A(z)} = \frac{1}{1 - \sum_{i=1}^{p} a_i z^{-i}}, \tag{9.15}$$

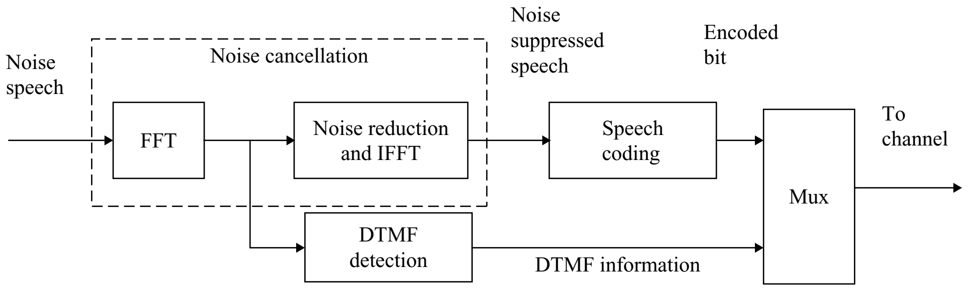


Figure 9.5 DTMF detection embedded in a noise cancelation system

where a_i is the short-term LPC coefficient and p is the LPC filter order. The calculation of LPC coefficients can be found in Section 11.4. This all-pole filter can be further decomposed with several second-order sections. If the LPC order p is an even number, it can be written as

$$\frac{1}{A(z)} = \frac{1}{(1 + a_{11}z^{-1} + a_{12}z^{-2})(1 + a_{21}z^{-1} + a_{22}z^{-2}) \cdots (1 + a_{q1}z^{-1} + a_{q2}z^{-2})} \tag{9.16}$$

with $q = p/2$. If p is an odd number with $q = (p - 1)/2$, the first-order component $(1 + a_{q+1}z^{-1})$ is used and Equation (9.16) can be modified as

$$\frac{1}{A(z)} = \frac{1}{(1 + a_{11}z^{-1} + a_{12}z^{-2}) \cdots (1 + a_{q1}z^{-1} + a_{q2}z^{-2})(1 + a_{q+1}z^{-1})}. \tag{9.17}$$

We assume that we have LPC coefficients and they are shared between a speech coder and a DTMF detector.

Example 9.1: Compare the similarity of the FFT spectrum of the DTMF digit '5' and the frequency response of a 10th-order LPC synthesis filter. The frequencies used for DTMF digit '5' are $f_L = 770$ Hz and $f_H = 1336$ Hz at sampling rate 8000 Hz, and the DTMF signal can be generated by MATLAB as

$$x(1:N) = \sin(2\pi f_L(1:N)) + \sin(2\pi f_H(1:N)).$$

Using MATLAB function `levinson`, we can compute the LPC coefficients from its autocorrelation function based on Equation (9.15) as follows:

```
lpcOrder=10;            % LPC order
w=hamming(N);           % Generate hamming window
x=x.*w';                % Windowing
m=0;
while (m<=lpcOrder);    % Calculation of auto-correlation
 r(m+1)=sum(x(1:(N-m)).*x((1+m):N)); m=m+1;
end;
a=levinson(r,lpcOrder); % Levinson algorithms
```

The generated LCP coefficients are listed as follows:

```
a[0] =  1.0000, a[1] = -1.5797, a[2]  = 1.4570, a[3] = -0.0021,
a[4] = -0.1805, a[5] =  0.1195, a[6]  = 0.3082, a[7] =  0.2145,
a[8] =  0.0230, a[9] = -0.0556, a[10] = 0.1797
```

Figure 9.6 shows the spectrum of DTMF tones for digit '5' and the spectrum from the LPC coefficients estimation. This example demonstrates that the roots of an all-pole filter, which represents the dual frequencies of DTMF tones, can be closely located using the LPC modeling.

Example 9.2: The roots of LPC synthesis filter coefficients can be computed using MATLAB function `roots(a)`. The angles of these roots can be converted from frequency in radian to Hz using MATLAB function `freq=((angle(r)*8000/(2*pi))`. The roots and angles are listed in Table 9.2 and the roots are also plotted in Figure 9.7.

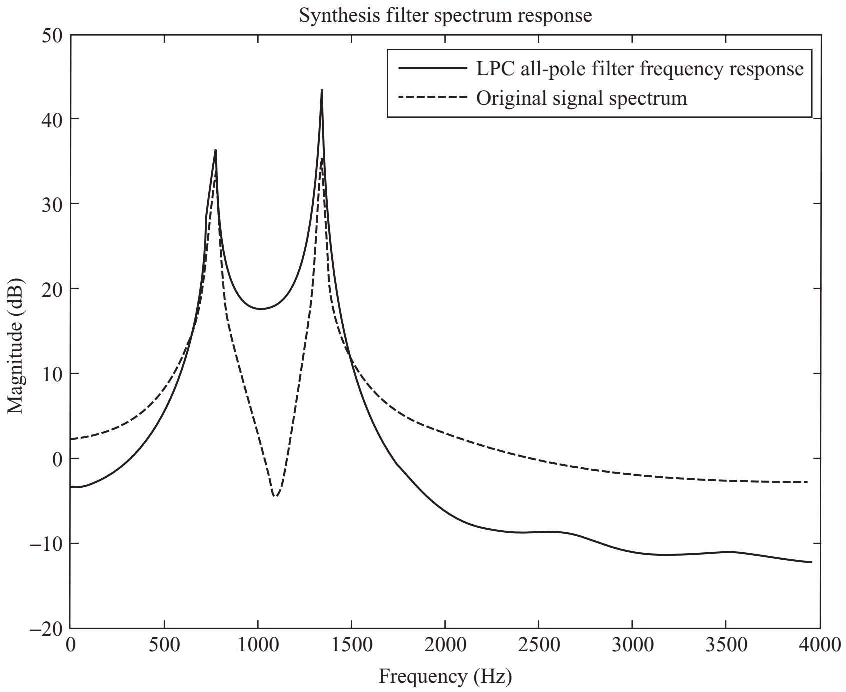


Figure 9.6 Comparison of spectrum estimated by LPC

The roots from Example 9.2 are complex-conjugated pairs. These roots represent five real second-order functions in Equation (9.16). The third and fourth pairs of roots are the most important since they are very close to the unit circle, and their frequencies are comparable to two frequencies used for digit '5'. The amplitudes of other roots are smaller since they are located inside the unit circle, and their frequencies are not within the DTMF frequency ranges. The roots with amplitudes close to unity dominate the magnitude response. For the example using DTMF digit '5', the relative differences in amplitude estimation are 0.0873 % for 770 Hz and 0.1274 % for 1336 Hz. The estimated frequency differences are 0.1799 % for 770 Hz and 0.1223 % for 1336 Hz. Examples 9.1 and 9.2 show that the estimated DTMF frequencies from LPC coefficients are very close to the DTMF frequencies defined by ITU Q.23 recommendation. Thus, the LPC coefficients from speech coders can be used for DTMF detection.

9.2.4 Implementation Considerations

The flow chart of DTMF tone detection algorithm is illustrated in Figure 9.8. At the beginning of each frame, the state variables $x(n)$, $w_k(n)$, $w_k(n-1)$, $w_k(n-2)$, and $y_k(n)$ for each of the eight Goertzel

Table 9.2 Roots and angles of 10th-order LPC synthesis filter

	Complex roots	Amplitude	Frequency (Hz)
1	$-0.6752 \pm j0.2510$	0.7204	±3546.8
2	$-0.3481 \pm j0.6506$	0.7378	±2625.5
3	$0.8225 \pm j0.5672$	0.9991	±0768.6
4	$0.4964 \pm j0.8666$	0.9987	±1337.6
5	$0.4942 \pm j0.6283$	0.7993	±1151.4

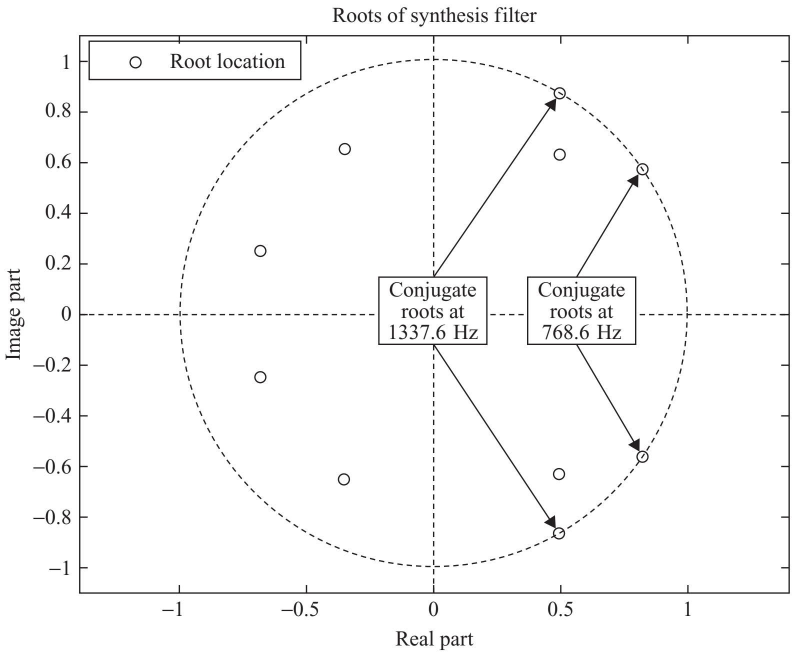


Figure 9.7 Plot of roots of 10th-order LPC synthesis filter

filters and the energy are set to zero. For each new sample, the recursive part of filtering defined in Equation (9.12) is executed. At the end of each frame, i.e., $n = N - 1$, the squared magnitude $|X(k)|^2$ for each DTMF frequency is computed based on Equation (9.14). Six tests are followed to determine if a valid DTMF digit has been detected:

Magnitude test: According to ITU Q.24, the maximum signal level transmit to the public network shall not exceed -9 dBm. This limits an average voice range of -35 dBm for a very weak long-distance call to -10 dBm for a local call. A DTMF receiver is expected to operate at an average range of -29 to $+1$ dBm. Thus, the largest magnitude in each band must be greater than a threshold of -29 dBm; otherwise, the DTMF signal should not be detected. For the magnitude test, the squared magnitude $|X(k)|^2$ defined in Equation (9.14) for each DTMF frequency is computed. The largest magnitude in each group is obtained.

Twist test: The tones may be attenuated according to the telephone system's gains at the tonal frequencies. Therefore, we do not expect the received tones to have same amplitude, even though they may be transmitted with the same strength. Twist is defined as the difference, in decibels, between the low- and high-frequency tone levels. In practice, the DTMF digits are generated with forward twist to compensate for greater losses at higher frequency within a long telephone cable. For example, Australia allows 10 dB of forward twist, Japan allows only 5 dB, and North America recommends not more than 8 dB of forward twist and 4 dB of reverse twist.

Frequency-offset test: This test prevents some broadband signals from being detected as DTMF tones. If the effective DTMF tones are present, the power levels at those two frequencies should be much higher than the power levels at the other frequencies. To perform this test, the largest magnitude in each group is compared to the magnitudes of other frequencies in that group. The difference must be greater than the predetermined threshold in each group.

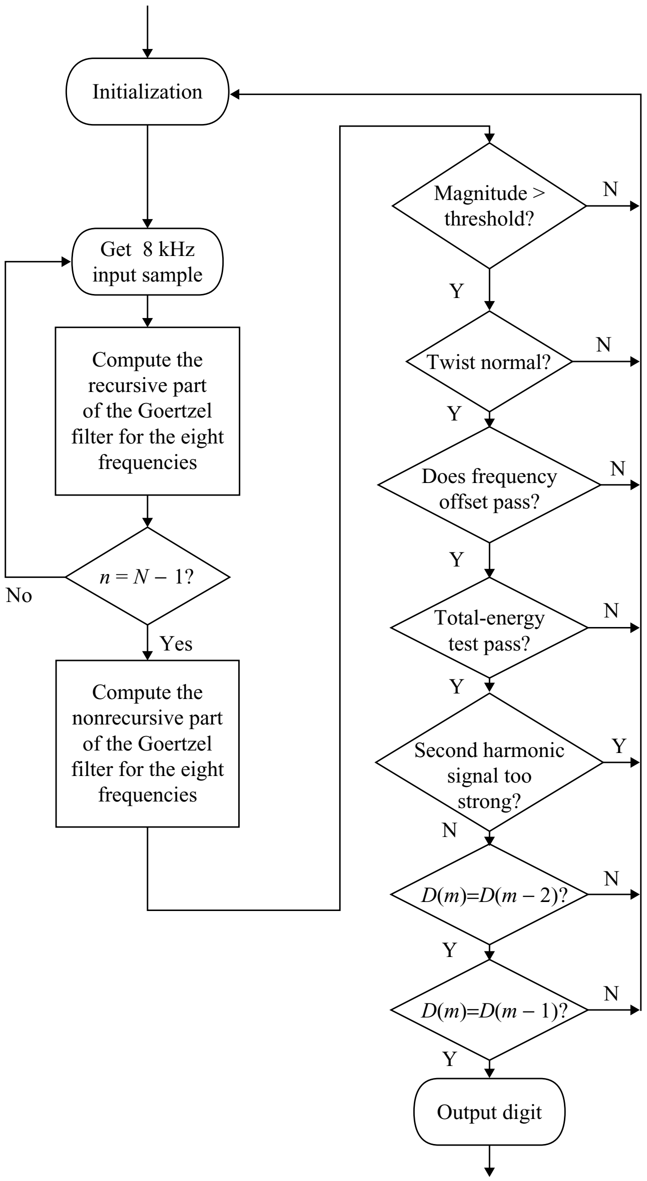

Initialization
Get 8 kHz input sample
Compute the recursive part of the Goertzel filter for the eight frequencies
$n = N - 1$?
No
Yes
Compute the nonrecursive part of the Goertzel filter for the eight frequencies
Magnitude > threshold?
Twist normal?
Does frequency offset pass?
Total-energy test pass?
Second harmonic signal too strong?
$D(m)=D(m-2)$?
$D(m)=D(m-1)$?
Y
N
Output digit

Figure 9.8 Flow chart for the DTMF tone detector

Total-energy test: Similar to the frequency-offset test, the goal of total-energy test is to reject some broadband signals to further improve the robustness of a DTMF decoder. To perform this test, three different constants c_1, c_2, and c_3 are used. The energy of the detected tone in the low-frequency group is weighted by c_1, the energy of the detected tone in the high-frequency group is weighted by c_2, and the sum of the two energies is weighted by c_3. Each of these terms must be greater than the summation of the energy from the rest of the filter outputs.

Second harmonic test: The objective of this test is to reject speech that has harmonics close to f_k so that they might be falsely detected as DTMF tones. Since DTMF tones are pure sinusoids, they contain very little second harmonic energy. Speech, on the other hand, contains a significant amount of second harmonic. To test the level of second harmonic, the detector must evaluate the second harmonic frequencies of all eight DTMF tones. These second harmonic frequencies (1394, 1540, 1704, 1882, 2418, 2672, 2954, and 3266 Hz) can also be detected using the Goertzel algorithm.

Digit decoder: Finally, if all five tests are passed, the tone pair is decoded and mapped to one of the 16 keys on a telephone touch-tone keypad. This decoded digit is placed in a memory location designated $D(m)$. If any of the tests fail, then '-1' representing 'no detection' is placed in $D(m)$. For a new valid digit to be declared, the current $D(m)$ must be the same in three successive frames, i.e., $D(m-2) = D(m-1) = D(m)$.

There are two reasons for checking three successive digits at each pass. First, the check eliminates the need to generate hits every time a tone is present. As long as the tone is present, it can be ignored until it changes. Second, comparing digits $D(m-2)$, $D(m-1)$, and $D(m)$ improves noise and speech immunity.

9.3 Internet Application Issues and Solutions

Voice coders have been designed for transmission of low-bit-rate signals while retaining reasonable audio quality and robustness over the networks. However, most of the vocoders overlook the importance of passing in-band tonal signals including DTMF. The approach for DTMF signaling over the networks can either use vocoders that are capable of passing in-band tones or use out-of-band signaling.

RFC2833 is a document for carrying DTMF signals and telephony events using RTP packets over the Internet. A separate RTP payload format is used due to the concern of low-rate vocoders that may not guarantee to reproduce the tone signals accurately for automatic recognition. The separate RTP payload format can be considered as the 'out-of-band' channel. Using separate payload formats also permits higher redundancy while maintaining a low-bit rate.

Figure 9.9 shows an example for Internet applications using DTMF generator and detector. The end user can use an automated computer system that recognizes the DTMF signaling and controls the access to the database, such as electronic bank accounts, voicemail systems, and automatic billing systems. Using the separated RTP payload allows the receiving side to recognize the in-band DTMF tones and regenerate these tones locally if necessary. RFC2833 also covers fax tones, modem tones, country-specific subscriber line tones, and trunk events.

The RTP payload type for vocoders will be discussed in Chapter 11. The DTMF typically uses dynamic payload type. The dynamic payload type means the type is negotiated using session description protocol between the two sides defined in RFC3551. Table 9.3 gives an example of the DTMF packet. The first 12 bytes are RTP header and the last 4 bytes are DTMF event. In the DTMF event data, the first byte

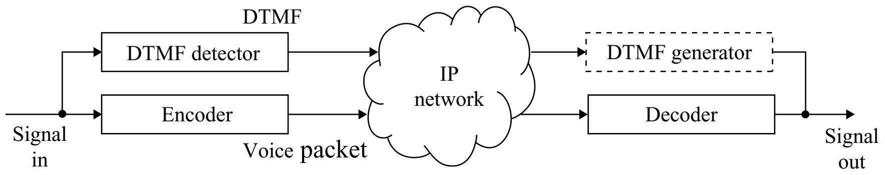


Figure 9.9 An example of DTMF detection and generation for Internet

Table 9.3 Example of RTP packet of DTMF digit '1'

```
80 62 f4 62 00 24 cb ac ac 24 a8 7a 01 08 00 a0
Packet summary lines                                      ; Data
Real-Time Transport Protocol                              ;
    10.. .... = Version: RFC 1889 Version (2)             ; 80
    ..0. .... = Padding: False                            ;
    ...0 .... = Extension: False                          ;
    .... 0000 = Contributing source identifiers count: 0  ;
    0... .... = Marker: False                             ; 62
    .110 0010 = Payload type: Unknown (98)                ;
    Sequence number: 62562                                ; f4 62
    Timestamp: 2411436                                    ; 00 24 cb ac
    Synchronization source identifier: 2888083578         ; ac 24 a8 7a
RFC 2833 RTP event
    Event ID: DTMF One 1 (1)                              ; 01
    0... .... = End of event: False                       ; 08
    .0.. .... = Reserved: False                           ;
    ..00 1000 = Volume: 8                                 ;
    Event duration: 160                                   ; 00 a0
```

`0x01` represents the digit '1', the last 6 bits of second byte, `0x08`, represent the volume of −8 dBm0, and the third and fourth bytes, `0x00a0`, represent the duration of 160 ms.

9.4 Experiments and Program Examples

In this section, we will use the MATLAB's CCS Link to connect MATLAB with the C5510 DSK for experiments on DTMF tone detection.

9.4.1 Implementation of Goertzel Algorithm Using Fixed-Point C

The Goertzel algorithm can be separated into two paths. The recursive path is defined by Equation (9.12). Table 9.4 lists the implementation of the recursive path in fixed-point C program. In the code, the pointer `d` points to the filter's delay-line buffer. The input is passed to the function by variable `in`. The variable `coef` is the filter coefficient. To prevent overflow, input data is scaled down by 7 bits. Note that the

Table 9.4 C implementation of Goertzel filter's recursive path

```c
void gFilter (short *d, short in, short coef)
{
  long AC0;
  d[0]  = in >> 7;  // Get input with scale down by 7 bit
  AC0   = (long) d[1] * coef;
  AC0 >>= 14;
  AC0  -= d[2];
  d[0] += (short)AC0;
  d[2]  = d[1];     // Update delay-line buffer
  d[1]  = d[0];
}
```

Table 9.5 C implementation of nonrecursive path of Goertzel filter

```
short computeOutput(short *d, short coef)
{
  long AC0, AC1;
  AC0 = (long) d[1] * d[1];  // Square d[1]
  AC0 += (long) d[2] * d[2]; // Add square d[2]
  AC1 = (long) d[1] * coef;
  AC1 >>= 14;
  AC1 = AC1 * d[2];
  AC0 -= AC1;
  d[1] = 0;
  d[2] = 0;
  return ((short)(AC0 >> 14));
}
```

fixed-point C implementation requires the data type conversion to be cast in `long` for multiplication, and the 14-bit shift is used to align the multiplication product to be stored in Q15 format. The recursive path calculation is carried out for every data sample.

The nonrecursive path of Goertzel filter described by Equation (9.14) is used once for every other N sample. The calculation of the final Goertzel filter output is implemented in C as shown in Table 9.5.

Table 9.6 lists the files used for this experiment. The test program `DTMFdecodeTest.c` opens the test parameter file `param.txt` to get the DTMF data filenames. This experiment analyzes the input data file and reports the detected DTMF digits in the output file `DTMFKEY.txt`.

Procedures of the experiment are listed as follows:

1. Open C55 project `fixedPoint_DTMF.pjt`, build, load, and run the program.
2. Examine the DTMF detection results to validate the correctness of the decoding process.
3. Modify DTMF generation experiment given in Section 8.4.4 to generate DTMF signals that can be used for testing DTMF magnitude level and frequency offset.

Table 9.6 File listing for experiment `exp9.4.1_fixedPointDTMF`

Files	Description
`DTMFdecodeTest.c`	C function for testing experiment
`gFilter.c`	C function computes recursive path
`computeOutput.c`	C function computes nonrecursive path
`dtmfFreq.c`	C function calculates all frequencies
`gFreqDetect.c`	C function maps frequencies to keypad row and column
`checkKey.c`	C function reports DTMF keys
`init.c`	C function for initialization
`dtmf.h`	C header file
`fixedPoint_DTMF.pjt`	DSP project file
`fixedPoint_DTMF.cmd`	DSP linker command file
`param.txt`	Parameter file
`DTMF16digits.pcm`	Data file containing 16 digits
`DTMF_with_noise.pcm`	Data file with noise

4. Modify the program to perform the magnitude test as described in Section 9.2.4.

5. Modify the program to perform the frequency test as described in Section 9.2.4.

9.4.2 Implementation of Goertzel Algorithm Using C55x Assembly Language

The efficient implementation of DTMF detection has been a design challenge for multichannel real-time applications. This experiment presents the implementation of Goertzel algorithm using the TMS320C55x assembly language. Table 9.7 lists the assembly routine that implements the Goertzel algorithm. The input data sample is passed in by register T0. The right-shifted 7 bits scale the input signal to prevent possible overflow during the filtering process. The Goertzel filter coefficient is stored in register T1. The auxiliary register AR0 is the pointer to `d[3]` in the delay-line buffer. The Goertzel filtering routine `gFilter` is called for every data sample.

After the recursive path has processed $N - 1$ samples, the final Goertzel filtering result will be computed at the Nth iteration for the nonrecursive path. Table 9.8 shows the assembly routine that computes the final Goertzel filter output. The caller passes two arguments, the auxiliary register AR0 is the pointer to the delay line `d[3]`, and register T0 contains the Goertzel filter coefficient of the given frequency. The final Goertzel filter output is returned to the caller via register T0 at the end of the routine. Table 9.9 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Open the project `asm_DTMF.pjt`, build, load, and run the program.

2. Examine the DTMF detection results to validate the correctness of the decoding process.

3. Modify DTMF generation experiment given in Section 8.4.4 to generate DTMF signals that can be used for testing DTMF twist level and the second harmonics.

Table 9.7 Assembly language implementation of Goertzel filter

```
   .global _gFilter
_gFilter:
    mov T0, AC0
    sfts AC0, #-7        ; d[0]  = in >> 7
    mov AC0, *AR0+
    mov *AR0+, AR1
    mov AR1, HI(AC1)
    mpy T1, AC1          ; AC0   = (long) d[1] * coef
    sfts AC1, #-14       ; AC0 >>= 14
    sub *AR0-, AC1, AR2 ; AC0  -= d[2]
    amar *AR0-
||  add AC0, AR2
    mov AR2, *AR0+       ; d[0] += (short)AC0
    mov AR2, *AR0+       ; d[1] = d[0]
    mov AR1, *AR0        ; d[2] = d[1]
    ret
```

Table 9.8 Assembly routine to compute the Goertzel filter output

```
    .global _computeOutput
_computeOutput:
     amar  *AR0+
     mpym  *AR0+, T0, AC1 ; AC1 = (long) d[1] * coef
     sfts  AC1, #-14      ; AC1 >>= 14;
     mov   AC1, T0
     mpym  *AR0-, T0, AC0 ; AC1  = AC1 * d[2]
     sqrm  *AR0+, AC1     ; AC0  = (long) d[1] * d[1];
     sqam  *AR0-, AC1     ; AC0 += (long) d[2] * d[2];
     sub   AC0, AC1       ; AC0 -= AC1
||   mov   #0, *AR0+      ; d[1] = 0
     sfts  AC1, #-14, AC0
||   mov   #0, *AR0       ; d[2] = 0
     mov   AC0, T0        ; out  = (short)(AC0 >> 14);
     ret
```

4. Use CCS profile tool to compare the number of cycles used between this assembly implementation and fixed-point C implementation given in Section 9.4.1.

9.4.3 DTMF Detection Using C5510 DSK

In this experiment, we will use MATLAB Link for CCS with the C5510 DSK to conduct the DTMF detection experiment. The flow of experiment is shown in Figure 9.10. Some of the frequently used CCS control commands that MATLAB supports are listed in Table 9.10.

We modified the MATLAB script of DTMF generator given in Chapter 8 for this experiment. Go through the following steps to create a new GUI for DTMF experiment:

1. Start MATLAB and set path to `..\experiments\exp9.4.3_MATLABCCSLink`. Copy the MATLAB files `DTMFGenerator.m` and `DTMFGenerator.fig` from Chapter 8 to current folder.

Table 9.9 File listing for experiment `exp9.4.2_asmDTMF`

Files	Description
`DTMFdecodeTest.c`	C function for testing experiment
`gFilter.asm`	Assembly function computes recursive path
`computeOutput.asm`	Assembly function computes nonrecursive path
`dtmfFreq.asm`	Assembly function calculates all frequencies
`gFreqDetect.c`	C function maps frequencies to keypad
`checkKey.c`	C function reports DTMF keys
`init.c`	C function for initialization
`dtmf.h`	C header file
`asm_DTMF.pjt`	DSP project file
`asm_DTMF.cmd`	DSP linker command file
`param.txt`	Parameter file
`DTMF16digits.pcm`	Data file containing 16 digits
`DTMF_with_noise.pcm`	Data file with noise

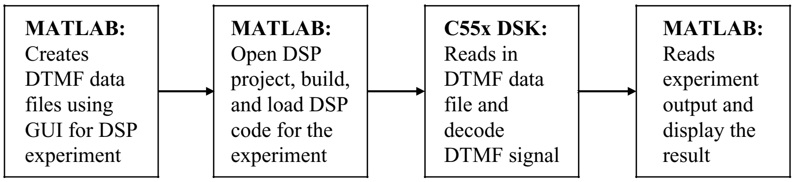

MATLAB: Creates DTMF data files using GUI for DSP experiment
MATLAB: Open DSP project, build, and load DSP code for the experiment
C55x DSK: Reads in DTMF data file and decode DTMF signal
MATLAB: Reads experiment output and display the result

Figure 9.10 MATLAB Link for CCS experiment flow

2. Enter the command `guide` at MATLAB command window to start **Guide Quick Start** menu. Choose **Open Exist GUI** tab to open file `DTMFGenerator.fig`. Add a new button key named **Decode DTMF** as shown in Figure 9.11.

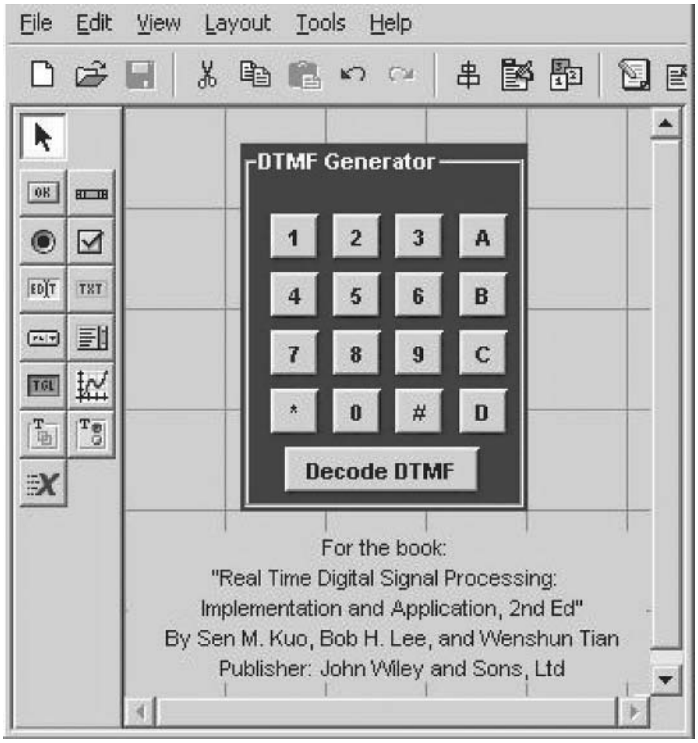
3. Save the GUI as `DTMF.fig` and a new file `DTMF.m`. The file `DTMF.m` is the same as `DTMFGenerator.m` except a new callback function is appended at the end. Table 9.11 shows partial code of the `DTMF.m`. The MATLAB files `DTMF.fig` and `DTMF.m` are used to control the project to conduct the DSK experiment.

4. Modify the function `dtmfGen( )` in `DTMF.m` to include the capability of saving keypress into a PCM file as the DTMF signal. The modified DTMF generator is listed as follows:

```
% --- DTMF signal generation
function x=dtmfGen(fl, fh)
fs = 8000;
N = [0:1/fs:0.1];
x = 0.5*(cos(2*pi*fl*N)+cos(2*pi*fh*N));
sound(x,fs)
x = int16(x*16383);
fid=fopen('.\\DTMF\\data\\data.pcm', 'ab'); % Write DTMF data
fwrite(fid, x, 'int16');
x = zeros(1, 400);
fwrite(fid, x, 'int16');
fclose(fid);
```

Table 9.10 MATLAB Link for CCS functions

MATLAB function	CCS command and status
`build`	Build the active project in CCS IDE
`ccsboardinfo`	Obtain information about the boards and simulators
`ccsdsp`	Create the link to CCS IDE
`clear`	Clear the link to CCS IDE
`halt`	Stop execution on the target board or simulator
`isrunning`	Check if the DSP processor is running
`load`	Load executable program file to target processor
`read`	Read global variables linked with CCS Link object
`reset`	Reset the target processor
`restart`	Place program counter to the entry point of the program
`run`	Execute program loaded on the target board or simulator
`visible`	Control the visibility for CCS IDE window
`write`	Write data to global variables linked with CCS Link object



For the book:
"Real Time Digital Signal Processing:
Implementation and Application, 2nd Ed"
By Sen M. Kuo, Bob H. Lee, and Wenshun Tian
Publisher: John Wiley and Sons, Ltd

Figure 9.11 MATLAB GUI for DTMF detection experiment

This modified function records each DTMF signal to a PCM file `data.pcm`. The duration of each DTMF signal is 100 ms followed by 50 ms of silence.

5. Add MATLAB Link for CCS code to the `DTMF.m`. Table 9.12 lists the MATLAB script.

This MATLAB script controls the execution of DSK. The function `ccsboardinfo` checks the DSP development system and returns the information regarding the board and processor that it has identified.

Table 9.11 MATLAB script `DTMF.m` generated by GUI editor

```
% --- DTMF signal generation
function x=dtmfGen(fl, fh)
fs = 8000;
N = [0:1/fs:0.1];
x = 0.5*(cos(2*pi*fl*N)+cos(2*pi*fh*N));
sound(x,fs)

% --- Executes on button press in pushbutton17
function pushbutton17_Callback(hObject, eventdata, handles)
```

Table 9.12 MATLAB script using Link for CCS to command the C5510 DSK

```
board = ccsboardinfo;           % Get DSP board & processor information
dsp = ccsdsp('boardnum',...     % Link DSP with CCS
              board.number,
             'procnum',
              board.proc(1,1).number);
set(dsp,'timeout',100);         % Set CCS timeout value to 100(s)
visible(dsp,1);                 % Force CCS to be visible on desktop
open(dsp,'DTMF\\ccsLink.pjt');  % Open project file
build(dsp,1500);                % Build the project if necessary
load(dsp, '.\\DTMF\\Debug\\ccsLink.out',300);
                                % Load project with timeout 300(s)
reset(dsp);                     % Reset the DSP processor
restart(dsp);                   % Restart the program
run(dsp);                       % Start execution or wait new command
cpurunstatus = isrunning(dsp);
while cpurunstatus == 1,        % Wait until processor completes task
    cpurunstatus = isrunning(dsp);
end
```

The `ccsdsp` function creates the link object to CCS using the information obtained from the function call to `ccsboardinfo`. The functions `open`, `build`, `load`, `reset`, `restart`, and `run` are the standard CCS commands that control the CCS IDE functions and status. The function `run` consists of several options. The option `main` is the same as CCS command **Go Main**. The option `tohalt` will start DSP processor and run until the program reaches a breakpoint or it is halted. The option `tofunc` will start and run the DSP processor until the program reaches the given function. The `build` function also has multiple options. The default build function makes an incremental build, while the option `all` will perform CCS command **Rebuild All**. In this experiment, the function `isrunning` is used to check if the DSK processing is completed.

The software for DTMF decoder using MATLAB Link for CCS includes the DSP project, source files, and MATLAB script files. Table 9.13 lists the files used for this experiment.

Table 9.13 File listing for experiment `exp9.4.3_MATLABCCSLink`

Files	Description
`DTMF.m`	MATLAB script for testing experiment
`DTMF.fig`	MATLAB GUI
`DTMFdecodeTest.c`	DTMF experiment test file
`gFilter.asm`	Assembly function computes recursive path
`computeOutput.asm`	Assembly function computes nonrecursive path
`dtmfFreq.asm`	Assembly function calculates all frequencies
`gFreqDetect.c`	C function maps frequencies to keypad
`checkKey.c`	C function reports DTMF keys
`Init.c`	C function for initialization
`dtmf.h`	C header file
`ccsLink.pjt`	DSP project file
`ccsLink.cmd`	DSP linker command file
`dtmfGen.m`	MATLAB function for DTMF tone generation
`dspDTMf.m`	MATLAB function for commanding C55xCCS

In this experiment, MATLAB command window will show each key that is pressed and display the DTMF detection result. Procedures of the experiment are listed as follows:

1. Connect DSK to the computer and power on the DSK.
2. Create and build the DSP project for the experiment. If no errors, exit CCS.
3. Start MATLAB and set the path to the directory `..\exp9.4.3_MATLABCCSLink`.
4. Type `DTMF` at MATLAB command window to display the DTMF experiment GUI.
5. Press several DTMF keys to generate a DTMF sequence.
6. Press the **Decode DTMF** key on the GUI to start CCS, build the DSP project, and then run the DTMF decoder.
7. Multichannel DTMF detection is widely used in industries. Modify the experiment such that it performs two-channel DTMF detection in parallel. The input data for the two-channel DTMF detection can be generated in time-division method. Since this experiment reads the input data from a PCM data file, we can create two DTMF signaling files and read both of them in for the two-channel experiment.

9.4.4 DTMF Detection Using All-Pole Modeling

In this experiment, we will present the MATLAB script to show the basic concept of DTMF detection using the LPC all-pole modeling. Table 9.14 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Copy the MATLAB files `DTMF.fig` and `DTMF.m` from the previous experiment to the directory `..\exp9.4.4_LPC`.
2. Modify `DTMF.m` to replace the Link for CCS function by the code listed in Table 9.15. This function reads the DTMF data. The all-pole function is implemented using the Levinson algorithm to avoid direct matrix inversion in computing the autocorrelation and LPC coefficients. The roots of LPC coefficients are calculated using the MATLAB function `roots`. Finally, the amplitude and angles are analyzed and compared to detect DTMF tones.
3. The user interface is the same as the previous experiment. Press DTMF keys to generate a PCM file. Press the **Decode DTMF** key to start the decoder.

Table 9.14 File listing for experiment `exp9.4.4_LPC`

Files	Description
`DTMF.m`	MATLAB script for testing experiment
`DTMF.fig`	MATLAB GUI

Table 9.15 MATLAB code for LPC all-pole modeling

```
N=256;               % Length of FFT
fs=8000;             % Sampling frequency
f=fs*(0:(N/2-1))/N;  % Frequency scale for display
KEY = 1:16;          % Keypad map lookup table index
%           col| row
  KEY(1+1) =0016+0001;
  KEY(1+2) =0032+0001;
  KEY(1+3) =0064+0001;
  KEY(1+10)=0128+0001;
  KEY(1+4) =0016+0002;
  KEY(1+5) =0032+0002;
  KEY(1+6) =0064+0002;
  KEY(1+11)=0128+0002;
  KEY(1+7) =0016+0004;
  KEY(1+8) =0032+0004;
  KEY(1+9) =0064+0004;
  KEY(1+12)=0128+0004;
  KEY(1+14)=0016+0008;
  KEY(1+0) =0032+0008;
  KEY(1+15)=0064+0008;
  KEY(1+13)=0128+0008;

% Table lookup for Keys
DIGIT =['0','1','2','3','4','5','6','7','8','9','A','B','C','D','*','#'];

freq = [697 770 852 941 1209 1336 1477 1633];
digi = [1 2 4 8 16 32 64 128];
lpcOrder=10;            % LPC order
w=hamming(N);           % Generate Hamming window
fid=fopen('.\\data\\data.pcm','r'); % Open the PCM data file
prevDigit = 0;
while ~ feof(fid)
  x = fread(fid,N,'short');
  if size(x) ~ = N
    continue;
  end
  % Check energy
  if sum(abs(x)) <= 200000
    prevDigit = 0;
  else
    % Compute autocorrelation
    x=x.*w;                  % Windowing
    m=0;
    while (m<=lpcOrder);
      r(m+1)=sum(x(1:(N-m)).*x((1+m):N)); m=m+1;
    end;
    a=levinson(r,lpcOrder);  % Levinson algorithm
    % Calculate root
    r=roots(a);              % Find roots
    amp=abs(r);              % Get amplitudes
    ang=(angle(r)*fs/pi/2);  % Get angles
    % Compare with the table
```

Table 9.15 (*continued*)

```
    AmpThreahold = 0.98;      % 0.02%
    AngThreahold = 5;         % 5 Hz
    dtmf =0;
    for i=1:2:(lpcOrder)
      if abs(amp(i)) >= AmpThreahold
        for j = 1:8
          if (abs(ang(i)) <= (freq(j)+AngThreahold))
            if (abs(ang(i)) >= (freq(j)-AngThreahold))
              dtmf = dtmf + digi(j);
            end
          end
        end
      end
    end
    % Check if dtmf detected
    dtmfDet=0;
    for i=1:16
      if dtmf == KEY(i)
        dtmfDet =i;
      end
    end
    % Display result
    if dtmfDet ~ = 0
      if (DIGIT(dtmfDet) ~ = prevDigit)
        disp(sprintf('%s is detected', DIGIT(dtmfDet)));
        prevDigit = DIGIT(dtmfDet);
      end
    else
      prevDigit = 0;
    end
  end
end
fclose(fid);
```

4. Validate the DTMF digits displayed on MATLAB command window are correctly decoded.
5. If the all-pole filter order is 4, is it possible to find the root of the filter that matches the DTMF frequency? Modify the experiment to validate your claim.

References

[1] ITU-T Recommendation Q.23, *Technical Features of Push-Button Telephone Sets*, 1993.
[2] ITU-T Recommendation Q.24, *Multifrequency Push-Button Signal Reception*, 1993.
[3] 3GPP TR-T12-26.975 V6.0.0, *Performance Characterization of the Adaptive Multi-Rate (AMR) Speech Codec* (Release 6), Dec. 2004.
[4] TI Application Report, *DTMF Tone Generation and Detection – An Implementation Using the TMS320C54x*, SPRA 096A, May 2000.
[5] W. Tian and Y. Lu, 'System and method for DTMF detection using likelihood ratios,' US Patent no. 6,873,701, Mar. 2005.

[6] Y. Lu and W. Tian, 'DTMF detection based on LPC coefficients,' US Patent no. 6,590,972, July 2003.
[7] F. F. Tzeng, 'Dual-tone multifrequency (DTMF) signaling transparency for low-data-rate vocoder,' US Patent no. 5,459,784, Oct. 1995.
[8] R. Rabipour and M. Beyrouti, 'LPC-based DTMF receiver for secondary signaling,' US Patent no. 4,853,958, Aug. 1989.
[9] C. Lee and D.Y. Wong, 'Digital tone decoder and method of decoding tones using linear prediction coding,' US Patent no. 4,689,760, Aug. 1987.
[10] N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1983.
[11] MATLAB, Version 7.0.1 Release 14, Sep. 2004.
[12] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989.
[13] S. J. Orfanidis, *Introduction to Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1996.
[14] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd Ed., Englewood Cliffs, NJ: Prentice Hall, 1996.
[15] A Bateman and W. Yates, *Digital Signal Processing Design*, New York: Computer Science Press, 1989.
[16] J. Hartung, S. L. Gay, and G. L. Smith, *Dual-tone Multifrequency Receiver Using the WE DSP16 Digital Signal Processor*, Application Note, AT&T, 1988.
[17] Analog Devices, *Digital Signal Processing Applications Using the ADSP-2100 Family*, Englewood Cliffs, NJ: Prentice Hall, 1990.
[18] P. Mock, *Add DTMF Generation and Decoding to DSP-uP Designs*, *Digital Signal Processing Applications with the TMS320 Family*, Texas Instruments, 1986, Chap. 19.
[19] J. S. Lim and A. V. Oppenheim, 'Enhancement and bandwidth compression of noisy speech,' *Proc. of the IEEE*, vol. 67, Dec. 1979, pp. 1586–1604.
[20] J. R. Deller, Jr., J. G. Proakis, and J. H. L. Hansen, *Discrete-Time Processing of Speech Signals*, New York: MacMillan, 1993.
[21] H. Schulzrinne and S. Petrack, *RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals,* Request for Comments 2833 (RFC2833), May 2000.
[22] H. Schulzrinne and S. Casner, *RTP Profile for Audio and Video Conferences with Minimal Control*, IETF RFC3551, July 2003.

Exercises

1. According to Table 9.1, DTMF frequency tolerance for operation is $\leq 1.5\,\%$ and nonoperation is $\geq 3.5\,\%$. Calculate the operation and nonoperation frequency boundaries of all eight frequencies.

2. For N-point DFT, the frequency resolution is $\frac{f_s}{2N} = \frac{8000}{2N}$ at 8000 Hz sampling rate. In Goertzel algorithm, the signal frequency f_k is approximated by $f_s \frac{k}{2N}$. If N is not properly selected, the signal frequency f_k may be off more than 1.5 % due to the DFT algorithm. By using the frequency operation tolerance 1.5 %, calculate $N \in [180, 256]$ which makes all eight frequencies meet the requirement.

3. A female voice contains the strong harmonic with pitch frequency of 210 Hz. Which digit may be falsely registered? Explain why? Assume this DTMF detector meets the frequency tolerance requirement.

4. Besides Goertzel algorithm, an IIR filterbank can be used for DTMF detection. Design a new experiment that uses eight fourth-order IIR filters for DTMF detection. Profile the performance and compare it with the decoder that uses the Goertzel algorithm.

5. DTMF digits can also be used for low-rate data communications. One digit can be treaded as a 4-bit symbol. Assuming each DTMF digit is sent every 80 ms, calculate the bit rate per second.

10
Adaptive Echo Cancelation

Adaptive echo cancelation is an important application of adaptive filtering for attenuating undesired echoes. In addition to canceling the voice echo in long-distance links and acoustic echo in hands-free speakerphones, adaptive echo cancelers are also widely used in full-duplex data transmission over two-wire circuits, such as high-speed modems. This chapter focuses on voice echo cancelers for long-distance networks, VoIP (voice over Internet protocol) applications, and hands-free speakerphones.

10.1 Introduction to Line Echoes

One of the main problems associated with telephone communications is the echo due to impedance mismatches at various points in the networks. Such echoes are called line (or network) echoes. If the time delay between the original speech and the echo is short, the echo may not be noticeable. The deleterious effects of echoes depend upon their loudness, spectral distortion, and delay. In general, longer delay requires more echo attenuation.

A simplified telecommunication network is illustrated in Figure 10.1, where the local telephone is connected to a central office by a two-wire line in which both directions of transmission are carried on a single wire pair. The connection between two central offices uses the four-wire facility, which physically segregates the transmission by two facilities. This is because long-distance transmission requires amplification that is a one-way function. The four-wire transmission path may include various equipments, including switches, cross-connects, and multiplexers. A hybrid (H) located in the central office makes the conversion between the two-wire and four-wire facilities.

An ideal hybrid is a bridge circuit with the balancing impedance that is exactly equal to the impedance of the two-wire circuit. Therefore, it will couple all energy on the incoming branch of the four-wire circuit into the two-wire circuit. In practice, the hybrid may be connected to any of the two-wire loops served by the central office. Thus, the balancing network can provide only a fixed and compromise impedance match. As a result, some of the incoming signals from the four-wire circuit leak into the outgoing four-wire circuit, and return to the source as an echo shown in Figure 10.1. This echo requires special treatment if the round-trip delay exceeds 40 ms.

Example 10.1: For Internet protocol (IP) trunk applications that use IP packets to relay the time division multiplex (TDM) traffic, the round-trip delay can easily exceed 40 ms. Figure 10.2 shows an example of VoIP applications using a gateway in which the voice is converted from the TDM circuits to the IP packets.

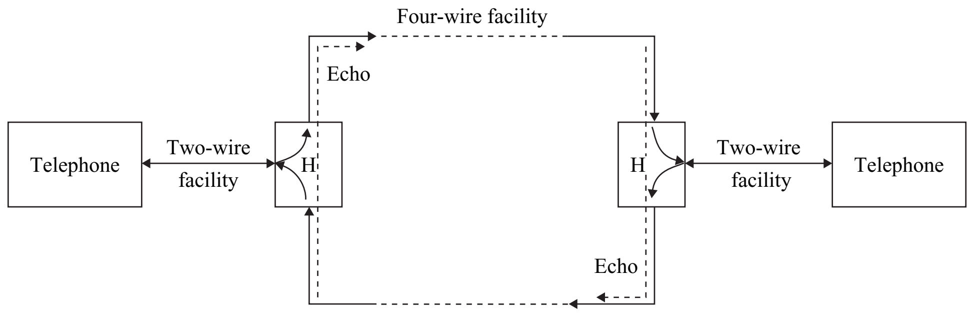


Figure 10.1 Long-distance telecommunication networks

The delay includes vocoders, jitter compensation algorithms, and the network delay. The ITU-T G.729 CODEC (will be introduced in Chapter 11) is widely used for VoIP applications for its good performance and low delay. The G.729 algorithm delay is 15 ms. If using 10-ms frame real-time protocol packet and 10-ms jitter compensation, the round-trip delay of G.729 will be at least $2 \times (15 + 10) = 50$ ms without counting the IP network delay and the processing delay. Such long delay is the reason why adaptive echo cancelation is required for VoIP applications if one or two ends are connected by TDM circuit.

10.2 Adaptive Echo Canceler

For telecommunication network using echo cancelation, the echo canceler is located in the four-wire section of the network near the origin of the echo source. The principle of the adaptive echo cancelation is illustrated in Figure 10.3. We show only one echo canceler located at the left end of network. To overcome the echoes in a full-duplex communication network, it is desirable to cancel the echoes in both directions of the trunk. The reason for showing a telephone and two-wire line is to indicate that this side is called the near-end, while the other side is referred to as the far-end.

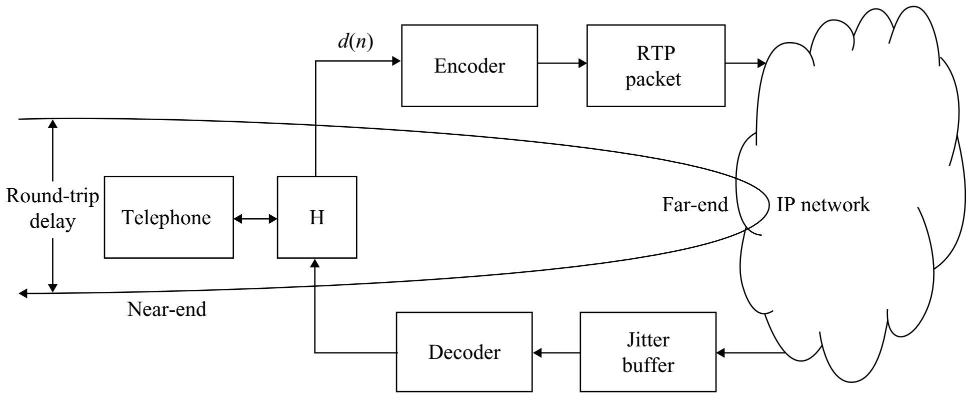


Figure 10.2 Example of round-trip delay for VoIP applications

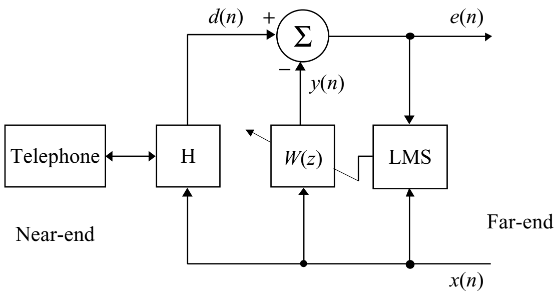


Figure 10.3 Block diagram of an adaptive echo canceler

10.2.1 Principles of Adaptive Echo Cancelation

To explain the principle of the adaptive echo cancelation in details, the function of the hybrid shown in Figure 10.3 can be illustrated in Figure 10.4, where the far-end signal $x(n)$ passing through the echo path $P(z)$ results in echo $r(n)$. The primary signal $d(n)$ is a combination of echo $r(n)$, near-end signal $u(n)$, and noise $v(n)$. The adaptive filter $W(z)$ models the echo path $P(z)$ using the far-end speech $x(n)$ as an excitation signal. The echo replica $y(n)$ is generated by $W(z)$, and is subtracted from the primary signal $d(n)$ to yield the error signal $e(n)$. Ideally, $y(n) \approx r(n)$ and the residual error $e(n)$ is substantially free of echo.

The impulse response of an echo path is shown in Figure 10.5. The time span over the hybrid is typically about 4 ms and is called the dispersive delay. Because of the four-wire circuit located between the location of the echo canceler and the hybrid, the impulse response of echo path has a flat delay. The flat delay depends on the transmission delay between the echo canceler and the hybrid, and the delay through the sharp filters associated with frequency division multiplex equipment. The sum of the flat delay and the dispersive delay is the tail delay.

Assuming that the echo path $P(z)$ is linear, time invariant, and with infinite impulse response $p(n)$, $n = 0, 1, \ldots, \infty$, the primary signal $d(n)$ can be expressed as

$$\begin{aligned} d(n) &= r(n) + u(n) + v(n) \\ &= \sum_{l=0}^{\infty} p(l)x(n-l) + u(n) + v(n), \end{aligned} \tag{10.1}$$

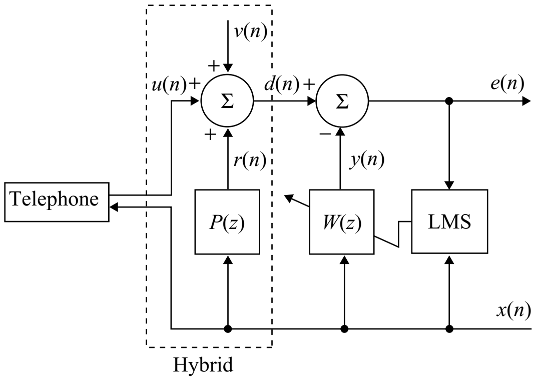


Figure 10.4 An echo canceler diagram with details of hybrid function

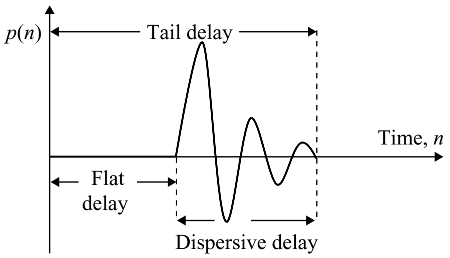


Figure 10.5 Impulse response of an echo path

where the additive noise $v(n)$ is assumed to be uncorrelated with the near-end speech $u(n)$ and the echo $r(n)$. The adaptive FIR filter $W(z)$ generates an echo estimation

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l), \tag{10.2}$$

which is used to cancel the echo. The error signal can be expressed as

$$\begin{aligned} e(n) &= d(n) - y(n) \\ &= \sum_{l=0}^{L-1} [p(l) - w_l(n)]x(n-l) + \sum_{l=L}^{\infty} p(l)x(n-l) + u(n) + v(n). \end{aligned} \tag{10.3}$$

The adaptive filter $W(z)$ adjusts its weights $w_l(n)$ to mimic the first L-sample impulse response of the echo path during the process of echo cancelation. The normalized LMS algorithm introduced in Section 7.3.4 is commonly used for adaptive echo cancelation applications. Assuming that disturbances $v(n)$ and voice $u(n)$ are uncorrelated with $x(n)$, $W(z)$ will converge to $P(z)$, i.e., $w_l(n) \approx p(l)$. As shown in Equation (10.3), the residual error after the $W(z)$ has converged can be expressed as

$$e(n) \approx \sum_{l=L}^{\infty} p(l)x(n-l) + u(n) + v(n). \tag{10.4}$$

By making the length of $W(z)$ sufficiently long, the residual echo can be minimized. However, as discussed in Section 7.3.3, the excess mean-square error (MSE) produced by the adaptive algorithm and finite-precision errors are also proportional to the filter length. Therefore, there is an optimum length L for echo cancelation.

The number of coefficients required for the FIR filter is determined by the tail delay. As mentioned earlier, the impulse response of the hybrid (dispersive delay) is relatively short. However, the flat delay from the echo canceler to the hybrid depends on the physical location of the echo canceler, and the processing delay of transmission equipments.

10.2.2 Performance Evaluation

The effectiveness of an echo canceler is usually measured by the echo return loss enhancement (ERLE) defined as

$$\text{ERLE} = 10\log\left\{\frac{E\left[d^2(n)\right]}{E\left[e^2(n)\right]}\right\}. \tag{10.5}$$

For a given application, the ERLE depends on the step size μ, the filter length L, the signal-to-noise ratio, and the nature of signal in terms of power and spectral contents. A larger step size provides a faster initial convergence, but the final ERLE is smaller due to the excess MSE and quantization errors. If the filter length is long enough to cover the length of echo tail, increasing L will further reduce the ERLE.

The ERLE achieved by an echo canceler is limited by many practical factors. Detailed requirements of an adaptive echo canceler are defined by ITU-T Recommendations G.165 and G.168, including the maximum residual echo level, the echo suppression effect on the hybrid, the convergence time, the initial setup time, and the degradation in a double-talk situation.

The first special-purpose chip for echo cancelation implements a single 128-tap adaptive echo canceler [5]. In the past, adaptive echo cancelers were implemented using customized devices in order to handle the heavy computation in real-time applications. Disadvantages of VLSI (very large scale integrated circuit) implementation are long development time, high development cost, and lack of flexibility to meet application-specific requirements and improvements such as the standard changed from G.165 to G.168. Therefore, recent activities in the design and implementation of adaptive echo cancelers for real-world applications are focus on programmable DSP processors.

10.3 Practical Considerations

This section discusses two practical techniques in designing adaptive echo canceler: prewhitening and delay detection.

10.3.1 Prewhitening of Signals

As discussed in Chapter 7, convergence time of adaptive FIR filter with the LMS algorithm is proportional to spectral ratio $\lambda_{max}/\lambda_{min}$. Thus, an input signal with flat spectrum such as white noise has faster convergence rate. Since speech signal is highly correlated with nonflat spectrum, the convergence speed is slow. The decorrelation (whitening) of input signal will improve the convergence speed.

Figure 10.6 shows a typical prewhitening structure for input signals. The whitening filter $F_W(z)$ is used for both the far-end and near-end signals. This fixed filter could be obtained using the reversed statistical or temporal averaged spectrum values. One example is the anti-tile filter used to lift up the high-frequency components since most speech signal's power is concentrated at the low-frequency region.

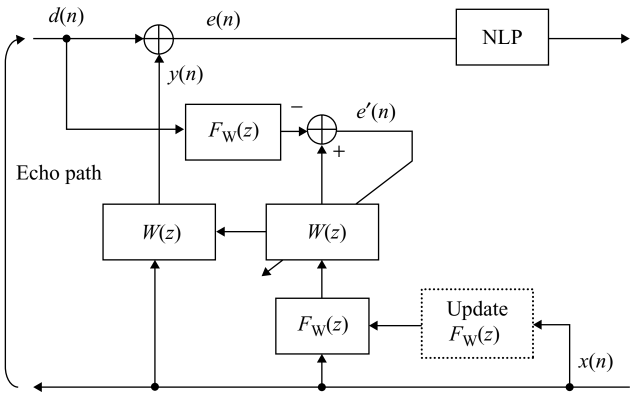


Figure 10.6 Block diagram of a signal prewhitening structure

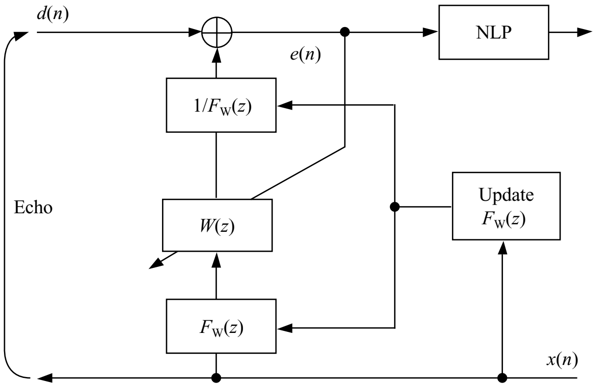


Figure 10.7 Block diagram of an adaptive prewhitening structure

The whitening filter can be adaptive based on the far-end signal $x(n)$. In this case, the filter coefficients update is synchronized for both branches. An equivalent structure of adaptive prewhitening is shown in Figure 10.7.

The adaptation of the whitening filter coefficients is similar to a perceptive weighting filter, which will be discussed in Chapter 11. We can use the Levinson–Durbin algorithm to estimate the input signal spectrum envelope and use the reversed function to filter the signal. The calculation of transfer function $F_W(z)$ is similar to the adaptive channel equalization discussed in Chapter 7. However, two conditions must be satisfied: the processing should be linear and the filter $F_W(z)$ must be reversible.

10.3.2 Delay Detection

As discussed in Section 10.2.1, the initial part of the impulse response of an echo path represents a transmission delay between the echo canceler and the hybrid. To take advantage of this flat delay, the structure illustrated in Figure 10.8 was developed. Here, Δ represents the number of flat-delay samples. By estimating the number of zero coefficients needed to cover the flat delay, the echo canceler $W(z)$ length can be shortened by Δ. This technique effectively reduces the computational requirements. However, there are three major difficulties: the existence of multiple echoes, the difficulty to estimate the flat delay, and the delay variation during a call.

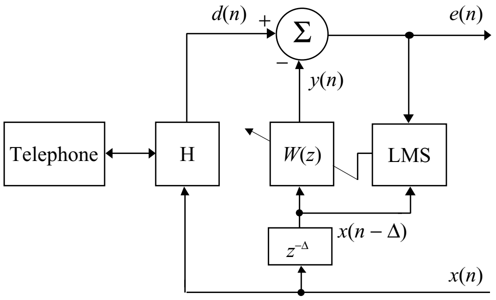


Figure 10.8 Adaptive echo canceler with effective flat-delay compensation

The crosscorrelation function between the far-end signal $x(n)$ and the near-end signal $d(n)$ can be used to estimate the delay. The normalized crosscorrelation function at time n with lag k can be estimated as

$$\rho(n,k) = \frac{r_{xd}(n,k)}{\sqrt{r_{xx}(n)r_{dd}(n)}}, \tag{10.6}$$

where $r_{xd}(n,k)$ is the crosscorrelation function defined as

$$r_{xd}(n,k) = \frac{1}{L}\sum_{l=0}^{L-1} x\,(n-(l+k))\,d(n-l), \tag{10.7}$$

and the autocorrelation for $x(n)$ and $d(n)$ are defined as

$$r_{xx}(n) = \frac{1}{L}\sum_{l=0}^{L-1} x\,(n-l)\,x(n-l), \tag{10.8}$$

$$r_{dd}(n) = \frac{1}{L}\sum_{l=0}^{L-1} d(n-l)d(n-l). \tag{10.9}$$

The typical value of length L is between 128 and 256 for 8 kHz sampling rate.

The flat delay will be the lag k that makes the maximum normalized crosscorrelation function as defined by Equation (10.6). Unfortunately, this method may have poor performance for speech signals as shown in Figure 10.9, although it has a good performance for signals with the flat-spectra such as white noise.

To improve the performance of crosscorrelation method, a bandpass filter using two or three formants in the passband can be considered. This makes the crosscorrelation technique more reliable by whitening the input signals over the passband. The multirate filtering (introduced in Section 4.4) can be used to further reduce the computational load. The normalized crosscorrelation function $\rho(n,k)$ is then computed using the subband signals $u(n)$ and $v(n)$. With properly chosen bandpass filter and downsampling factor D, the downsampled subband signals $u(n)$ and $v(n)$ are closer to that of the white noise.

Figure 10.10 shows the performance improvement using a 16-band filterbank. The third subband signal is decimated by a factor of 16 and the downsampled signal is used to calculate the crosscorrelation function defined in Equation (10.7). Because of the downsampling operations, the flat-delay estimation is divided into two steps. The first step finds T_0 in the downsampled domain. This delay has a resolution of the downsampling factor D. The exact delay will be between $T_0 - D/2$ and $T_0 + D/2$. The second step finds the resolution T_1 using the original signal that has maximum value of $\rho(n,k)$. This two-step approach requires less computation since the first step works at lower sampling rate and the second step needs to perform only limited fine searches. It requires about $1/D$ of the crosscorrelation computational requirements (refer to [4] for details).

Example 10.2: For a typical impulse response of echo path shown in Figure 10.5, calculate the required number of FIR filter coefficients given that the flat delay is 15 ms, the dispersive segment is 10 ms, and the sampling rate is 8 kHz. For this specific example, the adaptive echo canceler is located at tandem switch as shown in Figure 10.11.

Since the flat delay is a pure delay, this part can be implemented using a tapped delay line as $z^{-\Delta}$ where $\Delta = 120$. The actual filter length is 80 to cover the 10 ms of dispersive segment. In this case, the FIR filter coefficients need to compensate only for the dispersive delay of hybrid rather than the flat delay between the hybrid and the echo canceler. The delay estimation becomes very important since this flat delay may change for different connections.

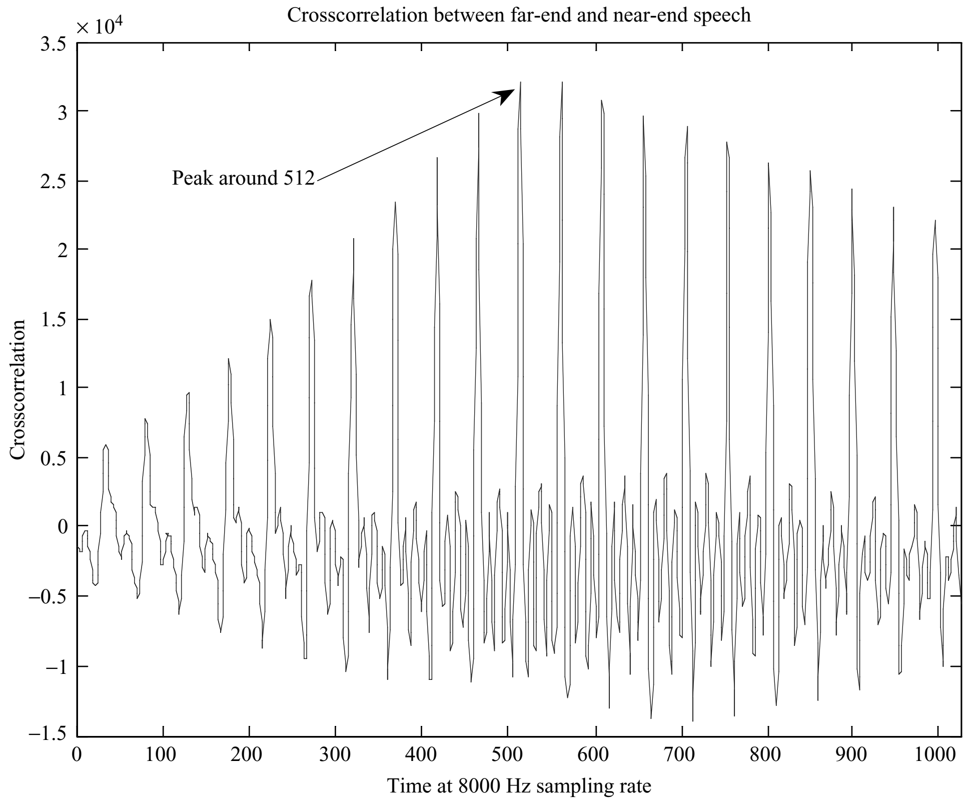


Figure 10.9 Crosscorrelation function of a voiced speech

10.4 Double-Talk Effects and Solutions

An extremely important issue of designing adaptive echo cancelers is to handle double talk, which occurs when the far-end and near-end talkers are speaking simultaneously. In this case, signal $d(n)$ consists of both echo $r(n)$ and near-end speech $u(n)$ as shown in Figure 10.3. During the double-talk periods, the error signal $e(n)$ described in Equation (10.4) contains the residual echo, the uncorrelated noise $v(n)$, and the near-end speech $u(n)$. To correctly identify the characteristics of $P(z)$, $d(n)$ must originate solely from its input signal $x(n)$.

In theory, the far-end signal $x(n)$ is uncorrelated with the near-end speech $u(n)$, and thus will not affect the asymptotic mean value of the adaptive filter coefficients. However, the variation in the filter coefficients about this mean will be increased substantially in the presence of the near-end speech. Thus, the echo cancelation performance is degraded. An unprotected algorithm may exhibit unacceptable behavior during double-talk periods.

An effective solution is to detect the occurrence of double talk and then to disable the adaptation of $W(z)$ during the double-talk periods. Note that only the coefficient adaptation as illustrated in Figure 10.12 is disabled. If the echo path does not change during the double-talk periods, the echo can be canceled by the previously converged $W(z)$, whose coefficients are fixed during double-talk periods.

As shown in Figure 10.12, the speech detection/control block is used to control the adaptation of the adaptive filter $W(z)$ and the nonlinear processor (NLP) that is used for reducing residual echo. The

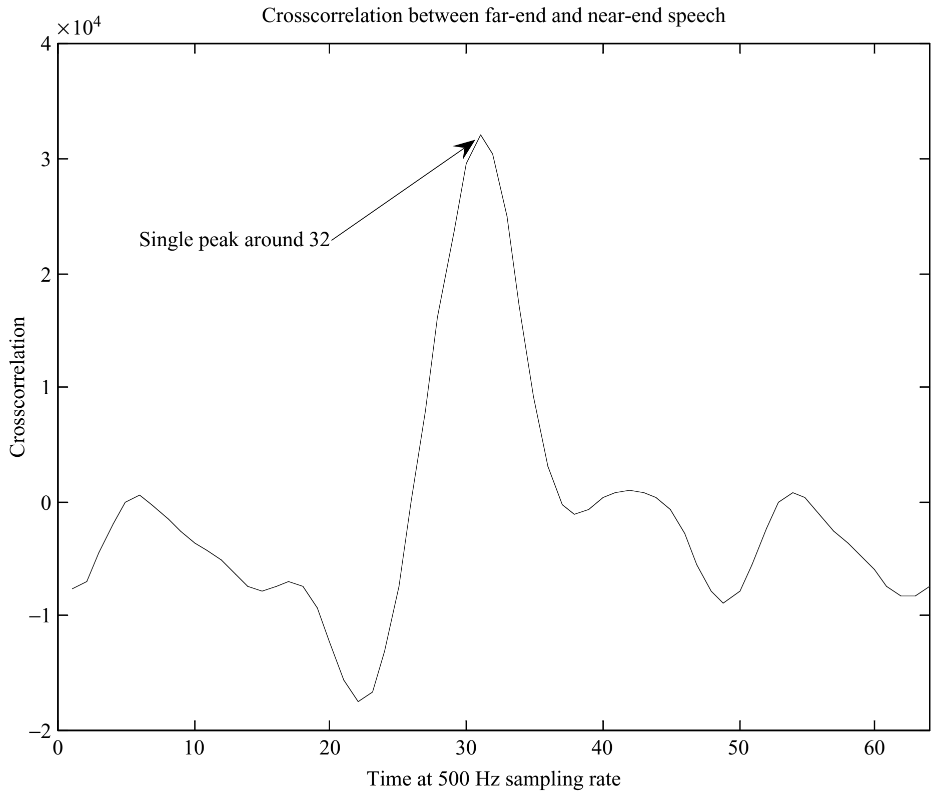


Figure 10.10 Improved resolution of the crosscorrelation peaks

double-talk detector (DTD), which detects the presence of near-end speech when the far-end speech is present, is a very critical element in echo cancelers.

The conventional DTD based on the echo return loss (ERL), or hybrid loss, can be expressed as

$$\rho = 20\log_{10}\left\{\frac{E\left[|x(n)|\right]}{E\left[|d(n)|\right]}\right\}. \tag{10.10}$$

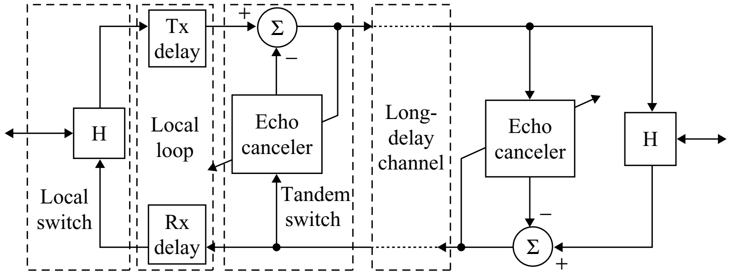


Figure 10.11 Configuration of echo cancelation with flat delay

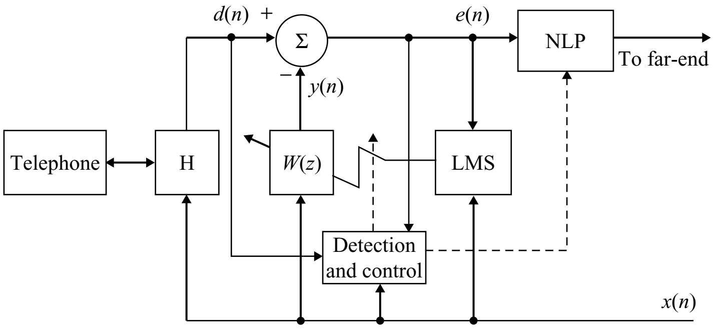


Figure 10.12 Adaptive echo canceler with speech detectors and nonlinear processor

In several adaptive echo cancelers such as defined by ITU standards, the ERL value is assumed to be 6 dB. Based on this assumption, the near-end speech is present if

$$|d(n)| > \frac{1}{2}\,|x(n)|\,. \tag{10.11}$$

However, we cannot just use the instantaneous absolute values $|d(n)|$ and $|x(n)|$ under the noisy condition. A modified near-end speech detection algorithm declares the presence of near-end speech if

$$|d(n)| > \frac{1}{2}\max\left\{|x(n)|\,,\ldots,|x(n-L+1)|\right\}. \tag{10.12}$$

Equation (10.12) compares an instantaneous absolute value $|d(n)|$ with the maximum absolute value of $x(n)$ over a time window spanning the echo path. The advantage of using an instantaneous power of $d(n)$ is its fast response to the near-end speech. However, it will increase the probability of false trigger if noise exists in the network.

A more robust version of speech detector replaces the instantaneous power $|x(n)|$ and $|d(n)|$ with the short-term power estimates $P_x(n)$ and $P_d(n)$. These short-term power estimates are implemented by the first-order IIR filter as

$$P_x(n) = (1-\alpha)P_x(n-1) + \alpha\,|x(n)| \tag{10.13}$$

and

$$P_d(n) = (1-\alpha)P_d(n-1) + \alpha\,|d(n)|\,, \tag{10.14}$$

where $0 < \alpha << 1$. The use of a larger α results in robust detector. However, it also causes slower response to the presence of near-end speech. With the modified short-term power estimate, the near-end speech is detected if

$$P_d(n) > \frac{1}{2}\max\left\{P_x(n),\,P_x(n-1),\ldots,P_x(n-L+1)\right\}. \tag{10.15}$$

It is important to note that a portion of the initial break-in near-end speech $u(n)$ may not be detected by this detector. Thus, adaptation would proceed in the presence of double talk. Furthermore, the requirement of maintaining a buffer to store L-power estimates increases the memory requirement and the complexity of algorithm.

The assumption that the ERL is a constant (6 dB) is not always correct. If the ERL is higher than 6 dB, it will take longer time to detect the presence of near-end speech. If the ERL is lower than 6 dB, most far-end speech will be falsely detected as near-end speech. For practical applications, it is better to dynamically estimate the time-varying threshold ρ by observing the signal level of $x(n)$ and $d(n)$ when the near-end speech $u(n)$ is absent.

10.5 Nonlinear Processor

The residual echo can be further reduced using an NLP realized as a center clipper. The comfort noise is inserted to minimize the adverse effects of the NLP.

10.5.1 Center Clipper

Nonlinearities in the echo path, noise in the circuits, and uncorrelated near-end speech limit the amount of achievable cancelation for a typical adaptive echo canceler. The NLP shown in Figure 10.12 removes the last vestiges of the remaining echoes. The most widely used NLP is a center clipper with the input–output characteristic illustrated by Figure 10.13. This nonlinear operation can be expressed as

$$y(n) = \begin{cases} 0, & |x(n)| \le \beta \\ x(n), & |x(n)| > \beta \end{cases}, \tag{10.16}$$

where β is the clipping threshold. This center clipper completely eliminates signals below the clipping threshold β, but leaves signals greater than the clipping threshold unaffected. A large value of β suppresses all the residual echoes but also deteriorates the quality of the near-end speech. Usually the threshold is chosen to be equal or to exceed the peak amplitude of return echo.

10.5.2 Comfort Noise

The NLP completely eliminates the residual echo and circuit noise, thus making the connection not 'real'. For example, if the near-end subscriber stops talking, the noise level will suddenly drop to zero since it has been clipped by the NLP. If the difference is significant, the far-end subscriber may think the call has

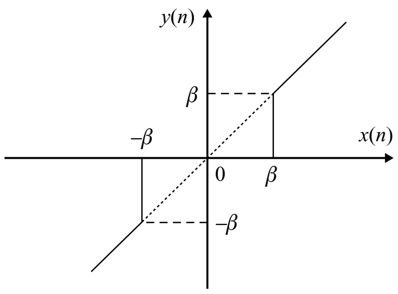


Figure 10.13 Input–output relationship of center clipper

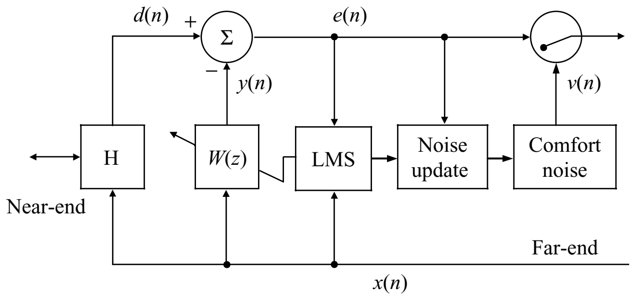


Figure 10.14 Implementation of G.168 with comfort noise insertion

been disconnected. Therefore, the complete suppression of a signal using NLP has an undesired effect. This problem can be solved by injecting a low-level comfort noise when the residual echo is suppressed.

As specified by Test 9 of G.168, the comfort noise must match the signal level and frequency contents of background noise. In order to match the spectrum, the comfort noise insertion is implemented in frequency domain by capturing the frequency characteristic of background noise. An alternate approach uses the linear predictive coding (LPC) coefficients to model the spectral information. In this case, the comfort noise is synthesized using a pth-order LPC all-pole filter, where the order p is between 6 and 10. The LPC coefficients are computed during the silence segments. The ITU-T G.168 recommends the level of comfort noise within ± 2 dB of the near-end noise.

An effective way of implementing NLP with comfort noise is shown in Figure 10.14, where the generated comfort noise $v(n)$ or echo canceler output $e(n)$ is selected as the output according to the control logic. The background noise is generated with a matched level and spectrum, heard by the far-end subscriber remaining constant during the call connection, and thus significantly contributing to the high-grade perceptive speech quality.

10.6 Acoustic Echo Cancelation

There has been a growing interest in applying acoustic echo cancelation for hands-free cellular phones in mobile environments and speakerphones in teleconferencing. Acoustic echoes consist of three major components: (1) acoustic energy coupling between the loudspeaker and the microphone; (2) multiple-path sound reflections of far-end speech; and (3) the sound reflections of the near-end speech signal. In this section, we focus on the cancelation of the first two echo components.

10.6.1 Acoustic Echoes

Speakerphone has become important office equipment because it provides the convenience of hands-free conversation. For reference purposes, the person using the speakerphone is the near-end talker and the person at the other end is the far-end talker. In Figure 10.15, the far-end speech is broadcasted through one or more loudspeakers inside the room. Unfortunately, the far-end speech played by the loudspeaker is also picked up by the microphone inside the room, and this acoustic echo is returned to the far end.

The basic concept of acoustic echo cancelation is similar to the line echo cancelation; however, the adaptive filter of acoustic echo canceler models the loudspeaker-room-microphone system instead of the hybrid. Thus, the acoustic echo canceler needs to cancel a long echo tail using a much high-order

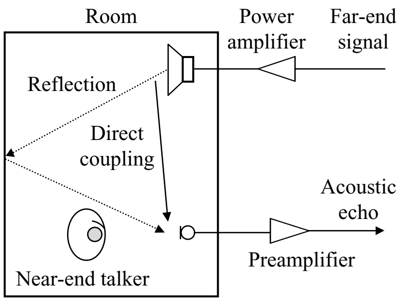


Figure 10.15 Acoustic echo generated by a speakerphone in a room

adaptive filter. One effective technique is the subband acoustic echo canceler, which splits the full-band signal into several overlapped subbands and uses an individual low-order filter for each subband.

Example 10.3: To evaluate an acoustic echo path, the impulse responses of a rectangular room ($246 \times 143 \times 111\,\text{in}^3$) were measured. The original data is sampled at 48 kHz, which is then bandlimited to 400 Hz and decimated to 1 kHz for display purpose. The room impulse response is stored in the file `imp.dat` and is shown in Figure 10.16 (`example10_3.m`):

```
load imp.dat;      % Room impulse response
plot(imp(1:1000)); % Display samples from 1 to 1000
```

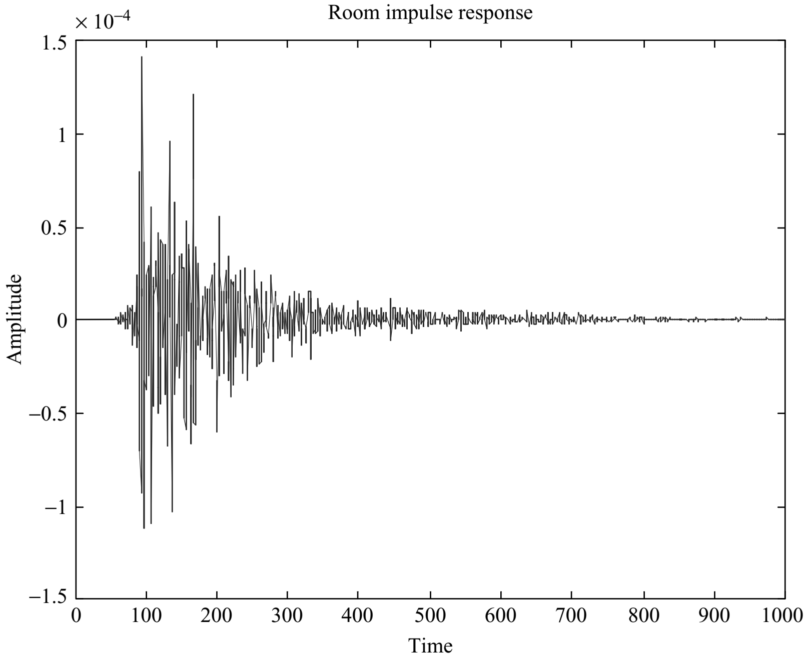


Figure 10.16 An example of room impulse response

There are three major factors making the acoustic echo cancelation far more challenging than the line echo cancelation for real-world applications:

1. The reverberation of a room causes a very long acoustic echo tail. The duration of the acoustic echo path is usually 400 ms in a typical conference room. For example, 3200 taps are needed to cancel 400 ms of echo at sampling rate 8 kHz.
2. The acoustic echo path may change rapidly due to the motion of people in the room, the change in position of the microphone, and some other factors like doors and/or windows opened or closed, etc. The acoustic echo canceler requires a faster convergence algorithm to track these fast changes.
3. The double-talk detection is much more difficult since we cannot assume the 6-dB acoustic loss as the hybrid loss in line echo canceler.

Therefore, acoustic echo cancelers require more computation power, faster convergence speed, and more sophisticated double-talk detector.

10.6.2 Acoustic Echo Canceler

The block diagram of an acoustic echo canceler is illustrated in Figure 10.17. The acoustic echo path $P(z)$ includes the transfer functions of the A/D and D/A converters, smoothing and antialiasing lowpass filters, speaker power amplifier, loudspeaker, microphone, microphone preamplifier, and the room transfer function from the loudspeaker to the microphone. The adaptive filter $W(z)$ models the acoustic echo path $P(z)$ and yields an echo replica $y(n)$ to cancel acoustic echo components in $d(n)$.

The adaptive filter $W(z)$ generates a replica of the echo as

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l). \tag{10.17}$$

This replica is then subtracted from the microphone signal $d(n)$ to generate $e(n)$. The coefficients of the $W(z)$ filter are updated by the normalized LMS algorithm as

$$w_l(n+1) = w_l(n) + \mu(n)e(n)x(n-l), \qquad l = 0, 1, \ldots, L-1, \tag{10.18}$$

where $\mu(n)$ is the normalized step size by the power estimation of $x(n)$.

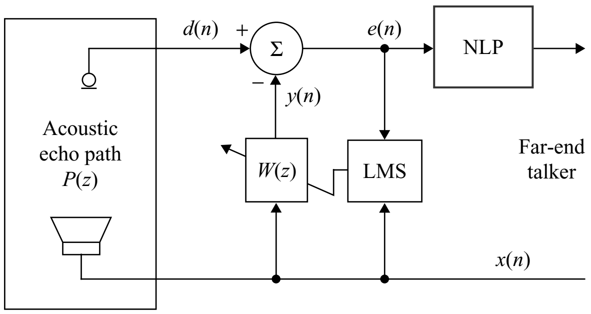


Figure 10.17 Block diagram of an acoustic echo canceler

10.6.3 Subband Implementations

Subband and frequency-domain adaptive filtering techniques have been developed to cancel long acoustic echoes. The advantages of using subband acoustic echo cancelers are (1) the decimation of subband signals reduces computational requirements, and (2) the signal whitening using normalized step size at each subband results in fast convergence.

A typical structure of subband echo canceler is shown in Figure 10.18, where $A_m(z)$ and $S_m(z)$ are analysis and synthesis filters, respectively. The number of subbands is M, and the decimation factor can be a number equal to or less than M. There are M adaptive FIR filters$W_m(z)$, one for each channel. Usually, these filter coefficients are in complex form with much lower order than the full-band adaptive filter $W(z)$ shown in Figure 10.17.

The filterbank design with complex coefficients reduces the filter length due to the relaxed antialiasing requirement. The drawback is the increased computation load because one complex multiplication requires four real multiplications. However, complex filterbank is still commonly used because of the difficulties to design a real coefficient bandpass filter with sharp cutoff and strict antialiasing requirements for adaptive echo cancelation.

An example of designing a 16-band filterbank with complex coefficients is highlighted as follows:

1. Using the MATLAB to design a prototype lowpass FIR filter with coefficients $h(n)$, $n = 0, 1, \ldots, N-1$, which meets the requirement of the 3-dB bandwidth at $\pi/2M$, where $M = 16$. The magnitude response of the prototype filter is shown in Figure 10.19(a), and the impulse response is given in Figure 10.19(b).

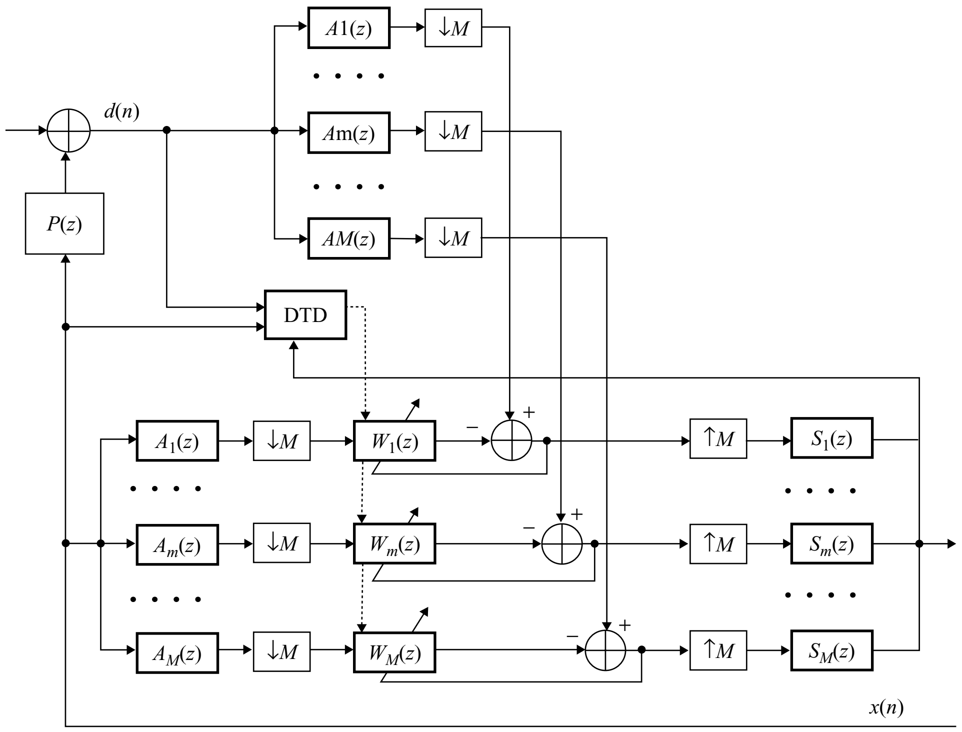


Figure 10.18 Block diagram of a subband echo canceler

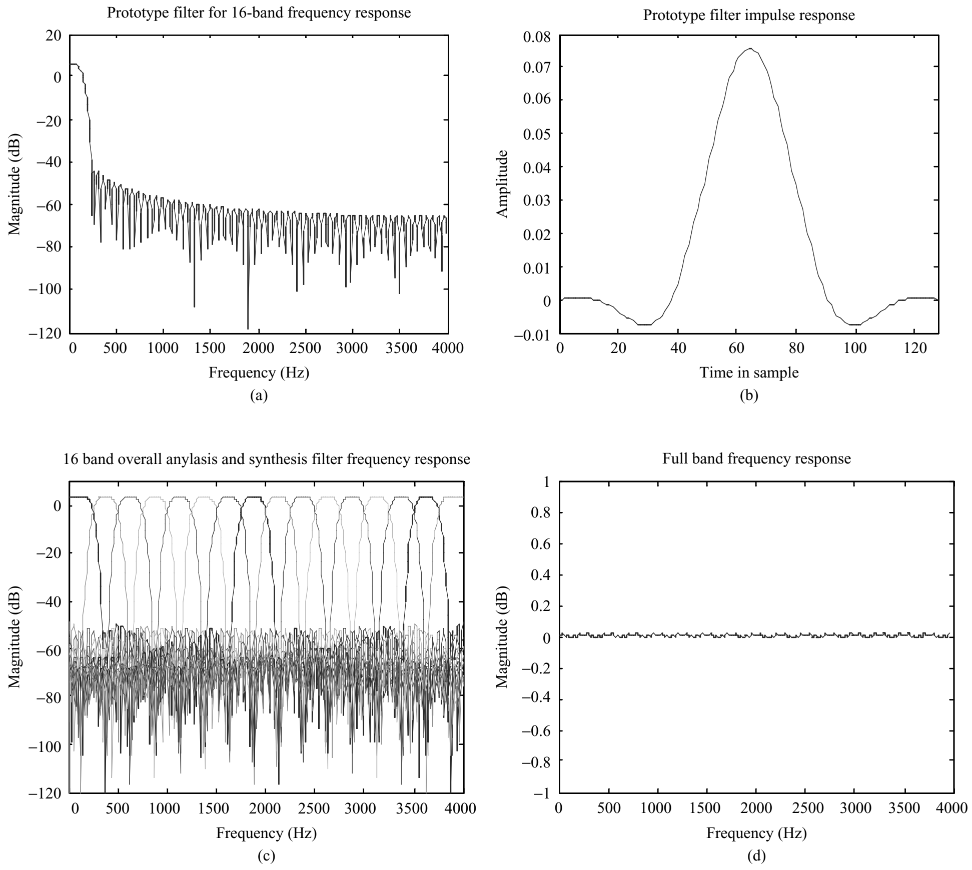


Figure 10.19 Example of filterbank with 16 complex subbands

2. Applying $\cos\left[\pi \frac{m-1/2}{M}\left(n - \frac{N+1}{2}\right)\right]$ and $\sin\left[\pi \frac{m-1/2}{M}\left(n - \frac{N+1}{2}\right)\right]$ to modulate the prototype filter to produce the complex-coefficient bandpass filters, $A_m(z)$, $m = 0, 1, \ldots, M - 1$, as shown in Figure 10.18. The overall filter's magnitude response is shown in Figure 10.19(c). In this example, the synthesis filterbank is identical to the analysis filterbank; i.e., $S_m(z) = A_m(z)$ for $m = 0, 1, \ldots, M - 1$.

3. Decimating filterbank outputs by M to produce the low-rate signals $s_m(n)$, $m = 0, 1, \ldots, M - 1$, for the far-end and $d_m(n)$ for the near-end.

4. Performing the adaptation and echo cancelation for each individual subband with $1/M$ sampling rate. This produces error signals $e_m(n)$, $m = 0, 1, \ldots, M - 1$.

5. The error signals at these M bands are synthesized back to the full-band signal using the bandpass filters $S_m(z)$. Figure 10.19(d) shows the filterbank performance.

Example 10.4: For the same tail length, compare the computational load between the adaptive echo cancelers of two subbands (assume real coefficients) and full band. More specifically, given that the tail length is 32 ms (256 samples at 8 kHz sampling rate), estimate the required number of multiply–add operations.

Subband implementation requires 2×128 multiplications and additions for updating coefficients at half of the sampling rate. In comparison, the full-band adaptive filter needs 256 multiplications and additions at sampling rate. This means subband implementation needs only half of the computations required for a full-band implementation. In this comparison, the computation load of splitting filter is not counted since this computation load is very small as compared to the coefficients update using the adaptive algorithm.

10.6.4 Delay-Free Structures

The inherent disadvantage of subband implementations is the extra delay introduced by the filterbank, which splits the full-band signal into multiple subbands and also synthesizes the processed subband signals into a full-band signal. Figure 10.20 shows the algorithm delay of subband adaptive echo canceler.

A delay-free subband acoustic echo canceler can be implemented by adding an additional short full-band adaptive FIR filters $W_0(z)$, which covers the first part of the echo path and its length is equal to the total delay introduced by the analysis/synthesis filters plus the block-processing size. The subband adaptive filters model the rest of the echo path. Figure 10.21 illustrates the structure of delay-free subband acoustic echo cancelation.

Example 10.5: For a 16-band subband acoustic echo canceler with delay-free structure, calculate the minimum filter length of the first FIR filter. Given that the filterbank is a linear phase FIR filter with 128 taps.

The filterbank (analysis and synthesis) delay is 128 samples and the processing block delay is 16 samples. Therefore, the total delay due to filterbank is $128 + 16 = 144$ samples. In this case, the length of the first FIR filter $W_0(z)$ is at least 144.

10.6.5 Implementation Considerations

As shown in Figure 10.8, an effective technique to reduce filter length is to introduce a delay buffer of Δ samples at the input of adaptive filter. This buffer compensates for delay in the echo path caused by the propagation delay from the loudspeaker to the microphone. This technique saves computation since it effectively covers Δ impulse response samples without using adaptive filter coefficients. For example,

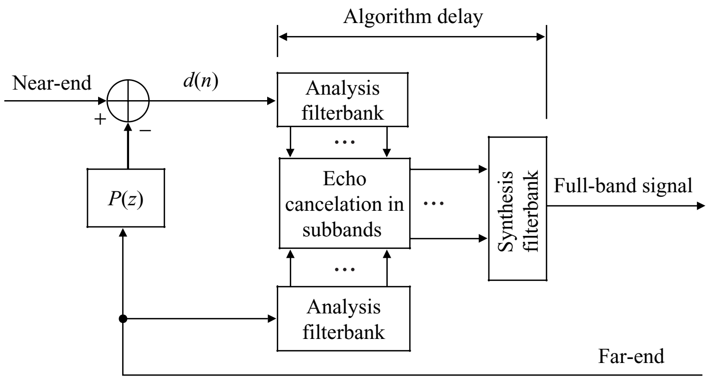


Figure 10.20 Illustration of algorithm delay due to filterbank

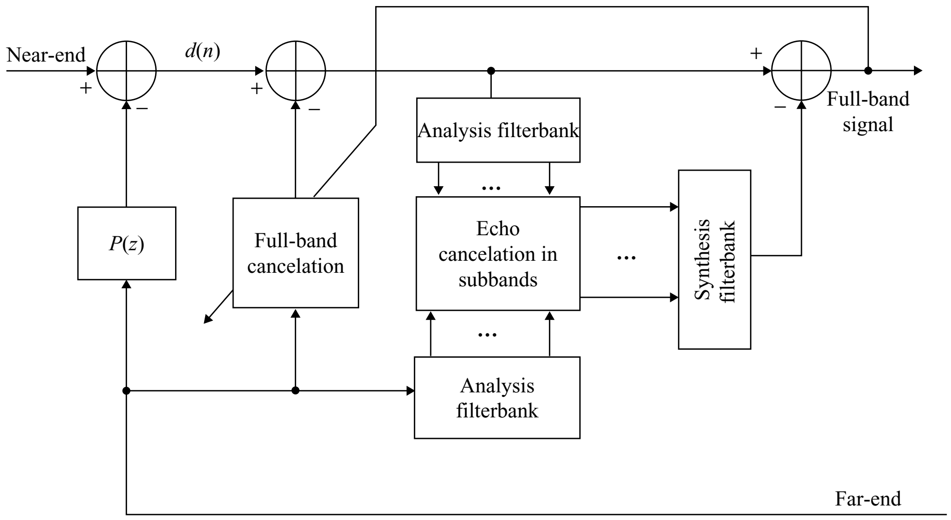


Figure 10.21 Structure of delay-free subband acoustic echo canceler

if the distance between the loudspeaker and the microphone is 1.5 m, the measured time delay in the system is about 4.526 ms based on the sound speed traveling at 331.4 m/s, which corresponds to $\Delta = 36$ at 8 kHz sampling rate.

As discussed in Chapter 7, if a fixed-point DSP processor is used for implementation and μ is sufficiently small, the excess MSE increases with a larger L, and the numerical errors (due to coefficient quantization and roundoff) increase with a larger L, too. Furthermore, roundoff error causes early termination of the adaptation if a small μ is used. In order to alleviate these problems, a larger dynamic range is required which can be achieved by using floating-point arithmetic. However, floating-point solution requires a more expensive hardware for implementation.

As mentioned earlier, the adaptation of coefficients must be temporarily stopped when the near-end talker is speaking. Most double-talk detectors for adaptive line echo cancelers are based on ERL. For acoustic echo cancelers, the echo return (or acoustic) loss is very small or may be even a gain because of the use of amplifiers in the system. Therefore, the higher level of acoustic echo makes detection of weak near-end speech very difficult.

10.6.6 Testing Standards

ITU G.167 specifies the procedure for evaluating the performance of an acoustic echo canceler. As shown in Figure 10.22, the echo canceler is tested with the input far-end signal R_{in} and near-end signal S_{in}, and the output near-end signal R_{out} and far-end signal S_{out}. The performance of echo cancelation is evaluated based on these signals. Some G.167 requirements are listed as follows:

- *Initial convergence time*: For all the applications, the attenuation of the echo shall be at least 20 dB after 1 s. This test evaluates the convergence time of adaptive filter. The filter structure, adaptation algorithm, step size μ, type of input signal, and prewhitening technique may affect this test.

- *Weighted terminal coupling loss during single talk*: For teleconferencing systems and hands-free communication, its value shall be at least 40 dB on both sides. The value is the difference between

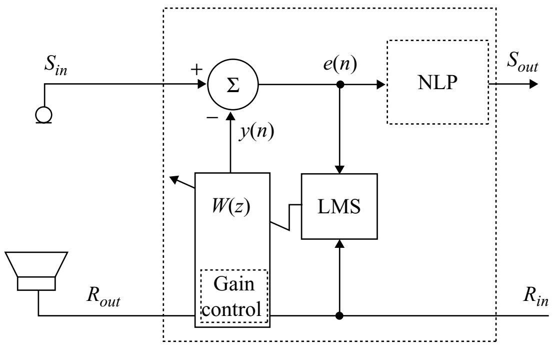


Figure 10.22 Simplified diagram for G.167 testing

the signal level (in S_{out}) without echo cancelation and the signal level with echo canceler in steady state. Test signal is applied to R_{in} and no other speech signal other than the acoustic return from the loudspeaker(s) is applied to the microphone.

- *Weighted terminal coupling loss during double talk*: For teleconferencing systems and hands-free communication, its value shall be at least 25 dB on both sides. After the echo canceler reaches the steady state, a near-end speech is applied at the S_{in} for 2 s. The adaptive filter coefficients are frozen and then the near-end speech is removed. This test evaluates how fast and accurate is the DTD to stop the coefficient update during the double talk.

- *Recovery time after echo path variation*: For all the applications, the attenuation of the echo should be at least 20 dB after 1 s. This test evaluates the echo canceler after the double talk; the coefficients may be affected but the system should not take more than 1 s to update to the optimum level.

One of the interesting observations for the tests specified by ITU-T G. 167 is that the test vectors are artificial voices according to ITU P.50 standard. These artificial voices are composed using the speech synthesis model. This makes the test easier with reduced limitation of human resources.

10.7 Experiments and Program Examples

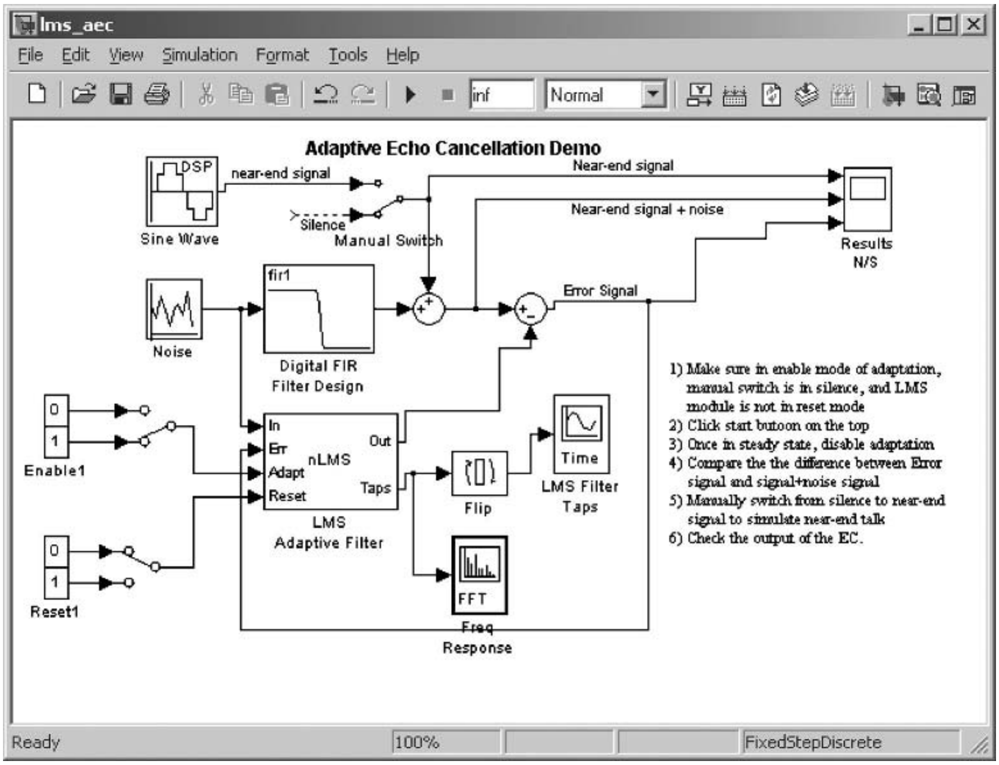
This section presents some echo cancelation modules using MATLAB, C, or C55x programs to further illustrate the algorithms and examine the performance.

10.7.1 MATLAB Implementation of AEC

This experiment is modified from the `lms` demo available in the MATLAB *Signal Processing Blockset*. Procedures of the experiment are listed as follows:

1. Start MATLAB and change to directory `..\experiments\exp10.7.1_matAec`.

2. Run the experiment by typing `lms_aec` in the MATLAB command window. This starts Simulink and creates a window that contains the model as shown in Figure 10.23.



1) Make sure in enable mode of adaptation, manual switch is in silence, and LMS module is not in reset mode
2) Click start butoon on the top
3) Once in steady state, disable adaptation
4) Compare the the difference between Error signal and signal+noise signal
5) Manually switch from silence to near-end signal to simulate near-end talk
6) Check the output of the EC.

Figure 10.23 Acoustic echo cancelation demo using Simulink

3. In the `lms_aec` window shown in Figure 10.23, make sure that:
   (a) nLMS module is connected to the Enable1's position 1;
   (b) Manual switch is in silence position; and
   (c) nLMS module is connected to the Reset1's position 0.
   If any of the connections is not met, double click the connection to change it.

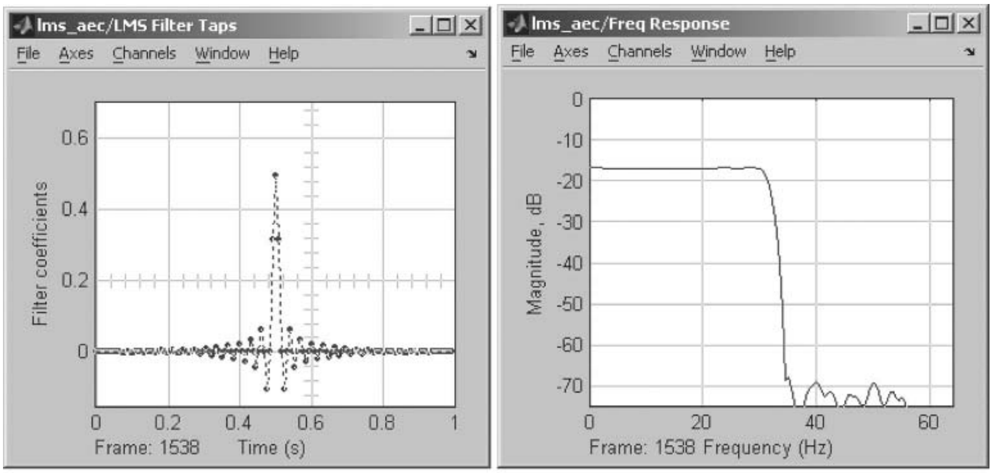
4. Open the **Simulation** pull-down menu and click **Start** to start Simulink.

5. After the algorithm reaches steady state, disable adaptation and freeze the coefficients by changing the 'Enable1' switch to position 0. The adaptive filter coefficients are shown in Figure 10.24(a). In this experiment, the echo path is simulated by a 128-tap FIR lowpass filter with normalized cutoff frequency of 0.5. The coefficients of echo canceler in steady state approximate the coefficients of lowpass filter. Figure 10.24(b) shows the magnitude response of the converged adaptive filter.

6. Figure 10.25(a) shows the near-end signal, error signal, and echo (signal + noise). The differences between the near-end signal and the error signal indicate the performance of echo cancelation.

7. Switch from silence to near-end signal to add the near-end signal (a sinewave).

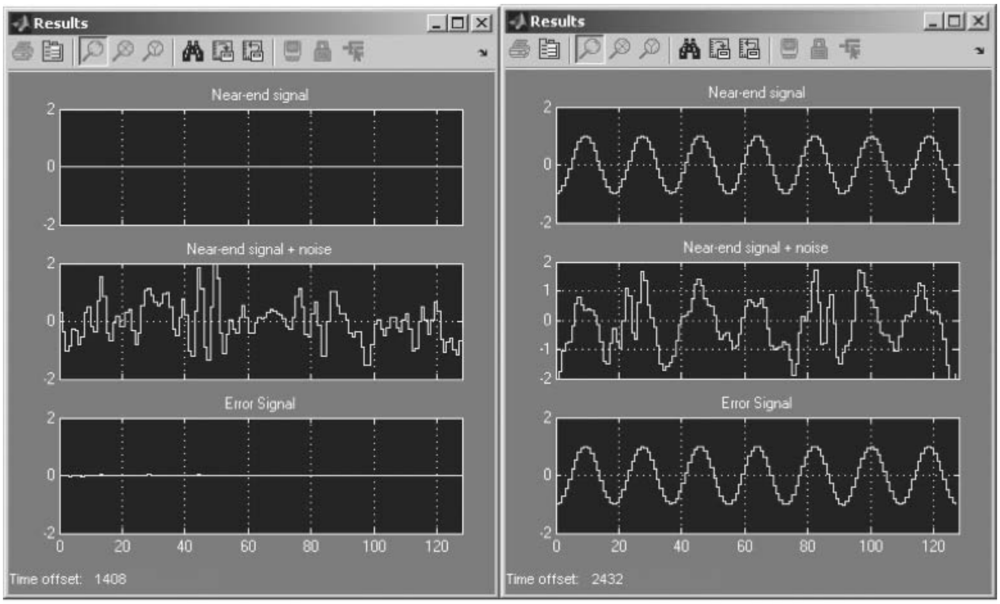
8. Figure 10.25(b) shows the output of the echo canceler, which is very close to the near-end signal.



(a) Impulse response (b) Frequency response

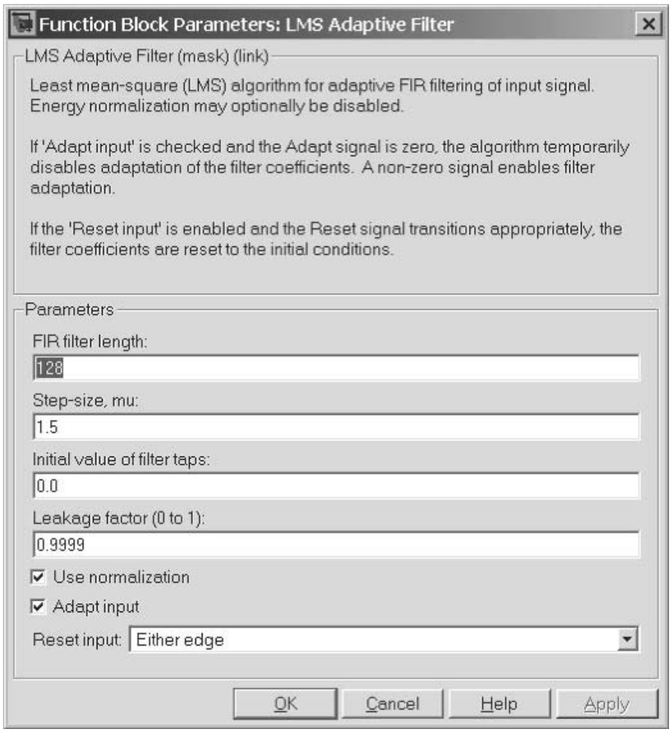
Figure 10.24 Adaptive filter in steady state: (a) impulse response; (b) frequency response

This experiment can be repeated with different parameters. For example, on double clicking the nLMS module shown in Figure 10.23, a function parameter configuration window will be displayed as shown in Figure 10.26. From this window, we can modify the step size, filter length, as well as leaky factor.



(a) Echo only (b) During double talk

Figure 10.25 Signal waveforms generated by Simulink model: (a) echo only; (b) during double talk



LMS Adaptive Filter (mask) (link)

Least mean-square (LMS) algorithm for adaptive FIR filtering of input signal. Energy normalization may optionally be disabled.

If 'Adapt input' is checked and the Adapt signal is zero, the algorithm temporarily disables adaptation of the filter coefficients. A non-zero signal enables filter adaptation.

If the 'Reset input' is enabled and the Reset signal transitions appropriately, the filter coefficients are reset to the initial conditions.

Parameters

FIR filter length: 128

Step-size, mu: 1.5

Initial value of filter taps: 0.0

Leakage factor (0 to 1): 0.9999

☑ Use normalization

☑ Adapt input

Reset input: Either edge

Figure 10.26 Configuration of the LMS adaptive filter

9. Try the following configurations and verify their performance. Explain why the echo cancelation performance becomes worse or better?
   - Change the echo path from an FIR filter to an IIR filter.
   - Change the switch from the silence to near-end signal position before disabling the adaptation with the LMS algorithm.
   - Change the step size μ from 1.5 to 0.1 and 4, and observe the results.

10. Select the FIR filter length of 256 and the LMS filter length of 64. Explain why the coefficients cannot match the echo path?

10.7.2 Acoustic Echo Cancelation Using Floating-Point C

An acoustic echo canceler implemented using floating-point C is presented in this experiment. The files used for this experiment are listed in Table 10.1. The data files used for experiment are captured using a PC sound card at 8-kHz sampling rate. The conversation is carried out in a room of size $11 \times 13 \times 9\,\text{ft}^3$. The far-end speech file `rtfar.pcm` and the near-end speech file `rtmic.pcm` are captured simultaneously. The near-end signal picked up by a microphone consists of the near-end speech and acoustic echoes generated from the far-end speech.

Table 10.1 File listing for experiment `exp10.7.2_floatingPointAec`

Files	Description
`AecTest.c`	Program for testing acoustic echo canceler
`AecInit.c`	Initialization function
`AecUtil.c`	Echo canceler utility functions
`AecCalc.c`	Main module for echo canceler
`Aec.h`	C header file
`floatPoint_aec.pjt`	DSP project file
`floatPoint_aec.cmd`	DSP linker command file
`rtfar.pcm`	Far-end data file
`rtmic.pcm`	Near-end data file

The adaptive echo canceler operates in four different modes based on the power of far-end and near-end signals. These four operating modes are defined as follows:

1. *Receive mode*: Only the far-end speaker is talking.
2. *Transmit mode*: Only the near-end speaker is talking.
3. *Idle mode*: Both ends are silence.
4. *Double-talk mode*: Both ends are talking.

Different operations are required for different modes. For example, the adaptive filter coefficients will be updated only at the receive mode. Typical operations at different modes are coded in Table 10.2.

Figure 10.27 illustrates the performance of acoustic echo canceler: (a) the far-end speech signal that is played via a loudspeaker in the room; (b) the near-end signal picked up by a microphone, which consists of the near-end speech as well as the echoes in the room generated from playing the far-end speech; and (c) the acoustic echo canceler output to be transmitted to the far-end. It clearly shows that the echo canceler output contains only the near-end speech. In this experiment, the double talk is not present. The echo canceler reduces the echo by more than 20 dB. More experiments can be conducted by using different parameters.

Procedures of the experiment are listed as follows:

1. Use an audio player or MATLAB to play the data files. The `rtfar.pcm` is transmitted from the far-end and played by a loudspeaker, which will generate acoustic echo in a room. The `rtmic.pcm` is the near-end signal captured by a microphone. We can clearly hear both the near-end speech and the echo generated from the far-end speech.
2. Open and build the experiment project.
3. Load and run the experiment using the provided data files. Verify the performance of acoustic echo canceler for removing the echo.
4. Open the C source file `AecInit.c`, adjust the following adaptive echo canceler parameters, and rerun the experiment to observe changing behavior:
   (a) echo tail length `aec->AECorder` (up to 1024);
   (b) leaky factor `aec->leaky` (1.0 disables leaky function); and
   (c) step size `aec->mu`.

Table 10.2 Partial C code for acoustic echo cancelation

```
 if (farFlag == 1)                                     // There is far-end speech
 {
   if ((nearFlag == 0) ||   (trainTime > 0))           // Receive mode
   {
     /* Receive mode operations */
     if (trainTime > 0)                                // Counter is no expire yet
     {
        trainTime--;                                   // Decrement the counter
        if (txGain > 0.25) txGain -= rampDown;         // Ramp down
        farEndOut = (float)(txGain*errorAEC);          // Attenuate by 12 dB
     }
     if (errorAECpowM<clipThres)                       // If ERLE > 18 dB
     {                                                 // Enable center clipper
         farEndOut = comfortNoise;                     // and inject comfort noise
     }
     else                                              // If ERLE < 18 dB
     {
         if (txGain > 0.25)txGain -= rampDown;         // Ramp down
         farEndOut = (float)(txGain*errorAEC);         // Disable center clipper
     }                                                 // Attenuated by 12 dB
     if (farInPowM < 16000.)       // Signal farEndIn is reasonable
     {
       /* Update AEC coefficients, otherwise skip adaptation*/
       temp = (float)((mu*errorAEC)                    /(spkOutPowM+saveMargin));
                                                       // Normalize step size
       for (k=0; k<AECorder; ++k)
       {
         /* Leaky normalized LMS update */
         AECcoef[k] = (float)(leaky*AECcoef[k] + temp*AECbuf[k]);
       }
     }
   }
   else                                                // Double talk mode
   {
     /* Double-talk mode operation */
     if (txGain > 0.5) txGain -= rampDown;             // Ramp down
     if (txGain < 0.5) txGain += rampUp;               // Ramp up
     farEndOut = (float)(txGain*errorAEC);             // Attenuate 6 dB
   }
 }
 else                                                  // No far-end speech
 {                                                     // Transmit mode operation
    if (nearFlag == 1)
    {
      if (txGain < 1) txGain += rampUp;
      farEndOut = txGain*microphoneIn;                // Full gain at trans-
mit path
    }
    else                                              // Idle mode operation
    {
      if (txGain > 0.5) txGain -= rampDown;           // Ramp down
      if (txGain < 0.5) txGain += rampUp;             // Ramp up
      farEndOut = (float)(txGain*microphoneIn);       // Attenuate 6 dB
    }
 }
```

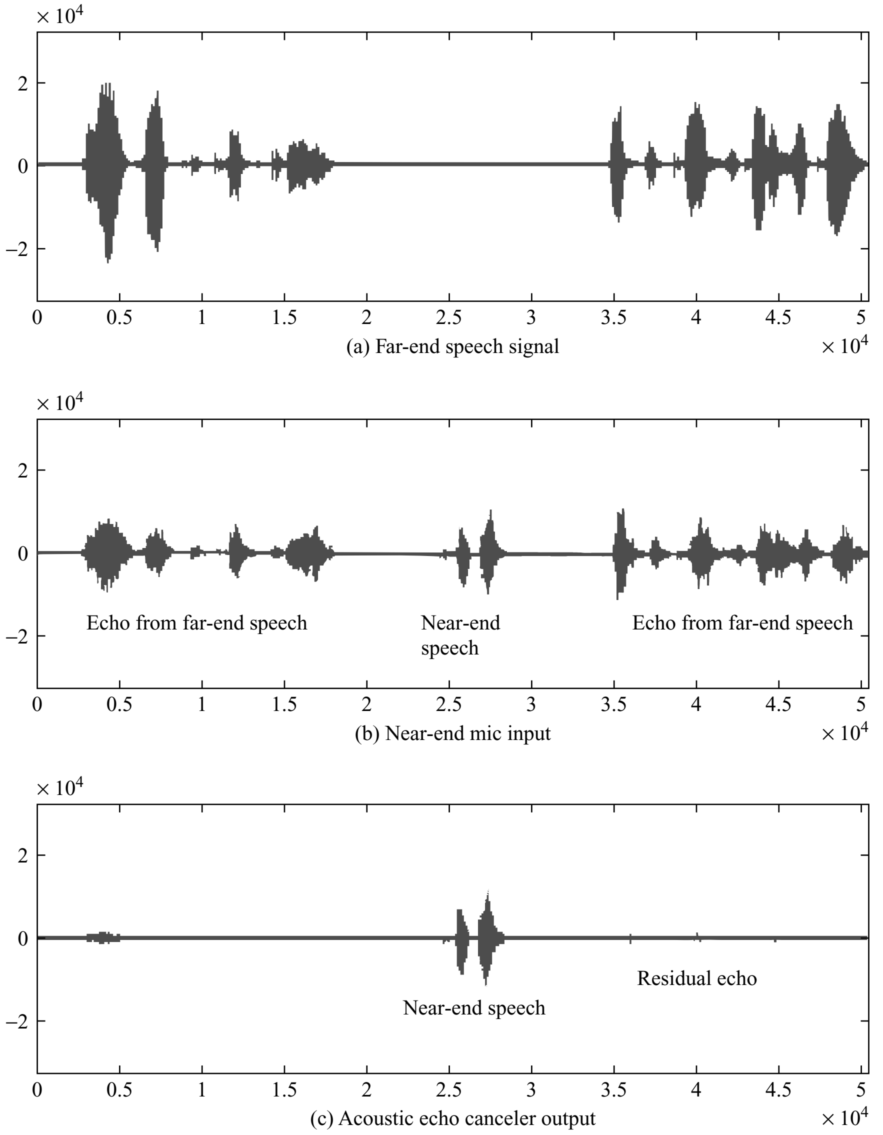


Figure 10.27 Experiment results of acoustic echo cancelation: (a) far-end speech signal; (b) near-end mic input; and (c) acoustic echo canceler output

5. Using the knowledge learned from previous experiments, write an assembly program to replace the adaptive filtering function used by this experiment.

6. Convert the rest of the experiment to fixed-point C implementation. Pay special attention on data type conversion and fixed-point implementation for C55x processors.

Table 10.3 File listing for experiment `exp10.7.3_intrinsicAec`

Files	Description
`intrinsic_aec.pjt`	C55x project file
`intrinsic_aec.cmd`	C55x linker command file
`fixPoint_leaky_lmsTest.c`	Main program
`fixPoint_aec_init.c`	Initialization function
`fixPoint_double_talk.c`	Double-talk detection function
`fixPoint_leaky_lms.c`	Major module for LMS update and filtering
`fixPoint_nlp.c`	NLP function
`utility.c`	Utility function of long division
`fixPoint_leaky_lms.h`	Header file
`gsm.h`	Header file for using intrinsics
`linkage.h`	Header file needed for intrinsics
`rtfar.pcm`	Data file of far-end signal
`rtmic.pcm`	Data file of near-end signal

10.7.3 Acoustic Echo Canceler Using C55x Intrinsics

This experiment shows the implementation of a fixed-point acoustic echo canceler. We use the normalized LMS algorithm presented in Chapter 7. In addition, we add an NLP function to further attenuate the residue echoes. The files used for this experiment are listed in Table 10.3.

Fixed-point C implementation of leaky NLMS algorithm using intrinsic functions has been discussed in Section 7.6.3. Using the same technique, the DTD can be implemented in fixed-point C using the C55x intrinsics. Table 10.4 lists portion of the C program for far-end speech detection. In the program, the function `aec_power_estimate( )` is used to estimate the signal power. The variable `dt->nfFar` is the noise floor of the far-end signal. If the signal power `dt->nfFar` is higher than the noise floor, the speech is detected.

Table 10.4 Partial fixed-point C code for far-end signal detection

```
// Update noise floor estimate of receiving far-end signal
// temp = |farEndIn|, estimate far-end signal power
   temp32a = L_deposit_h(lms->in);
   dt->farInPowS = aec_power_estimate(
   dt->farInPowS,temp32a,ALPHA_SHIFT_SHORT);
   if (dt->nfFar < dt->farInPowS) {
      // Onset of speech, slow update using long window
      dt->nfFar = aec_power_estimate(
   dt->nfFar,temp32a,ALPHA_SHIFT_MEDIUM);
   }
   else {
      dt->nfFar = aec_power_estimate(
   dt->nfFar,temp32a,ALPHA_SHIFT_SHORT);
   }
   // Threshold for far-end speech detector
   temp32b = L_mult(extract_h(dt->nfFar),VAD_POWER_THRESH);
   temp32b = L_add(temp32b,dt->nfFar);
   temp32b = L_add(temp32b,L_deposit_h(SAFE_MARGIN));
```

Table 10.4 (*continued*)

```
    if(temp32b <= L_deposit_h(200))
       temp32b = L_deposit_h(200);
    // Detect speech activity at far end
    if(dt->farInPowS > temp32b)         // temp32b = thresFar
    {   // Declare far-end speech
       dt->farFlag = 1;
       // Set hangover time counter
       dt->farHangCount = HANGOVER_TIME;
    }
    else
    {
       if (dt->farHangCount-- < 0)  // Decrement hangover counter
       {
          dt->farFlag = 0;
          dt->farHangCount = 0;     // Hangover counter expired
       }
    }
```

Procedures of the experiment are listed as follows:

1. Build, load, and run the experiment program.
2. The acoustic echo canceler output is saved in the file named `aecout.pcm`.
3. Use the CCS graph tool to plot the adaptive filter coefficients, `w`, of length 512, as shown in Figure 10.28.
4. With the same inputs as shown in Figure 10.27(a) and (b), the processed output by this fixed-point acoustic echo canceler is shown in Figure 10.29.

Further experiments include writing assembly programs to replace the intrinsics used in this experiment, and modifying the fixed-point C code to create an adaptive echo canceler using assembly program.

10.7.4 Experiment of Delay Estimation

This experiment uses the MATLAB scripts `exp10_7_4.m` to find the echo delay based on the crosscorrelation method. The program is listed in Table 10.5. The MATLAB function `xcorr(x,y, 'biased')` is used to calculate the crosscorrelation between the vectors `x` and `y`. In this experiment, we use `auco8khz.txt` as the far-end data (`y` vector), delay it by 200 samples, and copy it as the near-end data (`x` vector). The crosscorrelation between the vectors `x` and *`y`* is returned to `crossxy`. The MATLAB function `max( )` is used to find the maximum value *`m`* in the array, which represents the delay between the far-end and near-end signals.

The files used for this experiment are listed in Table 10.6, and procedures of the experiment are listed as follows:

1. Running the script, the delay value is estimated and printed as `The maximum corssXY(m)`. This simple technique works well for estimating a pure delay in noise-free environment.
2. In real applications with noises and multiple echoes, more complicated methods discussed in Section 10.3.2 are needed. The crosscorrelation function is shown in Figure 10.30.

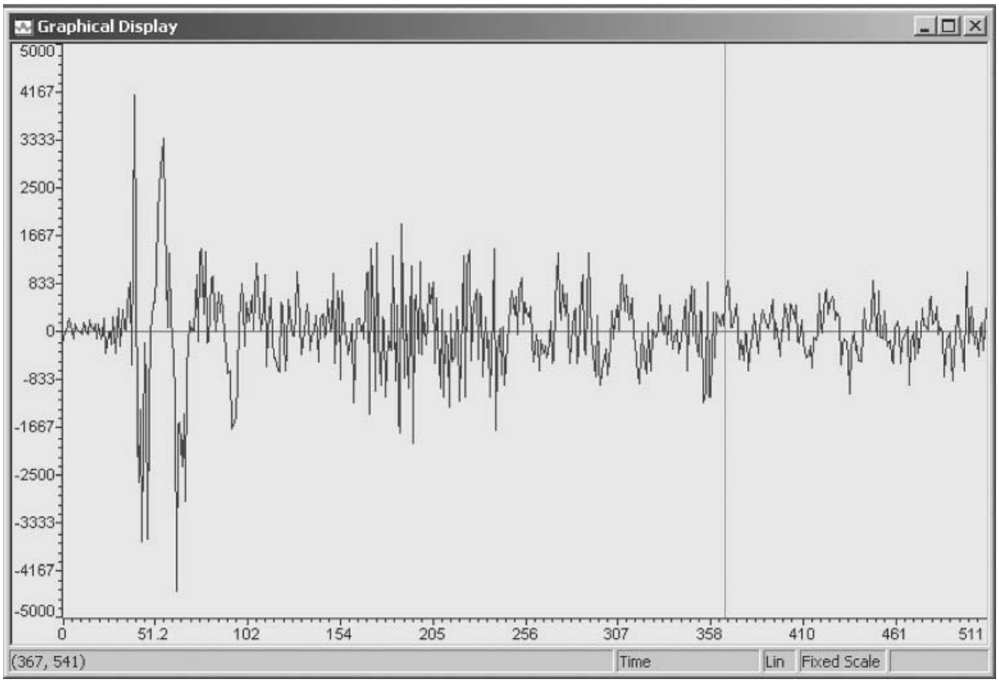


Figure 10.28 Adaptive filter coefficients in steady state

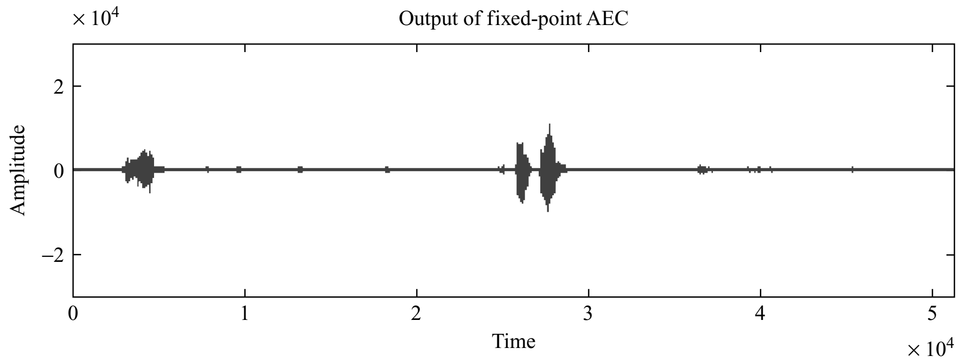


Figure 10.29 The error signal of fixed-point AEC output

Table 10.5 Crosscorrelation method for estimating the delay

```
% Open data files
fid1 = fopen('.//data//rtfar.pcm', 'rb');
fid2 = fopen('.//data//rtmic.pcm', 'rb');
% Read data files
x = fread(fid1, 'int16');
y = fread(fid2, 'int16');
% crossxy(m) = cxy(m-N), m=1, ..., 2N-1
crossxy = xcorr(x(1:800),y(1:800),'biased');
len=size(crossxy);
% Only half = cxy(m-N), m=1, ...
```

Table 10.5 (*continued*)

```
xy = abs(crossxy(((len-1)/2+1):len));
% Find max in xy
[ampxy,posxy]=max(xy);

plot(xy),;
title('Crosscorelation between x and y');
xlabel('Time at 8000 Hz sampling rate');
ylabel('Crosscorrelation');
text(posxy-1,ampxy,...
'\bullet\leftarrow\fontname{times} CorossXY(m) = MAXIMUM',
'FontSize',12)
disp(sprintf('The maximum corssXY(m) found at %d with value =%d \n',
posxy-1,ampxy));

fclose(fid1);
fclose(fid2);
```

Table 10.6 File listing for experiment `exp10.7.4_delayDetect`

Files	Description
`delayDetect.m`	MATLAB experiment program
`rtfar.pcm`	Data file for far-end signal
`rtmic.pcm`	Data file for near-end signal

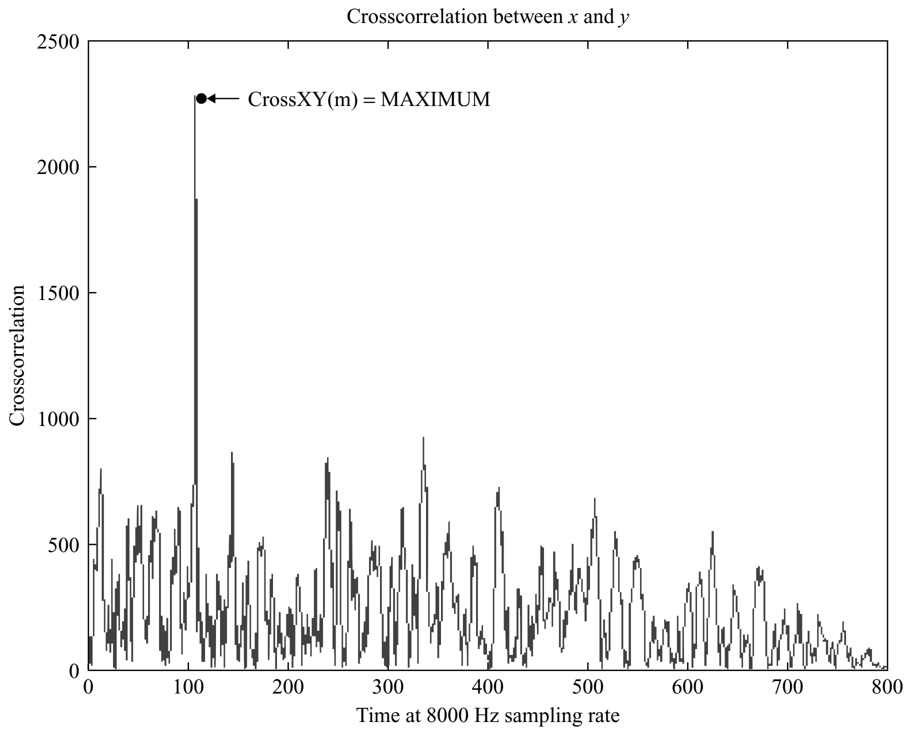


Figure 10.30 Crosscorrelation function to find a flat delay

References

[1] S. M. Kuo and D. R. Morgan, *Active Noise Control Systems – Algorithms and DSP Implementations*, New York: John Wiley & Sons, Inc., 1996.
[2] W. Tian and A. Alvarez, 'Echo canceller and method of canceling echo,' *World Intellectual Property Organization*, Patent WO 02/093774 A1, Nov. 2002.
[3] W. Tian and Y. Lu, 'System and method for comfort noise generation,' US Patent no. 6 766 020 B1, July 2004.
[4] Y. Lu, R. Fowler, W. Tian, and L. Thompson, 'Enhancing echo cancellation via estimation of delay,' *IEEE Trans. Signal. Process.*, vol. 53, no. 11, pp. 4159–4168, Nov. 2005.
[5] D. L. Duttweiler, 'A twelve-channel digital echo canceller,' *IEEE Trans. Comm.*, vol. COM-26, pp. 647–653, May 1978.
[6] D. L. Duttweiler and Y. S. Chen, 'A single-chip VLSI echo canceller,' *Bell Sys. Tech. J.*, vol. 59, pp. 149–160, Feb. 1980.
[7] K. Eneman and M. Moonen, 'Filterbank constrains for subband and frequency-domain adaptive filters,' *Proc. IEEE ASSP Workshop*, New Paltz, NY: Mohonk Mountain House, Oct. 1997.
[8] Math Works, Inc.,*Using MATLAB*, Version 6, 2000.
[9] Math Works, Inc.,*MATLAB Reference Guide*, 1992.
[10] Analog Devices, *Digital Signal Processing Applications Using the ADSP-2100 Family*, Englewood Cliffs, NJ: Prentice Hall, 1990.
[11] C. W. K. Gritton and D. W. Lin, 'Echo cancellation algorithms' *IEEE ASSP Mag.*, pp. 30–38, Apr. 1984.
[12] CCITT Recommendation G.165, *Echo Cancellers*, 1984.
[13] M. M. Sondhi and D. A. Berkley, 'Silencing echoes on the telephone network,' *Proc. IEEE*, vol. 68, pp. 948–963, Aug. 1980.
[14] M. M. Sondhi and W. Kellermann, 'Adaptive echo cancellation for speech signals,' in *Advances in Speech Signal Processing*, S. Furui and M. Sondhi, Eds., New York: Marcel Dekker, 1992, Chap. 11.
[15] Texas Instruments, Inc., *Acoustic Echo Cancellation Software for Hands-Free Wireless Systems*, Literature no. SPRA162, 1997
[16] Texas Instruments, Inc., *Echo Cancellation S/W for TMS320C54x*, Literature no. BPRA054, 1997
[17] Texas Instruments, Inc., *Implementing a Line-Echo Canceller Using Block Update & NLMS Algorithms- 'C54x*, Literature no. SPRA188, 1997
[18] ITU-T Recommendation G.167, *Acoustic Echo Controllers*, Mar. 1993.
[19] ITU-T Recommendation G.168, *Digital Network Echo Cancellers*, 2000.

Exercises

1. What are the first things to check if the adaptive filter is diverged during the designing of an adaptive echo canceler?

2. Assuming a full-band adaptive FIR filter is used and the sampling frequency is 8 kHz, calculate the following:

   (a) the number of taps needed to cover an echo tail of 128 ms;

   (b) the number of multiplications needed for coefficient adaptation; and

   (c) the number of taps if the sampling frequency is 16 kHz.

3. In Problem 2, the full-band signal is sampled at 8 kHz. If using 32 subbands and the subband signal is critically sampled, answer the following questions:

   (a) What is the sampling rate for each subband?

   (b) What is the minimum number of taps for each subband in order to cover the echo tail length of 128 ms?

   (c) What is the total number of taps and is this number the same as that of Problem 2?

   (d) At 8 kHz sampling rate, how many multiplications are needed for coefficient adaptation in each sampling period? You should see the savings in computations over Problem 2.

4. The C55x LMS instruction, `LMS Xmem, Ymem, ACx, ACy`, is very efficient to perform two parallel LMS operations in one cycle. Write a C55x code using this instruction to convert the fixed-point C code in the experiment given in Section 10.7.3.

5. For VoIP applications, if a conventional landline telephone user A calls an IP phone user B via the VoIP gateway, draw a diagram to show which side will hear line echo and which side needs a line echo canceler. If both sides are using IP phones, do you think we still need a line echo canceler?

6. In the experiment given in Section 10.7.2, the signal flow has been classified into transmit, receive, double talk, and idle mode. In each of these modes, summarize which processes, comfort noise insertion, ramping up, ramping down, or attenuation, are applied?

11 Speech-Coding Techniques

Communication infrastructures and services have been changed dramatically in recent years to include data and images. However, speech is still the most important and common service in the telecommunication networks. This chapter introduces speech-coding techniques to achieve the spectral efficiency, security, and easy storage.

11.1 Introduction to Speech-Coding

Speech-coding techniques compress the speech signals to achieve the efficiency in storage and transmission, and to decompress the digital codes to reconstruct the speech signals with satisfactory qualities. In order to preserve the best speech quality while reducing the bit rate, it uses sophisticated speech-coding algorithms that need more memory and computational load. The trade-offs between bit rate, speech quality, coding delay, and algorithm complexity are the main concerns for the system designers.

The simplest method to encode the speech is to quantize the time-domain waveform for the digital representation of speech, which is known as pulse code modulation (PCM). This linear quantization requires at least 12 bits per sample to maintain a satisfactory speech quality. Since most telecommunication systems use 8 kHz sampling rate, PCM coding requires a bit rate of 96 kbps. As briefly introduced in Chapter 1, lower bit rate can be achieved by using logarithmic quantization such as the μ-law or A-law companding, which compresses speech to 8 bits per sample and reduces the bit rate to 64 kbps. Further bit-rate reduction at 32 kbps can be achieved using the adaptive differential PCM (ADPCM), which uses adaptive predictor and quantizer to track the input speech signal.

Analysis–synthesis coding methods can achieve higher compression rate by analyzing the spectral parameters that represent the speech production model, and transmit these parameters to the receiver for synthesizing the speech. This type of coding algorithm is called vocoder (voice coder) since it uses an explicit speech production model. The most widely used vocoder uses the linear predictive coding (LPC) technique, which will be focused in this chapter.

The LPC method is based on the speech production model including excitation input, gain, and vocal-tract filter. It is necessary to determine a given segment or frame (usually in the range of 5–30 ms) in voiced or unvoiced speech. Segmentation is formed by multiplying the speech signal by a Hamming window. The successive windows are overlapped. For a voiced speech, the pitch period is estimated and used to generate the periodic excitation input. For an unvoiced speech, a random noise will be used as the excitation input. The vocal tract is modeled as an all-pole digital filter. The filter coefficients can be estimated by the Levinson–Durbin recursive algorithm, which will be introduced in Section 11.2.

In recent years, many LPC-based speech CODECs, especially code-excited linear predictive (CELP) at bit rate of 8 kbps or lower have been developed for wireless and network applications. The CELP-type speech CODECs are widely used in applications including wireless mobile and IP telephony communications, streaming media services, audio and video conferencing, and digital radio broadcastings. These speech CODECs include the 5.3 to 6.3-kbps ITU-T G.723.1 for multimedia communications, the low-delay G.728 at 16 kbps, the G.729 at 8 kbps, and the ISO (International Organization for Standardization) MPEG-4 CELP coding. In addition, there are regional standards that include Pan-European digital cellular radio (GSM) standard at 13 kbps, and GSM adaptive multirate (AMR) for third generation (3G) digital cellular telecommunication systems.

11.2 Overview of CELP Vocoders

CELP algorithms use an LPC approach. The coded parameters are analyzed to minimize the perceptually weighted error via a closed-loop optimization procedure. All CELP algorithms share the same basic functions including short-term synthesis filter, long-term prediction synthesis filter (or adaptive codebook), perceptual weighted error minimization procedure, and fixed-codebook excitation.

The basic structure of the CELP coding system is illustrated in Figure 11.1. The following three components can be optimized to obtain good synthesized speech:

1. time-varying filters, including short-term LPC synthesis filter $1/A(z)$, long-term pitch synthesis filter $P(z)$ (adaptive codebook), and post filter $F(z)$;

2. perceptually based error minimization procedure related to the perceptual weighting filter $W(z)$; and

3. fixed-codebook excitation signal $e_u(n)$, including excitation signal shape and gain.

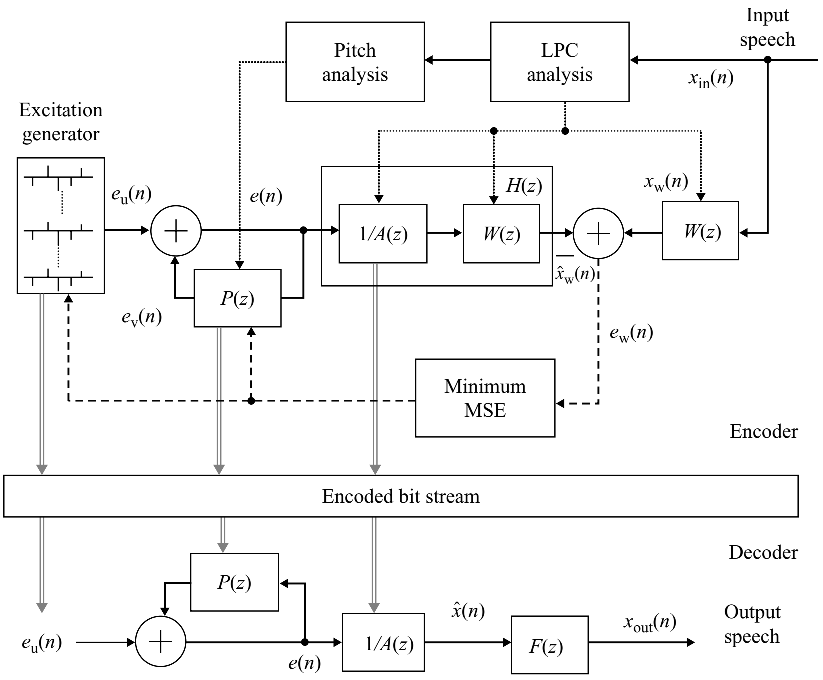


Figure 11.1 Block diagram of LPC coding scheme. The top portion of the diagram is the encoder, and the bottom portion is the decoder

In the encoder, the LPC and pitch analysis modules analyze speech to obtain the initial parameters for the speech synthesis model. Following these two modules, speech synthesis is conducted to minimize the weighted error. To develop an efficient search procedure, the number of operations can be reduced by moving the weighting filter into two branches before the error signal as shown in Figure 11.1. In the encoder, $x_{\text{in}}(n)$ is the input speech, $x_{\text{w}}(n)$ is the original speech weighted by the perceptual weighting filter $W(z)$, $\hat{x}_{\text{w}}(n)$ is the weighted reconstructed speech by passing excitation signal $e(n)$ through the combined filter $H(z)$, $e_{\text{u}}(n)$ is the excitation from the codebook, $e_{\text{v}}(n)$ is the output of pitch predictor $P(z)$, and $e_{\text{w}}(n)$ is the weighted error.

Parameters including excitation index, quantized LPC coefficients, and pitch predictor coefficients are encoded and transmitted. At the receiver, these parameters are used to synthesize the speech. The filter $W(z)$ is used only for minimizing the mean-square error loop, and its coefficients are not encoded. The coefficients of the post filter $F(z)$ are derived from the LPC coefficients and/or from the reconstructed speech.

In the decoder, the excitation signal $e(n)$ is first passed through the long-term pitch synthesis filter $P(z)$ and then the short-term LPC synthesis filter $1/A(z)$. The reconstructed signal $\hat{x}(n)$ is sent to the post filter $F(z)$, which emphasizes speech formants and attenuates the spectral valleys between formants.

11.2.1 Synthesis Filter

The time-varying short-term synthesis filter $1/A(z)$ and the long-term synthesis filter $P(z)$ are updated frame by frame using the Levinson–Durbin recursive algorithm. The synthesis filter $1/A(z)$ is expressed as

$$1/A(z) = \frac{1}{1 - \sum_{i=1}^{p} a_i z^{-i}}, \tag{11.1}$$

where a_i is the short-term LPC coefficient and p is the filter order.

The most popular method to calculate the LPC coefficients is the autocorrelation method. Due to the characteristics of speeches, we apply windows to calculate the autocorrelation coefficients as follows:

$$R_n(j) = \sum_{m=0}^{N-1-j} s_n(m)s_n(m+j), \qquad j = 0, 1, 2, \ldots, p, \tag{11.2}$$

where N is the window (or frame) size, n is the frame index, and m is the sample index in the frame.

We need to solve the following matrix equation to derive the prediction filter coefficients a_i:

$$\begin{bmatrix} R_n(0) & R_n(1) & \cdots & R_n(p-1) \\ R_n(1) & R_n(0) & \cdots & R_n(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ R_n(p-1) & R_n(p-2) & \cdots & R_n(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_p \end{bmatrix} = \begin{bmatrix} R_n(1) \\ R_n(2) \\ \vdots \\ R_n(p) \end{bmatrix}. \tag{11.3}$$

The left-hand side matrix is symmetric, all the elements on its main diagonal are equal, and the elements on any other diagonal parallel to the main diagonal are also equal. This square matrix is Toeplitz. Several efficient recursive algorithms have been derived for solving Equation (11.3). The most widely used algorithm is the Levinson–Durbin recursion summarized as follows:

$$E_n^{(0)} = R_n^{(0)} \tag{11.4}$$

$$k_i = \frac{R_n(i) - \sum_{j=1}^{i-1} a_j^{(i-1)} R_n(|i-j|)}{E_n^{(i-1)}} \tag{11.5}$$

$$a_i^{(i)} = k_i \tag{11.6}$$

$$a_j^{(i)} = a_j^{(i-1)} - k_i a_{i-j}^{(i-1)} \qquad 1 \le j \le i-1 \tag{11.7}$$

$$E_n^{(i)} = \left(1 - k_i^2\right) E_n^{(i-1)}. \tag{11.8}$$

After solving these equations recursively for $i = 1, 2, \ldots, p$, the parameters a_i are given by

$$a_j = a_j^{(p)} \qquad 1 \le j \le p. \tag{11.9}$$

Example 11.1: Consider the order $p = 3$ and given autocorrelation coefficients $R_n(j)$, $j = 0, 1, 2, 3$, for a frame of speech signal. Calculate the LPC coefficients.

We need to solve the following matrix equation:

$$\begin{bmatrix} R_n(0) & R_n(1) & R_n(2) \\ R_n(1) & R_n(0) & R_n(1) \\ R_n(2) & R_n(1) & R_n(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} R_n(1) \\ R_n(2) \\ R_n(3) \end{bmatrix}.$$

This matrix equation can be solved recursively as follows:

For $i = 1$:

$$E_n^{(0)} = R_n^{(0)}$$

$$k_1 = \frac{R_n(1)}{E_n^{(0)}} = \frac{R_n(1)}{R_n(0)}$$

$$a_1^{(1)} = k_1 = \frac{R_n(1)}{R_n(0)}$$

$$E_n^{(1)} = \left(1 - k_1^2\right) R_n(0) = \left[1 - \frac{R_n^2(1)}{R_n^2(0)}\right] R_n(0).$$

For $i = 2$, $E_n^{(1)}$ and $a_1^{(1)}$ are available from $i = 1$. Thus, we have

$$k_2 = \frac{R_n(2) - a_1^{(1)} R_n(1)}{E_n^{(1)}} = \frac{R_n(0)R_n(2) - R_n^2(1)}{R_n^2(0) - R_n^2(1)}$$

$$a_2^{(2)} = k_2$$

$$a_1^{(2)} = a_1^{(1)} - k_2 a_1^{(1)} = (1 - k_2)\, a_1^{(1)}$$

$$E_n^{(2)} = \left(1 - k_2^2\right) E_n^{(1)}.$$

For $i = 3$, $E_n^{(2)}$, $a_1^{(2)}$, and $a_2^{(2)}$ are available from $i = 2$. Thus, we get

$$\begin{aligned} k_3 &= \frac{R_n(3) - \left[a_1^{(2)} R_n(2) + a_2^{(2)} R_n(1)\right]}{E_n^{(2)}} \\ a_3^{(3)} &= k_3 \\ a_1^{(3)} &= a_1^{(2)} - k_3 a_2^{(2)} \\ a_2^{(3)} &= a_2^{(2)} - k_3 a_1^{(2)}. \end{aligned}$$

Finally, we have

$$a_0 = 1, a_1 = a_1^{(3)},\ a_2 = a_2^{(3)},\ a_3 = a_3^{(3)}.$$

We can use MATLAB functions provided in the *Signal Processing Toolbox* to calculate LPC coefficients. For example, the LPC coefficients can be calculated using the Levinson–Durbin recursion as

```
[a,e] = levinson(r,p)
```

The parameter `r` is a deterministic autocorrelation sequence (vector), `p` is the order of denominator polynomial $A(z)$, `a` $= [a(1)\ a(2) \cdots a(p+1)]$ where $a(1) = 1$, and the prediction error `e`.

The function `lpc(x,p)` determines the coefficients of forward linear predictor by minimizing the prediction error in the least-square sense. The command

```
[a,g] = lpc(x,p)
```

finds the coefficients of a linear predictor that predicts the current value of the real-valued time series `x` based on past samples. This function returns prediction coefficients `a` and error variances `g`.

Example 11.2: Given a speech file `voice4.pcm`, calculate the LPC coefficients and spectral response using the function `levinson( )`. Also, compare the contours of speech spectrum with the synthesis filter's frequency response. Assume that the LPC order is 10 and Hamming window size is 256. The partial MATLAB code to calculate the LPC coefficients is listed in Table 11.1. The complete MATLAB program is given in `example11_2.m`. The magnitude response of the synthesis filter shows the envelope of the speech spectrum as shown in Figure 11.2.

Example 11.3: Calculate the LPC coefficients as in Example 11.2 using the function `lpc` instead of `levinson`. In this case, `g` and `e` are identical if using the same order, and the LPC coefficients are identical to Example 11.2.

Using a high-order synthesis filter, the frequency response is closer to the original speech spectrum. Figure 11.3 shows the use of high order (42) to calculate the LPC coefficients using `lpc`.

Table 11.1 MATLAB code to calculate LPC coefficients and frequency response

```
fid=fopen('voice4.pcm','r');         % Open the pcm data file
b= fread(fid,20000,'short');         % Read 20000 samples to b

% Windowing
w=hamming(frame);                    % Generate Hamming window
x1=b((start+1):(start+frame));
x=x1.*w;                             % Windowing

y=fft(x,fftL);                       % FFT of the specified block
py=10*log10(y.*conj(y)/fftL);        % Magnitude response
fclose(fid);                         % Close the file

% Calculation of autocorrelation
m=0;
while (m<=lpcOrder);
r(m+1)=sum(x((m+1):(frame)).*x(1:frame-m));
r(m+1) = r(m+1)/frame; m=m+1;
end;

% Levinson algorithm
[a,e]=levinson(r,lpcOrder);
```

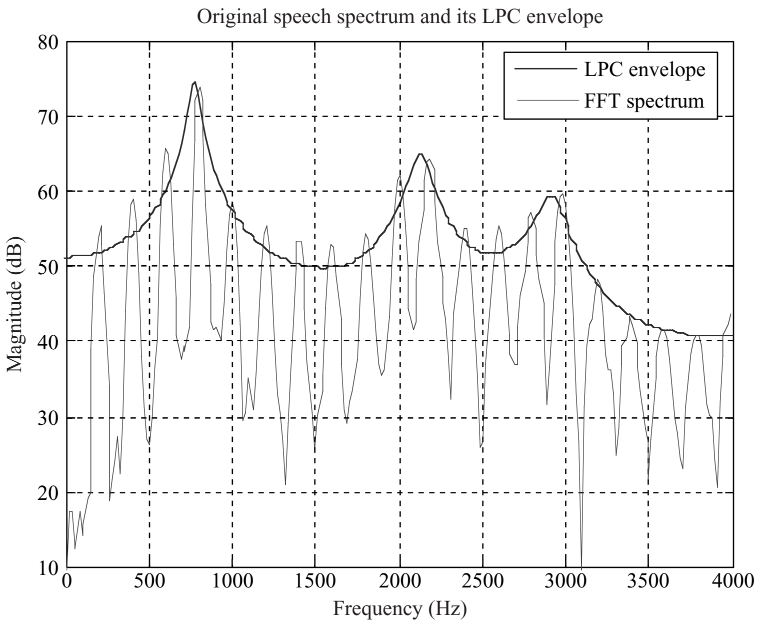


Figure 11.2 Spectral envelope of speech spectrum derived from the synthesis filter

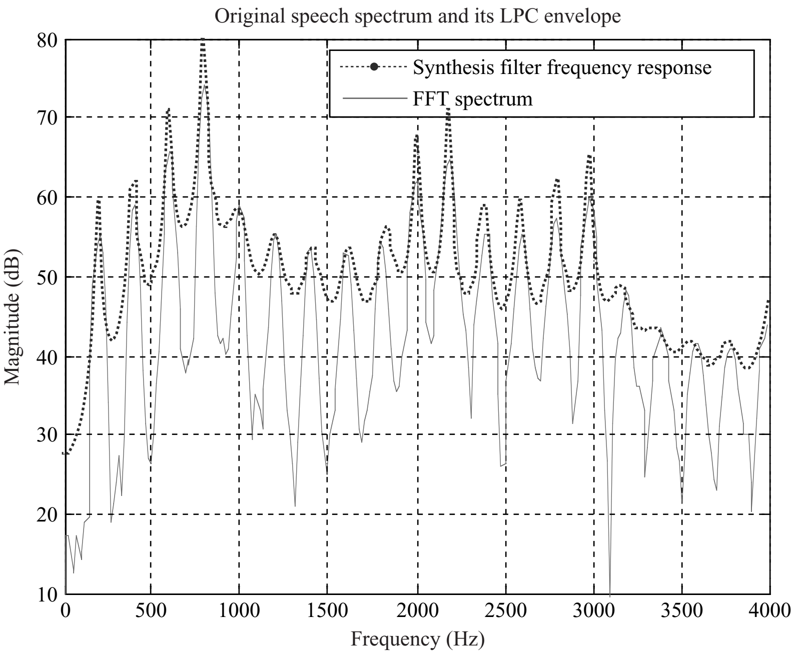


Figure 11.3 Magnitude response of synthesis filter with higher order

11.2.2 Long-Term Prediction Filter

The long-term prediction filter $P(z)$ models the long-term correlation in speech to provide fine spectral structure, and it has the following general form:

$$P(z) = \sum_{i=-I}^{I} b_i z^{-(L_{\text{opt}}+i)}, \tag{11.10}$$

where L_{opt} is the optimum pitch period, and b_i are the coefficients. Typically, $I = 0$ for one tap, $I = 1$ for three taps, and $I = 2$ for 5-tap pitch filters.

In some cases, the long-term prediction filter is also called an adaptive codebook since the excitation signals are adaptively updated. An example is given in the ITU-T G.723.1, which uses a fifth-order long-term prediction filter.

11.2.3 Perceptual Based Minimization Procedure

A reduction in perceived distortion is possible if the noise spectrum is shaped to place the majority of the error in the formant (high-energy) regions where human ears are relatively insensitive because of the auditory masking. On the other hand, more subjectively disturbing noise in the formant nulls must be reduced. The synthesis filter in Equation (11.1) is used to construct the perceptual weighting filter. The formant perceptual filter has the following transfer function:

$$W(z) = \frac{1 - \sum_{i=1}^{p} a_i z^{-i} \gamma_1^i}{1 - \sum_{i=1}^{p} a_i z^{-i} \gamma_2^i} = \frac{A(z/\gamma_2)}{A(z/\gamma_1)}, \tag{11.11}$$

where $0 < \gamma < 1$ is a bandwidth expansion factor with typical values $\gamma_1 = 0.9$ and $\gamma_2 = 0.5$.

The synthesis filter $1/A(z)$ and perceptual weighting filter $W(z)$ can be combined to form

$$H(z) = W(z)/A(z). \tag{11.12}$$

The impulse response of a combined or cascaded filter is denoted by $\{h(n), n = 0, 1, \ldots, N_{\text{sub}} - 1\}$, where N_{sub} is the subframe length.

11.2.4 Excitation Signal

For efficient temporal analysis, a speech frame is usually divided into a number of subframes. For example, there are four subframes defined in the G.723.1. For each subframe, the excitation signal is generated and the error is minimized to find the optimum excitation.

The excitation varies between the pulse train and the random noise. A general form of the excitation signal $e(n)$ shown in Figure 11.1 can be expressed as

$$e(n) = e_{\text{v}}(n) + e_{\text{u}}(n), \qquad 0 \le n \le N_{\text{sub}} - 1, \tag{11.13}$$

where $e_{\text{u}}(n)$ is the excitation from a fixed or secondary codebook given by

$$e_{\text{u}}(n) = G_{\text{u}} c_k(n), \qquad 0 \le n \le N_{\text{sub}} - 1, \tag{11.14}$$

where G_{u} is the gain, N_{sub} is the length of the excitation vector (or the subframe), and $c_k(n)$ is the nth-element of kth-vector in the codebook. In Equation (11.13), $e_{\text{v}}(n)$ is the excitation from the long-term prediction filter expressed as

$$e_{\text{v}}(n) = \sum_{j=-I}^{I} e(n + j - L_{\text{opt}}) b(j), \qquad n = 0, 1, \ldots, N_{\text{sub}} - 1. \tag{11.15}$$

Passing $e(n)$ through the combined filter $H(z)$, we have perceptually weighted synthesis speech given by

$$\hat{x}_{\text{w}}(n) = v(n) + u(n) = \sum_{j=0}^{n} e_{\text{v}}(j) h(n-j) + \sum_{j=0}^{n} e_{\text{u}}(j) h(n-j), \qquad n = 0, 1, \ldots, N_{\text{sub}} - 1, \tag{11.16}$$

where the first term $v(n) = \sum_{j=0}^{n} e_{\text{v}}(j) h(n-j)$ is from the long-term predictor and the second term $u(n) = \sum_{j=0}^{n} e_{\text{u}}(j) h(n-j)$ is from the secondary codebook. Therefore, the weighted error $e_{\text{w}}(n)$ can be described as

$$e_{\text{w}}(n) = x_{\text{w}}(n) - \hat{x}_{\text{w}}(n). \tag{11.17}$$

The squared error of $e_{\mathrm{w}}(n)$ is given by

$$E_{\mathrm{w}} = \sum_{n=0}^{N_{\mathrm{sub}}-1} e_{\mathrm{w}}^2(n). \tag{11.18}$$

By computing the above equations for all possible parameters including the pitch predictor coefficients b_i (pitch gain), lag L_{opt} (delay), optimum secondary excitation code vector $\mathbf{c}_k$ with elements $c_k(n)$, $n = 0, \ldots, N_{\mathrm{sub}} - 1$, and gain G_{u}, we can find the minimum $E_{\mathrm{w,min}}$ as

$$E_{\mathrm{w,min}} = \min\{E_{\mathrm{w}}\} = \min\left\{\sum_{n=0}^{N_{\mathrm{sub}}-1} e_{\mathrm{w}}^2(n)\right\}. \tag{11.19}$$

This minimization procedure is a joint optimization between the pitch prediction (adaptive codebook) and the secondary (fixed) codebook excitations. The joint optimization of all the excitation parameters, including L_{opt}, b_i, G_{u}, and $\mathbf{c}_k$, is possible but it is computationally intensive. A significant simplification can be achieved by assuming that the pitch prediction parameters are optimized independently from the secondary codebook excitation parameters. We call this a separate optimization procedure.

In a separate optimization, the excitation $e(n)$ given in Equation (11.13) contains only $e_{\mathrm{v}}(n)$ because $e_{\mathrm{u}}(n) = 0$. The optimized pitch lag and pitch gain are first found using historical excitations. The contribution of the pitch prediction $v(n)$ can be subtracted from the target signal $x_{\mathrm{w}}(n)$ to form a new target signal. The second round of minimization is conducted by approximating the secondary codebook contribution to this new target signal. An example of separate optimization procedure can be found in G.723.1.

11.2.5 Algebraic CELP

The algebraic CELP (ACELP) implies that the structure of the codebook is used to select the excitation codebook vector. The codebook vector consists of a set of interleaved permutation codes containing few nonzero elements [3, 4]. The ACELP fixed-codebook structures have been used in G.729 and G.723.1 low-bit rate at 5.3 kbps, and WCDMA AMR. The fixed-codebook structure used in G.729 is shown in Table 11.2.

In Table 11.2, m_k is the pulse position, k is the pulse number, the interleaving depth is 5. In this codebook, each codebook vector contains four nonzero pulses indexed by i_k. Each pulse can have either the amplitudes of $+1$ or -1, and can assume the positions given by Table 11.2. The codebook vector, $\mathbf{c}_k$, is determined by placing four unit pulses at the locations m_k multiplied with their signs (± 1) as follows:

$$c_k(n) = s_0\delta(n-m_0) + s_1\delta(n-m_1) + s_2\delta(n-m_2) + s_3\delta(n-m_3), \qquad n = 0, 1, \ldots, 39, \tag{11.20}$$

where $\delta(n)$ is a unit pulse.

Table 11.2 G.729 ACELP codebook

Pulse i_k	Sign s_k	Position m_k	Number of bits to code (sign + position)
i_0	± 1	0, 5, 10, 15, 20, 25, 30, 35	$1+3$
i_1	± 1	1, 6, 11, 16, 21, 26, 31, 36	$1+3$
i_2	± 1	2, 7, 12, 17, 22, 27, 32, 37	$1+3$
i_3	± 1	3, 8, 13, 18, 23, 28, 33, 38, 4, 9, 14, 19, 24, 29, 34, 39	$1+4$

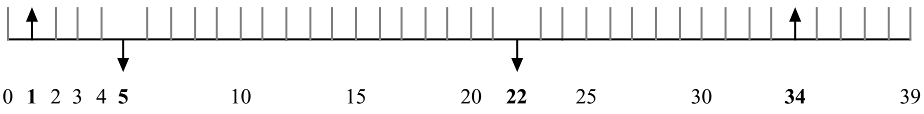


Figure 11.4 Four pulse locations in a 40-sample frame

Example 11.4: Assuming that all four pulses in a frame are located as shown in Figure 11.4, these pulse locations and signs are found by minimizing the squared error defined in Equation (11.18). These pulse positions are confined within certain positions with interleaving depth 5 as defined by Table 11.2. The pulse positions and signs can be coded into codeword as shown in Table 11.3. We assume that the positive sign is encoded with 0 and the negative sign is encoded with 1. The sign bit is the MSB. This example shows how to encode ACELP excitations. We need $4 + 4 + 4 + 5 = 17$ bits to code this ACELP information.

11.3 Overview of Some Popular CODECs

Different vocoders are used for different applications that depend on the bit rate, robustness to channel errors, algorithm delay, complexity, and sampling rate. Two most popular algorithms G.729 and G.723.1 are widely used in real-time communications over the Internet due to their low-bit rates and high qualities. The AMR is mainly used for the GSM and the WCDMA wireless systems for its flexible rate adaptation to error conditions of wireless channels.

11.3.1 Overview of G.723.1

One example of a CELP CODEC is the ITU-T G.723.1 [1], whose basic structure is illustrated in Figure 11.5. The predictor coefficients are updated using the forward adaptation method. The encoder operates on a frame of 240 samples, i.e., 30 ms at the 8 kHz sampling rate. The input speech is first buffered into frames of 240 samples, filtered by a highpass filter to remove the DC component, and then divided into four subframes of 60 samples each.

For every subframe m, a 10th-order LPC filter is computed using the highpass filtered signal $x_m(n)$. For each subframe, a window of 180 samples is centered on the current subframe as shown in Figure 11.6. A Hamming window is applied to these samples. Eleven autocorrelation coefficients are computed from the windowed signal. The linear predictive coefficients are found using the Levinson–Durbin algorithm. For every input frame, four LPC sets will be computed, one for each subframe. These LPC coefficient sets are used to construct the short-term perceptual weighting filter.

Table 11.3 An example of coding the ACELP pulses into codeword

Pulse i_k	Sign s_k	Position m_k	Sign + position = encoded code
i_0	`+1`	`5 (k=1)`	`0<<3 + 1 = 0001`
i_1	`-1`	`1 (k=0)`	`1<<3 + 0 = 1000`
i_2	`-1`	`22 (k=4)`	`1<<3 + 4 = 1100`
i_3	`+1`	`34 (k=14)`	`0<<4 + 14 = 01110`

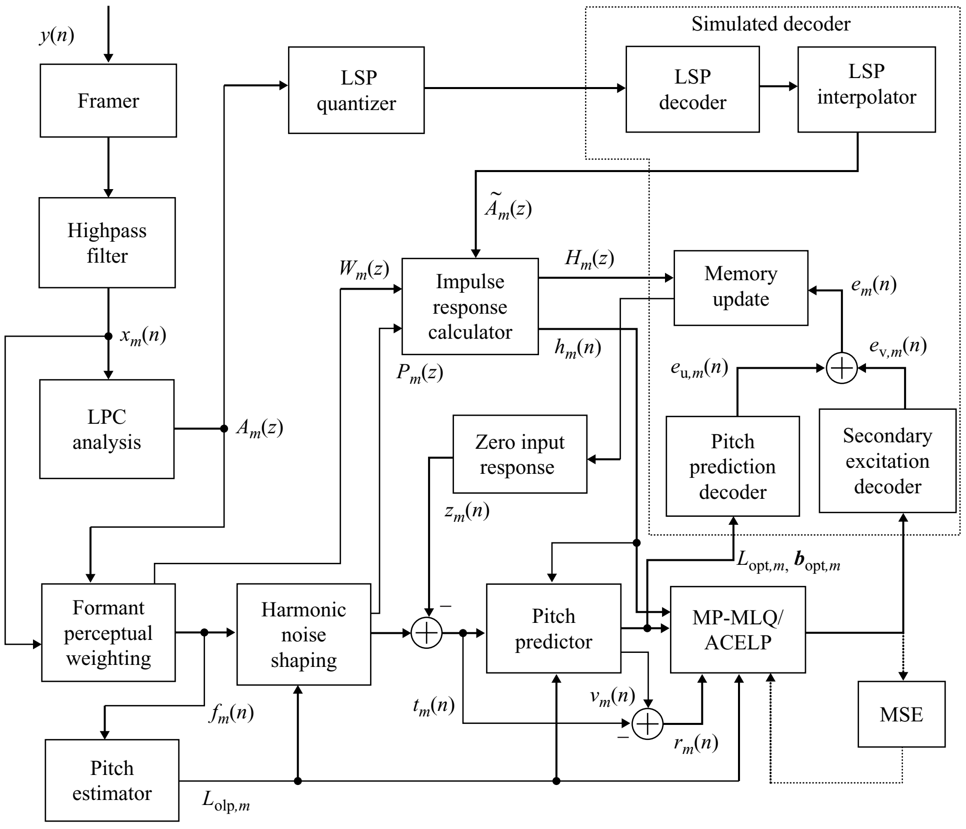


Figure 11.5 Block diagram of G.723.1 CODEC

Calculating the LPC coefficients requires the past, current, and future subframes. The LPC synthesis filter is defined as

$$\frac{1}{A_m(z)} = \frac{1}{1 - \sum_{i=1}^{10} a_{i,m} z^{-i}}, \tag{11.21}$$

where $m = 0, 1, 2, 3$ is the subframe index. As shown in Figure 11.5, the LPC filter coefficients of the last subframe are converted to LSP (line spectral pairs) coefficients. The reason for doing LPC to

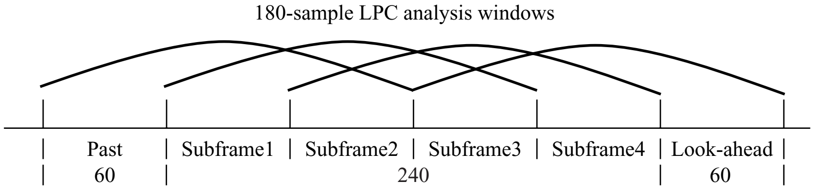


Figure 11.6 G.723.1 LPC analysis windows vs. subframes

Table 11.4 Procedures of LPC coefficient calculation and reconstruction

Seq.	Computed parameters	Subframe 0	Subframe 1	Subframe 2	Subframe 3
1	LPC coefficients	$\{a_{i,0}\}$	$\{a_{i,1}\}$	$\{a_{i,2}\}$	$\{a_{i,3}\}$
2	LSP coefficients				$\{\beta_{i,3}\}$
3	LSP coefficients quantization and dequantization				$\{\hat{\beta}_{i,3}\}$
4	Interpolated LSP coefficients	$\{\hat{\beta}_{i,0}\}$	$\{\hat{\beta}_{i,1}\}$	$\{\hat{\beta}_{i,2}\}$	$\{\hat{\beta}_{i,3}\}$
5	Reconstructed LPC coefficients	$\{\hat{a}_{i,0}\}$	$\{\hat{a}_{i,1}\}$	$\{\hat{a}_{i,2}\}$	$\{\hat{a}_{i,3}\}$

LSP coefficients conversion is to take the advantages of two properties of LSP coefficients: to verify the stability of the filter and to have higher coefficients correlation among subframes. The first property can be used to make synthesis filter stable after quantization, and the second is used to further remove redundancy. The LPC coefficients calculation and quantization can be further explained with Table 11.4 and the following procedures:

1. Compute $\{a_{i,m}\}$ for subframes $m = 0, 1, 2, 3$ and 10th-order LPC coefficients $i = 1, \ldots, 10$. With $\{a_{i,m}\}$, we can construct Equation (11.21). The unquantized LPC coefficients are used to construct the short-term perceptual weighting filter $W_m(z)$, which will be used to filter the entire frame to obtain the perceptually weighted speech signal.

2. Convert the last subframe's $\{a_{i,3}\}$ to LSP coefficients $\{\beta_{i,3}\}$.

3. Use vector quantization to quantize the 10 LSP coefficients into LSP index for transmitting. Dequantize the LSP coefficients to $\{\hat{\beta}_{i,3}\}$.

4. Use the LSP coefficients $\{\hat{\beta}_{i,3}\}$ from the current frame and the last frame to interpolate the LSP coefficients for each subframe $\{\hat{\beta}_{i,m}\}$.

5. Convert LSP coefficients $\{\hat{\beta}_{i,m}\}$ back to LPC coefficients $\{\hat{a}_{i,m}\}$ and construct the synthesis filter $1/\tilde{A}_m(z)$ for each subframe. Note that even in the encoder side, we also need this in order for both sides to use the same set of synthesis filter. Decoding side never has the unquantized LPC coefficients.

The pitch estimation is performed on two adjacent subframes of 120 samples. The pitch period is searched in the range from 18 to 142 samples. Using the estimated open-loop pitch period $L_{\mathrm{olp},m}$, a harmonic noise-shaping filter $P_m(z)$ can be constructed. The combination of the LPC synthesis filter with the formant perceptual weighting filter and the harmonic noise-shaping filter, $H_m(z) = W_m(z)\,P_m(z)\,/\tilde{A}_m(z)$, is used to create an impulse response $h_m(n)(n = 0, \ldots, 59)$. The adaptive excitation signals $e_{\mathrm{v},m}(n)$ and the secondary excitation signal $e_{\mathrm{u},m}(n)$ are filtered by this combined filter to provide the zero-state responses $u_m(n)$ and $v_m(n)$, respectively.

In adaptive codebook excitation, a fifth-order pitch predictor is used. The optimum pitch periods ($L_{\mathrm{opt},m}$) of subframe 0 and 2 are computed via closed-loop vector quantization around the open-loop pitch estimate $L_{\mathrm{olp},m}$. The optimum pitch periods in subframes 1 and 3 are searched for differential values around the previous optimum pitch periods of subframes 0 and 2, respectively. The pitch periods of subframes 0 and 2, and the differential values for subframes 1 and 3 are transmitted to the decoder.

Let $\mathbf{b}_k$, $k = 0, \ldots, N_{\mathrm{ltp}} - 1$ denotes the kth gain vector and its elements are $b_k(n)$, $n = 0, 1, 2, 3, 4$, where N_{ltp} is the size of pitch gain codebook. The adaptive codebook excitation is formed by

$$e_{\mathrm{v},m}(n) = \sum_{j=0}^{4} b_k(j) e_m(-L_m - 2 + n + j), \qquad 0 \le n \le 59, \quad 0 \le k \le N_{\mathrm{ltp}} - 1, \tag{11.22}$$

where $L_m = L_{\mathrm{olp},m} - 1$, $L_{\mathrm{olp},m}$, and $L_{\mathrm{olp},m} + 1$ for subframes 0 and 2, or $L_m = L_{\mathrm{opt},m-1} - 1$, $L_{\mathrm{opt},m-1}$, $L_{\mathrm{opt},m-1} + 1$, and $L_{\mathrm{opt},m-1} + 2$ for subframes 1 and 3, where $L_{\mathrm{opt},m-1}$ is the optimum pitch lag in the previous subframe 0 or 2.

By using the closed-loop quantization, the adaptive codebook contribution is computed as

$$v_m(n) = e_{\mathrm{v},m}(n) * h_m(n) = \sum_{j=0}^{n} e_{\mathrm{v},m}(j) h_m(n-j), \qquad 0 \le n \le 59. \tag{11.23}$$

The optimization procedure minimizes the mean-squared error E_{ac} given as

$$E_{\mathrm{ac}} = \sum_{n=0}^{59} [t_m(n) - v_m(n)]^2, \qquad 0 \le k \le N_{\mathrm{ltp}} - 1. \tag{11.24}$$

The best lag $L_{\mathrm{opt},m}$ and tap vector $\mathbf{b}_{\mathrm{opt},m}$ that minimize E_{ac} are identified in the current subframe as the adaptive codebook's lag value and gain vector, respectively. The optimum reconstructed adaptive codebook contribution is expressed as

$$v_m(n) = \sum_{j=0}^{4} b_{\mathrm{opt},m}(j) \sum_{k=0}^{n} e_m(k - L_{\mathrm{opt},m} - 2 + j) h_m(n-k), \qquad 0 \le n \le 59. \tag{11.25}$$

The contribution of the adaptive codebook $v_m(n)$ is then subtracted from the initial target vector $t_m(n)$ to obtain the new target signal $r_m(n)$ for the secondary codebook quantization.

Applying similar close-loop quantization technique, the secondary codebook parameters can be found by minimizing the error signal defined by

$$E_{\mathrm{sc}} = \sum_{n=0}^{59} [r_m(n) - u_m(n)]^2, \tag{11.26}$$

where the fixed-codebook contribution $u_m(n)$ is a function of pulse location m_k, sign s_k, and gain.

Finally, the residual signals are encoded by secondary excitation, either by ACELP or by MP-MLQ (multipulse maximum likelihood quantization) according to the bit rate used. The transmitted information to the receiver includes the LSP vector index, adaptive codebook lag and tap indices, and secondary excitation information.

The bit allocation of a vocoder presents a clear picture of what algorithm has been used, relationship between frame and subframe, encoded bit rate, and how many bits per frame. Table 11.5 lists the bit allocations of G.723.1 at 5.3 and 6.3 kbps. The major differences between these two rates are in the coding of pulse positions and amplitudes.

ITU G.723.1 has three Annexes. Annex A defines silence compression scheme, Annex B defines alternative specification based on floating-point arithmetic, and Annex C defines scalable channel-coding scheme for wireless applications.

Table 11.5 Bit allocations of G.723.1

Parameters coded		Subframe 0	Subframe 1	Subframe 2	Subframe 3	Total
LPC indices						24
Adaptive codebook lags		7	2	7	2	18
All the gains combined		12	12	12	12	48
6.3 kbps	Pulse positions	20	18	20	18	73*
	Pulse signs	6	5	6	5	22
	Grid index	1	1	1	1	4
	Total					**189**
5.3 kbps	Pulse positions	12	12	12	12	48
	Pulse signs	4	4	4	4	16
	Grid index	1	1	1	1	4
	Total					**158**

*Using the fact that the number of codewords in a fixed codebook is not a power of 2 or 3, additional bits are saved by combining the four MSBs of each pulse position index into a single 13-bit word.

11.3.2 Overview of G.729

G.729 is a low-bit rate, toll-quality CODEC using CS-ACELP (conjugate-structure algebraic-excited linear-prediction) based on the CELP coding algorithm. The coder operates on speech frames of 10 ms (80 samples) at sampling rate of 8 kHz. Each frame is further divided into two 40-sample subframes. For every frame, the speech signal is processed to extract the parameters of the CELP model including LPC coefficients, adaptive- and fixed-codebook indices and gains. These parameters are quantized, encoded, and transmitted over the channel. G.729 encoder block diagram is shown in Figure 11.7.

The LPC analysis window uses 6 subframes, 120 samples from the past speech frames, 80 samples in the present speech frame, and 40 samples from the future subframe. The future 40-sample subframe is

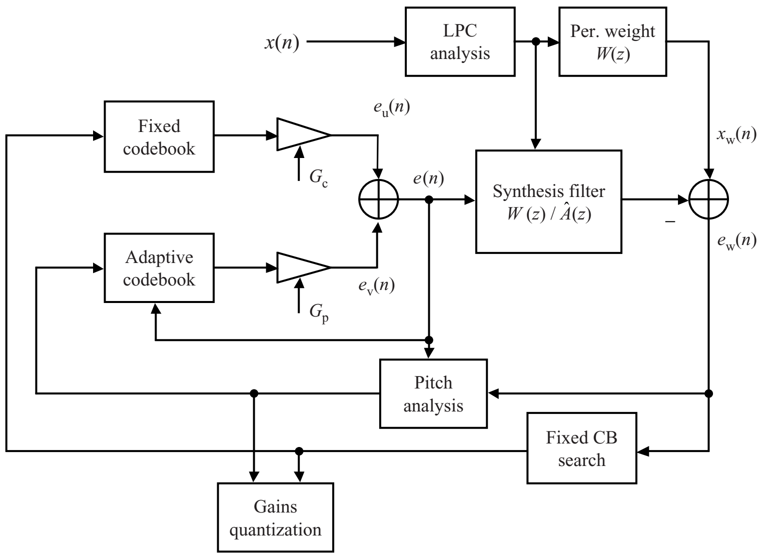


Figure 11.7 Encoding principles of the CS-ACELP encoder (adapted from G.729)

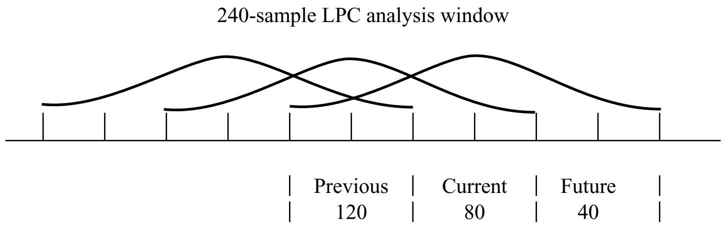


Figure 11.8 LPC analysis window operations

used as the 5-ms look-ahead in the LPC analysis, which introduces extra 5-ms algorithmic encoding delay. The window procedure is illustrated in Figure 11.8. Following the similar procedures as G.723.1, the LPC coefficients are computed and quantized. The combined reconstructed synthesis filter and perceptual weighting filter, $W(z)/\hat{A}(z)$, are used for synthesis as shown in Figure 11.7.

The excitation signal consists of the fixed-codebook vector $e_u(n)$ and the adaptive codebook vector $e_v(n)$, which are determined in each subframe. Closed-loop pitch analysis is used to find the adaptive codebook delay with fractional resolutions and gain. Different from the G.723.1, a fractional pitch is used to accurately represent the harmonic signals. In the first subframe, a fractional pitch delay T_1 with a resolution of $1/3$ is used in the range of $[19\frac{1}{3}, 84\frac{2}{3}]$, and the integer delay is used in the range of [85, 143]. For the second subframe, the resolution of only $1/3$-fractional delay T_2 is used. The pitch delay is encoded using 8 bits for the first subframe, and differentially encoded using 5 bits for the second subframe.

A 17-bit algebraic codebook is used for the fixed-codebook excitation. For algebraic codebook, each vector contains four nonzero pulses. The bit allocation for algebraic codebook is listed in Table 11.2. The 40-sample vector with four unit pulses is represented in Equation (11.20). The gains of the adaptive codebook (G_p) and fixed-codebook (G_c) contributions are vector quantized to 7 bits, with moving-averaging prediction applied to the fixed-codebook gain.

The block diagram of decoder is shown in Figure 11.9. The excitation and synthesis filter parameters are retrieved from the bit stream. First, the parameter's indices are extracted from the received bit stream. These indices are decoded to obtain the coder parameters corresponding to a 10-ms speech frame. The parameters include the LSP coefficients, two fractional pitch delays, two fixed-codebook vectors, and two sets of adaptive- and fixed-codebook gains. The LSP coefficients are interpolated and converted to the LPC filter coefficients for each subframe. For each 5-ms subframe, the following steps are performed: (1) the excitation is constructed by adding the adaptive- and fixed-codebook vectors scaled by their respective gains; (2) the speech is reconstructed by filtering the excitation through the LPC synthesis filter; and (3) the reconstructed speech signal is passed through a postprocessing stage, which includes an

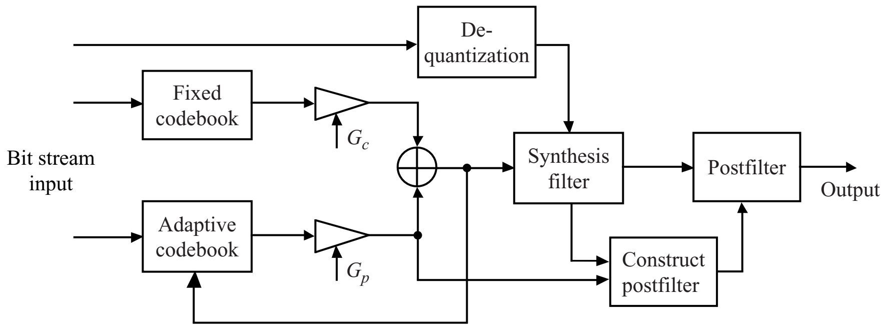


Figure 11.9 G.729 decoder

Table 11.6 Summary of G.729 bit rates of different annexes

Annexes	Annex A or Annex C	Annex D	Annex E	SID (Annex B)	Silence (Annex B)
Bits per 10 ms frame	80	64	118	16	0
Bit rate	8 kbps	6.4 kbps	11.8 kbps	1.6 kbps	0 kbps

adaptive postfilter based on the long-term and short-term synthesis filters, followed by a highpass filter and scaling operation.

In addition to G.729, other related annexes from ITU-T are listed here. They are used for different applications. The bit-rate information about these annexes is listed in Table 11.6.

- Annex A: ‘Reduced complexity 8-kbps CS-ACELP speech CODEC,’ which reduces about half-computational requirement at a minimal reduction in perceived quality.
- Annex B: ‘A silence compression scheme for G.729 optimized for terminals conforming to Recommendation V.70,’ which adds discontinuous transmission (DTX), voice activation detection (VAD), background noise modeling, comfort noise generation (CNG), and silence frame insertion to G.729 and G.729A.
- Annex C: ‘Reference floating-point implementation for G.729 CS-ACELP 8-kbps speech coding,’ which could be used for Pentium processor.
- Annexes D and E on 6.4 kbps and 11.8 kbps CS-ACELP speech-coding algorithms.
- Annexes F–I enhance capabilities of previous annexes (e.g., DTX/VAD/CNG) and integrate CODECs with different bit rates.

11.3.3 Overview of GSM AMR

GSM adaptive multirate (AMR) CODEC is a speech-coding standard introduced by the third generation partnership project (3GPP) for compression of toll-quality speech for mobile telephony applications. The AMR CODEC operates at eight different bit rates: 12.2, 10.2, 7.92, 7.40, 6.70, 5.90, 5.15, and 4.75 kbps. The data rate is selectable at run time by the system.

The vocoder processes signals using 20 ms frames. For compatibility with legacy systems, the 12.2 and 7.4 modes are compatible versions of the GSM-enhanced-full-rate and IS-136-enhanced-full-rate CODECs. In addition, the CODEC was designed to allow seamless switching on a frame-by-frame basis between the different modes listed as follows:

- *Channel mode*: GSM half-rate at 11.4 kbps or full-rate operation at 22.8 kbps.
- *Channel mode adaptation*: Control and selection of the full-rate or half-rate channel mode.
- *CODEC mode*: Bit-rate partitioning of the speech for a given channel mode.
- *CODEC mode adaptation*: Control and selection of the bit rates.

The AMR speech CODEC provides a high-quality speech service with the flexibility of the multirate approach, thus allowing a trade-off between the quality and the capacity as a function of the network

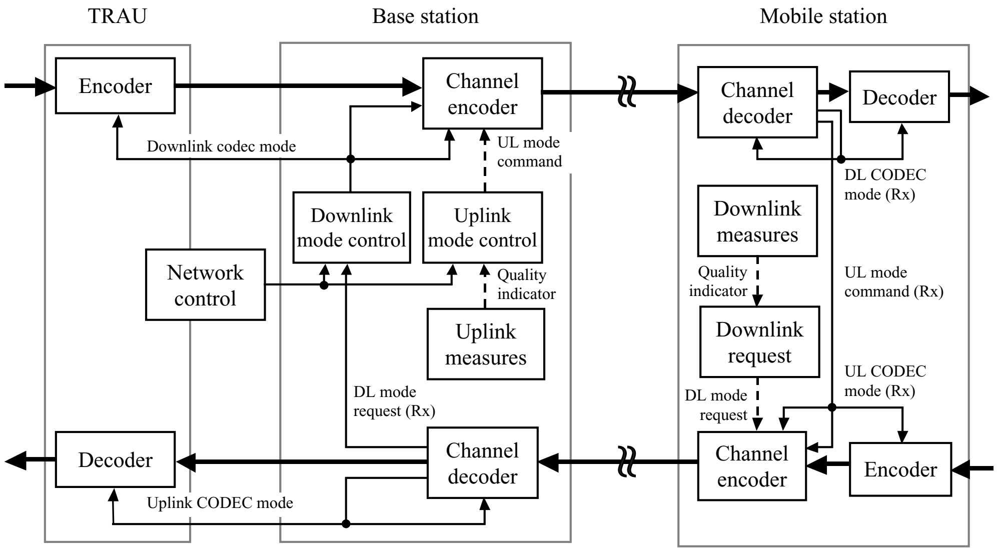


Figure 11.10 Block diagram of AMR system, where DL denotes downlink, UL denotes uplink, and Rx means received (adapted from GSM 06.71, version 7.0.2)

load. This flexibility is applicable to 2G and 3G networks. As the G.723.1 and G.729 algorithms, the GSM AMR also uses ACELP technique. Therefore, we will introduce its application in the real wireless networks, such as the rate adaptation rather than detailed algorithms.

In Figure 11.10, the speech encoder input is a 13-bit PCM signal from the mobile station, or from the public switched telephone network (PSTN) via an 8-bit A-law or μ-law to 13-bit linear PCM conversion. The encoder encodes the PCM signal to bit stream with the bit rate controlled by transcoding and rate adaptor unit (TRAU). The bit rates of the source CODEC for the adaptive multirate full-rate channel are listed in Table 11.7. The encoded bit stream is sent to the channel coding function to produce an encoded block. Full-rate bit stream has 456 bits and half rate has 228 bits.

The inverse operations are performed in the receive direction. The GSM 06.90 [7] describes the detailed mapping between the 160 input speech samples in 13-bit PCM format to the encoded bit stream. The decoder will reconstruct 160 speech samples from the bit steam.

Figure 11.10 presents a block diagram of the overall AMR system for uplink and downlink over the same radio interface. Mobile station, base transceiver station, and TRAU are shown in the figure. The AMR link adaptation, channel quality estimation, and inband signaling are given in GSM 05.09. The AMR multirate adaptation is based on the UL (uplink) and the DL (downlink) quality indicators that come from the radio channel quality measurements. The UL quality indicator is mapped to an UL mode command and the DL quality indicator is mapped to a DL mode request. The UL mode command and

Table 11.7 Relationship among speech bit rate, full rate, and half rate

	Encoded speech bit rate							
Speech bit rate (kbps)	12.2	10.2	7.95	7.40	6.70	5.90	5.15	4.75
Encoded bits per 20 ms frame	244	204	159	148	134	118	103	95
Gross bit rate at full-rate mode	22.8	22.8	22.8	22.8	22.8	22.8	22.8	22.8
Gross bit rate at half-rate mode			11.4	11.4	11.4	11.4	11.4	11.4

Table 11.8 Source bit rates for the AMR-WB CODEC [7]

Bits per 20 ms	477	461	397	365	317	285	253	177	132	35
Bit rate	23.85	23.05	19.85	18.25	15.85	14.25	12.65	8.85	6.6	1.75*

*Assuming SID (silence insertion descriptor) frames are continuously transmitted.

the DL mode request are sent to the transmitter using the communication reverse link. The UL CODEC mode and DL mode requests are sent as inband signals in the uplink radio channel. The DL CODEC mode and UL mode command are sent as inband signals in the downlink channel.

A GSM system using the AMR speech CODEC may select the optimum channel (half or full rate) and CODEC mode (speech and channel bit rates) to deliver the best combination of speech quality and system capacity. This flexibility provides a number of important benefits.

Improved speech quality in both half-rate and full-rate modes can be achieved by means of CODEC mode adaptation, i.e., by varying the balance between speech and channel coding for the same gross bit rate. When the radio quality is good (low-bit-error rate), the higher bit-rate CODEC is used. When the radio quality is worse, lower bit-rate CODEC is used while the overall channel bit rate is unchanged. Table 11.7 shows the bit allocations of all the rates.

In addition to AMR, 3GPP and ITU-T have standardized a wideband AMR, which is named as 3GPP GSM-AMR WB and also named as ITU-T G.722.2, for providing wideband telephony services. Wideband AMR is an adaptive multirate wideband CODEC with bit rates ranging from 6.6 to 23.85 kbps. The coder works on speech sampled at 16 kHz and a frame of 20 ms (320 speech samples). It supports VAD in combination with DTX and CNG.

The wideband speech is divided into two subbands; the lower subband from 75 Hz to 6.4 kHz and the upper subband from 6.4 to 7.0 kHz. The lower band signal is encoded based on the ACELP model. The higher band is encoded using the silence compression method, or treated as fractional noise and just encoded with spectral envelope. More details can be found in 3GPP technical specification TS 26.290.

11.4 Voice over Internet Protocol Applications

Voice over Internet protocol (VoIP) is the telephony service over the traditional data networks [9]. VoIP is also referred as IP (Internet protocol) telephony. The integration of the voice and data services is provided by packet-switched data networks. The development of VoIP has led to the cost-efficient gateway equipments that bridge analog telephony circuits and IP ports. The IP telephony converts the voice or fax into packet data suitable for transport over the networks. As a result, the expected next-generation communications systems that are capable of replacing the traditional PSTN are realized by integrating the voice and data over the IP.

This section uses an example to demonstrate the current status of VoIP system, its applications, and possible troubleshooting approaches.

11.4.1 Overview of VoIP

A simplified diagram of VoIP applications is shown in Figure 11.11. The functional units in Figure 11.11 are defined as follows:

1. *Analog-to-digital (A/D) and digital-to-analog (D/A) interface units*: The analog signals are converted to linear PCM samples for processing. The D/A converters convert the processed PCM digital samples to the analog signals.

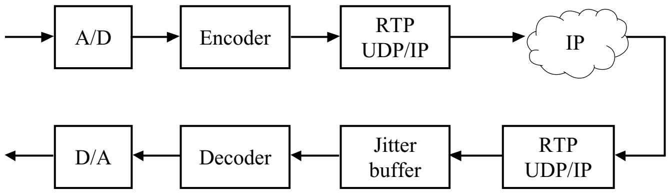


Figure 11.11 Simplified block diagram of VoIP applications

2. *Speech-coding unit*: The main processing unit encodes the PCM data samples received, or decodes the bit stream from network to the PCM data samples. Various compression algorithms could be used for different applications.

3. *Jitter buffer*: The jitter buffer is used to compensate for network jitters. This packet buffer holds incoming packets for a specified amount of time before forwarding them to the decoder. This has the effect of smoothing the packet flow, increasing the resiliency to delayed packets, and other transmission effects. The buffer size is configurable, and can be optimized for given network conditions. It is practical to set 20–100 ms for each direction.

4. *Real-time transport protocol (RTP)*: The RTP is designed for carrying the data that has the real-time properties such as voice or video. The RTP header contains the time stamp, payload type, and sequence information that are useful for the receiving side to reconstruct the data.

5. *User datagram protocol (UDP)*: UDP provides efficient but unreliable data transport real-time voice data because retransmission of real-time voice data would add too much delay.

6. *Internet protocol (IP)*: provides a standard encapsulation for data transmission over the network. It contains a source and destination addresses used for routing.

Example 11.5: The trace given in Table 11.9 shows a frame of RTP data for the ITU G.729 encapsulated in UDP, IP, and Ethernet protocols. In the IP layer, the 20-byte header includes the source (88.88.88.1) and destination (88.88.88.8) IP addresses which are used for routing the packets. In the UDP layer, the 8-byte header provides the payload length and port information. In the RTP layer, the 12-byte header provides the payload type, time stamp, and sequence number. Finally, the payload follows 12-byte RTP header.

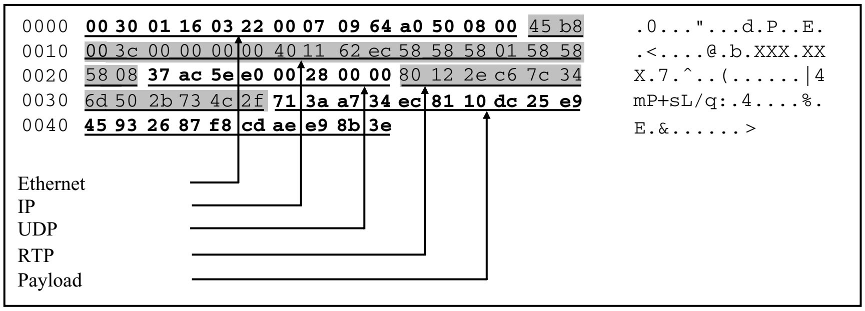
Figure 11.12 shows the complete data in hex format over the IP network. In this example, there is 54-byte overhead for each 20-ms G.729 encoded data, i.e., 20 bytes. Since the number of overhead is fixed, it is efficient to pack more payloads in one packet but the delay will be increased. Usually, the payload will be from 20 to 80 ms.

11.4.2 Real-Time Transport Protocol and Payload Type

VoIP data packets contained in RTP packets are encapsulated in UDP packets. RTP provides the solution enabling the receiver to put the packets back into the correct order, and not waiting too long for packets that either have lost or are taking too long to arrive. The format of RTP header is given in Figure 11.13.

Table 11.9 Example of G.729 payload

```
Frame 271 (74 bytes on wire, 74 bytes captured)
    Arrival Time: Dec 3, 2004 11:27:30.243551000
    Time delta from previous packet: 0.001989000 seconds
    Time since reference or first frame: 6.417555000 seconds
    Frame Number: 271
    Packet Length: 74 bytes
    Capture Length: 74 bytes
Ethernet II, Src: 00:07:09:64:a0:50, Dst: 00:30:01:16:03:22
    Destination: 00:30:01:16:03:22 (88.88.88.8)
    Source: 00:07:09:64:a0:50 (88.88.88.1)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 88.88.88.1 (88.88.88.1), Dst Addr:
    Version: 4
    Header length: 20 bytes
    Diff. Services Field: 0xb8 (DSCP 0x2e: Expedited FW;
ECN: 0x00)
        1011 10.. = Differentiated Services Codepoint: Expedited Forwarding
(0x2e)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 60
    Identification: 0x0000 (0)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: UDP (0x11)
    Header checksum: 0x1940
    Source: 88.88.88.1 (88.88.88.1)
    Destination: 88.88.88.8 (88.88.88.8)
User Datagram Protocol, Src Port: 14252 (14252), Dst Port: 24288 (24288)
    Source port: 14252 (14252)
    Destination port: 24288 (24288)
    Length: 40
    Checksum: 0x0000 (none)
Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    .001 0010 = Payload type: ITU-T G.729 (18)
    Sequence number: 11974
    Timestamp: 2083810640
    Synchronization Source identifier: 728976431
    Payload: 713AA734EC8110DC25E945932687F8CD
```



```
0000  00 30 01 16 03 22 00 07 09 64 a0 50 08 00 45 b8    .0..."...d.P..E.
0010  00 3c 00 00 00 00 40 11 62 ec 58 58 58 01 58 58    .<....@.b.XXX.XX
0020  58 08 37 ac 5e e0 00 28 00 00 80 12 2e c6 7c 34    X.7.^..(......|4
0030  6d 50 2b 73 4c 2f 71 3a a7 34 ec 81 10 dc 25 e9    mP+sL/q:.4....%.
0040  45 93 26 87 f8 cd ae e9 8b 3e                      E.&......>
```

Figure 11.12 Graphical explanations of IP/UDP/RTP packet data

The first 12 octets are present in every RTP packet, while the list of CSRC identifiers will present only when a mixer is inserted. Refer to RFC1889 for detailed definition of the field. The notations given in Figure 11.13 are defined as follows:

- V – Version of RTP used.
- P – Padding, a byte not used at bottom packet to reach the parity packet dimension.
- X – Presence of the header extension.
- CC – Number of CSRC identifiers following the fixed header. CSRC fields are used, for example, in conferencing case.
- M – Marker bit.
- PT – Payload type.

The payload type identifies the format of the RTP payload and determines its interpretation by applications. A profile may specify a default payload type codes. A set of default mappings for audio and video is specified in the RFC 3551. Some commonly used CODEC payload types are listed in Table 11.10.

GSM-AMR, G.729E, and G.726 use dynamic payload type while some of the early CODECs use the static payload type. The dynamic payload type uses the number between 96 and 127. For dynamic payload type, the payload type needs to be negotiated between two terminals.

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1st byte								2nd byte								3rd byte								4th byte							
V = 2		P	X	CC				M	PT							Sequence number															
Timestamp																															
Synchronization source (SSRC) identifier																															
Contributing source (CSRC) identifiers																															

Figure 11.13 Fixed-header fields (adapted from RFC1889)

Table 11.10 Partial payload types for audio encoding (adapted from RFC 3551)

Payload type	Encoding name	Comments
0	PCMU	G.711 PCM μ-law
3	GSM	GSM
4	G723.1	G.723.1 dual rate
8	PCMA	G.711 PCM A-law
13	CN	G.711 Annex II comfort noise
15	G728	G.728 at 16 kbps
18	G729	G.729 at 8 kbps
96–127	Dynamic	Dynamic payload type

11.4.3 Example of Packing G.729

Each G.729 and G.729A frame consists of 80 bits. The mapping of these parameters is given in Table 11.11 and Figure 11.14. The bits of each 32-bit word are numbered from 0 to 31, with the most significant bit on the left and numbered 0. The octets (bytes) of each word are transmitted with the most significant octet first.

A G.729 RTP packet may consist of zeros, or G.729 or G.729 Annex A frames, followed by zeros, or one G.729 Annex B frame. The presence of a comfort-noise frame can be deduced from the length of the RTP payload. The default packetization interval is 20 ms (two frames), but in some situations it may be desirable to send 10-ms packets. An example would be a transition from speech to comfort noise in the first 10 ms of the packet. For some applications, a longer packetization interval may be required to reduce the packet rate. Figure 11.14 shows the packet format of G.729 at 8 kbps, and Table 11.11 shows the description of parameters. The bit-stream ordering is reflected by the order in the Table 11.11. For each parameter, the MSB is transmitted first.

11.4.4 RTP Data Analysis Using Ethereal Trace

There are many commercially available equipment and software packages that are capable of analyzing the IP data. Ethereal is widely used software tool to capture the IP data for production use. Ethereal supports 620 protocols and the software can be downloaded from Internet. IP data can be captured from a live network connection or read from a captured file.

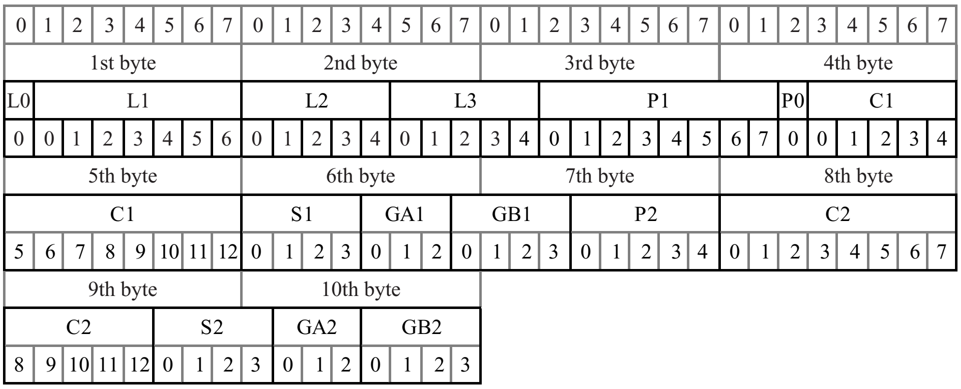


Figure 11.14 G.729 and G.729A bit packing

Table 11.11 Description of transmitted parameters indices

Symbol	Description	Bits
$L0$	Switched MA predictor of LSP quantizer	1
$L1$	First stage vector of quantizer	7
$L2$	Second stage lower vector of LSP quantizer	5
$L3$	Second stage higher vector of LSP quantizer	5
$P1$	Pitch delay first subframe	8
$P0$	Parity bit for pitch delay	1
$C1$	Fixed codebook first subframe	13
$S1$	Signs of fixed-codebook pulses first subframe	4
$GA1$	Gain codebook (stage 1) first subframe	3
$GB1$	Gain codebook (stage 2) first subframe	4
$P2$	Pitch delay second subframe	5
$C2$	Fixed codebook second subframe	13
$S2$	Signs of fixed-codebook pulses second subframe	4
$GA2$	Gain codebook (stage 1) second subframe	3
$GB2$	Gain codebook (stage 2) second subframe	4

An example of analyzing the RTP trace that uses `rtp` as the filter to display all RTP packets is shown in Figure 11.15. In this example, G.729 is the voice CODEC. The RTP data transfers back and forth between two IP terminals.

A very useful feature of this tool is performing statistic analysis using Statistic→RTP→Stream Analysis. The results shown in Figure 11.16 can help to understand the data format on the network side, investigate the possible cause of errors during transmission, and verify the interoperability with third-party equipments. The RTP data can also be saved using **Save Payload** for off-line processing.

11.4.5 Factors Affecting the Overall Voice Quality

The network delay, packet losses, packet jitters, and echoes are the major contributors to the perceived quality of VoIP. The ITU G.114 standard states that the end-to-end one-way delay should not exceed 150 ms. The overall voice packet delay can be improved with prioritizing the voice in the network packet. The speech-coding algorithms can also compensate these factors. Furthermore, efficient packet-loss concealment algorithms make the lost or discarded packet effects less noticeable.

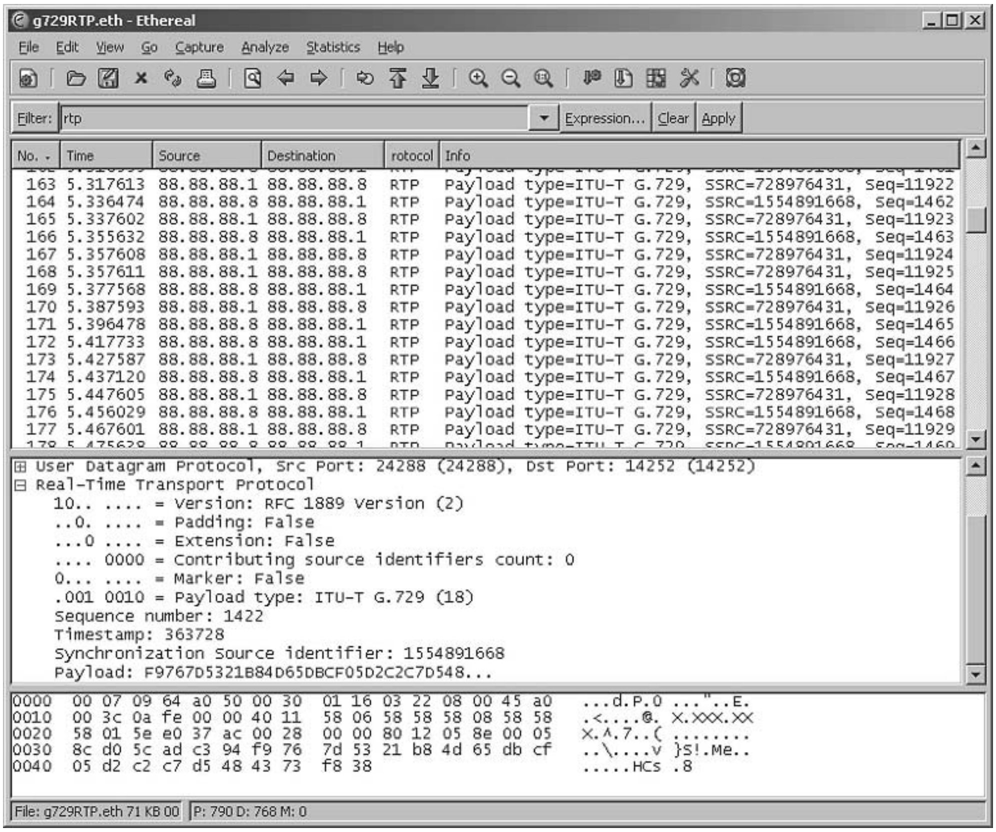
The delay can also be shortened by choosing low-delay speech CODECs which use small buffer size for block processing. A good jitter buffer algorithm can effectively compensate the jitter with the minimum buffer size to make the overall delay minimized. Network echoes can be effectively canceled or suppressed using adaptive echo cancelation algorithms introduced in Chapter 10.

11.5 Experiments and Program Examples

In this section, we will calculate the LPC coefficients using MATLAB, C, and C55x programs.

11.5.1 Calculating LPC Coefficients Using Floating-Point C

In this experiment, we implement the Levinson–Durbin algorithm using floating-point C to calculate the LPC coefficients. The main C program to access these functions is listed in Table 11.12, where three functions `calc_autoc( )`, `calc_lpc( )`, and `hmwindowing( )` are used to calculate the LPC

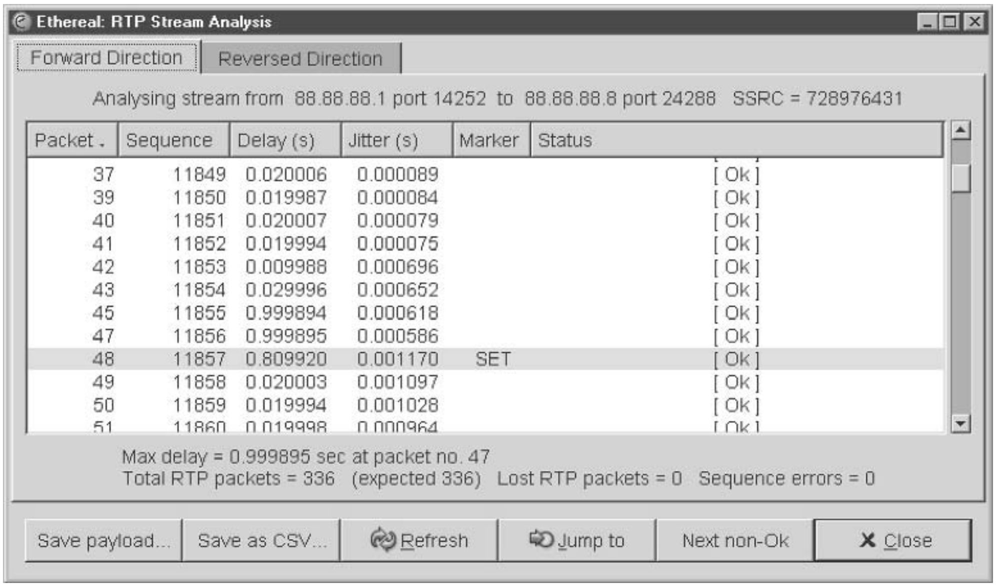

g729RTP.eth - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: rtp
Expression... Clear Apply
File: g729RTP.eth 71 KB 00
P: 790 D: 768 M: 0

```
No. -  Time      Source      Destination  rotocol  Info
  163 5.317613  88.88.88.1  88.88.88.8   RTP    Payload type=ITU-T G.729, SSRC=728976431, Seq=11922
  164 5.336474  88.88.88.8  88.88.88.1   RTP    Payload type=ITU-T G.729, SSRC=1554891668, Seq=1462
  165 5.337602  88.88.88.1  88.88.88.8   RTP    Payload type=ITU-T G.729, SSRC=728976431, Seq=11923
  166 5.355632  88.88.88.8  88.88.88.1   RTP    Payload type=ITU-T G.729, SSRC=1554891668, Seq=1463
  167 5.357608  88.88.88.1  88.88.88.8   RTP    Payload type=ITU-T G.729, SSRC=728976431, Seq=11924
  168 5.357611  88.88.88.1  88.88.88.8   RTP    Payload type=ITU-T G.729, SSRC=728976431, Seq=11925
  169 5.377568  88.88.88.8  88.88.88.1   RTP    Payload type=ITU-T G.729, SSRC=1554891668, Seq=1464
  170 5.387593  88.88.88.1  88.88.88.8   RTP    Payload type=ITU-T G.729, SSRC=728976431, Seq=11926
  171 5.396478  88.88.88.8  88.88.88.1   RTP    Payload type=ITU-T G.729, SSRC=1554891668, Seq=1465
  172 5.417733  88.88.88.8  88.88.88.1   RTP    Payload type=ITU-T G.729, SSRC=1554891668, Seq=1466
  173 5.427587  88.88.88.1  88.88.88.8   RTP    Payload type=ITU-T G.729, SSRC=728976431, Seq=11927
  174 5.437120  88.88.88.8  88.88.88.1   RTP    Payload type=ITU-T G.729, SSRC=1554891668, Seq=1467
  175 5.447605  88.88.88.1  88.88.88.8   RTP    Payload type=ITU-T G.729, SSRC=728976431, Seq=11928
  176 5.456029  88.88.88.8  88.88.88.1   RTP    Payload type=ITU-T G.729, SSRC=1554891668, Seq=1468
  177 5.467601  88.88.88.1  88.88.88.8   RTP    Payload type=ITU-T G.729, SSRC=728976431, Seq=11929
```

```
⊞ User Datagram Protocol, Src Port: 24288 (24288), Dst Port: 14252 (14252)
⊟ Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    .001 0010 = Payload type: ITU-T G.729 (18)
    Sequence number: 1422
    Timestamp: 363728
    Synchronization Source identifier: 1554891668
    Payload: F9767D5321B84D65DBCF05D2C2C7D548...
```

```
0000  00 07 09 64 a0 50 00 30  01 16 03 22 08 00 45 a0   ...d.P.0 ..."..E.
0010  00 3c 0a fe 00 00 40 11  58 06 58 58 58 08 58 58   .<....@. X.XXX.XX
0020  58 01 5e e0 37 ac 00 28  00 00 80 12 05 8e 00 05   X.^.7..( ........
0030  8c d0 5c ad c3 94 f9 76  7d 53 21 b8 4d 65 db cf   ..\....v }S!.Me..
0040  05 d2 c2 c7 d5 48 43 73  f8 38                     .....HCs .8
```

Figure 11.15 Example of RTP data between two terminals. It shows all the components of IP/UDP/RTP/payload



Analysing stream from 88.88.88.1 port 14252 to 88.88.88.8 port 24288 SSRC = 728976431

Packet .	Sequence	Delay (s)	Jitter (s)	Marker	Status
37	11849	0.020006	0.000089		[Ok]
39	11850	0.019987	0.000084		[Ok]
40	11851	0.020007	0.000079		[Ok]
41	11852	0.019994	0.000075		[Ok]
42	11853	0.009988	0.000696		[Ok]
43	11854	0.029996	0.000652		[Ok]
45	11855	0.999894	0.000618		[Ok]
47	11856	0.999895	0.000586		[Ok]
48	11857	0.809920	0.001170	SET	[Ok]
49	11858	0.020003	0.001097		[Ok]
50	11859	0.019994	0.001028		[Ok]
51	11860	0.019998	0.000964		[Ok]

Max delay = 0.999895 sec at packet no. 47
Total RTP packets = 336 (expected 336) Lost RTP packets = 0 Sequence errors = 0

Figure 11.16 Statistic of RTP data. It shows there are 336 RTP packets without packet loss and sequence errors

Table 11.12 Main program `lpc_mainTest.c` for computing LPC coefficients

```
while (fread(input,sizeof(short),N,fpin)==N)
{
  // Apply the Hamming window
  hmwindowing(N,input,ws);
  // Autocorrelation
  calc_autoc(ws, p_order, N,R);
  // Levension_Durbin
  calc_lpc( R, lpc, p_order);
}
```

coefficients. Function `calc_lpc( )` is listed in Table 11.13. The files used for this experiment are listed in Table 11.14.

Procedures of the experiment are listed as follows:

1. Start CCS, open the project, build, and load the program.

2. Edit the experiment parameter file `param.txt` located in the data directory, include input speech file, output coefficients file, LPC order, and frame size if necessary. The default setting uses the speech file `voice4.pcm`, 10th-order LPC, and frame size 180.

3. Modify the experiment test program such that the envelope of LPC coefficients will be plotted by CCS and displayed. The envelope will look similar to Figure 11.2.

4. Repeat the experiment with different speech files, such as male and female speakers, voiced and unvoiced segments.

11.5.2 Calculating LPC Coefficients Using C55x Intrinsics

A special library such as the ETSI (Europe Telecommunication Standard Institute) library can be used to represent the fixed-point operations for the CODECs including the G.723.1, G.729, and AMR. By using the basic operators defined in the ETSI library, the floating-point C code can be converted to fixed-point C code.

The floating-point C code must be modified for implementation on the fixed-point C55x. The input signal is normalized to maximally use the limited dynamic range and to avoid overflow. A vector normalization example is shown in Table 11.15. In the code, functions `round`, `norm_l`, `L_add`, `L_sub`, `L_shl`, and `L_shr` are used to simulate the processor operations of rounding, 16-bit normalization, 32-bit addition, subtraction, and 32-bit left and right shifts. After normalization, the Levinson algorithm is used to compute LPC coefficients. An example of calculating analysis filter coefficients is shown in Table 11.15.

The fixed-point C libraries do not have the efficiency that a DSP processor requires for real-time processing. Using the C55x intrinsics to replace functions in the fixed-point C libraries, we can achieve much better run-time efficiency. The results from fixed-point C implementation and intrinsics implementation are bit exact. This ensures the CODEC implementations in both fixed-point C and C55 intrinsics to have the same performance.

In Table 11.15, the operator `round(x)` converts a 32-bit data to 16 bits with rounding of the higher 16-bit word. Operator `norm_l(x)` calculates the leading sign bit. Using the same approach, the Levinson–Durbin algorithm can be converted as listed in Table 11.16. In the program, the LPC coefficients `a[ ]` and the reflection coefficients `K[ ]` are represented in Q.13 format to work with the LPC coefficients between -4 (0x8000/0x2000) and $+3\frac{8191}{8192}$ (0x7fff/0x2000). The prediction error `E[ ]` and correlation `R[ ]` are in Q.14 format to avoid overflow. The files used for this experiment are listed in Table 11.17.

Table 11.13 Function used for LPC coefficient computation

```
/*
|  calc_lpc()                          : lpc coefficients
|   Input         autoc                : autoc[0,.,frame_size-1]
|                 p                    : lpc order
|   Output        lpc                  : lpc[0,.,p_order]
*/
float *K;                                // Reflection coefficient
float *E;                                // Prediction error
float *a;                                // Intermediate lpc coefficients
static short first = 0;

void calc_lpc(float *autoc, float *lpc, short p)
{
    short i,j,p1;
    float acc0;
    float *R;                           // Correlation
    p1 = p+1;
    if (first == 0)
    {
        K = (float *)malloc((p1)*sizeof(float));
        a = (float *)malloc((p1)*(p1)*sizeof(float));
        E = (float *)malloc((p1)*sizeof(float));
        first++;
    }
    R = autoc;
    E[0] = R[0];                        // Equation (11.4)
    if (fabs(E[0]) < D16_MIN)           // To avoid divided by 0
        E[0] = D16_MIN;
    K[1] = R[1] / E[0];                 // Equation (11.5)
    a[1*p1+1]=K[1];                     // Equation (11.6)
    E[1]=(1-K[1]*K[1]) * E[0];          // Equation (11.8)
    for (i=2;i<=p;i++)
    {
        acc0=0.0;
        for (j=1;j<i;j++)
             acc0 += a[j*p1+i-1] * R[i-j];// partial Equation (11.5)
        K[i] = (R[i]-acc0) / E[i-1];// Equation (11.5)
        a[i*p1+i] = K[i];               // Equation (11.6)
        for (j=1;j<i;j++)
             a[j*p1+i] = a[j*p1+(i-1)] - K[i] * a[(i-j) * p1+(i-1)];
                                        // (Equation 11.7)
        E[i]=(1 - K[i] * K[i]) * E[i-1];// (Equation 11.8)
        if (fabs(E[i]) < D16_MIN)       // For division
            E[i] = signof(E[i]) * D16_MIN;
    }
    for (j=1;j<=p;j++)                  // Equation (11.9)
        lpc[j] = -a[j*p1+p];            // Same format as MATLAB
}
```

Table 11.14 File listing for experiment `exp11.5.1_floatingPointLpc`

Files	Description
`lpc_mainTest.c`	Main program for testing experiment
`lpc_auto.c`	C function computes autocorrelation
`lpc_lpc.c`	LPC analysis function
`lpc_hamming.c`	Hamming window function
`lpc_hamTable.c`	Generate Hamming window lookup table
`lpc.h`	C Header file
`floatPoint_lpc.pjt`	DSP project file
`floatPoint_lpc.cmd`	DSP linker command file
`param.txt`	Configuration file
`voice4.pcm`	Speech file

Table 11.15 Using intrinsics for calculating autocorrelation coefficients

```
/*
|  calc_autoc()              : autocorrelation
|   Input      ws           : ws[0,.,frame_size-1]
|              p_order      : lpc order
|              frame_size   : frame_size
|   Output     autoc        : autoc[0,.,p_order]
*/
void calc_autoc(short *ws, short p_order, short frame_size, short *autoc)
{
    short k,m,i,Exp;
    long acc0,acc1;
    /* Compute autoc[0] */
    acc1 = (long) 0 ;
    for ( i = 0 ; i < frame_size ; i++ )
    {
        acc0 = (long)ws[i]*ws[i];
        acc1 = L_add(acc1,acc0);
    }
    /* Normalize the energy */
    Exp = norm_l( acc1 ) ;
    acc1 = L_shl( acc1, Exp ) ;
    autoc[0] = round( acc1 );
    /* Compute autoc[i], i=1,..10 */
    for ( k = 1 ; k <= p_order ; k ++ ) {
        acc1 = (long) 0 ;
        for ( m = k ; m < frame_size ; m ++ )
        {
             acc0 = (long)ws[m]*ws[m-k];
             acc1 = L_add(acc1,acc0);
        }
        acc0 = L_shl( acc1, Exp );
        autoc[k] = round( acc0 );
}
```

Procedures of the experiment are listed as follows:

1. Start CCS, open the project, build, and load the program.

2. Check the experiment results and convert the fixed-point results to floating-point representation (scaling down by 8192) to evaluate the differences.

Table 11.16 Using intrinsics for calculating synthesis filter coefficients

```
/*
|  calc_lpc()                    : lpc coefficients
|   Input     autoc              : autoc[0,...,p_order]
|             p                  : lpc order
|   Output    lpc                : lpc[0,...,p_order]
*/
short K[LPCORDER+1];                // Reflection coefficient in Q.13
short a[(LPCORDER+1)*(LPCORDER+1)]; // LPC coefficients in Q.13
short E[LPCORDER+1];                // Prediction error in Q.14
void calc_lpc(short *autoc, short *lpc, short p)
{
    short i,j,p1;
    long acc0,acc1;
    short *R;                       // Correlation in Q.14
    short sign;
    p1= p+1;
    // Calculate LPC parameters
    R = autoc;
    E[0] = R[0];
    acc0 = L_shl(R[1],13);
    sign = signof(&acc0);
    acc0 = L_shl(acc0,1);
        K[1] = div_l(acc0, E[0]);
    if(sign== (short)-1)
    K[1] = negate(K[1]);
    a[1*p1+1]=K[1];
    /*E[1]=((8192-((K[1]*K[1])>>13)) * E[0])>>13;*/
    acc0 = (long)K[1] * K[1];
    acc0 = L_shr(acc0,13);
    acc0 = L_sub(8192, acc0);
    acc0 = E[0]* (short)(acc0);
    acc0 = L_shr(acc0,13);
    E[1] = (short) acc0;
    for (i=2;i<=p;i++)
    {
        acc0=0;
        for (j=1;j<i;j++)
        {
             acc1 = (long)a[j*p1+i-1] * R[i-j];
             acc0 = L_add(acc0,acc1);
        }
        acc1 = L_shl(R[i],13);
        acc0 = L_sub(acc1, acc0);
        sign = signof(&acc0);
```

Table 11.16 (*continued*)

```
        if (acc0 > L_shl(E[i-1],13))
            break;
        acc0 =L_shl(acc0,1);
        K[i] = div_l(acc0,E[i-1]);
        if(sign == (short)-1) K[i] = negate(K[i]);
        a[i*p1+i] = K[i];
        /* a[j*p1+i]=a[j*p1+(i-1)]-((K[i]*a[(i-j)*p1+(i-1)])>>13);*/
        for (j=1;j<i;j++)
        {
             acc0 = (long)K[i] * a[(i-j)*p1+(i-1)];
             acc1 = L_shl((int)a[j*p1+(i-1)],13);
             acc0 = L_sub(acc1,acc0);
             acc0 = L_shr(acc0,13);
             a[j*p1+i] = (short) acc0;
        }
        /* E[i]=((8192-((K[i]*K[i])>>13)) * E[i-1])>>13;*/
        acc0 = (long)K[i] * K[i];
        acc0 = L_shr(acc0,13);
        acc0 = L_sub((short)8192,acc0);
        acc0 = E[i-1]* (short)(acc0);
        acc0 = L_shr(acc0,13);
        E[i] = (short) acc0;
    }
    for (j=1;j<=p;j++)
        lpc[j] = negate(a[j*p1+p]);        // Same format as in MATLAB
}
```

3. Read `lpc` data from data memory and use MATLAB program (Example 11.2) to plot spectrum response of LPC synthesis filter against the input signal. We should see the LPC spectrum representing the envelope of the signal spectrum similar to Figure 11.2.

4. Modify the experiment to allow the test program accepting different parameters such as LPC order and input files.

Table 11.17 File listing for experiment `exp11.5.2_intrinsicLpc`

Files	Description
`intrinsic_lpc_mainTest.c`	Main program for testing experiment
`intrinsic_lpc_lpc.c`	Fixed-point function computes LPC coefficients
`intrinsic_lpc_auto.c`	Fixed-point function computes autocorrelation
`intrinsic_lpc_hamming.c`	Hamming window function using intrinsics
`intrinsic_lpc_hamTable.c`	Function generates Hamming window lookup table
`lpc.h`	C header file for the experiment
`gsm.h`	ETSI intrinsics header file provided by CCS
`linkage.h`	Header file used in conjunction with `gsm.h`
`intrinsic_lpc.pjt`	C55x project file
`intrinsic_lpc.cmd`	C55x linker command file
`voice4.pcm`	Speech file

5. Repeat the experiment with different experiment speech files, such as male and female, voiced and unvoiced segments.

6. Use scaling variables such as `E` and `K` rather than arrays `E[11]` and `K[11]` to reduce memory usage. Pay attention to the array `a[ ]` in floating- and fixed-point C, where the array size has been reduced from `p1*p1` in floating-point C to `p1` in fixed-point C. Rewrite the LPC code to use scaling variables `E` and `K`.

11.5.3 MATLAB Implementation of Formant Perceptual Weighting Filter

In Example 11.2, a MATLAB function is used to compute the LPC coefficients as `a = levinson(r,p)`. Using the LPC coefficients for synthesis filter $1/A(z)$, the weighting filter $W(z)$ can be calculated using Equation (11.11). The speech file `voice4.pcm` is used for the experiment, and the magnitude response of the perceptual weighting filter is plotted. The MATLAB script is listed in Table 11.18. The files used for the experiment is listed in Table 11.19.

Procedures of the experiment are listed as follows:

1. Start MATLAB and set the path to the directory `../exp11.5.3_matPwf`.

2. Run the experiment and examine the frequency responses of the synthesis filter and perceptual weighting filter (refer to Figure 11.17).

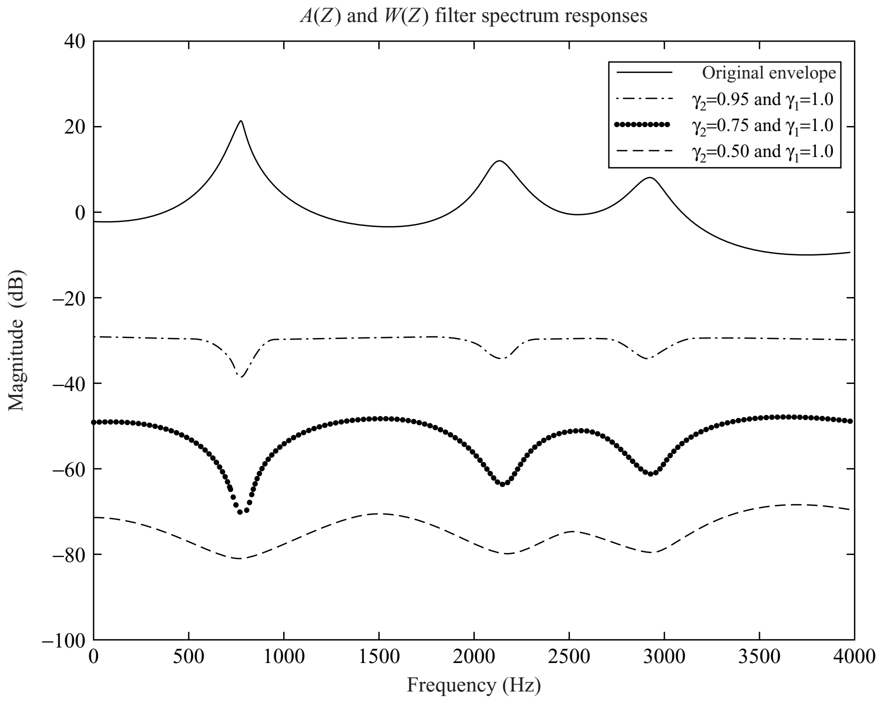


Figure 11.17 Spectrum of speech and its spectral envelopes defined by synthesis filters

Table 11.18 MATLAB code to calculate $W(z)$ and plot its frequency response

```
gama_1 = 0.95;
gama_2 = 0.70;
gama_3 = 0.50;

a=levinson(r,lpcOrder);           % Levinson
y=fft(a,fftL);
pyy=-10*log10(y.*conj(y));

a_1 = a*gama_1;
m=0;
while (m<=lpcOrder);
  a_1(m+1)=a(m+1)*(gama_1^(m-1));
  m=m+1;
end;
y=fft(a_1,fftL);
pyy_1=-30-pyy-10*log10(y.*conj(y)); % Offset 20 dB for display

a_2 = a*gama_2;
m=0;
while (m<=lpcOrder);
  a_2(m+1)=a(m+1)*(gama_2^(m-1));
  m=m+1;
end;
y=fft(a_2,fftL);
pyy_2=-50-pyy-10*log10(y.*conj(y)); % Offset 20 dB for display

a_3 = a*gama_2;
m=0;
while (m<=lpcOrder);
  a_3(m+1)=a(m+1)*(gama_3^(m-1));
  m=m+1;
end;
y=fft(a_3,fftL);
pyy_3=-70-pyy-10*log10(y.*conj(y)); % Offset 20 dB
plot(f,pyy(1:(fftL/2)),'-',f,pyy_1(1:(fftL/2)),'-.',
     f,pyy_2(1:(fftL/2)),'--.',f,pyy_3(1:(fftL/2)),'--');
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
title('A(Z) and W(Z) filter spectrum responses');
h = legend('Original envelop',' \gamma2=0.95&\gamma1=1.0',
    '\gamma2=0.75&\gamma1=1.0','\gamma2=0.50&\gamma1=1.0',4);
```

3. Change the parameters γ_2 to other values and observe the differences of weights.

4. Modify the MATLAB script to read different segment of the speech data from the speech file `voice4.pcm` and repeat the experiment. Observe the differences using different speech segments.

5. Change the order of LPC filter and repeat the experiment. Observe the changes with different LPC filter orders.

6. Use different frame sizes to calculate LPC coefficients and observe differences.

Table 11.19 File listing for experiment `exp11.5.3_matPwf`

Files	Description
`exp11_5_3_pwf.m`	MATLAB perceptive weighting filter experiment
`voice4.pcm`	Speech file

11.5.4 Implementation of Perceptual Weighting Filter Using C55x Intrinsics

For many speech-coding algorithms, perceptual weighting filter coefficients are calculated using LPC coefficients. In addition to all files used by experiment given in Section 11.5.2, the fixed-point C file `intrinsic_pwf_wz.c` listed in Table 11.20 is used to calculate the weighting filter coefficients for this experiment. In the program, `gamma1` is γ_1 and `gamma2` is γ_2 given in Equation (11.11). The files used for this experiment are listed in Table 11.21.

Procedures of the experiment are listed as follows:

1. Start CCS, open the project, build, and load the program.
2. Check the experiment results. Plot the weighting filter frequency response to verify the LPC envelope. It should resemble Figure 11.17.

Table 11.20 Computation of perceptual weighting filter coefficients, `intrinsic_pwf_wz.c`

```
/*
|    calc_wz()                  : Perceptual weighting filter
|                                 W(Z)=(wf1[z])/(wf2[z])
|    Input        lpc           : lpc[0,.,p_order]
|                 gamma1        : gamma1
|                 gamma2        : gamma2
|                 p_order       : lpc order
|    Output       wf1           : wf1[0,.,p_order]
|                 wf2           : wf2[0,.,p_order]
*/
void calc_wz(short *lpc, short gamma1,short gamma2, short p_order, short
*wf1, short *wf2)
{
   short I,gam1,gam2;
   wf1[0]=32767;
   wf2[0]=32767;
   gam1 = gamma1;
   gam2 = gamma2;

   for (i=1; i<=p_order; i++)
   {
       wf1[i] = mult_r(lpc[i],gam1);
       wf2[i] = mult_r(lpc[i],gam2);
       gam1 = mult_r(gam1, gamma1);
       gam2 = mult_r(gam1, gamma2);
   }
}
```

Table 11.21 File listing for experiment `exp11.5.4_intrinsicPwf`

Files	Description
`intrinsic_pwf_mainTest.c`	Main program for testing experiment
`intrinsic_pwf_lpc.c`	Fixed-point function computes LPC coefficients
`intrinsic_pwf_auto.c`	Fixed-point function computes autocorrelation
`intrinsic_pwf_hamming.c`	Hamming window function using intrinsics
`intrinsic_pwf_hamTable.c`	Function generates Hamming window lookup table
`intrinsic_pwf_wz.c`	Function calculates perceptual weighting filter coefficients
`pwf.h`	C header file
`gsm.h`	ETSI intrinsics header file provided by CCS
`linkage.h`	Header file used in conjunction with `gsm.h`
`intrinsic_pwf.pjt`	C55x project file
`intrinsic_pwf.cmd`	C55x linker command file
`voice4.pcm`	Speech file

References

[1] ITU-T Recommendation G.723.1, *Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 & 6.3 kbit/s*, Mar. 1996.

[2] CCITT Recommendation G.728, *Coding of Speech at 16 kbit/s Using Low-delay Code Excited Linear Prediction*, Geneva, 1992.

[3] ITU-T Recommendation G.729, *Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear Prediction* (*CS-ACELP*), Dec. 1995.

[4] MPEG-4 Working Draft, *CELP Coding*, ISO/IEC CD 14496-3 Subpart 3, ISO/JTC 1/SC 29/WG11, Oct. 1997.

[5] J. Natvig, 'Pan-European speech coding standard for digital mobile radio,' *Speech Communications*, vol. 7, 1988, pp. 113–123.

[6] 3G TS 26.190 V1.0.0 (2000–12), *Mandatory Speech CODEC Speech Processing Functions AMR Wideband Speech CODEC; Transcoding Functions* (Release 4), Dec. 2000.

[7] 3GPP TS 26.171, *Universal Mobile Telecommunications System* (*UMTS*)*; AMR Speech CODEC, Wideband; General Description* (Release 5), Mar. 2001.

[8] All G.729 related ITU-T standards including Annex A–I.

[9] A. M. Kondoz, *Digital Speech Coding for Low Bit Rate Communications Systems*, New York: John Wiley & Sons, Inc., 1995.

[10] W. E. Witowsky, 'IP telephone design and implementation issues,' Telogy white paper.

[11] W. Tian, *Narrow/Wide-Band Speech Coding Using CELP Model*, Ph.D. Dissertation, National University of Singapore, 1998.

[12] W. Tian, W. C. Wong, and C. Tsao, 'Low-delay subband CELP coding of wideband speech,' *IEEE Proc. Vision, Image Signal Process.*, vol. 144, pp. 313–316, Oct. 1997.

[13] W. Tian and W.C. Wong, 'Multi-pulse embedded coding of speech,' *Proc. IEEE APCC/ICCS'98*, pp. 107–111, Nov. 1998.

[14] W. Tian, W. C. Wong, C. Y. Law, and A. P. Tan, 'Pitch synchronous extended excitation in multi-mode CELP,' *IEEE Communications Letter*, vol. 03, pp. 275–276, Sep. 1999.

[15] W. Tian and W. C. Wong, '6 kbit/s partial joint optimization CELP,' *Proc. ICICS'99*, CDROM #1D1.1, Dec. 1999.

[16] W. Tian and A. Alvarez, 'Embedded coding of G.729x,' *Proc. ICICS'99*, CDROM #2D3.3, Dec. 1999.

[17] W. Tian, G. Hui, W. Ni, and D. Wang, 'Integration of LD-CELP codec and echo canceller,' *Proc. IEEE TENCON'93*, pp. 287–290, Oct. 1993.

[18] J. H. Chen and A. Gersho, 'Adaptive postfiltering for quality enhancement of coded speech,' *IEEE Trans. Speech Audio Process.*, vol. 3, pp. 59–71, Jan. 1995.

[19] P. E. Papamichalis, *Practical Approaches to Speech Coding*, Englewoods Cliffs, NJ: Prentice Hall, 1987.

[20] Math Works, Inc., *Using MATLAB*, Version 6, 2000.
[21] Texas Instruments, Inc., *A-Law and mu-Law Companding Implementations Using the TMS320C54x*, Literature no. SPRA163A, 1997.

Exercises

1. In Equation (11.11), let $\gamma_2 = 1$. Will the weighting filter still work? On the other hand, if we let $\gamma_1 = 1$, what will be the combined filter $H(z)$ in Equation (11.12)? You may modify the MATLAB script given in Section 11.5.3 to conduct this exercise.

2. For the ACELP representation, how many bits are needed in order to encode the eight possible positions? If this pulse amplitude is either $+1$ or -1, how many bits are needed in total to encode the pulse position and sign? If the pitch interval is from 20 to 147, how many bits are needed for encoding? If we need higher resolution, 1/2 sample, in the pitch range of 20–147, how many bits are needed?

3. The LSP coefficient vector in G.723.1 is a 10th-order vector. This vector is divided by three subvectors of dimensions 3, 3, and 4. If each subvector is vector equalized using an 8-bit codebook, how many bits are needed to represent the LSP quantization index?

4. If we use 16 bits to represent the LPC coefficients, and assume that the values of these coefficients are always between ± 3.00 and 0.0, what is the most efficient representation of the coefficients using the 16-bit fixed-point format?

5. For VoIP applications, assume that there are 60 bytes used for Ethernet/IP/UDP/RTP headers. If using ITU G.729 as the CODEC with the packet frame size of 20 ms, what is the actual bit rate over the IP network? If the voice activity is 40 % (means 60 % signal is silence), how much can be saved in bandwidth if we do not send anything during silence frames? If we increase the packet frame size from 20 to 40 ms, what is the actual bit rate over network during the active frames?

6. Given a RTP trace, if the 7-bit `Payload type` $= 0$, which CODEC has been used (refer to Table 11.10)? If the UDP payload length $= 100$ bytes, what is the frame size in samples for this CODEC? If we get a valid UDP payload length $= 21$ bytes, why the packet size is smaller? Assume that there are 12 bytes for RTP header and 8 bytes for UDP.

7. For the ACELP such as G.729 CODEC, if the first three pulse locations have been found to be at 5, 6, and 7, is it possible the fourth pulse is at 8? Is it possible the fourth one is at 9? Why?

12 Speech Enhancement Techniques

This chapter introduces the design and implementation of single-channel speech enhancement (or noise reduction) algorithms to enhance the speech corrupted by background noises. We will focus on the spectrum subtraction algorithm for reducing background noises.

12.1 Introduction to Noise Reduction Techniques

The use of cellular/wireless phones is often found in the noisy environments such as vehicles, restaurants, shopping malls, manufacturing plants, or airports. High background noises will degrade the quality or intelligibility of speech in these applications. Excessive noise level also will degrade the performance of existing signal processing techniques, such as speech coding, speech recognition, speaker identification, and adaptive echo cancelation, which are developed with the low-noise assumption. The purpose of many speech enhancement algorithms is to reduce noise, improve speech quality, or suppress undesired interference. The noise reduction (NR) becomes increasingly important to improve voice quality in noisy environments for hands-free applications.

There are three general classes of speech enhancement techniques: subtraction of interference, suppression of harmonic-related noises, and resynthesis using vocoders. Each technique has its own set of assumptions, advantages, and limitations. The first technique suppresses noise by subtracting the estimated noise spectrum, which will be discussed in Section 12.2. The second method employs fundamental frequency tracking using adaptive comb filter for reducing the periodic noises. The third technique focuses on estimating speech-modeling parameters using iterative methods, and uses these parameters to resynthesize the noise-free speech.

NR algorithms can be classified as single-channel or dual-channel (multiple-channel) techniques. In many real-world situations, only a single-channel system is available. A typical single-channel speech enhancement system is shown in Figure 12.1. The noisy speech $x(n)$ is the only available input signal for the system, which contains $s(n)$ from the speech source and $v(n)$ from the noise source. The output signal is the enhanced speech $\hat{s}(n)$. Under the assumption that the background noise is stationary, the characteristics of the noise can only be estimated during the silence periods between utterances.

This chapter concentrates on the single-channel speech enhancement systems. Since the systems estimate noise characteristics during the nonspeech periods, an accurate and robust voice activity detector (VAD) plays an important role in the performance of the system.

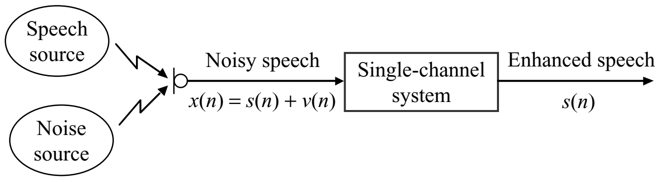


Figure 12.1 A single-channel speech enhancement system

Noise subtraction algorithms can be implemented in either time or frequency domain. The frequency-domain implementation based on short-time spectral amplitude estimation and subtraction is called the spectral subtraction. The basic idea of the spectral subtraction algorithm is to obtain the short-term magnitude spectrum of the noisy speech using the fast Fourier transform (FFT), subtract the estimated noise magnitude spectrum, and then to perform an inverse transforming on this subtracted spectral amplitude with the original phase.

Frequency-domain noise suppression can be implemented in time domain by decomposing the corrupted speech signal into overlapped frequency bands using a filterbank. The noise power of each subband is estimated during nonspeech periods. Noise suppression is achieved using the attenuation factors that correspond to the ratio of the temporal signal power to the estimated noise power. Since the spectral subtraction algorithm provides the basic concept of filterbank technique, it will be presented in details in the next section.

12.2 Spectral Subtraction Techniques

Spectral subtraction technique uses a computationally efficient FFT for reducing noise. As illustrated in Figure 12.2, this approach enhances the speech quality by subtracting the estimated noise spectrum from the noisy speech spectrum.

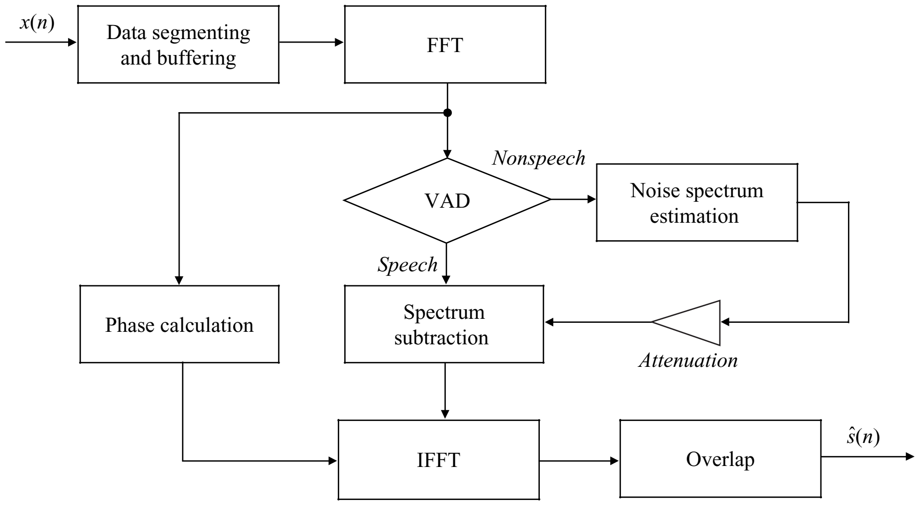


Figure 12.2 Block diagram of the spectral subtraction algorithm

12.2.1 Short-Time Spectrum Estimation

Assume that the noisy signal $x(n)$ consists of a speech $s(n)$ and an uncorrelated noise $v(n)$. This noisy speech is segmented and windowed. The FFT and magnitude spectrum are computed frame by frame. A VAD is used to determine if the current frame is the speech or nonspeech. For a speech frame, the algorithm performs the spectral subtraction to generate the enhanced speech signal $\hat{s}(n)$. During the nonspeech frames, the algorithm estimates noise spectrum and attenuates the signal in the buffer to reduce noise.

There are two methods for generating the output during nonspeech frames: (1) attenuate the output by a fixed scaling factor that is less than 1, and (2) set the output to zero. Experimental results show that having some residual noise during nonspeech frames will give a better subjective speech quality. This is because setting the output to zero has the effect of amplifying the noise during the speech frames. We must maintain the balance between the magnitude and the characteristics of the noise perceived during the speech and noise frames to avoid undesirable audio effects such as clicking, fluttering, or even slurring of the speech signal. A reasonable amount of attenuation is about 30 dB.

The input signal is segmented using the Hanning (or Hamming) window introduced in Section 4.2.3 to 50 % overlapped data buffers. After the noise subtraction, the enhanced speech waveform is reconstructed to time domain by the inverse FFT. These output frames are overlapped and added to produce the output signal.

Example 12.1: Given a frame of 256 samples, calculate the algorithm delay if 50 % overlap is used. Compare the computational load with the algorithms without using overlap.

If 50 % overlap is used, algorithm delay is 256 samples or 32 ms at 8 kHz sampling rate. The computational load will be double since the same block of data has been calculated twice.

12.2.2 Magnitude Subtraction

Several assumptions are made for the algorithm development. First, the algorithm assumes that the background noise is stationary such that the expected noise spectrum will not change during the following speech frames. If the environment changes, there will be enough time for the algorithm to estimate a new background noise spectrum before the presence of speech. Therefore, the algorithm must have an effective VAD to determine its operations. The algorithm also assumes that the NR can be achieved by removing the noise from the magnitude spectrum only.

If the speech $s(n)$ has been degraded by a zero-mean uncorrelated noise $v(n)$, the corrupted noisy signal can be expressed as

$$x(n) = s(n) + v(n). \tag{12.1}$$

Taking the discrete Fourier transform of $x(n)$ gives

$$X(k) = S(k) + V(k). \tag{12.2}$$

The estimate of $|S(k)|$ can be expressed as

$$|\hat{S}(k)| = |X(k)| - E|V(k)|, \tag{12.3}$$

where $E|V(k)|$ is the expected noise spectrum estimated during the nonspeech frames.

Given the $|\hat{S}(k)|$, the speech spectrum can be expressed as

$$\hat{S}(k) = |\hat{S}(k)|e^{j\theta_x(k)}, \tag{12.4}$$

where $\theta_x(k)$ is the phase of measured noisy signal and

$$e^{j\theta_x(k)} = \frac{X(k)}{|X(k)|}. \tag{12.5}$$

It assumes that the phase of noisy speech can be used for practical purposes. Therefore, we can reconstruct the enhanced speech with the short-term speech magnitude spectrum $|\hat{S}(k)|$ and the noisy phase $\theta_x(k)$.

Substituting Equations (12.3) and (12.5) into Equation (12.4), the speech estimate can be expressed as

$$\begin{aligned}\hat{S}(k) &= [|X(k)| - E|V(k)|]\frac{X(k)}{|X(k)|}\\ &= H(k)X(k),\end{aligned} \tag{12.6}$$

where

$$H(k) = 1 - \frac{E|V(k)|}{|X(k)|}. \tag{12.7}$$

Note that the spectral subtraction algorithm given in Equations (12.6) and (12.7) avoids the computation of phase $\theta_x(k)$, which is too complicated to implement on a DSP processor for real-time applications.

Example 12.2: In the derivation of spectral subtraction algorithm, identify which simplification has contributed to the distortion of the speech.

Using Equations (12.7) and (12.2), the estimation can be further decomposed as

$$\begin{aligned}\hat{S}(k) &= X(k)H(k) = [S(k) + V(k)]\left[1 - \frac{E\,|V(k)|}{|S(k) + V(k)|}\right]\\ &= S(k) + V(k) - E\,|V(k)|\,\frac{S(k) + V(k)}{|S(k) + V(k)|}\\ &= S(k) + V(k) - E\,|V(k)|\,e^{\theta_x(k)}.\end{aligned} \tag{12.8}$$

Here, $V(k) - E|V(k)|e^{\theta_x(k)} = 0$ results in a perfect cancelation. In practice, this is not realistic for the following reasons: (1) $V(k)$ is a temporal observation and the estimation of $V(k)$ by $E[V(k)]$ is very difficult in real-time applications; (2) the simplified algorithm uses $\theta_x(k)$ to represent $\theta_v(k)$ in order to avoid the heavy computation; (3) the VAD cannot be perfect, and thus it results in an inaccurate estimation of the noise spectrum.

Example 12.3: Read a frame of data from speech file `voice4.pcm` as the original speech signal $s(n)$. Add a white Gaussian noise $v(n)$ to $s(n)$ to form a noisy speech signal $x(n)$ as shown in Equation (12.1). The corresponding transformed signals are $S(k)$, $V(k)$, and $X(k)$. Calculate the angle difference between $S(k)$ and $X(k)$. The differences are in the range of $(-\pi, \pi)$. The original speech spectrum $S(k)$ and white noise spectrum $V(k)$ are illustrated in Figure 12.3 using the MATLAB script `example12_3.m`. The frame size is 256 and the white noise is generated using

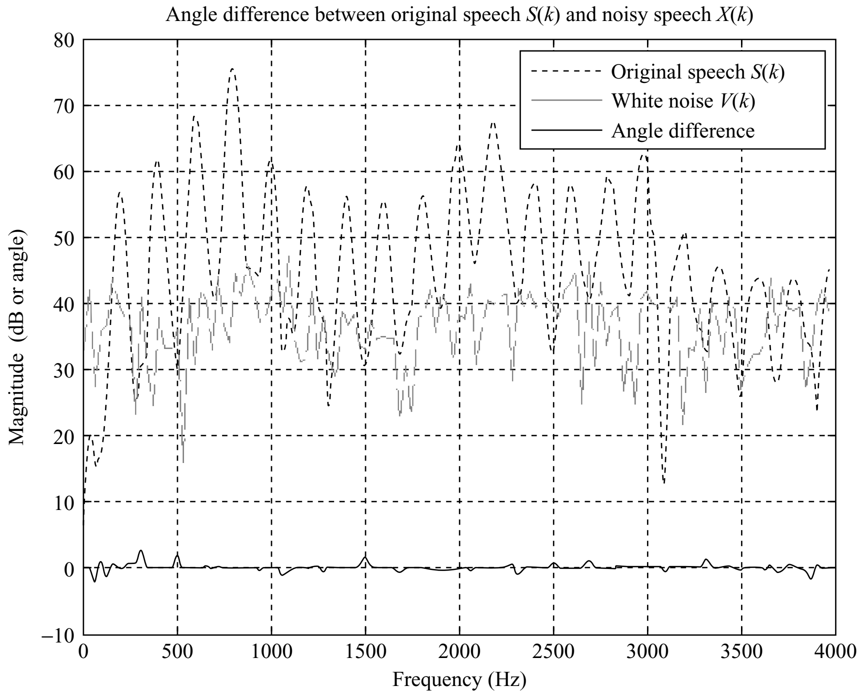


Figure 12.3 Angle difference between the original and noisy speeches

`v = wgn(frame,1,40)`. The noise level is relatively small compared with the speech $s(n)$ as shown in Figure 12.3.

The angle difference is large in the regions where the noise level is comparable or even higher than the speech. If the VAD detection is accurate, most of these large angle difference regions should be classified as silence. Example 12.3 indicates that the approximation of $\theta_s(k)$ by $\theta_x(k)$ is reasonable in the active bands.

In order to reduce music-tone effects, an oversubtraction factor may be applied to Equation (12.7) as

$$H(k) = 1 - \varepsilon \frac{E|V(k)|}{|X(k)|}, \tag{12.9}$$

where $\varepsilon \leq 1$. Using $\varepsilon < 1$ can relieve the effect of oversubtraction, but mildly decrease the signal-to-noise ratio of the processed signal.

12.3 Voice Activity Detection

VAD is one of the most important functions for spectral subtraction algorithms. The basic assumptions for a VAD algorithm are (1) the spectrum of the speech signal changes in short time, but background noise is relatively stationary and it changes slowly; and (2) the active speech level is usually higher than the background noise level. Several methods are used for VAD, such as voiced/unvoiced classification

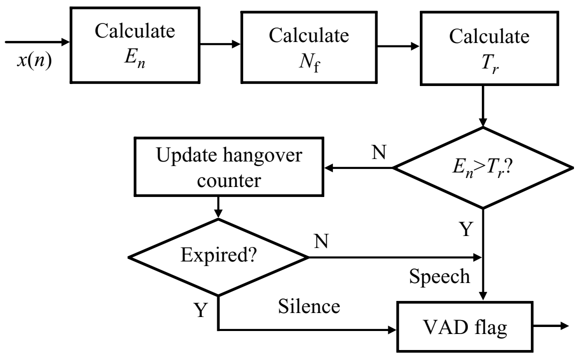


Figure 12.4 Block diagram of a simple VAD algorithm

used in ITU G.723.1, zero-crossing method used in G.729, and spectral comparison used in both G.729 and GSM vocoders in addition to different power thresholds validations. Some of them may be combined to offer a VAD with better performance.

This section introduces a generic form of VAD. In practical speech applications, the input signal is usually highpass filtered to remove the undesired low-frequency components. Assuming that $X(k)$ are the FFT bins of the input signal $x(n)$, the bins covering the frequencies from 300 to 1000 Hz (at 8 kHz sampling rate) are used to calculate the power with different window sizes as shown in Figure 12.4. The VAD algorithm is described as follows.

Calculate signal energy E_n as

$$E_n = \sum_{k=K1}^{K2} |X(k)|^2, \tag{12.10}$$

where $K1$ and $K2$ are the nearest integers of frequency indices (k) close to 300 and 1000 Hz, respectively. The energy of the signal in a short window is calculated as

$$E_s(j) = (1 - \alpha_s)E_s(j) + \alpha_s E_n, \tag{12.11}$$

and the long window signal energy is calculated as

$$E_l(j) = (1 - \alpha_l)E_l(j) + \alpha_l E_n, \tag{12.12}$$

where α_s and α_l are the window-length factor, and the subscript 's' represents the short window and 'l' represents long window, respectively. Usually, $\alpha_s = 1/16$ and $\alpha_l = 1/128$. The noise level at the nth frame, N_f, is updated on the basis of its previous value and the current E_n, and with different rates controlled by α_s or α_l.

The noise floor is calculate as

$$N_f = \begin{cases} (1 - \alpha_l)N_f + \alpha_l E_n, & \textit{if}\ N_f < E_s(j) \\ (1 - \alpha_s)N_f + \alpha_s E_n, & \textit{if}\ N_f \geq E_s(j) \end{cases}. \tag{12.13}$$

The threshold T_r used for signal energy comparison is calculated as

$$T_r = \frac{N_f}{1 - \alpha_l} + \text{margin}, \tag{12.14}$$

where the ‘margin’ is a reasonable value to avoid toggling between voice and silence if the noise level is flat. The current frame signal energy will be compared with this threshold and the decision will be made as follows:

$$\text{VAD Flag} = \begin{cases} 1, & \text{if } E_n > T_r \\ 1, & \text{if } E_n \le T \quad \text{and hangover not expired} \\ 0, & \text{if } E_n \le T \quad \text{and hangover expired} \end{cases}. \tag{12.15}$$

A hangover period is used for transitioning from active speech to silence in order to avoid false detection of the silence at the tail end of speech. During the tail period of the speech and before the hangover counter is expired, the signal frames are classified as active speech. The typical hangover time is about 90 ms.

If the noise level is changing and signal level is relatively low, this VAD may not response correctly, especially during the tail of speech segments. It may need the extra measurements such as zero-crossing rate and voiced segment detection. More details can be found in the G.723.1 and G.729 algorithms.

12.4 Implementation Considerations

Modifications must be made to the spectral subtraction algorithm illustrated in Figure 12.2 to reduce the auditory effects of spectral error. These modifications include spectral magnitude averaging, half-wave rectification, and residual NR.

12.4.1 Spectral Averaging

Since the spectral error is proportional to the difference between the noise spectrum and its mean, averaging of the magnitude spectra can be used to reduce the spectral error. The local average can be expressed as

$$|X(k)| = \frac{1}{M}\sum_{i=1}^{M} |X_i(k)|, \tag{12.16}$$

where $X_i(k)$ is the transform of ith frame of $x(n)$. A problem with this modification is that the speech signal is considered as short-term stationary for a maximum length of 30 ms. The average has the risk of some temporal smearing to short transitory sounds. From the experimental results, a reasonable compromise between variance reduction and time resolution appears to be an average of two to three frames.

12.4.2 Half-Wave Rectification

For each frequency bin where the magnitude spectrum $|X(k)|$ is less than the averaged noise magnitude spectrum $E|V(k)|$, the output is set to zero because the magnitude spectrum cannot be negative. This modification can be implemented by half-wave rectification of the spectral subtraction filter $H(k)$. Thus, Equation (12.6) becomes

$$\hat{S}(k) = \frac{H(k) + |H(k)|}{2} X(k). \tag{12.17}$$

The advantage of half-wave rectification is that it essentially eliminates low coherent tonal noise. The drawback occurs when the sum of the noise and speech is less than $E|V(k)|$ at frequency bin k. In this case, the speech information at that frequency bin is incorrectly removed, implying a possible decrease in intelligibility.

As mentioned earlier, a small amount of residual noise improved the output speech quality. This idea can be implemented by using a software constraint

$$|S(k)| \geq 0.02E\,|V(k)|\,, \tag{12.18}$$

where the minimum spectrum floor is –34 dB with respect to the estimated noise spectrum.

12.4.3 Residual Noise Reduction

For uncorrelated noise, the residual noise spectrum occurs randomly as narrowband magnitude spikes. This residual noise spectrum will have a magnitude between zero and a peak value measured during nonspeech periods. When these narrowband components are transformed back to the time domain, the residual noise will sound like the sum of tones with random fundamental frequency which is turned on and off at a rate of about 20 ms. During speech frames, the residual noise will also be perceived at frequencies that are not masked by the speech.

Since the residual noise will randomly fluctuate in amplitude in each frame, it can be suppressed by replacing its current value with its minimum value chosen from the adjacent frames. The minimum value is used only when $\left|\hat{S}(k)\right|$ is less than the maximum residual noise calculated during nonspeech periods. The reasons behind this scheme are:

1. If $|\hat{S}(k)|$ lies below the maximum residual noise and it varies radically from frame to frame, there is a high probability that the component at that frequency is due to noise. Therefore, it can be suppressed by taking the minimum value.

2. If $|\hat{S}(k)|$ lies below the maximum but has a nearly constant value, there is a high probability that the spectrum at that frequency is due to low-energy speech. Therefore, taking the minimum will retain the information.

3. If $|\hat{S}(k)|$ is greater than the maximum, the bias is sufficient. Thus, the estimated spectrum $|\hat{S}(k)|$ is used to reconstruct the output speech.

The disadvantages of this scheme are that more storage is required to save the maximum noise residuals and the magnitude values, and more computation is required to find the maximum and minimum values of spectra for three adjacent frames.

12.5 Combination of Acoustic Echo Cancelation with NR

There are many practical applications that combine the acoustic echo cancelation (AEC) introduced in Chapter 10 with the NR techniques introduced in this chapter as an integrated system. These applications include hands-free cell phones in noisy environments.

Assume that the vector $\mathbf{u}(n)$ consists of two scalar variables $x(n)$ and $y(n)$ as

$$\mathbf{u}(n) = \begin{bmatrix} x(n) \\ y(n) \end{bmatrix}. \tag{12.19}$$

If the estimator $\hat{s}(n)$ of $s(n)$ is a linear function of $\mathbf{u}(n)$, $S(z)$ is the z-transform of $s(n)$, and $\mathbf{U}(z)$ is the z-transform of $\mathbf{u}(n)$, the mean-square error (MSE) can be expressed as

$$E\left(|S(z) - \hat{S}(z)|^2\right) = E\left(|S(z) - \mathbf{F}^T(z)\mathbf{U}(z)|^2\right), \tag{12.20}$$

where $\mathbf{F}(z) = [F_x(z)F_y(z)]^T$ is the filter applied to these two variables. Minimizing the error in Equation (12.20) in relation to $\mathbf{F}(z)$ leads to the filter

$$E\left(S(z)\mathbf{U}^T(z)\right) = E\left(\mathbf{U}^T(z)\mathbf{F}^T(z)\mathbf{U}(z)\right)$$

and

$$E\left(\mathbf{U}(z)\mathbf{F}^T(z)\mathbf{U}(z)\right) = \mathbf{F}^T(z)\begin{bmatrix} \gamma_{xx}(z) & \gamma_{yx}(z) \\ \gamma_{xy}(z) & \gamma_{yy}(z) \end{bmatrix}, \tag{12.21}$$

where $\gamma_{yy}(z)$ is the power spectral density of $y(n)$, $\gamma_{xx}(z)$ is the power spectral density of $x(n)$, $\gamma_{yx}(z)$ is the cross-power spectral density between $y(n)$ and $x(n)$, and $\gamma_{xy}(z)$ is the cross-power spectral density between $x(n)$ and $y(n)$.

Substitution of Equation (12.21) to $\hat{S}(z)$ leads to

$$\hat{S}(z) = \mathbf{F}^T(z)\begin{bmatrix} X(z) \\ Y(z) \end{bmatrix} = \left[\begin{bmatrix} \gamma_{xx}(z) & \gamma_{yx}(z) \\ \gamma_{xy}(z) & \gamma_{yy}(z) \end{bmatrix}^{-1}\begin{bmatrix} \gamma_{sx}(z) \\ \gamma_{sy}(z) \end{bmatrix}\right]^H \begin{bmatrix} X(z) \\ Y(z) \end{bmatrix}, \tag{12.22}$$

where H means Hermitian (complex-conjugate) transpose, $\gamma_{sx}(z)$ is the cross-power spectral density between $s(n)$ and $x(n)$, and $\gamma_{sy}(z)$ is the cross-power spectral density between $s(n)$ and $y(n)$. From this equation, it is not difficult to derive the following equation:

$$\hat{S}(k) = [X(z) - W(z)X(z)]\,H(z), \tag{12.23}$$

where $W(z)$ is the transfer function of AEC, and $H(z)$ is the transfer function of the NR system. More details of the derivation can be found in reference [7].

The first part of the integrated system is the AEC and its output is further processed by the NR technique. The combined structure is shown in Figure 12.5. In the first stage, the echo is canceled by the AEC filter $W(z)$, leaving the desired signal with noise. The second stage reduces noise through the NR techniques such as Wiener filter or spectral subtraction. The AEC and NR algorithm shown in Figure 12.5 will be presented in Section 12.7.4.

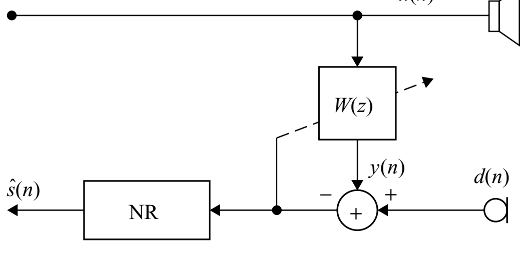


Figure 12.5 Combination of AEC with NR

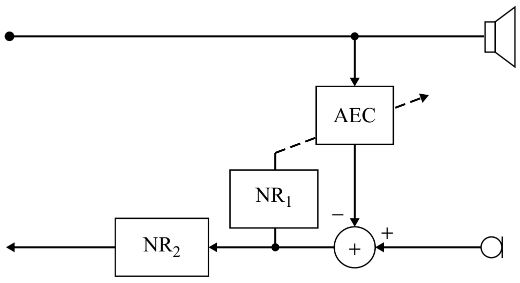


Figure 12.6 AEC followed by a closed-loop NR

There are several possible AEC and NR combinations. Figure 12.6 shows a different structure with the noise reduction filter NR_1 in the feedback loop of AEC. The noise in adaptation loop will disturb the update of AEC coefficients, thus NR_1 reduces the noise for updating AEC filter. The second filter NR_2 reduces noise in the voice path.

12.6 Voice Enhancement and Automatic Level Control

ITU-T has recommendations G.160 for voice enhancement device (VED) and G.169 for automatic level control (ALC). According to G.160, the NR module must be placed between the network echo canceler and the ALC module as shown in Figure 12.7. The NR reduces the effect of background noise originating from the microphone, and the ALC module compensates the network gain with a predetermined value.

12.6.1 Voice Enhancement Devices

Important functions of VED include the acoustic echo control, NR, and the recognition and accommodation of tandem-free operations. A G.160 compliant equipment ensures that the performance of overall network will not be degraded when the VED is installed in the network. According to G.160, NR functions have the following main characteristics:

- The ability to modify a noise-corrupted voice signal, thus improving the subjective quality.
- The ability to maintain the quality of the voice signal when it is not corrupted by noise.
- The ability to avoid corrupted voiceband or facsimile data.

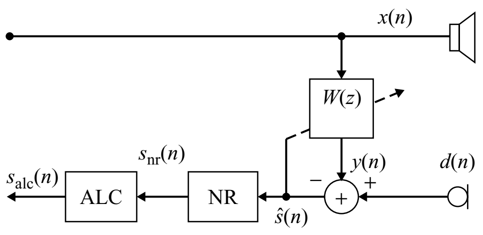


Figure 12.7 Echo canceler followed by NR and ALC

- The ability to prevent interferences caused by in-band network signaling tones.
- The ability to exhibit low throughput delay.
- The ability to provide 64-kbps bit-sequence integrity when VED was disabled.

12.6.2 Automatic Level Control

G.169 applies to the testing and evaluation of ALC devices used for digital network-based equipment. This standard defines test requirements and procedures such as requirements for passing DTMF tones, voiceband data, and frequency distortion. The ALC module is located in the digital transmission path to perform the signal level adjustment to meet a user-defined reference level. The ALC processes signals in the transmitting direction as illustrated in Figure 12.8.

If an echo cancelation device is used to reduce the reflected echoes, the ALC gain should meet certain stability. It assumes that the ALC device is installed at a location in the transmission path after the echo cancelation device. It also recommends that the maximum gain of the ALC device connected with an echo cancelation should not exceed +15 dB.

12.7 Experiments and Program Examples

In this section, we will discuss the implementation of the NR algorithm using MATLAB, C, and C55x programs.

12.7.1 Voice Activity Detection

In this experiment, we introduce a simple VAD using floating-point C. The experiment applies a 256-point complex FFT on the input signal with 8 kHz sampling rate. The FFT bins that cover the frequencies from 250 to 820 Hz are used for power calculation as shown in Figure 12.4. Table 12.1 lists the VAD algorithm used for the experiment.

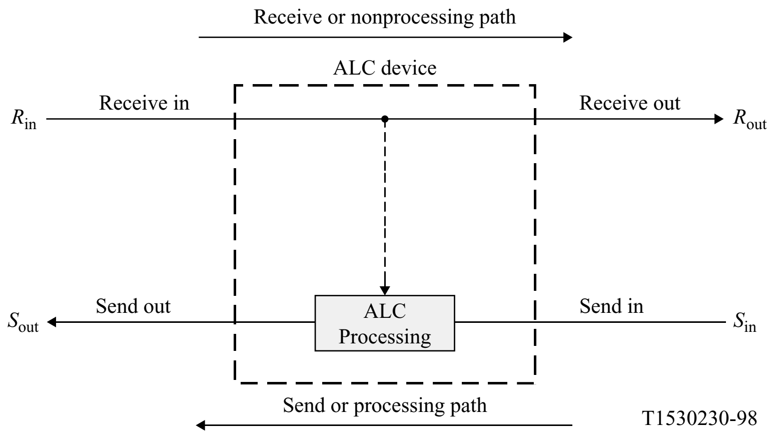


Figure 12.8 Block diagram of an ALC device (from G.169)

Table 12.1 Simple VAD algorithm, `floatPoint_vad.c`

```
short vad_vad(VAD_VAR *pvad)
{
    short k,VAD;
    float En;                              // Current frame power
    VAD_VAR *p = (VAD_VAR *)pvad;
    En = 0;                                // VAD algorithm (12.10)
    for (k=p->ss; k<=p->ee; k++)           // Power from 250 to 820 Hz
    {
         En += (float)(sqrt(p->D[k].real*p->D[k].real
            + p->D[k].imag*p->D[k].imag));
    }
    p->Em = p->am1*p->Em + p->alpham*En;   // Equation(12.11)
    if (p->Nf < p->Em)                     // Update noise floor (12.13)
        p->Nf = p->al1*p->Nf + p->alphal*En;
    else
        p->Nf = p->am1*p->Nf + p->alpham*En;
    p->thres = p->Nf + p->margin;
    VAD = 0;
    if (p->Em >= p->thres)
        VAD=1;
    if (VAD)               // Speech is detected since Em >= threshold
        p->hov = HOV;
    else                   // Silence is detected since Em < threshold
    {
        if (p->hov-- <=0)
            p->hov =0;
        else
            VAD=1;
    }
    return VAD;
}
```

In the program, the signal power computation starts from FFT bin `ss` and ends at `ee` according to the low and high frequencies, respectively. For a 256-point FFT and 8 kHz sampling frequency rate, the low frequency 250 Hz is the FFT bin $\texttt{ss} = 250 \cdot 256/8000 = 8$ and the high frequency 820 Hz is the FFT bin $\texttt{ee} = 820 \cdot 256/8000 = 26$. The short-term energy is computed using Equation (12.11). The detector uses a threshold `Em` to determine if the signal contains speech. If the energy is greater than the threshold, the detector will set VAD flag; otherwise, the flag will be cleared. In order to prevent detector oscillations, a small number is added to the threshold as a safety margin.

Table 12.2 lists the files used for this experiment. Figure 12.9 shows the VAD output against the noisy speech. If speech is detected, the output is 1 (indicated by amplitude 7500); otherwise, the output is silence.

Procedures of the experiment are listed as follows:

1. Open the CCS project file, rebuild, and load the program.
2. Run the experiment to obtain the output file.
3. Use MATLAB to plot both test speech file `speech.wav` and the VAD detection result file. Compare the VAD detector result with the speech file to evaluate the VAD detection.

Table 12.2 File listing for experiment `exp12.7.1_floatPointvad`

Files	Description
`floatPoint_vad_mainTest.c`	C function for testing VAD algorithm
`floatPoint_vad_vad.c`	NR uses VAD
`floatPoint_vad_hwindow.c`	Tabulated data table for Hanning window
`floatPoint_vad_ss.c`	FFT and preprocessing for VAD detection
`floatPoint_vad_init.c`	VAD initialization
`floatPoint_vad_fft.c`	FFT function and bit reversal
`floatPoint_vad.h`	C header file defines constant and function prototyping
`floatPoint_vad.pjt`	DSP project file
`floatPoint_vad.cmd`	DSP linker command file
`param.txt`	Parameter file
`speech.wav`	Data file

4. Modify the parameter file `param.txt` for different test files and lengths of the FFT.

5. Modify the test file `floatPoint_vad_mainTest.c` such that the experiment will output wave file controlled by the VAD flag. When VAD flag is cleared, write zeros to the output file. When the flag is set, copy the input to the output directly. Listen to the output wave file. Is the VAD detection accurate?

6. Adjust the hangover length and listen to the transition effect under different hangover lengths.

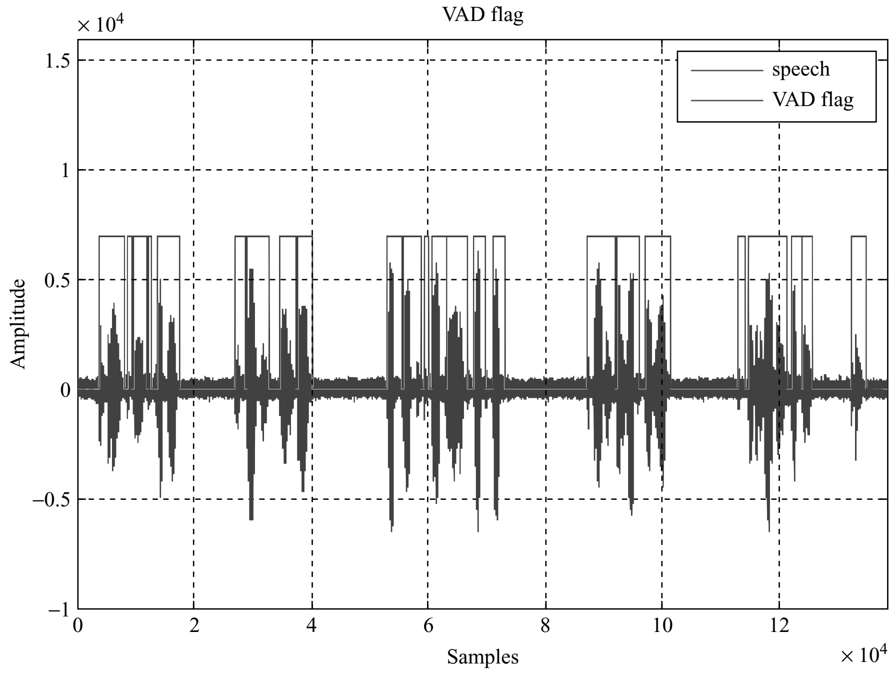


Figure 12.9 Voice activity detection of a noisy speech

12.7.2 MATLAB Implementation of NR Algorithm

This MATLAB experiment uses the block size of 256 with a moving window of length 128 (50 % overlap). The MATLAB script `nr.m` listed in Table 12.3 performs windowing, VAD, NR, and 50 % signal overlap. The signal array is `s[ ]`, and the processed data is stored in the array `ss[ ]`. The MATLAB script `NR_test.m` listed in Table 12.4 uses the spectrum subtraction function.

The waveforms before (top plot) and after (bottom plot) processing are shown in Figure 12.10. The noise level are attenuated about 30 dB. The horizontal axis is the sample index, and the vertical axis is the amplitude with full scales of 32 767 of 16-bit data.

Table 12.5 lists the files used for this experiment. Procedures of the experiment are listed as follows:

1. Start MATLAB and set working directory to the director `exp12.7.2_matlabNR`.
2. Run MATLAB program. Observe the processed data and compare it with the original noisy data file.
3. Listen to the output file and compare it with the original noisy data file.
4. Use different data files to evaluate the NR algorithm.

12.7.3 Floating-Point C Implementation of NR

The core part of spectral subtraction-based NR algorithm with VAD is listed in Table 12.6. The array `TB[k]` is the FFT bin of the input signals, and `NS[k]` is the estimated noise bin. Based on the VAD information, either spectrum subtraction or amplitude attenuation is applied. `N` is the half of frame size, and `h[k]` is the filter response. The files used for this experiment are listed in Table 12.7.

Procedures of the experiment are listed as follows:

1. Open the CCS project file, rebuild, and load the program.
2. Run the experiment to obtain the output file.
3. Use MATLAB to plot both the speech file `speech.wav` and the NR result file. Compare these two files to evaluate the NR algorithm.
4. Modify the parameter file `param.txt` for using different attenuation factors. Compare the results obtained using attenuation factors 0.5, 0.1, and 0.0.1.
5. Compare the experiment results with the experiment given in Section 12.7.1. Describe the differences between the results from both experiments.

12.7.4 Mixed C55x Assembly and Intrinsics Implementation of VAD

This experiment implements the voice activity detection using C55x intrinsics. The files used for this experiment are listed in Table 12.8. This experiment uses mixed C-and-assembly implementation that takes the advantages of efficient assembly programming for computational intensive functions, while leaves the control or configuration functions in C.

Table 12.3 List of MATLAB script for spectral subtraction

```
function [ss]=NR(s,Fs)

nw = 256;
overLap = nw/2;
nf=1+floor((ns-nw)/overLap);            % Number of frames

% Calculate Hamming windowing
win=hamming(nw);

% Processing loop
idx=nw;
for is=1:nf
   x=fft(s(((is-1)*overLap+1):1:idx).*win);
   yy=x;
   x2=x(1:129).*conj(x(1:129));

   % Windowing
   Ls = 7./8.;
   pxnS=Ls*pxnS+(1-Ls)*x2;
   Ll = 63./64.;
   pxnL=Ll*pxnL+(1-Ll)*x2;

   % VAD energy calculation
   ppxnS = pxnS(20:60)'*pxnS(20:60);
   ppxnL = pxnL(20:60)'*pxnL(20:60);

   % Update noise floor
   if (ppxnS > ppxnL)
     pn=Ll*pn+(1-Ll)*x2;
   else
     pn=Ls*pn+(1-Ls)*x2;
   end
   ppn = pn(20:60)'*pn(20:60);
   if(ppxnS > (1.05*ppn+20))
     hoCounter = 6;                     % Hangover counter
     os = pn./(pxnS);
     os = min(os,1);                    % Protect overdriving
     q = 1-sqrt(os);
   elseif (hoCounter >=1)
     os = pn./(pxnS);
     os = min(os,1);                    % Protect overdriving
     q = 1-sqrt(os);
     hoCounter = hoCounter-1;
   else
     q=0.04*ones(129,1);                % Silence frame
   end

   % Compose symmetric complex
   y =x(1:129).*q;
   yy(1:129)=y;
   yy(129:256)=conj(y(129:-1:2));
```

continues overleaf

Table 12.3 (*continued*)

```
   % Shift the memory for history
   ss2(1:nw)=ss1(1:nw);
   ss1(1:nw)=real(ifft(yy));

   % Doing 50% overlapping
   temp = ss1(1:128).*win(1:128);
   temp = temp + ss2(129:256).*win(129:256);
   ss(((is-1)*overLap+1):1:(idx-256+128)) =2*(temp);
   idx=idx+overLap;
end
```

Table 12.4 List of MATLAB code, `NR_test.m`

```
fs=8000;                                  % Sampling frequency
fid1=fopen('.\data\speech.pcm','r');      % Open the pcm input file
fid2=fopen('.\data\processed.pcm','w');   % Open the pcm output file
[s,COUNT]=fread(fid1,'int16');           % Read input file
[ss]=NR(s,fs);                           % Call spectral subtraction routine
fwrite(fid2,ss,'int16');                 % Write processed data to output
```

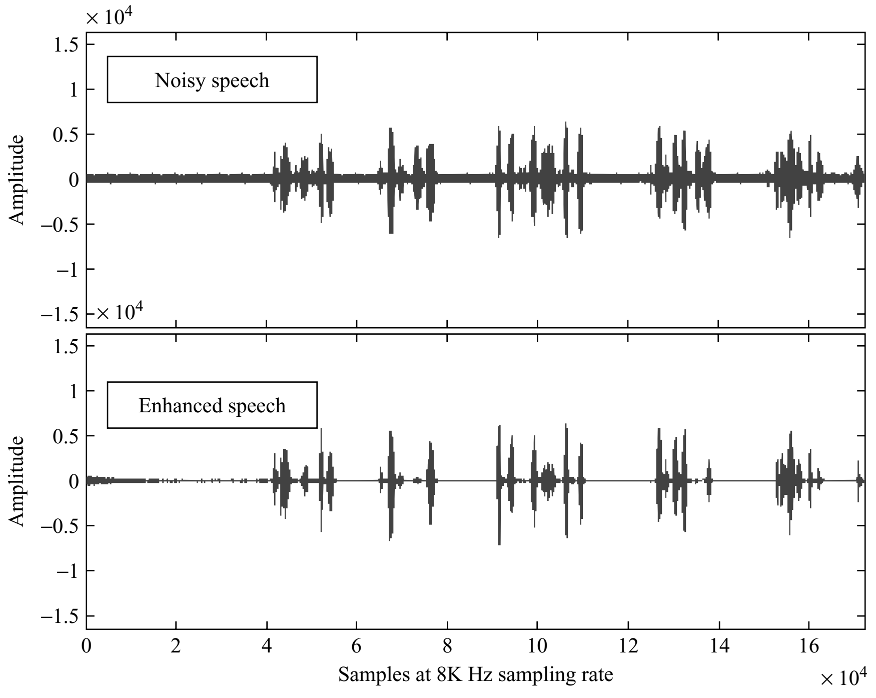


Figure 12.10 Speech waveform comparisons before and after NR

Table 12.5 File listing for experiment `exp12.7.2_matlabNR`

Files	Description
`NR_test.m`	Main function for testing NR function
`NR.m`	NR algorithm with VAD
`speech.pcm`	Speech file

Table 12.6 List of spectral subtraction algorithm

```
if (VAD)                              // Speech is detected since VAD=1
{
    for (k=0;k<=N;k++)
    {
        tmp = TB[k] - (127./128.)*NS[k];
        h[k] = tmp / TB[k];
    }
}
else                                  // Silence is detected since VAD=0
{
    Npw = 0.0;
    for (k=0;k<=N;k++)                // Update the noise spectrum
    {
        NS[k] = (1-alpha)*NS[k] + alpha*TB[k];
        Npw += NS[k];
    }
    Npw = Npw/Npw_normalfact;         // Normalized noise power
    margin = (127./128.)*margin+(1./128.)*En; // New margin
}
```

Table 12.7 File listing for experiment `exp12.7.3_floatPointNR`

Files	Description
`floatPoint_nr_mainTest.c`	C function for testing NR algorithm
`floatPoint_nr_vad.c`	Voice activity detector
`floatPoint_nr_hwindow.c`	Tabulated data table for Hanning window
`floatPoint_nr_ss.c`	Preprocessing before using FFT
`floatPoint_nr_init.c`	NR initialization
`floatPoint_vad_fft.c`	FFT function and bit reversal
`floatPoint_nr_proc.c`	NR control function
`floatPoint_nr.h`	C header file defines constant and function prototyping
`floatPoint_nr.pjt`	DSP project file
`floatPoint_nr.cmd`	DSP linker command file
`param.txt`	Parameter file
`speech.wav`	Data file

Table 12.8 File listing for experiment `exp12.7.4_mixed_VAD`

Files	Description
`mixed_vad_mainTest.c`	C function for testing VAD algorithm
`mixed_vad_vad.c`	Voice activity detection algorithm
`mixed_vad_tableGen.c`	Lookup table for Hanning window
`mixed_vad_ss.c`	Preprocessing before using FFT for VAD
`mixed_vad_init.c`	Initialization for VAD experiment
`mixed_vad_wtable.c`	Generate FFT twiddle factors
`fft.asm`	Assembly FFT function
`bit_rev.asm`	Assembly bit-reversal function
`dspFunc55.asm`	Assembly supporting functions for VAD experiment
`mixed_vad.h`	C header file defines the external variables
`mixed_VAD.pjt`	DSP project file
`mixed_VAD.cmd`	DSP linker command file
`param.txt`	Parameter file
`long_speech.pcm`	Data file
`short_speech.pcm`	Data file

In this mixed C-and-assembly implementation, FFT, square root function used in power calculation, and 32- by 16-bit multiplication are implemented in C55x assembly language `fft.asm` in and `dspFunc55.asm`. Procedures of the experiment are listed as follows:

1. Open the CCS project file, rebuild and load the experiment program.
2. Run the experiment to obtain the output file. Compare the result with the floating-point C experiment given in Section 12.7.1.
3. Use Section 12.7.3 as reference to modify this experiment using C55x intrinsics. Run the mixed C-and-assembly experiment and compare the result with experiment given in Section 12.7.3.
4. Use the DSP/BIOS knowledge learned from previous experiments to create a real-time NR experiment. The experiment uses a microphone as input device and a loudspeaker as the output. The sampling rate is 8 kHz and the FFT size is 256.

12.7.5 Combining AEC with NR

In this experiment, we combine AEC and NR as presented in Figure 12.5. The files used for this experiment are listed in Table 12.9. A portion of the C function is listed in Table 12.10 that performs the AEC followed by the NR.

The AEC is a sample-by-sample processing while the NR is a frame-based processing. The relationship among these signals is illustrated in Figure 12.11.

We may compare the outputs of combined AEC and NR to evaluate the improvement of AEC using the NR. The waveform comparison is illustrated in Figure 12.12. It shows that the noise at the output of AEC is further attenuated in the NR stage.

Procedures of the experiment are listed as follows:

1. Open the CCS project file, rebuild, and load the program.
2. Run the experiment to obtain the output file. Compare the result with the floating-point C experiment given in Section 10.7.2.

Table 12.9 File listing for experiment `exp12.7.5_floatPointAecNr`

Files	Description
`floatPoint_aecNr_mainTest.c`	Main function with file I/O for testing AEC or NR
`floatPoint_nr_vad.c`	NR uses VAD
`floatPoint_nr_hwindow.c`	Generate Hanning window lookup table
`floatPoint_nr_proc.c`	NR algorithm
`floatPoint_nr_ss.c`	Data preprocessing before calling VAD
`floatPoint_nr_init.c`	NR initialization
`floatPoint_nr_fft.c`	FFT function and bit reverse
`floatPoint_aec_calc.c`	AEC algorithm
`floatPoint_aec_util.c`	AEC supporting functions
`floatPoint_aec_init.c`	AEC initialization
`nr.h`	NR C header file
`aec.h`	AEC C header file
`float_AECNR.pjt`	DSP project file
`float_AECNR.cmd`	DSP linker command file
`param.txt`	Parameter file
`mic.pcm`	Data file of microphone input
`far.pcm`	Data file of far-end input

Table 12.10 Source code of `floatPoint_aecNr_mainTest.c`

```
while((fread(temp1,sizeof(char),2*nrvar.L,farIn)==(2*nrvar.L)))// Far-end
{
  fread(temp2,sizeof(char),2*nrvar.L,micIn);                  // Near-end
  for (i=0; i<2*nrvar.L; i+=2)
  {
    farEndIn     = (temp1[i]&0xFF) | (temp1[i+1]<<8);
    microphoneIn = (temp2[i]&0xFF) | (temp2[i+1]<<8);
    for (j=0;j<nrvar.L-1;j++)         // Buffer data for NR
    {
      input[j] = input[j+1];
    }
    input[j] = aecCalc(microphoneIn,farEndIn,&aec);
    count++;
    if(count == nrvar.N)              // Frame based noise reduction
    {
      count = 0;
      nrvar.vadFlag = nr_ss(pnr);
      nr_proc(pnr);
      for (j=0; j<nrvar.N; j++)
      {
          temp3[2*j]   = (input[j]&0xFF);
          temp3[2*j+1] = (input[j]>>8)&0xFF;
      }
      fwrite(temp3,sizeof(char),2*nrvar.N,txOut);    // Write farEndOut
    }
  }
}
```

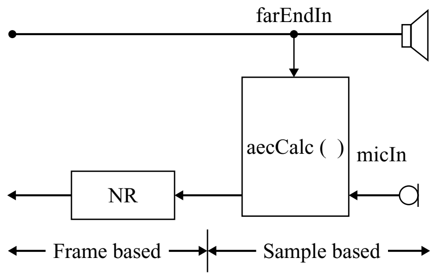


Figure 12.11 Combined AEC and NR functions

3. Use the same approach to convert this experiment to fixed-point C or use C55x intrinsics.

4. To further improve run-time efficiency, replace the fixed-point C FFT and inverse FFT with the C55x assembly routines.

5. Create a DSP/BIOS experiment for real-time AEC + NR experiment. Configure the DSK for 8 kHz sampling rate for input and output signals. Set AIC23 in stereo input such that one input is used for line-in and other for microphone input. Set AIC23 in stereo output such that one output plays the far-end input audio to a loudspeaker and the other output is used for the AEC + NR output. We can use a PC sound card to record the AEC + NR output and evaluate the real-time performance of AEC + NR.

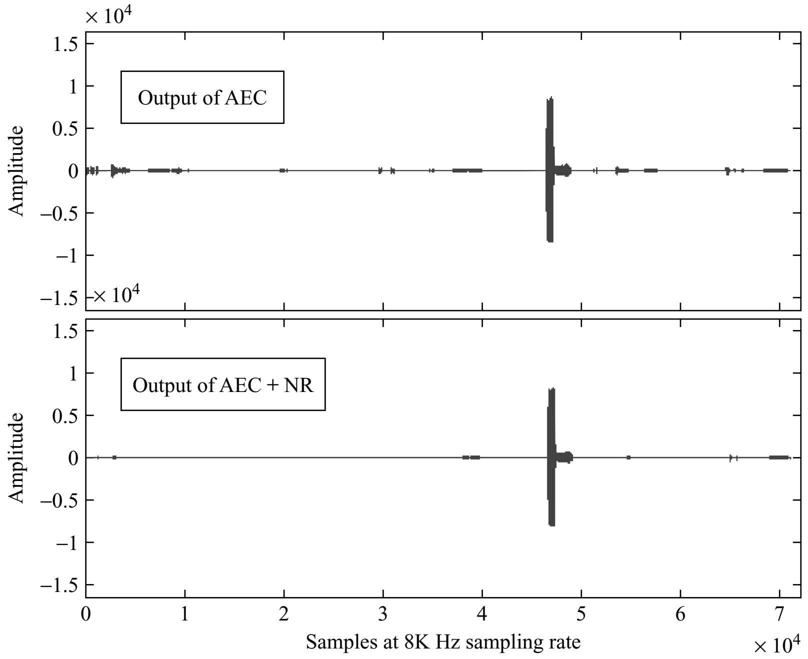


Figure 12.12 Comparison of waveforms between the output of AEC and output of AEC + NR

References

[1] S. M. Kuo and D. R. Morgan, *Active Noise Control Systems – Algorithms and DSP Implementations*, New York: John Wiley & Sons, Inc., 1996.
[2] MATLAB, Version 7.0.1, Release 14, Sep. 2004.
[3] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989.
[4] Texas Instruments, Inc., *Acoustic Echo Cancellation Software for Hands-Free Wireless Systems*, Literature no. SPRA162, 1997.
[5] Texas Instruments, Inc., *Echo Cancellation S/W for TMS320C54x*, Literature no. BPRA054, 1997.
[6] H. Gustafsson, S. E. Nordholm, and I. Claesson, 'Spectral subtraction using reduced delay convolution and adaptive averaging,' *IEEE Trans. Speech and Audio Process.*, vol. 9, no. 8, pp. 799–807, Nov. 2001.
[7] W. L. B. Jeannes, P. Scalart, G. Faucon, and C. Beaugeant, 'Combined noise and echo reduction in hands-free systems: A survey,' *IEEE Trans. Speech and Audio Process.*, vol. 9, no. 8, pp. 808–820, Nov. 2001.
[8] ITU-T Recommandation G. VED/G.160, *Voice Enhancement Devices*, Draft no. 6.1, 2002.
[9] ITU-T Recommandation G.169, *Automatic Level Control Devices*, June 1999.

Exercises

1. Using the experiment given in Section 12.7.1, calculate the percentage of silence frames for speech file `speech.wav`. In a normal conversation, there should be over 50 % of silence frames.

2. Explain why different window sizes are used for calculating signal energy used for VAD? During the onset of speech, which output is bigger using Equations (12.11) and (12.12)? What happens at the offset (tail) of speech?

3. The music-tone effect is due to the oversubtraction of certain frequencies. In order to make this effect less noticeable, the mild subtraction should be applied. As a result, the canceled noise is less as compared with the full subtraction and this will compromise the cancelation effect. Write a MATLAB function to control this parameter among different subtraction factors expressed in Equation (12.9).

4. Write a program to compare the music-tone effect of using Equation (12.6) to estimate the signal and using half-wave rectification method in Equation (12.17). Also compare the intelligibility of the processed speech between two methods.

5. Based on Equation (12.23), instead of using NR followed by AEC, the Winner filter for NR filter could be used in front of AEC as shown in Figure 12.13. The near-end filter $H(z)$ is used to remove the statistically known noise introduced from the microphone. Explain why we also need an identical filter in the far-end branch.

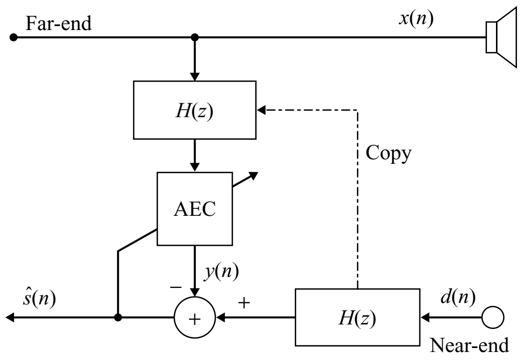


Figure 12.13 The structure with Winner filter in front of AEC

13 Audio Signal Processing

Digital audio signal processing techniques are widely used in consumer electronics such as CD players, high-definition televisions, portable audio devices, and home theaters. For professional audio, the applications can be found in the fields of digital sound broadcasting, program distribution, computer music, and digital storage. This chapter introduces the basic audio coding algorithms and multichannel audio CODECs.

13.1 Introduction

The compact disc (CD) is a very popular digital audio format. It stores audio signals with 44.1 kHz sampling rate and 16-bit PCM format, thus results in a bit rate of 1411.2 kbps for stereo audio. This posts a challenge on channel bandwidth or storage capacity for emerging digital audio applications such as multimedia streaming over IP network, wireless mobile, audio and video conferencing, and digital radio broadcasting. The increasing demand for better quality digital audio, such as multichannel audio coding (5–7 channels), or higher sampling rate (96 kHz), requires more sophisticated encoding and decoding techniques in order to minimize the transmission cost and provide cost-efficient storage.

The efficient speech coding using a vocal-tract model discussed in Chapter 11 is not applicable to audio in general. In addition, we have to deal with stereo or multichannel signal presentations, higher sampling rate, higher resolution, wider dynamic range, and higher listener expectation. Audio signal compression algorithms that satisfy these requirements include psychoacoustics, transform coding, Huffman coding, and reduction of interchannel redundancies. Combination of these techniques leads to the development of algorithms for perceptually transparent high-fidelity coding, CD-quality digital audio. In addition to the basic requirements of low-bit rate, high quality of reconstructed audio signals, robustness to channel bit errors and packet loss, and low complexity in decoding are also required.

Many audio coding algorithms have become international standards for commercial products, particularly the MPEG standards, which will be described in the following sections. In particular, MPEG Layer-3 (MP3) is a very popular media format for Internet audio delivery and portable audio players. MP3 provides a range of bit rates from 8 to 320 kbps, and supports the switching of bit rates between audio frames. MP3 at 320 kbps produces perceptually comparable quality with CD at 1411.2 kbps. In this chapter, the MP3 coding standard is discussed in details as an example for audio coding.

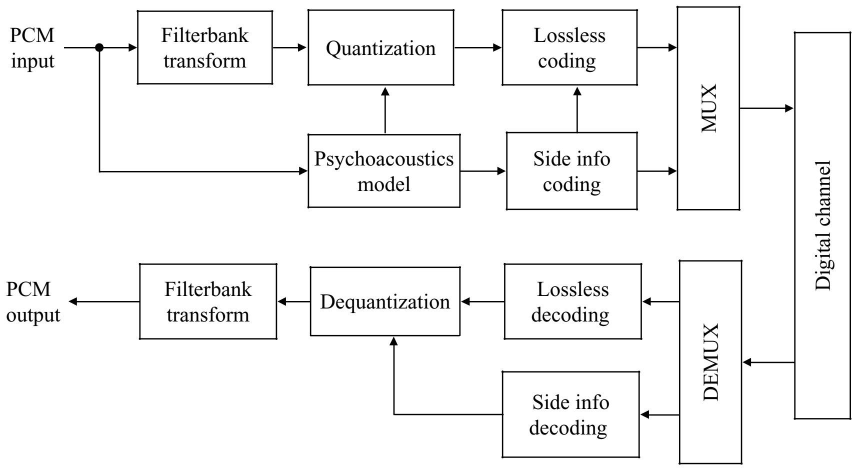


Figure 13.1 Basic structure of audio coding and decoding

13.2 Basic Principles of Audio Coding

Lossy compression is applied to speech and audio signals based on the noise shaping, where the noise below the masking threshold may not be audible. Lossless compression is also applied to audio signals because of large amount of data due to high sampling rate. This section describes the principles of these lossy compressions using psychoacoustics and lossless compression using Huffman coding.

Figure 13.1 shows the basic structure of an audio CODEC. The function of each module will be briefly described, and some modules will be further discussed in the following sections:

Filterbank: It splits the full-band signals into several subbands uniformly or according to the critical band model of the human auditory system. For example, there are 32 subbands used by MPEG-1.

Transform: It converts time-domain signals to frequency-domain coefficients. For example, the modified discrete cosine transform (MDCT) is used in MPEG-1 Layer 3. In MPEG-2 AAC (advanced audio coding) or Doubly AC-3 [1, 2], the MDCT functions as filterbank for splitting full-band signals.

Psychoacoustics model: It calculates masking threshold according to human auditory-masking effect from the spectral coefficients, and uses the masking threshold to quantize the MDCT coefficients.

Lossless coding: It further removes the coded bit-stream redundancy using entropy coding. For example, Huffman coding is used in MPEG-1 Layer 3.

Quantization and dequantization: It quantizes the MDCT coefficients to indices based on masking threshold provided by psychoacoustics model on the encoding, and converts the indexes back to the spectral coefficients on the decoding.

Side information: Bit-allocation information needed for the decoder.

Multiplexer and demultiplexer: It packs and unpacks the coded bits into bit stream.

The encoded bit-stream format for transmitting the compressed audio signal is shown in Figure 13.2 and described as follows [3]:

Header: It contains information about the format of the frame. It starts with a synchronization word that is used to find the beginning of a frame. The header also contains information about the bit stream, including layer number, bit rate, sampling frequency, and stereo encoding parameters. For example, the header field in MP3 has 32 bits.

Header	CRC (optional)	Side information	Main data	Ancillary information

Figure 13.2 Typical encoded audio bit-stream format

CRC (cyclic redundancy checksum): It protects the header. When it is present, the decoder calculates a CRC on the header and compares it with the CRC in the frame. If the CRC does not match, the decoder starts looking for a new sync word. For example, the sync word for MP3 is a 12-bit 0xFFF. More CRC implementation can be found in Section 14.2.3.

Side information: It contains information for decoding and processing the main data. Different steps in the decoding process use this global information. For example, when performing Huffman decoding of the main data in MP3, the information about which Huffman table has been used in the encoder is stored in the side information. The Huffman encoded data and the side information are combined in a single bit stream.

Main data: It consists of coded spectral coefficients and lossless encoded data. For example, MP3 includes scaling factors that are used to reconstruct the original frequency lines from the information in the Huffman data.

Ancillary data: It holds the user-defined information such as song title or optional song information.

13.2.1 Auditory-Masking Effects for Perceptual Coding

Auditory masking or psychoacoustics describes the principles that a low-level signal (the maskee) can become inaudible when a louder signal (the masker) occurs simultaneously since human ear does not respond equally to all frequency components. This phenomenon can be exploited in speech and audio coding by an appropriate noise shaping in the encoding process. The masking depends on the spectral distribution of masker and maskee, and on their variation with time. The perceptual weighting method for speech coding has been discussed in Chapter 11. Audio signals are also based on the similar auditory-masking effect but with wider bandwidth.

The quiet (absolute) threshold is approximated by the following nonlinear function [4]:

$$T_{\mathrm{q}}(f) = 3.64(f/1000)^{-0.8} - 6.5e^{-0.6(f/1000-3.3)^2} + 10^{-3}(f/1000)^4 \text{ (dB SPL)}, \tag{13.1}$$

where SPL represents sound pressure level. For example, a 16-bit full-scale sinusoid that is precisely resolved by the 512-point fast Fourier transform (FFT) in bin will yield a spectral line having 84-dB SPL. With 16-bit sampling resolution, SPL estimates very low amplitude input signals that are at or below the absolute threshold.

Equation (13.1) represents a listener with acute hearing and any signal level below will not be perceived. The approximation curve is shown in Figure 13.3 using the log scale in frequency. Most humans cannot sense frequencies below 20 Hz nor above 20 kHz. This range tends to narrow as we aged. For example, a middle-aged man will not hear much of the signals above 16 kHz [5]. Frequencies ranging from 2 to 4 kHz are the easiest to perceive at a relatively low volume. This frequency-domain phenomenon is the simultaneous masking in which a low-level signal can be inaudible (masked) by a simultaneously occurring stronger signal if masker and maskee are close enough to each other in frequency.

The simultaneous masking is dependent on the relationship between frequencies and their relative volumes. The temporal masking is based on time instead of frequency. For example, if a stronger tonal

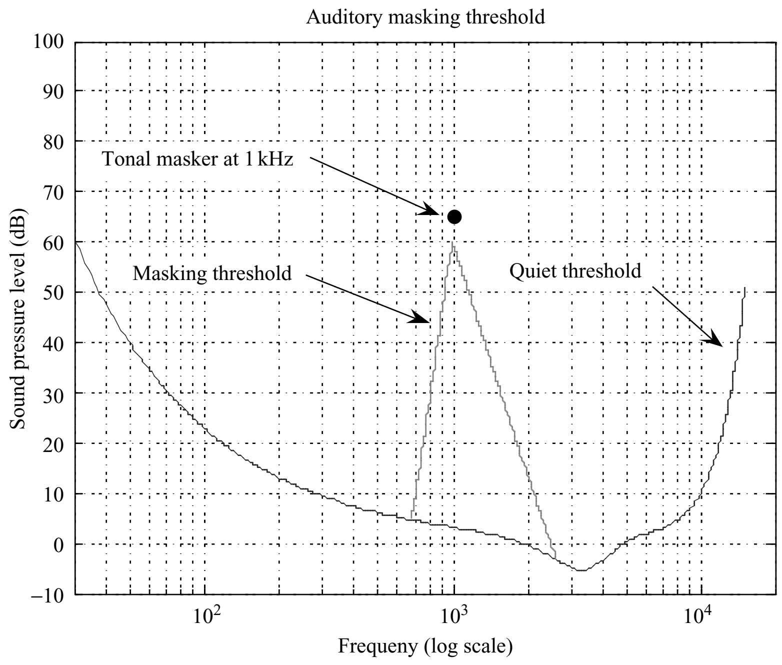


Figure 13.3 Auditory-masking thresholds

component is played before a quiet sound, the quiet sound may not be heard if it appears within the order of 50–200 ms [6]. The masking threshold will mask weaker signals around the masker.

Using the absolute threshold of hearing to shape the spectrum of coding distortion represents the first step toward perceptual coding. However, the most useful threshold is the masking threshold when stimuli (audio signals) are present. The detection threshold for spectral quantization noise is a combination of the absolute threshold and the shape determined by the stimuli present at any given time. Since stimuli are time varying, the masking threshold is also a time-varying function of the input signal. In order to estimate this threshold based on the rules highlighted in Figure 13.3, spectral coefficients are used to compare the relative magnitude.

Human does not respond linearly to all frequency components. The auditory system can be roughly divided into 26 critical bands; each band is a bandpass filter with bandwidth of 50–100 Hz for signal below 500 Hz, and with bandwidth up to 5000 Hz for signal at high frequencies. One critical band comprises one bark. Within each critical band, the auditory-masking threshold, which is also referred as the psychoacoustics masking threshold (or the threshold of the just noticeable distortion) can be determined. Frequencies in these critical bands (or barks) are harder to distinguish by human ears.

The conversion from frequency to bark (critical band) can be approximated using the following equation [4]:

$$z(f) = 13\tan^{-1}(0.00076f) + 3.5\tan^{-1}[(f/7500)^2] \text{ (Bark).} \tag{13.2}$$

This equation converts frequency in Hz to the bark scale. Thus, one critical bandwidth comprises one bark. The masking curves shown in Figures 13.3 and 13.4 are plotted using this approximation.

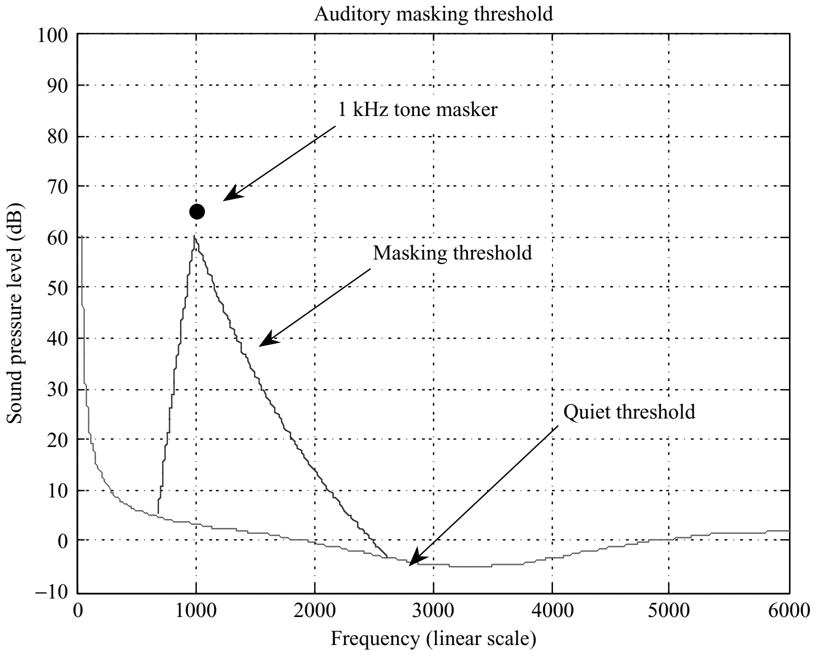


Figure 13.4 Masking thresholds with linear frequency scale

Suppose there is a dominant tonal component in an audio signal. Figure 13.4 shows that this dominant noise will introduce a masking threshold that masks out frequencies in the same critical band. This frequency-domain masking phenomenon is known as simultaneous masking, which has been observed within critical bands. This effect is also known as the spread of masking. It is often modeled in coding applications by a triangular spreading function that has slopes of 25 and -10 dB per Bark for the lower and higher frequencies, respectively. The 1 kHz tone masking threshold in Figures 13.3 and 13.4 is calculated based on this assumption.

Spreading function described here is a simplified version. However, psychoacoustics experiments showed that sloppiness of the spreading function depends on the masker loudness. This model assumes constant loudness during the encoding, which is not accurate but simple. Using fixed spreading function might lead to over or under masking in some cases that in turn reduces coder performance. However, this spreading masking is the best during the encoding since we do not know the sound level during play back. This masking model is a key factor to the efficiency of the algorithm.

Example 13.1: Using the masking thresholds shown in Figure 13.3, calculate the masking effect for the following scenarios: Given a 65 dB tone at 2 kHz and two test tones played at 2.5 (40 dB) and 1.5 kHz (40 dB), calculate if it is necessary to code these two tones or not.

We first use Equation (13.2) to calculate how many barks are there between these two tones and the masker. We will draw a picture with barks and masking contours to see if these two tones are under the masking threshold of 2 kHz tone or not. The magnitude contours of masker and masking threshold are plotted in Figure 13.5 using the MATLAB code `example13_1.m`. From the figure, it is clear that there are several bark bands across the tones between 2 and 1.5/2.5 kHz. Due to the sharper slope (25 dB per bark) on the left side (lower frequency) of the masker tone, the test tone at 1.5 kHz is much higher than the masking threshold of this masker. Therefore, this 1.5 kHz tone

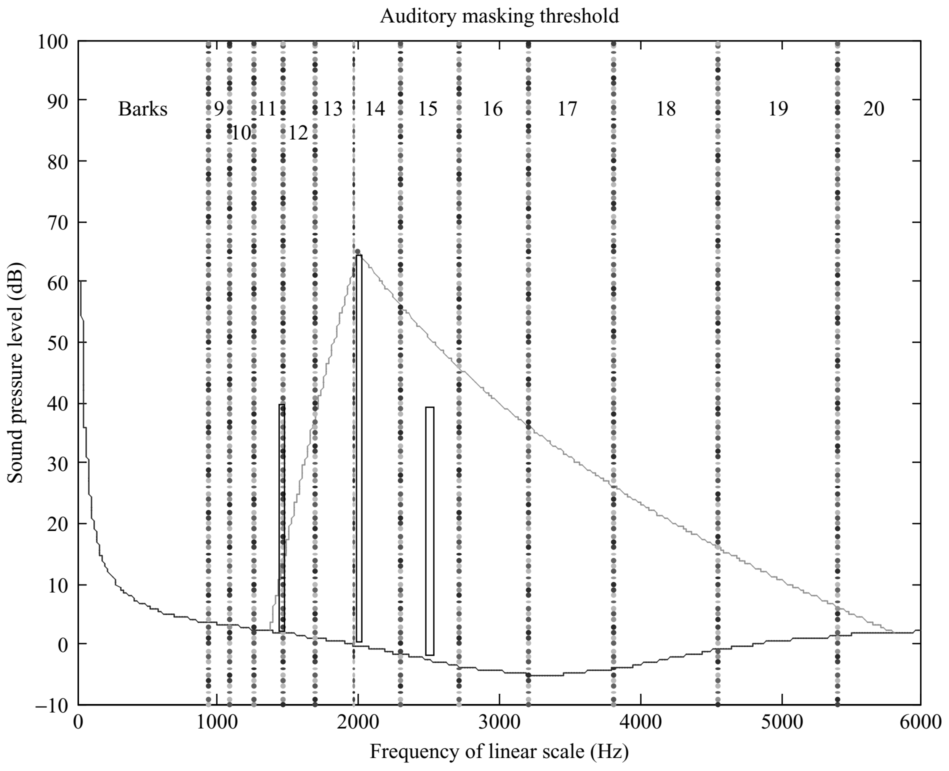


Figure 13.5 Example of masking effect of a 2 kHz tone

cannot be masked but be left as a masker tone. On the other hand, due to the slow slope (−11 dB per bark) on the right side (higher frequency) and wider frequency coverage in the bark bank in higher frequency, this 2.5 kHz tone is under the masking threshold of this 2 kHz masker tone's masking threshold. This example shows the masking effect is not symmetric with more effect on higher frequencies.

13.2.2 Frequency-Domain Coding

The MDCT is widely used for audio coding techniques. In addition to the energy compaction capability similar to discrete cosine transform (DCT), MDCT can simultaneously achieve critical sampling, reduction of block effects, and flexible window switching.

The MDCT uses the concept of time-domain aliasing cancelation, while the quadrature mirror filterbank (QMF) uses the concept of frequency-domain aliasing cancelation. This can be viewed as the duality of MDCT and QMF. However, the MDCT also cancels frequency-domain aliasing, while the QMF does not cancel time-domain aliasing. In other words, only the MDCT achieves perfect reconstruction.

Before the introduction of MDCT, transform-domain-based audio coding techniques used discrete Fourier transform (DFT) or DCT with window functions such as a rectangular window. However, these early coding techniques cannot meet the contradictory requirements, i.e., critical sampling vs. block effect. For example, when a rectangular window DFT (or DCT) analysis/synthesis system is critically sampled, the system suffers from poor frequency resolution in addition to block effects. Overlapped window provides better frequency response with the penalty of requiring additional values in the frequency domain. MDCT has solved this problem by introducing window switching to tackle possible preecho problems in the case of insufficient time resolutions.

The MDCT of $x(n)$, $n = 0, 1, \ldots, N-1$, is expressed as

$$X(k) = \sum_{n=0}^{N-1} x(n) \cos\left[\left(n + \frac{N+2}{4}\right)\left(k + \frac{1}{2}\right)\frac{2\pi}{N}\right], \qquad k = 0,\ 1,\ \ldots, N/2-1, \tag{13.3}$$

where $X(k)$ is the kth MDCT coefficient. The inverse MDCT (IMDCT) is defined as

$$x(n) = \frac{2}{N}\sum_{k=0}^{N/2-1} X(k) \cos\left[\left(n + \frac{N+2}{4}\right)\left(k + \frac{1}{2}\right)\frac{2\pi}{N}\right], \qquad n = 0,\ 1,\ \ldots, N-1. \tag{13.4}$$

The relationship between the MDCT and the DFT can be established via shifted DFT. Using the DFT definition given in Equation (6.13), if we shift the time index n by $(N + 2)/4$ and the index k by 1/2, the DFT becomes

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\left[\left(n+\frac{N+2}{4}\right)\left(k+\frac{1}{2}\right)\frac{2\pi}{N}\right]}, \qquad k = 0,\ 1,\ \ldots, N/2-1. \tag{13.5}$$

Similarly, the inverse DFT with same shifting is derived as

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k) e^{-j\left[\left(n+\frac{N+2}{4}\right)\left(k+\frac{1}{2}\right)\frac{2\pi}{N}\right]}, \qquad n = 0,\ 1,\ \ldots, N-1. \tag{13.6}$$

For real-valued signals, it is easy to prove that MDCT coefficients in Equations (13.3) and (13.4) are equivalent to the real part of shifted DFT in Equations (13.5) and (13.6), respectively. This fact provides the base to calculate MDCT using FFT method. For implementation of time-domain aliasing cancelation, the window needs to satisfy the following conditions to have perfect reconstruction [7]:

1. The analysis and synthesis windows must be equal, and the length N must be an even number.

2. The window coefficients must be symmetric as

$$h(n) = h(N-n-1). \tag{13.7}$$

3. These coefficients must satisfy power complimentary requirement as

$$h^2(n + N/2) + h^2(n) = 1. \tag{13.8}$$

Several windows satisfy those conditions. The simplest case but rarely used is the modified rectangular window expressed as

$$h(n) = 1/\sqrt{2}, \qquad 0 \le n \le N-1. \tag{13.9}$$

The sine window that is used by MP3 and AC-3 is expressed as

$$h(n) = \sin\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)\right], \qquad 0 \le n \le N-1. \tag{13.10}$$

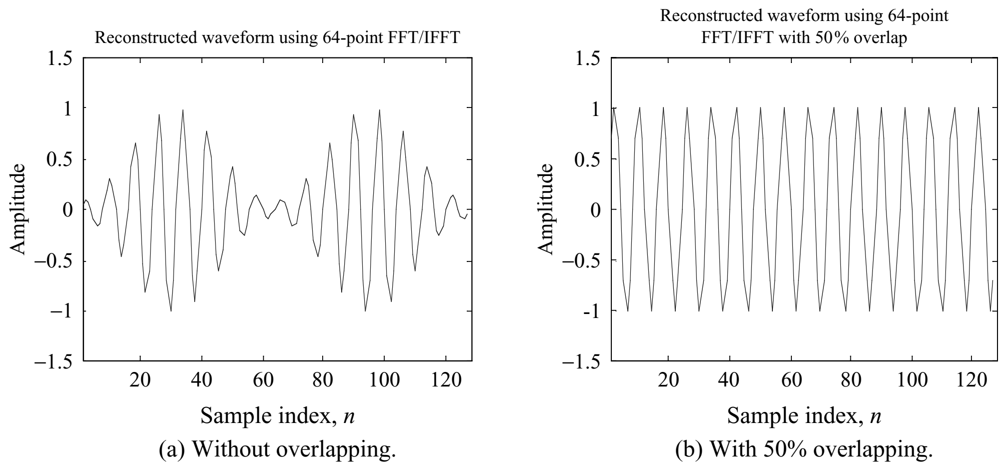


Figure 13.6 Hanning windowed FFT/IFFT: (a) without overlapping; (b) with 50 % overlapping

Note that the windows applied to the MDCT are different from the windows that are used for other types of signal analysis. One of the reasons is that MDCT windows are applied twice for both the MDCT and the IMDCT.

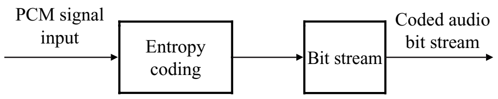
Example 13.2: This example illustrates the block effect of DFT. In order to have better frequency resolution, Hanning window is applied. Given a 1 kHz tone as the input, (1) perform the 64-point DFT without overlapping; and (2) with 50 % overlap for DFT. Compare the waveforms between (1) and (2), and calculate the total number of frequency coefficients needed for transmission in each case.

The waveform without overlapping is shown in Figure 13.6(a). In order to achieve the higher frequency resolution, the time-domain waveform is reconstructed with window. In this case, there are 32 bins (complex) that need to be quantized. The waveform with 50 % overlapping is reconstructed perfectly as shown in Figure 13.6(b). The penalty of this reconstruction is the need to quantize frequency bins twice, i.e., 64 complex bins.

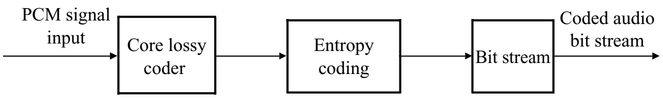
Preechoes are very common artifacts in the perceptual audio coding schemes using high-frequency resolution. The name 'preecho' describes the artifact occurred even before the music event that causes such noise. Switching window between the length $N = 512$ and 64 resolves this problem. For example, AC-3 uses different window sizes to achieve different resolutions, and MP3 decoder also supports the switching of window from 36 to 12 to increase the time-domain resolution. We will discuss more about the preecho effects in Section 13.5.4.

13.2.3 Lossless Audio Coding

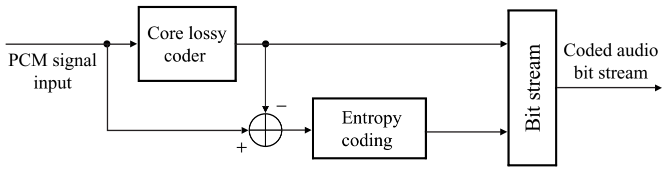
Lossless audio coding uses entropy code to further remove the redundancy of the coded data without any loss in quality. Figure 13.7 shows three typical types of lossless coder structures. Figure 13.7(a) is a pure entropy coding structure, or a lossless-only structure. The advantage of this scheme is simple but it is difficult to achieve compression gain. Huffman encoding is a lossless coding scheme that produces



(a) Lossless-only audio encoder.



(b) Extended lossless coding on encoded bit stream.



(c) Lossless audio encoder with a base lossy CODEC.

Figure 13.7 Three types of lossless coder structures: (a) lossless-only audio encoder; (b) extended lossless coding on encoded bit stream; and (c) lossless audio encoder with a base lossy CODEC

Huffman codes from input symbols. Based on the statistic contents of the input sequence, the symbols are mapped to Huffman codes. Symbols that occur more frequently are coded with a shorter code, while symbols that occur less frequently are coded with longer codes. In average, this will reduce the total number of bits if some combinations of input sequences are more likely to appear than others.

Figure 13.7(b) further reduces the redundancy by using entropy coding. The example of this method can be found in MP3 and MPEG-2 AAC. In MP3, the second compression process, Huffman coding, is used at the end of the perceptual coding process. Huffman coding is extremely fast because it utilizes a lookup table for mapping quantized coefficients to possible Huffman codes. On average, an additional 20 % of compression can be achieved.

Figure 13.7(c) is a hierarchical structure of mixed core lossy coder and lossless coder. For a given PCM audio input, more advanced scalable lossless coding generates a bit stream that can be decoded to a bit-exact reproduction of input PCM audio. The example can be found in MPEG-4 AAC scalable lossless coding standards. In this coding scheme, the audio is first encoded with AAC, and then the residual error between the original audio and the AAC is encoded as shown in Figure 13.7(c). The resultant compressed bit stream has two rates: the lossy bit rate (same as encoded AAC bit stream) and the lossless bit rate. On the decoding side, the decoder may use the core lossy encoded bit stream to produce a lossy reproduction of the original audio with lower quality, or use full bit streams to produce the highest quality audio. A better signal fidelity is always resulted from higher rates. Refer to [8] for detailed information.

13.3 Multichannel Audio Coding

This section briefly introduces different audio CODECs, and uses MP3 as an example for relatively detailed description.

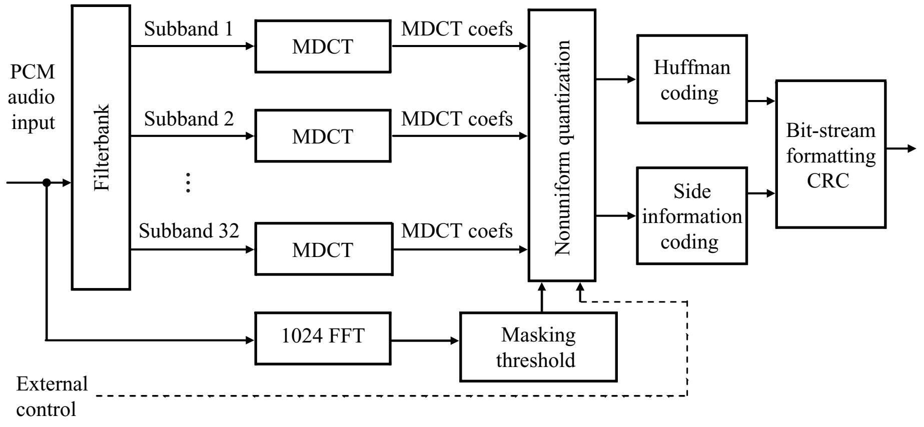


Figure 13.8 Block diagram of MP3 encoder

13.3.1 MP3

MP3 processes the audio data in 1152 samples per frame. This algorithm uses 32 subbands, each subband is critically sampled at 1/32 sampling rate, and 36 samples are buffered for MDCT block. Therefore, there are $32 \times 36 = 1152$ samples per frame (about 26 ms). The MP3 encoder is illustrated in Figure 13.8.

MP3 specifies two different MDCT block lengths: a long block of 18 samples and a short block of 6 samples. There is 50 % overlap between successive windows and so the window sizes are 36 and 12. The long block length allows better frequency resolution for audio signals with stationary characteristics, while the short block length provides better time resolution for transients. As discussed in Section 13.2.2, this window switching technique can reduce the pre- or postecho.

The 1152 samples per frame will result in 576 MDCT coefficients. These coefficients are quantized using psychoacoustics model that is calculated based on the 1024 FFT bins. In Figure 13.8, the control parameters are the sampling and bit rates as summarized in Table 13.1.

After quantization, the encoder arranges 576 quantized MDCT coefficients in the order of increasing frequency except for the short MDCT block mode. For short blocks, there are three sets of window values for a given frequency. Since higher energy audio components are concentrated in lower frequencies, the increasing frequency order moves the large values to the lower frequencies and small values to the higher frequencies. This is similar to a zigzag method used in image coding. The ordered stream is more favorable for Huffman coding.

Example 13.3: Figure 13.9 shows the amplitude distribution of MDCT coefficients. The data is obtained using the averaged absolute values of MDCT coefficients. There are 32 subbands and 18 samples in each subband. These coefficients are scaled by 32 768. It is clear that large coefficients are near DC frequency (indicated by subband number) and the coefficients with values close to 0 are at the higher frequencies.

Table 13.1 MP3 configurations for constant bit rate

Parameters	Configurations
Sampling rate (kHz)	48, 44.1, 32
Bit rate (kbit/s)	320, 256, 224, 192, 160, 128, 112, 96, 80
Compression rate	4.4, 5.5, 6.3, 7.4, 8.8, 11.0, 12.6, 14.7, 17.6

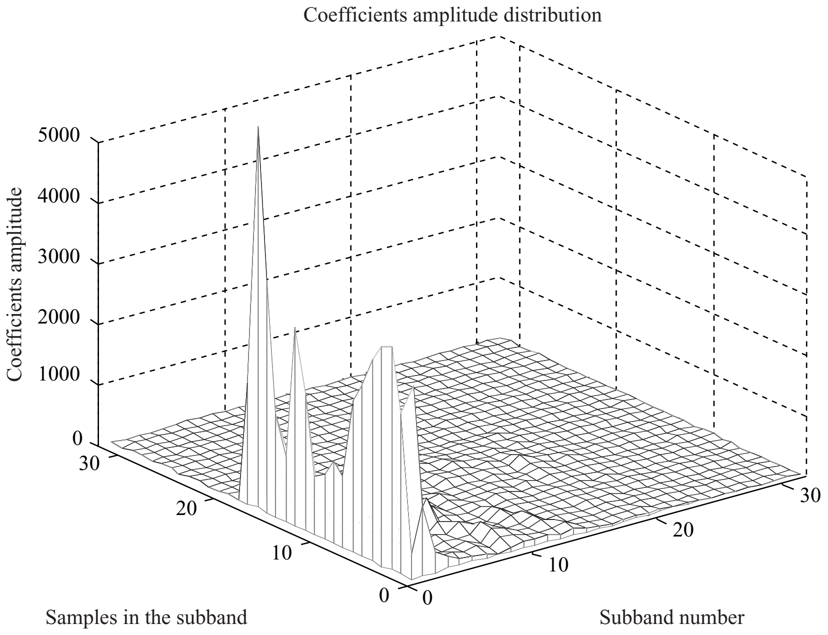


Figure 13.9 Amplitude distribution of MDCT coefficients

After ordering the coefficients, the frequency bins are divided into three regions: run-zeros, count-1, and big-values. Starting at the highest frequency, the encoder identifies the continuous run of all-zero values as one region of run-zeros. The run-zeros represent the frequency bins that have been removed by the encoder, and they should be filled with zeros by the decoder. The next region, count-1, can only be coded with the values 0, 1, or –1. Low-frequency components are represented by big values that are coded with the highest precision. The big-value region is divided into three subregions and coded using different Huffman tables. Therefore, each subregion is coded with different set of Huffman tables that match with the statistics of that region.

In MP3, there are 32 different Huffman tables for big values. These tables are predefined based on statistics suitable for compressing audio information. The side information specifies which table to use for decoding the current frame. The output from the Huffman decoder is 576 scaled frequency lines, represented with integer values.

13.3.2 Dolby AC-3

AC-3 is a high-quality, low-complexity, multichannel audio coding technique. It is the sound format used for digital televisions, digital versatile discs (DVDs), high-definition televisions, and digital cable and satellite transmissions.

Dolby digital provides six audio channels with the 5.1 format. The ‘5’ channels are left, center, right, left-surround, and right-surround. The ‘.1’ represents the low-frequency effects (LFE) channel for the subwoofer. The LFE channel is one-tenth of the bandwidth of other channels.

As shown in Figure 13.10, the AC-3 encoder uses the human auditory-masking and transform-coding techniques. AC-3 has the similar structure as shown in Figure 13.1, except that there is no lossless coding for the bit streams. AC-3 is a block-structured coder with typically 512 samples per block, and 256

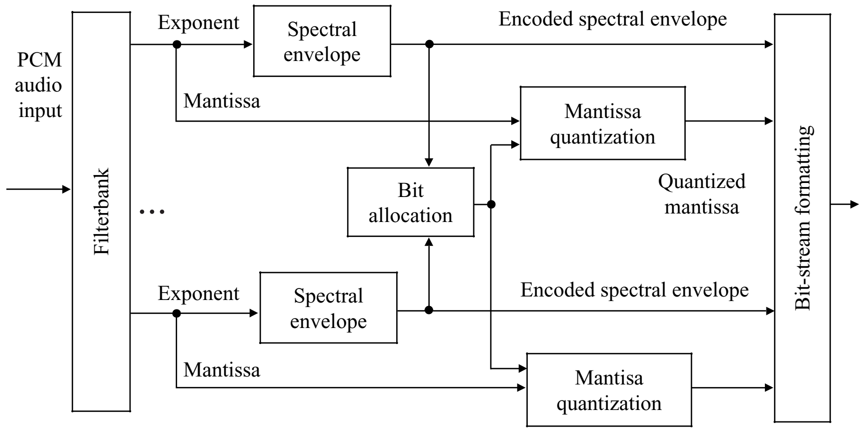


Figure 13.10 Block diagram of AC-3 encoder

samples per block may be used for better time resolution. For a block with 512 samples, 512 MDCT coefficients are calculated. These frequency-domain representations are decimated by a factor of 2 so that each block contains 256 coefficients. The MDCT coefficients are represented by floating-point format consisting of exponent and mantissa. These exponents are encoded as a coarse representation of the signal spectrum, which is referred as the spectral envelope.

These exponents are 5-bit values that indicate the number of leading zeros in the binary representation of a coefficient. The exponent acts as a scaling factor $2^{-\exp}$ for each mantissa. Exponent values range from 0 (for the largest coefficient values with no leading zero) to 24. Exponents for coefficients that have more than 24 leading zeroes are fixed at 24, and the corresponding mantissas are allowed to have leading zeros.

This spectral envelope is used by the core bit-allocation routine, which determines how many bits are used to encode each individual mantissa. The spectral envelope and the coarsely quantized mantissa for six audio blocks (total 1536 audio samples for 5.1 channels) are formatted into an AC-3 frame.

Decoding procedure is based on the inverse of the encoding process. The decoder unpacks various types of data such as the encoded spectral envelope and mantissas. The spectral envelope is decoded to produce the exponents. The bit-allocation routine is run and the results are used to unpack and dequantize the mantissas. The exponents and mantissas are transformed back to the time domain to produce the decoded PCM samples.

AC-3 can process 20-bit digital audio signals over a frequency range from 20 to 20 kHz (-3 dB at 3 Hz and 20.3 kHz). The LFE channel covers 20–120 Hz (-3 dB at 3 Hz and 121 Hz). It supports sampling rates of 32, 44.1, and 48 kHz. Data rates range from 32 kbit/s for a single mono channel to 640 kbit/s for special applications. The typical bit rate is 384 kbit/s.

AC-3 uses 256- or 512-point MDCT. Detailed procedures to implement the MDCT using a single $N/4$-point complex IFFT to calculate N-point IMDCT can be found in [1, 2].

13.3.3 MPEG-2 AAC

MPEG-2 AAC supports applications that do not request backward compatibility with the existing MPEG-1 stereo format. The AAC standard employs high-resolution filterbank, prediction techniques, and Huffman coding. The AAC standard offers high quality at lowest possible bit rates between 320 and 384 kbit/s for five channels. With sampling frequencies between 8 and 96 kHz and the number of channels from 1 to 48, this scheme is well prepared for future developments.

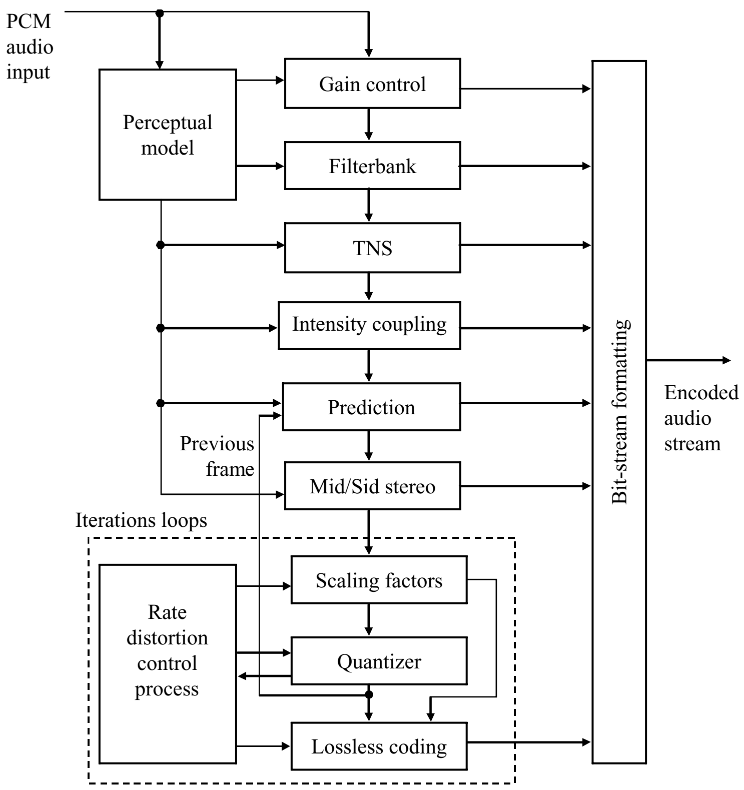


Figure 13.11 AAC structure diagram

AAC follows the same basic coding structure as MP3, such as high-resolution filterbank, nonuniform quantization, Huffman coding, and iteration loop structure. It improves MP3 in many areas using new coding tools [9]. The encoder structure is shown in Figure 13.11. The important differences between the AAC and its predecessor MP3 are summarized as follows:

Filterbank: MPEG-2 AAC uses MDCT with the increased window length. The MDCT supports block lengths of 2048 and 256 points that can be switched dynamically, while MP3 supports the block lengths of 1152 and 384 points. Compared to MP3, the length of the long block transform (2048) offers improved coding efficiency for stationary signals, and the short block length (256) provides optimized coding capabilities for transient signals.

Temporal noise shaping (*TNS*): TNS uses temporal masking technique to shape the distribution of quantization noise in time domain since the acoustic events just before the masking signal (premasking) and after this masking signal (postmasking) are not audible. The duration when premasking applies is short in the order of a few ms, whereas the postmasking is in the order of 50–200 ms. TNS filters the original spectrum and quantizes the filtered spectrum bins. This will lead to a temporally shaped distribution of quantization noise in the decoded audio signal. TNS is especially successful for the improvement of speech quality at low-bit rates.

Prediction: Prediction is commonly used in speech-coding algorithms. The frequency-domain prediction reduces redundancies of stationary signals by removing the redundancy between two successive coding frames.

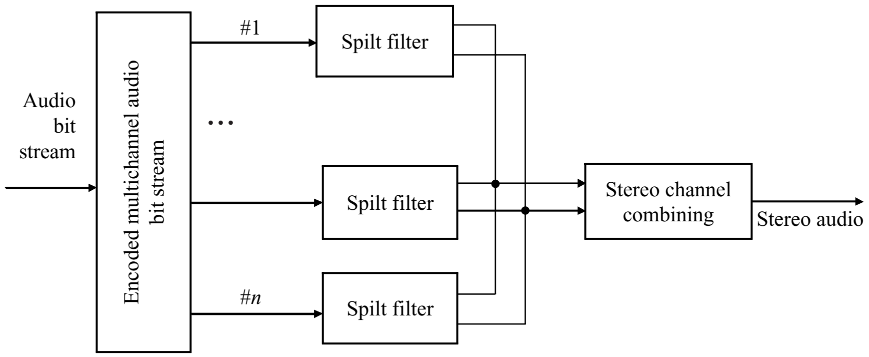


Figure 13.12 Conversion from multichannel to stereo channels

13.4 Connectivity Processing

Connectivity processing includes sampling-rate conversion and transcoding between two different codes; for example, to convert the 5.1-channel AC-3 code at 48 kHz to the 2-channel MP3 code at 44.1 kHz. Figure 13.12 shows a typical conversion structure from multichannel to stereo. The split filter controls the components going to the left and right channels. The simplest implementation of these filters could be a scaling factor. For example, if the original channel is the left channel in multichannel format, this signal may go to the left channel of the new stereo signal. For the center channel, 50 % go to the left and right channels.

Transcoding technologies [10] adjust the compression ratio of standard compressed bit stream. Comparing with decoding and then reencoding the media, transcoding achieves modest computation saving by skipping part of the compression operations, mainly the inverse and forward transforms. However, due to the complexity of mapping the encoded parameters from one coding standard to another, transcoding is not that easy to achieve.

13.5 Experiments and Program Examples

This section implements the MDCT, MP3 decoding, and preecho and postecho using MATLAB, C, or C55x programs.

13.5.1 Floating-Point Implementation of MDCT

Table 13.2 lists the floating-point C code used to implement direct MDCT and IMDCT for MP3. Given a PCM data file, this code will calculate the MDCT and IMDCT, and then compare the differences. Figure 13.13 shows the block diagram of the procedures that calculate MDCT. Even though it requires 50 % overlap, the final number of frequency coefficients is equal to the number of the original time samples.

The main program for the MDCT experiment is listed in Table 13.2. The MDCT and IMDCT functions used by the main program are listed in Tables 13.3 and 13.4, respectively.

In this experiment, the program initialization generates three tables. One is the sine window table `win`, the second is the `cos_enc` table used for MDCT, and the last one is the `cos_dec` table used for IMDCT. The original signal `input.pcm` is shown in Figure 13.14(a). The difference between the original input

Table 13.2 List of partial main program to test MDCT module

```
{
   mdct(pcm_data_in,mdct_enc16,FRAME); // Perform MDCT of N samples
   for (j = 0; j < M; j++)
   {
       pcm_data_in[j] = pcm_data_in[j+M];
   }
   fwrite(mdct_enc16, sizeof(short),(FRAME>>1),f_enc );
   inv_mdct(mdct_enc16,mdct_proc,FRAME);    // Inverse MDCT
   overlap(mdct_proc,prevblck,pcm_data_out);// Overlap addition
}
```

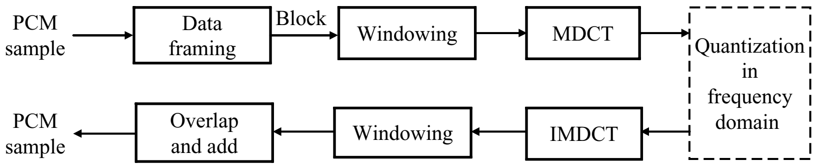


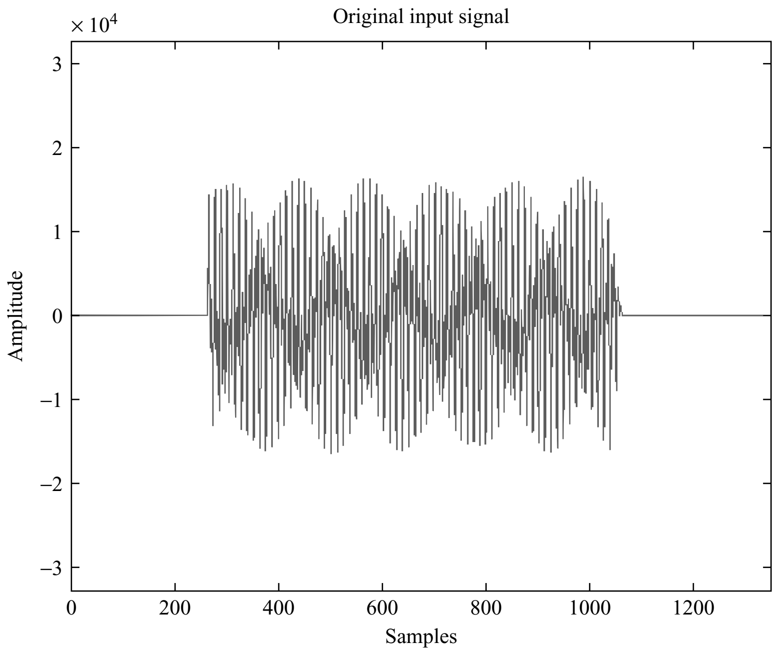
Figure 13.13 MDCT block processing

Table 13.3 C code for implementing MDCT

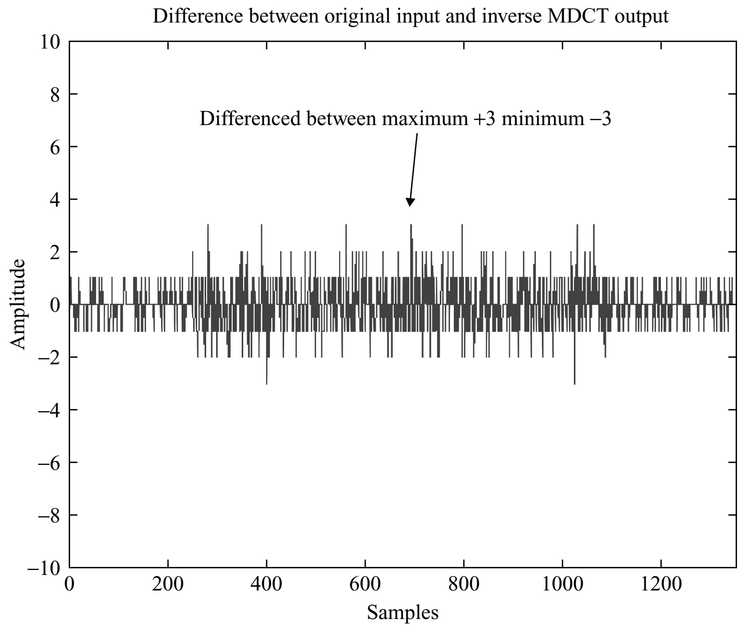
```
/* Function: Calculation of the direct MDCT */
void mdct(short *in, short *out, short N)
{
    short k,j;
    float acc0;
    for (j = 0; j < N / 2; j++)
    {
      for (acc0 = 0.0, k = 0; k < N; k++)
      {
        acc0 += win[k]*(float)in[k]*cos_enc[j][k];
      }
      out[j] = float2short(acc0);
    }
}
```

Table 13.4 C code for IMDCT

```
void inv_mdct(short *in, short *out, short N)
{
   short    j,k;
   float    acc0;
   for(j= 0;j<N;j++)
   {
       acc0 = 0.0;
       for(k=0;k<N/2;k++)
       {
         acc0 += (float)in[k]*cos_dec[((2*j+1+N/2)*(2*k+1))%(4*N)];
       }
       acc0 = acc0*win[j];
       out[j] = float2short(acc0);    // Convert to 16 bits
   }
}
```



(a) Original input signal.



(b) Amplitude differnce.

Figure 13.14 The original signal and amplitude difference between the original signal and the IMDCT output: (a) original input signal; (b) amplitude difference

Table 13.5 File listing for experiment `exp13.5.1_floatingPointMdct`

Files	Description
`floatPoint_mdctTest.c`	Main function for testing experiment
`floatPoint_mdct.c`	Direct and inverse MDCT functions
`floatPoint_mdct_init.c`	Generate window and coefficient tables
`floatPoint_mdct.h`	C header file
`floatPoint_mdct.prj`	DSP project file
`floatPoint_mdct.cmd`	DSP linker command file
`input.pcm`	Data file

signal `input.pcm` and the IMDCT output `mdctProc.pcm` is plotted in Figure 13.14(b). The maximum distortion is ±3. Table 13.5 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Start CCS, open the project, build, and load the program.

2. Check the experiment results with different frame sizes defined in `floatPoint_mdct.h`.

3. Further optimization of table to make the table size smaller. Change the two-dimensional array `cos_enc[ ][ ]` table to one-dimensional array `cos_enc[ ]` and try to reduce the table size by taking the advantage of periodic property.

13.5.2 Implementation of MDCT Using C55x Intrinsics

In this experiment, we replace all ‘float’ variables with ‘short’ in the floating-point C code, and use the C55x intrinsics for multiplication and other DSP-specific functions. The intrinsic implementation of MDCT is listed in Table 13.6. Comparing the code between Tables 13.3 and 13.6, the differences are using `mult_r` and `L_mac` functions to implement the multiplication and accumulation.

Table 13.6 C55x intrinsics implementation of MDCT

```
void mdct(short *in, short *out, short N)
{
    short k,j;
    long acc0;
    short temp16;
    for (j = 0; j < N / 2; j++)
    {
      acc0 = 0;
      for (k = 0; k < N; k++)
      {
          temp16 = mult_r(win[k],(in[k]));
          acc0 = L_mac(acc0,temp16,(cos_enc[j][k]));
      }
      acc0 = L_add(acc0, 0x8000L);
      out[j] = (short) (acc0>>SFT16);
    }
}
```

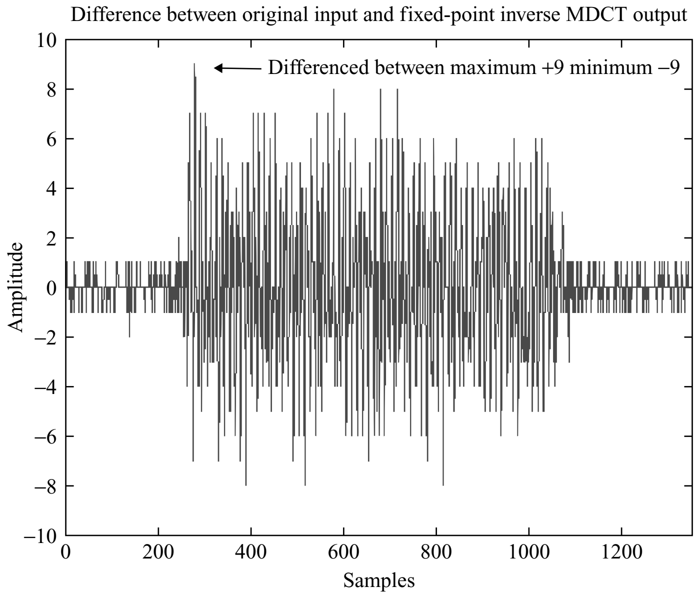


Figure 13.15 Amplitude distortion between the original signal and the C55x IMDCT output

Similar to the experiment given in Section 13.5.1, we compare the result with the original data. The distortion is shown in Figure 13.15. The distortion is higher in the 16-bit C55x implementation as compared with the 32-bit floating-point implementation. At the active segment, the maximum distortion in amplitude is ±9, which is higher than the floating-point implementation with the maximum distortion of ±3. The files used for this experiment are listed in Table 13.7.

Procedures of the experiment are listed as follows:

1. Start CCS, open the project, build, and load the program.
2. Run the experiment using different frame sizes (defined in `intrinsic_mdct.h`). Check the experiment results obtained with different frame sizes.
3. Convert the direct and reverse MDCT functions in fixed-point C code to C55 assembly code, and benchmark the cycles needed for both programs.

Table 13.7 File listing for experiment `exp13.5.2_intinsicMdct`

Files	Description
`intrinsic_mdctTest.c`	Main function for testing experiment
`intrinsic_mdct.c`	MDCT and IMDCT functions
`intrinsic_mdctInit.c`	Experiment initialization
`intrinsic_mdct.h`	C header file
`intrinsic_mdct.pjt`	DSP project file
`intrinsic_mdct.cmd`	C55x linker command file
`input.pcm`	Data file

Table 13.8 File listing for experiment `exp13.5.3_preEcho`

Files	Description
`floatPoint_preEchoTest.c`	Main program for testing experiment
`floatPoint_preEchoMdct.c`	Direct and inverse MDCT function
`floatPoint_preEchoInit.c`	Generate widowing and coefficient tables
`floatPoint_preEchoQnt.c`	Simulate log quantization
`floatPoint_preEcho.h`	C header function
`preEcho.pjt`	DSP project file
`preEcho.cmd`	DSP linker command file
`dtmf_digit2.pcm`	Data file

13.5.3 Experiments of Preecho Effects

Based on the experiment given in Section 13.5.2, we can add an MDCT coefficient quantization function to verify the preecho effect. We will compare the preecho effects between using 512- and 64-point MDCT block sizes with the original signal as shown in Figure 13.16(a). The files used for this experiment are listed in Table 13.8.

Procedures of the experiment are listed as follows:

1. Start CCS, open the project, build, and load the program.

2. Run the program and examine the resulting data file.

3. Convert this floating-point C experiment to C55x assembly functions. Rerun the assembly version of the experiment and compare the result with its floating-point experiment.

4. There are two constants `FRAME` and `NUM_QNT` defined in `floatPoint_preEcho.h` file. `FRAME` is the frame size used for MDCT block. `NUM_QNT` is the number of levels in log scale that the MDCT coefficients will be quantized. Change these two parameters to check the experiment results.

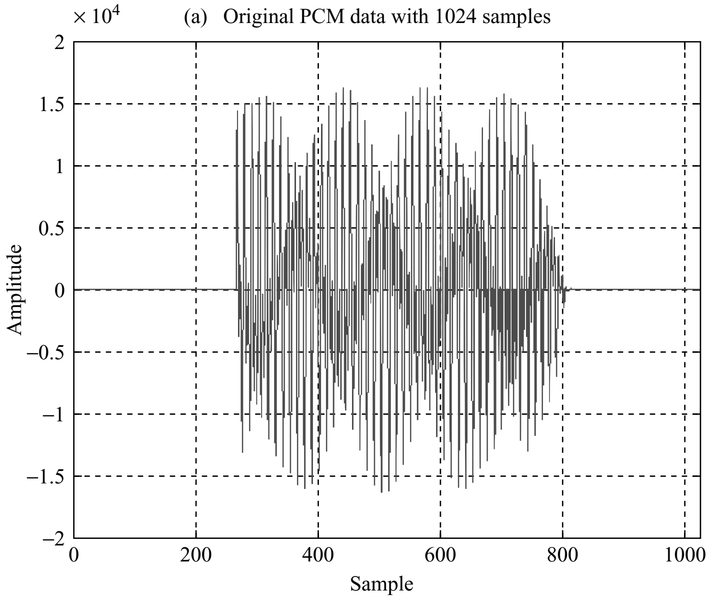
5. Perform 512-point MDCT/IMDCT with 50 % overlaps. The absolute values of MDCT coefficients are quantized using 16 steps in log scale with 16-bit maximum (32 767) corresponding to the highest step. The ripples before and after the signal segment are clearly shown in Figure 13.16(b) due to the quantization errors.

6. For comparison, perform 64-point MDCT/IMDCT with 50 % overlaps. With the increased time-domain resolution due to shorter length of MDCT, the ripples are reduced as shown in Figure 13.16(c).

13.5.4 Floating-Point C Implementation of MP3 Decoding

This section presents experiment of the MP3 decoding. The ISO reference source codes for MPEG-1 Layer I, II, and III in `dist10.zip` can be downloaded from Web sites. These files are listed in Table 13.9, where the file `musicout.c` is the main function that parsers the parameters and file I/O.

Unzip the file `dist10.zip`, and place the corresponding files under the `src`, `inc`, or `tables` folders as listed in Table 13.9. Using the provided workspace file `lsfDec.dsw` under Microsoft Visual C environment, we can easily compile the files to have executable code under `Debug` folder. After running the program, we will see the output as listed in Table 13.10.



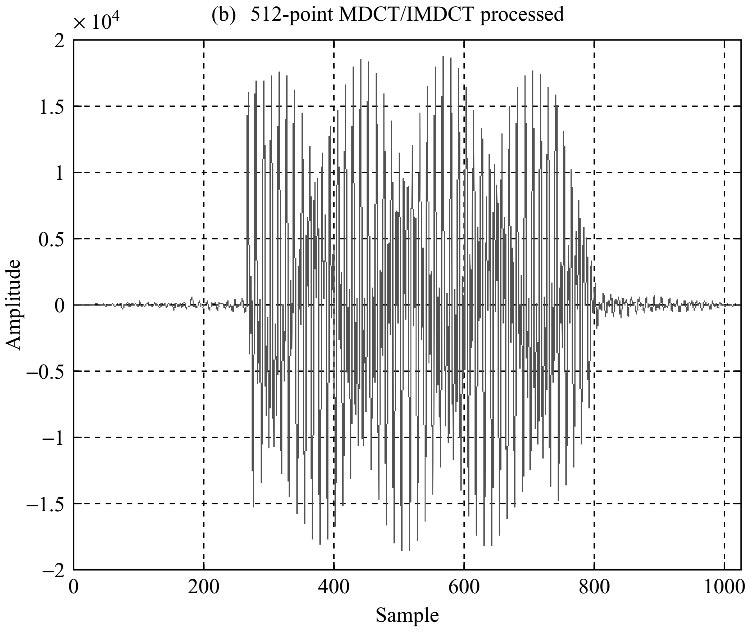


Figure 13.16 (a) Original PCM data with 1024 samples; (b) 512-point MDCT/IMDCT processed with 16-step quantization in log scale; and (c) 64-point MDCT/IMDCT processed with 16-step quantization in log scale

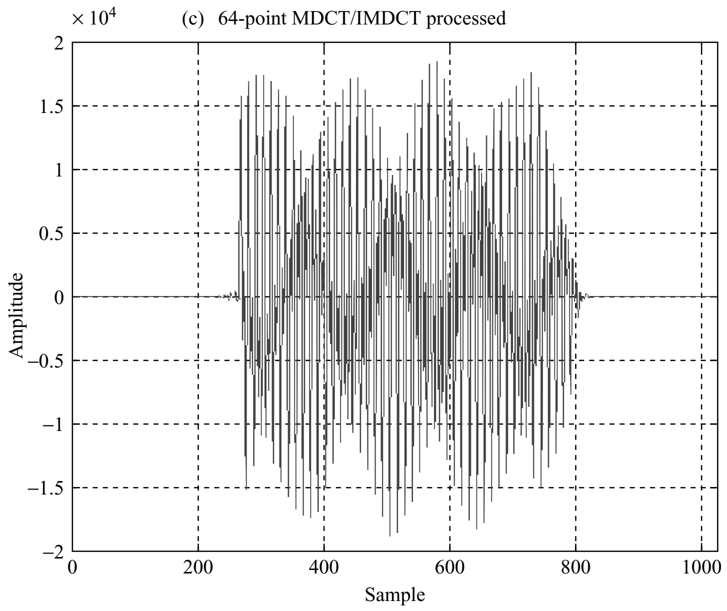


Figure 13.16 (*continued*)

Procedures of the experiment are listed as follows:

1. The executable code is located in the directory ..\debug\lsfDec.exe. Run the batch file mp3_dec.bat in ..\data folder. MP3 data file musicD_44p1_128bps.mp3 is read as input.

2. The output information from MP3 decoder is listed in Table 13.10. This shows it is an MP3 encoded at 44.1 kHz with the bit rate of 128 kbps in two-channel joint stereo format.

3. The output of the MP3 decoder musicD_44p1_128bps.mp3.dec is a stereo PCM file and its stereo data samples are arranged in the order of left, right, left, right, ... The 16-bit PCM data sample is in Motorola PCM format (MSB, LSB). Verify this PCM data as stereo audio at Motorola PCM format sampled at 44.1 kHz using MATLAB. Since MATLAB uses Intel PCM format (LSB, MSB), you need to change the data format from Motorola PCM to Intel PCM. The following C code can be used to convert Motorola PCM format to Intel PCM format.

```
FILE *fpIn,*fpOut;
short x;

fpIn  = fopen("musicD_44p1_128bps.mp3.dec", "rb");
fpOut = fopen("pcmFile.pcm", "wb");
while( fread(&x, sizeof(short), 1, fpIn) == 1)
{
   x = ((x>>8)&0xff)| ((x&0xff)<<8);
   fwrite(&x, sizeof(short), 1, fpOut);
}
fclose(fpIn);
fclose(fpOut);
```

Table 13.9 File listing for experiment `exp13.5.4_isoMp3Dec`

Directory	File	Description
`exp13.5.4_isoMp3Dec`	`lsfDec.dsw`	MP3 decoder workspace
src	`musicout.c`	Main file to parser the parameters and access all individual functions
	`common.c`	Common functions with sampling frequency conversion, bit-rate conversion, file I/O access
	`decode.c`	Bit-stream decoding, parameter decoding, sample dequantization, synthesis filters, decoder used functions
	`huffman.c`	Huffman decoding functions
	`ieeefloat.c`	data format conversion
	`portableio.c`	I/O functions
inc	`common.h`	Header file for `common.c`
	`decoder.h`	Header file for `decoder.h`
	`huffman.h`	Header file for `huffman.h`
	`ieeefloat.h`	Header file for `ieeefloat.h`
	`portableio.h`	Header file for `portableio.h`

tables	`1cb0 - 1cb6, 1th0 - 1th6`
	`2cb0 - 2cb6, 2th0 - 2th6`
	`absthr_0 - absthr_2`
	`alloc_0 - alloc_4`
	`dewindow, enwindow`
	`huffdec`
		Constant data

debug	`lsfDec.exe`	Executable files
data	`musicD_44p1_128bps.mp3`	Input file for decoder

The following MATLAB script separates the stereo PCM into two channels, the left audio channel and right audio channel, plots both channels, and plays each audio at 44.1 kHz rate.

```
fid=fopen('pcmFile.pcm', 'rb');
pcmData = fread(fid, 'int16');
len = length(pcmData);
leftChannel = pcmData(1:2:len-1);
rightChannel = pcmData(2:2:len);
subplot(2,1,1); plot(leftChannel);
subplot(2,1,2); plot(rightChannel);
soundsc(leftChannel, 44100);
soundsc(rightChannel, 44100);
```

4. Convert the floating-point C to fixed-point C and then to assembly program approach for the MP3 decoder using C55x DSK. The MP3 files to be decoded are stored in the computer. The MP3 files can be read by DSK via file I/O we learned from previous experiments.

5. Configure DSK AIC23 in stereo output in 48 kHz.

Table 13.10 Decoding information running of `mp3_dec.bat`

```
input file = '..\data\musicD_44p1_128bps.mp3'
output file = '..\ data\musicD_44p1_128bps.mp3.dec'
the bit stream file ..\data\musicD_44p1_128bps.mp3 is a BINARY file
HDR: s=FFF, id=1, l=3, ep=on, br=9, sf=0, pd=1, pr=0,
     m=1, js=2, c=0, o=0, e=0
alg.=MPEG-1, layer=III, tot bitrate=128, sfrq=44.1
mode=j-stereo, sblim=32, jsbd=8, ch=2
```

6. Use PC version decoder as reference to test the C55x decoder.

7. Play back the MP3 music files via the DSK decoder through a loudspeaker.

References

[1] Digital Audio Compression (AC-3, Enhanced AC-3) Standard, ETSI TS 102 366 V1.1.1 Feb. 2005.
[2] *ATSC Standard: Digital Audio Compression (AC-3)*, Revision A, Aug. 2001.
[3] S. Gadd and T. Lenart, *A Hardware Accelerated MP3 Decoder with Bluetooth Streaming Capabilities*, M.S. Thesis, Nov. 2001, http://www.es.lth.se/home/tlt/publications/masterthesis.pdf.
[4] T. Painter and A. Spanias, 'Perceptual coding of digital audio,' *Proc. IEEE*, vol. 88, no. 4, pp. 415–513, Apr. 2000.
[5] R. Raissi, 'The theory behind MP3,' Dec. 2002, http://rassol.com/cv/mp3.pdf.
[6] P. Noll and D. Pan, 'ISO/MPEG audio coding,' *Int. J High Speed Electron. Syst.*, vol. 8, no. 1, pp. 69–118, 1997.
[7] A. J. Ferreira, *Spectral Coding and Post-Processing of High Quality Audio*, Ph. D Thesis, University of Porto, 1998.
[8] ISO/IEC JTC1/SC29/WG11/N7018, *Scalable Lossless Coding*, Jan. 2005.
[9] K. Brandenburg, 'MP3 and AAC explained,' *Proc. AES 17th Int. Conf. High Qual. Audio Coding*, http://www.telos-systems.com/techtalk/aacpaper_2/AAC_3.pdf.
[10] J. Li, 'Embedded audio coding (EAC) with implicit auditory masking,' *Proc. ACM Multimedia 2002*, pp. 592–601, Dec. 2002, http://portal.acm.org/citation.cfm?doid=641126.
[11] ISO Reference Source Code of MPEG-1 Layer I, II and III: http://www.mp3-tech.org/programmer/sources/dist10.tgz.
[12] I. Dimkovic, 'Improved ISO AAC coder,' http://www.mp3-tech.org/programmer/docs/di042001.pdf.
[13] Y. Wang, L. Yaroslavsky, M. Vilermo, and M. Vaananen, 'Some peculiar properties of the MDCT,' *Proc. 5th Int. Conf. Signal Process.*, pp. 61–64, 2000.
[14] Y. Wang and M. Vilermo, 'The modified discrete cosine transform: Its implications for audio coding and error concealment,' *Proc. AES 22nd Int. Conf. Virtual Synth. Entertain. Audio,* pp. 223–232, 2002, http://www.comp.nus.edu.sg/∼wangye/papers/AES/00027_aes22.pdf.
[15] K. Brandenburg and H. Popp, 'An introduction to MPEG layer-3,' http://www.mp3-tech.org/programmer/docs/trev_283-popp.pdf.

Exercises

1. Draw a curve of masking threshold based on psychoacoustics experiment. Using MATLAB plays back a 1 kHz tone (masker) at 60 dB plus one of the tones listed in the following table. Since the levels of these bark band tones are changing, pick up the one you just cannot hear as the masking threshold. You may also try to raise the masker tone by 20 dB (to 80 dB) to see if the masking threshold is still the same as at 60 dB. The published bark band centers in Hz are listed in Table 13.11 [4].

Table 13.11 Summary of center frequencies of bark bands in Hz

Band no.	Center frequency	Band no.	Center frequency	Band no.	Center frequency	Band no.	Center frequency
1	50	7	700	13	1850	19	4800
2	150	8	840	14	2150	20	5800
3	250	9	1000	15	2500	21	7000
4	350	10	1175	16	2900	22	8500
5	450	11	1370	17	3400	23	10 500
6	570	12	1600	18	4000	24	13 500

2. Given a 36-point MDCT used in MP3, calculate the minimum number of multiplication and addition needed for calculating the direct MDCT coefficients. Also, compare this number with three individual 18-point MDCT blocks.

3. For preecho effect, do you think that higher resolution of quantization will also resolve the problem? Use the experiment given in Section 13.5.3 as reference to calculate 512-point MDCT/IMDCT with 50 % overlap, and the MDCT absolute coefficients are quantized using 64 steps in log scales. Verify that with the increased quantization signal-to-noise ratio, the amplitude of ripples should be smaller as compared with the experiment given in Section 13.5.3, where the MDCT coefficients are quantized using 16 steps.

4. Section 13.5.1 uses three tables, `win[ ]`, `cos_enc[ ][ ]`, and `cos_dec[ ]`. These tables become larger with the increase of frame size. Modify the experiment such that the MDCT and IMDCT will compute these table values at run time using cosine and sine functions. To improve run-time efficiency, implement cosine and sine functions in assembly for this experiment. Verify the implementation using frame sizes of 64, 256, and 512.

5. In experiment given in Section 13.5.3, we did not use table `cos_enc[ ][ ]` as in Section 13.5.1. Instead, we directly use the rum-time function `cosCoef = ((float)cos((PI/(2*N))*(2*k+1+N/2)*(2*j+1))/(N/4))` to generate the coefficients. This is not efficient but we have to do so due to the memory limitation for 512-point MDCT. If we rearrange the data in the table, the table access may become simple:

   (a) Reduce the table size N by $N/2$ of `cos_enc[ ][ ]` to a reasonable number.

   (b) Convert this two-dimensional table `cos_enc[ ][ ]` to one dimensional.

6. If MP3 encoded bit stream is 128 kbit/s using constant coding scheme and the sampling rate is 48 kHz, calculate the compression ratio. If the sampling rate is 16 kHz and the bit rate remains the same, calculate the compression ratio. For both cases, assume input is stereo.

7. The downloaded MPEG-1 Layer I, II, and III file `dist10.zip` also contains the MP3 encoder source code. Using the experiment given in Section 13.5.4 as reference, compile the MP3 encoder program, and encode the decoded linear stereo PCM data from Section 13.5.4 into MP3 bit stream.

14 Channel Coding Techniques

Channel coding is very important in digital communications. Many communication systems use the forward error-correction (FEC) coding to detect and reduce transmission errors. In addition to FEC codes, cyclic redundancy check (CRC) is also widely used to verify the data correctness at the receiver. This chapter introduces basic channel coding techniques using convolutional codes, Viterbi decoding, Reed–Solomon codes, and CRC.

14.1 Introduction

An error-correction code (ECC) protects data against noises or surface defects in communication or disk-storage systems. The ECC allows decoder to detect and possibly correct transmission errors directly at the receiver. An ECC adds redundancy that is a small fraction of the actual data for transmission. The ECC achieves the coding gain by decreasing the required bit energy over noise for acceptable transmission quality. The achieved coding gain can be used to save bandwidth or reduce power requirements.

Channel coding techniques are usually classified into two categories: block codes and convolutional codes. In block codes such as Reed–Solomon codes, the data stream is divided into consecutive blocks of certain length. The redundant bits are added to these data blocks for transmission, and the receiver decodes the data block by block. In convolutional codes, redundant bits are added continuously to the coder output. The values of these bits are determined by a combination of preceding information bits. The convolutional code is commonly used as trellis-coded modulation in modem applications. Figure 14.1 shows a typical transmission system with channel coding.

The channel encoder introduces redundancy for error detection, error correction, or both, thus increases data bit rate. If k is the number of information bits and n is the number of encoder output bits, the data rate of the encoder is

$$d_{\mathrm{b}} = (n/k)d_i, \tag{14.1}$$

where d_i is the source (information) data rate. The ratio $R_c = k/n$ is defined as the code rate. Channel coding reduces the received energy per symbol. However, an increased symbol data rate also results in an increased modulation bandwidth with the factor n/k.

The performance of channel coding can be improved through soft decoding, in which the channel decoder uses information on previously received data. This can be achieved by using long constraint lengths for convolutional codes, or long codewords for block codes. However, this results in undesired longer delay in buffering data and highly complex channel decoders. Thus, the viable coding schemes must consider the issues of the delay, complexity, and technological limitations.

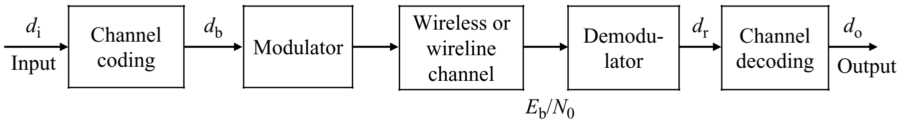


Figure 14.1 A typical transmission system with channel coding

A simple technique called automatic request for retransmission scheme can be used in error control applications. In this method, the transmitter stops and waits until a correct receipt of the data is acknowledged, or a request for retransmission is received on the backward channel. The request for retransmission can be simply based on a CRC method.

14.2 Block Codes

A block code [2] consists of a set of fixed-length vectors called codewords. The length n of a codeword is the number of elements in the vector. The elements of a codeword are selected from an alphabet of q elements. When the alphabet consists of two elements 0 and 1, the code is a binary code, and these elements are called bits. The Galois field (*GF*) arithmetic is a finite state field, and $GF(q)$ is Galois field of order q. The finite field with two elements is denoted as $GF(2)$. When the elements of a codeword are selected from an alphabet having q elements where $q > 2$, the code is a nonbinary code. When q is a power of 2 (i.e., $q = 2^m$ where m is a positive integer), each q-ary element has an equivalent binary representation consisting of m bits. Thus, a nonbinary code of block length N can be mapped into a binary code of block length $n = mN$. This nonbinary finite field is denoted by GF(q) or GF(2^m) where $m > 1$.

There are 2^n possible codewords in a binary block code of length n. From these 2^n codewords, we may select $M = 2^k$ codewords ($k < n$) to form a code. Thus, a block of k information bits is mapped into a codeword of length n selected from the set of $M = 2^k$ codewords. The resulting block code is called an (n, k) code, and the code rate is $R_c = k/n$. An (n, k) Reed–Solomon (RS) code is capable of correcting t symbol errors where $t = (n - k)/2$. The block length of standard Reed–Soloman codes is $n = q - 1$, and the code rate is $R_c = k/n$. Examples of codes with different values of m, q, n, and t are summarized in Table 14.1.

The family of linear block codes is illustrated in Figure 14.2. The cyclic, BCH (Bose–Chaudhuri–Hocquenghem), Hamming, and Reed–Solomon codes are special classes of linear block codes:

Linear block codes [4]: Suppose C_1 and C_2 are two codewords in an (n, k) block code. Let α_1 and α_2 be any two elements selected from the alphabet. The code is linear if and only if $\alpha_1 C_1 + \alpha_2 C_2$ is also a codeword.

Table 14.1 Examples of Reed–Solomon codes

M	$q = 2^m$	n (Block length)	k (Information symbols)	$2t$(Parity symbols)	$R_c = k/n$	Applications
6	64	63	47	6	47/63	U.S. cellular digital packet data
8	256	219	201	22	201/219	Intelsat IESS310
8	256	28	24	4	24/28	Compact CD C1 encoder
8	256	32	28	4	28/32	Compact CD C2 encoder
8	256	255	233	22	233/255	Infrared wireless audio

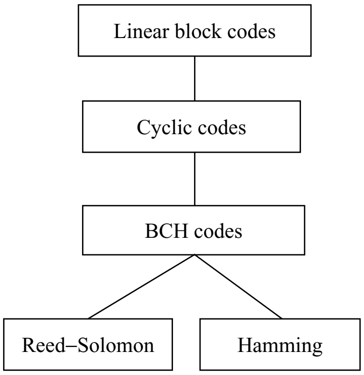


Figure 14.2 Relationship of different block codes

Cyclic codes: A cyclic code is a linear block code with the property that a cyclic shift of a codeword is also a codeword. For example, if $C = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0]$ is a codeword of a cyclic code, then $[c_{n-2}, \ldots, c_1, c_0, c_{n-1}]$ obtained from the cyclic shift of the element C is also a codeword. This cyclic property means cyclic code possesses a considerable amount of structures that can be exploited in the encoding and decoding operations. An (n, k) cyclic code is completely specified by the following generator polynomial:

$$g(x) = 1 + g_1 x + \cdots + g_{n-k-1} x^{n-k-1} + g_{n-k} x^{n-k}. \tag{14.2}$$

BCH codes [5]: The BCH code is a class of cyclic codes whose generator polynomial is the product of distinct minimal polynomials corresponding to $\alpha, \alpha^2, \ldots, \alpha^{2t}$, where $\alpha \in GF(2^m)$ is the root of the primitive polynomial $p(x)$. The most important and common class of nonbinary BCH codes is the Reed–Solomon codes. Figure 14.3 shows a Reed–Solomon codeword in which the data is unchanged while the parity bits are suffixed to the data bits. The Reed–Solomon codes are the most commonly used for practical applications.

Reed–Solomon codes use nonbinary fields $GF(2^m)$. These fields have more than two elements and are extensions of the binary field $GF(2) = \{0, 1\}$. The additional elements in the extension field use a new symbol α to represent the elements other than 0 and 1. Each nonzero element can be represented by a power of α. Some important Galois field properties are:

1. An element with order $(q - 1)$ in $GF(q)$ is called a primitive element. Each field contains at least one primitive element α. All nonzero elements in $GF(q)$ can be represented as $(q - 1)$ consecutive powers of a primitive element α.

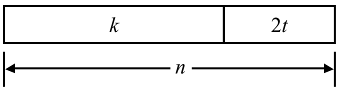


Figure 14.3 Codeword of Reed–Solomon codes

Table 14.2 Example of construction of $GF(8)$

Exponential representation	Polynomial representation	3-bit symbol representation
0	0	0 0 0
α^0, α^7	1	0 0 1
α	α	0 1 0
α^2	α^2	1 0 0
α^3	$\alpha + 1$	0 1 1
α^4	$\alpha^2 + \alpha$	1 1 0
α^5	$\alpha^3 + \alpha^2 = \alpha^2 + \alpha + 1$	1 1 1
α^6	$\alpha^2 + 1$	1 0 1

2. Polynomials over Galois fields, $GF(q)[x]$, are the collection of all polynomials $\alpha_0 \oplus \alpha_1 x \oplus \alpha_2 x^2 \oplus \cdots \oplus \alpha_n x^n$ of arbitrary degree with coefficients $\{\alpha_j\}$ in the finite field $GF(q)$.

3. The operator $\oplus$ is an exclusive-OR (XOR) operation. The multiplication of two m-bit numbers under $GF(2^m)$ can become exponential addition with modular $(2^m - 1)$.

Example 14.1: Construct $GF(8)$ with $p(x) = x^3 + x + 1$ being the primitive in $GF(8)[x]$. Let α be the root of $p(x)|_{x=\alpha} = \alpha^3 + \alpha + 1 = 0 \Rightarrow \alpha^3 = \alpha + 1$. The exponential and 3-bit symbol representations are shown in Table 14.2.

Using the equation $\alpha^3 = \alpha + 1$ and the exponential and symbol representations listed in Table 14.2, we can easily calculate the following operations:

1. *Addition*: $\alpha^5 + \alpha^3 = \alpha^2(\alpha + 1) + (\alpha + 1) = \alpha^2 + (\alpha^3 + \alpha + 1) = \alpha^2$. If we use 3-bit symbol representation, the addition can be also derived as $\alpha^5 + \alpha^3 = (111)_2 + (011)_2 = (100)_2 = \alpha^2$.

2. *Multiplication*: $\alpha^5\alpha^3 = \alpha^{8 \bmod (7)} = \alpha$, $\alpha^5(\alpha^3 + 1) = \alpha^5(\alpha) = \alpha^6$ and $\alpha^5(\alpha^3 + 1) = \alpha^8 + \alpha^5 = \alpha^1 + \alpha^5 = (010)_2 + (111)_2 = (101)_2 = \alpha^6$.

14.2.1 Reed–Solomon Codes

A Reed–Solomon code is a block sequence of finite field $GF(2^m)$ with 2^m binary symbols, where m is the number of bits per symbol. An (n, k) Reed–Solomon code with symbols from $GF(2^m)$ has the following parameters [5]:

1. $n \le 2^m - 1$: codeword length in symbols;

2. $k = n - 2t$: number of information symbols; and

3. $n - k = 2t$: number of parity symbols.

Reed–Solomon encoder

The straightforward method of obtaining the remainder from the division process by the polynomial $g(x)$ is to connect a shift register according to $g(x)$ as shown in Figure 14.4 [4]. In the figure, the symbol $\oplus$ represents an XOR operation of two m-bit numbers, the symbol $\otimes$ represents a multiplication of two

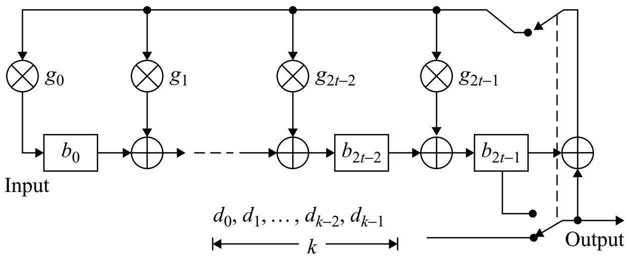


Figure 14.4 Reed–Solomon code generator

m-bit numbers under $GF(2^m)$, and each m-bit register contains an m-bit number denoted by b_i. The raw information polynomial is expressed as

$$d(x) = c_{2t}x^{2t} + c_{2t+1}x^{2t+1} + \cdots + c_{n-2}x^{n-2} + c_{n-1}x^{n-1}. \tag{14.3}$$

The parity polynomial is expressed as

$$p_t(x) = c_0 + c_1x^1 \cdots + c_{2t-1}x^{2t-1}. \tag{14.4}$$

The generator polynomial for t error corrections is

$$g(x) = (x+\alpha)(x+\alpha^2)\cdots(x+\alpha^{2t}) = \sum_{i=0}^{2t} g_i x^i. \tag{14.5}$$

A common method for encoding a cyclic code is to derive $p(x)$ by dividing $d(x)$ with $g(x)$. This yields an irrelevant quotient polynomial $q(x)$ and an import remainder polynomial $r(x)$ as

$$d(x) = g(x)q(x) + r(x). \tag{14.6}$$

The codeword polynomial can be expressed as

$$c(x) = p_t(x) + g(x)q(x) + r(x). \tag{14.7}$$

If we define $p_t(x) = -r(x)$, then

$$c(x) = g(x)q(x). \tag{14.8}$$

Equation (14.8) ensures that the codeword is always an integer of generator polynomial $g(x)$. Figure 14.4 shows the block diagram of a basic Reed–Solomon encoder. The data sequence $d(x)$ is shifted through the encoder circuits beginning with symbol d_{k-1}. After shifting d_0 into the circuit, the $2t$ redundant symbols are taken from the shift register stages, or these symbols are shifted out when the switch is changed to upper position.

Example 14.2: Use the shift registers in Figure 14.4 to generate the parity data for a Reed–Solomon (255, 239) code. The primitive polynomial is $p(x) = 1 + x^2 + x^3 + x^4 + x^8$. The resulted generation polynomial coefficients are $\{g_0, g_1, \ldots, g_{16}\} = \{$`0x4f`, `0x2c`, `0x51`, `0x64`, `0x31`, `0xb7`, `0x38`, `0x11`, `0xe8`, `0xbb`, `0x7e`, `0x68`, `0x1f`, `0x67`, `0x34`, `0x76`, `0x01`$\}$. The input data is listed in Table 14.3.

Table 14.3 The encoded Reed–Solomon (255, 239) codeword in Hex format

data	0x00:	33,6c,c1,c2,bf,c8,ad,fe,0b,e4,59,fa,17,c0,c5,b6
data	0x10:	e3,5c,f1,32,6f,b8,dd,6e,bb,d4,89,6a,c7,b0,f5,26
data	0x20:	93,4c,21,a2,1f,a8,0d,de,6b,c4,b9,da,77,a0,25,96
data	0x30:	43,3c,51,12,cf,98,3d,4e,1b,b4,e9,4a,27,90,55,06
data	0x40:	f3,2c,81,82,7f,88,6d,be,cb,a4,19,ba,d7,80,85,76
data	0x50:	a3,1c,b1,f2,2f,78,9d,2e,7b,94,49,2a,87,70,b5,e6
data	0x60:	53,0c,e1,62,df,68,cd,9e,2b,84,79,9a,37,60,e5,56
data	0x70:	03,fc,11,d2,8f,58,fd,0e,db,74,a9,0a,e7,50,15,c6
data	0x80:	b3,ec,41,42,3f,48,2d,7e,8b,64,d9,7a,97,40,45,36
data	0x90:	63,dc,71,b2,ef,38,5d,ee,3b,54,09,ea,47,30,75,a6
data	0xa0:	13,cc,a1,22,9f,28,8d,5e,eb,44,39,5a,f7,20,a5,16
data	0xb0:	c3,bc,d1,92,4f,18,bd,ce,9b,34,69,ca,a7,10,d5,86
data	0xc0:	73,ac,01,02,ff,08,ed,3e,4b,24,99,3a,57,00,05,f6
data	0xd0:	23,9c,31,72,af,f8,1d,ae,fb,14,c9,aa,07,f0,35,66
data	0xe0:	d3,8c,61,e2,5f,e8,4d,1e,ab,04,f9,1a,b7,e0,65,a2
parity	0xf0:	a2,47,a1,b7,a4,f1,0c,65,91,13,b7,e7,d6,f3,0e,cc

In Figure 14.4, the multiplication operation of $g_i b_i$ is over $GF(256)$. The easy way to do multiplication over $GF(256)$ is to use exponential representation. For example, the symbol representation of g_0, `0x4f`, can be converted to exponential representation as α^{0x88}. By doing the same conversion for register byte b_i, the multiplication over Galois field becomes the addition of the exponents (see Section 14.4.2 for details). In that example, the exponential representation of generation polynomial coefficients are $\{g_0, g_1, \ldots, g_{16}\} = \{$`0x88`, `0xf0`, `0xd0`, `0xc3`, `0xb5`, `0x9e`, `0xc9`, `0x64`, `0x0b`, `0x53`, `0xa7`, `0x6b`, `0x71`, `0x6e`, `0x6a`, `0x79`, `0x00`$\}$.

The addition illustrated in Figure 14.4 is an XOR operation. In order to perform the XOR operation, we convert $g_i b_i$ from exponential format to symbol format. The table used for conversion is given in `\example14.2\rs_enc.dat`. The encoded data is shown in Table 14.3.

Reed–Solomon decoder

A typical Reed–Solomon decoder includes five distinct algorithms as shown in Figure 14.5. The first algorithm calculates $2t$ partial syndromes. The Berlekamp–Massev algorithm calculates the error locator polynomial. The Forney algorithm computes the error magnitudes. The Chien search finds the error location. Finally, we know both the error locations in the received codeword and the magnitude of error at each location.

The decoder first constructs the syndrome polynomial by evaluating the received codeword at all the roots of the generator polynomial $g(x)$. From Equation (14.8), all the roots for $g(x)$ will be the roots for codeword polynomial $c(x)$ if there is no transmission error. If there is error in the received codeword,

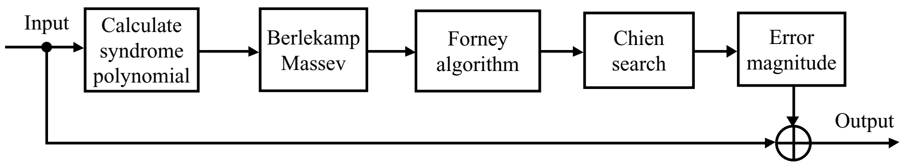


Figure 14.5 A block diagram of the Reed–Solomon decoder

received codeword polynomial $r(x)$ can be expressed as

$$r(x) = c(x) + e(x), \tag{14.9}$$

where $e(x)$ is the error polynomial. The syndrome polynomial $s(x)$ is obtained by evaluating the received word at each root of the generator polynomial as follows:

$$s_j(x) = r(x)\,|_{x=\alpha^j}\ , \qquad j = 1, \ldots, 2t. \tag{14.10}$$

Once the syndrome polynomial has been constructed, it can be used to calculate the error locator polynomial using the Berlekamp–Massev algorithm if they are not all zeros. This leads to finding the error location by Chien search.

Example 14.3: Using the encoded data shown in Table 14.3, randomly change eight locations as shown in Table 14.4 and assume this is the data received from the channel. Following the decoding procedures illustrated in Figure 14.5, these eight errors (in boldface) are eventually corrected step by step as presented in the following procedures.

If there is no error, the 16 syndrome polynomials defined in Equation (14.9) will equal to zero. The syndrome polynomial evaluates the received data with roots of generation polynomial $g(x)$. In this example, the syndrome functions are not equal to zero and they are listed as {`0x05`, `0x23`, `0x5a`, `0x53`, `0xd1`, `0xb5`, `0x6b`, `0x01`, `0x5e`, `0x76`, `0x80`, `0xeb`, `0x49`, `0x1f`, `0x4f`, `0x35`}. This means there are errors in the received data. Berlekamp's iterative algorithm finds the error locator polynomial, and eventually finds the error locations via Chien search as {`0x45`, `0x52`, `0x70`, `0x76`, `0x79`, `0xcf`, `0xdd`, `0xe7`}. Use Forney algorithm to compute the error magnitudes with respect to the above locations; the error magnitudes are calculated as {`0x8a`, `0x95`, `0x5b`, `0x8e`, `0x29`, `0xf4`, `0xd0`, `0x67`}. Once the error locations and amplitudes are found, the erased bytes can be corrected by XOR operation of the error amplitude and received data as shown in Table 14.5.

MATLAB provides functions for the Reed–Solomon encoder `rsenc( )`, decoder `rsdec( )`, generator polynomial of Reed–Solomon code `rsgenpoly( )`, and Galois field array creator `gf( )`. Section 14.4.1 will present an experiment to encode and decode Reed–Solomon codes using these functions.

Table 14.4 The received codewords

data:	0x00:	33,6c,c1,c2,bf,c8,ad,fe,0b,e4,59,fa,17,c0,c5,b6
data:	0x10:	e3,5c,f1,32,6f,b8,dd,6e,bb,d4,89,6a,c7,b0,f5,26
data:	0x20:	93,4c,21,a2,1f,a8, d,de,6b,c4,b9,da,77,a0,25,96
data:	0x30:	43,3c,51,12,cf,98,3d,4e,1b,b4,e9,4a,27,90,55,06
data:	0x40:	f3,2c,81,82,7f,**02**,6d,be,cb,a4,19,ba,d7,80,85,76
data:	0x50:	a3,1c,**24**,f2,2f,78,9d,2e,7b,94,49,2a,87,70,b5,e6
data:	0x60:	53,0c,e1,62,df,68,cd,9e,2b,84,79,9a,37,60,e5,56
data:	0x70:	**58**,fc,11,d2,8f,58,**73**,0e,db,**5d**,a9, a,e7,50,15,c6
data:	0x80:	b3,ec,41,42,3f,48,2d,7e,8b,64,d9,7a,97,40,45,36
data:	0x90:	63,dc,71,b2,ef,38,5d,ee,3b,54,09,ea,47,30,75,a6
data:	0xa0:	13,cc,a1,22,9f,28,8d,5e,eb,44,39,5a,f7,20,a5,16
data:	0xb0:	c3,bc,d1,92,4f,18,bd,ce,9b,34,69,ca,a7,10,d5,86
data:	0xc0:	73,ac,01,02,ff,08,ed,3e,4b,24,99,3a,57,00,05,**02**
data:	0xd0:	23,9c,31,72,af,f8,1d,ae,fb,14,c9,aa,07,**20**,35,66
data:	0xe0:	d3,8c,61,e2,5f,e8,4d,**79**,ab,04,f9,1a,b7,e0,65,a2
parity:	0xf0:	a2,47,a1,b7,a4,f1,0c,65,91,13,b7,e7,d6,f3,0e,cc

Table 14.5 Error-correction simulation results

Number	Error location	Send	Receive	Error amplitude	Correction	Output
7	0x45	0x88	0x02	0x8a	0x8a XOR 0x02	0x88
6	0x52	0xb1	0x24	0x95	0x95 XOR 0x24	0xb1
5	0x70	0x03	0x58	0x5b	0x5b XOR 0x58	0x03
4	0x76	0xfd	0x73	0x8e	0x8e XOR 0x73	0xfd
3	0x79	0x74	0x5d	0x29	0x29 XOR 0x5d	0x74
2	0xcf	0xf6	0x02	0xf4	0xf4 XOR 0x02	0xf6
1	0xdd	0xf0	0x20	0xD0	0xd0 XOR 0x20	0xf0
0	0xe7	0x1e	0x79	0x67	0x67 XOR 0x79	0x1e

14.2.2 Applications of Reed–Solomon Codes

This section briefly introduces two applications of Reed–Solomon codes in media storage and wireless transmission.

Compact disc

In compact disc (CD), the audio signal is sampled with 16-bit A/D converter and these samples are split into two 8-bit words called symbols. This technique is called the cross interleave Reed–Solomon error-correction code (CIRC). This method can deal with both random and burst errors. The principle of CIRC is shown in Figure 14.6. The following steps are performed from encoder to decoder [7]:

1. Twelve 16-bit samples (or 24 8-bit symbols) are applied to the scrambling, symbol delay, and Reed–Solomon encoder for adding four parity symbols. The output is 28 symbols.

2. These 28 symbols are then applied to different delay lines with unique delays.

3. The second Reed–Solomon encoder adds another four parity bytes to form a 32-symbol block.

4. On play back, the decoding circuit restores the original 16-bit samples and sends them to the D/A converter.

This technique results in the maximum correctable burst error of length 4000 bits. On average, 4 bits are recorded for every three data bits [7].

Infrared wireless audio

In this application, the block size of Reed–Solomon codes is 255. One ECC superframe consists of two ECC subframes. $GF(2^8)$ is used, and the unit of the symbol is byte. The information data has 239 bytes and the parity data has 16 bytes. The correctable errors are of 8 bytes. The block diagram of ECC is shown in Figure 14.7 [8].

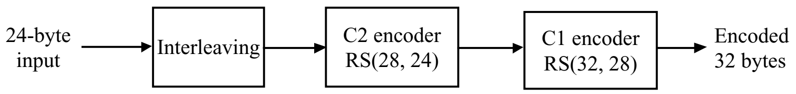


Figure 14.6 Compact disc encoding

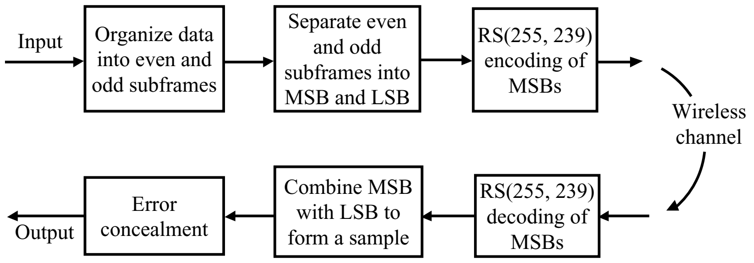

Input
Organize data into even and odd subframes
Separate even and odd subframes into MSB and LSB
RS(255, 239) encoding of MSBs
Wireless channel
RS(255, 239) decoding of MSBs
Combine MSB with LSB to form a sample
Error concealment
Output

Figure 14.7 Reed–Solomon coding in wireless audio applications

This example only protects the MSB (8-bit). The separation of even and odd subframes enables the receiver to interpolate the uncorrectable frame of data in the error concealment module if that frame is corrupted.

14.2.3 Cyclic Redundant Codes

The CRC is an intelligent alternative for block checksum. It is also a special case of the block codes. The CRC is calculated by dividing a block of bit string by a generator polynomial. The value of the CRC is the remainder, which is 1 bit shorter than the generator polynomial. It can be implemented using shift registers and XOR-operations in both hardware and software.

Most communication systems implement the CRC using the shift register to detect transmission errors. For example, a 7-bit CRC uses the following polynomial generator:

$$b_{\text{CRC}} = 1 \oplus x \oplus x^2 \oplus x^4 \oplus x^5 \oplus x^7. \tag{14.11}$$

This CRC generator can produce a unique CRC code of a block sample up to 128 (2^7) bits. To generate the CRC code for longer data streams, use a longer CRC generator such as the most commonly used CRC-16 or CRC-32 polynomial specified by the ITU standards. As the number of bits in the CRC increases, the probability of encountering two different blocks with the same CRC during the data transmission approaches zero. Thus, the CRC-32 is enough for many applications.

Each codeword of a binary (n, k) CRC code consists of $n = k + r$ bits. The block of r parity bits is computed from the block of k information bits. The code has a degree r generator polynomial $g(x)$. It also has the property of linear code, i.e., the bitwise addition of any two codewords yields a new codeword.

Error detection at the receiver is made by computing the parity bits from the received information block and comparing them with the received parity bits. An undetected error occurs when the received word is a codeword, but is different from the one that is transmitted. This is possible only when the error pattern is a codeword by itself because of the linearity of the code. The performance of a CRC code is measured by the probability of undetected errors. The comprehensive comparison among several different CRC codes can be found in [10].

Example 14.4: The following CRC-32 generator polynomial is used to generate 32-bit CRC codewords:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1. \tag{14.12}$$

The polynomial (0xEDB88320L) is used in Equation (14.12). Note that we take the reversed order and put the highest order term in the lowest order bit. Thus, the term x^{32} is added and the LSB is the term x^{31}. The first term (x^0) results in the MSB being 1.

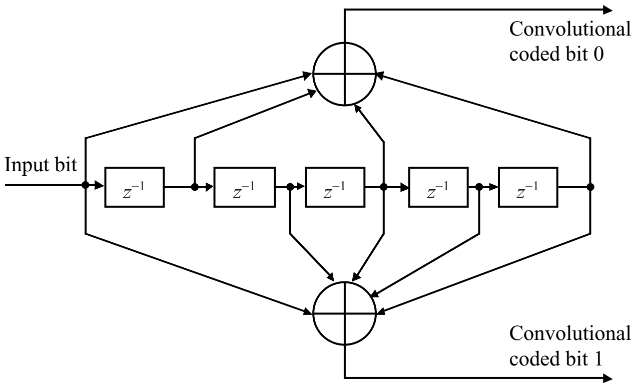


Figure 14.8 A rate 1/2, constraint length 5 convolutional encoder

14.3 Convolutional Codes

A convolutional code is generated by passing the information sequence through a linear finite-state shift register. In general, the shift register consists of $L(k\text{-bit})$ stages and n linear algebraic function generators. The encoder shifts the binary data into the shift register k bits at a time. The number of output bits is n for each k-bit input sequence. This results in the code rate of $R_c = k/n$. The parameter L is the constraint length of the convolutional code.

Convolutional code is a special case of error-control code. Unlike a block code, a convolutional coder is a device with memory. Even though a convolutional coder accepts a fixed number of information symbols and produces a fixed number of code symbols, its computation depends not only on the current set of input symbols, but also on some of the previous input symbols.

14.3.1 Convolutional Encoding

Convolutional codes provide error-correction capability by adding redundancy bits to the information bits. The convolutional encoder is implemented by either the table-lookup or shift register method. Figure 14.8 shows a rate one-half (1/2) convolutional coder.

The convolutional encoder shown in Figure 14.8 uses the following two polynomial generators:

$$b_0 = 1 \oplus x \oplus x^3 \oplus x^5 \tag{14.13}$$

$$b_1 = 1 \oplus x^2 \oplus x^3 \oplus x^4 \oplus x^5. \tag{14.14}$$

For the rate $1/2$ convolutional encoder, each input information bit has two encoded bits, where bit 0 is generated by Equation (14.13) and bit 1 is generated by Equation (14.14). This redundancy enables the Viterbi decoder to choose the correct bits under noise conditions.

14.3.2 Viterbi Decoding

Convolutional encoder can also be represented using the states. The basic block diagram of a 32-state trellis is illustrated in Figure 14.9. In this scheme, each encoding state at time n is linked to two states at time $n + 1$ as shown in Figure 14.9. Each link from the old state at time n to the new state at time

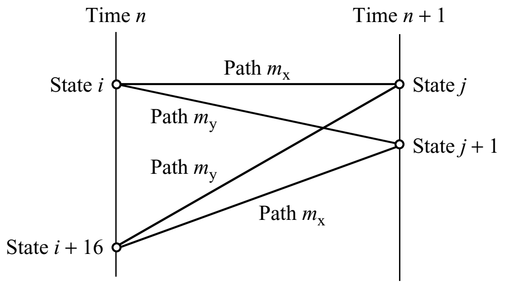


Figure 14.9 Trellis diagram of the rate 1/2, constraint length 5 convolutional codes

$n+1$ associates with a transition path. The new state at time $n+1$ ending up with state j or state $j+1$ depends on the input bit path being m_x or m_y.

In the decoding side, decoder also maintains the states to track the minimum distance with consideration of all possible transition paths. For example, the transition path m_x is from state i to state j, and m_y is the transition path from state $i+16$ to state j. The accumulated path history is calculated as

$$\text{state}(j) = \min\left\{\text{state}(i) + m_\text{X}, \text{state}(i+16) + m_\text{Y}\right\}, \tag{14.15}$$

where the new state history, state(j), is chosen as the small one of the two accumulated past history paths state(i) and state($i+16$) plus the transition paths m_x and m_y, respectively.

The Viterbi algorithm decodes the trellis coded information bits by expanding the trellis over the received symbols. The Viterbi algorithm reduces the computational load by taking advantage of the special structure of the trellis codes. It calculates the 'distance' between the received signal path and all the accumulated trellis paths entering each state. After doing comparison at each state, keep only the most likely path (called surviving path – the path with the shortest distance) based on the current and past path history, and discard all other unlikely paths. Such early rejection of unlikely paths greatly reduces the computation needed for the decoding process.

Example 14.5: Consider the following convolutional code:

$$b_0 = 1 \oplus x \oplus x^2. \tag{14.16}$$

$$b_1 = 1 \oplus x^2. \tag{14.17}$$

The input is '`00101001`' and the initial state is set to 00. The output of this $^1/_2$ convolutional encoder is '`0000110100011111`'. This can also be done by running MATLAB script as follows:

```
trel = poly2trellis(3,[5 7]);  % Define trellis
msg = [0 0 1 0 1 0 0 1]';      % Information data
code = convenc(msg,trel)       % Encode
```

In `poly2trellis(3,[5 7])`, 3 is the constraint length, 5 in octet format (101 in binary) is the generator polynomial defined in Equation (14.17), and 7 in octet format (111 in binary) is the generator polynomial defined in Equation (14.16). `convenc(msg,trel)` will return `code = 0000110100011111`. The decoding path is illustrated in Figure 14.10 without error.

In Figure 14.10, there is no error. The decoding path follows the minimum distance. The distance is the Hamming distance [5] defined as

$$d = d(\mathbf{c}, \mathbf{r}) = r_0 \oplus c_0 + r_1 \oplus c_1 + \cdots r_{n-1} \oplus c_{n-1}, \tag{14.18}$$

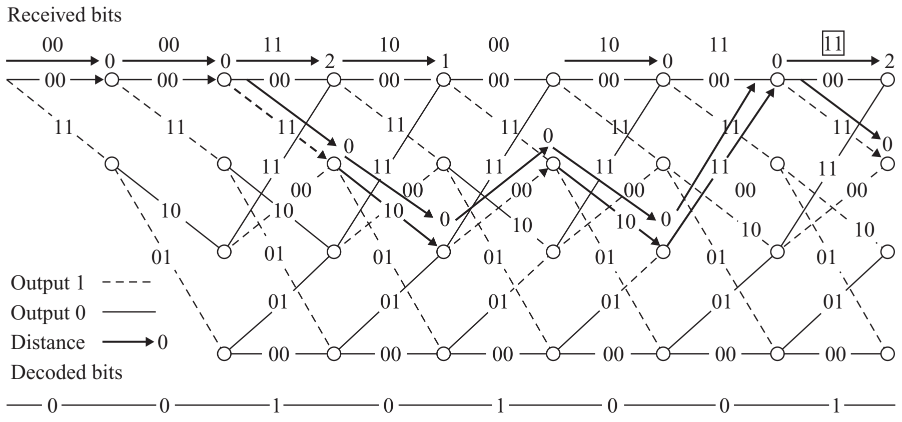


Figure 14.10 An example of trellis decoding without error

where $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ is any n-bit source codeword and $\mathbf{r} = (r_0, r_1, \ldots, r_{n-1})$ is the received codeword. In this example, the possible d in a single node can be 0, 1, 2, or 3. Since there is no error, there is only one path that makes all distance to be minimum 0.

Example 14.6: Using the same convolutional code given in Example 14.5, find the decoding path that leads to the minimum Hamming distance. The following MATLAB script is used to simulate the channel errors and perform Viterbi decoding:

```
ncode = code;                    % Copy code from Example 14.5
ncode(6) =0;                     % Error bit
ncode(11) =0;                    % Error bit
tblen = 3;                       % Trace back length
decoded = vitdec(ncode,trel,tblen,'cont','hard'); % Hard decision
```

The Viterbi decoding function `vitdec( )` will return `decoded = 00101001`. The decoding path has been illustrated in Figure 14.11. There is an error in the third pair of bits. Due to this error, there is no path with distance 0. In this case, there are two possible paths resulting in 0 or 1 since both path distances are the same. After the fourth pair comes, calculate four possible paths, and only one path results in the accumulated minimum distance 1, which survives and decodes with the proper value. Eliminate the bad paths, and a surviving path continues. The same rule applies to the second error that occurs at the sixth pair of bits.

14.3.3 Applications of Viterbi Decoding

Wireless communication technologies have been greatly improved in recent years. Digital cellular systems include both the infrastructures (such as the cellular base stations) and the handsets. A simplified wireless communication system illustrated in Figure 14.12 consists of three sections: transmitter, receiver, and channel. The system contains speech coding and decoding, channel coding and decoding, and finally, modulation and demodulation.

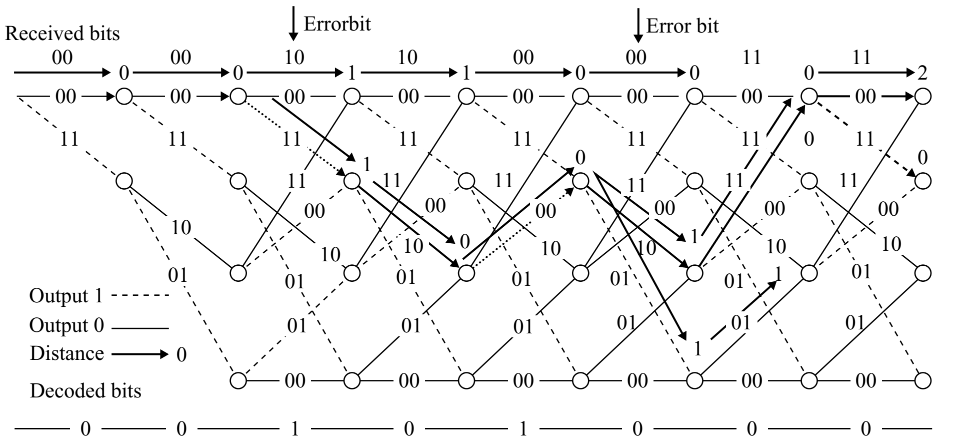


Figure 14.11 An example of trellis decoding with error

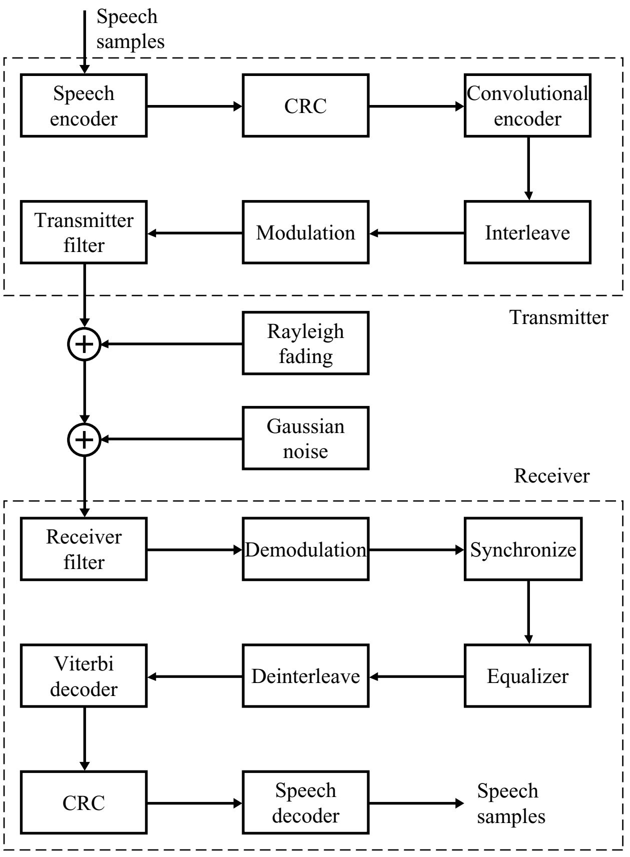


Figure 14.12 Simplified wireless communication system

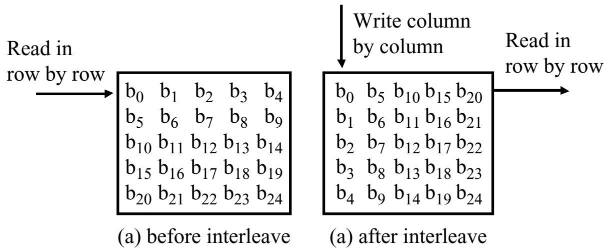


Figure 14.13 A simple example of interleave scheme: (a) before interleave; (b) after interleave

The speech (or source) coding is one of the important DSP applications in wireless communications. As discussed in Chapter 11, the vocoders compress speech signals for bandwidth limited communication channels. The most popular vocoders for wireless communications compress speech from 64 kbps to the range of 6–13 kbps.

The FEC used in the system shown in Figure 14.12 consists of the convolutional encoding and Viterbi decoding algorithms. Modern DSP processors such as the TMS320C55x have special instructions for efficient implementation of Viterbi decoders. The most computational intensive operations of the Viterbi decoding comprise of many add-compare-select iterations. The number of add-compare-select calculations depends on the constraint length L and is equal to $2(L - 2)$. As L increases, the coding gain increases with soft decision. However, the number of add-compare-select calculations increases exponentially. The C55x processors perform the add-compare-select operation in single cycle using the dedicated instructions such as `addsub`, `subadd`, and `maxdiff`.

Channel equalization and estimation are important and challenging DSP tasks for digital communications. The channels of wireless mobile communications are far more complicated than the dial-up channels due to the deep channel-fading characteristics and multipath interferences. Some of the wireless communication devices use equalizers, but most of them use channel-estimation techniques. To combat the burst errors, interleave schemes are used. Although a severe fading may destroy an entire frame, it is unusual for the fading to last for several frames. By spreading the data bits across a longer sequence, the Viterbi algorithm can recover some of the lost bits at the receiver. As illustrated in Figure 14.13, a simple example of the interleaving technique is to read symbols row by row and write them out column by column.

In this example, the input data is 5 bits per frame $\{b_0, b_1, b_2, b_3, b_4\}$, $\{b_5, b_6\ b_7, b_8, b_9\}$, etc., as shown in Figure 14.13(a). These bits are written into a buffer column by column as shown in Figure 14.13(b), but are read out row by row as $\{b_0, b_5, b_{10}, b_{15}, b_{20}\}$, etc. Therefore, not all the bits from one frame will be lost by a bad received slot due to the channel fading.

Another important component in Figure 14.12 is the Rayleigh fading-channel model. In a mobile communication environment, the radio signals picked up by a receiver antenna come from many paths caused by the surrounding buildings, trees, and many other objects. These signals can become constructive or destructive. As a mobile phone user travels, the relationship between the antenna and those signal paths changes, which causes the fading to be randomly combined. Such effects can be modeled by the Rayleigh distribution called Rayleigh fading. In order to provide a mobile communication environment for wireless design and research, effects of multipath fading must be considered.

Example 14.7: Two Rayleigh fading models have been developed, one was proposed by Jackes [14], and the second model uses a second-order IIR filter. The Jackes fading-channel model can be implemented in C as listed in Table 14.6. The channel noise is simulated using a white Gaussian noise.

Table 14.6 C implementation of fading-channel model proposed by Jakes

```
/* PI = 3.14
   C = 300000000 m/s
   V = Mobile speed in mph
   Fc = Carrier frequency in Hz
   N = Number of simulated multi-path signals
   N0 = N/4 - 1/2, the number of oscillators
*/
   wm = 2*PI*V*Fc/C;
   xc(t) = sqrt(2)*cos(PI/4)*cos(wm*t);
   xs(t) = sqrt(2)*sin(PI/4)*cos(wm*t);
   for(n=1;n<=N0;n++)
   {
      wn = wm*cos(2*PI*n/N);
      xc(t) += 2*cos(PI*n/N0)*cos(wm*t);
      xs(t) += 2*sin(PI*n/N0)*cos(wm*t);
   }
```

14.4 Experiments and Program Examples

In this section, we will implement the Reed–Solomon encoder, decoder, convolutional encoder, Viterbi decoder, and CRC code using MATLAB, C, or C55x programs.

14.4.1 Reed–Solomon Coding Using MATALB

This section implements the RS(7, 3) Reed–Solomon code using MATLAB. The codeword length is $n = 7$ and the message length is $k = 3$. The data to be encoded is given in Table 14.7. Using the primitive polynomial (1011), the generator polynomial is generated by MATLAB function `rsgenpoly(7,3)` provided in the *Communication Toolbox*.

In this experiment, the primitive polynomial is $p(x) = 1 + x + x^3$. The output of generation polynomial function `rsgenpoly(7,3)` is (`1 3 1 2 3`), which is in symbol format and can be converted to polynomial presentation using Table 14.2. Thus, the generator polynomial generated by `rsgenpoly(7,3)` is $g(x) = x^4 + (\alpha + 1)x^3 + x^2 + \alpha x + (\alpha + 1)$, where α is the root of $p(x)$.

Using MATLAB functions that support the Reed–Solomon encoder `rsenc( )`, decoder `rsdec( )`, and Galois field array creator `gf( )`, we can easily simulate a communication system. The MATLAB script is listed in Table 14.8. In this experiment, a 3-symbol array `msg` is encoded and a 7-symbol `code`

Table 14.7 RS(7, 3) encoded, erased, and decoded data

Symbols	Raw data (`msg`)	Encoded (`code`)	Erased (`rxcode`)	Decoded (`decod`)
1	3	3	**2**	3
2	5	5	5	5
3	0	0	0	0
4		3	3	
5		6	6	
6		6	6	
7		5	**2**	

Table 14.8 Listing of Reed–Solomon (7, 3) encoding and decoding

```
msg1= [3 5 0];               % Information
m = 3;                       % Number of bits per symbol
n = 2^m-1;                   % Word lengths for code
k= 3;                        % Number of information symbols
msg=gf([msg1],m)             % Galois array
gen=rsgenpoly(n,k);          % Specify RS generation polynomial
code = rsenc(msg,n,k,gen)    % Encode the information symbols
rxcode = code;               % Transmit and receive
rxcode(1)= 2;                % Error eraser
rxcode(7) = 2;               % Error eraser
decod = rsdec(code,n,k,gen) % Decoding
```

is generated. The received `rxcode` contains two errors. The decoded symbols are stored in `decod`. All these arrays are listed in Table 14.7.

Procedures of the experiment are listed as follows:

1. Start MATLAB and change the working directory to `..\experiments\exp14.4.1_RS_codec`.
2. Run the experiment by typing `rs_codec` in the MATLAB command window.
3. Modify the error locations and the number of errors in a frame to check decoder output `decod`. The maximum correctable errors in a frame are two symbols.
4. Modify the code to use RS(15, 11). Repeat the same tests.
5. Write floating-point C implementation of the RS(7, 3) and compare the result with the MATLAB output.
6. Convert the floating-point C functions to C55x assembly functions, and compare the experiment results against the MATLAB results.

14.4.2 Reed–Solomon Coding Using Simulink

Using the same generation polynomial given in the previous experiment and the same RS(7, 3) coding, we can use a Simulink model to simulate a wireless system. As shown in Figure 14.14, we have signal source Bernoulli binary generator, RS(7, 3) encoder and decoder modules, modulation and demodulation modules, parallel to serial and serial to parallel conversions, AWGN channel, and bit-error monitors. The parameters such as codeword length, bit-error rate (BER), n and k of RS(n, k) can be configured. For example, double click on the `AWGN` module to change signal-to-noise ratio (SNR) which will affect the BER.

Procedures of the experiment are listed as follows:

1. Start MATLAB and change the working directory to `..\experiments\exp 14.4.2_RS_simulink`.
2. Run the experiment by typing `Reed_Solomon` in the MATLAB command window. This starts the Simulink and creates a window as shown in Figure 14.14.

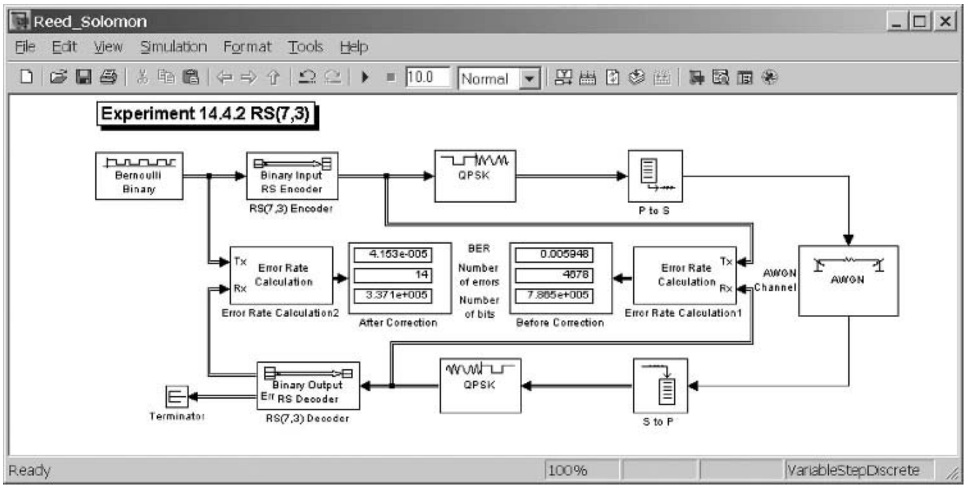


Figure 14.14 Simulink model for Reed–Solomon code

3. Modify SNR of WGN channel, and then check the BER.

4. Modify the code to use RS(15, 11) encoder and decoder. Repeat the same tests.

5. Compare the BER before and after correction. As shown in Figure 14.14, there are 4678 and 14 bit errors corresponding to BER of 0.005948 and 0.00004153, respectively, before and after correction. Note that the bit rate before correction is 7/3 times higher than that of after correction.

14.4.3 Verification of RS(255, 239) Generation Polynomial

As discussed in Example 14.2, we used the generation polynomial $\{g_0, g_1, \ldots, g_{16}\} = \{$`0x4f`, `0x2c`, `0x51`, `0x64`, `0x31`, `0xb7`, `0x38`, `0x11`, `0xe8`, `0xbb`, `0x7e`, `0x68`, `0x1f`, `0x67`, `0x34`, `0x76`, `0x01`$\}$, stored in `gpoly_alpha`, and the primitive polynomial $p(x) = 1 + x^2 + x^3 + x^4 + x^8$. All α^i, $i = 1, \ldots, 16$, are the roots of this generation polynomial defined in Equation (14.5).

With the given primitive polynomial, we can compose a table for $GF(256)$. The elements are populated as follows:

$\alpha^0 - \alpha^7$ = `0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80`
$\alpha^8 = \alpha^4 + \alpha^3 + \alpha^2 + 1$ = `0x1d`
$\alpha^9 = \alpha^8\alpha^1 = (\alpha^4 + \alpha^3 + \alpha^2 + \alpha^1)\alpha^1 = \alpha^5 + \alpha^4 + \alpha^3 + \alpha^1$ = `0x3a`

Using the same method, all data for α^i, $i = 0, \ldots, 255$ over $GF(256)$ can be calculated and this data is generated in the table `alpha_tabl[ ]`.

Using a similar method, we can also convert the generation polynomial to index-based table as `int gpoly_index[17] = {0x88, 0xf0, 0xd0, 0xc3, 0xb5, 0x9e, 0xc9, 0x64,0x0b, 0x53, 0xa7, 0x6b, 0x71, 0x6e, 0x6a, 0x79, 0x00}`. With these tables, we can verify that α^i, $i = 1, \ldots, 16$, are all the roots of $g(x)$. Table 14.9 lists the C code to verify if $g(\alpha^i) = 0$ for $i = 1, \ldots, 16$. The files used for this experiment are listed in Table 14.10.

Table 14.9 C code to verify the roots of generation polynomial

```
void main(void)
{
   short i,j;
   short x[17];
   for (i=1;i<=16;i++)    // alpha ^1, ^2, ..., ^16
   {
      x[i] = gpoly_alpha[0];
      for (j=1;j<=16;j++)
      {
        x[i] = x[i]  ^ (alpha_tabl[(gpoly_index[j]+j*i)%255]);
      }
      printf("alpha^%2d makes the generation polynomial = %d\ n",i, x[i]);
   }
}
```

Procedures of the experiment are listed as follows:

1. Start CCS, open the project `verify_root.prj`, build, and load the program.
2. Check the output to see if all of them are zeros.
3. Modify `x[i]` with one error and do the same calculation to see if the corresponding output is zero.

14.4.4 Convolutional Codes

This experiment implements convolutional codes using the Simulink model `conv_vit.mdl` shown in Figure 14.15. Using the modules given in Section 14.4.2, we replaced the Reed–Solomon encoder with convolutional encoder and the Reed–Solomon decoder with the Viterbi decoder. This example uses the rate 1/2 convolutional code with constraint length of 7.

The polynomial for bit 0 in the upper addition node is 1011011 (Octet 133), which can be expressed as

$$b_0 = 1 \oplus x^2 \oplus x^3 \oplus x^5 \oplus x^6. \tag{14.19}$$

The polynomial for bit 1 in the lower addition node is 1111001 (Octet 171), and can be expressed as

$$b_1 = 1 \oplus x^1 \oplus x^2 \oplus x^3 \oplus x^6. \tag{14.20}$$

Table 14.10 File listing for experiment `exp14.4.3_RS_root`

Files	Description
`verify_root.prj`	C55 project file
`root_test.c`	C file calculates the polynomial with α
`table.c`	α-indexed data over $GF(256)$
`verify_root.cmd`	C55x linker command file

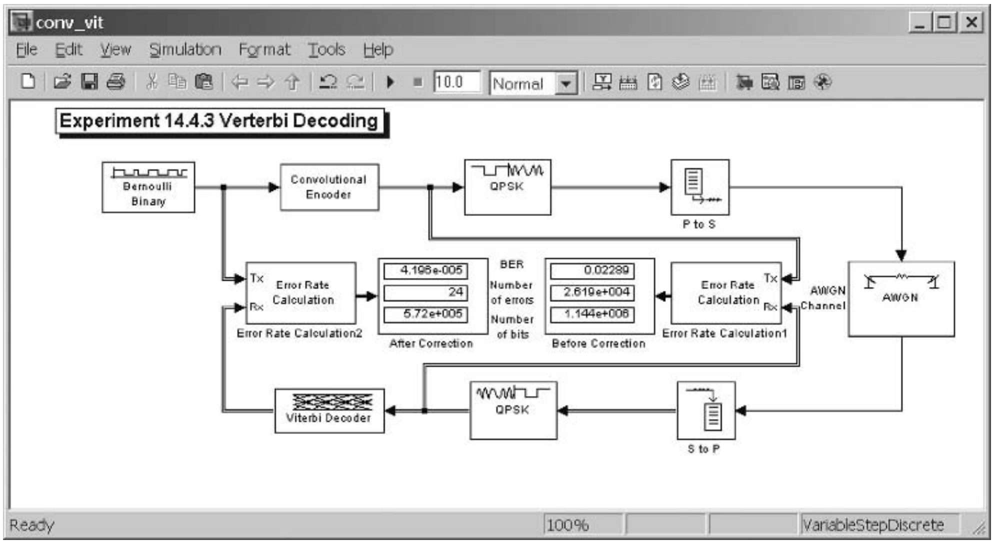


Figure 14.15 Convolutional encoding and Viterbi decoding with AWGN channel

We use function `poly2trellis(7, [133 171])` in the field for the trellis structure. Similar to Example 14.5, the constraint length is 7 in this trellis structure, 133 in octet format is the generator polynomial defined in Equation (14.20), and 171 in octet format is the generator polynomial defined in Equation (14.19). In the Viterbi decoding side, the trace-back depth or decision types can be modified as in Example 14.6.

Procedures of the experiment are listed as follows:

1. Start MATLAB and change the directory to `..\experiments\exp14.4.4_Viterbi`.
2. Run the experiment by typing `conv_vit` in the MATLAB command window. This starts the Simulink and creates a window that contains the wireless communication system as shown in Figure 14.15.
3. Modify the SNR of WGN channel, and then check the BER.
4. Modify the convolutional encoder and Viterbi decoder modules to use `poly2trellis(5, [27 25])`, and check the BER.
5. Change track depth in Viterbi decoder module from 96 to 72 to see the performance difference.
6. Compare the BER before and after correction. As shown in Figure 14.15, there are bit errors of 26 190 and 24 corresponding to BER of 0.02289 and 0.00004196, respectively, before and after correction. Note that the bit rate before correction is two times higher than that of after correction.

14.4.5 Implementation of Convolutional Codes Using C

This experiment implements the rate 1/2 convolutional code with constraint length 7. The files used for this experiment are listed in Table 14.11.

The table `xorOp[256]` is generated to simulate the XOR operations of all 8 bits in byte. The C code which performs 1/2 convolutional code is listed in Table 14.12, where `encState` represents the

Table 14.11 File listing for experiment `exp14.4.5_CONV`

Files	Description
`conv_trs27.pjt`	DSP project file
`conv_trs27Test.c`	Main program for testing experiment
`table.c`	Precalculated generation data
`conv_trs27.c`	Calculation of convolutional code
`conv_trs27.h`	Header file with constants and prototyping
`conv_trs27.cmd`	C55x linker command file
`dtmf12.pcm`	Data file

Table 14.12 C code of $^1/_2$ rate convolutional encoding

```
#define POLYGEN0 0x6d
#define POLYGEN1 0x4f

void conv_trs27 (FILE *fpin ,FILE *fpout)
{
   unsigned short encState;
   short i,input;
   unsigned char byte0,byte1;
   encState = 0;
   while( (input=getc(fpin)) != EOF)
   {
     input = (input)&0x00ff;
     byte0 = byte1 = 0;
     for(i=7;i>=0;i--){
       encState = (encState << 1) |  ((input >> 7) & 1);
       input <<= 1;
       /* to compose a byte */
       byte0 <<=1;
       byte1 <<=1;
       byte0 | = xorOp[encState & POLYGEN0];
       byte1 | = xorOp[encState & POLYGEN1];
     }
    fwrite(&byte0,sizeof(char),1,fpout);
    fwrite(&byte1,sizeof(char),1,fpout);
   }
   /* Generate last one */
   byte0 = byte1 = 0;
   for(i=5;i>=0;i--){
       encState = encState << 1;
       byte0 <<=1;
       byte1 <<=1;
       byte0 | = xorOp[encState & POLYGEN0];
       byte1 | = xorOp[encState & POLYGEN1];
   }
   fwrite(&byte0,sizeof(char),1,fpout);
   fwrite(&byte1,sizeof(char),1,fpout);
}
```

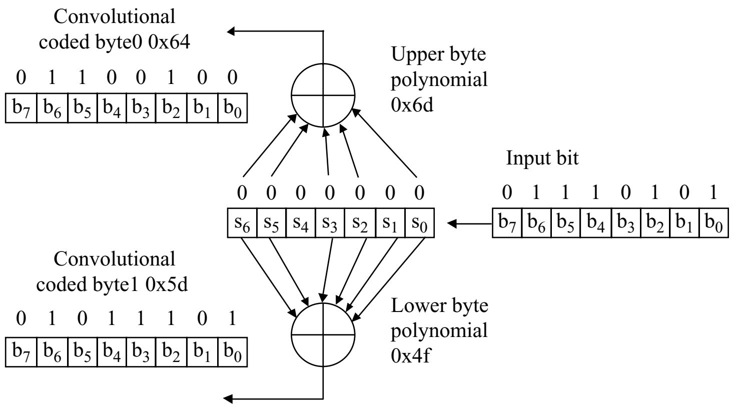


Figure 14.16 Example of $^1/_2$ rate, 7-bit constraint convolutional encoding

encoder states, and `byte0` and `byte1` are the outputs of path 0 and path 1, respectively, as shown in Figure 14.16.

When running the C55x program to encode the data from file `dtmf12.pcm`, the first four codewords are listed in Table 14.13. Figure 14.16 illustrates this example by showing how to interpret the polynomials `POLYGEN0=0x6d` and `POLYGEN1=0x4f` from the source code to the shift circuits.

Procedures of the experiment are listed as follows:

1. Start CCS, open the project `conv_trs27.pjt`, build, and load the program.
2. Using the MATLAB script given in Example 14.5 as reference, write a MATLAB program to perform the same task.
3. Calculate the first four results by hand as shown in Table 14.13.

14.4.6 Implementation of CRC-32

This experiment implements the CRC-32 introduced in Example 14.4. The files used for this experiment are listed in Table 14.14. Using the project `CRC32` in the directory `..\experiments\exp14.4.6_CRC32`, compile, load, and run the code.

The code `crc32_calc.c` is listed in Table 14.15. The `crcTable[256]` in the file `crc32Table.h` contains all the precalculated data that is the 1-byte output of the generation polynomial.

Table 14.13 The first four encoded codewords

Sequence	Input	Initial state	End state	Upper byte	Lower byte
1	0x75	0x00	0x75	0x64	0x5d
2	0xfd	0x75	0xfd	0x45	0x14
3	0x18	0xfd	0x18	0xe1	0x85
4	0xfc	0x18	0xfc	0x78	0xc1

Table 14.14 File listing for experiment `exp14.4.6_CRC32`

Files	Description
`crc32.pjt`	C55 project file
`crc32_test.c`	Main program for testing experiment
`table.c`	Precalculated generation data
`crc32_calc.c`	CRC-32 calculation
`crc32.h`	Header file
`crc32Table.h`	Precalculated CRC table data
`crc32.cmd`	C55x linker command file
`dtmf12.pcm`	Data file

Table 14.15 C code to calculate CRC-32

```
/*
poly: X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8+X^7+X^5+X^4+X^2+X^1+X^0
  0xEDB88320L =1110,1101,1011,1000,1000,0011,0010,0000B
*/

unsigned long calcCrc( FILE *fp )
{
    unsigned long crc, acc1,acc0;
    short byteIn;
    crc=0;
    while( (byteIn=getc(fp)) != EOF ) {
      acc0 = crc^MAX32BIT;
      acc1 = (acc0>>8) & MAX24BIT;
      acc0 = crcTable[(acc0^byteIn)&0xFF];
      crc = acc0^(acc1) ;
    }
    return(crc);
}
```

If processing the same `dtmf12.pcm` using this code and the standard WinZip, the output CRC check value is the same `crc32 = 0x5290f9e8`. Procedures of the experiment are listed as follows:

1. Start CCS, open the project `crc32.prj`, build, and load the program.

2. Compare the CRC value against the CRC generated from WinZip for the same file.

3. This experiment uses table for CRC-32 computation. Modify the experiment such that the CRC-32 is run-time computed using shift registers.

References

[1] D. J. Rauschmayer, *ADSL/VDSL Principles: A Practical and Precise Study of Asymmetric Digital Subscriber Lines and Very High Speed Digital Subscriber Lines*, Indiana: Macmillian, 1999.

[2] K. Hiltunen, *Coding and Interleaving in CDMA*, System Services Department, FIN-02420 Jorvas, Finland, Oct. 1997.

[3] MATLAB, Version 7.0.1 Release 14, Sep. 2004.

[4] T. S. Rappaport, *Wireless Communications – Principles and Practice*, Prentice Hall, NJ: IEEE Press, 1996.
[5] M. Y. Rhee, *Error Correcting Coding Theory*, New York: McGraw-Hill, 1989.
[6] A. Matache,'Encoding/decoding Reed Solomon codes,' Oct. 1996, http://www.ee.ucla.edu/∼matache/rsc/slide.html.
[7] L. Baert, *Digital Audio and Compact Disc Technology*, 2nd Ed., UK: BH Newnes, 1995.
[8] W. Tian, 'ECC scheme for wireless digital audio signal transmission,' US patent no. 6 327 689, Dec. 2001.
[9] E. Alhoniemi, 'Error detection and control in data transfer,' Nov. 1998, http://users.tkk.fi/∼eal/essay10.html.
[10] D. Sheinwald, J. Satran, P. Thaler, and V. Cavanna. 'Internet protocol small computer system interface (iSCSI) cyclic redundancy check (CRC)/checksum considerations,' IETF RFC 3385, Sep. 2002.
[11] A. J. Viterbi, 'Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,' *IEEE Trans. Inf. Theory*, vol. IT-13, pp. 260–269, Apr. 1967.
[12] Texas Instruments, Inc.,*Viterbi Decoding Techniques in the TMS320C54x Family*, Literature no. SPRA071, 1996.
[13] Texas Instruments, Inc., *TMS320C55x DSP Programmer's Guide*, Literature no. SPRU376A, 2001.
[14] W. C. Jakes, Jr., *Microwave Mobile Communications*, New York: John Wiley & Sons, 1974.

Exercises

1. Figure 14.17 shows a rate $^1/_2$, constraint length 3 convolutional encoder:
   (a) Find the upper and lower generation polynomials.
   (b) Given the input data '00110011' and assuming all initial states are zero, calculate the output bits from the encoder.

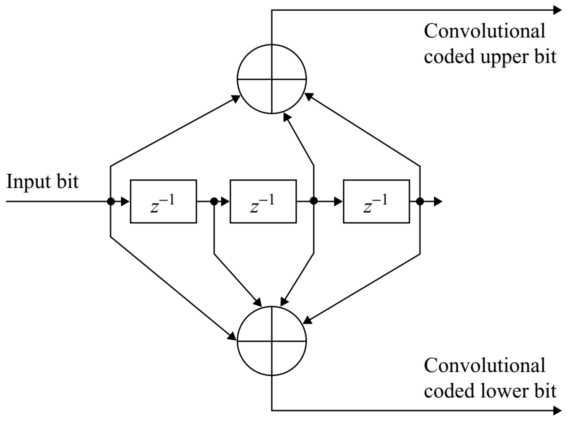


Figure 14.17 A rate $^1/_2$, constraint length 3 convolutional encoder

2. For the primitive polynomial $p(x) = 1 + x + x^6$, α is the root of the primitive polynomial. This yields $\alpha^6 + \alpha + 1 = 0$. Using Table 14.2 as an example, list the exponential, polynomial, and symbol presentations. You may use MATLAB script to find these values.

3. In Figure 14.11, the trellis-encoded data contains errors. If the third pair of bits has been changed from 11 to 00, draw a decoding path to see if this error can be corrected or not.

4. In wireless communications, if the FEC is used with convolutional code, usually a CRC will be generated along with the convolutional encoded data. Why this CRC code is necessary? On the other hand, why the Reed–Solomon encoded data may not need CRC?

15 Introduction to Digital Image Processing

Digital images and videos have become an integral part of entertainment, business, and education in our daily life. Since the video and image compression algorithms have a very broad scope, this chapter will focus only on some fundamental image processing methods. We will use the C5510 DSK for experiments and MATLAB functions available in the *Image Processing Toolbox* for image processing, analysis, visualization, and algorithm development.

15.1 Digital Images and Systems

Image processing applications such as high-definition televisions (HDTV), digital still cameras, Internet TV, and personal media centers are everywhere. Image processing and video signal processing have become highly demanding skills for DSP engineers. Due to the tremendous amount of digital data to be processed, DSP processors used for image and video processing require efficient processing units, fast I/O throughput, and even to be equipped with hardware accelerators for special image processing functions such as discrete cosine transform (DCT), inverse DCT (IDCT), and variable length coding and decoding.

Texas Instruments' TMS320C55x family is capable of performing many image processing tasks. For example, the TMS320C5510 includes hardware accelerators for motion estimation, DCT, and interpolation functions. These image accelerators can dramatically improve the processing speed for video and image compression algorithms used by H.263, MPEG-4, MPEG-2, and JPEG. Digital image processing has many common characteristics as one-dimensional (1-D) signal processing introduced in previous chapters, but with many specific aspects. In this section, we will introduce the basic concepts of digital images and digital image systems.

15.1.1 Digital Images

A digital image is a set of sampled data that mapped onto a two-dimensional (2-D) grid of colors. Similar to 1-D signals, digital images can be represented using digits (samples) in a 2-D space. Each image sample is called a pixel, which stands for picture element. For a black-and-white (B&W) image, each pixel consists of one number, while the color image consists of multiple numbers for each pixel. To display the intensity used for B&W images, the digital numbers are usually converted to gray levels between 0 and 255, where '0' represents a black pixel, while '255' corresponds to a white pixel. For machine vision, B&W image usually can provide adequate information. Each pixel of color images can be represented using three numbers for the three primary colors: red (R), green (G), and blue (B), thus is often referred as RGB data. Proper mixing of these three primary colors can generate different colors. Each data can be stored as a byte in memory, and this forms the commonly used 24-bit RGB data (8 bits for R, 8 bits for G, and 8 bits for B). Color images are widely used in photographs, television broadcasts, and computer displays.

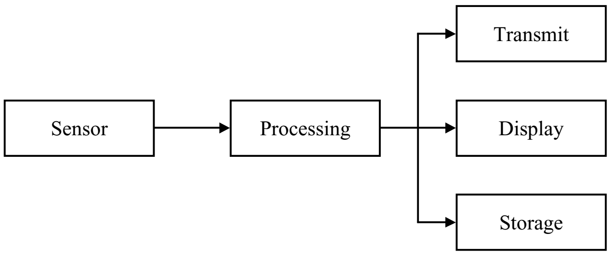
The 1-D signal processing is constrained by the sampling theorem, while the digital image is bonded in spatial domain. The image resolution is the ability of distinguishing spatial details of images. The terms dots-per-inch and pixel-per-inch are commonly used for image resolution. In audio applications, higher sampling rate generally yields better audio quality with the trade-off of higher data rate. Likewise, an image with more pixels generally has better (or finer) spatial resolution since this image is equivalent to having a higher 'sampling rate'.

Another term often used is pixel dimensions, which means the image width and height in terms of the numbers of pixels. For example, the National Television System Committee (NTSC) defines that the television standard for North America is 720 pixels (width) by 480 pixels (height). Today, most of the digital images are in the size of 1024 $\times$ 768 pixels or larger for computers and HDTVs, and 720 $\times$ 480 or smaller for real-time transmission via networks.

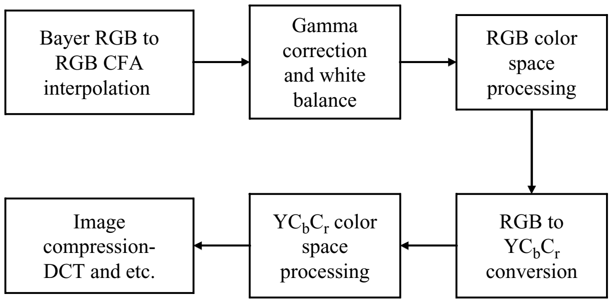
Although high-resolution images are desired for viewing, many applications still use lower resolution images because the amount of data to be processed limits the real-time processing ability and the storage and transmission capacity. For video applications, the data samples are usually measured in megabytes per second.

15.1.2 Digital Image Systems

A simplified digital image system is shown in Figure 15.1. The input image captured by the sensor is sent to the processing unit, and the processed image is presented to storage media, display device, or



(a) Simplified digital image system block diagram.


Bayer RGB to RGB CFA interpolation
Gamma correction and white balance
RGB color space processing
RGB to YC_bC_r conversion
YC_bC_r color space processing
Image compression-DCT and etc.

(b) Functional blocks of the image processing system.

Figure 15.1 Block diagram of digital image system: (a) simplified digital image system block diagram; (b) functional blocks of the image processing system

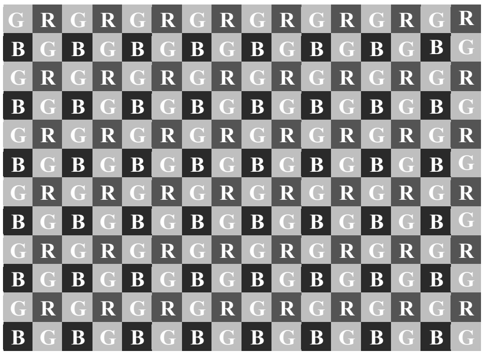

G R G R G R G R G R G R G R G R
B G B G B G B G B G B G B G B G

Figure 15.2 A 16 × 12 Bayer RGB color pattern

transmitted over the network. The data acquisition device can be a charge-coupled device (CCD) or a complimentary metal-oxide semiconductor (CMOS) sensor. These image sensors convert the light, or scene, into electrical signals and store them in an array of area with n-by-m memory. Most CCD and CMOS sensors for digital still cameras range from 2560 × 1920 pixels (5M pixels) to 3072 × 2304 (7M pixels).

The sensor image data stored in memory is usually arranged as an array of n-by-m colored digital elements. This format is called Bayer RGB pattern. The Bayer RGB pattern is the most popular image format used for CCD and CMOS sensors. Figure 15.2 shows a 16 × 12 Bayer RGB color image. For these 192 pixels, half of them are green, one-quarter are red, and one-quarter are blue.

The CCD or CMOS sensors can output pixels in a line-by-line sequential order. For example, the output can be in the form of *GRGRGR... GR* for even lines, and *BGBGBG... BG* for odd lines. There are several representations of color images used in digital video display and image processing. A color space represents colors using digital pixels. A B&W image uses one number for each pixel, while the color image needs multiple numbers per pixel.

15.2 RGB Color Spaces and Color Filter Array Interpolation

For digital cameras or video camcorders, a sensor is often used to capture the photon signal and convert it into electronic signal. A CCD sensor has high quality and often used for high-end photographic equipments; however, the CCD sensor also has high power consumption. A CMOS sensor requires less power, and thus is often used in portable devices such as wireless camera phones where power consumption is an important issue. The image sensors output 2-D Bayer RGB color signals as shown in Figure 15.2. The RGB pattern from an image sensor is called color filter array (CFA) RGB. In a CFA RGB pattern, each pixel contains only partial information. As shown in Figure 15.2, the 16 × 12 Bayer RGB pattern is consisted only of 96 green pixels, 48 red pixels, and 48 blue pixels. To make a true RGB color space, each pixel must have three data values to represent the three primary color components. The process of converting the Bayer RGB to a true RGB color space is called CFA RGB interpolation. The interpolation process finds and fills in the missing pixels. Figure 15.3 shows the RGB color space after the CFA interpolation from the Bayer RGB color pattern given in Figure 15.2. Since each color pixel uses three numbers, the RGB color space contains three times of numbers than the Bayer RGB. The RGB color space shown in Figure 15.3 contains 192 × 3 numbers. In RGB color space, the other colors are obtained by proper mixing of the R, G, and B primary colors.

To convert the Bayer RGB pattern to the RGB color space, we need to interpolate two missing color values for each pixel. There are several methods to perform CFA interpolation, including linear, nearest neighbor, and cubic etc., with different complexities and speeds. Because the pixel interpolation is

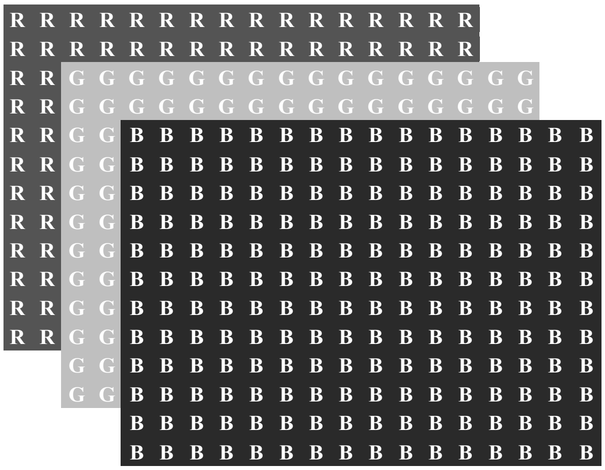


Figure 15.3 A 16 × 12 RGB color space; each pixel has three numbers for R, G, and B

performed for every pixel of the Bayer RGB pattern to obtain the RGB color space, the CFA interpolation demands a high computational load, especially for the real-time processing of large-size images.

The nearest neighbor interpolation method is illustrated here. To interpolate a red or blue pixel, there are four possible conditions for each interpolation, as shown in Figure 15.4(a)–(h). For example, the missing red pixel in Figure 15.4(b) can be interpolated from two neighboring red pixels R_1 and R_2. The interpolated R_0 or B_0 is located in the center in the figures. The equations used for interpolating R and B colors are listed as follows:

(a) $R_0 = R_0$.

(b) $R_0 = (R_1 + R_2)/2$.

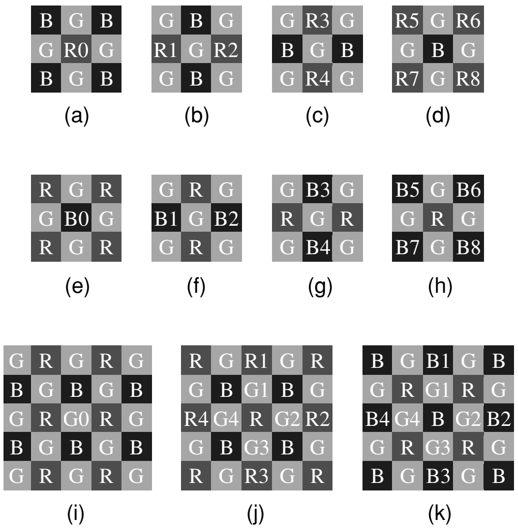

B G B / G R0 G / B G B
(a)
G B G / R1 G R2 / G B G
(b)
G R3 G / B G B / G R4 G
(c)
R5 G R6 / G B G / R7 G R8
(d)
R G R / G B0 G / R G R
(e)
G R G / B1 G B2 / G R G
(f)
G B3 G / R G R / G B4 G
(g)
B5 G B6 / G R G / B7 G B8
(h)
G R G R G / B G B G B / G R G0 R G / B G B G B / G R G R G
(i)
R G R1 G R / G B G1 B G / R4 G4 R G2 R2 / G B G3 B G / R G R3 G R
(j)
B G B1 G B / G R G1 R G / B4 G4 B G2 B2 / G R G3 R G / B G B3 G B
(k)

Figure 15.4 Interpolation of red, blue, and green pixels

(c) $R_0 = (R_3 + R_4)/2$.

(d) $R_0 = (R_5 + R_6 + R_7 + R_8)/4$.

(e) $B_0 = B_0$.

(f) $B_0 = (B_1 + B_2)/2$.

(g) $B_0 = (B_3 + B_4)/2$.

(h) $B_0 = (B_5 + B_6 + B_7 + B_8)/4$.

For green pixel interpolation, we refer to Sakamoto's adaptive interpolation. As shown in Figure 15.4(i)–(k), there are three possible methods for green pixel interpolation. The adaptive interpolation uses the correlation in the neighboring pixels to adapt the interpolation. When the horizontal correlation is stronger, $|R_2 - R_4|$ will be smaller than $|R_1 - R_3|$, the horizontal pixels G_2 and G_4 are used to interpolate the missing center green pixel. When the vertical correlation is stronger, $|R_2 - R_4|$ will be greater than $|R_1 - R_3|$, the vertical pixels G_1 and G_3 are used. When there is no dominating correlation, we use all four neighboring green pixels. The CFA interpolation of missing pixels for the G color shown in Figure 15.4(i)–(k) can be computed by the following equations. In the figures, the interpolated G_0 is in the center location.

(i) $G_0 = G_0$.

(j) $$G_0 = \begin{cases} (G_1 + G_3)/2 & \text{if} \quad |R_1 - R_3| < |R_2 - R_4| \\ (G_2 + G_4)/2 & \text{if} \quad |R_1 - R_3| > |R_2 - R_4| \\ (G_1 + G_2 + G_3 + G_4)/4 & \text{if} \quad |R_1 - R_3| = |R_2 - R_4| \end{cases}.$$

(k) $$G_0 = \begin{cases} (G_1 + G_3)/2 & \text{if} \quad |B_1 - B_3| < |B_2 - B_4| \\ (G_2 + G_4)/2 & \text{if} \quad |B_1 - B_3| > |B_2 - B_4| \\ (G_1 + G_2 + G_3 + G_4)/4 & \text{if} \quad |B_1 - B_3| = |B_2 - B_4| \end{cases}.$$

The wordlength of Bayer RGB samples from a sensor is usually from 8 to 14 bits. For the 8-bit RGB color space, a pixel has three 8-bit numbers ranging from 0 to 255 to equally represent each color. The RGB color can be viewed as a cube shown in Figure 15.5. For the 24-bit RGB space, the number 0 means no color and the number 255 represents the fully saturated color. In Figure 15.5, the primary color number range of [0, 255] has been normalized to [0, 1].

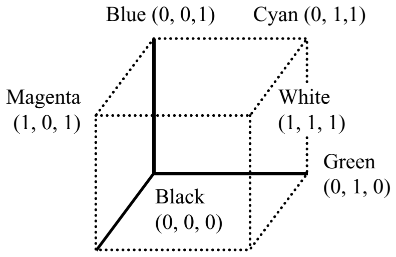


Figure 15.5 An RGB cube

15.3 Color Spaces

The RGB is the most widely used color space in color image processing. However, there are several other color spaces for specific applications.

15.3.1 YC_bC_r and YUV Color Spaces

The RGB representation is based on technical reproduction of color information. However, the human vision is more sensitive to brightness than color changes. This means a human being perceives a similar image even if the color varies slightly. This fact leads to other representations such as the YC_bC_r color space. The relation between the RGB color space and the YC_bC_r color space is defined by ITU CCIR 601 standard. The conversion matrix can be expressed as

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}. \tag{15.1}$$

The YC_bC_r is often used by JPEG and MPEG standards. ITU Recommendation 601 defines the number Y with 220 positive quantization values, ranging from 16 to 235. The C_b and C_r values are ranged from 16 to 240, and centered at 128. The numbers 0 and 255 should not be used for image coding in the 8-bit color spaces because these two numbers are used by some manufactures for special synchronization codes.

The YC_bC_r color space is mainly used for computer images, while the YUV color space is generally used for composite color video standards, such as NTSC and PAL (phase alternating line), etc. The relations between the RGB color space and the YUV color space are defined by ITU CCIR 601 standard as

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{15.2}$$

MATLAB *Image Processing Toolbox* provides several color space conversion functions to convert color images from one color space to another.

Example 15.1: MATLAB function `rgb2ycbcr` converts the RGB color space image to the YC_bC_r color space. The function `imshow` displays image in either RGB color or gray level. The function `ycbcr2rgb` converts the YC_bC_r color space back to the RGB color space. For example,

```
YCbCr = rgb2ycbcr(RGB); % RGB to YCbCr conversion
imshow(YCbCr(:,:,1));   % Show Y component of YCbCr data
```

The function `imshow` displays the luminance of the image `Y=YCbCr(:,:,1)` as a grayscale image. The C_b and C_r are represented by the matrices `YCbCr(:,:,2)` and `YCbCr(:,:,3)`, respectively.

The YUV and YC_bC_r data samples can be arranged in two ways: the sequential and the interleaved. The sequential method is used for backward compatibility with the B&W television signals. The sequential method stores all the luminance (Y) data in one continuous memory block, followed by the color difference U block, and finally the V block. The data memory is arranged as *YY. . . .YYUU. . . UUVV..VV*. This arrangement allows television decoder to access Y data continuously for the B&W televisions. For today's image processing algorithms such as MPEG-4 and H.264, the data is usually arranged as interleaved YC_bC_r format for fast access and to reduce the memory storage requirement.

15.3.2 CYMK Color Space

In addition to the RGB, YC_bC_r, and YUV color spaces, there are several other color spaces used by modern digital image systems to fit their unique applications. For example, the CMYK (cyan, magenta, yellow, and black) color space is a subtractive color space used primarily for color printers. The RGB color space generates different colors by adding different portions of red, green, and blue color. On the other hand, the CMYK describes different colors by subtracting from white color. This is because the printer puts color on paper as a result of reflection. The color space conversion between the RGB and CMYK color spaces is defined as

$$\begin{aligned} C &= 1 - R \\ M &= 1 - G \\ Y &= 1 - B \end{aligned} \tag{15.3}$$

The black (K) in the CMYK color space is the minimum of the C, M, and Y.

15.3.3 YIQ Color Space

The YIQ color space is similar to the YUV color space except that it uses an orthogonal quadrature I–Q-axes. The I stands for in phase and the Q means quadrature phase. This color space takes advantage of human visual color response. It is adapted by the early NTSC systems. The relationship between YUV and YIQ color spaces is defined as

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\sin(33^\circ) & \cos(33^\circ) \\ 0 & \cos(33^\circ) & \sin(33^\circ) \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}. \tag{15.4}$$

Example 15.2: MATLAB function `rgb2ntsc` converts the RGB color space data to the YIQ color space as for early NTSC standard. The MATLAB function `ntsc2rgb` converts the YIQ color space back to the RGB color space. MATLAB script `example15_2.m` demonstrates the RGB to YIQ color space conversion.

15.3.4 HSV Color Space

The HSV color space stands for hue, saturation, and value. The value is the gray level corresponding to the luminance of the YUV color space. The saturation and hue (angle position) present the UV plane in polar coordinates. The HSV function is a useful color space for publishing and art designing. The hue (or angle) determines the color on a triangle as shown in Figure 15.6. Table 15.1 lists the relationship between the hue and the color.

The saturation in HSV system represents the strength of the color. A large saturation number means the color is purer. The value is responsible for the brightness or darkness of the color. A larger value results in a brighter image.

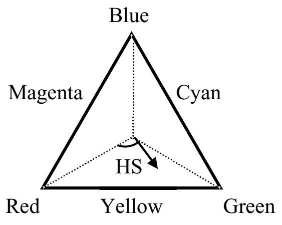


Figure 15.6 HSV triangle representation of color space

Example 15.3: MATLAB function `rgb2hsv` converts the RGB color space image to the HSV color space. The MATLAB function `hsv2rgb` converts the HSV color space back to the RGB color space. MATLAB script `example15_3.m` demonstrates the RGB–HSV color space conversion.

15.4 YC_bC_r Subsampled Color Spaces

The YC_bC_r color space can represent a color image more efficiently by subsampling the color space. This is because the human vision does not perceive the chrominance with the same clarity as luminance. Therefore, some data reduction from chrominance has little loss of visual content. Figure 15.7 shows four YC_bC_r subsampling patterns. All four schemes have the identical image resolution; that is, they produce the same size of images. However, the total numbers of bits used to represent these images are different.

YC_bC_r4:4:4 does not involve subsampling. For every Y sample, there is a C_r sample and a C_b sample associated with it. Thus, this scheme preserves the full color fidelity of the chrominance. YC_bC_r4:2:2 scheme subsamples the chrominance of YC_bC_r4:4:4 by 2, which removes half of the chrominance samples. In this scheme, every four Y samples are only associated with two C_b and two C_r samples. More chrominance samples are removed in YC_bC_r4:2:0 and YC_bC_r4:1:1 subsampling schemes. Table 15.2 lists the total number of bits needed for a 720×480 digital image. For MPEG-4 video and JPEG image compressions, YC_bC_r4:2:0 is often used to reduce the bit-stream requirement with reasonable image quality.

15.5 Color Balance and Correction

The images captured by an image sensor may not exactly represent the real scene as viewed by human eyes. The differences come from many factors such as image sensor's electronic characteristics are not the same over the entire color spectrum, lighting condition variations when the images are captured, the reflection of the object under different light sources, image acquisition system architectures, as well as display or printing devices. Therefore, color correction includes color balance and color adjustment that is necessary in digital cameras and camcorders.

Table 15.1 Hue and color relations

Color	Hue angle	RGB cube
Red	0°	(1, 0, 0)
Yellow	60°	(1, 1, 0)
Green	120°	(0, 1, 0)
Cyan	180°	(0, 1, 1)
Blue	240°	(0, 0, 1)
Magenta	300°	(1, 0, 1)

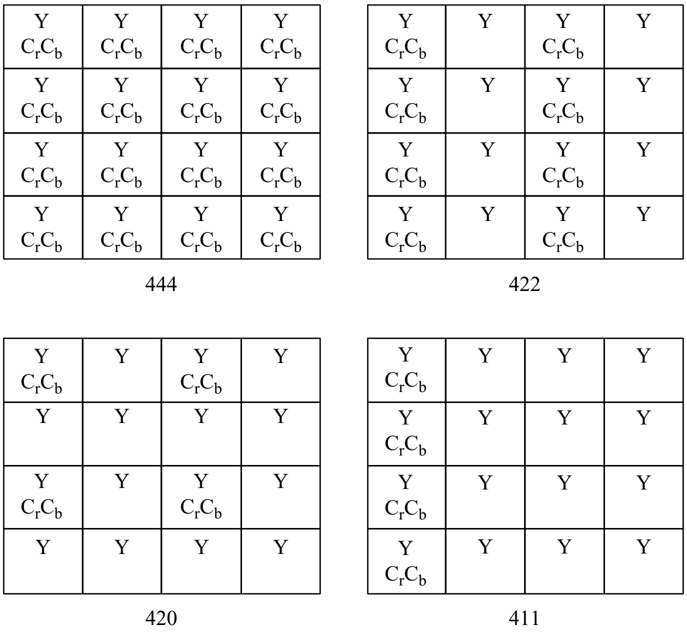


Figure 15.7 YC_bC_r subsampling schemes, YC_bC_r4:4:4, YC_bC_r4:2:2, YC_bC_r4:2:0, and YC_bC_r4:1:1

15.5.1 Color Balance

Color balance is also called white balance. The white balance is intended to correct the color bias caused by lighting and other variations in conditions. For example, a picture taken under the indoor incandescent light may appear reddish, while a picture taken under the high noon of a sunny day may appear bluish. The white balance algorithm is to mimic human vision system to adjust the images in digital cameras and camcorders. The white balance of the RGB color space can be achieved by

$$\begin{aligned} R_\mathrm{w} &= R g_\mathrm{R} \\ G_\mathrm{w} &= G g_\mathrm{G}\,, \\ B_\mathrm{w} &= B g_\mathrm{B} \end{aligned} \tag{15.5}$$

where the subscript w indicates the white balanced RGB color components, g_R, g_G, and g_B are the gain factors for red, green, and blue, respectively. The white balance algorithms can be applied in spectral domain. The spectral-based algorithm requires spectral information of the imaging sensor and

Table 15.2 Number of bits needed for the YC_bC_r subsampling schemes

729 × 480 pixels	Bits for Y	Bits for C_b	Bits for C_r	Total bits
YCrCb4:4:4	720 × 480 × 8	720 × 480 × 8	720 × 480 × 8	8 294 400
YCrCb4:2:2	720 × 480 × 8	360 × 480 × 8	360 × 480 × 8	5 529 600
YCrCb4:2:0	720 × 480 × 8	360 × 240 × 8	360 × 240 × 8	4 147 200
YCrCb4:1:1	720 × 480 × 8	180 × 480 × 8	180 × 480 × 8	4 147 200

lighting source, and thus is more accurate but computational expensive. The RGB-color-space-based white balance algorithm, on the other hand, is simple, less expensive, and easy to implement. The white balance algorithm can be implemented on the Bayer RGB or RGB color spaces. If the white balance is on the Bayer RGB, the following steps are needed:

1. Take the sensor output from the Bayer RGB pattern and save it in memory.
2. Calculate the total value of the Bayer RGB pattern data.
3. Take the reciprocal of the total value of the Bayer RGB to calculate all white balance gain factors.
4. Apply the white balance gain factors to the Bayer RGB data.

In order to obtain the accurate gain factors, the RGB data must contain a rich spectrum of colors. That is, too few colors will result in false white balance gain factors. For example, the gain factor's calculation will be undesirable if the RGB color space consists of red color only.

Example 15.4: MATLAB function `imread` can read different image files including the widely used JPEG, TIF, GIF, BMP, and PNG formats. The `imread` function will return a three-dimensional (3-D) array for color images and a 2-D array for grayscale images. For most images, the image array is converted into 8-bit RGB from these files. In real applications, the white balance gains are usually normalized to G pixels or 1. The following MATLAB script computes the white balance gains from the RGB array:

```
R = sum(sum(RGB(:,:,1))); % Compute sum of R
G = sum(sum(RGB(:,:,2))); % Compute sum of G
B = sum(sum(RGB(:,:,3))); % Compute sum of B
gr = G/R;                 % Gain factor for R is normalized to G
gb = G/B;                 % Gain factor for B is normalized to G
Rw=RGB(:,:,1)*gr;         % Apply normalized gain factor to R
Gw=RGB(:,:,2);            % G has a gain factor of 1
Bw=RGB(:,:,3)*gb;         % Apply normalized gain factor to B
```

15.5.2 Color Adjustment

The RGB color in a digital camera or camcorder may not be the same as seen by human eyes. Chromatic correction can compensate the color offset to make the digital images close to what human beings see. The chromatic correction is also called color correction or color saturation correction. The color correction applies a 3 × 3 matrix to the white balanced RGB color space as

$$\begin{bmatrix} R_C \\ G_C \\ B_C \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \begin{bmatrix} R_w \\ G_w \\ B_w \end{bmatrix}, \tag{15.6}$$

where

$$\min\left\{\sum_{i=1}^{3}\sum_{j=1}^{3}\left[c_{ij} \times RGB(j)_w - RGB(j)_{ref}\right]^2\right\} \quad \text{for } i \neq j \tag{15.7}$$

$$c_{ij} = 1 \quad \text{for } i = j. \tag{15.8}$$

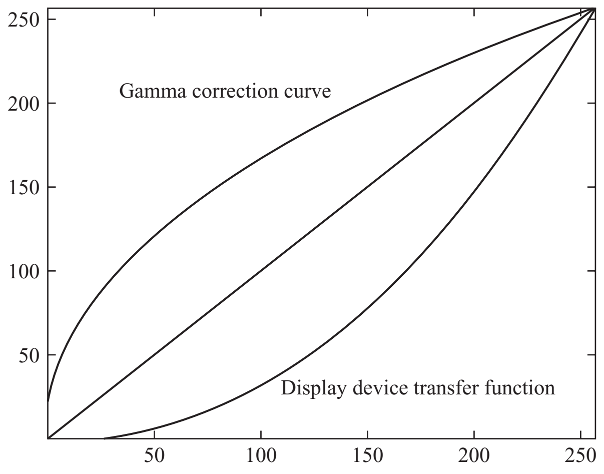


Figure 15.8 Gamma correction curve vs. display device transfer function

The 3 × 3 matrix in Equation (15.6) is called the color correction matrix. Equation (15.7) states that the color correction is achieved by adjusting the coefficients c_{ij} (rotating the color vector) to find an optimal 3 × 3 color correction matrix that minimizes the mean-square error between the white balanced RGB color space and the reference RGB color space. In order to preserve the white balance and luminance, the diagonal elements of the matrix need to be normalized to 1, as defined in Equation (15.8).

15.5.3 Gamma Correction

The electrical circuits of the televisions and computer monitors do not produce linear output with the linear RGB input. When we send a color image directly to a TV, the displayed color will be dimmer than the original image. This is because the display of TV monitors follows a gamma curve. The gamma correction compensates for the nonlinearity of display devices to obtain a linear response of display. Figure 15.8 shows the gamma correction curve vs. the nonlinearity of the display devices. In order to display the image in linear output, the gamma correction prewraps the input RGB data with different gamma values to compensate for the display device's nonlinear characteristics. The gamma values for TV systems are specified in ITU CCIR Report 624-4. Microsoft Window computer monitors use gamma value of 2.20, while the Apple Power-PC monitors have gamma value of 1.80. For digital cameras and video camcorders, the gamma correction is usually implemented using a table-lookup method. The gamma correction table is precalculated and stored in the device's memory.

For an 8-bit RGB sample, a 256-word table is needed. The formulas for gamma correction are given as

$$\begin{aligned} R_\gamma &= gR_{\mathrm{c}}^{1/\gamma} \\ G_\gamma &= gG_{\mathrm{c}}^{1/\gamma}, \\ B_\gamma &= gB_{\mathrm{c}}^{1/\gamma} \end{aligned} \tag{15.9}$$

where g is the conversion gain factor, γ is the gamma value, and the R_{c}, G_{c}, and B_{c} form the color corrected RGB color space. The gamma corrected RGB color space is denoted by R_γ, G_γ, and B_γ.

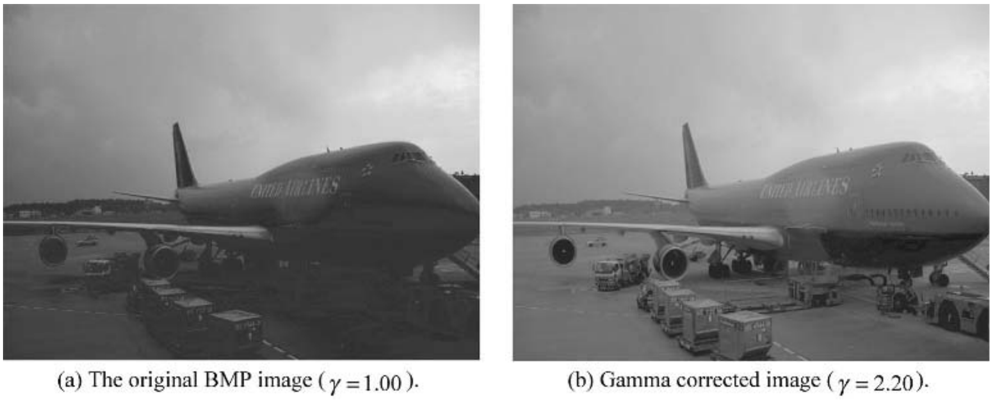

(a) The original BMP image ($\gamma = 1.00$).
(b) Gamma corrected image ($\gamma = 2.20$).

Figure 15.9 Gamma corrected image vs. the original image: (a) the original BMP image ($\gamma = 1.00$); (b) gamma corrected image ($\gamma = 2.20$)

Example 15.5: Create an 8-bit gamma ($\gamma = 2.20$) correction table as shown in Figure 15.8, and apply the gamma correction to the given bitmapped (BMP) image ($\gamma = 1.00$). The images before and after gamma correction are shown in Figure 15.9 using the MATLAB script `example15_5.m`.

15.6 Image Histogram

A digital image can be analyzed using histogram, which represents an image's pixel distribution. For an 8-bit image, the histogram will have 256 entries. The first entry shows the total number of pixels in the image that equals to '0', the second entry shows the numbers of pixels that equal to '1', and so on. The sum of all of the values in the histogram equals to the total number of the pixels in the image, expressed as

$$N = \sum_{i=0}^{M-1} h_i, \tag{15.10}$$

where M is the number of entries in histogram, h_i is the histogram, and N is the total number of pixels of the image. The histogram counts the image pixels that have the same values, and can be efficiently computed by DSP processors.

Histogram is an important characteristic of digital images. Since an image may contain millions of pixels, we can compute the mean m_x and the standard deviation σ_x^2 of image easily using its histogram as follows:

$$m_x = \frac{1}{N} \sum_{i=0}^{M-1} i h_i \tag{15.11}$$

$$\sigma_x^2 = \frac{1}{N-1} \sum_{i=0}^{M-1} (i - m_x)^2 h_i . \tag{15.12}$$

Proper brightness and contrast will make the image easy to view. The brightness is the overall luminance level of the image, and the contrast is the difference in brightness. For a digital camera, the brightness is associated with the exposure when the picture is taken. Brightness adjustments can improve viewing ability of the darker or brighter areas by increasing or decreasing the luminance value of each pixel. Since the modification of brightness changes every pixel in the image, the entire image will become brighter or darker. Saturation may occur with the brightness adjustment. Changing brightness of an image will not affect the contrast of the image. Contrast can be adjusted by varying the luminance value of each pixel. The adjustment to the contrast may also result in saturation if the pixel values reach the limit.

Histogram equalization uses a monotonic nonlinear mapping process to redistribute the intensity values of an image such that the resulting image has a much uniform distribution of intensities. Histogram equalization may not work well on all images because the redistribution does not use the priori knowledge of the image. Sometimes, it may even make the image worse. However, histogram equalization works well especially for fine details in the darker regions or for B&W images. The histogram equalization consists of the following three steps:

1. Compute the histogram of the image.
2. Normalize the histogram.
3. Normalize the image using the normalized histogram.

Example 15.6: This example computes the histogram of a given image and equalizes this image based on its histogram. Figure 15.10 shows the original image, the equalized image, and their histograms using the MATLAB script (`example15_6.m`).

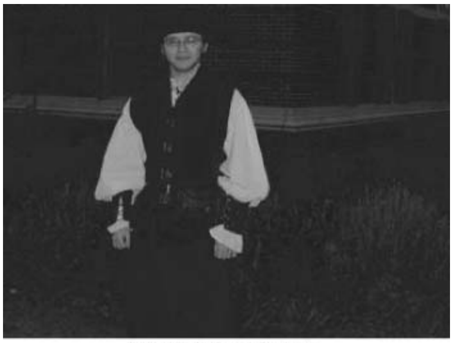
As shown in the figure, the original image is very dark because it was taken in the evening without using flashlight. Most of the pixels are concentrated in the lower portion of the histogram as shown in Figure 15.10(c). The viewing quality of this image can be improved by increasing the contrast using histogram equalization. Figure 15.10(b) shows that most of the dark background scene has been clearly revealed. Figure 15.10(d) is the histogram of the equalized image. The histogram of the equalized image redistributes the pixel values more evenly by adjusting the contrast level.

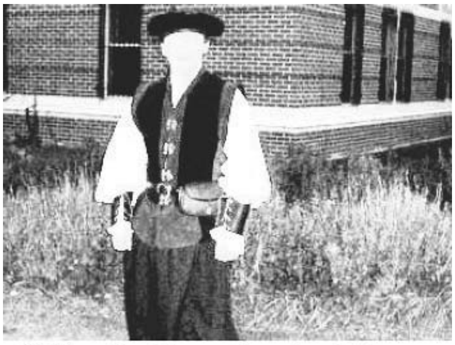
Example 15.6 shows that the histogram equalization is an automated process to replace the trial-and-error manual adjustment. It is an effective way to enhance the contrast. However, since the histogram equalization applies an approximated uniform histogram blindly, some undesired effects could occur. For example, in Figure 15.10(b), the facial details are lost after the histogram equalization although the surrounding scene is enhanced. This problem may be overcome by using an adaptive histogram equalization, which divides the image to several regions and individually equalizes each smaller region instead of entire image. To reduce the artifacts caused by the boundaries of the small regions, additional smooth process should be used.

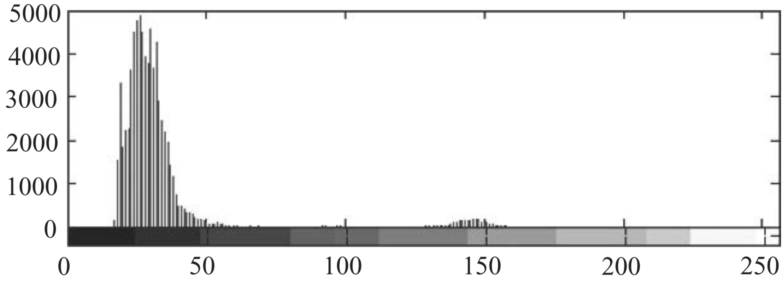
15.7 Image Filtering

Many video and image applications use 2-D filters for image processing. If the input to the system is an impulse (delta) function $\delta(x,\ y)$ at the origin, the output is the system's impulse response. When the system's response remains the same regardless of the position of the impulse function, the system is defined as linear space-invariant system. A linear space-invariant system can be described by its impulse response as Figure 15.11.

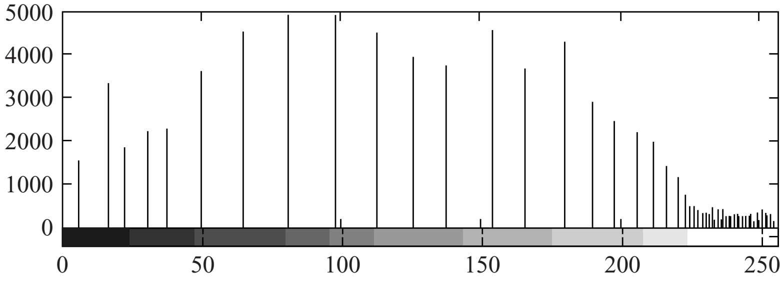
An image processing system is a 2-D system, and thus an image filtering uses a 2-D filter. Similar to a 1-D FIR filtering, the image filtering is a 2-D convolution of the filter kernel with the image, which can


(a) Original image.


(b) Equalized image based on histogram.



(c) Histograms of the original image.



(d) Histogram of the equalized image.

Figure 15.10 Image equalization based on its histogram: (a) original image; (b) equalized image based on histogram; (c) histograms of the original image; and (d) histogram of the equalized image

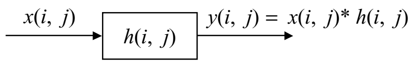


Figure 15.11 A linear space-invariant 2-D system

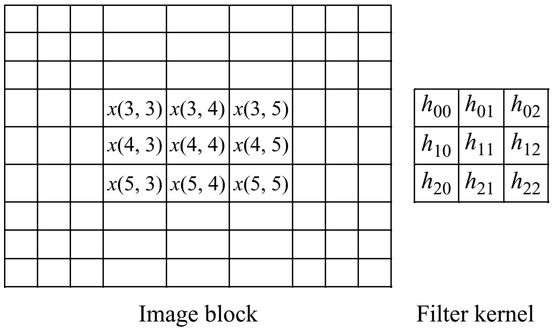


Figure 15.12 An example of 3 × 3 image convolution: (a) image block; (b) filter kernel

be expressed as

$$y(n,\ m) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} h(i,\ j)x(n-i,\ m-j), \tag{15.13}$$

where M and N are the width and height of the 2-D filter, respectively. Figure 15.12 shows an example of 3 × 3 image convolution expressed as

$$\begin{aligned} y(4,4) = {} & h_{0,0}x(3,3) + h_{0,1}x(3,4) + h_{0,2}x(3,5) \\ & + h_{1,0}x(4,3) + h_{1,1}x(4,4) + h_{1,2}x(4,5) \\ & + h_{2,0}x(5,3) + h_{2,1}x(5,4) + h_{2,2}x(5,5) \end{aligned}$$

where the image data is denoted as $x(n, m)$, and the filter kernel is denoted by $h_{i,j}$.

Image filtering can affect the digital image in many ways, such as reducing the noise, enhancing the edges, sharpening the image, and blurring the image. Image filtering can also generate many special effects for the digital photos. Linear filters are widely used in image processing and editing. Linear smoothing (or lowpass) filters are effective in reducing noises. The common noises in images are Gaussian noise introduced by camera's image sensors, impulse noise caused by sudden intensity change in white values (may also result from bad image sensor), and B&W high-frequency noises called salt and pepper noises. The linear filters usually use weighted coefficients. Typically, the sum of the filter coefficients equals to 1 in order to keep the same image intensity. If the sum is larger than 1, the resulting image will be brighter; otherwise, it will be darker. Some commonly used 2-D filters are the 3 × 3 kernels summarized as follows:

(a) *Delta filter*: $h_{i,j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

(b) *Lowpass filter*: $h_{i,j} = \dfrac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$.

(c) *Highpass filter*: $h_{i,j} = \frac{1}{6}\begin{bmatrix} -1 & -4 & -1 \\ -4 & 26 & -4 \\ -1 & -4 & -1 \end{bmatrix}$.

(d) *Sobel filter*: $h_{i,j} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$.

(e) *Laplacian filter*: $h_{i,j} = \begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}$.

(f) *Emboss filter*: $h_{i,j} = \begin{bmatrix} -4 & -4 & 0 \\ -4 & 1 & 4 \\ 0 & 4 & 4 \end{bmatrix}$.

A 3×3 lowpass filter (b) is called mean filter. An important note for a smoothing filter is that the filter kernel must have only one peak value and must be symmetric in both horizontal and vertical directions. A highpass filter (c) can be obtained by subtracting the lowpass filter kernel from the delta filter kernel. In real applications, highpass filters are often used for image sharpening.

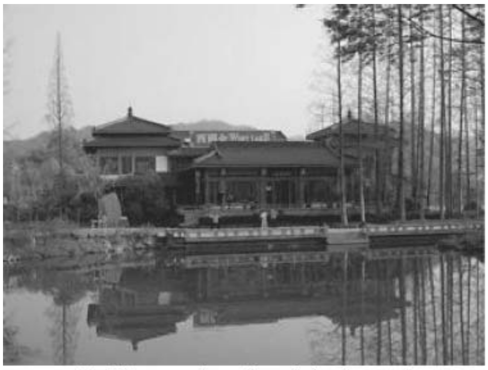
Edges have sharp changes of local intensity level. Edge detection is the first and the key step in image analysis and recovery. For machine vision, edge detection is used to determine the objects. Sobel filter kernel is widely used as an edge filter. Similar to the Sobel kernel, Prewitt filter kernel can be used for edge filtering. However, both operators can apply only to one direction at a time. The horizontal Sobel kernel (d) can be rotated by 90^o to obtain the vertical kernel. A vertical kernel using the Prewitt operator is given in MATLAB script `example15_7.m`. Laplacian kernel (e) is a 2-D operator that checks the numbers of zero crossing.

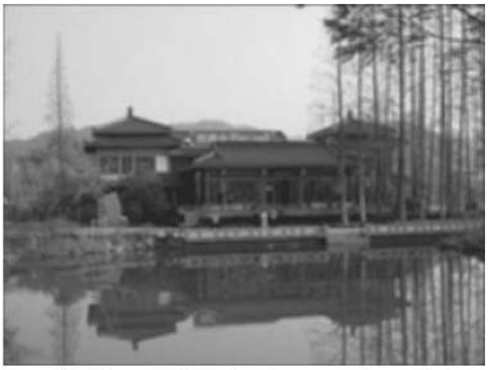
Example 15.7: MATLAB *Image Processing Toolbox* provides the function `filter2` to implement a 2-D filtering using 2-D correlation method. The examples of using `filter2` are:

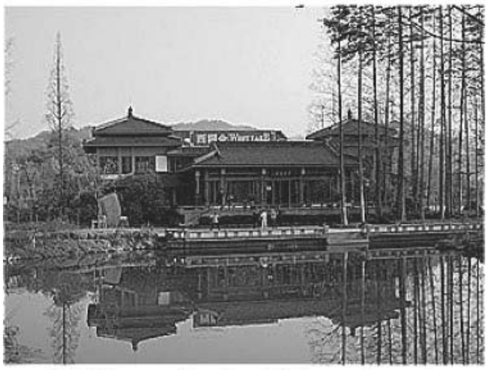
```
R = filter2(coeff, RGB(:,:,1));
G = filter2(coeff, RGB(:,:,2));
B = filter2(coeff, RGB(:,:,3));
```

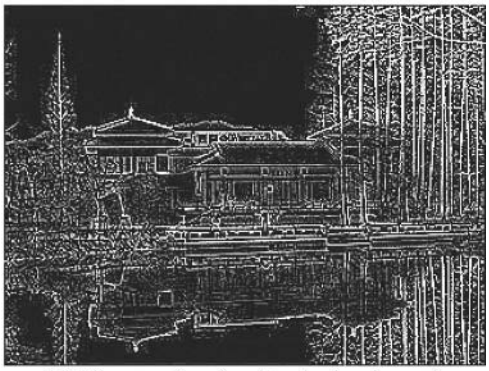
where `coeff` is the 2-D filter kernel, `RGB` is the input image matrix, and `R`, `G`, and `B` are the arrays of filtered output of the RGB components. MATLAB also provides the built-in function `imfilter` for filtering images. The `imfilter` can replace the `filter2` function with the following syntax: `newRGB = imfilter(RGB, coeff);` The 2-D filtering results are shown in Figure 15.13.

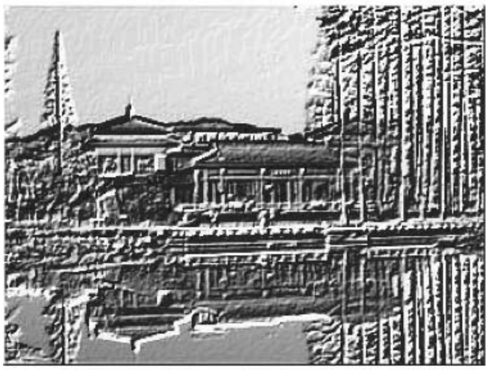
Figure 15.13(a) shows the image after applying delta function. The output image remains unchanged when it uses delta filter kernel. The lowpass filter averages the image pixels so the filtered image is smoothed as shown in Figure 15.13(b). The lowpass filter is often used to reduce the noise in images. Figure 15.13(c) is the result of highpass filter, which emphasizes the high-frequency components to increase its sharpness. The Laplacian filter sharpens the edges as shown in Figure 15.13(d), where the edges of output image are highlighted. Figure 15.13(e) shows the Emboss filter creating a 3-D like image. A Sobel filter kernel emphasizes the image's horizontal edge as shown in Figure 15.13(f), but the edges in vertical direction are not being emphasized. Since the sum of the filter kernel equals to zero, the output image is very dark with only the edges being highlighted. When placing the filter coefficients in the


(a) The result using delta kernel.


(b) The result using lowpass kernel.


(c) The result using highpass kernel.


(d) The result using Laplacian kernel.


(e) The result using Emboss kernel.

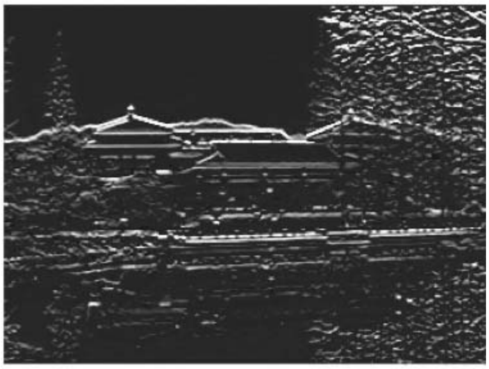
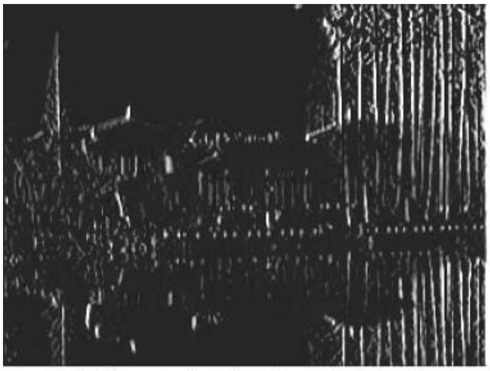

(f) The result using Sobel kernel.

Figure 15.13 Results of 2-D image filtering using different 3 × 3 filter kernels: (a) the result using delta kernel; (b) the result using lowpass kernel; (c) the result using highpass kernel; (d) the result using Laplacian kernel; (e) the result using Emboss kernel; (f) the result using Sobel kernel; (g) the result using Prewitt kernel; and (h) the result using blur kernel


(g) The result using Prewitt kernel.

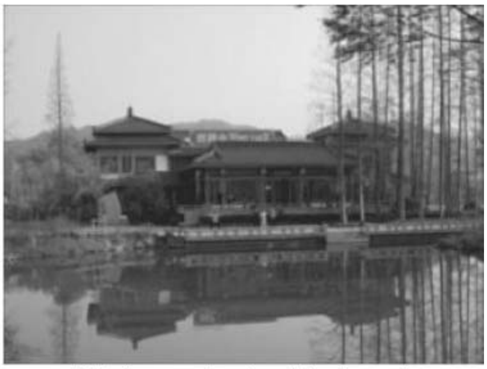

(h) The result using blur kernel.

Figure 15.13 *(continued)*

vertical direction, the filter can be used to emphasize the vertical edges of the image. Figure 15.13(g) shows that the vertical edges are emphasized using a Prewitt filter. Figure 15.13(h) is the result of using a blur filter kernel, which blurs the image by averaging the four neighboring pixels with the center pixel.

When implementing a fixed-point filter for image processing, the overflow problem must be carefully handled. For example, when an image with 8-bit data is processed, the pixel value must be limited to 255 if the filtered output is greater than 255. The data must be set to zero when the filter output is negative.

The implementation of a 2-D filter requires four nested loops on each pixel for every coefficient of the filter kernel. Therefore, the image filtering is a very computational intensive process in image applications. For many real-time image and video applications, it is limited to low-order filter kernels such as 3×3 or 5×5.

15.8 Image Filtering Using Fast Convolution

A 720×480 image's RGB color space consists of $720 \times 480 \times 3$ data samples. For the NTSC motion video at 30 frames per second, there will be $720 \times 480 \times 3 \times 30 = 31\,104\,000$ bytes/s. As the image filtering requires four nested loops for each pixel and every filter coefficient in the kernel, this can be very computational intensive for large-size image even by using a 3×3 filter. An alternative approach is to use fast convolution method. The 2-D fast convolution using the 2-D FFT is considerably fast for filtering a large-size image because the spatial-domain convolution becomes multiplication in frequency domain. If the image signal is orthogonal, we can further reduce computational requirements by applying filter in one direction at a time.

Given a function $f(m, n)$ of two spatial variables m and n, the 2-D M-by-N DFT $F(k, l)$ is defined as

$$F(k, l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) e^{-j(2\pi/M)km} e^{-j(2\pi/N)ln}, \tag{15.14}$$

where k and l are frequency indices for $k = [0, M-1]$ and $l = [0, N-1]$, respectively. The 2-D inverse DFT is defined as

$$f(n, m) = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} F(k, l) e^{j(2\pi/M)km} e^{j(2\pi/N)ln}. \tag{15.15}$$

MATLAB *Image Processing Toolbox* provides a 2-D FFT function `fft2` and an inverse FFT function `ifft2`. The 2-D fast convolution can be computed using the following steps:

1. The filter coefficient matrix and image data matrix must have the same dimensions. If one matrix is smaller than the other, we can pad zeroes on the smaller matrix.

2. Compute the 2-D FFT of both the image and coefficient matrices using the MATLAB function `fft2`.

3. Multiply the frequency-domain coefficient matrix with image matrix using dot product.

4. Apply the inverse 2-D FFT function `ifft2` to obtain the filtered results.

Example 15.8: Instead of performing 2-D convolution, the frequency-domain filtering uses multiplication. The use of MATLAB functions `fft2` and `ifft2` on RGB images can be expressed as

```
fft2R = fft2(double(RGB(:,:,1)));
[imHeight imWidth] = size(fft2R);
fft2Filt = fft2(coeff, imHeight, imWidth);
fft2FiltR = fft2Filt .* fft2R;
newRGB(:,:,1) = uint8(ifft2(fft2FiltR));
```

Here, `fft2FiltR` is the 2-D fast convolution result of the red components. The filter coefficients are zero padded and the frequency-domain matrix is obtained using the `fft2` function. The symbol '`.*`' in the fourth line of MATLAB script is the dot-product operator. Figure 15.14 shows the 2-D filtering results.

The fast convolution results (Figure 15.14) show that when dealing with a large image, the filtering can be performed in frequency domain. The computational requirement for fast convolution is much less than the computation in the spatial domain using the 2-D convolution.

15.9 Practical Applications

Image processing has been used in many practical applications. More and more images we have today become larger and larger in size. Therefore, image compression plays a key role in image applications. Many international standards, such as JPEG, are commonly used for efficient storage and transmission. JPEG can achieve up to 10:1 compression ratio.

15.9.1 JPEG Standard

JPEG compression is defined in the ITU-T Recommendation T.81. JPEG file format has been widely used in printing, digital photography, video editing, security, and medical imaging applications. Figure 15.15 shows the basic functions of JPEG encoder.

In JPEG encoding process, the image pixels are grouped into 8×8 blocks. Each block is transformed by a forward DCT to obtain 64 DCT coefficients. The first coefficient is called the DC coefficient, and the rest coefficients are referred as AC coefficients. These 64 DCT coefficients are quantized by a quantizer using one of 64 corresponding values from a quantization table specified by ITU-TT.81.

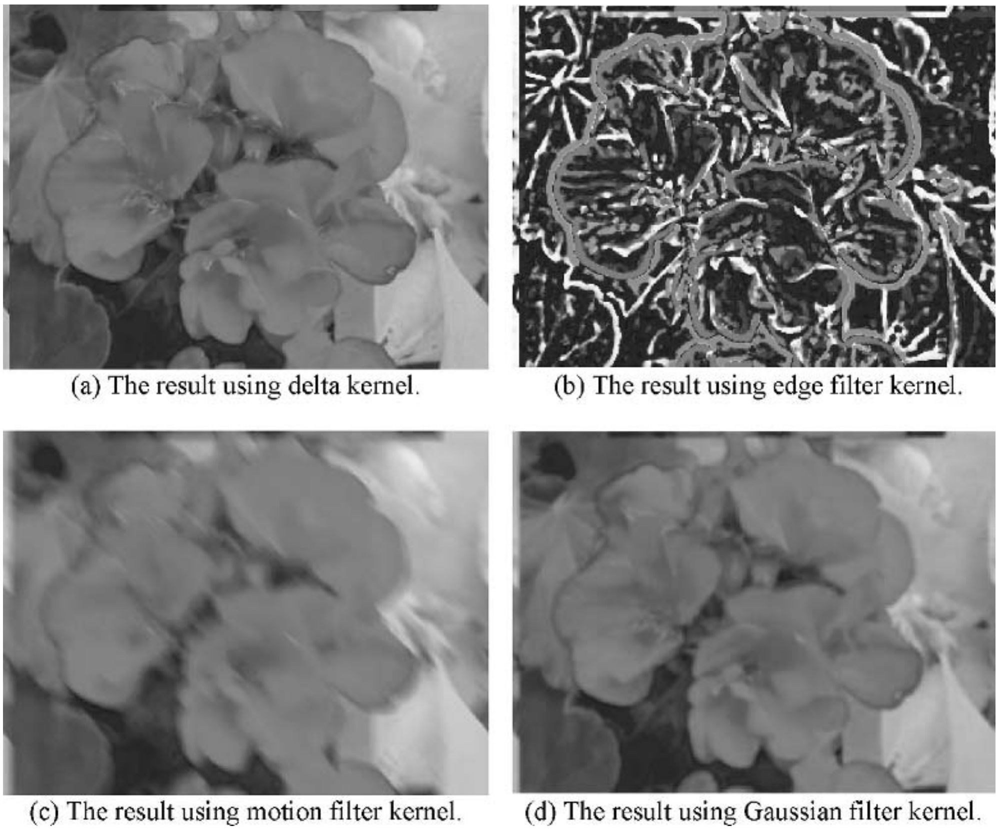

(a) The result using delta kernel.
(b) The result using edge filter kernel.
(c) The result using motion filter kernel.
(d) The result using Gaussian filter kernel.

Figure 15.14 Results of 2-D image filtering using fast convolution: (a) the result using delta kernel; ((b) the result using edge filter kernel; (c) the result using motion filter kernel; and (d) the result using Gaussian filter kernel

Instead of quantizing the current DC coefficient, JPEG computes the differences between the previous quantized DC coefficient and the current DC value, and encodes the difference. The rest of the 63 AC coefficients are quantized. These 64 quantized DCT coefficients are arranged as a 1-D zigzag sequence shown in Figure 15.16. The quantized and reordered coefficients are then passed to an entropy coder that further compresses the coefficients.

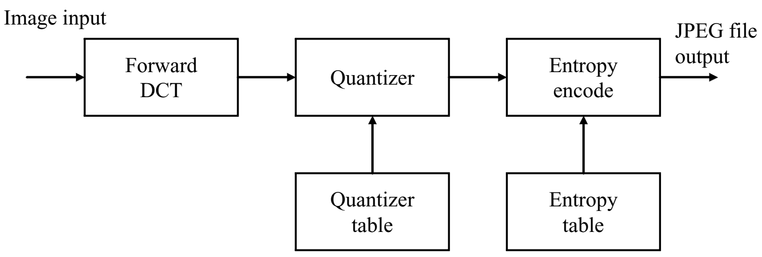


Figure 15.15 Baseline JPEG encoder block diagram

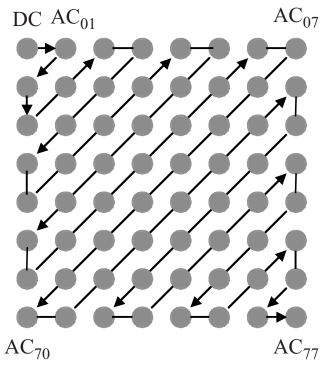


Figure 15.16 Reordering of zigzag DCT coefficients

There are two entropy-coding procedures defined by JPEG standard: Huffman encoding and arithmetic encoding. Each encoding method uses its own encoding table as specified by T.81 standard. The baseline JPEG is a sequential DCT-based operation. The image pixels are arranged as 8×8 blocks. We process one block at a time from left to right for the first row of blocks, repeat the processing for the second row of blocks, and so on from top to bottom. After the last data block has been processed by the forward DCT and quantization, the 64 quantized DCT coefficients can be entropy encoded and output as compressed image bit stream.

15.9.2 2-D Discrete Cosine Transform

Two-dimensional DCT and IDCT are important algorithms in many image and video compression techniques including the JPEG standard. The 2-D DCT and IDCT of $N \times N$ image are defined as

$$F(u,\ v) = \frac{2}{N}C(u)C(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,\ y)\cos\left[\frac{(2x+1)u\pi}{2N}\right]\cos\left[\frac{(2y+1)v\pi}{2N}\right], \tag{15.16}$$

$$f(x,\ y) = \frac{2}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} C(u)C(v)F(u,\ v)\cos\left[\frac{(2x+1)u\pi}{2N}\right]\cos\left[\frac{(2y+1)v\pi}{2N}\right], \tag{15.17}$$

where $f(x,\ y)$ is the pixel intensity and $F(u,\ v)$ is the corresponding DCT coefficient at the image location $(x,\ y)$, and

$$C(u) = C(v) = \begin{cases} \sqrt{2}/2, & u = v = 0 \\ 1, & \text{otherwise} \end{cases}.$$

Most image compression algorithms use $N = 8$, and the 8×8 DCT and IDCT specified in the ITU-TT.81 are expressed as

$$F(u,\ v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,\ y)\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right] \tag{15.18}$$

$$f(x,\ y) = \frac{1}{4}\sum_{x=0}^{7}\sum_{y=0}^{7} C(u)C(v)F(u,\ v)\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]. \tag{15.19}$$

The 2-D DCT and IDCT can be implemented using two separate 1-D operations: one for horizontal direction and the other for vertical direction. Using efficient 1-D DCT and IDCT, the computation can be reduced dramatically. The 1-D 8-point DCT and IDCT can be implemented using the following equations:

$$F(u) = \frac{1}{2}C(u)\sum_{x=0}^{7} f(x)\cos\left[\frac{(2x+1)u\pi}{16}\right] \tag{15.20}$$

$$f(x) = \frac{1}{2}\sum_{x=0}^{7} C(u)F(u)\cos\left[\frac{(2x+1)u\pi}{16}\right], \tag{15.21}$$

where

$$C(u) = \begin{cases} \sqrt{2}/2, & u = 0 \\ 1, & \text{otherwise} \end{cases}.$$

> *Example 15.9:* MATLAB provides both 1-D and 2-D DCT and IDCT functions. The use of 1-D functions given by Equations (15.20) and (15.21) to transform an image is identical to the use of the 2-D functions defined in Equations (15.18) and (15.19). The MATLAB implementation is given in script `example15_9.m`.

JPEG applies the DCT to an image in a block of 8 × 8 (64) pixels at a time. This is called a block transform. The resulting DCT coefficients are then quantized by the entropy encoder and reordered in a zigzag fashion. This process is shown in Figure 15.17.

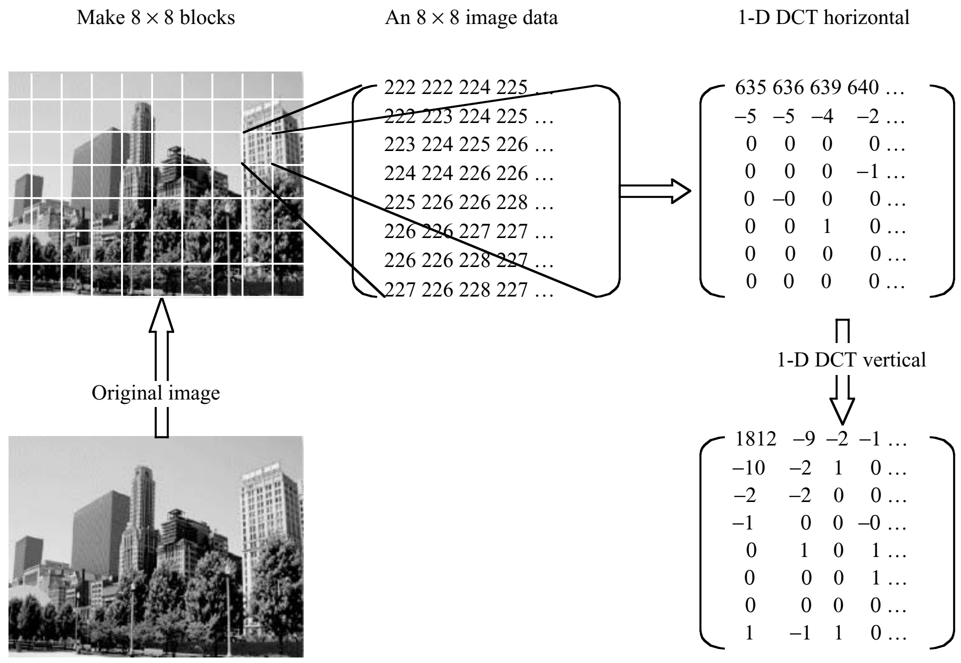


Figure 15.17 A DCT block transform in JPEG coding process

Example 15.10: JPEG encoding process divides the image into many smaller 8×8 blocks, and then apply the DCT on these 8×8 image blocks. The following MATLAB code performs the block DCT:

```
for n=1:8:imHeight
  for m=1:8:imwidth
    for i=0:7
      for j=0:7
        mbY(i+1,j+1) = Y(n+i,m+j,1);  % Form an 8x8 block
      end
    end
    mbY = dct(double(mbY));           % Perform 1-D DCT horizontally
    mbY = dct(double(mbY'));          % Perform 1-D DCT vertically
  end
end
```

The use of 1-D functions `dct` and `idct` to perform DCT is given in script `example15_10.m`.

15.10 Experiments and Program Examples

This section uses MATLAB, CCS, and C5510 DSK for experiments with digital images. The speed is usually very critical for processing digital images, especially for real-time video, and the finite wordlength effects are also very important for fixed-point implementation. MATLAB supports Link for CCS including the TMS320C55x, which can prepare image data files. We will use MATLAB Link for CCS for most of the experiments using the C5510 DSK.

15.10.1 YC_bC_r to RGB Conversion

The color space conversion between the RGB and the YC_bC_r introduced in Section 15.3 will be implemented in this experiment using the C5510 DSK. Since the 24-bit RGB color space is widely used by today's image applications, we will implement the 8-bit YC_bC_r to RGB color space conversion using the fixed-point C language. The RGB to YC_bC_r color space conversion will be introduced in the next experiment.

The conversion matrix is expressed as

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.046 & 0 & 0.0063 \\ 0.046 & -0.0015 & -0.0032 \\ 0.046 & 0.0079 & 0 \end{bmatrix} \begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}, \qquad (15.22)$$

which is used by MATLAB function `ycbcr2rgb`. When choosing the Q15 format for the conversion coefficients in the fixed-point implementation, the conversion matrix can be represented as

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 151 & 0 & 206 \\ 151 & -49 & -105 \\ 151 & 259 & 0 \end{bmatrix} \begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}.$$

Because the YC_bC_r uses only an 8-bit data, the wordlength of the conversion matrix will affect accuracy and cumulative errors mitigation during the conversion process. In order to achieve higher precisions and minimize finite wordlength effects, we shall use a data format that provides adequate integer range for coefficients. In this experiment, we represent the fixed-point coefficients by 24-bit integers in the range from 0x800000 (–1.0) to 0x7FFFFF (1.0–2^{-23}). The conversion coefficient matrix using 24-bit integer is expressed as

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 4823 & 0 & 6606 \\ 4823 & -1573 & -3355 \\ 4823 & 8284 & 0 \end{bmatrix} \begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}.$$

The result of conversion must be scaled back to maintain the proper 8-bit RGB data format. Since two 8-bit data can be packed into one 16-bit memory location, this experiment uses packed data format and processes two pixels at a time. To prevent overflow and underflow, the upper and lower limits are set to 255 and 0, respectively. Table 15.3 lists the C function for the YC_bC_r to RGB conversion.

Table 15.3 Fixed-point C function to convert YC_bC_r to RGB color space

```
#define SHIFT 12
#define UINT (0x100000-1)

static short COEF_YCbCr2RGB[7] = {
    (short)( 0.00456621*UINT+0.5),(short)( 0.00625893*UINT+0.5),
    (short)( 0.00456621*UINT+0.5),(short)(-0.00153632*UINT-0.5),
    (short)(-0.00318811*UINT-0.5),(short)( 0.00456621*UINT+0.5),
    (short)( 0.00791071*UINT+0.5) };

void ycbcr2rgb_cdsp(ycbcr2rgbImg img)
{
   long AC0, AC1;
   short *cPtrB;
   unsigned short *r,*g,*b,*y,*cb,*cr;
   unsigned short i,j,len;
   short yhi,ylo,cbhi,cblo,crhi,crlo;

   len = img.width>>1;
   y = img.y;
   cb = img.cb;
   cr = img.cr;
   r = img.r;
   g = img.g;
   b = img.b;
   // This implementation is very sensitive to overflow and underflow
   // R,G,B must be limited to the range of [0, 255]
   for (j=0; j<img.height;j++)
   {
      for (i=0;i<len;i++)
      {
         cPtrB = COEF_YCbCr2RGB;
         ylo  = ((*y>>8)&0xff) - 16;
         yhi  = ((*y++)&0xff) - 16;
         cblo = ((*cb>>8)&0xff) - 128;
         cbhi = ((*cb++)&0xff) - 128;
         crlo = ((*cr>>8)&0xff) - 128;
         crhi = ((*cr++)&0xff) - 128;
```

Table 15.3 (*continued*)

```
            AC0  = (long)yhi   * *cPtrB;
            AC1  = (long)ylo   * *cPtrB++;
            AC0  += (long)crhi * *cPtrB;
            AC1  += (long)crlo * *cPtrB++;
            AC0  >>= SHIFT;
            AC1  >>= SHIFT;
            if (AC0 < 0) AC0 = 0;
            if (AC1 < 0) AC1 = 0;
            if (AC0 > 255) AC0 = 255;
            if (AC1 > 255) AC1 = 255;
            *r++  = (short)( AC0 | (AC1<<8) );
            AC0   = (long)yhi  * *cPtrB;
            AC1   = (long)ylo  * *cPtrB++;
            AC0  += (long)cbhi * *cPtrB;
            AC1  += (long)cblo * *cPtrB++;
            AC0  += (long)crhi * *cPtrB;
            AC1  += (long)crlo * *cPtrB++;
            AC0  >>= SHIFT;
            AC1  >>= SHIFT;
            if (AC0 < 0) AC0 = 0;
            if (AC1 < 0) AC1 = 0;
            if (AC0 > 255) AC0 = 255;
            if (AC1 > 255) AC1 = 255;
            *g++ = (short)( AC0 | (AC1<<8) );
            AC0  = (long)yhi  * *cPtrB;
            AC1  = (long)ylo  * *cPtrB++;
            AC0  += (long)cbhi* *cPtrB;
            AC1  += (long)cblo* *cPtrB++;
            AC0  >>= SHIFT;
            AC1  >>= SHIFT;
            if (AC0 < 0) AC0 = 0;
            if (AC1 < 0) AC1 = 0;
            if (AC0 > 255) AC0 = 255;
            if (AC1 > 255) AC1 = 255;
            *b++ = (short)( AC0 |  (AC1<<8) );
        }
    }
    return;
}
```

This experiment takes in 8-bit YC_bC_r data and converts the image to the RGB color space. After the conversion, a bitmap file will be created using the resulting RGB data for display. Table 15.4 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Open the DSP project and rebuild the project.

2. Run the project to convert the given YC_bC_r data files located in `..\data` directory to RGB data files.

3. Display the resulting bitmap file to view the conversion result.

Table 15.4 File listing for experiment `exp15.10.1_ycbcr2rgb`

Files	Description
`YCbCr2RGB.c`	YC_bC_r to RGB color space conversion function
`YCbCr2RGBTest.c`	Main program for testing experiment
`RGB2BMPHeader.c`	C function uses RGB data to create a BMP file
`ycbcr2rgb.h`	C header file
`ycbcr2rgb.cmd`	DSP linker command file
`param.txt`	Parameter file
`ycbcr2rgb.pjt`	DSP project file
`Jenni160x120Y8.YUV`	Y component data file
`Jenni160x120Cb8.YUV`	C_b component data file
`Jenni160x120Cr8.YUV`	C_r component data file

4. Replace the conversion matrix given in Equation (15.22) with a Q15-valued matrix, and evaluate the conversion results in different wordlengths.

5. Rewrite the C function `ycbcr2rgb( )` with C55x intrinsics to improve the run-time efficiency of conversion function. Verify the intrinsics implementation by comparing the results with the fixed-point C function.

6. Write the `ycbcr2rgb( )` function using C55x assembly language. Verify the correctness by examining the bit-exactness of resulting R, G, and B data with the results obtained using C function.

7. Measure the run-time benchmark of the `ycbcr2rgb( )` function written by fixed-point C, intrinsics, and assembly language using the CCS profile tool. What percentage of improvement can be achieved by using the intrinsics over the fixed-point C function? What percentage of gain the assembly routine implementation can achieve over the fixed-point C function?

15.10.2 Using CCS Link with DSK and Simulator

In Chapter 9, we have shown how to use MATLAB Link for CCS to control and manipulate the DTMF experiments with the C5510 DSK. This experiment uses the Link for CCS to conduct the RGB to YC_bC_r color space conversion using the C5510 DSK. The flow of experiment is shown in Figure 15.18.

The MATLAB script `ccsLink.m` used for this experiment is listed in Table 15.5. The top portion of the script uses MATLAB *Image Processing Toolbox* functions to open the given image file for experiment. The bottom portion of the script displays the image after conversion. Using the MATLAB image functions

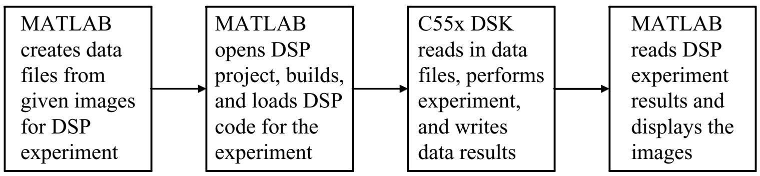

MATLAB creates data files from given images for DSP experiment
MATLAB opens DSP project, builds, and loads DSP code for the experiment
C55x DSK reads in data files, performs experiment, and writes data results
MATLAB reads DSP experiment results and displays the images

Figure 15.18 MATLAB Link for CCS experiment flow

Table 15.5 MATLAB script for converting RGB to YC_bC_r

```
RGB = imread(name);                           % Read in image data
RGB = uint8(RGB);                             % Convert to uint8
[height, width, color] = size(RGB);

fid=fopen('.\\ccsLink\\data\\R8.RGB', 'wb');       % Write RGB data out
fwrite(fid, RGB(:,:,1)', 'uint8'), fclose(fid);   % Write R
fid=fopen('.\\ccsLink\\data\\G8.RGB', 'wb');
fwrite(fid, RGB(:,:,2)', 'uint8'), fclose(fid);   % Write G
fid=fopen('.\\ccsLink\\data\\B8.RGB', 'wb');
fwrite(fid, RGB(:,:,3)', 'uint8'), fclose(fid);   % write B

board = ccsboardinfo;      % Get DSP board and processor information
dsp = ccsdsp('boardnum',...% Link DSP with CCS
              board.number,...
              'procnum',...
              board.proc(1,1).number);
set(dsp,'timeout',100);    % Set CCS default timeout value to 100(s)
visible(dsp,1);            % Force CCS to be visible on PC desktop
open(dsp,' ccsLink\\ccsLink.pjt'); % Open project file
build(dsp,'all',1500);     % Build the project if necessary
load(dsp, ...              % Load project with timeout 300(s)
    '.\\ccsLink\\Debug\\ccsLink.out',300);
reset(dsp);                % Reset the DSP processor
restart(dsp);              % Restart the program
run(dsp);                  % Start execution
cpurunstatus = isrunning(dsp);
while cpurunstatus == 1,   % Wait until DSP completes the task
    cpurunstatus = isrunning(dsp);
end

                           % Read YCbCr data
fid = fopen('.\\ccsLink\\data\\Y8.YUV','r');
Y = fread(fid); fclose(fid);
fid = fopen('.\\ccsLink\\data\\Cb8.YUV','r');
Cb = fread(fid); fclose(fid);
fid = fopen('.\\ccsLink\\data\\Cr8.YUV','r');
Cr = fread(fid); fclose(fid);
                           % Form the YCbCr color space
YCbCr = cat(3, reshape(Y/255, width, height)', ...
               reshape(Cb/255, width, height)', ...
               reshape(Cr/255, width, height)');
RGB1 = ycbcr2rgb(YCbCr);   % Convert to RGB color space
RGB1 = uint8(RGB1*255);    % Convert double to uint8
figure; imshow(RGB1);      % Show result as RGB image
```

along with the Link for CCS functions, the development, debug, as well as the test procedure can be simplified.

The middle portion of the script listed in Table 15.5 controls the execution of DSP experiment. The MATLAB function `ccsboardinfo` obtains the DSP development system's information. The MATLAB function `ccsdsp` creates the link object for the CCS using the information obtained from the

Table 15.6 File listing for experiment `exp15.10.2_ccsLink`

Files	Description
`ccsLink.m`	MATLAB script controls the experiment
`RGB2YCbCr.c`	RGB color space to YC_bC_r conversion function
`RGB2YCbCrTest.c`	Main program for testing experiment
`rgb2ycbcr.h`	C header file
`ccsLink.cmd`	DSP linker command file
`param.txt`	Parameter file
`ccsLink.pjt`	DSP project file
`loveStar160x120.bmp`	Image file
`image300x300.jpg`	Image file
`color960x720.jpg`	Image file

function call to `ccsboardinfo`. In this experiment, the function `isrunning` continuously monitors the processing.

The software used for this experiment includes the DSP project, source files, data files, and MATLAB script files as summarized in Table 15.6.

The MATLAB script automatically starts the experiment if the DSP program does not have any error. This experiment reads the given images from the `data` directory and writes these images to the RGB color space in R, G, and B data files. Finally, MATLAB reads in the converted YC_bC_r data files and displays the conversion results.

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK or simulator to the computer.

2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.2_ccsLink` where the MATLAB script `ccsLink.m` is located.

3. Enter `ccsLink` from the MATLAB command window to start the script. The CCS will be started, DSK will be initialized, and the DSP project will be opened. Then MATLAB script will command the CCS to rebuild the project, load, and run the code for color space conversion. After the conversion, MATLAB reads and displays the resulting images.

4. MATLAB `imread()function` supports different image files. Replace the experiment image file, `loveStar.bmp`, with the `image300x300.jpg` (included in the CD) and rerun the experiment. Note that since the image sizes are different, the DSP parameter file, `param.txt`, needs to be modified accordingly.

5. Because the C5510 DSK has limited memory, the size of the image used by the experiment will be limited by the DSK memory. For large-size images, we need to adjust data block size. Modify the program such that the experiment can perform image color space conversion for the large-size image, `color960x720.jpg` (included in the data directory).

6. Image processing on a pixel-by-pixel basis requires very high computational power. The conversion function should be implemented in assembly language to take the advantages of C55x architectures such as parallel processing, dual-MAC units, and zero-overhead local-repeat loop features. Write an assembly routine to replace the RGB to YC_bC_r conversion function.

7. Profile the assembly language for the RGB to YC_bC_r conversion routine and compare its performance against the fixed-point C function.

Table 15.7 Color temperatures of different light sources

Color temperature (Kelvin)	Light sources
1000–2000 K	Candle light
2500–3500 K	Incandescent (household) lights
3000–4000 K	Sunrise or sunset
4000–5000 K	Fluorescent (office) lights
5000–5500 K	Electronic flash lights
5000–6500 K	Sunny daylight
6500–8000 K	Bright overcast sky

15.10.3 White Balance

Under different lighting sources, the object will show in different colors. Human vision can adapt these differences to view the proper colors. However, machine vision devices as well as digital cameras and camcorders cannot automatically adjust the colors. If the device is not correctly configured, the image color will differ under various light sources.

White balance is a process that corrects the unrealistic color resulted from different light sources. Traditional mechanical cameras use special optical filters for color corrections when taking pictures, while digital cameras use the auto-white balance and the manual-white balance. Auto-white balance uses the data from image sensor to compute the 'true' white color to balance the R, G, and B color components. Manual (or fixed) white balance uses predefined color balance setting on the particular lighting conditions, such as beach and snow scenes, outdoor daylight, and indoor candescent light. The light sources are usually described using color temperatures. Table 15.7 lists the common light sources and their temperatures.

Incorrect color balance will produce the bluish or reddish images. Example 15.4 in Section 15.5 shows an example of performing image auto-white balance. In this experiment, we will write a C55x program for auto-white balance. MATLAB Link for CCS will be used to control the experiment. Figure 15.19 shows the flow of the auto-white balance color correction.

Table 15.8 lists a section of the fixed-point C function that computes the sum of the pixels for white balance process, Table 15.9 shows a portion of the C program that calculates the white-balance gain, and Table 15.10 provides an example of the white balance correction on image pixels.

The experiment is controlled by MATLAB script `whiteBalance.m`. MATLAB Link for CCS allows MATLAB program to access global data variables using the `write( )` function. The experiment also uses MATLAB function `size( )` to find the width and height of the image file and pass these parameters to DSP object as follows:

```
[height, width, color] = size(RGB);
write(dsp,address(dsp,'imWidth'), uint16(width));
write(dsp,address(dsp,'imHeight'),uint16(height));
```

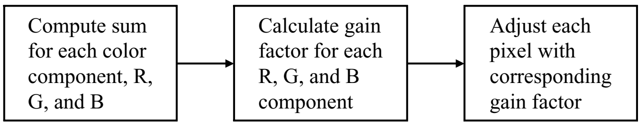

Compute sum for each color component, R, G, and B
Calculate gain factor for each R, G, and B component
Adjust each pixel with corresponding gain factor

Figure 15.19 Auto-white balance color correction

Table 15.8 Auto-white balance code example for computing the sum

```
AC0 = 0;
for (j=0; j<height; j++)
{
   for (i=0; i<width/2; i++)
   {
      AC0 += (*r>>8)&0xff;
      AC0 += (*r++)&0xff;
   }
}
```

Table 15.9 Auto-white balance code example for calculating gain

```
rGain = 255;
gGain = 255;
bGain = 255;
if ( (gSum <= rSum)&&(gSum <= bSum) )
{
    rGain = (unsigned short)((gSum*255) / rSum);
    bGain = (unsigned short)((gSum*255) / bSum);
}
```

In the above example, the image width and height are obtained from the RGB matrix. The width and height are then written to processor's global data variables `imWidth` and `imHeight` as 16-bit unsigned integers. This feature creates a flexible program that is no longer depending on the parameter file. This experiment also uses two packed pixels for each 16-bit data memory. The white balance program is written for handling dual-pixel data format. We use three pictures taken by a digital camera, which is set to fixed white balance at 4150 K for this experiment. These pictures are taken under incandescent light source (2850 K), fluorescent light source (4150 K), and daylight (6500 K). Before applying the white balance, the picture taken under candescent light looks reddish, while the picture taken under the bright daylight

Table 15.10 Auto-white balance code example for white balance function

```
for (j=0; j<height; j++)
{
   for (i=0; i<width/2; i++)
   {
      AC0 = (*r>>8)&0xff;
      AC0 *= rGain;
      if (AC0 > 0xff00)
         AC0 = 0xff00;
         AC0 &= 0xff00;
         AC1 = (*r)&0xff;
         AC1 *= rGain;
         AC1 >>= 8;
         if (AC1 > 0xff)
            AC1 = 0xff;
         *r++ = (unsigned short)(AC0| AC1);
   }
}
```

Table 15.11 File listing for experiment `exp15.10.3_whiteBalance`

Files	Description
`whiteBalance.m`	MATLAB script controls the experiment
`whitebalance.c`	Fixed-point C function for white balance
`whitebalanceTest.c`	Program for testing experiment
`whitebalanceGain.asm`	C55x assembly routine calculates white-balance gain
`whitebalanceSum.asm`	C55x assembly routine computes the sum
`whiteBalance.h`	C header file
`whiteBalance.cmd`	DSP linker command file
`whiteBalance.pjt`	DSP project file
`Tory_2850k.jpg`	Image file
`Tory_4150k.jpg`	Image file
`Tory_6500k.jpg`	Image file

appears bluish. The white balance function corrects the unrealistic color so they all look similar as they were taken under the fluorescent light source. The files used for this experiment are listed in Table 15.11.

The experiment has three stages: (1) compute the sums of R, G, and B pixels, (2) calculate the gain values for R, G, and B correction, and (3) perform white balance on R, G, and B pixels. These three stages are controlled by setting the `img->status` to 0, 1, and 2.

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK or simulator to the computer.

2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.3_white-Balance` where the MATLAB script `whiteBalance.m` is located.

3. Enter `whiteBalance` from MATLAB command window to start script for white balance experiment. When the experiment is completed, it displays both the original image and the white-balanced image for comparison.

4. Profile these stages using CCS profile function.

5. In the `src` directory, there are two C55x assembly files `whitebalanceSum.asm` and `whitebal-anceGain.asm`. Replace the C function `whitebalance.c` with these two assembly routines in the DSP project. Open the C header file `whiteBalance.h` from the DSP project and define the compiling-time condition to enable the use of assembly functions. This can be done by changing `//#define USE_ASM` to `#define USE_ASM`. After enabling the compiler condition for using assembly routines, rebuild the project.

6. Profile the experiment using fixed-point C functions and assembly functions. The profile result will show when the optimization option is set; the fixed-point C function requires 3.45 cycles per pixel to compute the sum, while it takes about 2.2 cycles if using the assembly routine. For a 320×240 image, the C function requires $320 \times 240 \times 3 \times 3.45 = 794\,880$ cycles to compute the sums of R, G, and B pixels; while the assembly routine only uses $320 \times 240 \times 3 \times 2.2 = 506\,880$ cycles. It clearly shows that the assembly routine can improve the efficiency by about 40 %.

7. Refer to the C function under the condition `img->status==1` to write an assembly routine that performs the white balance. Profile the performance improvement of the assembly routine over the C function.

8. Compare the performance improvements obtained by the assembly routines with the C functions that compute the sums and make white balance correction. It shows that the DSP code optimization should be concentrated on the portion of program that has nested loops and process data frequently. Profile the performance differences between C function that calculates the gain factors (`img->status==2`) and the assembly routine. This experiment shows the gain calculation using assembly routine reducing the run time from 610 cycles per call to 465 cycles. However, because this routine is called only once, it makes minor contributions to the overall run-time improvement. We can leave it in C code if it is possible. Identifying bottlenecks is an important task for real-time DSP programmers. We must know where the computational intensities are located and find ways to improve their efficiency. For some functions that are rarely called, we may be able to leave them in C functions. A careful analysis and proper trade-offs can yield a good balance among performance, code development complexity, and ease of maintenance.

15.10.4 Gamma Correction and Contrast Adjustment

Image gamma correction and contrast adjustment are often implemented using lookup tables. For an 8-bit image system, this method requires a 256-value table to map the image. Figure 15.20 shows the relations between the input image (solid line) and the table mapped output image (dotted line). Figure 15.20(b) maps each pixel according to the gamma curve $\gamma = 2.20$. Figure 15.20(c) maps input image pixels to a low-contrast image. Finally, Figure 15.20(d) maps the input image to produce an image with high contrast.

The tables used for table-lookup methods are usually generated off line or during the system initialization. In this experiment, we generate the gamma table, low contrast table, and high contrast table during initialization. If a dynamic table generation is required at run time, an efficient DSP program with fast

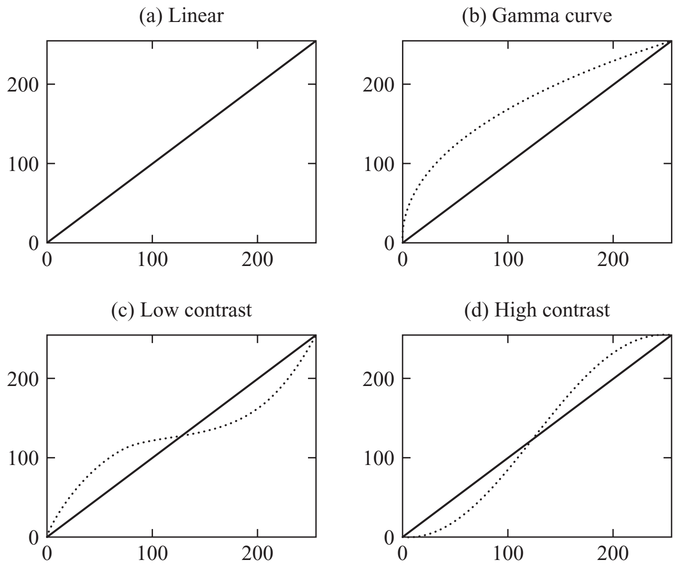


Figure 15.20 Table-lookup methods for different image mappings: (a) linear; (b) gamma; (c) low-contrast; and (d) high-contrast

Table 15.12 File listing for experiment `exp15.10.4_gammaContrast`

Files	Description
`gammaContrast.m`	MATLAB script controls the experiment
`tableGen.c`	Fixed-point C function generates gamma table
`imageMapping.c`	Fixed-point C function for gamma correction
`gammaContrastTest.c`	Program for testing experiment
`gammacontrast.h`	C header file
`gammaContrast.cmd`	DSP linker command file
`gammaContrast.pjt`	DSP project file
`boat.jpg`	Image file
`temple.jpg`	Image file

implementation will be needed. Some of the math functions may be realized using function approximation methods introduced in Chapter 3. As discussed in Section 15.5, gamma correction is a prewarping process to compensate the display devices' nonlinear output characteristics. Most personal computers use a gamma value of 2.20. The contrast adjustment is achieved by changing the image distribution as described by the contrast curves shown in Figure 15.20.

The experiment programs and MATLAB scripts are under the directory `exp15.10.4_ gammaContrast`. Two images are used for this experiment. One is for gamma correction and the other is for contrast adjustment. Table 15.12 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK or simulator to the computer.
2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.4_gammaContrast` where the MATLAB script `gammaContrast.m` is located.
3. Start the experiment using the MATLAB script and observe the resulting images. The gamma curve correction makes the image more bright for viewing. Applying low contrast adds more information in the middle range, but reduces the image dynamic range in bright and dark regions. High contrast can provide better details in the bright areas, but the image loses fine details in darken areas.
4. We can use MATLAB to prepare static data for DSP programs. We can remove the C file `tableGen.c` and use MATLAB to generate the gamma table, low contrast table, and high contrast table. Modify the script `gammaContrast.m` such that the MATLAB will generate these tables and use the MATLAB write function to initialize the gamma table and contrast table.
5. Some applications require both gamma correction and contrast adjustment. In order to improve the run-time efficiency, these two table-lookup implementations can be combined. Modify the experiment such that it will use a combined table that is generated with gamma $\gamma = 1.80$ and high contrast $a = -0.00035$.

15.10.5 Histogram and Histogram Equalization

In Example 15.6, we have shown the histogram equalization on luminance of the YC_bC_r color space. In this experiment, we will implement the histogram equalization using the C5510 DSK. The histogram equalization includes three important functions as shown in Figure 15.21.

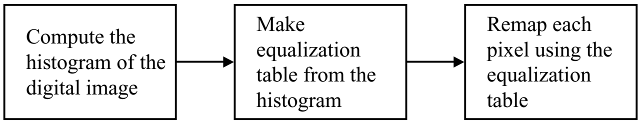

Compute the histogram of the digital image
Make equalization table from the histogram
Remap each pixel using the equalization table

Figure 15.21 Process flow of the histogram equalization

We write the computation of histogram and equalization functions using C55x assembly language. The assembly program takes advantages of zero-overhead local-block-repeat loop (`rptblocal`) instruction and bit-field extract (`bfxtr`) instruction to effectively improve the run-time speed. Tables 15.13 and 15.14 list the histogram and histogram equalization routines, respectively.

Table 15.13 Assembly implementation of histogram

```
_historgam:
    mov   dbl(*AR0+), XAR4     ; x = hist->x;
    mov   dbl(*AR0+), XAR2     ; hBuf = hist->histBuf;
    mov   *AR0+, T0            ; Get width
    mov   *AR0+, AR1           ; Get height
    sub   #1, AR1
    mov   AR1, BRC0
    sftl  T0, #-1
    sub   #1, T0
    mov   T0, BRC1
    rptblocal hightLoop-1      ; for (j=0; j<hist->height; j++){
    rptblocal widthLoop-1      ;    for(i=0; i<hist->width>>1; i++){
    mov   *AR4+, AC0           ;      pixel = *x++;
    bfxtr #0xff00, AC0, T0     ;   hBuf[(pixel>>8)&0xff] += 1;
    add   #1, *AR2(T0)
    and   #0x00ff, AC0, T0     ;   hBuf[pixel&0xff] += 1;
    add   #1, *AR2(T0)
widthLoop:                     ; }
    nop
hightLoop:                     ; }
    ret
```

Table 15.14 Assembly implementation of histogram equalization

```
_histEqualizer:
    mov   dbl(*AR0), XAR3   ; x = hist->x;
||  aadd  #4, AR0           ; Add 4 for large memory pointer offset
    mov   *AR0+, AR1        ; Get width
    sftl  AR1, #-1
||  mov   *AR0+, AR2        ; Get height
    sub   #1, AR1
    mov   AR1, BRC1
    sub   #1, AR2
    mov   AR2, BRC0
    mov   dbl(*AR0), XAR2   ; eqTbl = hist->eqTable;
||  rptblocal heightLoop-1  ; for (j=0; j<hist->height; j++){
```

Table 15.14 (*continued*)

```
    rptblocal widthLoop-1  ;    for(i=0; i<hist->width>>1; i++){
    mov   *AR3, AC0        ;    data = *x;
    bfxtr #0xff00, AC0, T0 ;    out1 = eqTbl[(data>>8)&0xff];
    and   #0x00ff, AC0, T0 ;    out2 = eqTbl[data&0xff];
||  mov   *AR2(T0), AC1
    mov   *AR2(T0), AC0
    or    AC1<<#8, AC0
    mov   AC0, *AR3+       ;    *x++ = (out1<<8)| out2;
widthLoop:                 ;    }
    nop
heightLoop:                ; }
    ret
```

This experiment is controlled using the MATLAB script `histogramEqualization.m`. Table 15.15 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK or simulator to the computer.
2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.5_histogramEQ` where the MATLAB script `histogramEqualization.m` is located.
3. Run the experiment using the MATLAB script and observe the resulting images. The equalized image has a larger dynamic range.
4. The histogram equalization is a useful tool for adjusting underexposed images especially for the B&W images. Try different underexposed and overexposed images and observe the histogram equalization results.

15.10.6 2-D Image Filtering

In this experiment, we introduce the basic image filtering using a 3×3 filter kernel. Image filtering requires extremely high computational power because of the large amount of pixels. For real-time applications,

Table 15.15 File listing for experiment `exp15.10.5_histogramEQ`

Files	Description
`histogramEqualization.m`	MATLAB script controls the experiment
`histEqTable.c`	Fixed-point function computes histogram EQ table
`histogramInit.c`	Fixed-point function for initialization
`histogramEqTest.c`	Program for testing experiment
`histogram.asm`	C55x assembly routine computes histogram
`histEqualizer.asm`	C55x assembly routine performs equalization
`histogramEQ.h`	C header file
`histogramEQ.cmd`	DSP linker command file
`histogramEQ.pjt`	DSP project file
`hallway.jpg`	Image file
`street.jpg`	Image file

Table 15.16 C code for 2-D filtering of image

```
for(x=0; x<imageWidth; x++)
{
  for(m=0; m<M; m++)
  {
    for(n=0; n<N; n++)
    {
      temp32 += (unsigned long)pixel[row++][x] * filter[n][m];
      if (row == 3)
      {
          row = 0;
      }
    }
  }
}
```

the image filtering must be implemented efficiently using assembly routines, or we should rely on specific hardware accelerator to meet the real-time constraints.

Fixed-point C implementation

This experiment uses the basic 3 × 3 kernel as the image filter to achieve different effects as shown in Example 15.7. The implementation of the 2-D filtering defined in Equation (15.13) is listed in Table 15.16.

In Table 15.16, the input image located at position `x` of `row` in the data array, `pixel[row][x]`, is filtered by an `n`-by-`m` 2-D filter `filter[n][m]`. The middle pixel corresponds to the current input. The input and its neighboring pixels are all contributing to the filtering results. To utilize the limited DSK memory, this experiment uses three rows of input image data for the 3 × 3 image filter kernel. These rows are arranged as shown in Figure 15.22.

The processing starts with three rows of data. The current input and output are represented by index n as shown in Figure 15.22(a). The row indexed with $n-1$ contains the previous row of image data and $n+1$ contains the next row. The filtering process continues from the first column to the last column for each row. The update of the data buffer is carried one row at a time. The update of the filter delay line (three rows) is actually achieved by adjusting the 2-D data array index, `row`. As shown in Figure 15.22(a), row n is the current center row when the filtering process starts. After the first update, the new current row is labeled with index $n+1$ (see Figure 15.22(b)). After another update, the current row is $n+2$ as shown in Figure 15.22(c). This pattern is repeated. The filtering process is depicted in Figure 15.12. It computes the dot product starting from columns then rows between the 3 × 3 data block and the 3 × 3 filter kernel.

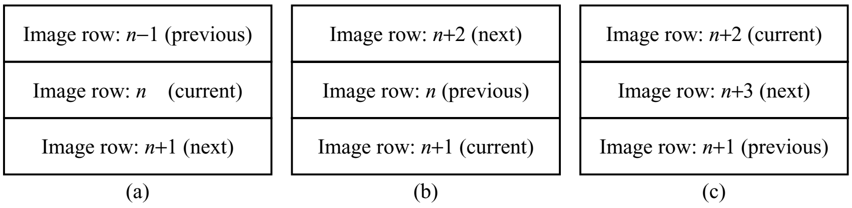


Image row: n–1 (previous)
Image row: n (current)
Image row: n+1 (next)
(a)
Image row: n+2 (next)
Image row: n (previous)
Image row: n+1 (current)
(b)
Image row: n+2 (current)
Image row: n+3 (next)
Image row: n+1 (previous)
(c)

Figure 15.22 Image data update scheme

Table 15.17 File listing for experiment `exp15.10.6_2DFilter`

Files	Description
`image2DFilter.m`	MATLAB script controls the experiment
`filter2D.c`	Fixed-point 2-D filter kernel function
`filter2D.asm`	C55x assembly 2-D filter kernel function
`filter2DTest.c`	Program for testing experiment
`filter2D.h`	C header file
`2DFilter.cmd`	DSP linker command file
`2DFilter.pjt`	DSP project file
`eagle.jpg`	Image file
`flower.jpg`	Image file

This experiment uses a 3×3 highpass filter kernel. The MATLAB script generates RGB data set from the given images, commands the DSP processor to perform the 2-D filtering, and finally, displays the results on the computer screen. Table 15.17 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK or simulator to the computer.

2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.6_2DFilter` where the MATLAB script `image2DFilter.m` is located.

3. Start the experiment using the MATLAB script and observe the resulting images. The filtered images have sharpened edges from the highpass filtering.

4. Use CCS to profile the DSP run-time requirements for filtering each pixel.

5. This experiment filters R, G, and B data in the RGB domain. Modify the MATLAB script such that the image data to be filtered is the luminance component Y in the YC_bC_r color space. Modify the experiment to filter the luminance only. Use MATLAB to reconstruct the filtered image and display it on computer.

6. Compare the RGB color space filtering results with the YC_bC_r color space. What are the DSP loading requirements for the RGB and YC_bC_r color spaces using the same image?

7. Since the 2-D filtering uses a 3×3 image filter kernel, the center element of the matrix corresponds to the current input and output. This means the boundary conditions are not considered by this experiment. The first row of filtered output is actually corresponding to the second row of the image data, and the last row is not processed correctly. The data in the first column and last column are also not processed. This problem can be solved by patching one row of data values on the top and the bottom of the image, one column to the left and the right of the image. The patch value can be 0 (black), 0xFF (white), or can use the neighboring pixels. Implement a patching scheme so the 2-D filtering will process all the pixels of any given image.

TMS320C55x assembly language implementation

The 2-D image filtering is usually written in assembly language because it requires very intense computation. This experiment presents assembly implementation of 2-D filtering listed in Table 15.18.

Table 15.18 Assembly implementation of 2-D filter kernel

```
.global _filter2D

_filter2D:
   psh T3, T2

; Initialization
;
   mov   *AR0+, T0                     ; mn = imfilt->m2D * imfilt->n2D
   mpym  *AR0+, T0, AC0
   sub   #1, AC0
   mov   mmap(AC0L), BRC1
   mov   *AR0+, T3                     ; row = imfilt->state
   mov   *AR0+, T0                     ; state
   neg   T0, T2
|| mov   dbl(*AR0+), XAR4              ; filter
   mov   dbl(*AR0+), XAR3              ; pixel = imfilt->inData
   mov   dbl(*AR0+), XAR1
   mov   *AR0, T1                      ; imfilt->imWidth
   mov   XAR1,XAR0
   sub   #1, T1, AR1
   mov   AR1, BRC0
;
; Process one row of data
;
   mov   #0, AC2
|| rptblocal row_loop-1                ; for(x=0; x<imfilt->imWidth; x++)
   amar  *AR4, XAR2                    ; filter = imfilt->filter
   mov   #0, AC0
|| rptblocal filter_loop-1             ; for(temp32=0, i=0; i<mn; i++)
   macm  *(T3), T1, AC2, AC1           ; temp32 +=
   mov   AC1, T0                       ; (long)pixel[(row*w)+x] * *filter++;
   mov   *AR3(T0) << #16, AC1
   macm  *AR2+, AC1, AC0
   add   #1, T3                        ; row++
   cmp   *(T3) == #3, TC1              ; if (row == 3)
   xcc   TC1                           ; row = 0
|| mov   #0, T3
filter_loop:
;
; Shift to compensate integer coefficient then limit to 8-bit
;
   sfts  AC0, T2, AC0                  ; temp32 >>= imfilt->shift
|| mov   #256, AC1
   cmp   AC0 >= AC1, TC1               ; if (temp32 > 255) temp32 = 255
   xcc   AC0 < #0                      ; if (temp32 < 0) temp32 = 0
   mov   #0, AC0
   xcc   TC1
   mov   #255, AC0
   mov   AC0, *AR0+                    ; *imfilt->outData++ = (char)(temp32)
   add   #1, AC2
row_loop:
   pop   T3,T2
   ret
```

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK or simulator to the computer.
2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.6_2DFilter` where the MATLAB script `image2DFilter.m` is located.
3. Replace the fixed-point C function `filter2D.c` with the C55x assembly routine `filter2D.asm`.
4. Run the experiment using the MATLAB script and observe the resulting images. The filtered images have sharpened edges from the highpass filter.
5. Use the FIR filtering techniques that we have learned in Chapter 4 to further improve the efficiency of assembly routine. Try to use dual-MAC or asymmetric filtering methods.

15.10.7 Implementation of DCT and IDCT

In this experiment, we implement the most commonly used 8×8 DCT on the C55x. We will show the fixed-point C functions and use assembly language routines for the forward DCT and IDCT. Assembly functions or hardware accelerators are often chosen for DCT and IDCT implementations.

Fixed-point C implementation

The 8×8 DCT and IDCT can be implemented using fixed-point C as shown in Table 15.19. The 64 pixels for DCT are arranged in a 1-D array pointed by the pointer `block`, and the DCT coefficients are stored in a 2-D array pointed by the pointer `dctCoef`. The implementation separates the 2-D DCT into two 1-D DCTs. First, the DCT is applied on each column of the 8×8 matrix. The results are stored in an intermediate 8×8 buffer `temp16[64]`. Then the DCT is applied to the rows of the matrix to obtain the final results. The final DCT results are stored back to the original data buffer pointed by the pointer `block`. This method is called in-place DCT because the results overwrite the original data when the transform is completed. Although the pixels and coefficients can be arranged in 2-D arrays, 1-D pointers are used in the DCT implementation. Using data pointers allows us to write more effective assembly code with autoincrement addressing mode, thus avoids extra computations to update the array pointers.

Table 15.19 C code of 8×8 DCT function

```
void dct8x8(short *block, short (*dctCoef)[8])
{
  long AC0;
  short i,j,h;
  short *tmpDat,*curData,*tmpCoef;
  short temp16[64];

  // Apply 1-D DCT on the columns of 8x8 data block
  for(i=0; i<8; i++)
  {
    curData = block + i;
```

continues overleaf

Table 15.19 (*continued*)

```
    tmpCoef = &dctCoef[0][0];
    for(h=0; h<8; h++)
    {
      tmpDat = curData;
      AC0 = *tmpCoef++ * (long)*tmpDat;
      tmpDat += 8;
      for(j=0; j<7; j++)
      {
        AC0 += *tmpCoef++ * (long)(*tmpDat);
        tmpDat += 8;
      }
      temp16[i+(h<<3)] = (short)(AC0>>12);
    }
  }
  // Apply 1-D DCT on the resulting rows
  for(i=0; i<64; i+=8)
  {
    curData = temp16 + i;
    tmpCoef = &dctCoef[0][0];
    for(h=0; h<8; h++)
    {
      tmpDat = curData;
      AC0 = *tmpCoef++ * (long)(*tmpDat++);
      for(j=0; j<7; j++)
      {
        AC0 += *tmpCoef++ * (long)(*tmpDat++);
      }
      block[i+h] = (short)(AC0>>15);
    }
  }
}
```

The DCT coefficients are generated according to Equation (15.20). In order to reduce quantization effects and preserve enough resolution, the DCT coefficients are converted to 12-bit integers. Table 15.20 shows the C function that generates 12-bit DCT coefficients. By examining the DCT and IDCT coefficients computed from Equations (15.20) and (15.21), respectively, we found that the IDCT coefficient array is the transpose of the DCT coefficient array.

Table 15.20 C function to generate DCT coefficients

```
#define PI          3.1415926
#define UNITS       (short)(4095*1.414*2)
void DCTcoefGen(short (*dctCoef)[8])
{
  short u,x;
  double cu;
  for (u=0;u<8;u++)
  {
    for(x=0;x<8;x++)
```

Table 15.20 (*continued*)

```
      {
        if (u==0)
        {
          cu = 0.70710678/2.0;
        }
        else
        {
          cu = 1.0/2.0;
        }
        dctCoef[u][x]=(short)(cu*cos(((2.0*x+1.0)*PI*u)/16.0)*UNITS+0.5);
      }
    }
  }
```

Due to the memory limitation of C5510 DSK, it is difficult to load entire image file for experiments. In this experiment, we read eight rows of image data each time for the 8×8 DCT processing. Once all eight rows of image data have been processed, the experiment will read next eight rows until it reaches the bottom eight rows. In this case, the width and height of digital image are constrained to the multiple of 8 pixels. The files used for this experiment are listed in Table 15.21.

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK or simulator to the computer.
2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.7_DCT` where the MATLAB script `DCT.m` is located.
3. Run the experiment using the MATLAB script and observe the resulting images.
4. Use CCS to profile the DSP run-time requirements of DCT and IDCT functions.
5. Due to the memory constraint of DSK, the test image width is 120 pixels. Modify the experiment such that it can accept any size of image files. Validate the changes by using a JPEG image of size greater than 640×480.

Table 15.21 File listing for experiment `exp15.10.7_DCT`

Files	Description
`DCT.m`	MATLAB script controls the experiment
`DCT.c`	Fixed-point DCT function
`IDCT.c`	Fixed-point IDCT function
`DCT.asm`	C55x assembly DCT function
`IDCT.asm`	C55x assembly IDCT function
`DCTTest.c`	Program for testing experiment
`DCT.h`	C header file
`DCT.cmd`	DSP linker command file
`DCT.pjt`	DSP project file
`totem.jpg`	Image file
`monument.jpg`	Image file

Assembly language implementation

The DCT and IDCT are computational intensive functions for image processing applications. Implementing these functions in assembly language is often necessary to meet strict real-time constraints. Table 15.22 lists the assembly routine for the IDCT function. Using local-repeat loop and index addressing mode, assembly routine reduces the processing time by about 37 %.

Table 15.22 Assembly routine of IDCT

```
_idct8x8:
  pshboth XAR5
  aadd  #-BLOCK_SIZE, SP      ; Adjust for temp16[64] buffer
;
; Apply 1-D IDCT on columns of 8x8 data block
;
   mov  #7, BRC0
   mov  #7, BRC1
   mov  #8, T0
||  amar *AR0, XAR5
   mov  #0, T1
||  rptblocal columnLoop-1     ; for(i=0; i<8; i++)
   mov  XSP, XAR4
   amar *AR1, XAR2            ; tmpCoef = &idctCoef[0][0]
   aadd T1, AR4
||  rptblocal dataLoop1-1      ; for(h=0; h<8; h++)
   amar *AR5, XAR3                       ; tmpDat = curData
   mpym *(AR3+T0), *AR2+, AC0 ; AC0=*tmpCoef++*(long)(*tmpDat)
                              ; tmpDat += 8
||  rpt #6                     ; for(j=0; j<7; j++)
   macm *(AR3+T0), *AR2+, AC0 ; AC0 += *tmpCoef++ * (long)(*tmpDat)
                              ; tmpDat += 8
   sfts AC0, #-12             ; temp16[i+(h<<3)] = (short)(AC0>>12)
   mov AC0, *(AR4+T0)
dataLoop1
   add  #1, T1
||  amar *AR5+
columnLoop
;
; Apply 1-D IDCT on resulting rows
;
   mov  #7, BRC0
   mov  #7, BRC1
   mov  XSP, XAR5
   rptblocal rowLoop-1    ; for(i=0; i<64; i+=8)
   amar *AR0, XAR4
   amar *AR1, XAR2        ; tmpCoef = &idctCoef[0][0]
||  rptblocal dataLoop2-1  ; for(h=0; h<8; h++)
   amar *AR5, XAR3        ; tmpDat = curData
   mpym *AR3+, *AR2+, AC0 ; AC0 = *tmpCoef++ * (long)(*tmpDat++)
||  rpt  #6                ; for(j=0; j<7; j++)
   macm *AR3+, *AR2+, AC0 ; AC0 += *tmpCoef++ * (long)(*tmpDat++)
   sfts AC0, #-15         ; block[i+h] = (short)(AC0>>15)
   mov  AC0, *AR4+
```

Table 15.22 (*continued*)

```
dataLoop2
   amar *(AR0+T0)
   amar *(AR5+T0)
rowLoop
   aadd #BLOCK_SIZE, SP
   popboth XAR5
   ret
```

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK or simulator to the computer.
2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.7_DCT` where the MATLAB script `DCT.m` is located.
3. Replace the fixed-point C functions `DCT.c` and `IDCT.c` with the C55x assembly routines `DCT.asm` and `IDCT.asm`.
4. Run the experiment using the MATLAB script and observe the resulting images. Compare the run-time improvement with the fixed-point C functions.
5. The DCT and IDCT routines can be further optimized using parallel processing features of the C55x processors and taking advantages of double memory read and store instructions. Modify the given assembly routines such that they are processing two pixels in parallel in each iteration.

15.10.8 TMS320C55x Image Accelerator for DCT and IDCT

The TMS320C5510 processors contain hardware accelerators to boost the performance of image and video processing algorithms. These hardware accelerators (extensions) treated as coprocessors are suitable for image and video applications such as JPEG and MPEG algorithms. The coprocessor achieves high performance using its built-in highly parallel hardware extensions. User can issue special commands to control the coprocessors to achieve the predetermined functions. The coprocessor instructions can be categorized into three types as summarized in Table 15.23.

The 8-bit constant `k8` in the coprocessor instructions represents the instruction code. The input values are represented using `Xmem` and `Ymem`. ACx, ACy, and ACz are the accumulators that contain data values and intermediate results. To use the hardware accelerator, the proper instruction sequences must be executed according to Texas Instruments' IMGLIB (image/video processing library) requirements. For

Table 15.23 TMS320C55x coprocessor instructions

Syntax	Description
`copr #k8, ACx, Xmem, Ymem, Acy`	Load + computation + transfer to accumulators

`copr #k8, ACx, ACy, ACy`
 `|| mov ACz, dbl(Lmem)`
	Computation + transfer to accumulators + write to memory

`corp #k8, ACx, Acy`	Special instructions

Table 15.24 DCT hardware accelerator instruction code

Sequence	Column code	Row code	Example
1	36	36	`copr #36,AC0,*(AR2+T0),*(AR1+T0),AC0`
2	32	32	`copr #32,AC0,AC1`
			|| mov AC0, dbl(*AR3+)
3	33	33	`copr #33,AC1,AC0`
			|| mov AC1, dbl(*AR3+)
4	51	51	`copr #51,AC0,AC1`
			|| mov AC0, dbl(*AR3+)
5	50	50	`copr #50,AC1,AC0`
			|| mov AC1, dbl(*AR3+)
6	38	38	`copr #38,AC0,*(AR2+T0),*(AR1+T0),AC1`
7	39	29	`copr #39,AC0,*(AR2+T0),*(AR1+T0),AC0`
8	37	34	`copr #37,AC0,*(AR2-T1),*(AR1-T1),AC1`

example, the 8×8 DCT and IDCT can be separated into 8-column transfers and 8-row transfers. Each transfer takes one cycle. The hardware accelerator instruction code for the DCT operation is given in Table 15.24, and the code for the IDCT operation is given in Table 15.25.

These instructions are defined for the TMS320C55x coprocessor to accomplish the DCT and IDCT algorithms. More details about the instruction codes for other video and image algorithms can be found in reference [6]. Table 15.26 lists the files used for this experiment. Because C55x hardware accelerator contains DCT and IDCT coefficients, we do not need to generate these coefficients.

Table 15.27 lists the performance of DCT and IDCT using the fixed-point C, assembly program, and hardware accelerator. The profile results show that the C55x hardware accelerator improves the performance of DCT and IDCT algorithms by a factor of 10 as compared with the fixed-point C implementation. The DCT and IDCT functions prototypes are

```
void hwdct8x8 (short *data, short *buffer);
void hwidct8x8 (short *data, short *buffer);
```

where `*data` is the pointer to the 8×8 image data block and `*buffer` is the intermediate working buffer of size 8×9. The DCT or IDCT results are written back to the data buffer pointed by `*data`.

Table 15.25 IDCT hardware accelerator instruction code

Sequence	Column code	Row code	Example
1	45	45	`copr #45,AC0,*(AR2+T0),*(AR1+T0),AC0`
2	47	47	`copr #47,AC0,AC1`
			|| mov AC0, dbl(*AR3+)
3	46	46	`copr #46,AC1,AC0`
			|| mov AC1, dbl(*AR3+)
4	58	58	`copr #58,AC0,AC1`
			|| mov AC0, dbl(*AR3+)
5	59	59	`copr #59,AC1,AC0`
			|| mov AC1, dbl(*AR3+)
6	41	41	`copr #41,AC0,*(AR2+T0),*(AR1+T0),AC1`
7	40	40	`copr #40,AC0,*(AR2+T0),*(AR1+T0),AC0`
8	44	42	`copr #44,AC0,*(AR2-T1),*(AR1-T1),AC1`

Table 15.26 File listing for experiment `exp15.10.8_HwAccelerator`

Files	Description
`hwAccelerator.m`	MATLAB script controls the experiment
`hwDCT.asm`	C55x assembly DCT using HW accelerator
`hwIDCT.asm`	C55x assembly IDCT using HW accelerator
`hwAcceleratorTest.c`	Program for testing experiment
`DCT.h`	C header file
`hwAccelerator.cmd`	DSP linker command file
`hwAccelerator.pjt`	DSP project file
`flower.jpg`	Image file
`statue.jpg`	Image file

Table 15.27 DCT and IDCT performance analysis

	Cycles per 8 × 8 block		Cycles per pixel	
	DCT	IDCT	DCT	IDCT
Fixed-point C	2422	2423	37.8	37.8
C55x assembly	1521	1524	23.8	23.8
Hardware accelerator	251	182	3.9	2.8

Both `data`[64] and `buffer`[72] arrays must be aligned to 32-bit boundary because the double-store instruction is used. To avoid memory access contention, the `data`[64] and `buffer`[72] arrays are placed into two separated memory blocks. We use the `pragma` keyword to control where and how the `data`[64] and `buffer`[72] arrays are placed.

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK to the computer and power on the DSK.
2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.8_HwAccelerator` where the MATLAB script `hwAccelerator.m` is located.
3. Run the experiment using the MATLAB script and observe the resulting images.
4. Examine the source code `hwDCT.asm` and refer to reference [6] for knowing how to use the coprocessor.

15.10.9 TMS320C55x Hardware Accelerator Image/Video Processing Library

The C55x hardware accelerator can perform several other video and image algorithms, including motion estimation, pixel interpolations, and computing absolute distances. These algorithms are essential for video and image applications such as the MPEG-4. To simplify the programming requirements for using these algorithms, Texas Instruments provides the IMGLIB library with C-callable interfaces to these video and image functions.

The IMGLIB can be downloaded from Texas Instruments' Web site. The version of the IMGLIB used in this experiment is Version 2.30. Table 15.28 lists the IMGLIB functions supported by the hardware

Table 15.28 List of IMGLIB functions using the hardware accelerator

Function	Description
`IMG_fdct_8x8`	2-D DCT for 8 × 8 image block
`IMG_idct_8x8`	2-D IDCT for 8 × 8 image block
`IMG_scale_by_2`	Image upscale by factor of 2
`IMG_mad_8x8`	8 × 8 minimum absolute difference
`IMG_mad_16x16`	16 × 16 minimum absolute difference
`IMG_sad_8x8`	Sum of absolute difference on single 8 × 8 block
`IMG_sad_16x16`	Sum of absolute difference on single 16 × 16 block
`IMG_pix_inter_16x16`	Pixel interpolation
`IMG_mad_16x16_4step`	Motion estimate by four-step search

accelerator for video and image applications. The complete list of the image library functions can be found in reference [7]. User can avoid directly programming coprocessor by using the C-callable IMGLIB functions. Many other video and image algorithms without using hardware accelerators are also included in the IMGLIB. In this experiment, we will use the hardware DCT and IDCT functions from the IMGLIB.

To use the IMGLIB functions, the C header files `imagelib.h` and `55ximage.lib` (or `55cimagex.lib` for large memory model) must be included in the DSP project. The DCT and IDCT functions syntaxes are

```
void IMG_fdct_8x8 (short *data, short *buffer);
void IMG_idct_8x8 (short *data, short *buffer);
```

where `*data` is the pointer to the 8 × 8 image data block and `*buffer` is the intermediate working buffer of size 8 × 9. The DCT or IDCT results are written back to the data buffer pointed by `*data`. Both `data`[64] and `buffer`[72] arrays must be aligned to 32-bit boundary in order to use double-store instruction. The `data`[64] and `buffer`[72] arrays are placed into two separated memory blocks to avoid possible memory access contention. Table 15.29 lists the files used for this experiment.

Procedures of the experiment are listed as follows:

1. Connect the C5510 DSK to the computer and power on the DSK.
2. Start MATLAB and set the MATLAB working directory to the directory `..\exp15.10.9_IMGLIB` where the MATLAB script `imglib.m` is located.
3. Add `55ximagex.lib` and `imagelib.h` to the project.

Table 15.29 File listing for experiment `exp15.10.9_HwAccelerator`

Files	Description
`imglib.m`	MATLAB script controls the experiment
`imgLibTest.c`	Program for testing experiment
`imglib.h`	C header file for the experiment
`imagelib.h`	C header file for IMGLIB functions
`55ximagex.lib`	IMGLIB for large memory model
`imglib.cmd`	DSP linker command file
`imglib.pjt`	DSP project file
`ship.jpg`	Image file
`smokingClock.jpg`	Image file

4. Run the experiment using the MATLAB script and observe the resulting images.

5. Using the TMS320C5510 IMGLIB histogram function, `IMG_histogram( )`, implement the histogram equalization experiment.

6. Using the 3 × 3 convolution function `IMG_conv_3x3( )` in the TMS320C5510 IMGLIB, implement the image filtering experiment. Repeat the filtering experiment in Section 15.10.6 and compare the filtering results.

References

[1] T. Sakamoto, C. Nakanishi, and T. Hase, 'Software pixel interpolation for digital still cameras suitable for a 32-bit MCU,' *IEEE Trans. Consum. Electron.*, vol. 44, pp. 1342–1352, Nov. 1998.

[2] ITU-R Recommendation BT.601–5, *Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios,* Oct. 1995.

[3] ITU CCIR Report 624-4, *Characteristics of Systems for Monochrome and Color Television*, June 2003.

[4] ITU-T Recommendation T.81, *Information Technology – Digital Compression and Coding of Continuous-Tone Still Images – Requirements and Guidelines*, 1992.

[5] F. A. McGovern, R. F. Woods, and M. Yan, 'Novel VLSI implementation of (8x8) point 2-D DCT,' *Electron. Lett.*, vol. 30, pp. 624–626, Apr. 1994.

[6] Texas Instruments, Inc., *TMS320C55x Hardware Extensions for Image/Video Applications Programmer's Reference*, Literature no. SPRU 098, Feb. 2002.

[7] Texas Instruments, Inc., *TMS320C55x Hardware Extensions for Image/Video Processing Library Programmer's Reference*, Literature no. SPRU 037B, Mar. 2003.

Exercises

1. Refer to Example 15.1 to write a MATLAB script that reads in R, G, and B data files resulting from the YC_bC_r to RGB conversion in Section 15.10.1, and display the RGB image using MATLAB function `imshow`. Use visual inspection to compare the RGB image displayed by MATLAB and the bitmap image generated by Section 15.10.1.

2. Implement the HSV to RGB color space conversion in fixed-point C. The data file `Sunset160x120.HSV` is provided in the companion CD.

3. Develop an experiment that converts the HSV color space to RGB color space using C55x assembly language.

4. Example 15.8 implements 2-D image filter in frequency domain. The resulting image has been shifted toward the bottom right. Modify the MATLAB script such that edge-affects will be minimized.

5. In Section 15.10.3, the white-balanced image for `Tory_2850k.jpg` is dimmed slightly. What causes this problem? Modify the white balance algorithm such that it will keep the image intensity unchanged after white balance.

6. Develop a new experiment that uses adaptive histogram equalization to divide the image into several regions and individually equalizes each smaller region instead of entire image.

7. Develop an experiment that uses a lowpass filter to remove the high-frequency noise in the image file `noiseImage.jpg`.

8. In experiments 15.10.1 and 15.10.2, we have introduced RGB and YC_bC_r conversion and in experiment 15.10.6, we have presented 2-D image filtering. Combine the RGB and YC_bC_r conversion with the 2-D image filter to develop a new experiment. For this new experiment, the image data is converted from RGB color space to YC_bC_r space, and then the 2-D image filter applies only to the luminance in the YC_bC_r color space. After the filtering, convert the YC_bC_r to RGB color space. Finally, use the RGB data to create a bitmap file for computer display.

9. McGovern *et al*. [5] have shown that the 8×8 DCT can be implemented with only 12 multiplications instead of 64 by taking advantage of the DCT and IDCT coefficients' redundancies. Implement the McGovern's algorithm and profile the improved DCT performance.

Appendix A
Some Useful Formulas and Definitions

This appendix briefly summarizes some basic formulas and definitions of algebra that will be used extensively in this book.

A.1 Trigonometric Identities

Trigonometric identities are often required in the manipulation of Fourier series, transforms, and harmonic analysis. Some of the most common identities are listed as follows:

$$\sin(-\alpha) = -\sin\alpha \tag{A.1a}$$

$$\cos(-\alpha) = \cos\alpha \tag{A.1b}$$

$$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta \tag{A.2a}$$

$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta \tag{A.2b}$$

$$2\sin\alpha\sin\beta = \cos(\alpha-\beta) - \cos(\alpha+\beta) \tag{A.3a}$$

$$2\cos\alpha\cos\beta = \cos(\alpha+\beta) + \cos(\alpha-\beta) \tag{A.3b}$$

$$2\sin\alpha\cos\beta = \sin(\alpha+\beta) + \sin(\alpha-\beta) \tag{A.3c}$$

$$\sin\alpha \pm \sin\beta = 2\sin\left(\frac{\alpha\pm\beta}{2}\right)\cos\left(\frac{\alpha\mp\beta}{2}\right) \tag{A.4a}$$

$$\cos\alpha + \cos\beta = 2\cos\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right) \tag{A.4b}$$

$$\cos\alpha - \cos\beta = -2\sin\left(\frac{\alpha+\beta}{2}\right)\sin\left(\frac{\alpha-\beta}{2}\right) \tag{A.4c}$$

$$\sin(2\alpha) = 2\sin\alpha\cos\alpha \tag{A.5a}$$

$$\cos(2\alpha) = 2\cos^2\alpha - 1 = 1 - 2\sin^2\alpha \tag{A.5b}$$

$$\sin\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1}{2}(1-\cos\alpha)} \tag{A.6a}$$

$$\cos\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1}{2}(1+\cos\alpha)} \tag{A.6b}$$

$$\sin^2\alpha + \cos^2\alpha = 1 \tag{A.7a}$$

$$\sin^2\alpha = \frac{1}{2}[1-\cos(2\alpha)] \tag{A.7b}$$

$$\cos^2\alpha = \frac{1}{2}[1+\cos(2\alpha)] \tag{A.7c}$$

$$e^{\pm j\alpha} = \cos\alpha \pm j\sin\alpha \tag{A.8a}$$

$$\sin\alpha = \frac{1}{2j}\left(e^{j\alpha} - e^{-j\alpha}\right) \tag{A.8b}$$

$$\cos\alpha = \frac{1}{2}\left(e^{j\alpha} + e^{-j\alpha}\right) \tag{A.8c}$$

In Euler's theorem given in Equation (A.8), $j = \sqrt{-1}$. The basic concepts and manipulations of complex number will be reviewed in Section A.3.

A.2 Geometric Series

The geometric series is used in discrete time signal analysis to evaluate functions in closed form. Its basic form is

$$\sum_{n=0}^{N-1} x^n = \frac{1-x^N}{1-x}, \qquad x \neq 1. \tag{A.9}$$

This is a widely used identity. For example,

$$\sum_{n=0}^{N-1} e^{-j\omega n} = \sum_{n=0}^{N-1} \left(e^{-j\omega}\right)^n = \frac{1-e^{-j\omega N}}{1-e^{-j\omega}}. \tag{A.10}$$

If the magnitude of x is less than 1 and not equal to zero, the infinite geometric series converges to

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}, \qquad 0 < |x| < 1. \tag{A.11}$$

A.3 Complex Variables

A complex number z can be expressed in rectangular (Cartesian) form as

$$z = x + jy = \mathrm{Re}[z] + j\mathrm{Im}[z]. \tag{A.12}$$

Since the complex number z represents the point (x, y) in the two-dimensional plane, it can be drawn as a vector illustrated in Figure A.1. The horizontal coordinate x is called the real part, and the vertical coordinate y is the imaginary part.

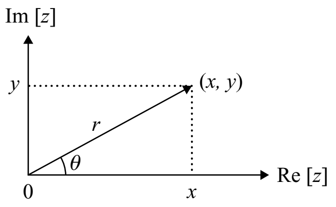


Figure A.1 Complex numbers represented as a vector

As shown in Figure A.1, the vector z can also be defined by its length (radius) r and its direction (angle) θ. The x and y coordinates of the vector are given by

$$x = r\cos\theta, \quad \text{and} \quad y = r\sin\theta. \tag{A.13}$$

Therefore, the vector z can be expressed in polar form as

$$z = r\cos\theta + jr\sin\theta = re^{j\theta}, \tag{A.14}$$

where

$$r = |z| = \sqrt{x^2 + y^2} \tag{A.15}$$

is the magnitude of the vector z and

$$\theta = \tan^{-1}\left(\frac{y}{x}\right) \tag{A.16}$$

is its phase in radians.

The basic arithmetic operations for two complex numbers $z_1 = x_1 + jy_1$ and $z_2 = x_2 + jy_2$ are listed as follows:

$$z_1 \pm z_2 = (x_1 \pm x_2) + j\,(y_1 \pm y_2) \tag{A.17}$$

$$z_1 z_2 = (x_1 x_2 - y_1 y_2) + j\,(x_1 y_2 + x_2 y_1) \tag{A.18a}$$

$$= (r_1 r_2)\, e^{j(\theta_1 + \theta_2)} \tag{A.18b}$$

$$\frac{z_1}{z_2} = \frac{(x_1 x_2 + y_1 y_2) + j\,(x_2 y_1 - x_1 y_2)}{x_2^2 + y_2^2} \tag{A.19a}$$

$$= \frac{r_1}{r_2} e^{j(\theta_1 - \theta_2)} \tag{A.19b}$$

Note that addition and subtraction are straightforward in rectangular form, but are difficult in polar form. Division is simple in polar form, but is complicated in rectangular form.

The complex arithmetic of the complex number x can be listed as

$$z^* = x - jy = re^{-j\theta}, \tag{A.20}$$

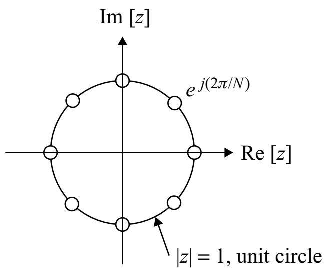


Figure A.2 Graphical display of the Nth roots of unity, $N = 8$

where * denotes complex-conjugate operation. In addition,

$$zz^* = |z|^2 \tag{A.21}$$

$$z^{-1} = \frac{1}{z} = \frac{1}{r}e^{-j\theta}, \tag{A.22}$$

$$z^N = r^N e^{jN\theta}. \tag{A.23}$$

The solution of

$$z^N = 1 \tag{A.24}$$

is

$$z_k = e^{j\theta_k} = e^{j(2\pi k/N)}, \qquad k = 0, 1, \ldots, N-1. \tag{A.25}$$

As illustrated in Figure A.2, these N solutions are equally spaced around the unit circle $|z| = 1$. The angular spacing between them is $\theta = 2\pi/N$.

A.4 Units of Power

Power and energy calculations are important in circuit analysis. Power is defined as the time rate of expending or absorbing energy, and can be expressed in the form of a derivative as

$$P = \frac{\mathrm{d}E}{\mathrm{d}t}, \tag{A.26}$$

where P is the power in watts, E is the energy in joules, and t is the time in seconds. The power associated with the voltage and current can be expressed as

$$P = vi = \frac{v^2}{R} = i^2 R, \tag{A.27}$$

where v is the voltage in volts, i is the current in amperes, and R is the resistance in ohms.

In engineering applications, the most popular description of signal strength is decibel (dB) defined as

$$N = 10\log_{10}\left(\frac{P_x}{P_y}\right) dB. \tag{A.28}$$

Therefore, the decibel unit is used to describe the ratio of two powers and requires a reference value, P_y for comparison.

It is important to note that both the current $i(t)$ and the voltage $v(t)$ can be considered as an analog signal $x(t)$, and thus the power of signal is proportional to the square of signal amplitude. For example, if the signal $x(t)$ is amplified by a factor g, that is, $x(t) = gy(t)$, the signal gain can be expressed in decibel as

$$\text{Gain} = 10\log_{10}\left(\frac{P_x}{P_y}\right) = 20\log_{10}(g), \tag{A.29}$$

since the power is a function of the square of the voltage (or current) as shown in Equation (A.27). As the second example, consider that the sound-pressure level, L_P, in decibels corresponds to a sound pressure P_x referenced to $P_y = 20\mu\,\text{Pa}$ (pascals). When the reference signal $y(t)$ has power P_y equal to 1 mW, the power unit of $x(t)$ is called dBm (dB with respect to 1 mW).

Digital reference level dBm0 is the digital milliwatt as defined in ITU-T Recommendation G.168. The method defined for measuring the input level of the signals is a root mean square (RMS) method. The dBm0 is measured as

$$P_k = 3.14 + 20\log\left[\frac{\sqrt{\frac{2}{N}\sum_{i=k}^{k-N+1} x_i^2}}{4096}\right] \text{ (A-law encoding)}, \tag{A.30a}$$

$$P_k = 3.17 + 20\log\left[\frac{\sqrt{\frac{2}{N}\sum_{i=k}^{k-N+1} x_i^2}}{8159}\right] \text{ (μ-law encoding)}, \tag{A.30b}$$

where P_k is signal level in dBm0, x_i is linear equivalent of the PCM encoded signal at time i, k is a discrete time index, and N is the number of samples over which the RMS measurement is made.

References

[1] J. J. Tuma, *Engineering Mathematics Handbook*, New York: McGraw-Hall, 1979.
[2] ITU-T Recommendation G.168, *Digital Network Echo Cancellers*, 2000.

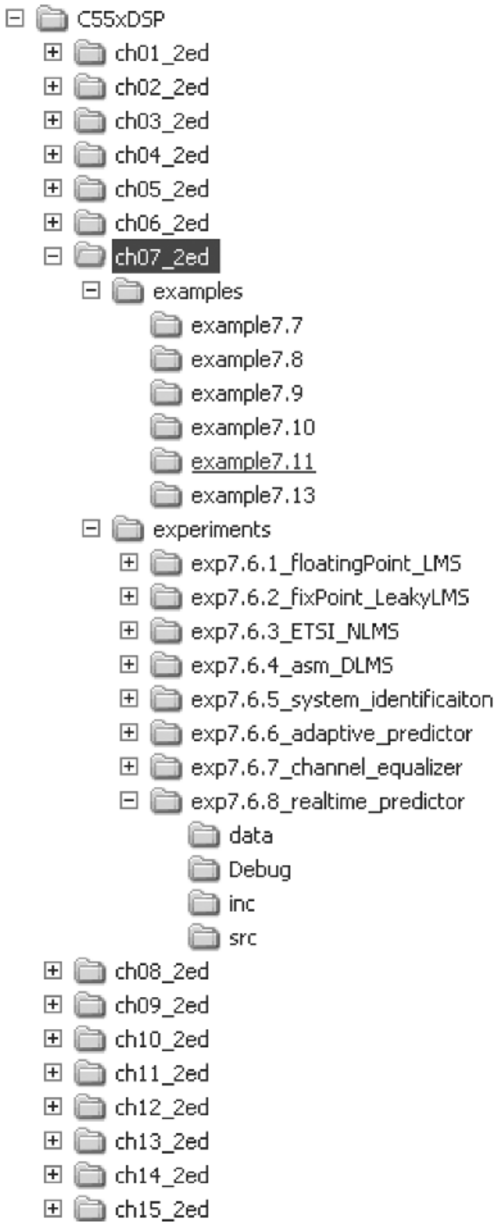
Appendix B
Software Organization and List of Experiments

The companion CD includes all the program and data files used for examples and experiments. Figure B.1 shows the directory structure of the software including examples, exercises, and experiments in the companion CD. The software is arranged by chapters where the software is referenced.

Each chapter contains two directories: `examples` and `experiments`. Some chapters include the `exercises` directory, which contains the necessary data files and software programs for the exercise problems at the end of that chapter. The `examples` directory consists of one or more subdirectories. Each subdirectory is named according to its example number. For example, the directory named by `example7.8` contains the MATLAB program `example7_8.m` that is used by Example 7.8 in Chapter 7.

The `experiments` directory contains all the experiments for that chapter. The name of the subdirectory under the experiments directory begins with the experiment number and is followed by the experiment name. For example, `exp7.6.8_realtime_predictor` as shown in Figure B.1 consists of all the program and data files for experiment given in Section 7.6.8. The `data` directory under the experiment subdirectory contains the data files used by the given experiment. The `Debug` directory is used by the CCS to store temporary files and the executable program. The `inc` directory is used for C and assembly include files (`.h` and `.inc`). The `src` directory has all the source programs including `.c` and `.asm` files. Table B.1 lists the file types and formats used by the book.

Table B.2 lists the experiments provided by the book. These experiments are primarily designed for using C5510 DSK; however, some experiments use MATLAB, Simulink, and C.



```
C55xDSP
    ch01_2ed
    ch02_2ed
    ch03_2ed
    ch04_2ed
    ch05_2ed
    ch06_2ed
    ch07_2ed
        examples
            example7.7
            example7.8
            example7.9
            example7.10
            example7.11
            example7.13
        experiments
            exp7.6.1_floatingPoint_LMS
            exp7.6.2_fixPoint_LeakyLMS
            exp7.6.3_ETSI_NLMS
            exp7.6.4_asm_DLMS
            exp7.6.5_system_identificaiton
            exp7.6.6_adaptive_predictor
            exp7.6.7_channel_equalizer
            exp7.6.8_realtime_predictor
                data
                Debug
                inc
                src
    ch08_2ed
    ch09_2ed
    ch10_2ed
    ch11_2ed
    ch12_2ed
    ch13_2ed
    ch14_2ed
    ch15_2ed
```

Figure B.1 Software directory structure

Table B.1 File types and formats used in the book

File extension	File type and format	Description
.asm	ASCII text	C55x assembly program source file
.bin	Binary	Data file
.bmp	Binary	Bitmap image file
.c	ASCII text	C program source file
.cdb	ASCII text	C55x CCS DSP/BIOS configuration file
.cmd	ASCII text	C55x linker command file
.dat	ASCII text	Data file or parameter file
.dsp	ASCII text	Microsoft Visual C IDE build file
.dsw	ASCII text	Microsoft Visual C IDE workspace file
.exe	Binary	Microsoft Visual C IDE executable file
.fig	Binary	MATLAB M-file
.h	ASCII text	C program header file
.HSV	Binary	HSV image file
.inc	ASCII text	C55x assembly program include file
.jpg	Binary	JPEG image file
.lib	Binary	C55x CCS run-time support library
.m	ASCII text	MATLAB script file
.map	ASCII text	C55x linker generated memory map file
.mdl	ASCII text	MATLAB Simulink script file
.mp3	Binary	MP3 audio file
.obj	Binary	C55x C compiler generated object file
.out	Binary	C55x linker generated executable file
.pcm	Binary	Linear PCM data file
.pjt	ASCII text	C55x DSP project file
.RGB	Binary	RGB image file
.txt	ASCII text	ASCII text file
.wks	Binary	C55x CCS IDE workspace file
.wav	Binary	Microsoft linear PCM wave file
.YUV	Binary	YUV or YC_bC_r image file

Table B.2 List of experiments for the book

Chapter	Experiment	Purpose	Platform
1	1.6.1	Familiar with CCS and DSK	CCS and C5510 DSK
	1.6.2	Learning CCS debugging tools	CCS and C5510 DSK
	1.6.3	Use CCS probe point	CCS and C5510 DSK
	1.6.4	Use CCS file IO	CCS and C5510 DSK
	1.6.5	Learn CCS profile	CCS and C5510 DSK
	1.6.6	Real-time loopback	CCS and C5510 DSK
	1.6.7	Sampling theory	MATLAB
	1.6.8	Understand ADC quantization	MATLAB
2	2.10.1	Use mixed C-and-assembly code	CCS and C5510 DSK
	2.10.2	C55x addressing mode	CCS and C5510 DSK
	2.10.3	Work with C55x DSP timer	CCS and C5510 DSK
	2.10.4	Use C55x EMIF and SDRAM	CCS and C5510 DSK
	2.10.5	Program DSK flash memory	CCS and C5510 DSK
	2.10.6	Build a McBSP library	CCS and C5510 DSK
	2.10.7	Set up AIC23	CCS and C5510 DSK
	2.10.8	Use C55x DMA	CCS and C5510 DSK

continues overleaf

Table B.2 (*continued*)

Chapter	Experiment	Purpose	Platform
3	3.6.1	Quantization of sinewave	CCS and C5510 DSK
	3.6.2	Quantization of audio signal	CCS and C5510 DSK
	3.6.3	Quantization of coefficients	CCS and C5510 DSK
	3.6.4	Manage overflow	CCS and C5510 DSK
	3.6.5	Function approximation	CCS and C5510 DSK
	3.6.6	Real-time signal generation	CCS and C5510 DSK
4	4.5.1	Fixed-point block FIR filter	CCS and C5510 DSK
	4.5.2	FIR in assembly function	CCS and C5510 DSK
	4.5.3	Symmetric block FIR filter	CCS and C5510 DSK
	4.5.4	Dual-MAC block FIR filter	CCS and C5510 DSK
	4.5.5	Decimation and decimator	CCS and C5510 DSK
	4.5.6	Interpolation and interpolator	CCS and C5510 DSK
	4.5.7	Sample rate converter	CCS and C5510 DSK
	4.5.8	Real-time SRC	CCS and C5510 DSK
5	5.7.1	Floating-point IIR filer	CCS and C5510 DSK
	5.7.2	Fixed-point IIR filter	CCS and C5510 DSK
	5.7.3	Cascade second-order IIR	CCS and C5510 DSK
	5.7.4	Using intrinsics	CCS and C5510 DSK
	5.7.5	Assembly implementation of IIR	CCS and C5510 DSK
	5.7.6	DSP/BIOS real-time application	CCS and C5510 DSK
	5.7.7	Parametric equalizer	CCS and C5510 DSK
	5.7.8	Real-time two-band equalizer	CCS and C5510 DSK
6	6.6.1	Floating-point DFT	CCS and C5510 DSK
	6.6.2	Assembly implementation of DFT	CCS and C5510 DSK
	6.6.3	Floating-point FFT	CCS and C5510 DSK
	6.6.4	Intrinsics implementation of FFT	CCS and C5510 DSK
	6.6.5	Assembly implementation of FFT	CCS and C5510 DSK
	6.6.6	Fast convolution	CCS and C5510 DSK
	6.6.7	Real-time FFT with DSP/BIOS	CCS and C5510 DSK
	6.6.8	Real-time FFT filter	CCS and C5510 DSK
7	7.6.1	Floating-point LMS	CCS and C5510 DSK
	7.6.2	Fixed-point leaky LMS	CCS and C5510 DSK
	7.6.3	ETSI normalized LMS	CCS and C5510 DSK
	7.6.4	Assembly implementation of DLMS	CCS and C5510 DSK
	7.6.5	System identification	CCS and C5510 DSK
	7.6.6	Adaptive predictor	CCS and C5510 DSK
	7.6.7	Channel equalizer	CCS and C5510 DSK
	7.6.8	Real-time adaptive predictor	CCS and C5510 DSK
8	8.4.1	Sinewave generator	CCS and C5510 DSK
	8.4.2	White Noise generator	CCS and C5510 DSK
	8.4.3	Siren generator	CCS and C5510 DSK
	8.4.4	DTMF generator	CCS and C5510 DSK
	8.4.5	MATLAB DTMF generation	MATLAB
9	9.4.1	Fixed-point DTMF detector	CCS and C5510 DSK
	9.4.2	Assembly implementation DTMF	CCS and C5510 DSK
	9.4.3	MATLAB Link for CCS	CCS and MATLAB
	9.4.4	LPC	MATLAB

Table B.2 (*continued*)

Chapter	Experiment	Purpose	Platform
10	10.7.1	MATLAB implementation of AEC	MATLAB
	10.7.2	Floating-point AEC	CCS and C5510 DSK
	10.7.3	Intrinsics implementation of AEC	CCS and C5510 DSK
	10.7.4	Delay detection	MATLAB
11	11.5.1	Floating-point implementation of LPC	CCS and C5510 DSK
	11.5.2	Intrinsics implementation of LPC	CCS and C5510 DSK
	11.5.3	MATLAB implementation of PWF	MATLAB
	11.5.4	Intrinsics implementation of PWF	CCS and C5510 DSK
12	12.7.1	Floating-point VAD	CCS and C5510 DSK
	12.7.2	MATLAB implementation of NR	MATLAB
	12.7.3	Floating-point noise reduction	CCS and C5510 DSK
	12.7.4	Mixed C-and-assembly VAD	CCS and C5510 DSK
	12.7.5	Floating-point AEC with NR	CCS and C5510 DSK
13	13.5.1	Floating-point MDCT	CCS and C5510 DSK
	13.5.2	Intrinsics implementation of MDCT	CCS and C5510 DSK
	13.5.3	Floating-point pre-echo	CCS and C5510 DSK
	13.5.4	ISO MP3 decoder	PC
14	14.4.1	Reed-Solomon code	MATLAB
	14.4.2	Simulink Reed-Solomon code	MATLAB/Simulink
	14.4.3	Verify Reed-Solomon root	CCS and C5510 DSK
	14.4.4	Viterbi decoding	MATLAB/Simulink
	14.4.5	Convolutional code	CCS and C5510 DSK
	14.4.6	CRC-32 implementation	CCS and C5510 DSK
15	15.10.1	YCbCr to RGB conversion	CCS and C5510 DSK
	15.10.2	RGB to YUV conversion	MATLAB/CCS/DSK
	15.10.3	White balance	MATLAB/CCS/DSK
	15.10.4	Gamma correction and contrast	MATLAB/CCS/DSK
	15.10.5	Histogram equalization	MATLAB/CCS/DSK
	15.10.6	Image 2-D filtering	MATLAB/CCS/DSK
	15.10.7	DCT implementation	MATLAB/CCS/DSK
	15.10.8	Use C55x HW accelerator	MATLAB/CCS/DSK
	15.10.9	Introduction to IMGLIB	MATLAB/CCS/DSK

Index