Priyabrata Sinha

Speech Processing in Embedded Systems

Springer

Speech Processing in Embedded Systems

Priyabrata Sinha

Speech Processing in Embedded Systems

Springer

Priyabrata Sinha
Microchip Technology, Inc.,
Chandler AZ,
USA
priyabrata.sinha@microchip.com

ISBN 978-0-387-75580-9 e-ISBN 978-0-387-75581-6
DOI 10.1007/978-0-387-75581-6
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2009933603

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Speech Processing has rapidly emerged as one of the most widespread and well-understood application areas in the broader discipline of Digital Signal Processing. Besides the telecommunications applications that have hitherto been the largest users of speech processing algorithms, several nontraditional embedded processor applications are enhancing their functionality and user interfaces by utilizing various aspects of speech processing. At the same time, embedded systems, especially those based on high-performance microcontrollers and digital signal processors, are rapidly becoming ubiquitous in everyday life. Communications equipment, consumer appliances, medical, military, security, and industrial control are some of the many segments that can potentially exploit speech processing algorithms to add more value to their users. With new embedded processor families providing powerful and flexible CPU and peripheral capabilities, the range of embedded applications that employ speech processing techniques is becoming wider than ever before.

While working as an Applications Engineer at Microchip Technology and helping customers incorporate speech processing functionality into mainstream embedded applications, I realized that there was an acute need for literature that addresses the embedded application and computational aspects of speech processing. This need is not effectively met by the existing speech processing texts, most of which are overwhelmingly mathematics intensive and only focus on theoretical concepts and derivations. Most speech processing books only discuss the building blocks of speech processing but do not provide much insight into what applications and end-systems can utilize these building blocks. I sincerely hope my book is a step in the right direction of providing the bridge between speech processing theory and its implementation in real-life applications.

Moreover, the bulk of existing speech processing books is primarily targeted toward audiences who have significant prior exposure to signal processing fundamentals. Increasingly, the system software and hardware developers who are involved in integrating speech processing algorithms in embedded end-applications are not DSP experts but general-purpose embedded system developers (often coming from the microcontroller world) who do not have a substantive theoretical background in DSP or much experience in developing complex speech processing algorithms. This large and growing base of engineers requires books and other sources of information that bring speech processing algorithms and concepts into the practical domain and also help them understand the CPU and peripheral needs for accomplishing such tasks. It is primarily this audience that this book is designed for, though I believe theoretical DSP engineers and researchers would also benefit by referring to this book as it would provide an real-world implementation-oriented perspective that would help fine-tune the design of future algorithms for practical implementability.

This book starts with Chap. 1 providing a general overview of the historical and emerging trends in embedded systems, the general signal chain used in speech processing applications, several applications of speech processing in our daily life, and a listing of some key speech processing tasks. Chapter 2 provides a detailed analysis of several key signal processing concepts, and Chap. 3 builds on this foundation by explaining many additional concepts and techniques that need to be understood by anyone implementing speech processing applications. Chapter 4 describes the various types of processor architectures that can be utilized by embedded speech processing applications, with special focus on those characteristic features that enable efficient and effective execution of signal processing algorithms. Chapter 5 provides readers with a description of some of the most important peripheral features that form an important criterion for the selection of a suitable processing platform for any application. Chapters 6–8 describe the operation and usage of a wide variety of Speech Compression algorithms, perhaps the most widely used class of speech processing operations in embedded systems. Chapter 9 describes techniques for Noise and Echo Cancellation, another important class of algorithms for several practical embedded applications. Chapter 10 provides an overview of Speech Recognition algorithms, while Chap. 11 explains Speech Synthesis. Finally, Chap. 12 concludes the book and tries to provide some pointers to future trends in embedded speech processing applications and related algorithms.

While writing this book I have been helped by several individuals in small but vital ways. First, this book would not have been possible without the constant encouragement and motivation provided by my wife Hoimonti and other members of our family. I would also like to thank my colleagues at Microchip Technology, including Sunil Fernandes, Jayanth Madapura, Veena Kudva, and others, for helping with some of the block diagrams and illustrations used in this book, and especially Sunil for lending me some of his books for reference. I sincerely hope that the effort that has gone into developing this book helps embedded hardware and software developers to provide the most optimal, high-quality, and cost-effective solutions for their end customers and to society at large.

Chandler, AZ Priyabrata Sinha

Contents

Chapter 1
Introduction

The ability to communicate with each other using spoken words is probably one of the most defining characteristics of human beings, one that distinguishes our species from the rest of the living world. Indeed, speech is considered by most people to be the most natural means of transferring thoughts, ideas, directions, and emotions from one person to another. While the written word, in the form of texts and letters, may have been the origin of modern civilization as we know it, talking and listening is a much more interactive medium of communication, as this allows two persons (or a person and a machine, as we will see in this book) to communicate with each other not only instantaneously but also simultaneously.

It is, therefore, not surprising that the recording, playback, and communication of human voice were the main objective of several early electrical systems. Microphones, loudspeakers, and telephones emerged out of this desire to capture and transmit information in the form of speech signals. Such primitive "speech processing" systems gradually evolved into more sophisticated electronic products that made extensive use of transistors, diodes, and other discrete components. The development of integrated circuits (ICs) that combined multiple discrete components together into individual silicon chips led to a tremendous growth of consumer electronic products and voice communications equipment. The size and reliability of these systems were enhanced to the point where homes and offices could widely use such equipment.

Digital vs. Analog Systems

Till recently, most electronic products handled speech signals (and other signals, such as images, video, and physical measurements) in the form of analog signals: continuously varying voltage levels representing the audio waveform. This is true even now in some areas of electronics, which is not surprising since all information in the physical world exists in an essentially analog form, e.g., sound waveforms and temperature variations. A large variety of low-cost electronic devices, signal conditioning circuits, and system design techniques exist for manipulating analog signals; indeed, even modern digital systems are incomplete without some analog components such as amplifiers, potentiometers, and voltage regulators.

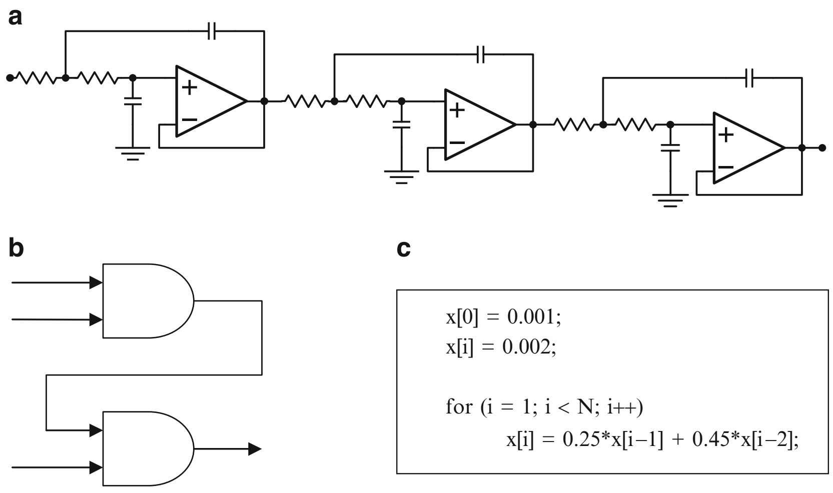
However, an all-analog electronic system has its own disadvantages:

- Analog signal processing systems require a lot of electronic circuitry, as all computations and manipulations of the signal have to be performed using a combination of analog ICs and discrete components. This naturally adds to system cost and size, especially in implementing rigorous and sophisticated functionality.
- Analog circuits are inherently prone to inaccuracy caused by component tolerances. Moreover, the characteristics of analog components tend to vary over time, both in the short term ("drift") and in the long term ("ageing").
- Analog signals are difficult to store for later review or processing. It may be possible to hold a voltage level for sometime using capacitors, but only while the circuit is powered. It is also possible to store longer-duration speech information in magnetic media like cassette tapes, but this usually precludes accessing the information in any order other than in time sequence.
- The very nature of an analog implementation, a hardware circuit, makes it very inflexible. Every possible function or operation requires a different circuit. Even a slight upgrade in the features provided by a product, e.g., a new model of a consumer product, necessitates redesigning the hardware, or at least changing a few discrete component values.

Digital signal processing, on the other hand, divides the dynamic range of any physical or calculated quantity into a finite set of discrete steps and represents the value of the signal at any given time as the binary representation of the step nearest to it. Thus, instead of an analog voltage level, the signal is stored or transferred as a binary number having a certain (system-dependent) number of bits. This helps digital implementations to overcome some of the drawbacks of analog systems [1]:

- The signal value can be encoded and multiplexed in creative ways to optimize the amount of circuit components, thereby reducing system cost and space usage.
- Since a digital circuit uses binary states (0 or 1) instead of absolute voltages, it is less affected by noise, as a slight difference in the signal level is usually not large enough for the signal to be interpreted as a 0 instead of a 1 or vice versa.
- Digital representations of signals are easier to store, e.g., in a CD player.
- Most importantly, substantial parts of digital logic can be incorporated into a microprocessor, in which most of the functionality can be controlled and adjusted using powerful and optimized software programs. This also lends itself to simple upgrades and improvements of product features via software upgrades, effectively eliminating the need to modify the hardware design on products already deployed in the field.

Figure 1.1 illustrates examples of an all-analog system and an all-digital system, respectively. The analog system shown here (an antialiasing filter) can be implemented using op-amps and discrete components such as resistors and capacitors (a). On the contrary, digital systems can be implemented either using digital hardware such as counters and logic gates (b) or using software running on a PC or embedded processor (c).



```
x[0] = 0.001;
x[i] = 0.002;

for (i = 1; i < N; i++)
        x[i] = 0.25*x[i−1] + 0.45*x[i−2];
```

Fig. 1.1 (**a**) Example of an analog system, with op-amps and discrete components. (**b**) Example of a digital system, implemented with hardware logic. (**c**) Example of a digital system, implemented only using software

Embedded Systems Overview

We have just seen that the utilization of computer programs running on a microprocessor to describe and control the way in which signals are processed provides a high degree of sophistication and flexibility to a digital system. The most traditional context in which microprocessors and software are used is in personal computers and other stand-alone computing systems. For example, a person's speech can be recorded and saved on the hard drive of a PC and played out through the computer speaker using a media player utility. However, this is a very limited and narrow method of using speech and other physical signals in our everyday life.

As microprocessors grew in their capabilities and speed of operation, system designers began to use them in settings besides traditional computing environments. However, microprocessors in their traditional form have some limitations when it comes to usage in day-to-day life. Since real-world signals such as speech are analog to begin with, some means must be available to convert these analog signals (typically converted from some other form of energy like sound to electrical energy using transducers) to digital values. On the output path, processed digital values must be converted back into analog form so that they can then be converted to other forms of energy. These transformations require special devices called Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC), respectively. There also needs to be some mechanism to maintain and keep track of timings and synchronize various operations and processes in the system, requiring peripheral devices called Timers. Most importantly, there need to be specialized programmable peripherals to communicate digital data and also to store data values for temporary and

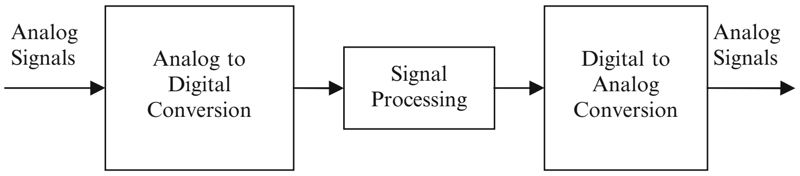


Fig. 1.2 Typical speech processing signal chain

permanent use. Ideally, all these peripheral functions should be incorporated within the processing device itself in order for the control logic to be compact and inexpensive (which is essential especially when used in consumer electronics). Figure 1.2 illustrates the overall speech processing signal chain in a typical digital system.

This kind of an integrated processor, with on-chip peripherals, memory, as well as mechanisms to process data transfer requests and event notifications (collectively known as "interrupts"), is referred to as Micro-Controller Units (MCU), reflecting their original intended use in industrial and other control equipment. Another category of integrated microprocessors, specially optimized for computationally intensive tasks such as speech and image processing, is called a Digital Signal Processor (DSP). In recent years, with an explosive increase in the variety of control-oriented applications using digital signal processing algorithms, a new breed of hybrid processors have emerged that combines the best features of an MCU and a DSP. This class of processors is referred to as a Digital Signal Controller (DSC) [7]. We shall explore the features of a DSP, MCU, and DSC in greater detail, especially in the context of speech processing applications, in Chaps. 4 and 5.

Finally, it may be noted that some general-purpose Microprocessors have also evolved into Embedded Microprocessors, with changes designed to make them more suitable for nontraditional applications.

Chapters 4 and 5 will describe the CPU and peripheral features in typical DSP/DSC architectures that enable the efficient implementation of Speech Processing operations.

Speech Processing in Everyday Life

The proliferation of embedded systems in consumer electronic products, industrial control equipment, automobiles, and telecommunication devices and networks has brought the previously narrow discipline of speech signal processing into everyday life. The availability of low-cost and versatile microprocessor architectures that can be integrated into speech processing systems has made it much easier to incorporate speech-oriented features even in applications not traditionally associated with speech or audio signals.

Perhaps the most conventional application area for speech processing is Telecommunications. Traditional wired telephone units and network equipment are now overwhelmingly digital systems, employing advanced signal processing techniques like speech compression and line echo cancellation. Accessories used with telephones, such as Caller ID systems, answering machines, and headsets are also major users of speech processing algorithms. Speakerphones, intercom systems, and medical emergency notification devices have their own sophisticated speech processing requirements to allow effective and clear two-way communications, and wireless devices like walkie-talkies and amateur radio systems need to address their communication bandwidth and noise issues. Mobile telephony has opened the floodgates to a wide variety of speech processing techniques to allow optimal use of bandwidth and employ value-added features like voice-activated dialing. Mobile hands-free kits are widely used in an automotive environment.

Industrial control and diagnostics is an emerging application segment for speech processing. Devices used to test and log data from industrial machinery, utility meters, network equipment, and building monitoring systems can employ voice-prompts and prerecorded audio messages to instruct the users of such tools as well as user-interface enhancements like voice commands. This is especially useful in environments wherein it is difficult to operate conventional user interfaces like keypads and touch screens. Some closely related applications are building security panels, audio explanations for museum exhibits, emergency evacuation alarms, and educational and linguistic tools. Automotive applications like hands-free kits, GPS devices, Bluetooth headsets/helmets, and traffic announcements are also fast emerging as adopters of speech processing.

With ever-increasing acceptance of speech signal processing algorithms and inexpensive hardware solutions to accomplish them, speech-based features and interfaces are finding their way into the home. Future consumer appliances will incorporate voice commands, speech recording and playback, and voice-based communication of commands between appliances. Usage instructions could be vocalized through synthesized speech generated from user manuals. Convergence of consumer appliances and voice communication systems will gradually lead to even greater integration of speech processing in devices as diverse as refrigerators and microwave ovens to cable set-top boxes and digital voice recorders.

Table 1.1 lists some common speech processing applications in some key market segments: Telecommunications, Automotive, Consumer/Medical, and Industrial/Military. This is by no means an exhaustive list; indeed, we will explore several speech processing applications in the chapters that follow. This list is merely intended to demonstrate the variety of roles speech processing plays in our daily life (either directly or indirectly).

Common Speech Processing Tasks

Figure 1.3 depicts some common categories of signal processing tasks that are widely required and utilized in Speech Processing applications, or even general-purpose embedded control applications that involve speech signals.

Table 1.1 Speech processing application examples in various market segments

Telecom	Automotive	Consumer/medical	Industrial/military
Intercom systems	Car mobile hands-free kits	Talking toys	Test equipment with spoken instructions
Speakerphones	Talking GPS units	Medical emergency phones	Satellite phones
Walkie-talkies	Voice recorders during car service	Appliances with spoken instructions	Radios
Voice-over-IP phones	Voice activated dialing	Recorders for physician's notes	Noise cancelling helmets
Analog telephone adapters	Voice instructions during car service	Appliances with voice record and playback	Public address systems
Mobile phones	Public announcement systems	Bluetooth headsets	Noise cancelling headsets
Telephones	Voice activated car controls	Dolls with customized voices	Security panels

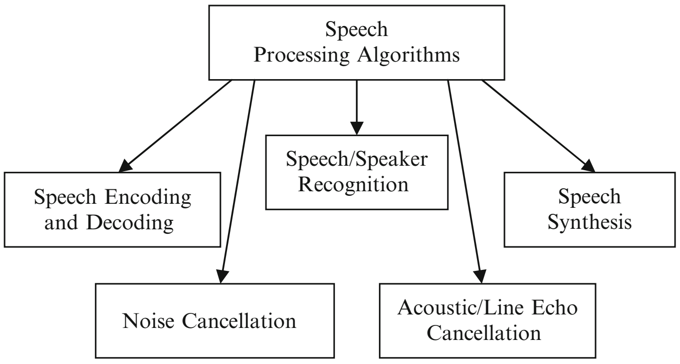

Speech Processing Algorithms
Speech Encoding and Decoding
Speech/Speaker Recognition
Speech Synthesis
Noise Cancellation
Acoustic/Line Echo Cancellation

Fig. 1.3 Popular signal processing tasks required in speech-based applications

Most of these tasks are fairly complex, and are detailed topics by themselves, with a substantial amount of research literature about them. Several embedded systems manufacturers (particularly DSP and DSC vendors) also provide software libraries and/or application notes to enable system hardware/software developers to easily incorporate these algorithms into their end-applications. Hence, it is often not critical for system developers to know the inner workings of these algorithms, and a knowledge of the corresponding Application Programming Interface (API) might suffice.

However, in order to make truly informed decisions about which specific speech processing algorithms are suitable for performing a certain task in the application, it is necessary to understand these techniques to some degree. Moreover, each of these speech processing tasks can be addressed by a tremendous variety of different algorithms, each with different sets of capabilities and configurations and providing different levels of speech quality. The system designer would need to understand the differences between the various available algorithms/techniques and select the most effective algorithm based on the application's requirements. Another significant factor that cannot be analyzed without some Speech Processing knowledge is the computational and peripheral requirements of the technique being considered.

Summary

For the above reasons, and also to facilitate a general understanding of Speech Processing concepts and techniques among embedded application developers for whom Speech Processing might (thought not necessarily) be somewhat unfamiliar terrain, several chapters of this book describe the different classes of Speech Processing operations illustrated in Fig. 1.3. Rather than delving too deep into the mathematical derivations and research evolutions of these algorithms, the focus of these chapters will be primarily on understanding the concepts behind these techniques, their usage in end-applications, as well as implementation considerations.

- Chapters 6–8 explain Speech Encoding and Decoding.
- Chapter 9 describes Noise and Echo Cancellation.
- Chapter 10 describes Speech Recognition.
- Chapter 11 describes Speech Synthesis.

References

1. Proakis JG, Manolakis DG Digital signal processing – principles, algorithms and applications, Prentice Hall, 1995.
2. Rabiner LR, Schafer RW Digital processing of speech signals, Prentice Hall, 1978.
3. Chau WC, Speech coding algorithms, Wiley-Interscience, 2003.
4. Spanias AS (1994) Speech coding: a tutorial review. Proc IEEE 82(10):1541–1582.
5. Hennessy JL, Patterson DA Computer architecture – a quantitative approach, Morgan Kaufmann, 2007.
6. Holmes J, Holmes W Speech synthesis and recognition, CRC Press, 2001.
7. Sinha P (2005) DSC is an SoC innovation. Electron Eng Times, July 2005, pages 51–52.
8. Sinha P (2007) Speech compression for embedded systems. In: Embedded systems conference, Boston, October 2007.

Chapter 2
Signal Processing Fundamentals

Abstract The first stepping stone to understanding the concepts and applications of Speech Processing is to be familiar with the fundamental principles of digital signal processing. Since all real-world signals are essentially analog, these must be converted into a digital format suitable for computations on a microprocessor. Sampling the signal and quantizing it into suitable digital values are critical considerations in being able to represent the signal accurately. Processing the signal often involves evaluating the effect of a predesigned system, which is accomplished using mathematical operations such as convolution. It also requires understanding the similarity or other relationship between two signals, through operations like autocorrelation and cross-correlation. Often, the frequency content of the signal is the parameter of primary importance, and in many cases this frequency content is manipulated through signal filtering techniques. This chapter will explore many of these foundational signal processing techniques and considerations, as well as the algorithmic structures that enable such processing.

Signals and Systems [1]

Before we look at what signal processing involves, we need to really comprehend what we imply by the term "signal." To put it very generally, a signal is any time-varying physical quantity. In most cases, signals are real-world parameters such as temperature, pressure, sound, light, and electricity. In the context of electrical systems, the signal being processed or analyzed is usually not the physical quantity itself, but rather a time-varying electrical parameter such as voltage or current that simply represents that physical quantity. It follows, therefore, that some kind of "transducer" converts the heat, light, sound, or other form of energy into electrical energy. For example, a microphone takes the varying air pressure exerted by sound waves and converts it into a time-varying voltage. Consider another example: a thermocouple generates a voltage that is roughly proportional to the temperature at the junction between two dissimilar metals. Often, the signal varies not only with time but also spatially; for example, the sound captured by a microphone is unlikely to be the same in all directions.

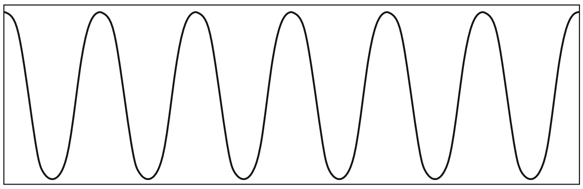


Fig. 2.1 A sinusoidal waveform – a classic example of an analog signal

At this point, I would like to point out the difference between two possible representations of a signal: analog and digital. An analog representation of a signal is where the exact value of the physical quantity (or its electrical equivalent) is utilized for further analysis and processing. For example, a single-tone sinusoidal sound wave would be represented as a sinusoidally varying electrical voltage (Fig. 2.1). The values would be continuous in terms of both its instantaneous level as well as the time instants in which it is measured. Thus, every possible voltage value (within a given range, of course) has its equivalent electrical representation. Moreover, this time-varying voltage can be measured at every possible instant of time. In other words, the signal measurement and representation system has infinite resolution both in terms of signal level and time. The raw voltage output from a microphone or a thermocouple, or indeed from most sensor elements, is essentially an analog signal; it should also be apparent that most real-world physical quantities are really analog signals to begin with!

So, how does this differ from a digital representation of the same signal? In digital format, snapshots of the original signal are measured and stored at regular intervals of time (but not continuously); thus, digital signals are always "discrete-time" signals. Consider an analog signal, like the sinusoidal example we discussed:

$$x_a(t) = A\sin(2\pi F t). \tag{2.1}$$

Now, let us assume that we take a snapshot of this signal at regular intervals of $T_\mathrm{s} = (1/F_\mathrm{s})$ seconds, where F_s is the rate at which snapshots of the signal are taken. Let us represent t/T_s as a discrete-time index n. The discrete-time representation of the above signal would be represented as:

$$x_a(nT_\mathrm{s}) = A\sin(2\pi F n T_\mathrm{s}). \tag{2.2}$$

Figure 2.2 illustrates how a "sampled" signal (in this example, a sampled sinusoidal wave) would look like.

Since the sampling interval is known a priori, we can simply represent the above discrete-time signal as an array, in terms of the sampling index n. Also, F/F_s can be denoted as the "normalized" frequency f, resulting in the simplified equation:

$$x[n] = A\sin(2\pi f n). \tag{2.3}$$

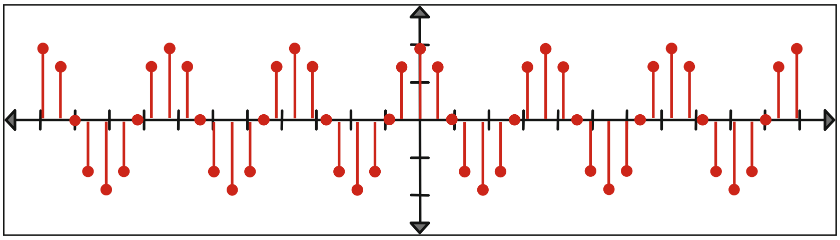


Fig. 2.2 The sampled equivalent of an analog sinusoidal waveform

However, to perform computations or analysis on a signal using a digital computer (or any other digital circuit for that matter), it is necessary but not sufficient to sample the signal at discrete intervals of time. A digital system of representation, by definition, represents and communicates data as a series of 0s and 1s: it breaks down any numerical quantity into its corresponding binary-number representation. The number of binary bits allocated to each number (i.e., each "analog" value) depends on various factors, including the data sizes supported by a particular digital circuit or microprocessor architecture or simply the way a particular software program may be using the data. Therefore, it is essential for the discrete-time analog samples to be converted to one of a finite set of possible discrete values as well. In other words, not only the sampling intervals but also the sampled signal values themselves must be discrete. From a processing standpoint, it follows that the signal needs to be both "sampled" (to make it discrete-time) and "quantized" (to make it discrete-valued); but more on that later.

The other fundamental concept in any signal processing task is the term "system." A system may be defined as anything (a physical object, electrical circuit, or mathematical operation) that affects the values or properties of the signal. For example, we might want to adjust the frequency components of the signal such that some frequency bands are emphasized more than others, or eliminate some frequencies completely. Alternatively, we might want to analyze the frequency spectrum or spatial signature of the signal. Depending on whether the signal is processed in the analog or digital domain, this system might be an analog signal processing system or a digital one.

Sampling and Quantization [1,2]

Since real-life signals are almost invariably in an analog form, it should be apparent that a digital signal processing system must include some means of converting the analog signal to a digital representation (through sampling and quantization), and vice versa, as shown in Fig. 2.3.

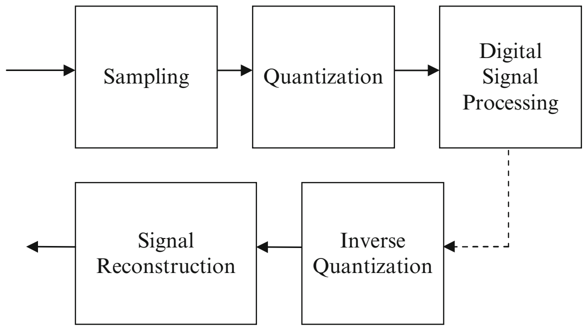


Fig. 2.3 Typical signal chain, including sampling and quantization of an analog signal

Sampling of an Analog Signal

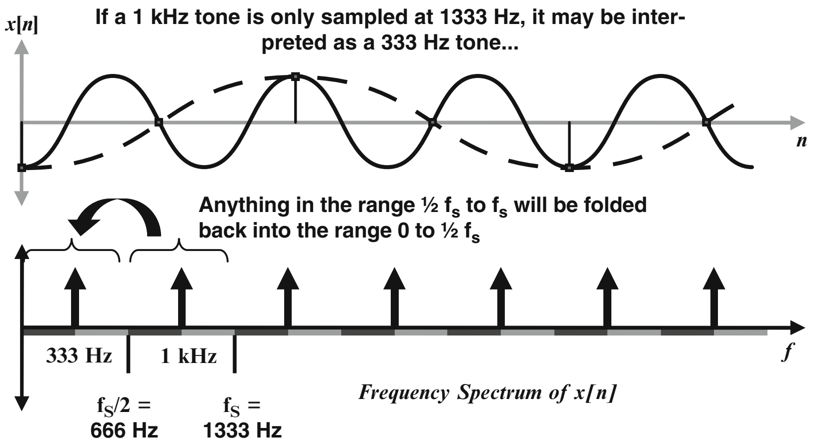
As discussed in the preceding section, the level of any analog signal must be captured, or "sampled," at a uniform Sampling Rate in order to convert it to a digital format. This operation is typically performed by an on-chip or off-chip Analog-to-Digital Converter (ADC) or a Speech/Audio Coder–Decoder (Codec) device. While we will investigate the architectural and peripheral aspects of analog-to-digital conversion in greater detail in Chap. 4, it is pertinent at this point to discuss some fundamental considerations in determining the sampling rate used by whichever sampling mechanism has been chosen by the system designer. For simplicity, we will assume that the sampling interval is invariant, i.e., that the sampling is uniform or periodic.

The periodic nature of the sampling process introduces the potential for injecting some spurious frequency components, or "artifacts," into the sampled version of the signal. This in turn makes it impossible to reconstruct the original signal from its samples. To avoid this problem, there are some restrictions imposed on the minimum rate at which the signal must be sampled.

Consider the following simplistic example: a 1-kHz sinusoidal signal sampled at a rate of 1.333 kHz. As can be seen from Fig. 2.4, due to the relatively low sampling rate, several transition points within the waveform are completely missed by the sampling process. If the sampled points are joined together in an effort to interpolate the intermediate missed samples, the resulting waveform looks very different from the original waveform. In fact, the signal now appears to have a single frequency component of 333 Hz! This effect is referred to as Aliasing, as the 1-kHz signal has introduced an unexpected lower-frequency component, or "alias."

The key to avoiding this phenomenon lies in sampling the analog signal at a high enough rate so that at least two (and preferably a lot more) samples within each period of the waveform are captured. Essentially, the chosen sampling rate must satisfy the Nyquist–Shannon Sampling Theorem.

The Nyquist–Shannon Sampling Theorem is a fundamental signal processing concept that imposes a constraint on the minimum rate at which an analog signal must be sampled for conversion into digital form, such that the original signal can



If a 1 kHz tone is only sampled at 1333 Hz, it may be interpreted as a 333 Hz tone...
$x[n]$
n
Anything in the range ½ f_s to f_s will be folded back into the range 0 to ½ f_s
333 Hz
1 kHz
f
$f_S/2$ = 666 Hz
f_S = 1333 Hz
Frequency Spectrum of $x[n]$

Fig. 2.4 Effect of aliasing caused by sampling less frequently than the Nyquist frequency

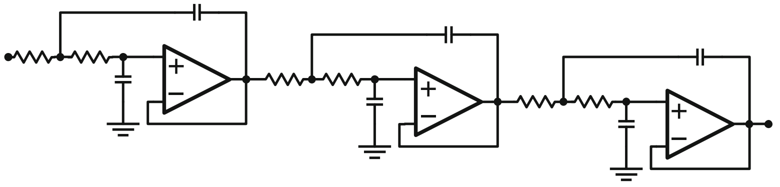


Fig. 2.5 A typical analog antialiasing filter, to be used before the signal is sampled

later be reconstructed perfectly. Essentially, it states that this sampling rate must be at least twice the maximum frequency component present in the signal; in other words, the sample rate must be twice the overall bandwidth of the original signal, in our case produced by a sensor.

The Nyquist–Shannon Theorem is a key requirement for effective signal processing in the digital domain. If it is not possible to increase the sampling rate significantly, an analog low-pass filter called Antialiasing Filter should be used to ensure that the signal bandwidth is less than half of the sampling frequency. It is important to note that this filtering must be performed before the signal is sampled, as an aliased signal is already irreversibly corrupted once the sample is sampled.

It follows, therefore, that Antialiasing Filters are essentially analog filters that restrict the maximum frequency component of the input signal to be less than half of the sampling rate. A common topology of an analog filter structure used for this purpose is a Sallen–Key Filter, as shown in Fig. 2.5. For speech signals used in telephony applications, for example, it is common to use antialiasing filters that have an upper cutoff frequency at around 3.4 kHz, since the sampling rate is usually 8 kHz.

One possible disadvantage of band-limiting the input signal using antialiasing filters is that there may be legitimate higher-frequency components in the signal that would get rejected as part of the antialiasing process. In such cases, whether to use an antialiasing filter or not is a key design decision for the system designer. In some applications, it may not be possible to sample at a high enough rate to completely avoid aliasing, due to the higher burden this places on the CPU and ADC; in yet other systems, it may be far more desirable to increase the sampling rate significantly (thereby "oversampling" the signal) than to expend hardware resources and physical space on implementing an analog filter. For speech processing applications, the choice of sampling rate is of particular importance, as certain sounds may be more pronounced at the higher range of the overall speech frequency spectrum; but more on that are discussed in Chap. 3.

In any case, several easy-to-use software tools exist to help system designers design antialiasing filters without being concerned with calculating the discrete components and operational amplifiers being used. For instance, the Filter Lab tool from Microchip Technology allows the user to simply enter filter parameters such as cutoff frequencies and attenuation, and the tool generate ready-to-use analog circuit that can be directly implemented in the application.

Quantization of a Sampled Signal

Quantization is the operation of assigning a sampled signal value to one of the many discrete steps within the signal's expected dynamic range. The signal is assumed to only have values corresponding to these steps, and any intermediate values are assigned to the step immediately below it or the step immediately above it. For example, if a signal can have a value between 0 and 5 V, and there are 250 discrete steps, then each step corresponds to a 20-mV range. In this case, 20 mV is denoted as the Quantization Step Size Δ. If a signal's sampled level is 1.005 V, then it is assigned a value of 1.00 V, while if it is 1.015 V it is assigned a value of 1.02 V. Thus, quantization is essentially akin to rounding off data, as shown in the simplistic 8-bit quantization example shown in Fig. 2.6.

The number of quantization steps and the size of each step are dependent on the capabilities of the specific analog-to-digital conversion mechanism being used. For example, if an ADC generates 12-bit conversion results, and if it can accept inputs up to 5 V, then the number of quantization steps $= 2^{12} = 4{,}096$ and the size of each quantization step is $(5/4{,}096) = 1.22\,\text{mV}$.

In general, if B is the data representation in binary bits:

$$\text{Number of Quantization Steps} = 2^B, \tag{2.4}$$

$$\text{Quantization Step Size} = (V_{\max} - V_{\min})/2^B. \tag{2.5}$$

The effect of quantization on the accuracy of the resultant digital data is generally quantified as the Signal-to-Quantization Noise Ratio (SQNR), which can be computed mathematically as:

$$\text{SQNR} = 1.5 \times 2^{(2B)}. \tag{2.6}$$

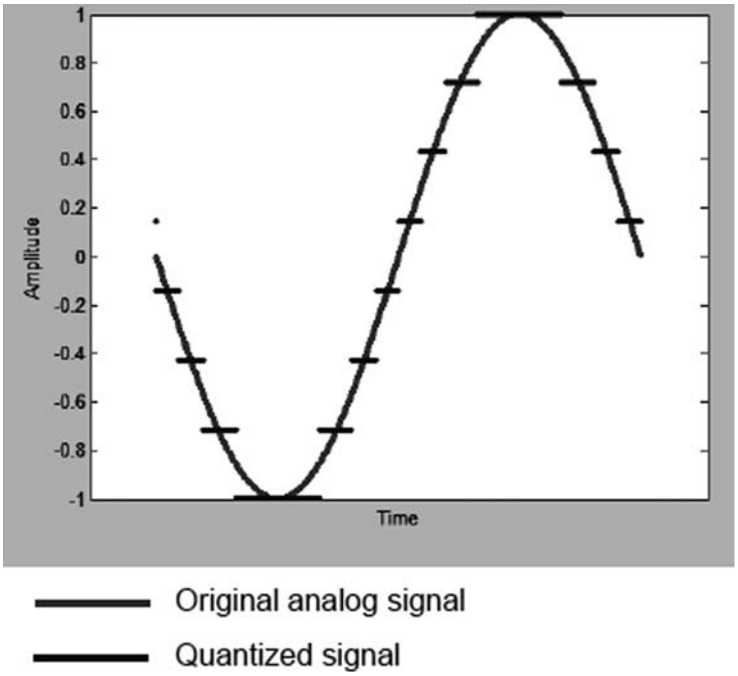


Fig. 2.6 Quantization steps for 8-bit quantization

On a logarithmic scale, this can be expressed as:

$$\text{SQNR (dB)} = 1.76 + 6.02B. \tag{2.7}$$

Thus, it can be seen that every additional bit of resolution added to the digital data results in a 6-dB improvement in the SQNR. In general, a high-resolution analog-to-digital conversion alleviates the adverse effect of quantization noise. However, other factors such as the cost of using a higher-resolution ADC (or a DSP with a higher-resolution ADC) as well as the accuracy and linearity specifications of the ADCs being considered. Most standard speech processing algorithms (and indeed, a large proportion of Digital Signal Processing tasks) operate on 16-bit data; so 16-bit quantization is generally considered more than sufficient for most embedded speech applications. In practice, 12-bit quantization would suffice in most applications provided the quantization process is accurate enough.

Convolution and Correlation [2,3]

Convolution and correlation are two extremely popular and fundamental common signal processing operations that are particularly relevant to speech processing, and therefore merit a brief description here. As we will see later, the convolution concept is a building block of digital filtering techniques too. In general, Convolution is a mathematical computation that measures the effect of a system on a signal, whereas Correlation measures the similarity between two signals.

The Convolution Operation

For now, let us limit our discussion on systems to a linear time-invariant (LTI) system, and let us also assume that both the signal and system are in digital form. A time-invariant system is one whose effect on a signal does not change with time. A linear system is one which satisfies the condition of linearity, which is that if S represents the effect of a system on a signal (i.e., system response), and $x_1[n]$ and $x_2[n]$ are two input signals, then:

$$S(a_1x_1[n] + a_2x_1[n]) = a_1S(x_1[n]) + a_2S(x_2[n]). \tag{2.8}$$

The most common method of describing a linear system is by its Impulse Response. The impulse response of a system is the series of output samples it would generate over time if a unit impulse signal (an instantaneous pulse of infinitesimally small duration which is zero for all subsequent sampling instants) were to be fed as its input. The concept of Impulse Response is illustrated in Fig. 2.7.

Let us denote the discrete-time input signal of interest as $x[n]$, the impulse response of the system as $h[n]$, and the output of the system as $y[n]$. Then the Convolution is mathematically computed as:

$$y[n] = h[n] \times x[n] = \sum_{k=-\infty}^{+\infty} h[k]x[n-k]. \tag{2.9}$$

Therefore, the convolution sum indicates how the system would behave with a particular signal. A common utility of the Convolution operation is to "smooth" speech signals, i.e., to alleviate the effect that block-processing discontinuities might have on the analysis of a signal.

As we will see in the following sections, this operation is also used to compute the output of a digital filter, one of the most common signal processing tasks in any

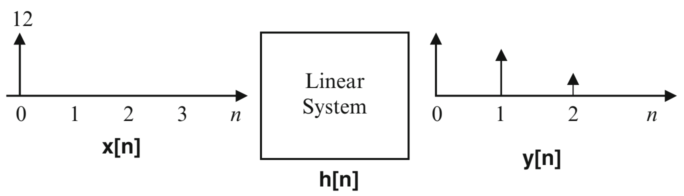


Fig. 2.7 Impulse response of a linear digital system

application. There is also a very interesting relationship between the convolution operation and its effect on frequency spectrum, which we will see when we explore the frequency transform of a signal.

Cross-correlation

The Correlation operation is very similar in its mathematical form to the Convolution. However, in this case the objective is to find the degree of similarity between two input signals relative to the time shift between them. If two different signals are considered the operation is called Cross-correlation, whereas if a signal is correlated with itself it is referred to as Autocorrelation.

Cross-correlation has several uses in various applications; for example, in a sonar or radar application one might find the correlation between the transmitted signal and the received signal for various time delays. In this case, the received signal is simply a delayed version of the transmitted signal, and one can estimate this delay (and thereby the distance of the object that caused the signal to be reflected) by computing the correlation for various time delays.

Mathematically, the cross-correlation between two signals is:

$$r_{xy}[k] = \sum_{n=-\infty}^{+\infty} x[n]y[n-k], \quad \text{where } k = 0, \pm 1, \pm 2, \text{etc.} \tag{2.10}$$

It should be apparent from the above equation that for $k = 0$, the value of the cross-correlation is maximum when two signals are similar and completely in phase with each other; it would get minimized when two signals are similar but completely out of phase. In general, if two signals are similar in their waveform characteristics, the location of their cross-correlation would indicate how much out of phase they are relative to each other. As a consequence, in some applications where a transmitted signal is reflected back after a certain delay and the application needs to estimate this delay, this can be accomplished by computing the cross-correlation of the transmitted and received (reflected) signals. For instance, this could be used as a simplistic form of echo estimation, albeit for single reflective paths.

Autocorrelation

For the special case of $y[n] = x[n]$, (2.10) reduces to the following operation, denoted as the Autocorrelation of the signal $x[n]$:

$$r_{xx}[k] = \sum_{n=-\infty}^{+\infty} x[n]x[n-k], \quad \text{where } k = 0, \pm 1, \pm 2, \text{etc.} \tag{2.11}$$

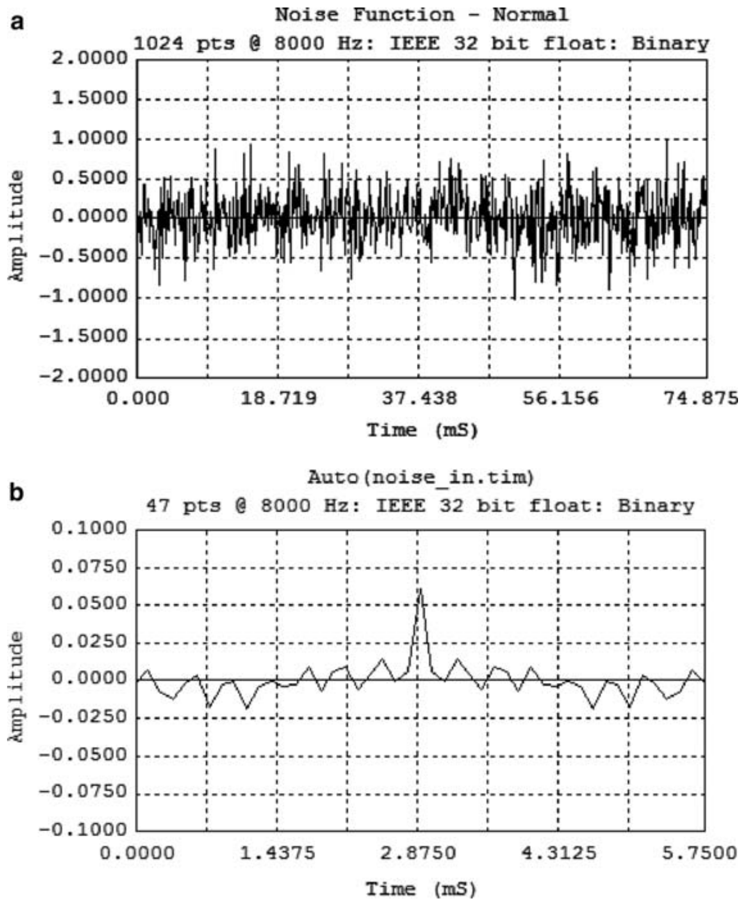
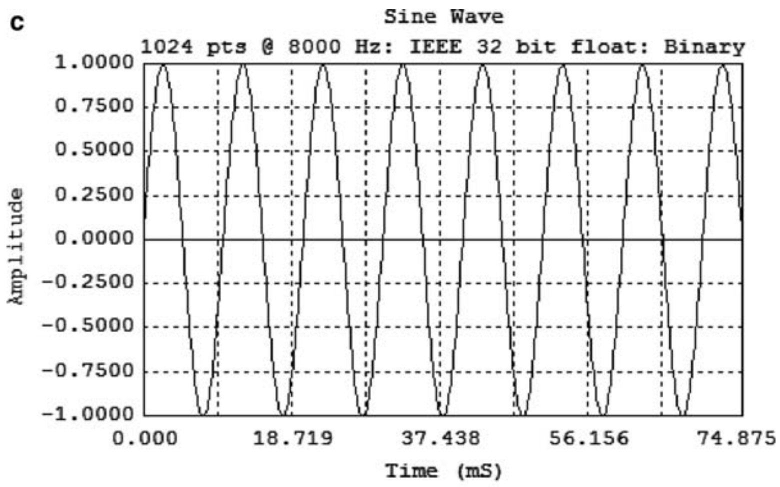


Fig. 2.8 (**a**) Random noise signal. (**b**) Autocorrelation of a random noise signal

Autocorrelation is used to find the similarity of a signal with delayed versions of itself. This is very useful, particularly in evaluating the periodicity of a signal: a perfectly periodic signal will have its highest autocorrelation value when k is the same as its period.

Let us consider an example of a random noise signal, as shown in Fig. 2.8a. Its autocorrelation, shown in Fig. 2.8b, is a sharp peak for a delay of zero and of negligible magnitude elsewhere. On the other hand, the autocorrelation of a periodic sinusoidal signal, shown in Fig. 2.8c, has a prominent periodic nature too, as depicted in Fig. 2.8d.

The above operation is particularly critical in voice coder applications wherein a speech signal might need to be classified into various categories based on its periodicity properties. As we will study in Chap. 3, different types of sounds generated by the human speech generation system have different types of frequency c



d

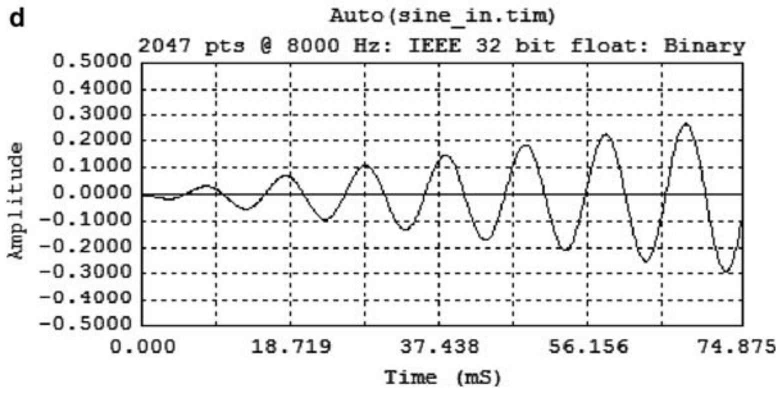


Fig. 2.8 (continued) (**c**) Sinusoidal signal with a frequency of 100 Hz. (**d**) Autocorrelation of a 100-Hz sinusoidal signal

characteristics: some are quasiperiodic whereas others are more like random noise. The Autocorrelation operation, when applied to speech signals, can clearly distinguish between these two types of sounds. Moreover, Autocorrelation can be used to extract desired periodic signals that may be buried in noise, e.g., a sonar wave buried in ambient acoustic noise.

The Autocorrelation operation also has an additional special property that has wide utility in several speech processing and other signal processing applications. When $k = 0$, the autocorrelation $r_{xx}[0]$ represents the energy of the signal. If the computation of Autocorrelation is performed over a short range of delays, and repeated for every block of speech signal samples, it can be used to estimate the energy present in that particular segment of speech. This is significant too, as we will learn in Chap. 3.

Frequency Transformations and FFT [1]

An overwhelming majority of signal processing algorithms and applications require some kind of analysis of the spectral content of a signal. In other words, one needs to find out the strength of each possible frequency component within the signal. There are several popular and not-so-popular methods of computing these frequency components (also known as "frequency bins," if you visualize the overall frequency spectrum of a signal to be comprised of smaller frequency ranges). However, the technique that is by far the most popular, at least as far as Speech Processing is concerned, is the Fourier Transform.

The fundamental principle behind the Fourier Transform is that any signal can be thought of as being composed of a certain number of distinct sinusoidal signals. This implies that one can accurately represent any signal as a linear combination of multiple sinusoidal signals. Therefore, the problem of finding the frequency components of a signal is reduced to finding the frequencies, amplitudes, and phases of the individual sinusoids that make up the overall signal of interest.

The Fourier Transform of a signal $x(t)$ is given by:

$$X(f) = \int_{t=-\infty}^{+\infty} x(t)\mathrm{e}^{-\mathrm{j}2\pi f t}\mathrm{d}t. \tag{2.12}$$

Discrete Fourier Transform

When the input signal is sampled at discrete intervals of time, as is the case in Digital Signal Processing algorithms, the above equation can be written in terms of discrete samples $x[n]$ and discrete frequency components $X[k]$. This formulation is known as the Discrete Fourier Transform (DFT), and is shown below.

$$X[k] = \sum_{n=0}^{N-1} x[n]\mathrm{e}^{-\mathrm{j}2\pi nk/N}, \tag{2.13}$$

where $k = 0,\ 1,\ 2,\ 3,\ \ldots,\ N$ are the various frequency bins.

Considering a sample rate of Fs, the resolution of the DFT can be given by

$$\text{Resolution} = \text{Fs}/N. \tag{2.14}$$

It is apparent from (2.14) that to achieve a finer analysis of the frequency bins of the signal, the sampling rate should be as less as possible (subject to satisfying the Nyquist–Shannon Theorem, of course) and the processing block size must be as large as possible. Having a large block size comes with a great computational cost in terms of processor speed and memory requirements, and must be carefully evaluated so as to achieve a golden balance between signal processing efficacy and computational efficiency.

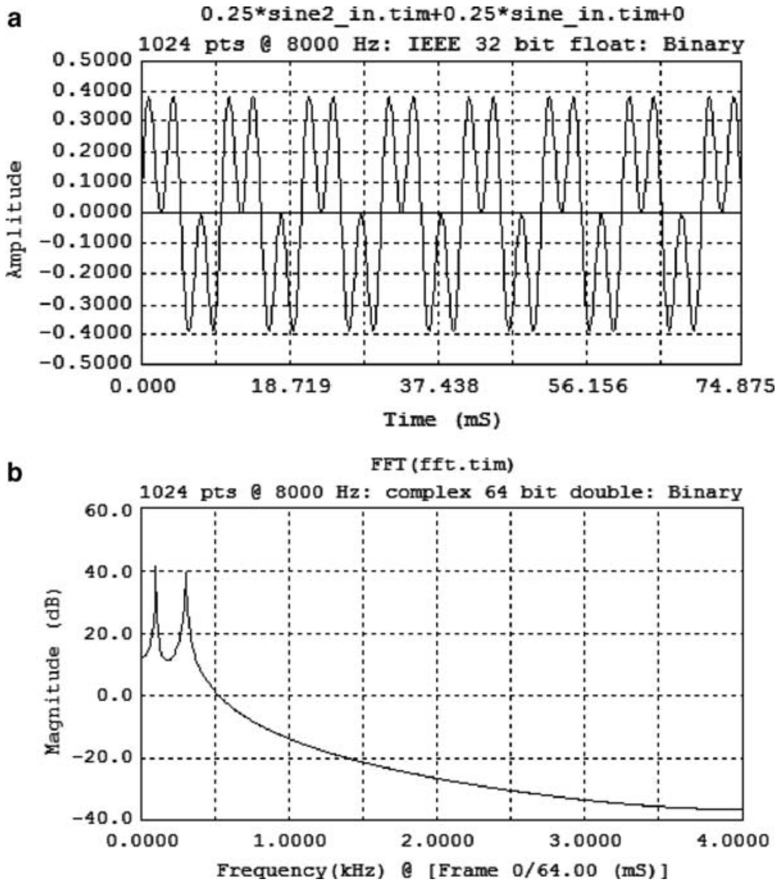


Fig. 2.9 (**a**) Signal created by a linear combination of a 100-Hz and a 300-Hz signal. (**b**) DFT frequency spectrum showing distinct 100- and 300-Hz peaks

The DFT completely defines the frequency content of a sampled signal (which is also known as the Frequency Domain representation of the Time Domain signal), as evidenced by the time-domain and frequency-domain plots shown in Fig. 2.9a, b.

Frequency Domain representations are used to “compress” the information of a speech waveform into information about strong frequency components, since not all frequency components are present or even relevant to the application. The frequency spectrum can also be utilized to distinguish between different types of speech sounds, something that has already been mentioned as an important task in some Speech Processing algorithms.

An interesting property of the DFT is that multiplication of two sets of DFT outputs is equivalent to performing the convolution of two blocks of Time Domain data. Thus, if you perform the Inverse DFT of the two sets of Frequency Domain data, and then multiply them point-by-point, the resultant data would be identical to the convolution of the two signals. This characteristic can be utilized to filter signals in the Frequency Domain rather than the Time Domain.

In general, it is not so much the DFT outputs themselves that are of interest but rather the magnitude and phase of each FFT output. The Squared-Magnitude (which is just as useful as the Magnitude and avoids the computation of a square-root) can be computed as follows:

$$|X[k]|^2 = \mathrm{Re}[X[k]]^2 + \mathrm{Im}\,[X[k]]^2 \tag{2.15}$$

and the phase of each FFT output is computed as:

$$\mathrm{Arg}[k] = \arctan\left(\frac{\mathrm{Im}[X[k]]}{\mathrm{Re}\,[X[k]]}\right). \tag{2.16}$$

Fast Fourier Transform

However, in order to use the DFT operation in a real-time embedded application, it needs to be computationally efficient. An inspection of (2.13) indicates that it requires a large number of mathematical operations, especially when the processing block size is increased. To be precise, it requires $N(N-1)$ complex multiplications and N^2 complex additions. Remember that a complex addition is actually two real additions, since you need to add the real and imaginary parts of the products. Similarly, each complex multiplication involves four real multiplication operations. Therefore, the order of complexity of the DFT algorithm (assuming all frequency bins are computed) is $O(N^2)$. This is obviously not very efficient, especially when it comes to large block sizes (256, 512, 1024, etc.) that are common in many signal processing applications.

Fortunately, a faster and more popular method exists to compute the DFT of a signal: it is called a Fast Fourier Transform (FFT). There are broadly two classes of FFT algorithms: Decimation in Time (DIT) and Decimation in Frequency (DIF). Both of these are based on a general algorithm optimization methodology called Divide-and-Conquer. I shall briefly discuss the core concepts behind the DIT methodology without delving into the mathematical intricacies. The DIF method is left for the reader to study from several good references on this topic, if needed.

The basic principles behind the Radix-2 DIT algorithm (the most popular variant of DIT there is) can be summarized as follows:

- Ensure that the block size N is a power-of-two (which is why it is called a Radix-2 FFT).
- The N-point data sequence is split into two $N/2$-point data sequences
  - Even data samples ($n = 0,\ 2,\ 4,\ \ldots,\ N/2$)
  - Odd data samples ($n = 1,\ 3,\ 5,\ \ldots,\ N/2-1$).

- Each such sequence is repeatedly split ("decimated") as shown above, finally obtaining $N/2$ data sequences of only two data samples each. Thus, a larger problem has effectively been broken down into the smallest problem size possible.
- These 2-point FFTs are first computed, an operation popularly known as a "butterfly."
- The outputs of these butterfly computations are progressively combined in successive stages, until the entire N-point FFT has been obtained. This happens when $\log_2(N)$ stages of computation have been executed.
- The above decimation methodology exploits the inherent periodicity of the $e^{-j2\pi nk/N}$ term.

A simplified conceptual diagram of an 8-point FFT is shown in Fig. 2.10.

The coefficients $e^{-j2\pi k/N}$ are the key multiplication factors in the Butterfly computations. These coefficients are called Twiddle Factors, and $N/2$ Twiddle Factors are required to be stored in memory in order to compute N-point Radix-2 FFT. Since there are only $N/2$ Twiddle Factors, and since there are $\log_2(N)$ stages, computing an FFT requires $(N/2)\log_2(N)$ complex multiplications and $(N/2)\log_2(N)$ complex additions. This leads to substantial savings in number of mathematical operations, as shown in Table 2.1.

A Radix-4 algorithm can result in a further reduction in number of operations required compared with Radix-2; however, the Radix-4 FFT algorithm requires the

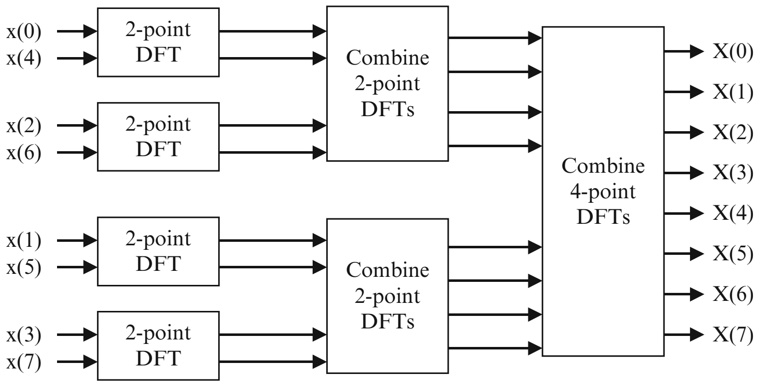


Fig. 2.10 FFT data flow example: 8-point Radix-2 DIT FFT

Table 2.1 Complex multiplications and additions needed by FFT and DFT

	DFT		FFT	
N	Multiplications	Additions	Multiplications	Additions
128	16,384	16,256	448	896
256	65,536	65,280	1,024	2,048
512	262,144	261,632	2,304	4,608

FFT block size by a power of 4, which is not always feasible to have in many embedded applications. As a result, Radix-2 is by far the most popular form of FFT. The DIF algorithm operates somewhat similar to the DIT algorithm, with one key difference: it is the output data sequence (i.e., the Frequency Domain data) that is decimated in the DIF rather than the input data sequence.

Benefits of Windowing

Since a DFT or FFT is computed over a finite number of data points, it is possible that the block sizes and signal frequencies are such that there are discontinuities between the end of one block and the beginning of the next, as illustrated in Fig. 2.11a. These discontinuities manifest themselves as undesirable artifacts in the frequency response, and these artifacts tend to quite spread-out over the spectrum due to their abrupt nature. This can be alleviated somewhat by tapering the edges of the block such that the discontinuities are minimized. This is accomplished very effectively using Window functions.

Window functions are basically mathematical functions of time that impart certain characteristics to the resulting frequency spectrum. They are applied to the input data before computing the FFT, by convolving the impulse response of the Window (Fig. 2.12) with the data samples. This is yet another useful application of the Convolution operation we had seen earlier in this chapter. In practice, application developers may not need to develop software to compute the impulse response of these windows, as processor and third-party DSP. Library vendors often provide ready-to-use functions to compute Window functions. Some of these involve some complicated mathematical operations, but the Window computation need only be performed once by the application and hence does not adversely affect real-time performance in any way.

Now that we have learnt how to analyze the frequency components of any Time Domain signal, let us explore some methods of manipulating the frequencies in the

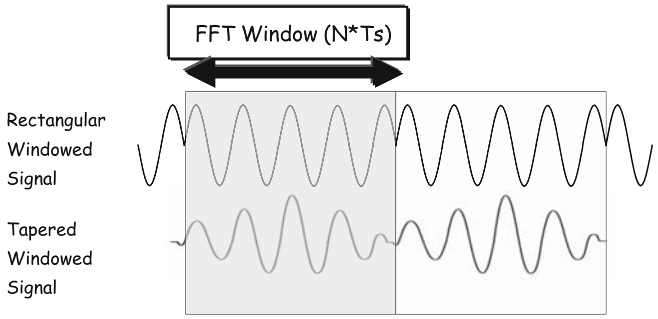


Fig. 2.11 Discontinuities across FFT blocks, reduced by applying a Window

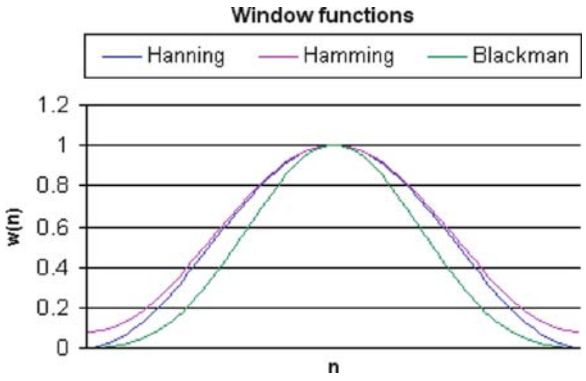


Fig. 2.12 Impulse response plots of some popular Window functions

signal. Such methods are known as Filters, and are another vital recipe in all signal processing applications; indeed, Digital Filters may be considered the backbone of Digital Signal Processing.

Introduction to Filters [1]

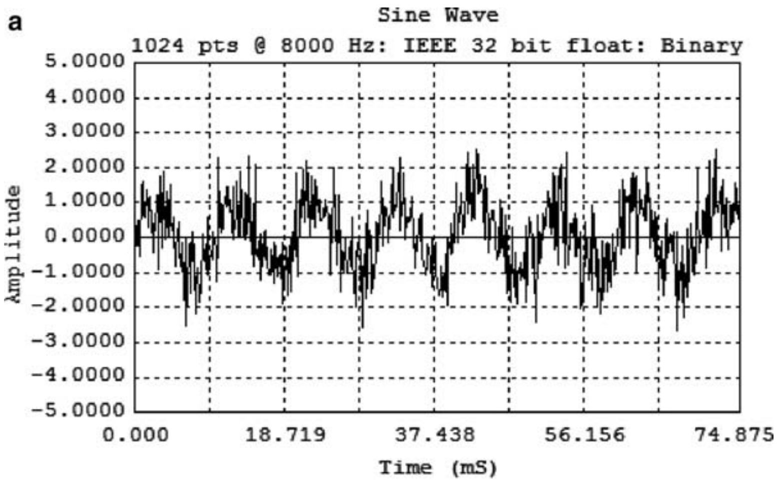
Filtering is the process of selectively allowing certain frequencies (or range of frequencies) in a signal and attenuating frequency components outside the desired range. In most instances, the objective of filtering is to eliminate or reduce the amount of undesired noise that may be corrupting a signal of interest. In some cases, the objective may simply be to manipulate the relative frequency content of a signal in order to change its spectral characteristics.

For example, consider the time-domain signal illustrated in Fig. 2.13a. Looking at the waveform, it does appear to follow the general envelope of a sinusoidal wave; however, it is heavily corrupted by noise. This is confirmed by looking at the FFT output in Fig. 2.13b, which clearly indicates a sharp peak at 100 Hz but high levels of noise throughout the overall frequency spectrum. In this particular scenario, it is relatively easy to filter the effect of the noise, because the desired signal is concentrated at a specific frequency whereas the noise is equally spread out across all frequencies. This can be done by employing a narrowly selective Band Pass Filter (we will discuss the various types of filters shortly).

Low-Pass, High-Pass, Band-Pass and Band-Stop Filters

Basic filtering problems can be broadly classified into four types:

- Low-Pass Filters
- High-Pass Filters



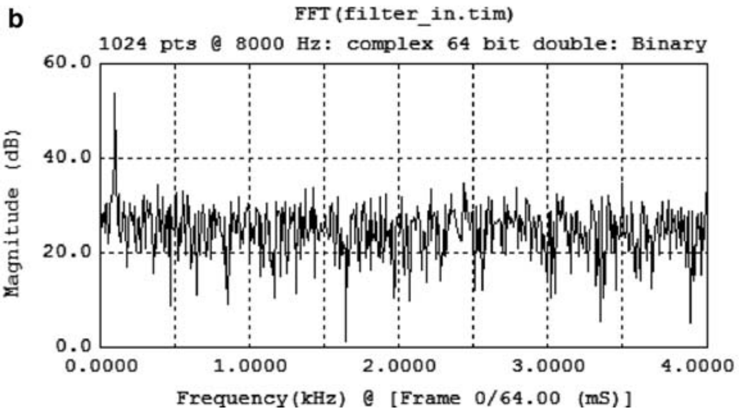


Fig. 2.13 (**a**) Sinusoidal signal heavily corrupted by Gaussian noise. (**b**) Frequency spectrum of the noise-corrupted sinusoidal signal

- Band-Pass Filters
- Band-Stop Filters

The filtering problems stated above differ according to what frequency range is desired relative to the frequency ranges that need to be attenuated, as listed below. Their idealized Frequency Response plots are depicted in Fig. 2.14.

- If it is required to allow frequencies up to a certain cutoff limit and suppress frequency components higher than the cutoff frequency, such a filter is called a Low-Pass Filter. For example, in many systems most of the noise or harmonics may be concentrated at higher frequencies, so it makes sense to perform Low-Pass Filtering.

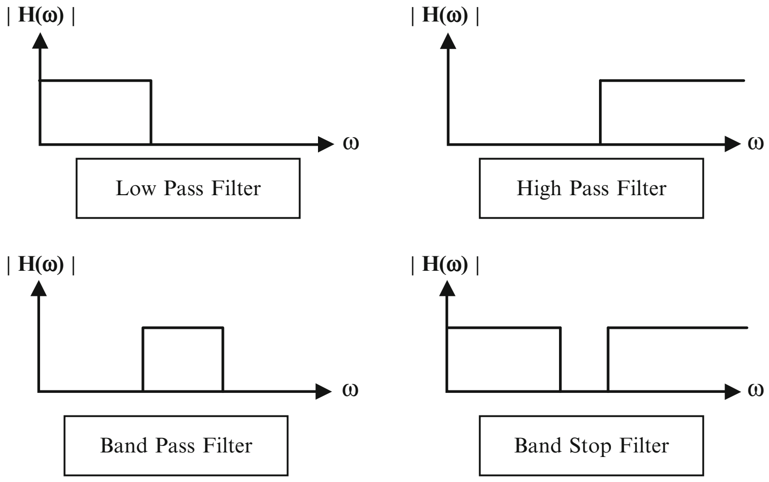


Fig. 2.14 Different filter types: low pass, high pass, band pass, and band stop

- If it is required to allow frequencies beyond a certain cutoff limit and suppress frequency components lower than the cutoff frequency, such a filter is called a High-Pass Filter. For example, some systems may be affected by DC bias or interference from the mains supply (50 or 60 Hz) which need to be rejected.
- If a certain band of frequencies is required and everything outside this range (either lower or higher) is undesired, then a Band-Pass Filter is the appropriate choice. In many Speech Processing applications, the frequencies of interest may lie in the 200–3,400 Hz range, so anything outside this band can be removed using Band-Pass Filters.
- If a certain band of frequencies needs to be eliminated and everything outside this range (either lower or higher) is acceptable, then a Band-Stop Filter should be used. For example, a signal chain could be affected by certain known sources of noise concentrated at certain frequencies, in which case a Band-Pass Filter (or a combination of multiple filters) could be used to eliminate these noise sources. A highly selective (i.e., narrowband) Band-Pass Filter is also known as a Notch Filter.

At this point, it is pertinent to understand the various parameters, or Filter Specifications, that define the desired response of the filter. It is apparent from Fig. 2.14 that the cutoff frequencies (single frequency in the case of Low-Pass and High-Pass Filters and pair of frequencies in the case of Band-Pass and Band-Stop Filters) are the key parameters. However, these Frequency Responses are idealized and therefore impractical to implement on real hardware or software. Indeed, it is fairly typical to see a band of transition between a point (on the Frequency Spectrum) where the frequency is passed and a nearby point where the frequency is suppressed. Therefore, it is generally necessary to define both the passband frequency and the stopband frequency (or a pair of pass band and stop band frequencies in the case of Band-Pass and Band-Stop Filters); these specifications have a direct bearing on how complex the filter would be to implement (e.g., a sharper transition band implies more computationally intensive filter software).

Another pair of Filter Specifications that directly affect the choice and complexity of filter implementation are the Passband Ripple, Stopband Ripple, and the Stopband Attenuation. All of these parameters are typically expressed in decibels (dB).

- Passband Ripple is the amount of variation of the Frequency Response (and therefore filter output) within the desired Passband.
- Similarly, Stopband Ripple is the amount of variation within the desired Stopband.
- Stopband Attenuation defines the extent by which undesired frequency ranges are suppressed.

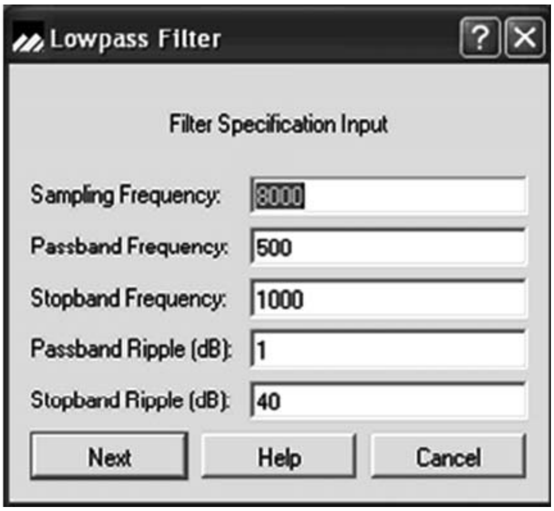
Analog and Digital Filters

Given a certain set of filter specifications, the next choice the application designer would need to make is whether to implement this filter in the Analog or Digital domain. Analog Filtering involves implementing the desired Frequency Response as an analog circuit consisting mostly of operational amplifiers and discrete components such as resistors and capacitors. In some cases, an Analog Filter is absolutely necessary, such as the Antialiasing Filters discussed earlier. In most other instances, however, Digital Filters have distinct advantages over Analog Filters when it comes to implementing in an embedded application, especially one with space, cost, or power consumption constraints. In Chap. 1, we have already seen some general advantages of digital systems over analog systems, so let me simply reiterate some of the key advantages of Digital Filters:

- Digital Filters are less affected by noise. Analog Filters can work quite erratically when subjected to high levels of noise in the system or communication channels.
- Digital Filters are not subject to temperature-related drift in characteristics and also not affected by ageing effects. Both of these are significant constraints on the long-term reliability of Analog Filters.
- Digital Filters typically consume less power, which is an enormous advantage in power-sensitive or battery-operated applications such as an Emergency Phone or Walkie-Talkie.
- Most importantly, Digital Filters are usually implemented as a software program running on the processor. Like any other software, these filters can be periodically updated for bug-fixes, reprogrammed with feature enhancements, or reused for multiple end-applications with the appropriate software customization. This level of flexibility for product developers is simply not possible with Analog Filters: changing a hardware circuit on products that have already been manufactured is a costly process and a logistical nightmare, to say the least.

An additional practical advantage of Digital Filters is that the application developer might not even need to develop the filtering software routines: many DSP/DSC suppliers and third-party software tool vendors provide GUI-based tools which not only design the filter structure (at least for the more basic, common filter types) but also generate the software to perform the filtering. All the developers might need to do at that point is to call the appropriate API functions, and the filter is a reality! The screen-shots in Fig. 2.15 show one such tool, the dsPIC Filter Design tool for the

a


Lowpass Filter

Filter Specification Input

Sampling Frequency: 8000

Passband Frequency: 500

Stopband Frequency: 1000

Passband Ripple (dB): 1

Stopband Ripple (dB): 40

b

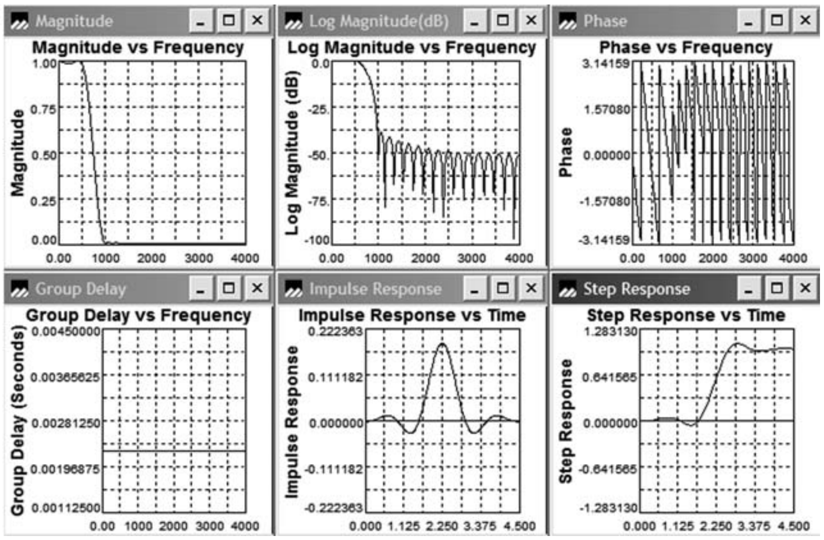


Fig. 2.15 (**a**) Entering filter specifications in a typical GUI-based filter design tool. (**b**) Filter response plots to evaluate the expected response of the filter

dsPIC DSC processor family, wherein you simply enter the Filter Specifications (a) and it shows you the expected Frequency Response of the filter (b). Once the developer is happy that the expected filter response will fulfill the needs of the application, it is a matter of few clicks to automatically generate all the software required for the filter.

FIR and IIR Filters [1, 2]

Like any other linear digital systems, the effect of a Digital Filter on an input signal is also defined by its Impulse Response. The Impulse Response, in turn, is directly related to the Frequency Response of the Digital Filter, as shown in Fig. 2.16. It follows, therefore, that a judicious design of the Impulse Response of the filter directly controls which frequency components from the input signal the filter will allow and which frequencies will be attenuated (and by how much). Filtering can, of course, be performed directly in the Frequency Domain by manipulating the FFT of a signal and then transforming it back into Time Domain by calculating its Inverse FFT. But this method is computationally more intensive and hence not as popular in embedded systems. Hence, let us focus our attention on filtering performed in the Time Domain.

Digital Filters can be classified into two primary categories based on the nature of their Impulse Responses. Each class has its own distinct characteristics and implementation requirements.

- Finite Impulse Response (FIR) Filters
- Infinite Impulse Response (IIR) Filters

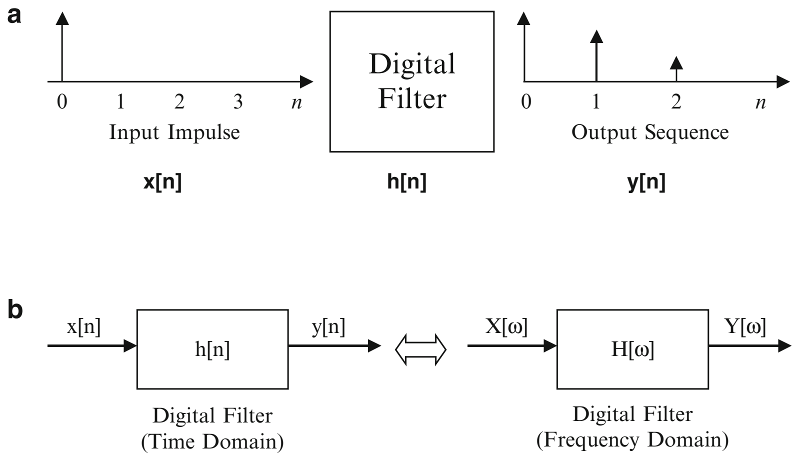


Fig. 2.16 (**a**) Impulse response of a digital filter. (**b**): Duality between a filter's impulse response and frequency response

FIR Filters

The relationship between the input and output of an FIR Filter is given by

$$y[n] = \sum_{k=0}^{T-1} b_k x[n-k], \tag{2.17}$$

where T is the Filter Order (number of filter coefficients or "taps") and b_k is the kth coefficient of the FIR Filter.

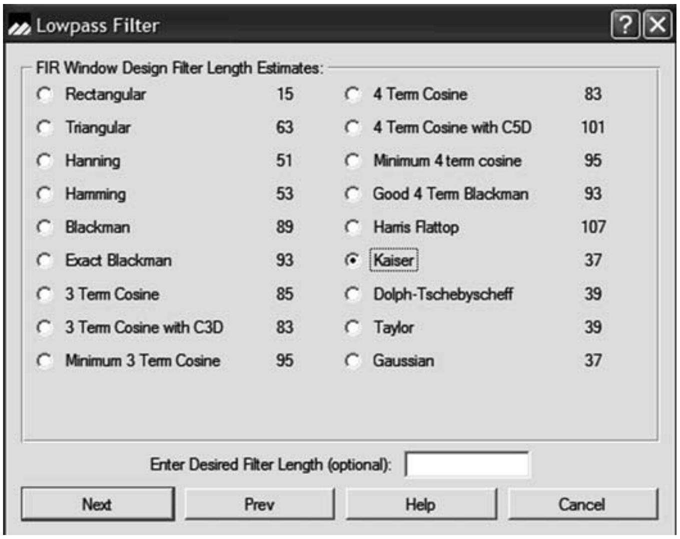
There are a couple of things that can be observed easily from the above equation:

- This computation mainly involves delays (i.e., data buffers), multiplications, and computing a sum-of-products. In other words, this is nothing but a Vector Dot Product operation. As we will see in Chap. 4, this is the most fundamental and common mathematical operation in DSP algorithms, and DSP/DSC processor architectures attempt to optimize this computation to the fullest.
- There are no feedback terms in (2.17), only feed-forward terms: each output data sample only depends on a certain number of past input samples (depending on the filter length) and there is no dependence on past outputs. This characteristic makes the FIR Filter operation inherently stable (from a mathematical standpoint) and less prone to being affected by the constraints of quantization in a processor.

Of course, it is natural to wonder why these filters are called Finite Impulse Response. This is because irrespective of the Filter Specifications, the Impulse Response invariably decays to zero after a certain number of samples, i.e., the Impulse Response of such filters is guaranteed to be "finite," similar to the general example shown in Fig. 2.16a. This is a direct by-product of the fact that there are no feedback terms in the filter equation. Another interesting feature of FIR Filters is that the phase of the input and output signals always have a linear relationship. This is a significant benefit to some applications, especially in the area of digital communications.

The most popular method of designing an FIR filter (i.e., calculating what the filter coefficients should be and how many coefficients are needed to provide the desired response) is by using Windows. As we have already seen, Window functions serve not only to smooth discontinuities in the signal across block boundaries but also to impart certain characteristics to the Frequency Response. To achieve the same set of Filter Specifications, different windows require different number of filter coefficients, as illustrated in the screen-shot from dsPIC Filter Design tool in Fig. 2.17.

FIR Filters have certain disadvantages too, relative to IIR Filters that we are going to discuss next. To satisfy similar Filter Specifications, an FIR Filter requires dramatically larger filter lengths compared to an IIR Filter. Moreover, with large filter lengths there is some unavoidable delay needed for the data buffer (also known as the Delay Line) to fill up, so there may not be a reasonable output signal for several samples.


Lowpass Filter

FIR Window Design Filter Length Estimates:

Rectangular	15	4 Term Cosine	83
Triangular	63	4 Term Cosine with C5D	101
Hanning	51	Minimum 4 term cosine	95
Hamming	53	Good 4 Term Blackman	93
Blackman	89	Harris Flattop	107
Exact Blackman	93	[✓ Kaiser]	37
3 Term Cosine	85	Dolph-Tschebyscheff	39
3 Term Cosine with C3D	83	Taylor	39
Minimum 3 Term Cosine	95	Gaussian	37

Enter Desired Filter Length (optional):

Fig. 2.17 Example of different Window functions needing different filter lengths

IIR Filters

The relationship between the input and output of an IIR Filter is given by the following equation

$$y[n] = \sum_{k=1}^{p} a_k y[n-k] + \sum_{k=0}^{p} b_k x[n-k], \tag{2.18}$$

where P is the Filter Order (number of filter coefficients or "taps"), a_k the kth feedback coefficient of the IIR Filter, and b_k is the kth feed-forward coefficient of the IIR Filter.

The following characteristics can be inferred from a close study of (2.18):

- This computation involves delays (i.e., data buffers) for both input and output data samples, unlike FIR which only required storage and computations on past input samples. It also requires multiplications and computing two different sum-of-products. However, the filter order is generally much lower than FIR Filters, so the sum-of-product computations in IIR Filters are not nearly as intense as in FIR Filters.
- There are both feedback and feed-forward terms in (2.18): each output data sample only depends on a certain number of past input samples (depending on the filter length) but also on a certain number of past output samples. While this characteristic makes the FIR Filter operation potentially stable (from a mathematical

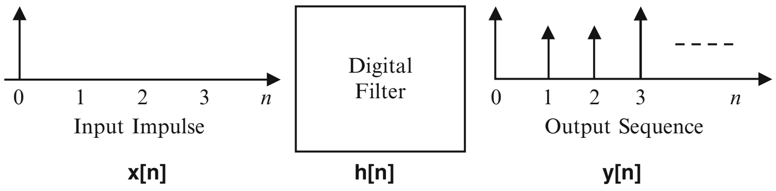


Fig. 2.18 Impulse response of an IIR Filter

standpoint) and more susceptible to quantization effects, it also ensures that the filter characteristics obtained are much sharper than a comparable FIR Filter even with smaller filter lengths.

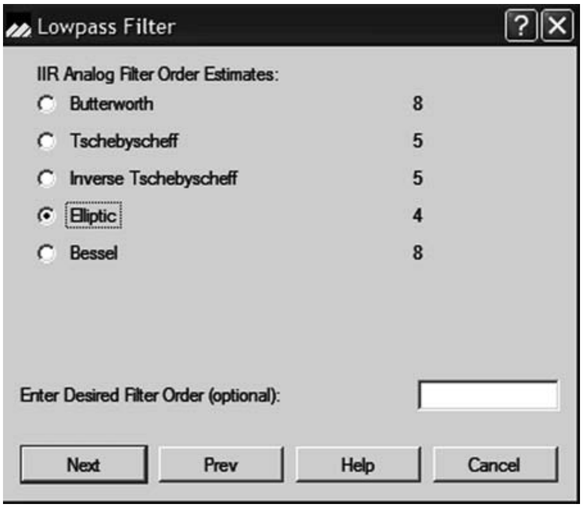
Unlike FIR Filters, the Impulse Response of an IIR Filter is not guaranteed to decay to zero after a certain number of samples, i.e., it is possible for the Impulse Response of such filters to be "infinite," as shown in Fig. 2.18. This is a direct by-product of the fact that there are feedback terms in the filter equation. Also, unlike FIR Filters the input–output phase relationship is generally nonlinear.

Unlike FIR Filters, the coefficients of an IIR Filter are never designed directly in the digital domain. Interestingly, the standard practice in IIR Filter Design is to start with a suitable Analog Filter Model and then apply mathematical transformations on the Analog Filter (usually in the z-transform domain) to obtain the corresponding Digital IIR Filter that would provide a similar Frequency Response. Several methods exist for this approach, such as Impulse Invariance and Bilinear Transformation. A detailed study on these design methodologies is left to the reader, as several texts exist on this topic. In any case, many embedded system designers and DSP developers are spared of the task of calculating IIR Filter coefficients as software tools can perform this task automatically and generate IIR Filter coefficients (and sometimes IIR Filter software routines) that the user can directly incorporate into the application. An example screen-shot is shown in Fig. 2.19, where different Analog Filter models are listed that would result in IIR Filters of different filter orders.

Since IIR Filters tend to be susceptible to quantization effects and numerical overflows, especially at higher filter orders, a very popular practice is to implement an IIR Filter as a cascaded section of "biquads" (second-order IIR Filter sections) with the output of each biquad section providing the input for the next. By keeping each filter section limited to the second order, this strategy greatly enhances the numerical stability of the IIR Filter.

No discussion of IIR Filters would be complete without discussing the various filter structures or "topologies" than can be used to obtain the same filter response. Note that all of the topologies listed below are derived from the same base input–output relationship as defined by (2.18).

The topology that results from a direct application of (2.18) is called Direct Form I, as shown in Fig. 2.20. As is evident from the figure, this topology requires twice as many delay elements (and hence data buffer size) as the filter order, and is therefore not considered optimal from a memory usage perspective.


Lowpass Filter

IIR Analog Filter Order Estimates:

( ) Butterworth	8
( ) Tschebyscheff	5
( ) Inverse Tschebyscheff	5
[✓ Elliptic]	4
( ) Bessel	8

Enter Desired Filter Order (optional):

Fig. 2.19 Analog filter models used to generate IIR filter coefficients

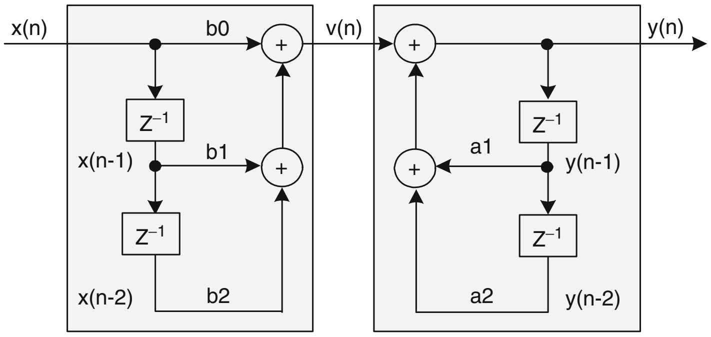


Fig. 2.20 IIR direct form I topology

On the contrary, the topology illustrated in Fig. 2.21, which has been obtained simply by rearranging Fig. 2.20 and ensuring that the delay elements are shared between the feedback and feed-forward sections, is more optimized from a memory usage standpoint. This structure is called Direct Form II and is also referred to as the Canonic Form of IIR Filters because it requires the minimum number of delay elements.

Taking Fig. 2.21 and merely swapping the order of execution of all the elements therein, i.e., by "transposing" Direct Form II, we obtain the topology of Fig. 2.22. This is known as the Transposed Form of IIR Filters, for obvious reasons. In reality, the Transposed Form is simply an alternate representation of Direct Form I.

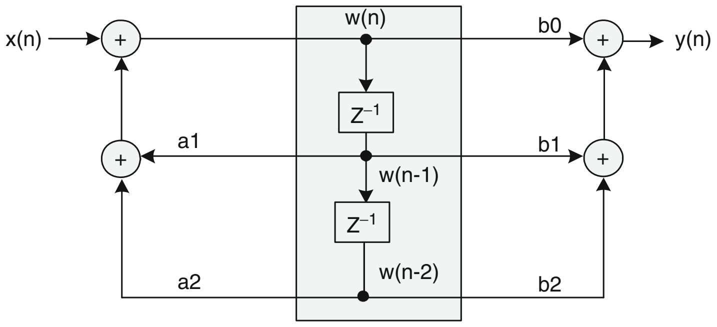


Fig. 2.21 IIR direct form II (canonic) topology

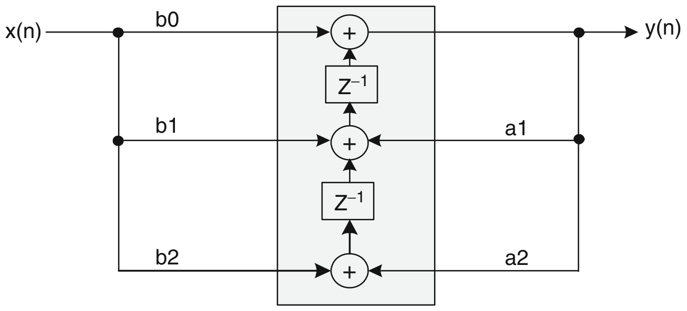


Fig. 2.22 IIR transposed direct form II topology

Interpolation and Decimation [1, 2]

In many signal processing applications, the number of samples of a signal that are available for a given time period may need to be changed to match the requirements of specific algorithms or processes used in the application. This is especially common in speech processing applications; for example, suppose a system is using an external codec (coder–decoder) device to sample and convert an analog speech signal (e.g., from a microphone) operating at a fixed sampling rate of 12 kHz. Now, let us assume the speech needs to be compressed and recorded using a standardized compression algorithm known as G.729, which happens to have been designed to operate at an 8-kHz sampling rate. Obviously, there would be a mismatch in the number of samples available for each speech frame (processing interval). This can be compensated by downsampling the sampled signal to 8 kHz. Conversely, while decompressing and playing back the recorded signal the samples need to be upsampled to 12 kHz. In signal processing lingo, the processes of upsampling and downsampling are known as Interpolation and Decimation, respectively. In general, Interpolation and Decimation are applied to increase/decrease the sampling rate by an integral factor, whereas in this specific example the ratio of interest is 1.5:1, hence a combination of interpolation and decimation must be used. For example, on the input side the samples need to be interpolated by a factor of 2, resulting in a 24-kHz sampling rate; then, it needs to be decimated by a factor of 3 to yield the needed 8-kHz sampling rate.

In practice, *after* interpolating by a factor of L (typically by inserting $L-1$ zeros after each original sample), the signal must be low-pass filtered to band limit it in accordance with the Nyquist–Shannon Theorem, in order to prevent aliasing effects. Conversely, *before* decimating a signal by a factor of M (typically by picking out every Mth sample in the frame), the signal must be low-pass filtered. Thus, Interpolation and Decimation may also be thought of as filtering operations combined with sample-rate conversion; indeed, Interpolation and Decimation functions are usually incorporated within filtering routines.

Summary

In this chapter, I have briefly described some fundamental concepts and techniques in Digital Signal Processing, with special focus on FIR and IIR Filters, DFT and FFT, as well as more basic operations such as Convolution and Correlation. I hope this discussion should provide embedded application developers with some foundational understanding and appreciation of the role played by various key signal processing operations, as well as the different design choices available for each of these operations. With the help of this knowledge and judicious usage of available development tools and software libraries for the embedded processing platform of interest, application developers should be well equipped to incorporate these techniques into their product development efforts.

Now that we have a reasonably good understanding of fundamental signal processing topics; let us delve into those concepts and techniques that form the building blocks in the subject of Speech Processing. As we will see in Chap. 3, Speech Processing relies heavily on the specific characteristics of speech signals and the human speech production system and is closely related to other areas of study such as Acoustics and Linguistics.

References

1. Proakis JG, Manolakis DG Digital signal processing – principles, algorithms and applications, Morgan Kaufmann, 2007.
2. Rabiner LR, Schafer RW Digital processing of speech signals, Prentice Hall, 1998.
3. Chau WC Speech coding algorithms, CRC Press, 2001.

Chapter 3
Basic Speech Processing Concepts

Abstract Before we explore the algorithms and techniques used to process speech signals to accomplish various objectives in an embedded application, we need to understand some fundamental principles behind the nature of speech signals. Of particular importance are the temporal and spectral characteristics of different types of vocal sounds produced by humans and what role the human speech production system itself plays in determining the properties of these sounds. This knowledge enables us to efficiently model the sounds generated, thereby providing the foundation of sophisticated techniques for compressing speech. Moreover, any spoken language is based on a combination and sequence of such sounds; hence understanding their salient features is useful for the design and implementation of effective speech recognition and synthesis techniques. In this section, we will learn how to classify the basic types of sounds generated by human voice and the underlying time-domain and frequency-domain characteristics behind these different types of sounds. Finally, and most importantly, we will explore some popular speech processing building-block techniques that enable us to extract critical pieces of information from the speech signal, such as which category a speech segment belongs to, the pitch of the sound, and the energy contained therein.

Mechanism of Human Speech Production [1, 3]

In order to better understand the different types of sounds produced by humans, we need to first gain a rudimentary understanding of the mechanisms which produce these sounds in the first place, i.e., the human vocal system. Speech is basically generated when sound pressure waves (typically originating from air pushed out by the lungs) are channeled or restricted in various ways by manual control of the anatomical components of the human vocal system. How the sound wave travels through the entire anatomical system until it is radiated from the lips to the outside world controls the type of sound produced (and heard by a listener). The study and classification of these various sounds is known as Phonetics.

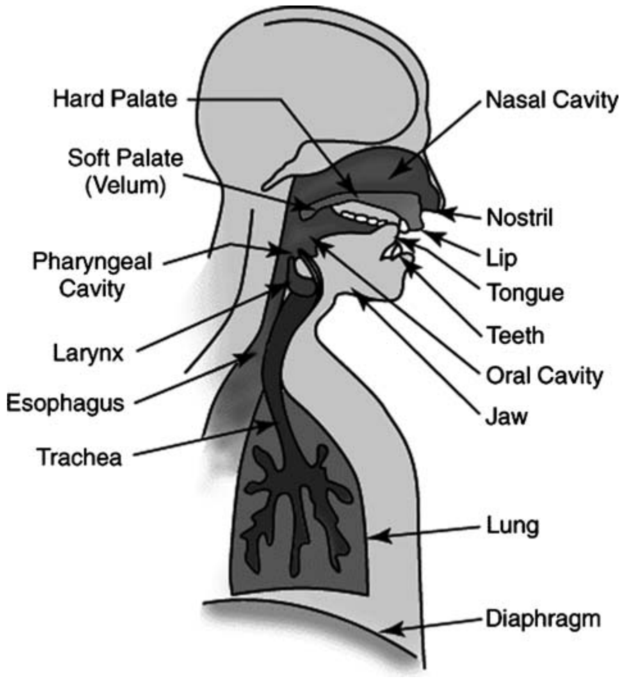


Fig. 3.1 Basic components of the human speech production system [5]

Figure 3.1 illustrates the key anatomical components of the human speech production system. While not delving a detailed biological discussion of this system, let us look at the key portions that affect how speech is produced.

- The subglottal section of this structure, comprising of the lungs, Bronchi, Trachea, and Esophagus, serves as the original source of the sound energy, i.e., the place where the sound waves emanate.
- The Pharynx, or pharyngeal cavity, is the section from the Esophagus to the mouth.
- The Oral Cavity is the cavity just inside the mouth. Together, the Pharynx and the Oral Cavity (which really forms a composite region where sound pressure waves can travel and be manipulated) are popularly known as the Vocal Tract.
- A critical part of the Larynx is a set of folds of tissue which are collectively known as the Vocal Cords. As we will see shortly, the Vocal Cords play a direct role in controlling the periodicity (pitch) of certain types of speech.
- An alternate path for sound waves to emerge into the ambient air is the cavity leading to the nose, which is known as the Nasal Cavity.
- Apart from the three cavities and vocal cords listed above, other organs such as the lips, teeth, tongue, nostril, and soft palate (Velum) also play a critical role in controlling the specifics of the generated sound.

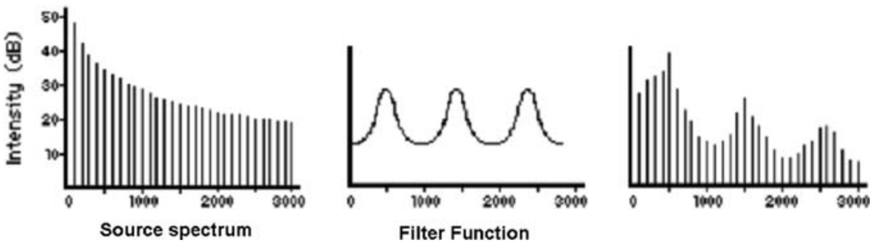


Fig. 3.2 Effect of formant frequencies in the vocal tract response [6]

When air is pushed out of the lungs, the resulting air flow can be subjected to different constrictions and perturbations, all of which together determine what the generated sound would be. From a linguistic perspective, spoken words or syllables are conceptually broken up into subsyllables called Phonemes. For example, the utterance of a single letter might be an example of a Phoneme, though there could certainly be multiple phonemes associated with the same letter when pronounced in different ways and in different languages.

The Vocal Tract acts as a filter for the sound waves entering it, effectively creating poles or maxima at a certain frequency (known as Fundamental Frequency) and its harmonics, as shown in Fig. 3.2. These pole frequencies are collectively referred to as Formant Frequencies, and the Formant Frequencies uniquely characterize the vocal tract response for a particular voiced sound. It will be seen in a later chapter that successfully estimating the Formant Frequencies enables effective and efficient Speech Compression.

Types of Speech Signals [1, 2]

We have already seen the basic structure of the human speech production system. To understand and appreciate the effect of the anatomical structures and movements even further, let us now discuss the main categories of vocal sounds (voiced, unvoiced, fricatives, stops, and nasal) and how they are influenced by the interaction of the vocal cords and the vocal tract. As we will see in later chapters, classification of speech sounds is a significant and important problem in speech signal processing algorithms.

Voiced Sounds

Voiced sounds are produced when air flows produced by the lungs are forced through the Glottis on to the vocal cords. The vocal cords consist of folds of tissue stretched across the opening of the Larynx; the tension on these tissues can be

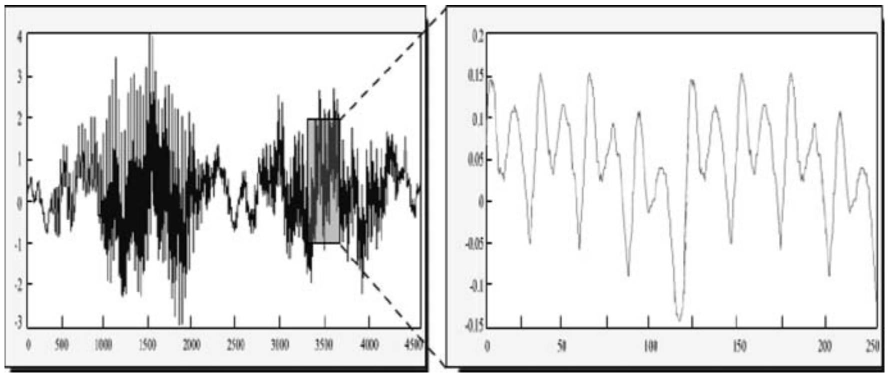


Fig. 3.3 Example of a voiced segment within a speech signal

adjusted by the person, thereby producing a lateral movement of the vocal cords that cause them to vibrate. These vibrations of the vocal cord cause the air flow to exit the Larynx (through a slit between the tissues, known as Glottis) in a roughly periodic or "quasiperiodic" pattern. These periodic pulses of air, when not subject to any other major constrictions further down the vocal tract, produce speech at the lips that have strong quasiperiodic characteristics. This process of modulation of the pressure wave by vibrating vocal cords is commonly known as Phonation, and the specific characteristics of this modulation are determined not only by the tension applied to the vocal cords but also by the structure and mass of the vocal cords themselves. In summary, voiced sounds are distinguished by the presence of periodicity in the corresponding acoustic waveforms, as shown in the voiced speech segment highlighted in Fig. 3.3.

The resonant frequency at which the vocal cords vibrate, which is known as the Pitch Frequency, is directly related to the perception of Pitch in voiced speech. This Pitch Frequency is typically in the range of 50–200 Hz in the case of male speakers, whereas it can be more than twice as much in the case of female speakers; hence the perception of the average female voice has a "high pitch" relative to the average male voice. In general, voiced speech segments are associated with the vowel sounds of the English language. In some cases, the associated phoneme might consist of not one vowel but two or more vowels occurring successively: these are also known as Diphtongs.

From Fig. 3.3 (or any graphical representations of speech utterances), it should also be apparent that a speech sequence consists of a series of shorter speech segments, and each segment should be analyzed to determine if it contains voiced speech or not. Fortunately, the characteristics of speech signals (i.e., the type of sound) do not change very frequently, and analysis windows in the 20–30 ms range are the most common (and in fact standardized in several speech processing industry standards). A Short Time Fourier Transform (or an FFT computation thereof) can be utilized to inspect the changing periodic properties of successive speech segments. As we will see in the context of Voice Coders, successful identification of voiced speech segments is vital to the effectiveness of many algorithms (although this identification might not be made by explicitly computing an FFT. . .but more on that later).

Unvoiced Sounds

Unlike voiced speech, unvoiced sounds do not have any underlying periodicity, and therefore do not have a distinct association with Pitch. In fact, in speech processing algorithms, the unvoiced speech entering the vocal tract is generally modeled as a random white noise source, which turns out to be a fairly good approximation in practical applications. The critical difference between unvoiced and voiced sounds is that in unvoiced sounds there is no significant vibration of the vocal cords (which was the source of the periodicity in voiced speech).

In general, unvoiced sounds are closely related to utterance of consonants in the English language; even intuitively it can be observed that one cannot estimate the pitch of a person's voice by only hearing a consonant-based phoneme being uttered.

Voiced and Unvoiced Fricatives

In a class of phonemes known as Fricatives, the source of the excitation is a partial constriction in the vocal tract, resulting in a localized turbulence and hence noise-like properties in the generated speech. Fricatives can be either unvoiced (e.g., /f/ or /sh/) or voiced (e.g., /th/ or /z/). The main difference is that in voiced Fricatives, the noisy characteristics caused by the constriction are also accompanied by vibrations of the vocal cords, thereby imparting some periodicity in the produced sound.

Voiced and Unvoiced Stops

Many unvoiced sounds are produced when there is some kind of constriction in the vocal tract that causes it to be completely closed. For example, when uttering the phoneme /p/ this constriction is at the lips, whereas for /g/ it is at the back of the teeth. In fact, these two examples are part of a category of sounds referred to as Stops. It may be noted that not all Stop sounds are unvoiced; indeed, /b/ and /d/ are examples of voiced Stop sounds. The main difference is that when pressure builds up in the vocal tract due to the constriction, the vocal cords do not vibrate in the case of unvoiced Stops whereas they do vibrate in the case of voiced Stops.

Nasal Sounds

Finally, there are several sounds that are inherently nasal in their characteristics, such as /m/ and /n/. These are generated when the Velum is lowered such that oral cavity is constricted (though still coupled with the Pharynx) and the air pressure flows through the Nasal Tract instead. Here, the mouth acts as a resonant cavity for certain frequencies. The spectral responses for nasal sounds are typically broader (damped) than for voiced sounds due to the coupling of the oral and nasal tracts. Note that there is also greater energy loss due to the intricate structure of the nasal passage.

Digital Models for the Speech Production System [1, 2]

In many speech processing applications, it is necessary to utilize a model of the speech production system that can be incorporated into the algorithm of interest. For example, most sophisticated speech compression techniques represent a given segment (frame) of speech in terms of a finite set of parameters in order to reduce the amount of data to be processed. These parameters often represent various aspects of the sound generated as well as the specific formant frequency characteristics of the vocal tract for that sound or utterance. Needless to say, the models employed are simply mathematical approximations of the actual biological and acoustical processes rather than accurate representations. The key requirement is that the model should provide just enough information about the speech signals that the speech can be analyzed (or synthesized, as the case may be) to satisfy the application requirements.

In general, the items that need to be incorporated in any speech production model are:

- Lungs and vocal cords
- Vocal tract, including any effects of radiation at the lips and losses in the nasal cavity

The first item is referred to as the Excitation, essentially meaning the source of the sound waves and associated vibrations (if any). The second item represents the effect of the vocal tract and other passages the air flow passes through, i.e., the Articulation; in signal processing or control theory terminology this may be thought of as the "system."

As we have discussed in previous sections, the nature of the acoustic wave entering the vocal tract is dependent on the type of sound. Often (though not necessarily), a determination must be made on a per frame basis about whether the sound produced is voiced or unvoiced. Based on our knowledge of the action of the vocal cords for voiced speech, the excitation can be modeled as a periodic pulse generator in the case of frames with voiced sounds. For unvoiced sounds, we can ignore the effect of the vocal cords and model the lungs as a source of random white noise signals during frames with unvoiced speech.

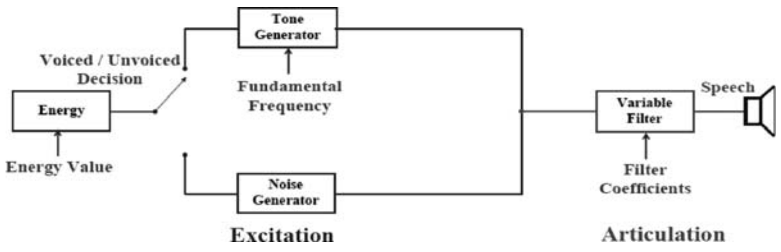


Fig. 3.4 Simplified digital model for human speech production system

We have also seen that the vocal tract, with its Formant Frequency characteristics, shapes the spectral characteristics of the sound passing through it. In the Frequency Domain, this basically means that the formant frequencies are emphasized relative to others. In the Time Domain, this easily lends itself to a linear filtering model, enabling the vocal tract system to be modeled purely as a linear filter whose characteristics may vary from one speech segment to another based on the changing shape of the vocal tract for different phonemes.

The overall model is shown in Fig. 3.4. As mentioned before, this is a simplistic representation, and the models employed by modern speech coders and other speech processing algorithms contain a number of refinements and additional subtleties to allow more accurate speech analysis and/or synthesis.

The filter used to model the vocal tract response can be a simple all-pole filter. Thus, the impulse response of the filter is:

$$H(z) = \frac{G}{1 - \sum_{k=1}^{N} a_k z^{-k}}, \tag{3.1}$$

where G is the gain of the vocal tract response and a_k is the filter coefficient.

However, to obtain a reasonably accurate model for nasal and fricative sounds, it is beneficial to have zeros as well as poles (since both resonances and antiresonances are involved), resulting in a conventional pole–zero IIR filter.

Since the coefficients are real, the roots of the denominator polynomials are either real numbers or appear as complex conjugate pairs, which lead to some interesting optimizations to enhance the stability of the poles (this is described in greater detail in later chapters in the context of LPC coefficients). As described in Chap. 2, this IIR filter can also be implemented as cascaded sections of second-order biquad sections.

Alternative Filtering Methodologies Used in Speech Processing [2, 4]

At this point, it would be pertinent to briefly explain two alternative formulations for filter topologies, both of which are particularly suited to certain speech processing algorithms. These are as follows:

- Lattice Structure, as an alternative to Direct Form structures.
- Zero-Input Zero-State method for computation of a filter on a frame-by-frame basis, as an alternative to the usual convolution-sum method.

Lattice Realization of a Digital Filter

The Lattice Form of an all-pole filter is shown in Fig. 3.5 as a Signal Flow Graph. The small circles indicate addition and the k_n multiplication factors are coefficients known as Reflection Coefficients. Readers are advised to see the references if a detailed explanation or derivations are of interest.

From a study of a detailed model of the vocal tract, it is seen that the Reflection Coefficients represent the sound energy that is reflected at the junctions of successive tube sections of the vocal tract (assuming a cascaded tube model for the entire vocal tract). The Reflection Coefficients can be computed from the Direct Form coefficients as given below.

For $l = M, M-1, \ldots, 1$:

$$k_l = -a_l^{(l)}, \tag{3.2}$$

$$k_l = \frac{a_i^{(l)} + k_l a_{l-i}^{(l)}}{1-k_l^2} a_l^{(l)} \quad \text{for } i = 1, 2, \ldots, (l-1), \tag{3.3}$$

where $a_i = a_i^{(M)}$.

Using the Reflection Coefficients thus obtained for each stage, the $v_i[n]$, $y[n]$, and $u_i[n]$ values can be obtained iteratively. These computations are shown in (3.4).

$$\begin{aligned}
v_{M-1}[n] &= x[n] + k_M u_{M-1}[n-1] \\
v_{M-2}[n] &= v_{M-1}[n] + k_{M-1} u_{M-2}[n-1] \\
&\vdots \\
v_1[n] &= v_2[n] + k_2 u_1[n-1] \\
y[n] &= v_1[n] + k_1 y[n-1]
\end{aligned} \tag{3.4}$$

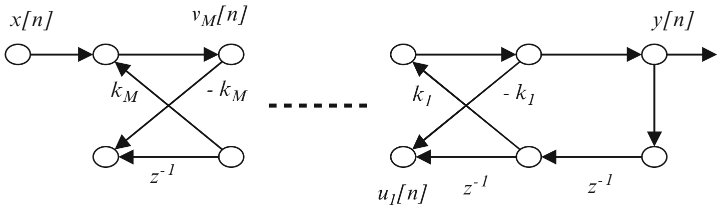


Fig. 3.5 Lattice form realization of an all-pole filter

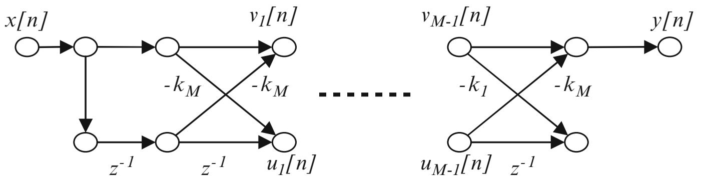


Fig. 3.6 Lattice form realization of an all-zero filter

$$u_1[n] = -k_1 y[n] + y[n-1]$$
$$u_2[n] = -k_2 v_1[n] + u_1[n-1]$$
$$\vdots$$
$$u_{M-1}[n] = -k_{M-1} v_{M-2}[n] + u_{M-2}[n-1]$$

The Lattice Form of an all-zero filter is shown in Fig. 3.6.

Just as with all-pole filters, the Reflection Coefficients can be computed from the Direct Form coefficients as given below.

For $l = M, M-1, \ldots, 1$:

$$k_l = -a_l^{(l)} \tag{3.5}$$

$$k_l = \frac{a_i^{(l)} + k_l a_{l-i}^{(l)}}{1-k_l^2} a_l^{(l)} \quad \text{for } i = 1, 2, \ldots, (l-1), \tag{3.6}$$

where $a_j = a_j^{(M)}$.

Using the Reflection Coefficients thus obtained for each stage, the $v_i[n]$, $y[n]$, and $u_i[n]$ values can be obtained iteratively. These computations are shown in (3.7).

$$v_1[n] = x[n] - k_1 x[n-1]$$
$$u_1[n] = -k_1 x[n] + x[n-1]$$
$$v_2[n] = v_1[n] - k_2 u_1[n-1]$$
$$u_2[n] = -k_2 v_1[n] + u_1[n-1]$$
$$\vdots$$
$$y[n] = v_{M-1}[n] - k_M u_{M-1}[n-1] \tag{3.7}$$

Note that many DSP and Digital Filtering Libraries provided by DSP/DSC and tool vendors often include Lattice Filtering functions that may be directly invoked by the application software.

Looking at Figs. 3.5 and 3.6 as well as the resultant equations, it may seem impractical to implement the more computationally intensive Lattice filter topology instead of the Direct Form topologies. Indeed, in many cases using Direct Form may be the better implementation choice, and many standardized speech processing algorithms recommend it too.

However, the Lattice methodology has an added advantage in that the Reflection Coefficients are automatically obtained as a by-product of the filter computation. As we will see in later chapters, some speech compression techniques involve computing the inverse of Autocorrelation matrices, and the popular shortcut methods that exist to accomplish this operation utilize these Reflection Coefficients in the computation. Hence, there are certainly applications and algorithms in which the overall benefit of using the Lattice Structure outweighs the greater computation costs.

Zero-Input Zero-State Filtering

This method of computing a filter output is an alternative to the conventional method wherein the filter state for the current frame (speech segment) is saved in a delay line and used during the filter computation in the next frame. While this is definitely the simplest and most efficient method of implementing a filtering algorithm, there are some speech processing algorithms (most notably the CELP speech compression algorithm) that benefit from using an approach called Zero-Input Zero-State Filtering and can also utilize a technique called Recursive Convolution to reduce the computational burden of the Zero-State portion of the computation.

In this methodology, the filter output is split into two different responses and computed separately:

- Zero-Input: This is the part of the filter output that exists only due to the effect of past history. This component completely ignores the effect of the new input samples from the current frame.

$$s_r^{zi}[n] = y_{r-1}[n+N], \quad \text{where } -M \le n \le -1, \tag{3.8}$$

$$s_r^{zi}[n] = -\sum_{i=1}^{M} a_i s_r^{zi}[n-i], \quad \text{where } 0 \le n \le N-1. \tag{3.9}$$

- Zero-State: This is the part of the filter output that exists only due to the effect of the new input samples from the current frame. This component completely ignores any effect of the existing state from previous frames.

$$s_r^{zs}[n] = 0, \quad \text{where } -M \le n \le -1, \tag{3.10}$$

$$s_r^{zs}[n] = x_r[n] - \sum_{i=1}^{M} a_i s_r^{zs}[n-i] \quad \text{where } 0 \le n \le N-1. \tag{3.11}$$

- Final Output: The overall output $y_r[n]$ is obtained simply by adding the above outputs for each sample index n.

$$y_r[n] = s_r^{zi}[n] + s_r^{zs}[n], \quad \text{where } 0 \le n \le N-1. \tag{3.12}$$

Some Basic Speech Processing Operations [1, 2]

Let us briefly explore some commonly used speech processing operations that can be utilized to distinguish between voiced and unvoiced speech segments, estimate the pitch period, and also to distinguish between periods of speech and silence. In many cases, using more than one of these operations is useful in order to enhance the robustness of the speech processing system.

Short-Time Energy

The Short-Time Energy, computed on a segment of speech samples, can help differentiate between voiced and unvoiced speech on a per-frame basis. This is because the amplitude of unvoiced sounds is typically smaller than that of voiced sounds. This can in turn be used to fine-tune the speech production model for every speech segment by generating the appropriate excitation signal (white noise or periodic tones).

The Short-Time Energy of a speech segment with N samples is given by:

$$E_n = \sum_{n=0}^{N-1} (x[n])^2. \tag{3.13}$$

This equation assumes that a rectangular window of length N samples has been used. To smooth out the energy within a given segment, a window other than a Rectangular Window can be applied to the signal.

Keep in mind that the length of the window should be reasonably long so that fluctuations of the energy within the frame can be smoothed out (since the window acts as a low pass filter). However, the window should not be so long that the legitimate variations in energy from one segment to another are overlooked. One disadvantage of using the Short-Time Energy metric is that large signal levels will tend to skew the overall energy estimate considerably as the computation is based on the square of individual sample values.

Average Magnitude

The sensitivity to temporarily large signals that is a disadvantage of Short-Time Energy can be avoided by using the Average Magnitude metric instead. This is defined as:

$$M_n = \sum_{m=0}^{N-1} (|x[m]| \times w[n-m]), \tag{3.14}$$

where $w[n]$ is any suitable window.

In this case, the weighted sum of absolute values of the signal is calculated, thereby completely avoiding the squared term and also resulting in reduced computation. However, a disadvantage is that the difference between the average magnitude values for voiced and unvoiced speech is not as visible as when Short-Time Energy is used, hence Average Magnitude by itself may be less effective for voiced/unvoiced decision making.

Short-Time Average Zero-Crossing Rate

The Short-Time Average Zero-Crossing Rate is a simplistic means of determining the periodicity of a speech signal. It can also be used in conjunction with the Short-Time Energy to differentiate between segments with and without speech, thereby providing a reasonably good estimate of when a period of speech has begun, e.g., to detect word beginnings and endings in an isolated word recognition algorithm. Moreover, some types of speech sounds such as Plosives are characterized by a sharp transition between speech and silence and these can be detected by using a combination of these two terms.

Zero-Crossing Rate merely counts the number of times the sign of speech samples changes within a given frame. Thus, the number of zero crossings is a crude estimation of the period of a signal, e.g., a 100-Hz sinusoid can be expected to have four zero-crossings within a 20-ms measurement interval.

Pitch Period Estimation Using Autocorrelation

As discussed in Chap. 2, the autocorrelation of a signal is periodic when the signal itself is periodic, and therefore the peaks in the autocorrelation output can be used as a rough measure of the periodicity of the signal.

For a given signal $s[n]$ and a frame of length N, the autocorrelation value for shift index m is given by

$$R[l,m] = \sum_{n=m-N+1}^{m} s[n] \times s[n-l]. \tag{3.15}$$

A peak-picking procedure can be employed on the autocorrelation output points to determine the number of periods of the signal present in the processing interval, thereby yielding the period of voiced speech. For unvoiced speech, this measure is irrelevant. Also, bear in mind that the length of the frame should be large enough that it encompasses all possible values of l, since the value of l in (3.15) define the range in which the Pitch Estimation is being performed.

Pitch Period Estimation Using Magnitude Difference Function

The autocorrelation requires a large amount of computation, especially multiplications. A substantially simpler technique to estimate the pitch of a speech signal is to simply subtract successive samples of speech and then add these differences over the entire frame, as given in (3.16). When this expression (known as Magnitude Difference Function or simply MDF) is computed for different values of lag corresponding to different pitch periods, the pitch period can be estimated. The lag can be iteratively adjusted to arrive at the best-fit period.

The MDF of a signal $s[n]$ for a lag l and index m is given by

$$\mathrm{MDF}[l, m] = \sum_{n=m-N+1}^{m} |s[n] - s[n-l]|. \tag{3.16}$$

Key Characteristics of the Human Auditory System [3]

Gaining a basic understanding of how the human auditory system, i.e., the listening system, works and the ways in which the properties of human hearing affect the perception of spoken words is an important tool in optimizing speech processing systems. Speech processing algorithms can be designed in such a way as to exploit the response of the human ear to sounds of various types, particularly in terms of the amplitude and frequency of sounds. For example, many speech compression techniques utilize techniques to postprocess the encoded parameters of speech such that those portions of data to which that the human ear is more sensitive are emphasized relative to those that are less clearly perceived by the human auditory system.

Basic Structure of the Human Auditory System

Figure 3.7 depicts the basic structure of the human auditory system. The sound waves reaching the human ear (the external organ is also known as the Pinna)

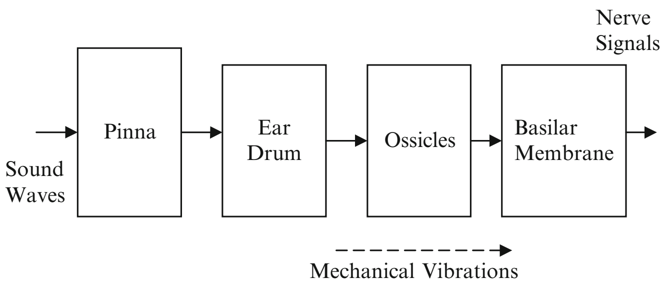


Fig. 3.7 Simplified structure of the human auditory system

pass through the ear canal and is funneled by it. The sound waves then reach the eardrum, whose main function is to convert these waves into mechanical vibrations through a fine membrane, much like many microphones do. These vibrations are then transferred by bones known as Ossicles into a spiral-shaped hard organ called the Cochlea, which in turn transfer the information about the impinging sound to the neural system, and therefore to the brain for further analysis.

Now, there is a membrane called the Basilar Membrane which vibrates at varying frequencies depending on the nature of sound waves transferred by the Ossicles. Along different portions of the membrane, the frequency responses are different so much, so that it can be reasonably modeled by a bank of digital filters. The movements of the Basilar Membrane are detected by hair cells in the Cochlea that faithfully transmit this "frequency domain" data to the nerves, thereby completing the sound reception signal chain.

Absolute Threshold

The Absolute Threshold of a human auditory system is simply the lowest level of incoming sound (in decibels, or dB) that can actually be detected by the person. In other words, it defines the "threshold of hearing" of a human ear for a single uncorrupted sound wave. This characteristic varies across the entire audible frequency range, and is therefore typically plotted as a frequency response curve. It must be noted that the Absolute Threshold characteristic of each human being is somewhat different, and therefore a particular Absolute Threshold curve is only relevant for a particular listener. While the biological maximum listening range of human beings is 20 Hz to 20 kHz, the auditory system is typically more sensitive in the 1–4 kHz range, with the minimum required sound level requirement increasing both at lower and higher frequencies.

The implications of Absolute Threshold for speech processing algorithms are obvious: by ignoring (or at least reducing the weight given to) frequencies outside this sensitive region, the amount of information needed to represent a given speech can be reduced. On the contrary, more emphasis can be given to accurately representing the speech data within the region in which the human ear is more sensitive. As noted above, this depends on the particular listener, but some advantage may be derived from knowledge of average human auditory capabilities.

Masking

Masking refers to the unique property of the human auditory system due to which a louder sound renders a lower-level sound inaudible as long as both have the same frequency. On the other hand, the same two sounds would be distinctly perceived by the same listener if they happen to have different frequencies. In practice, there is a band of frequency wherein the masking effect occurs, i.e., even if the two frequencies are not exactly the same but differ slightly, the louder sound would still "drown out" the softer sound. As was the case with Absolute Threshold, the Masking curve varies from one person to another depending on the specific characteristics of the Basilar Membrane.

As we will see in later chapters, many speech compression and noise reduction techniques exploit this Masking exhibited by the human auditory system by manipulating the frequency spectrum of (undesired) noise by emphasizing it in those frequency bands where the (desired) signal level is relatively high, and suppressing it in those regions where the signal level is low.

Phase Perception (or Lack Thereof)

In general, it has been observed that a human ear is completely impervious to phase differences between two signals. For instance, the way a tone is perceived by a person would be exactly the same were the tone to be phase shifted; the amplitude and frequency of incoming sound wave is definitely far more significant to the human auditory system.

The lack of phase perception is utilized by many speech processing algorithms, especially in the area of speech compression, by ignoring the information about the phase of speech samples while encoding, transmitting, or storing them. The magnitude or frequency response is what is primarily of interest in most algorithms.

Evaluation of Speech Quality [2]

Measurement or evaluation of the quality of a speech signal is a multifaceted subject, and one that is still evolving. There are several criteria that define speech quality, depending on the specific application of interest and what exactly is being done with the speech signal. For example, the words and meaning of what is being spoken must be intelligible (which, of course, may not be the case if the speech is heavily corrupted by noise).

In applications where speech is compressed (encoded) and subsequently decompressed (decoded) and played back, it is also important that the listener be able to recognize the speaker even while listening to the decoded speech (also known as Synthetic Speech in this scenario). Moreover, the synthetic speech must qualitatively sound like a live speaker, which implies that the speech must be devoid of annoying artifacts like clicking sounds and unwanted reverberation.

In any procedure or technique used to evaluate speech quality, there are several conditions to which the test should be subjected. For instance, the evaluation should be performed across multiple speakers and languages, so as to ensure that the speech processing techniques being evaluated are not only independent of the person speaking but also independent of the specific nuances of individual languages.

The performance of the speech processing system should also be evaluated across the entire valid dynamic range of the signal, i.e., the speech quality should be sufficiently good irrespective of signal amplitudes. Moreover, the system should be tested in the presence of specific types of noise depending on the intended applications.

In communication applications, it is also important to make sure that successive encoding and decoding signal chains (a scenario popularly known as Tandem Coding) do not cause significant degradation, as well as to evaluate the performance of speech communications when subjected to errors or noise in the communication channel. Finally, the response of the system to nonspeech signals such as signaling tones and musical chimes may be of interest in some applications.

Traditionally, all speech quality evaluations were done using objective metrics such as Signal-to-Noise Ratio (SNR) and Segmental Signal-to-Noise Ratio (SSNR).

Signal-to-Noise Ratio

The SNR of the output or synthetic speech signal $y[n]$ relative to the original or desired speech signal $x[n]$ is given by:

$$\mathrm{SNR} = 10\log_{10}\left(\frac{\sum\limits_{n} x[n]^2}{\sum\limits_{n} (x[n]-y[n])^2}\right). \tag{3.17}$$

Segmental Signal-to-Noise Ratio

When frames with high power content are interspersed with frames with low power content, then some speech processing algorithms fail to accurately analyze or synthesize the low power segments. The SNR metric, unfortunately, does not adequately reflect such scenarios, as the high-power frames will dominate the overall result. This problem is alleviated by using the SSNR as a basis for quality evaluation instead of the SNR. In this computation, the SNR is measured over a smaller number of adjacent frames and averaged, resulting in the equation given below:

$$\mathrm{SSNR} = \frac{1}{N}\sum_{m=1}^{N} \mathrm{SNR}_m. \tag{3.18}$$

Mean Opinion Score

Purely objective techniques like SNR and SSNR strive to determine the closeness of the synthetic or output waveform to the original signal. While these are perfectly effective for speech processing techniques that aim to preserve or reproduce the samples of the actual waveform, they are not appropriate in evaluating those speech compression techniques that encode speech in terms of parameters rather than maintaining or recreating the original waveform during synthesis.

A speech coder that produces very good speech from a perceptual standpoint might have a poor SNR score as the synthesized waveform is not expected to match the original one sample-by-sample. In such scenarios, it is far more preferable to employ subjective evaluation metrics. In subjective methods, a large number of listeners are asked to listen to the output speech and provide a rating of perceived quality. Several metrics exist in the subjective category as well.

In one approach, called Absolute Category Rating (ACR), the synthetic speech is rated by itself. The resultant rating, on a scale of 1–5, with 1 signifying “bad” and 5 signifying “excellent,” is known as the Mean Opinion Score or MOS.

In another approach, called Degradation Category Rating (DCR), the listeners listed to both the original and synthesized speech and then rate the synthesized speech in comparison to the original speech; thus, this is a relative measurement and the resultant rating is called Degradation MOS (DMOS). Just as in the case with MOS, DMOS is also rated on a scale of 1–5.

In yet another approach, multiple pairs of synthesized speech may be rated in comparison with each other instead of comparing with the original speech; this technique is called Comparison Category Rating (CCR).

In addition, to simplify the logistics and subjectivity involved in MOS evaluations, many mathematical algorithms have been found that operate on the speech signals and try to automatically model MOS ratings. Such techniques, such as standardized Perceptual Evaluation of Speech Quality (PESQ) algorithm, eliminate the need to involve live listeners and are a promising and versatile tool for evaluating speech quality for a variety of applications.

Summary

In this chapter, we have studied some basic concepts of speech processing that are needed to understand various types of speech processing algorithms and applications. We have looked at the human speech production system and how it can be modeled in a form appropriate for digital systems. We also discussed some fundamental speech processing operations and some speech-specific enhancements to general signal processing operations such as filters. Finally, we also learnt about various schemes for evaluating the qualitative performance of speech compression algorithms. The material in this chapter is intended to provide a solid foundation to help understand not only the algorithms to be described in subsequent chapters but also other speech processing techniques that may be of interest to the reader.

References

1. Rabiner LR, Schafer RW Digital processing of speech signals, Prentice Hall, 1998.
2. Chau WC Speech coding algorithms, Wiley-Interscience, 2003.
3. Holmes J, Holmes W Speech synthesis and recognition, CRC Press, 2001.
4. Proakis JG, Manolakis DG Digital Signal Processing – Principles, Algorithms and Applications, Prentice Hall, 1995.
5. Flanagan JL, Speech Analysis and Perception, Springer-Verlag, 1965.
6. Rubin P, Vatikiotis-Bateson E Measuring and Modeling Speech Production, Animal Acoustic Communication, Springer-Verlag, 1998.

Chapter 4
CPU Architectures for Speech Processing

Abstract In the world of embedded applications such as the ones we discussed in the Introduction, the application system is typically implemented as a combination of software running on some kind of a microprocessor and external hardware. Each microprocessor available in the marketplace is associated with its own structure, functionality, and capabilities, and ultimately it is the capabilities of the microprocessor that determines what functions may or may not be executed by the software. Therefore, understanding the different types of processor architectures, both in terms of Central Processing Unit (CPU) functionality and on-chip peripheral features, is a key component of making the right system design choices for any given application. This chapter focuses on various types of CPU architectures that are commonly used for speech processing applications as well as general control applications that might include some speech processing functionality. There are several architectural features that serve as enablers for efficient execution of the signal processing and speech processing building-block functions we have discussed in the two previous chapters (not to mention the more complex algorithms we are going to explore in the remaining chapters). These architectural features and considerations are discussed in this chapter in some detail. A discussion about on-chip peripherals is left for the next chapter.

The Microprocessor Concept

In the past, the control of any electronic system was accomplished exclusively using discrete circuit elements such as resistors, capacitors, diodes, and transistors. For example, a filtering operation would be implemented using a network of suitable resistors, capacitors, and inductors. Since even simple electronic functionality often required a considerable number of such components, this approach was only suitable for rudimentary tasks. As mentioned in previous chapters, the characteristics of discrete circuit components vary quite a bit across temperature (note that many real-life applications require operation over a wide temperature range) and over the age of the circuit. This results in significant degradation of the system performance over time, particularly in the case of analog circuits.

As integrated circuits (also called microchips or ICs) became ubiquitous, it became possible to integrate many such basic circuits within an enclosed, packaged device. Single-chip implementations are now available for an enormous variety of analog and digital circuits; for example, the same amplifier circuit that required discrete transistors can now be implemented largely using a single chip that contains multiple operational amplifiers. Although the appearance of ICs reduced the physical dimensions of these circuits to some extent, it still does not provide really complex functionality in a small form-factor. Moreover, the temperature drift and aging problems associated with analog circuits are present in their IC counterparts too.

As end-product designers (irrespective of which market segment they were operating in) strived to differentiate their offerings from competitors by providing wider features and greater functionality, a greater level of integration became necessary than was provided by simple ICs. This gave rise to a concept called Application-Specific Integrated Circuit (ASIC). An ASIC is a relatively complex IC that integrates large chunks of circuits required for a particular application within a single chip. By using one or more of these ASICs, an entire system can be implemented with relative few external components.

ASICs are an effective way to incorporate complex functionality into a single chip; as a simplistic example, an entire signal filtering subsystem could be implemented in one IC. As another example at the other end of the complexity spectrum, most of the functions needed by a mobile telephone may be implemented by a couple of ASICs specially designed for that product. Over the years, ASICs have become more and more powerful and complex, providing larger levels of analog and digital circuit functionality (including the ability to perform specific arithmetic computations), albeit specifically designed for a particular application (and in many cases, for a particular customer).

At first glance, this seems to be a viable approach for most digital control or signal processing applications. However, it is the "application-specific" nature of this class of solutions that is its greatest disadvantage: the circuit generally cannot be modified to add user features, create variations of the product depending on the market being served, or even correct bugs found after the product has been released. Any of these changes would require a redesign of the entire circuit, which leads to considerable loss of flexibility and is a costly effort to say the least. Moreover, since such ASICs are produced to serve a narrow application or a single customer, they are usually manufactured in smaller volumes, often leading to higher prices.

The introduction of programmable circuits, particularly microprocessors, led to a paradigm shift in how electronic systems can be controlled and how an end-application is implemented (though ASICs are still common in some application segments). A Microprocessor Unit (MPU) is a large set of logic circuits that can be programmed with computer software to do whatever the user wants it to do (subject to its capabilities and limitations, of course). If a product designer wants to customize certain features to suit different product variants, all that is needed is to modify and reprogram the software.

It should be apparent that this inherent flexibility has enabled microprocessors to propagate into thousands of electronic systems that hitherto used only discrete circuits or ASIC-based solutions. Increasingly, even formerly electro-mechanical systems such as Power Converters and Motor Controllers are utilizing microprocessor-based systems, not due to flexibility, cost, and space considerations but also to utilize many advanced mathematical techniques that have been invented to improve performance, for example, to reduce the power factor of a motor control system, or the power consumption of a UPS unit. The term MPU is generally used for programmable devices used in computers, for example, the Intel Pentium; Embedded Systems (our primary area of interest in this book) typically use a class of microprocessor devices known as Microcontroller Units (MCU).

Microcontroller Units Architecture Overview [3–5, 9, 10]

The architecture of an MCU lends itself well to performing control tasks, mainly due to the fact that an MCU provides a greater level of integration than an MPU. Not only does an MCU contain large amounts of on-chip memory (traditional MPUs require off-chip memory to operate) but also contain a large array of peripheral subsystems within the chip (but more on peripherals in the next chapter). Like a general-purpose MPU, and unlike an ASIC, an MCU is generally not application-specific, though sometimes the peripheral set available may be geared toward certain classes of end-applications. Also, an MCU is almost never designed to be customer-specific: its main selling proposition is the flexibility and programmability it provides.

An MCU architecture can be classified as 8-bit, 16-bit, or 32-bit, depending on the size of data it can operate on. For example, an 8-bit MCU can add two 8-bit numbers in a single instruction, but to add two 16-bit numbers it needs to break down the problem into steps and operate on 8-bit numbers in each stage. Thus, it follows that if the data sizes involved in a particular operation is larger than the native data size supported by an MCU architecture, the operation takes longer to complete (which might not satisfy the real-time constraints of the application).

Often, the data size also has a bearing on the amount of memory that can be addressed by the processor: for example, an MCU with a 16-bit data memory (RAM) addressing logic can only address 65,536 bytes of RAM. In some cases, the address width may be larger than the native data size of the architecture; for example, the PIC24 MCU devices from Microchip Technology have a 24-bit program memory address register (thus allowing the CPU to address up to 4 MB of memory), even though only 16-bit data can be processed.

Besides the computational features (enabled by the presence of an Arithmetic and Logic Unit, or ALU) that make microprocessors so versatile, an MCU architecture provides some additional features that make them particularly effective in embedded control applications, including those that involve speech processing:

- Periodically servicing interrupts, for example, to obtain periodic samples of a speech signal while recording a segment of speech uttered by a user and transferring the data to a buffer area.
- Capturing data from multiple sensors and control inputs, for example, simultaneously measuring the temperature and chemical composition of a liquid in an industrial container in order to trigger a verbal alarm as needed.
- Controlling actuators, for example, sending a PWM signal to open and close an automatic secure door in response to a voice command.
- Manipulating individual I/O pins through bit-manipulation and bit-scanning instructions, for example, activating a switch to turn on a Microwave Oven in response to a spoken command.
- Sharing the data with other controller modules in a distributed system, for example, various subsystems periodically sending status data to a diagnostics module through Ethernet.
- Field programmability of Program Memory, for example, a boot loader utility might automatically upgrade the software in products that have already been deployed in the field.
- Small packages with low pin-counts, which is absolutely critical for space-constrained applications, for example, a hearing-aid.

Figure 4.1 illustrates the on-chip feature set of an example MCU, the 16-bit PIC24 MCU from Microchip Technology Inc.

What a traditional MCU is generally not optimized for is the ability to perform repeated fast mathematical operations, such as computing the energy of a signal in a given time interval. This is particularly true for the average 8-bit MCU, but may

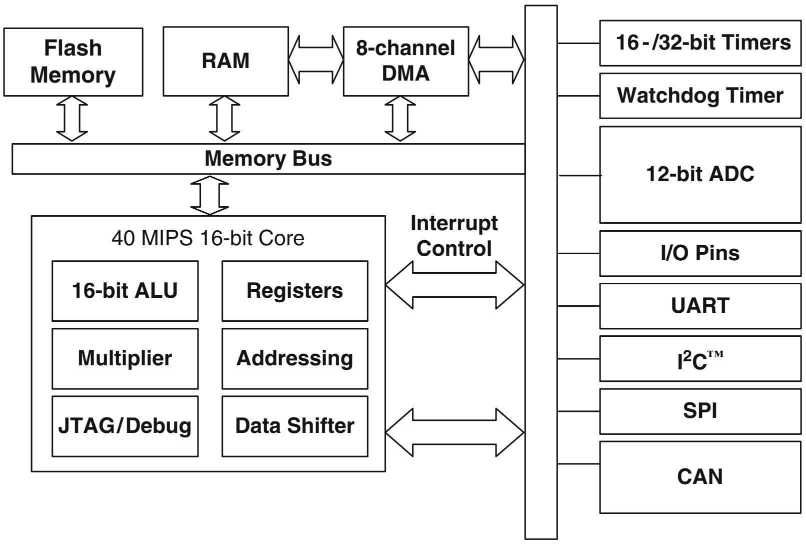


Fig. 4.1 Example 16-bit MCU Feature Set – PIC24 MCU from microchip technology

not be as much of a limitation on some 32-bit MCU architectures as the larger data size can sometimes compensate for the lack of dedicated mathematical hardware. However, in some cost-critical applications the higher cost of a 32-bit processor may not be acceptable.

The need to perform fast mathematics in tightly repeated blocks of instructions (loops), which is fairly typical of applications that involve analysis or manipulation of real-world signals in real-time (i.e., signal processing) led to the introduction of a specialized class of embedded processors called Digital Signal Processors (DSP). As we will see in the context of speech processing applications, DSP devices are particularly suited for speech processing tasks such as filtering, FFT, and compression.

Digital Signal Processor Architecture Overview [3–5, 7, 8]

In the previous two chapters, we have already learnt that many mathematical operations constitute the building block of any application that involves signal processing, especially speech processing. FFT computations, FIR and IIR filtering, data normalization and scaling, formant analysis and adaptive quantization – the list of signal/speech processing tasks that are mathematics-intensive is indeed extensive. Of course, this is not to say that other embedded control algorithms are never mathematically costly; indeed, high-speed PID controllers and motor control algorithms also benefit greatly from the math-optimized architecture of DSP devices. We will explore some of these features in greater detail in the following sections, but a brief overview of typical DSP architectural features are presented below.

Probably the most powerful feature of a DSP is the availability of extensive mathematical processing capability. Many DSP architectures include two 40-bit accumulators, which can be used to store the results of two independent 16-bit $\times$ 16-bit multiplication operations. Most signal processing algorithms involve the calculation of a running "sum-of-products." Special instructions such as MAC (Multiply-and-Accumulate) provide the ability to multiply two 16-bit numbers, add the result to the accumulator, and prefetch a pair of data values from RAM, all in a single instruction cycle. With two accumulators, it is also possible to simultaneously write back the data in one accumulator while computations are being performed on the other one.

An accumulator width of 40 bits rather than 32 enables the data to temporarily "overflow" (e.g., when accumulating a large number of values in the accumulator). Moreover, a DSP can optionally keep the value within the permitted range by a mechanism called Saturation, and also round or scale the data when it is written back to RAM. An additional characteristic of a DSP (that is generally absent in traditional MCUs) is the ability to perform fractional arithmetic.

Add to the above features a wide variety of data-addressing modes for efficiently moving data around, support for circular buffers and bit-reversed addressing, and zero-overhead loops, and it becomes obvious that a DSP provides a very potent yet user-friendly CPU architecture for speech processing applications.

Digital Signal Controller Architecture Overview [3–5]

In spite of all the powerful features listed above, it is noteworthy that a traditional DSP operating by itself (without an associated MCU handling most of the control tasks) is not very suitable for use in dynamic environments that are characteristic of most real-time embedded control applications, for four main reasons:

- DSPs do not have flexible interrupt structures
- DSPs are not very efficient at manipulating bits
- DSPs rely largely on off-chip memory and peripherals
- DSPs are rarely available in low pin counts and are thus not suitable for space-constrained modules

These drawbacks are alleviated by a new and increasingly popular class of embedded processors called Digital Signal Controllers (DSCs). As the name suggests, a typical DSC architecture incorporates the best features of a DSP while eliminating the common drawbacks of DSP architectures. Indeed, DSC architectures (especially those that evolved from MCU architectures to begin with) not only contain all the beneficial aspects of MCU devices, but also contain a strong set of the typical DSP features that were listed in the previous section. For example, a DSC can compute an FIR filter in a tight loop, while simultaneously transferring data from communication peripherals to RAM using a zero-overhead DMA channel. It can efficiently process real-time interrupts without distorting the results of mathematical operations, and also manipulate I/O pins to perform real-time control tasks in an embedded control environment. Last but not the least, a typical DSC is significantly less expensive than a traditional DSP and is available in much smaller packages.

If all that the processor needs to do is perform a dedicated intensive signal processing task (which is rare in cost-sensitive and space-constrained and low-power embedded systems), a pure DSP should suffice. However, due to the reasons listed above, a DSC is the ideal processor architecture when the application system needs to perform not just signal processing operations but also other embedded control tasks including communications and user interfaces. The high-level components of a typical DSC device are depicted in Fig. 4.2.

Now that we have seen some key differences between Microprocessors, Microcontrollers, DSPs, and DSCs, as well as architecture selection considerations for embedded applications that include speech processing functionality, let us explore some of the aforementioned typical DSP/DSC processor features in more detail. But before that, let us look at the most common data formats supported by such architectures.

Fixed-Point and Floating-Point Processors [1, 2, 6]

Most signal processing, and certainly all speech processing algorithms, use a fractional number representation for all speech samples, filter coefficients, FFT outputs, etc. Fractional numbers can be represented in a number of different formats; in this

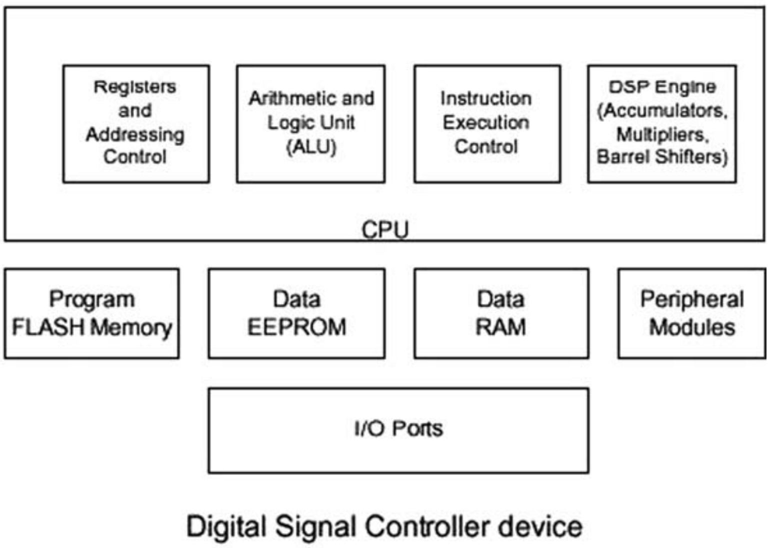

Registers and Addressing Control
Arithmetic and Logic Unit (ALU)
Instruction Execution Control
DSP Engine (Accumulators, Multipliers, Barrel Shifters)
CPU
Program FLASH Memory
Data EEPROM
Data RAM
Peripheral Modules
I/O Ports
Digital Signal Controller device

Fig. 4.2 High-level components of a typical DSC device

section, we will look at the two most common formats: fixed-point and floating-point. Some DSP architectures operate on floating-point data (or at least provided some kind of a coprocessor that enables floating-point arithmetic), whereas others operate on fixed-point data. At the time of this writing, all DSC architectures use fixed-point arithmetic exclusively. It may be noted that most microcontrollers do not support either format, and rely natively on integer arithmetic with appropriate software to handle fractional numbers.

In fixed-point representation, a fractional number (say, a 16-bit one) is assumed to have a certain number of bits representing the integer portion of the number and the remaining bits represent the fractional part. In other words, for any specific fixed-point format there is an implied radix point. A common nomenclature used to specify the number of integer and fractional bits is the x.y nomenclature. For example, a 1.15 format implies that there is only one integer bit (and that is the sign bit, since you must have at least one bit to indicate whether the number is positive or negative), with a dynamic range in the $+1.0$ to -1.0 range. This is also referred to as the Q15 format. As we will see in the next chapter, it is all a question of how speech sample values are numerically interpreted relative to the analog dynamic range provided by ADCs, DACs, and other peripherals. Another common format is the 16.16 or IQ16 format implies that there are 16 integer bits (of which one is the sign bit) and 16 fractional bits, providing a numerical dynamic range of approximately $+327,680$ to $-327,680$. Bear in mind that many industry-standard Speech Processing algorithms assume a 16-bit (1.15) representation of the input data, hence 16-bit fixed-point arithmetic is the most popular choice for embedded speech processing tasks.

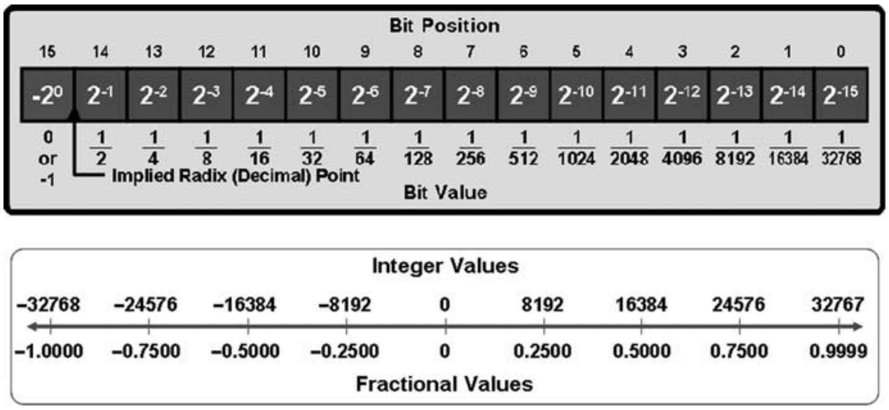


Fig. 4.3 Numerical representation of bits in 1.15 Fixed-Point Format

Figure 4.3 shows the numerical interpretation of individual bits in the 1.15 format, as well as its equivalence with the 16-bit integer representation. Note that DSC architectures support both fractional and integer data formats and arithmetic, which may not be the case with traditional DSP architectures. This is very useful in applications that involve both speech processing and control tasks.

The Floating-Point format does not reserve bits for the integer portion and fractional portion of the number. Instead, it uses a $(\mathrm{X} \times 2^{\mathrm{Y}})$ representation of the number, and reserves certain number of bits for the quantities X and Y, respectively. Here, the number X (which usually ranges between $+1$ and -1) is called the Mantissa or Coefficient and the number Y is called the Exponent. This is analogous to the Scientific Number representation in decimal arithmetic, and is standardized by the IEEE 754–2,008 Binary Floating-Point Standard, and readers are advised to refer to this standard for specific details about the floating-point format.

DSP devices with Floating-Point representation tend to be more expensive and therefore less popular for cost-sensitive embedded applications, hence the subsequent discussions in this book will primarily assume fixed-point data. However, this number format does have the advantage that the dynamic range is effectively much larger even for the same number of bits of data. Moreover, due to this increased dynamic range the possibility of numerical instability and data overflow conditions is negligible.

Accumulators and MAC Operations [2,3]

Some of the most common signal processing operations used in the embedded applications are digital filters, for example, FIR Filter. In Chap. 2, we have seen that the relationship between the input and output of an FIR Filter is given by:

$$y[n] = \sum_{k=0}^{T-1} b_k x[n-k], \tag{4.1}$$

where T is the Filter Order (number of filter coefficients or "taps"), b_k is the kth coefficient of the FIR Filter.

From the equation, it becomes clear that the FIR filter computation involves multiplying each filter coefficient by the corresponding data sample from the delay line, and then adding all these products together. This kind of sum-of-products operation (also seen in operations such as auto-correlation, cross-correlation, and convolution) is essentially a Vector Dot Product, and lies at the heart of numerous signal processing and speech processing algorithms.

Similarly, recall the relationship between the input and output of an IIR Filter:

$$y[n] = \sum_{k=1}^{p} a_k y[n-k] + \sum_{k=0}^{p} b_k x[n-k], \tag{4.2}$$

where P is the Filter Order (number of filter coefficients or "taps"), a_k is the kth feedback coefficient of the IIR Filter, and b_k is the kth feed-forward coefficient of the IIR Filter.

In this case, there are two sum-of-products computations involved, which makes it even more critical to be able to perform such computations as fast as possible. The instruction sets of general-purpose processors or MCUs generally require at least one instruction to compute a multiplication, and then another instruction or more to add the product to a running sum. The final value of the running sum is the required sum-of-products. To eliminate the redundant additions, DSP/DSC architectures usually support the execution of the multiplication and the addition in a single instruction-cycle. In other words, a single instruction is all it takes to multiply two numbers and then "accumulate" the product into a running sum, an operation widely known as Multiply-Accumulate or simply MAC. If the inputs and coefficients are 16-bit fractional numbers (which is usually the case for fixed-point architectures), each resultant product is a 32-bit fractional number. The running sum can well exceed the bounds of a 32-bit Accumulator register, hence the needs arises to provide some extra bits in the Accumulator to allow for greater dynamic range during the computation.

Because of the accumulation of a large number of products, it is very likely that the intermediate results are numerically too large to be correctly stored in the 40-bit Accumulator (and this is even more likely to occur if the architecture only provides 32-bit Accumulators). This is a condition known as Accumulator Overflow. IIR Filters, and any other signal processing and control operations that involve feedback terms, are particularly vulnerable to such numerical instability.

Normally, an Overflow condition causes the data to be corrupted. Many DSP/DSC architectures provide some kind of event flag bit or exception to detect such scenarios, but there is not much the application software can do to correct the situation, other than rolling back the most recent computation and performing some manipulation or replacement of the input data. A better approach to alleviating the effect of numerical overflows is provided by a feature called Accumulator Saturation, which is common on fixed-point architectures. Saturation basically implies that the CPU hardware logic automatically detects the occurrence of an overflow. The logic then deduces (usually based on the state of the sign bit) what the correct sign of the result should be, and replaces the result with the maximum possible positive or negative 32-bit fractional number (depending on the expected sign bit) that can be stored in the Accumulator. In the absence of Saturation, the resultant value in the Accumulator would have completely got corrupted, whereas when Saturation is used the "modified" value is much closer to the expected result.

To further clarify the concept of saturation, one can draw an analogy with decimal numbers: if 99 is added to 5 in an "accumulator" that only has two digits, the saturated value of $+99$ is very close to the expected result of $+104$; in the absence of saturation the result would simply have been the wildly inaccurate value of $+04$ (which is $+104$ with the upper digit discarded).

Some processor architectures (e.g., the dsPIC30F and dsPIC3F DSC families from Microchip Technology) provide multiple saturation modes: one can saturate the result to the maximum possible 32-bit fractional value (providing a dynamic range of approximately $+1.0$ to -1.0), to the maximum 40-bit fractional value (in this configuration the dynamic range is increased to $+256.0$ to -256.0), or choose to saturate the data that gets written back to memory, for example, at the end of the sum-of-products operation.

Another popular architectural feature provided by DSP and DSC devices is automatic rounding of data when the Accumulator contents get written back into memory or another working register.

Sum-of-product operations are by no means the only signal processing tasks that benefit from MAC operations. The FFT butterfly computation also utilizes MAC instructions, and so does the coefficient update procedure in adaptive filters. In fact, the term MAC is often used in a wider sense to refer to the whole set of Accumulator-based multiplication-oriented operations natively supported by a given DSP or DSC architecture. For example, the required operation may be a simple multiply-and-store rather than a multiply-and-accumulate, with the product always overwriting the current Accumulator contents; alternatively, the product may be subtracted from the Accumulator rather than added to it. Some architectures also provide some specialized MAC instructions for important operations such as computing the Euclidean Distance between two vectors (a critical function in most Pattern Recognition applications as well as some Speech Compression algorithms).

Besides the fact that a multiplication and an addition is performed by the same instruction of code (and usually within the same instruction clock cycle), the real power of the MAC instruction provided by many DSP and DSC devices is that the next pair of input operands (e.g., the next coefficient and data sample value in the case of an FIR filter) can be prefetched in preparation for the next MAC operation. This results in significant computational savings in most signal processing routines, as it completely eliminates the need to separately fetch two input words from memory using two successive data transfer (move or load) instructions. Moreover,

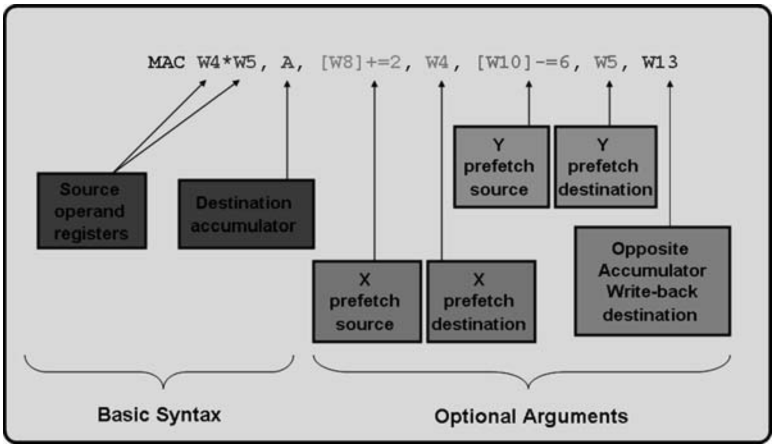


Fig. 4.4 MAC instruction syntax example – dsPIC30F/dsPIC33F DSC architecture

since MAC operations are often executed as part of a loop, many DSP/DSC MAC instructions can also postmodify the values of the pointers used to prefetch the data, thereby automatically pointing to the next pair of operands to be fetched and further reducing the number of instruction cycles needed to execute the overall loop.

Finally, it is noteworthy that some architectures that have two Accumulators allow the other Accumulator value (the one not used in the current instruction) to be written back into either a temporary working register to be used again or to a RAM address, which is typically an array location. Writing back to a working register is useful in algorithms wherein the value written back needs to be used again very soon, such as the intermediate results in an FFT butterfly computation. Writing back the Accumulator to an array location is useful when the value written back will not be used again the current processing loop, for example, writing back an updated filter coefficient in an adaptive filtering algorithm.

As an example of the various capabilities provided by MAC operations in a DSP/DSC architecture, Fig. 4.4 presents the MAC instruction syntax in an example DSC architecture: the dsPIC30F/dsPIC33F family.

Multiplication, Division, and 32-Bit Operations [3, 4]

We have seen in the previous section that a Multiply-Accumulate operation is one of the most widely used arithmetic operations in the signal processing world. However, in many cases only a multiplication is required. In embedded control applications, different functions of the application (only some of which might pertain to speech processing) might require different types of multiplication operations, including signed, unsigned, mixed-sign, integer, and fractional. Most traditional DSP architectures only support signed fractional multiplications; however, DSC architectures typically support a variety of multiplication variants, including support for different data sizes, for example, 16-bit × 16-bit multiply and 8-bit × 8-bit multiply. Some of these operations might use the Accumulator whereas others might not.

DSC architectures also support a variety of division operations, for example, 32-bit/16-bit integer divide, 16-bit/8-bit integer divide, 16-bit/16-bit signed fractional divide, etc. Again, since embedded control applications are multifaceted and different portions of the application's functionality might need different data formats, the variety offered by DSC devices are more suitable for embedded speech processing.

Some 16-bit DSC architectures, specifically those that have more than one Accumulators, also support some basic 32-bit (or 40-bit) arithmetic operations such adding and subtracting two 32-bit or 40-bit numbers with each other.

Program Flow Control [3, 4]

Signal processing operations often occur as part of a large-count loop. For example, a 256-point FIR Filter would involve the same Multiply-Accumulate operation to be repeated 256 times. Moreover, signal samples are generally processed in blocks or frames (especially as far as speech processing is concerned, where block processing is essential for most operations), which implies that multiple filter outputs must be calculated, thereby adding another loop outside the core filter loop. Therefore, a key thrust of DSP and DSC architectures is to make execution and management of such loops (including nested loops) more efficient. One mechanism for efficient loop handling is automatic hardware support for loop management. A single loop instruction that specifies the loop size and loop count, when working in conjunction with internal registers that keep track of the number of completed iterations and current nesting level, can completely eliminate all the software overhead that would have been needed in defining a loop counter, decrementing it after every iteration and then checking to see if the loop is complete. As the filter order and block size increase, the loop management overhead with dedicated Hardware Loop Control instructions is practically zero. An example of an FIR Filter loop implemented using a hardware loop control instruction (DO instruction) extracted from Microchip Technology's DSP Library is shown below:

```
; Perform filtering of all samples.
do    #255, _endFilter

; Move next data sample to delay line.
mov    [w2++], [w10]

; Clear Accumulator A
clr    a, [w8]+=2, w5, [w10]+=2, w6
```

```
; Filter each sample.
; (Perform all but two last MACs.)

repeat w4
mac    w5*w6, a, [w8]+ = 2, w5, [w10]+ = 2, w6

; (Perform second last MAC.)
mac    w5*w6, a, [w8]+ = 2, w5, [w10], w6

; (Perform last MAC.)
mac    w5*w6, a

_endFilter:
; Save filtered result.
sac.r a, [w1++]
```

The above example is intended to demonstrate how a judicious combination of different loop control instructions can be used in a nested fashion to efficiently accomplish signal processing tasks.

Special Addressing Modes [3, 4]

Many DSP architectures, and some DSC architectures, provide some special addressing modes that allow efficient data buffer management in a wide variety of signal processing algorithms. The most common ones are Modulo Addressing and Bit-Reversed Addressing.

Modulo Addressing

To fully appreciate the utility of modulo addressing, we should first understand the concept of a Circular Buffer. A Circular Buffer is basically a region in memory which is used to store a fixed-size data array, such that after reaching the end of the array one needs to "wrap around" and continue accessing data from the beginning of the array. The data access is thus required to be always bounded within the limits of the buffer. A common use of Circular Buffers is to access the delay line in an FIR filter computation, since the most recent data sample must always overwrite the oldest sample currently stored in the delay line.

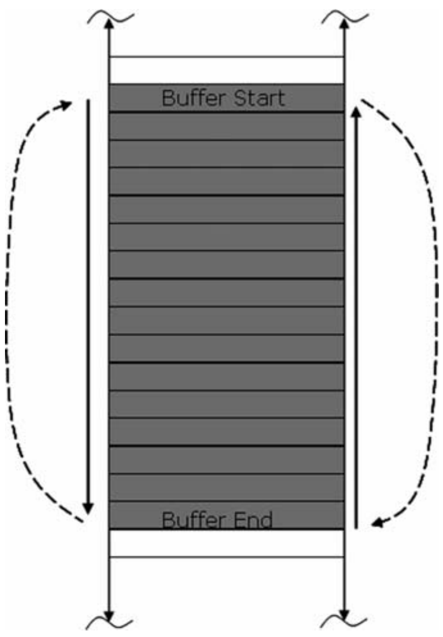


Fig. 4.5 Structure of a circular buffer, showing automatic address correction

In-built Modulo Addressing support in many DSP and DSC architectures makes it unnecessary for the user software to manually monitor the buffer pointer and correct the pointer if it is crossing the buffer limits. When an access is made to an address outside the modulo buffer limits, the CPU automatically performs an address "correction" such that the Effective Address generated lies within the buffer. If the pointer was premodified or postmodified, then the working register used as the pointer now contains the modulo-corrected address, thereby ensuring that pointer increments and decrements always happen in a modulo sense (Fig. 4.5).

Modulo Addressing and Circular Buffers is not only beneficial to speed up signal processing tasks. Communication data buffers (e.g., characters queued up to be encoded and transmitted using a wireless communication protocol such as IrDA) and display buffers (e.g., a large string of characters queued up to be exhibited as a rolling display on a finite set of LEDs), integral to so many embedded control applications, can also utilize Modulo Addressing to implement circular buffers as needed.

Bit-Reversed Addressing

Bit-reversed addressing is a special feature provided in some DSP and DSC architectures to support efficient implementation of FFT algorithms. This is perhaps the sole utility of this special addressing mode.

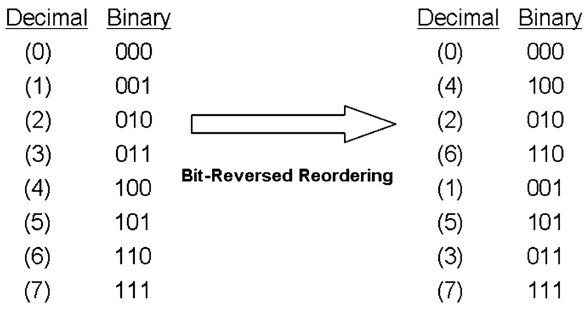


Fig. 4.6 Relationship between sequential and bit-reversed addresses

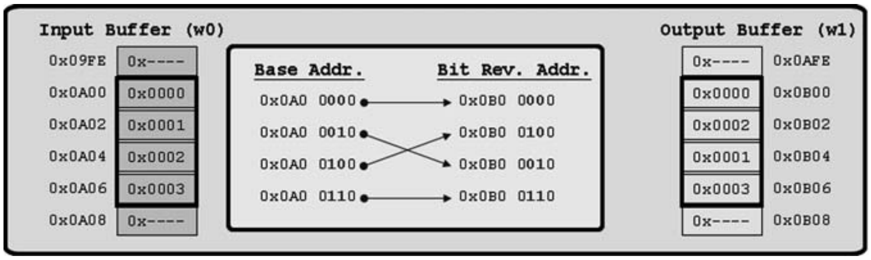


Fig. 4.7 Automatic hardware generation of the next address in bit-reversed sequence

As we have seen in Chap. 2, a Radix-2 FFT algorithm implicitly rearranges the data array being processed. As can be seen from Fig. 4.6, by reading the binary representation of each sequential array index in reverse order, one can determine the corresponding bit-reversed array index. To compensate for this reordering of data that is an intrinsic property of the FFT algorithm, bit-reversed reordering of data is typically done either at the beginning or at the end of a Radix-2 FFT algorithm, so that in the end we obtain the data arranged in sequential order.

Bit-Reversed Addressing is a special addressing mode wherein (given the address of a particular element in the array) the CPU hardware automatically computes the address of the next element in the bit-reversed sequence rather than the next address in a linear/sequential sense.

This bit-reversed reordering can be performed by copying data words from a sequentially addressed array into a bit-reverse addressed array. DSP libraries from DSP and 3rd-party software vendors contain easy-to-use and optimized functions for reordering FFT data using bit-reversed addressing. In some cases, the bit-reversed reordering is performed within the FFT function itself.

Figure 4.7 shows how successive bit-reversed addresses are generated by the hardware, from the current bit-reversed address pointer value and the specified bit-reversed address modifier. The sequential (input) buffer in the example can be reordered, by reading the input buffer elements in sequential order and writing them to the output buffer using bit-reversed addressing. This reordering can also be done "in-place," that is, the same buffer is both read in sequential order by one W register and written in bit-reversed order by another W register. This helps to conserve RAM space.

Data Scaling, Normalization, and Bit Manipulation Support [2–4]

Another important feature of DSP architectures is the ability to shift or scale data by multiple bits (typically by up to 16 bits left or right) in a single instruction cycle. This is a key enabler for a variety of scaling operations needed in signal processing algorithms, for example, scaling an ADC input to align with the data width requirements of a standardized speech compression algorithm, scaling down by a certain number of bits as a fast method of averaging (assuming the number of data points is a power of two, of course), etc. Often, the Accumulator is wider than the data bus width; hence 16-bit shifts need to be utilized to properly align data loaded from memory into the Accumulator. The CPU hardware logic that performs these multibit shift and rotate operations is commonly termed as the Barrel Shifter.

An important point to be aware of is that there are two types of right shifts: Logical and Arithmetic. In a Logical Right Shift, the data are simply treated as a collection of bits, and during the process of shifting the least significant bit (bit 0) is transferred to the Carry bit. In an Arithmetic Right Shift, the data are treated as an arithmetic entity, hence the Carry bit is not part of the data-shift chain: the least significant bit is lost and the Carry bit is unaltered.

Perhaps the most important scaling operation in signal processing is Data Normalization. Normalization is the process of scaling a quantized signal sample so that it can utilize the entire dynamic range available to it. In many embedded signal processing applications, the data may be of very low amplitude, which would mean that only a few bits of the fractional data representation is used. As the data are processed (e.g., convolved with another signal), the data become even smaller and even less bits are used; this can ultimately lead to the loss of resolution and accuracy in the computations. To alleviate this situation, a block of data is first analyzed to determine by how many bits (maximum) every data sample in the block can be shifted up while ensuring that no sample gets corrupted.

The actual data shift can be performed using the Barrel Shifter instructions provided by DSP/DSC architectures within a hardware loop for extra speed. However, before the actual shift is performed, the data need to be analyzed, essentially to find the minimum number of leading bits that are unused by the samples in the block. Some of the architectures such as the 16-bit dsPIC30F/dsPIC33F DSCs provide special instructions for this purpose. One such instruction is the FBCL (Find First Bit Change from Left), which counts the number of bits from the left after which the bit changes (relative to the sign bit). After this operation has been executed on the entire block, the minimum value produced by this operation is the amount of shift needed. Thus, Data Normalization can be performed very efficiently due to the presence of certain instructions optimized for this purpose.

Last but not the least, there are several types of operations that are typically supported by MCU architectures but not by traditional DSP architectures. One notable example is bit operations. As mentioned earlier, DSP architectures are very weak in their ability to quickly manipulate and inspect the state of bits, a vital requirement for any embedded control application. Hence, applications that involve frequent parametric and I/O pin control in addition to pure signal processing routines would benefit more from using DSC architectures, since DSC devices (such as MCUs) are extremely efficient at handling I/O Port bits and other such bit manipulation operations.

Other Architectural Considerations [5, 6]

We have primarily discussed architectural features that are found to enable the efficient execution of signal processing algorithms, as well as control-oriented features that enable effective usage of the device in embedded control systems. Although these are certainly vital features whose presence should be considered for speech processing applications, there are several other factors that should be understood in the context of CPU architecture performance and memory usage. A couple of these considerations are described here.

Pipelining

To enable faster execution of instructions on an average, most high-speed processor architectures use some kind of a pipeline structure in its instruction flow. A pipeline is a mechanism by which the processor's hardware resources are shared between multiple instructions. A pipeline contains multiple stages, ranging from as low as 2 to as many as 10, depending on the speed and complexity of the device.

Some pipelines are very simple: an instruction is fetched in one instruction cycle and executed in the next instruction cycle, for example, the dsPIC30F/33F DSC and PIC24F/PIC24H MCU families from Microchip. Of course, within each of the pipeline stages different actions may be occurring at different times within the instruction cycle: for example, a register or RAM read might occur in the first half and a write might happen at the very end of the instruction cycle. The key here is that while one instruction is being executed, the next one can be prefetched. This results in an Instruction Latency of only 2 instruction cycles. This means that from the time the instruction prefetch begins, it takes 2 instruction cycles for the results of the instruction to be available, that is, any register/memory writes, reads, and arithmetic calculations are complete and all status flags have been updated. While the latency is minimal, the CPU throughput is not necessarily optimal, as only 2 instructions are utilizing the resources of the CPU at any given time. Although this may not lead to very high maximum processor speeds, it is very effective for embedded control applications. This is because if there is a branch or some other program flow change (e.g., a function call), the pipeline must be flushed and the prefetched instruction discarded, as the destination of the branch may not be known early enough. The deeper the pipeline, the greater the impact of a program flow change.

More deeply pipelined architectures have several stages. For instance, 32-bit MIPS-based architectures such as the PIC32 MCU from Microchip feature a 5-stage pipeline: Instruction Fetch, Instruction Decode/Register Fetch, Execute/Address Calculation, Memory Access, and Write Back. This results in high speeds and CPU throughput, since as many as 5 instructions may be sharing and alternating between the processor resources. However, the latency of each instruction by itself is now 5 instruction cycles rather than 2.

There are special methods by which deeply-pipelined processors alleviate the impact of program flow changes. For example, branch destinations may be "predicted" based on a general profile of how many branches are taken and how many are not taken in typical applications. If the application is highly branch-oriented or involves a lot of function calls and interrupts, it is critical to analyze the efficacy of the techniques used to avert such pipeline flush situations.

Also, there may be conflicts between successive instructions that may be accessing the same resource such as a common register. For example, it is possible that the previous instruction has not yet reached the stage where it writes to the register and a subsequent instruction needs to read the updated register value (a scenario known as Read-after-Write condition). Conversely, an instruction may have written to a register but the previous instruction has not read the register yet, resulting in the previous instruction inadvertently getting a modified value (this situation is known as Write-after-Read). Such situations are known as Pipeline Hazards, and constitute another disadvantage of deep pipelines (or any pipelines for that matter, deeper ones are simply more susceptible to them). Pipeline Hazards are generally alleviated by methods such as Forwarding, wherein the CPU hardware detects such scenarios and transfers the correct values to the instructions that need them at the appropriate times, thereby avoiding such scenarios altogether but adding complexity to the instruction sequencing hardware logic and possibly raising the cost of the processor.

Memory Caches

A popular feature, especially in 32-bit architectures, is memory caches. The concept of caches has mostly been inherited from the general-purpose processor world and been adapted to embedded processors specially for memory-intensive applications. A cache essentially involves a temporary memory storage area (for either data or instructions) that can be accessed very fast compared with the rest of the memory.

For example, if a block of data memory is being accessed frequently, the processor might fetch that block of memory into the cache. For any data memory access, it first checks if it is present in the cache; if it is, it is simply read from the cache, thereby reducing the memory read latency. Obviously, this would work more efficiently if the same block of memory is used repeatedly, because every time a memory address is needed that is not in the cache, an additional latency is incurred in fetching it and populating it in the cache. Moreover, there are data consistency issues that the architecture typically needs to handle using extra hardware logic.

In embedded controllers, it is often of greater utility to store program memory instructions in the cache rather than data. This is especially beneficial if certain sections of code are used repeatedly, such as in a tight DSP routine such as a filter, or peripheral driver functions that are executed repeatedly and frequently. If the application involves a less deterministic pattern of branches and function calls, it may be more efficient to avoid using caches. Needless to say, the regular (noncached) program memory access times for the processor of interest must be considered for making these decisions.

Floating Point Support

Most 8-bit and 16-bit embedded processors, including DSPs and MCUs, feature exclusively fixed-point Arithmetic and Logic Units (ALU): there is no floating-point support whatsoever and any floating-point operations must be handled purely through software routines without any architectural support except perhaps a generally efficient instruction set. Indeed, many embedded applications (including most standards-based Speech Processing algorithms) do not even require the high dynamic range and data resolution provided by floating-point processors. In fact, the larger memory requirement of floating-point data and variables may not be tolerable in some cost-conscious applications using low-memory devices. However, in applications in which floating-point arithmetic is needed, having floating-point architectural support is definitely beneficial for application performance.

Many 32-bit processors do provide support for floating-point arithmetic routines. Some 32-bit DSPs are inherently floating-point architectures: in such processors all numbers and DSP instructions use floating-point arithmetic almost exclusively. In 32-bit MCUs such as the MIPS32 architecture, on the other hand, the core architecture is integer-based and DSP instructions use fixed-point computations, but floating-point support is also provided through the presence of on-chip Floating Point Coprocessors that help execute special floating-point instructions.

Exception Processing

In application areas such as embedded speech processing that involve large amounts of speech data sampling and playback and other forms of peripheral handling, the latency of interrupt processing is a critical criterion in selecting processor architecture. Besides short interrupt processing latency, some real-time mission-critical applications (especially those that operate in a Real Time Operating System framework) require deterministic interrupt latency: entering an interrupt handler must always take the same time.

Summary

I hope this chapter has provided the readers with an overall understanding of the various key features of DSPs and DSCs that make these architectures particularly useful for speech processing applications. In addition, DSCs have a number of powerful features inherited from MCU architectural trends that enable them to be used very effectively in embedded control applications as well. Since one of our primary interests in this book is in incorporating speech processing functionality into general-purpose embedded control applications, DSC architectures provide a good balance between the best features of DSPs and MCUs. However, in some applications, especially those that require large memory usage (not always necessitated by speech processing requirements) that can only be enabled by 32-bit addressing, a 32-bit MCU may be a more appropriate choice. So far we have only explored the CPU-oriented features and architectural considerations related to embedded speech processing. In the next chapter, we will learn what on-chip peripherals are useful in developing a complete application solution for an embedded speech processing application.

References

1. JL Hennessy, DA Patterson Computer Architecture – A Quantitative Approach, Morgan Kaufmann, 2007.
2. JG Proakis, DG Manolakis Digital Signal Processing – Principles, Algorithms and Applications, Prentice Hall, 1995.
3. Microchip Technology Inc dsPIC30F/33F Programmer's Reference Manual.
4. Microchip Technology Inc dsPIC33F Family Reference Manual.
5. P Sinha (2005) DSC is an SoC Innovation, Electronic Engineering Times, July 2005, pages 51–52.
6. D Sweetman See MIPS Run, Morgan Kaufmann, 1999.
7. www.ti.com – website of Texas Instruments.
8. www.analog.com – website of Analog Devices.
9. www.freescale.com – website of Freescale Semiconductor.
10. www.arm.com – website of ARM Semiconductor.

Chapter 5
Peripherals for Speech Processing

Abstract In the previous chapters, we have seen the importance of CPU architecture in enabling efficient and reliable implementation of speech processing applications. Although signal processing algorithms are executed by the CPU, and indeed some of these algorithms are complex enough that they can be greatly benefited by powerful and flexible architectural features, these are necessary but not sufficient for implementing a complete system. Sampling and conversion of real-world analog signals such as speech recordings, communicating control data to other components of the system, and accessing memory to store and access required speech samples and algorithmic parameters, are all things that are needed as part of a whole application solution. This necessitates the presence of a robust set of peripheral modules for data conversion, communications, and storage. The processor-side interface to these peripherals also needs to be efficient to utilize them effectively. Therefore, the availability of suitable peripherals on-chip is one of the vital criteria for selection of a processor platform for any embedded application, including those that include speech processing functionality. In this chapter, we will investigate the various peripheral features that are necessary or useful for implementing embedded speech processing applications.

Speech Sampling Using Analog-to-Digital Converters [2, 3, 8–10]

In any embedded speech processing application, the most fundamental operation that usually needs to be performed is to sample speech signals and convert them into digital form. Indeed, besides speech signals the overall application might often also need to capture various real-world physical or electrical quantities such as temperature, pressure, potentiometer settings, etc. that are invariably analog in nature. While all such signals require analog-to-digital conversion, we will limit our discussion here mostly on the analog-to-digital conversion requirements for speech signals, with an understanding that the speech may be only one of the multiple analog signals being converted in the system.

An Analog-to-Digital Converter (ADC) refers to any mixed-signal electronic device or circuit that can convert one or more analog signals and convert them into a digital binary value. As discussed in Chap. 2, this process essentially quantizes the signal into discrete steps, and represents the sample as a multibit binary representation of the specific step assigned to the signal at any given instant. To reduce system cost and to address space constraints that are common in embedded applications, it is often preferable that the ADC be a peripheral module embedded within the processing device (DSP, DSC, or MCU) itself rather than an external stand-alone ADC. In fact, most embedded processors come equipped with some sort of on-chip ADC module (or in some cases, multiple such modules). These devices have multiple pins that are designated as analog inputs, though these are often multiplexed with digital I/O or other peripheral functions.

In an actual system, the analog signal of interest is fed into one of these analog input pins, and the ADC module converts the signal and stores the conversion result in a register. In many devices, multiple ADC results can be buffered in an entire block of registers or even directly in RAM, thereby reducing the need for the processor to read every individual result as soon as it is available. Indeed, efficient buffering or transfer of ADC results is one of the key selection criteria for any ADC module used in signal processing applications. Figure 5.1 shows an ADC module present in an example 16-bit DSC device family: the dsPIC33F. This example is shown to give readers a general idea of the common parts of an ADC module; some of these components will become clearer as we discuss ADCs further.

Types of ADC

The first item to look at when selecting a processor device with a suitable ADC is the type of ADC hardware that is present. ADCs come in various flavors, a small subset of which are listed below:

- Flash ADC: These are a bank of comparator devices whose outputs feed into a digital logic circuit that converts the outputs to a binary value that can be stored in a register. Each comparator directly corresponds to a specific quantization step, hence to get an N-bit result the number of such comparators needed is 2^N. It is a very fast method of signal conversion, but because of the large number of comparators needed, the resolution of this class of ADCs is rarely more than 8 bits. Hence, it is not as useful for speech processing, which generally requires a larger number of bits to represent the speech samples and transitions accurately.
- Successive Approximation Register (SAR) ADC: These ADCs use a comparator (or a Sample/Hold Amplifier) to iteratively compare an estimate of the voltage with the actual analog voltage input. Thus, it inherently includes a Digital-to-Analog Converter (DAC) in a feedback path, and keeps adjusting the DAC voltage until a very close match is reached. Because of the feedback process, the conversion time tends to be slower and often not suitable for extremely fast measurements, but they are generally sufficient for speech processing purposes. Most speech processing algorithms operate at relatively lower sampling rates such as 8 kHz (also denoted as kilo-samples per second in the context of ADC speeds),

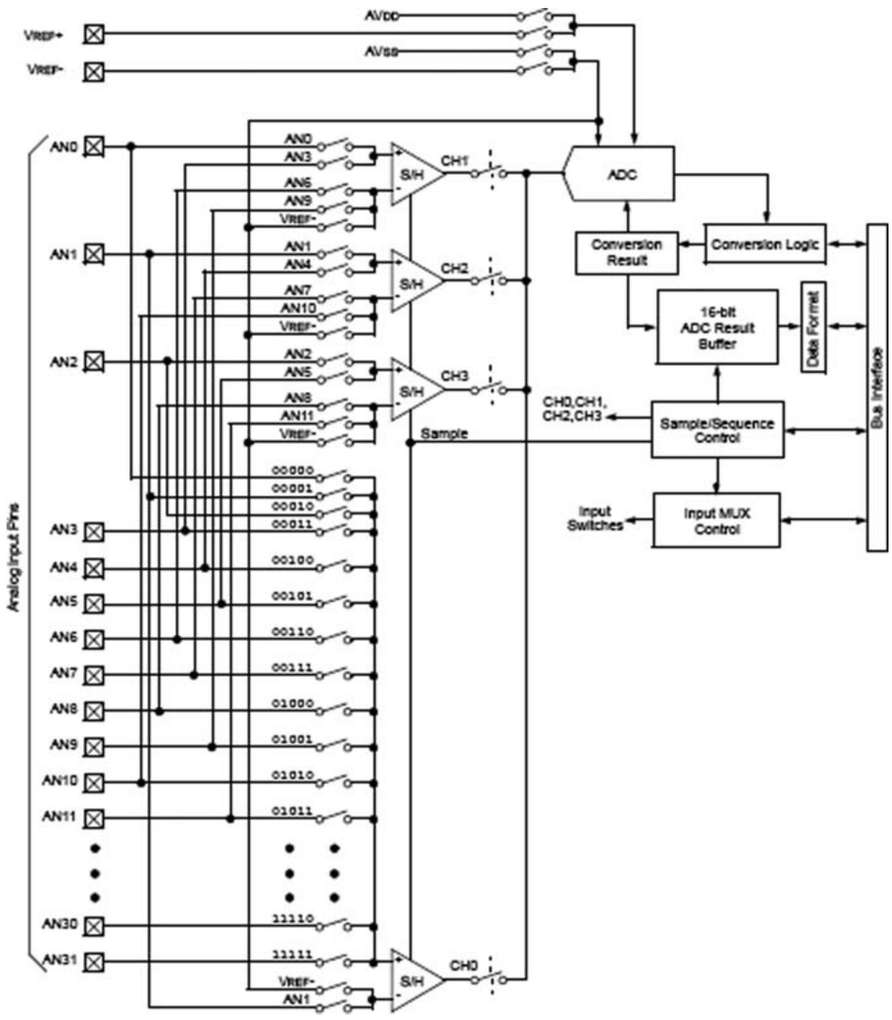


Fig. 5.1 On-chip ADC module example: dsPIC® DSC family (source: dsPIC33F Family Reference Manual)

16 kHz, or 32 kHz at best. These sampling rates are easily slow enough for SAR ADCs. A more important requirement is high resolution (12 bits or more), which is easily fulfilled by SAR ADCs as they require very few comparators.

- Ramp-Compare ADC: This is somewhat similar to SAR ADCs in that the analog input is compared to a changing reference, but here the reference is a saw-tooth waveform that ramps up and triggers a timer whenever the voltage equals that of the analog input signal. In this case, a fast and high precision timer is needed.
- Sigma-Delta ADC: This type of ADC has a smaller number of bits of resolution, but operates at substantially higher sampling rates than what is actually

needed. After the signal has been filtered, it is converted by a relatively simple Flash ADC, and the previous ADC output is then fed back and subtracted from the new output, effectively generating a signal that is the difference between the current and previous samples. Moreover, due to the oversampling, the effect of quantization noise is spread out over a wider frequency bandwidth, and a low-pass decimation filter then removes the high-frequency components. Thus, the effect of quantization error is greatly reduced, increasing the effective resolution of the ADC. Note that this oversampling technique can also be used in software on other types of ADC, to increase the effective resolution of any ADC (provided, of course, the CPU is fast enough to handle this larger data processing and filtering load). Sigma-Delta ADC uses relatively complex circuitry, and is therefore less common in on-chip ADCs (compared with stand-alone ADC devices) except to obtain very high sampling rates with adequate resolution. If other signals in the application require very high sampling rates (even though the speech signal typically does not), then this may be a suitable choice.

ADC Accuracy Specifications

It must be said that all ADCs, even those that may have the same resolution, do not perform with equal accuracy and reliability. While selecting a processor with the appropriate on-chip ADC, it is also critical to inspect the accuracy specifications published by the DSP/DSC/MCU vendor. Some of the most common parameters are the Gain Error, Offset Error, Integral Nonlinearity (INL), Differential Nonlinearity (DNL), and Monotonicity. Although these are perfectly valid parameters for gauging the accuracy of individual conversion results, the dynamic accuracy performance of the ADC is more critical for speech processing purposes. Here are some common dynamic performance specifications:

- Total Harmonic Distortion (THD): This is a measure of how much the harmonics of the fundamental frequency are contributing to the overall conversion result. In other words, given a single-frequency signal this parameter measures how much the harmonics of the frequency appear in the converted signal.
- Signal to Noise and Distortion Ratio (SINAD): This is the measure of the signal energy relative to combination of noise and harmonics in the converted signal.
- Spurious Free Dynamic Range (SFDR): This is the ratio of the signal to the maximum noise floor present in the converted signal.
- Input Signal Bandwidth: This indicates the maximum frequency of the input signal for which the ADC can correctly convert the signal samples. This is, of course, directly related to the maximum sampling rate of the ADC by the Nyquist-Shannon Sampling Theorem.
- Effective Number of Bits (ENOB): This is the overall effect of DC accuracy specifications such as the Gain Error, Offset Error and DNL, and indicates the overall average accuracy of individual conversion results.

Other Desirable ADC Features

ADC modules used for speech processing applications should provide a number of other features. The relative importance of these features is ultimately dependent on the nature of speech signal and the specifics of how the application is going to use the signal. A few of these features are given below:

- The ability to alternate or cycle between multiple inputs, for example, to capture the speech from multiple microphones.
- Differential Sampling: This may be useful in scenarios where two signals might need to be subtracted, for example, as a crude method of noise cancellation.
- The ability to sample different inputs at different sampling rates, for example, one might need to sample speech more frequently than a temperature sensor signal within the same application.
- A large number of analog inputs, depending on the complexity and scale of the application.
- Buffering of a sufficiently number of results, ideally using DMA to directly transfer the data into RAM.
- Automatic start of sampling, as well as end of sampling and start of conversion of the signal.
- Flexible set of triggering options to control the start of conversion, for example, convert a speech signal only on a timer period match.
- Ability to sample multiple inputs simultaneously, for example, if two sets of signal samples are needed to be synchronous to each other.

ADC Signal Conditioning Considerations

There are some other design considerations that need to be kept in mind when interfacing a speech signal to an ADC. If the signal is small in amplitude, for example, the signal from a microphone, a preamplifier may be needed. If the ADC is contained within the DSC or MCU device, which is desirable in embedded applications, it is very common for the on-chip ADC to be unipolar. This means that the ADC can only sample positive voltage levels, as opposed to a bipolar ADC that can sample signals that take both positive and negative values over time. For example, the ADC input might accept voltages in the 0–3.3 V range, but not voltages in the -1.65 to $+1.65$ V range. However, sensor output signals are typically bipolar. Therefore, a special type of inverting op-amp circuit called a Level Shifter is required in such cases. As discussed in Chap. 2, an Anti-Aliasing Filter will also be needed prior to sampling by the ADC; this can be implemented using the op-amp-based Sallen-Key filter structure. Another special signal conditioning circuit that may be required is a Buffer Amplifier, which is an op-amp circuit that is used to transfer a voltage from a circuit with high output impedance to a circuit with low input impedance. A simple Level Shifter circuit is shown in Fig. 5.2. The Level

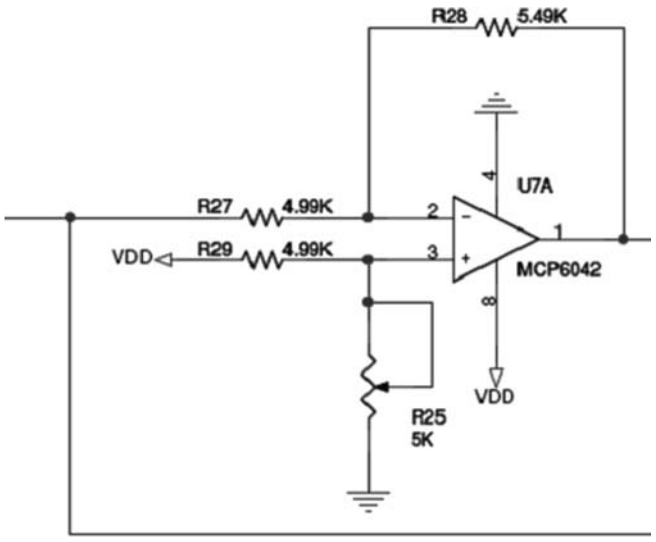


Fig. 5.2 A level shifter circuit to interface a bipolar signal with a unipolar ADC

Shifter is essentially an inverting differential amplifier, with the noninverting input derived from the positive voltage rail. Note that a potentiometer is used to adjust the midpoint voltage of the Level Shifter output.

The varying characteristics and capabilities of the ADC also place some unique requirements on the signal conditioning circuitry. Some of these are simply good circuit design practices: for example, adding a decoupling capacitor between every pair of positive voltage supply and ground pins on the DSP/DSC/MCU device helps to enhance ADC accuracy by keeping the voltage rails stable during the conversion process. Sometimes, the configuration in which the ADC is used can play a major role in determining the data conversion accuracy of the ADC: for example, using an external voltage reference pin to provide the ADC voltage rails generally produces better ADC performance when compared with using the processor's own analog power supply as reference. In addition, having the reference voltage rails identical to the device power supply levels might produce the best results.

In summary, the key to the best possible sampling and quantization of an input speech is to select the ADC with the optimal set of features for the specific application, selecting the right configuration to use, and designing the application hardware to meet the signal conditioning needs of the ADC and input signal.

Speech Playback Using Digital-to-Analog Converters [2,3]

Many speech processing applications require not only just capturing a speech signal from a microphone or other analog source, but also playing back the speech in audible form for the end-users. In fact, many applications only require playback of

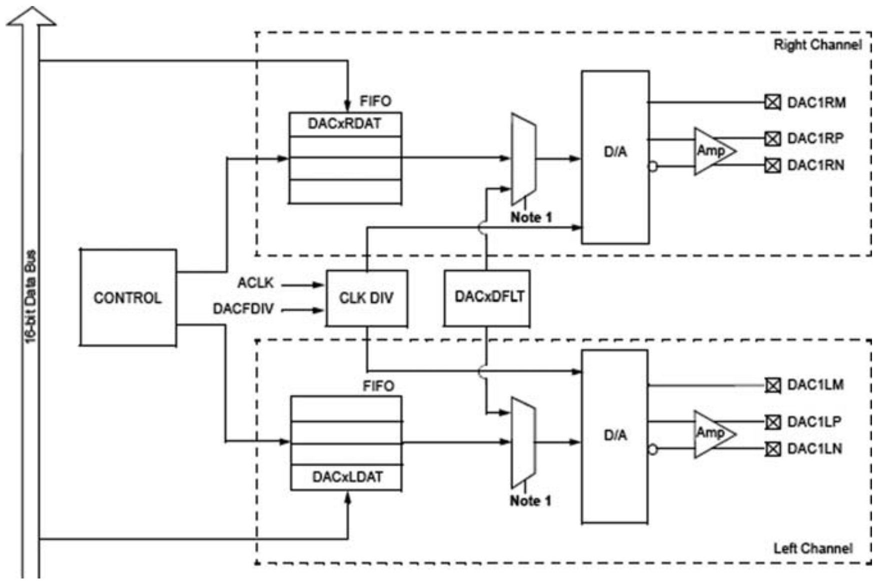


Fig. 5.3 On-chip DAC module example: dsPIC33F DSC family (courtesy: dsPIC33F family reference manual)

speech without the need for any corresponding input signal path. In either case, one of the most efficient methods of playing back speech samples is through a DAC. Just like with ADCs, there are many stand-alone DAC devices available; however, the cost and space constraints of many embedded applications dictate that an on-chip DAC be used wherever possible. As a typical example, the block diagram of the DAC in the dsPIC33F DSC device family is shown in Fig. 5.3.

Note that some DAC modules generate differential voltage outputs; in such cases a simple Differential Amplifier would be needed to convert the outputs to a single-end voltage output. If the objective is to drive a speaker, a Power Amplifier would also be needed to amplify the output speech signal to the desired level.

Similar to ADCs, the system designer should choose a device with a DAC whose accuracy specifications fulfill the needs of the application. Some of the key specifications are Resolution, ENOB, Maximum Sampling Rate, Maximum Input Data Rate, INL, Signal-to-Noise Ratio (SNR), and SINAD. Also, it is highly desirable for the device to support DMA transfers to the DAC data registers.

Speech Playback Using Pulse Width Modulation [3, 4]

Not all DSP/DSC devices have an Audio DAC on-chip (and almost no MCU does), and using an external DAC may not suit the cost or space constraints of many embedded applications. Fortunately, there does exist a popular low-cost technique

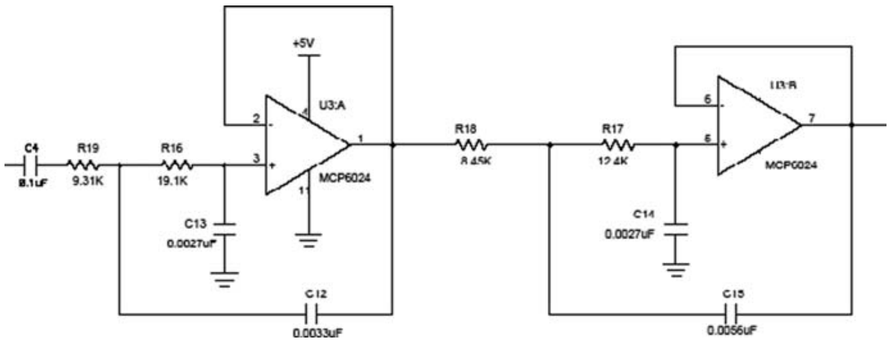


Fig. 5.4 Example circuit to convert digital PWM waveforms to analog speech signals

for generating analog speech signals from digital samples using a simple Pulse Width Modulation (PWM) waveform and a minimal external op-amp filter circuit, as shown in Fig. 5.4. Since most MCU/DSC devices and some DSPs contain an on-chip user-programmable PWM generation feature (typically as an operating mode of an Output Compare channel), external components are minimized to a couple of op-amps and a few discrete resistors and capacitors (forming an analog low-pass filter circuit).

This method operates on a simple principle: if a PWM waveform is low-pass filtered over its period, what we obtain is an average of the PWM waveform over the duration of one PWM period. Now, let us assume the Duty Cycle for the PWM waveform is continually varied at a rate that is a multiple of the intended speech sampling rate. If the Duty Cycle is always configured to be directly proportional to the speech sample value for that period, the PWM output ends up becoming the required analog speech signal. Basically, what we are doing is converting a pulse-width into an instantaneous analog amplitude value.

Interfacing with Audio Codec Devices [3, 8–10]

Converting analog speech signals to digital, and vice-versa, can be accomplished in the most cost-effective and compact manner by using an ADC on the input side and either DAC or PWM on the output side. However, in many applications, a higher digital resolution is required compared with what typical on-chip ADC and DAC devices can provide. In the case of PWM-based playback, one may need to update the duty-cycle four times more frequently than the required data rate, which might require more peripheral or interrupt handling than what some applications might be able to tolerate. For all these reasons, it sometimes makes sense to use a single external device that integrates both ADC and DAC functionality with high data resolution such as 16 bits. Such a device is known as a Coder-Decoder, or

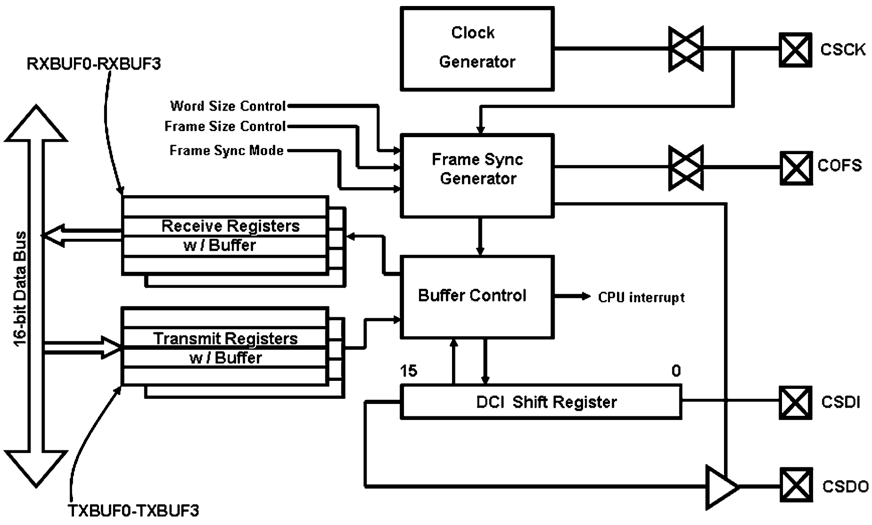


Fig. 5.5 Codec interface module example: DCI module from dsPIC® DSC family

simply a Codec. Some codecs also contain an internal Anti-Aliasing Filter, which enables the system designer to eliminate all other Anti-Aliasing Filters from the application circuit.

Codecs generally communicate digital data to/from a processor in a full-duplex fashion, using a variety of standard and generic serial communication interfaces. To respond more effectively to the interface needs of codecs, some DSC and DSP devices contain a specialized codec interface module (essentially a serial communication module with some enhanced features). These on-chip peripheral modules can typically buffer multiple data words, and in some cases they can also transfer data to/from RAM through DMA channels. An example of such a module is the Data Converter Interface (DCI) module on the dsPIC30F and dsPIC33F DSC device families, illustrated in the block diagram (Fig. 5.5).

Ideally, a codec interface used for speech processing applications should support multiple codec communication protocols. In general, most codecs communicate with the processor using some kind of a framed serial communication protocol. A framed serial communication interface is one in which there is a continuous serial bit communication clock to transfer individual bits of digital data over a full-duplex serial link (provided by either the processor, the codec, or an external clock source), that is data are transferred in both directions simultaneously. A Frame Synchronization pulse (again generated by either the processor, the codec, or from a common source) delineates the start of the transfer of a single word of data, and is generated only when actual data transfer needs to commence. Once the data transfer has been completed in both directions, the processor can utilize an interrupt handler or DMA to read the received speech sample data and also set up a new word (speech sample) to be transmitted.

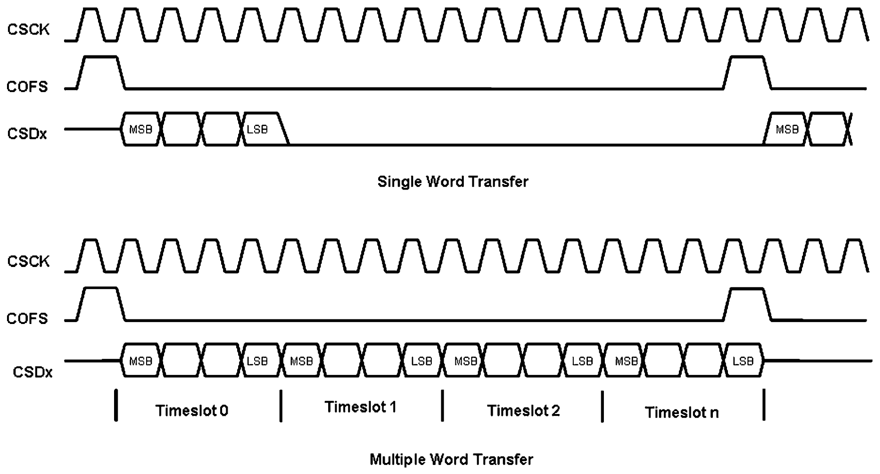


Fig. 5.6 Example of single-channel and multichannel codec communications

The above description really pertains to the simplistic case of a single-channel communication protocol. Indeed, many codecs can only process a single channel (one analog input converted to digital and one digital input converted to analog). In practice, the communications typically occurs in a multislot manner using a Time Division Multiplexed (TDM) communication protocol for the Si3000 codec from Silicon Labs, as shown in Fig. 5.6. In the TDM format, a Frame Synchronization pulse still define the start of the first word transmission/reception (also known as a "time-slot" from a TDM perspective), but the subsequent time-slots do not need distinct Frame Synchronization pulses unless they happen to be data sent to multiple codecs. If multiple codecs are used, they can be arranged in a daisy-chained fashion with the Frame Synchronization pulse propagating from one codec to another so that each one knows when to begin shifting data in/out through their individual shift registers. Multichannel TDM codecs are extremely popular in speech processing applications, and also form the Analog Front End for some other communication chipsets.

In some cases, the codec itself supports more than one channels. For example, the MSM7704–01 codec from Oki Semiconductor is a dual-channel codec. Dual-channel codecs are particularly useful in telephony applications such as Mobile Hands-Free Kits, wherein one channel can be used to interface with a microphone and speaker while the other channel can be used to interface with a cellphone (assuming the cellphone being used sends and receives analog speech signals).

Another very popular codec communications protocol is an industry-standard interface called I^2S specified by Philips. This protocol is specially designed for communicating stereo audio data, and therefore by definition uses 2 time-slots. The Frame Synchronization pulse stays low for the duration of one word (the Left Channel) and then stays high for the duration of the other word (the Right Channel). In other words, the Frame Synchronization pulse always has a 50% duty cycle in

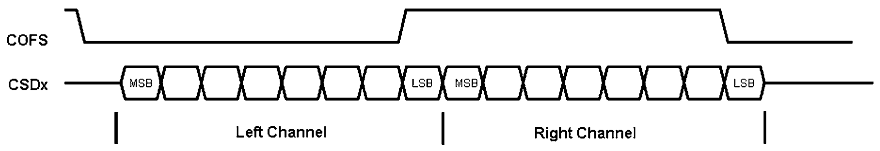


Fig. 5.7 Example of I2S codec communications

this protocol. Although this protocol has wide applicability for voiceband signals, it was primarily designed for audio applications such as music. A timing diagram for this protocol is shown in Fig. 5.7.

Finally, it must be noted that there are other possible codec communication protocols. For example, the AC97 codec standard from Intel is another popular protocol, but is more widely used (and designed for) in PC sound card and gaming applications rather than a wide range of embedded speech processing systems. Another point to keep in mind that some codecs (especially those designed for telephony applications) communicate with the processor using A-law or μ-law compressed speech samples rather than raw quantized data. In such cases, the application software needs to implement A-law/μ-law routines to interface with these codecs.

Communication Peripherals [3, 5–7]

The primary usage of serial communication interfaces in speech processing applications is to talk to external codec devices and certain communication chipsets. However, there are other serial communication peripherals and protocols that are equally useful for embedded speech processing applications. In many cases, these peripherals are not used for carrying actual speech samples, but rather for conveying either control information to other chips or subsystems in the overall application, transmit status information to a host or monitoring node, or simply used to initialize the internal parameters or registers of a codec before speech samples can be communicated. In some cases though, the communication peripherals are used to communicate speech samples to an external communication chipset or transceiver (either directly or encapsulated within some higher-layer protocol such as TCP/IP or ZigBee®). Let us therefore briefly survey the most popular communication interfaces that are present on a majority of MCUs and DSCs (and some DSPs too).

Universal Asynchronous Receiver/Transmitter

The Universal Asynchronous Receiver/Transmitter (UART) protocol is by far the most popular and easy-to-use serial communication interface for transmitting control and status information to other nodes on a network. The UART protocol is used for transmission and reception of 8-bit or 9-bit data over a serial transmit line and a serial receive line. Many DSC/MCU/DSP devices have multiple UART modules, and therefore have multiple pairs of transmit and receive pins. Like the codec communication interfaces, UART communication is full-duplex, which means that a transmission and a reception can proceed simultaneously. It is also an example of Asynchronous communication, as each communicating entity is responsible for internally generating its own timing and there is no separate serial communication clock signal (unlike the codec interfaces). This also implies that the source of the bit-timing must be rather accurate, or else the bits may be sampled and transmitted at incorrect instants.

UART is a generic interface, but it may be used with standard protocols such as RS-232, RS-422, RS-485, and Local Interconnect Network or LIN. Some standards such as LIN (which is widely used in smaller networks within automobiles and other vehicles) require some additional steps to ensure robust communications. For example, an all-zero Break character is used to signal an impending transmission, followed by a byte containing alternate zeros and ones that is used by each node to internally synchronize with the transmitting node's clock timing. Multipoint protocols such as RS-485 also support addressing of nodes using 9-bit data transmissions.

Another application of UART is to serve as the underlying physical layer interface for wireless control protocols such as IrDA®: in fact, some DSCs and MCUs can actually encode and decode UART bytes as IrDA® messages.

A typical example of a UART interface module is shown for the dsPIC30F DSC in Fig. 5.8.

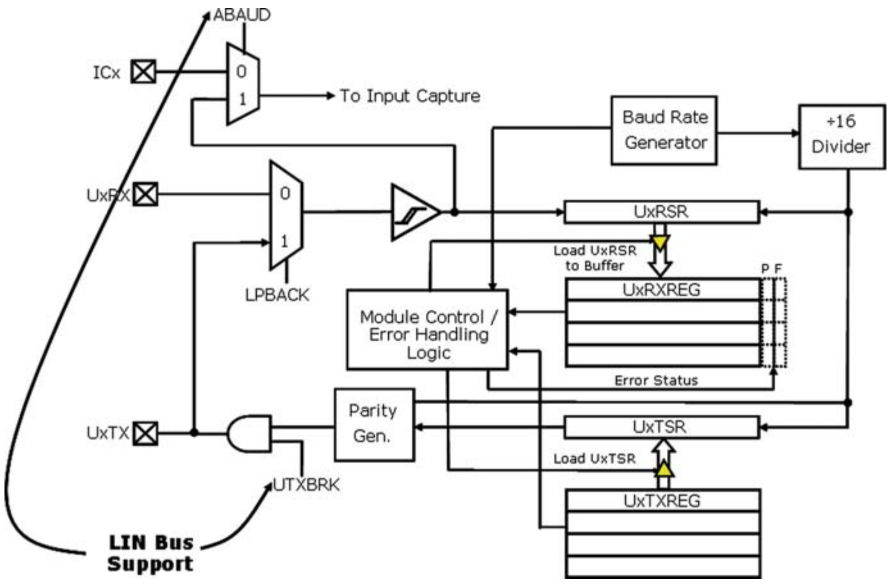


Fig. 5.8 UART interface example: dsPIC30F DSC family

Serial Peripheral Interface

The Serial Peripheral Interface (SPI) interface (a Motorola-specified standard) is commonly used for transmission and reception of bytes or words over a 3-wire interface consisting of a serial clock line (SCL), SCK, a serial data input line, SDI, and a serial data output line, SDO. Many DSC/DSP/MCU devices contain multiple SPI modules, and therefore have two sets of SCK, SDI, and SDO pins. SPI communication is full-duplex, which means that a transmission and a reception proceed simultaneously. It is also an example of Synchronous communication, as both communicating entities use the same communication clock signal. Moreover, a variety of clock polarity and data transition edge options are provided in the SPI standard, resulting in a lot of flexibility for application developers. It is also typically used at much faster bit-rates than UART, but at smaller distances (typically for communicating with devices mounted on the PCB or the same hardware system).

The SPI protocol has a wide gamut of uses in speech processing applications. It may be used for communicating speech data with external peripherals such as Analog-to-Digital and DACs, as well as communication devices such as wireless transceivers and Ethernet MAC/PHY chipsets. It can also be used to send characters to some LCD display drivers. It is one of the most common interfaces for external memory devices such as Serial EEPROMs, Serial Flash, and Serial RAM. Moreover, SPI can also form the basis of a local communication network consisting of multiple processor devices (e.g., in a parallel processing system).

Some processors also support a special framed mode of the SPI, which is identical to a single-channel codec interface. Unlike other SPI configurations, framed versions of SPI require a separate Frame Synchronization pulse.

A block diagram of the SPI module used on Microchip's dsPIC® DSCs and PIC24 MCUs is shown in Fig. 5.9 as an example of a typical SPI module.

Inter-Integrated Circuit

Inter-Integrated Circuit (I^2C^{TM}) is another Philips standard that is extremely popular in the industry, especially for short-range communication with external peripherals such as ADCs and DACs and memory devices such as Serial EEPROM, Serial Flash, and Serial RAM. Some audio codecs like the WM8510 from Wolfson, as well as some Bluetooth wireless communication chipsets, utilize I^2C^{TM} as a secondary control channel for the processor to send initialization information to the codec, in addition to the I^2S interface that is used for sending/receiving speech data samples.

The I^2C^{TM} interface is used for transmission and reception of 8-bit data over a 2-wire interface consisting of a SCL and a serial data line (SDA). I^2C^{TM} communication is half-duplex, which means either a transmission or a reception can occur at a time. It is also an example of Synchronous communication, as both communicating entities use the same serial clock signal. The I^2C^{TM} standard specifies standardized communication bit-rates of 100 and 400 kHz, though some I^2C^{TM} interface

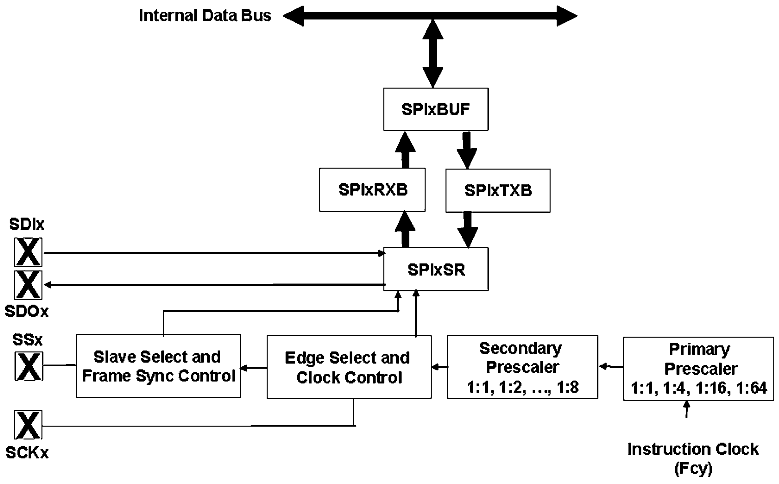

Internal Data Bus
SPIxBUF
SPIxRXB
SPIxTXB
SPIxSR
SDIx
SDOx
SSx
SCKx
Slave Select and Frame Sync Control
Edge Select and Clock Control
Secondary Prescaler 1:1, 1:2, ..., 1:8
Primary Prescaler 1:1, 1:4, 1:16, 1:64
Instruction Clock (Fcy)

Fig. 5.9 SPI module example: dsPIC30F/33F DSC family

implementations support custom bit-rates beyond these (typically not more than 1 MHz). Thus, in general I^2C^{TM} is slower than both UART and SPI.

The I^2C^{TM} protocol is comparatively more complex than UART and SPI, so readers are advised to refer to the I^2C^{TM} specs for a detailed description. However, the protocol is briefly outlined here, followed by a set of timing diagrams in Fig. 5.10.

An I^2C Master device can initiate communications on the Serial Data line by initiating a Master START sequence. The START condition is characterized by the Serial Data line going to a low logic level while the SCL is still high. Once a START condition has occurred, the Slave waits for the Master to send a Slave address byte over the Serial Data line.

The Master then transmits a Slave address byte to the I^2C^{TM} Transmit Register, starting with the MSB and ending with the LSB. The address byte consists of 7 address bits, followed by a "Read-Write" bit, which signals to the Slave whether the Master wants to initiate a reception or a transmission; in other words, it signals the required data transfer direction. If the Read-Write bit is cleared, it implies that the Master wants to send data to the Slave. If the Read-Write bit is set, the Master is expecting to receive data from the Slave.

Prior to any communication, the Slave software must assign a Slave address to the device. During communication, when an address byte is received from the Master, the Slave hardware compares this received address with its own address. If the two addresses match, then the Slave hardware automatically concludes that this particular Slave device is being addressed by the Master. During the ninth Serial Clock pulse, the Slave drives the Serial Data line low, which signifies an "Acknowledge" condition; otherwise, the Slave hardware continues to wait for an address match.

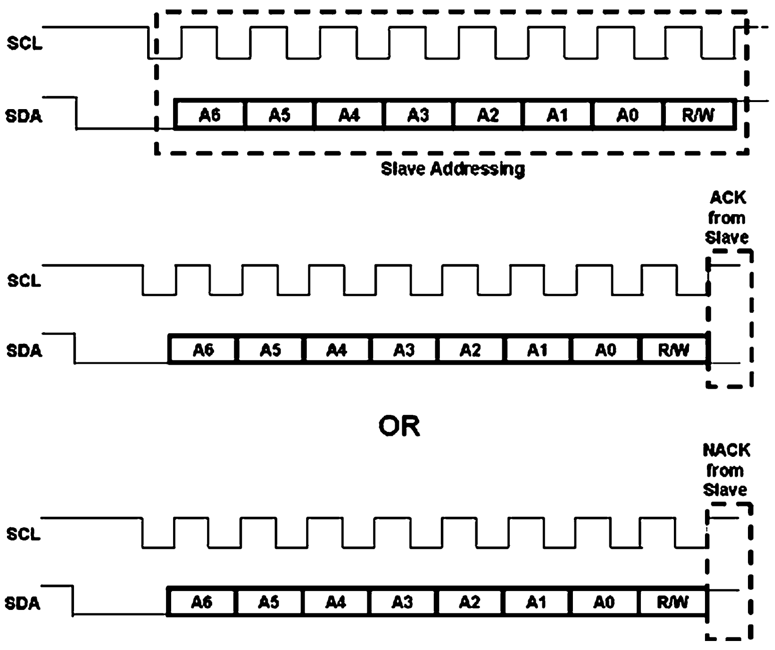


Fig. 5.10 I^{2}C^{TM} protocol timing diagrams

Controller Area Network

Controller Area Network (CAN) is an industry standard serial asynchronous communications protocol. It was originally designed for the automotive industry, but has gained wide acceptance in the industrial and medical application segments as well and has now begun to be utilized in a wider variety of applications. The CAN interface is not directly related to speech processing or communication of either speech data or command/control of codecs or memory devices; however, since speech processing is increasingly gaining usage in these same application segments. For example, a car hands-free kit system can send a CAN message to the audio subsystem of the car to mute the car radio. In another scenario, a speech recognition unit could communicate with other subsystems over CAN to implement voice commands for car functions; or, car engine status reports and alarm signals can be communicated via CAN and played back to the driver through a speech synthesis system.

The specifications for a CAN bus are described in the International Standards Organization specification ISO-11898, and the readers are advised to refer to official CAN specifications for a more detailed description of the protocol.

Other Peripheral Features [1, 8–10]

External Memory and Storage Devices

From compressed speech, large codebooks used for vocoders, linguistic syntax rule tables required for some speech synthesis techniques and speech reference templates (Hidden Markov Models) stored in nonvolatile memory, to freshly-recorded speech stored in RAM for subsequent playback, speech processing tasks are memory-intensive to say the least. For some applications (e.g., if only short durations of speech need to be stored by a speech compression application), the on-chip Flash Memory provided by the processing device may suffice. But for many other applications, the amount of data may simply be too large to be stored on-chip, and off-chip storage such as Serial EEPROM, Serial Flash, Serial RAM devices must be used, which mainly communicate with the processor using SPI, I^2C^{TM}, or some proprietary protocol. Increasingly, smart card technologies such as SD and MMC devices are in vogue because of their reliability, ease of use, and portability. These devices often use a Parallel Master Port or Parallel Slave Port interface to send entire bytes or words of data concurrently at high speeds.

Direct Memory Access

Direct Memory Access (DMA) is a mechanism by which the processor device can automatically transfer data between various peripheral modules and user RAM space without the CPU having to manually read or write individual bytes/words of data. This significantly frees up CPU bandwidth for executing computation-intensive tasks such as speech processing algorithms. Ideally, the DMA architecture should have its own dedicated transfer paths or buses so that there is minimal competition with the CPU for bus resources (i.e., no cycle-stealing).

Summary

In this chapter, we have delved into various critical peripheral features that enable efficient implementation of speech processing applications. Some of these features have a direct bearing on the speech signal conversion and communications, whereas others have an indirect impact in that they are used for controlling external devices that are important elements of the overall application system. Just as with CPU architectural features, we have seen that a careful selection of devices that have the peripheral feature-set suitable for the system requirements combined with a judicious usage of the available peripherals in the appropriate configurations, the capabilities of peripherals can be unleashed and used for effective implementation of whole system solutions that incorporate speech processing as a core element.

References

1. JL Hennessy, DA Patterson Computer Architecture – A Quantitative Approach, Morgan Kaufmann, 2007.
2. JG Proakis, DG Manolakis Digital Signal Processing – Principles, Algorithms and Applications, Prentice Hall, 1995.
3. Microchip Technology Inc dsPIC33F Family Reference Manual.
4. P Sinha (2005) DSC is an SoC Innovation, Electronic Engineering Times, July 2005, pages 51–52.
5. SPI Specifications, Motorola.
6. I^2C and I^2S Specifications, Philips.
7. CAN Specifications, Bosch.
8. http://www.ti.com – website of Texas Instruments.
9. http://www.analog.com – website of Analog Devices.
10. http://www.freescale.com – website of Freescale Semiconductor.

Chapter 6
Speech Compression Overview

Abstract Speech compression, also known as speech encoding or simply speech coding, means reducing the amount of data required to represent speech. Speech encoding is becoming increasingly vital for embedded systems that involve speech processing. This section will discuss general principles and types of speech encoding techniques, and briefly address the usage of speech compression techniques in many different types of embedded systems. Advantages and technical considerations associated with incorporating speech compression algorithms into specific embedded applications will be reviewed, which will hopefully provide some insight into the process of selecting the appropriate speech compression techniques for embedded applications.

Speech Compression and Embedded Applications [2, 3, 5]

Speech compression means reducing the amount of data required to represent speech. Speech encoding is becoming increasingly vital for embedded systems that involve speech processing. Speech compression has two primary benefits – first, in applications that require recording and/or playback of speech messages (e.g., in answering machines or verbal security alarms), it helps to reduce storage requirements, enabling longer speech segments to be recorded. Second, in applications requiring speech communications (e.g., walkie-talkies), compression reduces the amount of data to be transmitted, thereby more efficiently utilizing the available communications bandwidth.

In general, applications employing speech compression or decompression may be divided into three possible configurations, based upon when speech compression and decompression are performed relative to each other. These configurations include full-duplex, half-duplex, and simplex. Each has its own priorities and computational resource requirements, and hence the choice of compression technique to be used for an application should be made carefully.

Full-Duplex Systems

In a full-duplex configuration (Fig. 6.1), both compression and decompression are performed simultaneously. Typically, speech samples obtained from a local microphone are compressed for transmission over the communication channel, while encoded speech data received over the communication channel are decompressed for playback over a local speaker.

The widespread perception about the utility of full-duplex speech compression is in the context of traditional telephony and mobile-communication equipment, and indeed such applications benefit tremendously from reducing the amount of data that needs to be communicated. Wired telephone networks utilize the ITU standard G.711 (described later in this chapter), while different generations of mobile telecommunication standards use a variety of low or variable bit-rate algorithms recommended by standards organizations such as TIA and ETSI. In recent years, Voice-over-IP (VoIP) systems such as IP Phones and Analog Telephone Adapters have gained prominence as major users of speech compression, particularly of standard ITU coders such as G.723.1, G.726A, and G.729. This is not surprising, as bandwidth reduction and interoperability are key requirements in such systems.

Other common applications of full-duplex speech compression and decompression are Teleconference Phones, Door Security Intercoms, Medical Emergency Phones, Digital 2-way Radios, and Walkie-Talkies. The last two utilize wireless methods of communication and therefore require data reduction to conserve bandwidth.

Half-Duplex Systems

In a half-duplex application, compression and decompression are both performed, but never concurrently. Generally, speech samples recorded from a local microphone are encoded, to be decoded and played out through a speaker at a later time

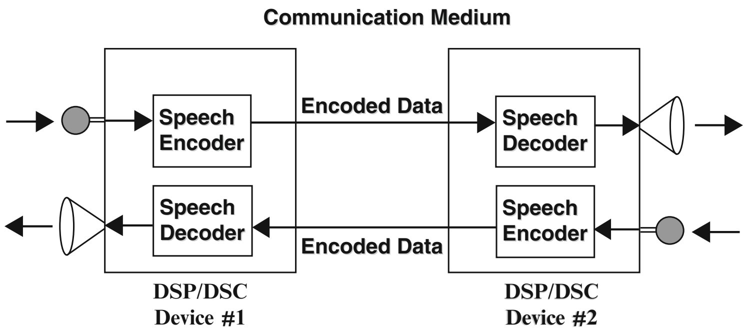


Fig. 6.1 Typical full-duplex speech compression application [5]

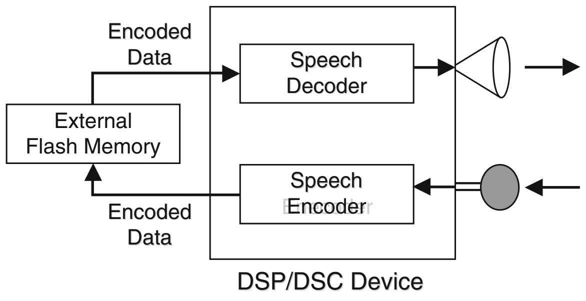


Fig. 6.2 Typical half-duplex speech compression application [5]

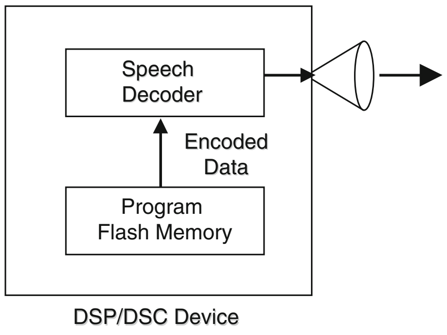


Fig. 6.3 Typical speech playback application [5]

(Fig. 6.2). As a result, these applications are computationally less demanding than full-duplex applications. For example, an incoming voice message is compressed and stored in some kind of nonvolatile memory by a tapeless answering machine, and later decompressed for playback. Another related half-duplex application is a digital voice recorder.

Simplex Systems

In a simplex configuration, either compression or decompression is performed, but never both. In most applications of this class, voice messages are precompressed offline, and only decompressed and played when needed (Fig. 6.3). Some common playback-only applications are the following:

- Talking toys
- Security alarms (e.g., to announce building evacuations)
- Museum exhibit guides
- Any embedded application that might utilize voice prompts

Types of Speech Compression Techniques [1–4]

Any speech compression/decompression (encoding/decoding) method involves sampling an input speech signal and quantizing the sample values, to minimize the number of bits used to represent them. This must be performed without compromising the perceptual quality of the encoded speech – defined as the quality of the speech as heard by the human ear – not necessarily a bit-to-bit closeness of the original and reconstructed signals. To meet these objectives, numerous speech-encoding techniques have been proposed. The choice of encoding algorithm used by an application depends upon the perceptual quality required at the decoder end, and often involves a trade-off between speech quality, output data rate (based on the compression ratio), computational complexity, and memory requirements.

Choice of Input Sampling Rate

First and foremost, the input speech sampling rate must be considered before selecting a compression technique for an application, not only because different speech compression algorithms are designed for different input speech sampling rates (G.711, G.723.1, G.726A, G.728, and G.729 are "narrowband" coders using 8 kHz sampling, whereas G.722 and G.722.1 are "wideband" coders using 16 kHz sampling), but also because different types of applications have different bandwidth requirements and therefore need different sampling rates. A 200–3,400 Hz pass-band is sufficient for telephone-quality speech, whereas a 50–7,000 Hz pass-band is more suitable for better-quality audio signals as it adequately represents both low-frequency "voiced" speech and high-frequency "unvoiced" speech. It may be noted that a few compression algorithms such as Speex support both narrowband and wideband modes.

Choice of Output Data Rate

Second, each speech compression algorithm is associated with a different output data rate, ranging from high bit rates (e.g., 64 Kbps for G.711) to low bit rates (5.3 or 6.3 Kbps for G.723.1). As we will observe later, lower output-bit rates often (though not necessarily) come at the cost of increased computational complexity and decreased speech quality.

Lossless and Lossy Compression Techniques

In general, data-compression techniques may be broadly classified into lossless or lossy methods. Lossless encoding algorithms are those in which the encoded signal, when decoded, yields a bit-exact reproduction of the original set of samples. Most speech- and audio-compression techniques use lossy algorithms, which rely on removing redundancies inherent in the data or in human hearing, rather than trying to maintain bit-exact reproducibility.

Direct and Parametric Quantization

Speech-encoding algorithms can also be classified according to the type of data that is quantized. On the one hand, in direct quantization, the encoded data are simply a binary representation of the speech samples, themselves. Parametric quantization, on the other hand, refers to encoding a set of parameters that represent various characteristics of the speech.

Waveform and Voice Coders

Parametric quantization can be either waveform or voice coders (also known as vocoders). Waveform coders encode information about the original time-domain waveform, while removing statistical redundancies in the signal. Also, waveform coders can typically be used for signals other than speech. On the contrary, vocoders try to model characteristics of the human speech-generation system and transmit a set of parameters that help to reproduce high-perceptual-quality speech, without reproducing the waveform itself.

Scalar and Vector Quantization

Many common waveform coders are based upon scalar and vector quantization. In scalar quantization (such as in the G.711 and G.726 algorithms), samples are processed and quantized individually. In vector quantization, the samples are encoded in blocks, usually making use of vector-indexed tables (referred to as codebooks). Vocoders are the method of choice for low bit-rate coders, and for several ITU-standard encoding techniques. The Speex algorithm described in this chapter is a closed-loop vocoder algorithm, known as "analysis by synthesis."

Comparison of Speech Coders [5]

Different types of speech-coding algorithms provide different ranges of functionality in terms of their bit-rates and computational complexity. The computational requirements, in particular, are a critical consideration for embedded applications.

This is especially critical because many embedded systems involve combining the speech compression/decompression with other control and user-interface tasks (e.g., a toy might generate speech as well as perform a range of motions).

On one extreme, the simple G.711 coder provides a 2:1 compression ratio. At the other end of the spectrum, vocoders such as the Speex coder provides an open-source implementation of a complex CELP-based algorithm, which provides a 16:1 compression ratio (among many others supported by the coder). The space in between is occupied by the coding techniques such as the ADPCM-based G.726A coder, which provides multiple compression ratios including 4:1 and 8:1.

The following graph illustrates the output data rates and speech quality (MOS rating) obtained by various popular speech-coding algorithms, including the three techniques described in this chapter. The trade-off between output data rate (and therefore, compression ratio) and quality of decoded speech is obvious. It is also apparent that open-source algorithms such as Speex often provide a similar range of characteristics as more expensive but industry-standard algorithms such as G.729. The speech coders described in the following two sections cover a wide range of output data rates and MOS ratings (Fig. 6.4).

Figure 6.5 shows the relationship between the million instructions per second (MIPS) requirements of various speech-encoding algorithms in relation to the compression ratio obtained by each of these algorithms. As expected, coders with higher compression ratios (e.g., Speex, G.729) use more complex, parametric computations and filter adaptations, and therefore require greater computation resources. As stated earlier, the Speex and G.729 have similar computational resource requirements but

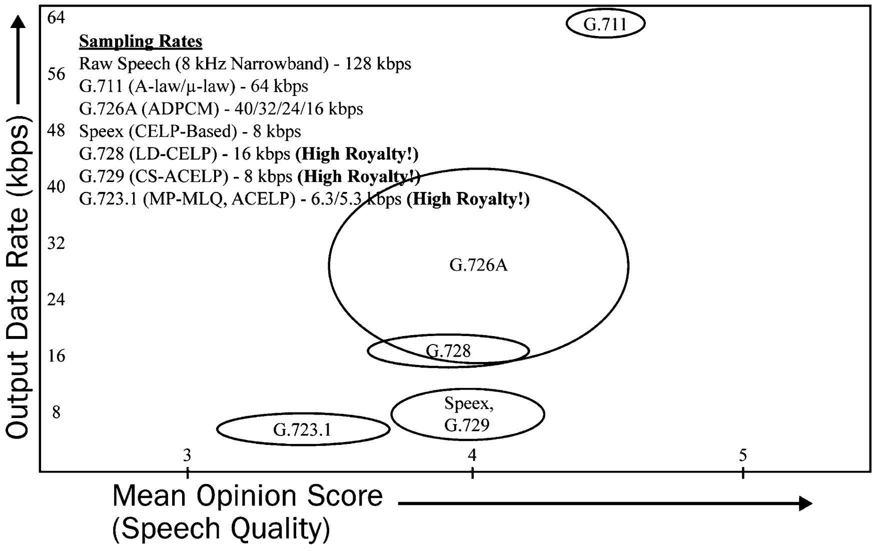


Fig. 6.4 Output data rate and MOS ratings for various speech coders [5]

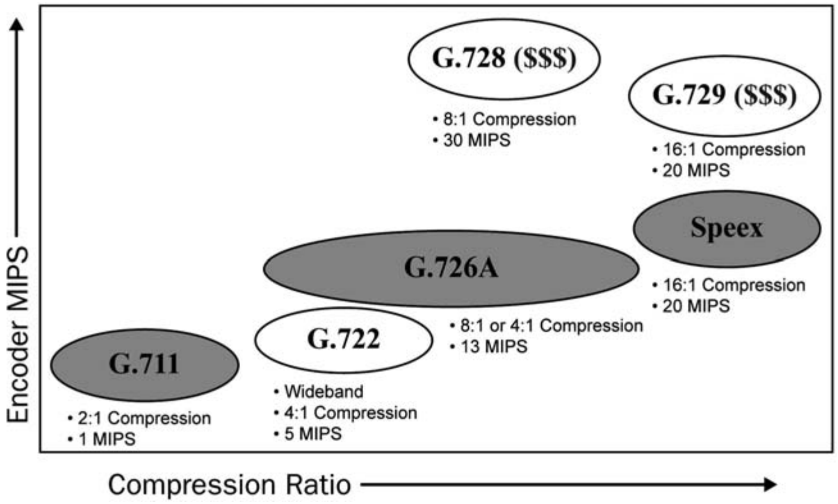


Fig. 6.5 Encoder MIPS and compression ratio for various speech coders [5]

Table 6.1 Computational resource usage of some popular speech coders [5]

	G.711	G.726A	Speex
MIPS	1	13	20
Flash (KB)	3.5	6	30
RAM (KB)	3.5	4	7
Memory (KB) needed to store 1 s of encoded speech	8	2, 3, 4, or 5	1

vary in cost of implementation and standardization requirements. At the other end of the spectrum is the G.711 coder, which extensively uses look-up tables and usually requires less than 1 MIPS on most processors.

Finally, Table 6.1 shows the MIPS, Flash Memory, and RAM requirements of three popular speech coding algorithms described in the next two chapters, using a digital signal controller (DSC) device: the dsPIC33F family. It can be observed that the MIPS, Flash, and RAM usage increases significantly, as we move toward increasingly complex coding algorithms such as Speex. Also, in some speech coding algorithms, encoding requires significantly more MIPS than decoding. For example, of the 20 MIPS required for Speex encoding and decoding, 16 MIPS are used by the encoding algorithm. This implies that applications that require speech playback only (e.g., not speech recording or communications) utilize substantially fewer computational resources.

Summary

The existence and ongoing development of increasingly sophisticated and computationally optimized speech coding algorithms present an immense opportunity for integrating speech recording and playback features into embedded applications.

Besides the telephony, wireless communications, and VoIP applications that are traditionally associated with speech processing, a wide variety of embedded control applications can also exploit speech capability through judicious selection of speech compression algorithms and the existence of CPU and peripheral features that can execute such algorithms efficiently.

In general, Speech Compression techniques utilize either the statistical characteristics of the waveform itself or strive to model and exploit the human speech production system to analyze and synthesize speech segments. Theoretically, it is also possible to utilize our knowledge of the human auditory system, for example, how the ear perceives speech of different frequencies and amplitudes. Although some of this knowledge is indeed utilized in the design of some speech coders, it is the area of Audio Compression (not covered in this book) that is the primary beneficiary of human auditory models.

Finally, it should be noted that the topic of Speech Compression is closely related to other speech processing tasks such as Speech/Speaker Recognition and Speech Synthesis, especially when techniques devised from Speech Compression research such as Linear Predictive coding are applied directly to provide the speech templates and sources for these other classes of algorithms.

In the following two sections, several popular speech coding techniques will be described in terms of the algorithms used, typical applications, and implementation considerations in the context of the architectural and peripheral features we have discussed in Sects. 4 and 5. Section 7 describes the usage of Waveform Coders such as G.711 and G.726A, whereas Sect. 8 describes Voice Coders such as G.729 and Speex. The algorithms described cover a wide range of output bit-rate: high, medium, and low.

References

1. Rabiner LR, Schafer RW Digital processing of speech signals, Prentice Hall, 1998.
2. WC Chau Speech coding algorithms, Wiley-Interscience, 2003.
3. AS Spanias (1994) Speech Coding: A Tutorial Review, Proceedings of the IEEE, vol 82, No. 10, October 1994.
4. J Holmes, W. Holmes Speech synthesis and recognition, CRC Press, 2001.
5. P Sinha (2007) Speech compression for embedded systems, Embedded Systems Conference, Boston, October 2007.

Chapter 7
Waveform Coders

Abstract In Chap. 6, we saw that there exist a wide variety of speech encoding/decoding algorithms with varying capabilities in terms of quality of decoded speech, input sampling rate, and output data rate. One way to broadly classify speech coders is based on whether the encoding algorithm makes an attempt to represent the original waveform in some form or whether the encoded data purely consists of specific parameters that try to model the characteristics of human speech for a particular speech segment and speaker. The former category of speech coders is known as Waveform Coders. Waveform coders are immensely popular in embedded applications due to their low cost, low computational resource usage, and high speech quality, even though they do not provide as high a compression ratio as the latter category of speech coders known as Voice Coders. In this chapter, different types of quantization will be discussed and correlated with various popular Waveform Coders. A special focus will be on those coders that have been standardized by the International Telecommunications Union: G.711, G.722, and G.726A.

Introduction to Scalar Quantization

We have already seen in previous chapters that after an analog signal has been sampled (e.g., by an ADC), it needs to be represented as a numerical value so that it can be stored and processed in a digital form. Naturally, this involves assigning a particular discrete signal level to it, even though an analog signal can occupy an infinite number of values in its possible range. If we had an infinite number of bits to represent an analog signal in digital form, in theory any analog signal level would correspond to a different digital value. However, in a real-life computational system, there are only a small number of bits that are available to represent each number. Therefore, the number of equivalent digital levels is also finite, and one must assign the analog signal level at any given time to the numerical value that is closest to it. This process is known as Quantization, and the difference between the actual, exact analog value and its corresponding assigned numerical value is known as the Quantization Error.

DOI 10.1007/978-0-387-75581-6_7, © Springer Science+Business Media, LLC 2010

Primarily, two factors determine the number of bits used for quantization. First, it depends on the number of bits of resolution available in the Analog-to-Digital Converter or codec (whether an on-chip peripheral module or an external device) being used. However, even if the ADC resolution is high, the number of quantization steps is ultimately limited by the innate processing/storage data width of the processor device. For example, if a 16-bit DSP is being used, the number of quantization steps is limited to what can be stored in 16 bits, i.e., a theoretical maximum of 2^{16} (or 65,536) discrete quantization levels.

The term Scalar Quantization refers to a process of quantization in which each sample is represented in isolation, by assigning it to a single numerical value for digital storage, processing, or communication. This value can be assigned in a linear fashion based on uniform step sizes, or it can be specifically optimized based on certain criteria such as Mean Squared Error or other distance metric.

Uniform Quantization [1–4]

When the assignment of analog signal levels to quantize values is done in a linear or uniform manner, it is known as Uniform Quantization. For example, if the allowed range of a signal is from 0 to 2.55 V, and an 8-bit uniform quantizer is used, then a 10-mV analog level is assigned a decimal value of 1, a 20-mV level is assigned a value of 2, and so on; in this case, there can be 256 possible digital values or quantization levels, from 0 to 255.

The size of each step ($\varDelta$) is given by the ratio of the difference between the maximum and minimum possible sampled level of the signal ($V_{\max} - V_{\min}$) and the number of steps (N).

$$\varDelta = \frac{(V\text{max} - V\text{min})}{N}. \tag{7.1}$$

The number of bits used to represent the quantized value is

$$B = \log_2(N). \tag{7.2}$$

Most ADC devices or modules assign each quantized digital value to be the midpoint of the analog signal range that particular step represents. Thus, the Quantization Error always lies in the range of $(-\varDelta/2)$ to $(+\varDelta/2)$. This results in a mean Quantization Error of 0 and the variance of the Quantization Error is $(\varDelta^2/12)$. As we have already seen in Chap. 2, the performance of any quantization scheme is denoted by its Signal-to-Quantization Noise Ratio.

In any quantization scheme, the number of bits of resolution (and in turn, the number of quantization steps) has a direct bearing on its accuracy relative to the original signal. Obviously, the finer the quantization steps, the closer the quantized signal would be to the original signal. This is especially true for Uniform Quantization, where the Signal-to-Quantization Noise Ratio increases by 6 dB for every additional bit used. Uniform Quantization can be thought of as a brute-force method of representing the original signal, and does not exploit the statistical probabilities or redundancies present in the analog signal values.

Logarithmic Quantization [1–4]

From an analysis of the probability distributions of typical speech signals, it has been found that the percentage of occurrence of different speech signal amplitudes decreases as the amplitude is increased. In other words, lower-level speech sounds occur more frequently than louder ones, as shown in the histogram (Fig. 7.1). Therefore, Uniform Quantization is not an optimal method as far as the statistical distribution of speech signals is concerned; it is more efficient to allocate the quantization steps in a nonuniform fashion with more steps (and therefore finer resolution) allocated to smaller levels. An appropriate nonlinear mapping of the input sample levels to the quantized output values is needed to accomplish this.

The most popular technique of nonlinear mapping is by using various types of Logarithmic Quantization. In its most simplistic form, the logarithm of the input signal sample is computed, and then the logarithm is quantized using uniform step sizes as described above. The slope of the input-to-output mapping curve is reduced for large input amplitudes and increased for small input amplitudes. Essentially, the logarithm operation ends up "compressing" the signal distribution by allocating finer steps to the lower levels where the speech signal samples occur more frequently. There are now less steps in the quantized output than in the input, and therefore the number of bits needed to store the quantized output samples reduces. The logarithmic function is approximated to a discrete input-to-output mapping using Piecewise Linear Approximation. Logarithmic Quantization techniques are also known as Companding (Compressing–Expanding). Pulse Code Modulation (PCM) is another name used to refer to Companding, though strictly speaking Uniform Quantization can be described as PCM as well.

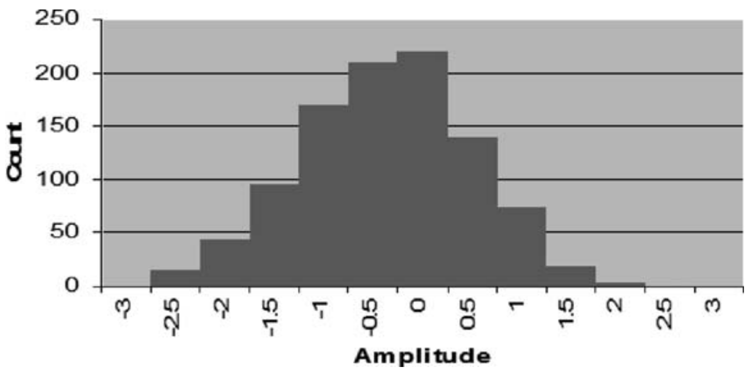


Fig. 7.1 Probability distribution showing the greater frequency of smaller signal levels

ITU-T G.711 Speech Coder [5]

Two standardized forms of Logarithmic Quantization, called A-Law and μ-Law, have been standardized by the International Telecommunications Union in its G.711 standard for standard telephony networks, with A-Law being standardized for Europe and international telephone calls and μ-Law being standardized in USA, Japan, and many other countries. Hands-free kits and other mobile-phone accessories that communicate with mobile phones or wireless devices, such as Bluetooth® chipsets, also use these methods of speech compression. Many codec devices also assume an A-law or μ-law data format for communicating with DSP/DSC/MCU devices.

For any given input signal x, the mapping function, or characteristic function, for μ-Law compression is given by

$$f(x) = \frac{\ln(1 + \mu \times |x|)}{\ln(1 + \mu)} \operatorname{sgn}(x), \tag{7.3}$$

where x is the Input signal amplitude as a fraction of $V_{\max}$ and μ is the Constant that controls amount of nonlinearity.

Similarly, the characteristic function for A-Law compression is given by

$$f(x) = \frac{A \times |x|}{1 + \ln(A)} \operatorname{sgn}(x) \tag{7.4}$$

when $|x| < 1/A$.

$$f(x) = \frac{1 + \ln(A \times |x|)}{1 + \ln(A)} \operatorname{sgn}(x) \tag{7.5}$$

when $1/A \le |x| \le 1$.

Figure 7.2 illustrates the effect of A-Law or μ-Law compression on the uniform PCM samples.

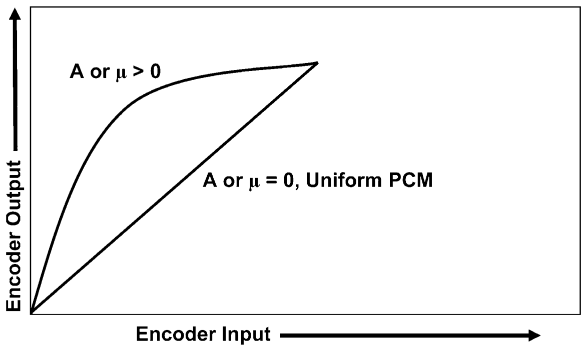


Fig. 7.2 Effect of companding on input quantization

The G.711 standard uses a μ value of 255 and an A value of 87.6. However, embedded applications that do not have an interoperability requirement (e.g., a closed communication network that is communicating to a remote entity, or a stand-alone voice recorder) can choose to use other values for these constants.

In a real-life embedded implementation of the G.711 standard, it is not necessary to compute (7.3) or (7.4) and (7.5). The G.711 Recommendations document explicitly specifies the quantized output value for each possible value of the sampled input. For the mapping, it is assumed that the input signal was sampled (e.g., by an ADC) with a resolution of 13 bits for A-Law and 14 bits for μ-Law; thus, each input sample is assumed to be a uniformly quantized digital input. Also, note that in both A-Law and μ-Law, each of the output samples is an 8-bit value. Considering the typical example of a 16-bit fixed-point DSP/DSC, the inputs would effectively be 16-bit values since they would have to be stored in 16-bit registers or variables anyway. Thus, the Compression Ratio for G.711 compression is effectively 2:1 in a real system. The G.711 standard has been designed for an 8-kHz input sampling rate (in fact, most telephony applications assume an 8-kHz sampling rate which produces reasonable "toll-quality" speech). The output data rate is therefore $(8 \text{ bits} \times 8{,}000 \text{ samples/s}) = 64$ Kbps.

Due to the high output data rate, G.711 has been observed to produce a very high Mean Opinion Score (MOS) of around 4.3. It should also be noted that the G.711 speech coder requires minimal computation: less than 1 MIPS on most processors. Thus, even though the Compression Ratio is only 2:1, this method of speech compression is perfectly suited for applications which need to perform other computationally intensive tasks (e.g., an application that is also running tight control loops for a power conversion application, or one that is performing sophisticated motor control or pattern recognition tasks). The memory requirements are also negligible, as the processing can be performed on a sample-by-sample basis without the need for any buffering. Of course, an application may buffer an entire frame and execute speech compression on it if it proves more beneficial for the real-time constraints of the application.

ITU-T G.726 and G.726A Speech Coders [7,8]

The ITU standard G.726A coder is a variant of the G.726 speech coder, which is another immensely popular scalar quantization technique. The main difference between G.726 and G.726A is that G.726 assumes that its input samples are in A-law or μ-law encoded format. G.726A, on the other hand, operates on 14-bit Uniform PCM input samples and therefore does not require additional G.711 encoder/decoder functionality.

G.726A supports multiple output data rates – 16 Kbps, 24 Kbps, 32 Kbps, and 40 Kbps, with the higher data rates providing a significantly better speech quality (MOS rating). For example, at 40 Kbps and 32 Kbps, the MOS is 4.5 and 4.4, respectively. This decreases to 4.1 at 24 Kbps and 3.4 at 16 Kbps. Nonetheless, the G.726A coder straddles a wide range of output data rates between G.711 on the one hand and low bit-rate coders on the other.

The G.726A algorithm is based upon the principle of ADPCM which, like all Differential PCM-coding schemes, compresses data by trying to remove the redundancies in the input-data stream. This is accomplished by exploiting the fact that adjacent speech samples are highly correlated. In the simplest case, the difference between two adjacent samples can be quantized using far lower bits than would be required to quantize the samples themselves, thus reducing the overall number of bits needed to represent the waveform.

In practice, the difference (known as the error signal) between the current sample and a linear combination of past samples is quantized, providing the encoder output that may be transmitted or stored. G.726A quantizes each error value using 5 bits in the 40-Kbps mode, 4 bits in the 32-Kbps mode, 3 bits in the 24-Kbps mode and 2 bits in the 16-Kbps mode. The process of predicting the current input sample by filtering past samples is known as a Prediction Filter.

In an ADPCM algorithm, the coefficients of this prediction filter are not fixed. Rather, they are continually changed based upon the encoded error values and a reconstructed version of the original signal. In many algorithms, the quantizer used to quantize the error signal is itself adaptive. Likewise, in some algorithms, the quantizer scale factor and quantizer adaptation speeds are iteratively adjusted based upon the error values and prediction-filter coefficients. In fact, this is the approach stipulated in the G.726A standard.

Encoder

The G.726A encoding algorithm uses a prediction filter with eight coefficients – two poles and six zeros. These coefficients are adapted using a gradient algorithm that tries to minimize the mean square of the error values (e.g., the encoder output). The G.726A encoder uses backward adaptation, which means that the prediction filter parameters are estimated from past speech data (unlike forward adaptation, where current speech data is used). This has a significant advantage – the past speech data is already available at the decoder end.

Figure 7.3 is a simplified block diagram of the G.726A encoder. The G.726 and G.726A Recommendation documents contain the specific steps that are used to compute the Adaptive Predictor coefficients as well as the input/output characteristics for the Adaptive Quantizer. The computations involved are reasonably simple (though collectively numerous) and do not require repeated multiply accumulate operations or too many multiplications, unlike many typical speech or other signal processing tasks. In fact, some functional blocks such as the Quantizer are completely implemented using standardized lookup tables specified in the standard documents.

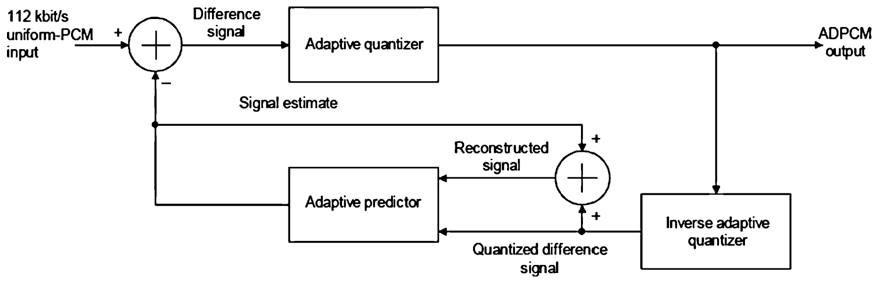


Fig. 7.3 Block diagram of G.726A speech encoder (taken from ITU-T G.726A recommendations)

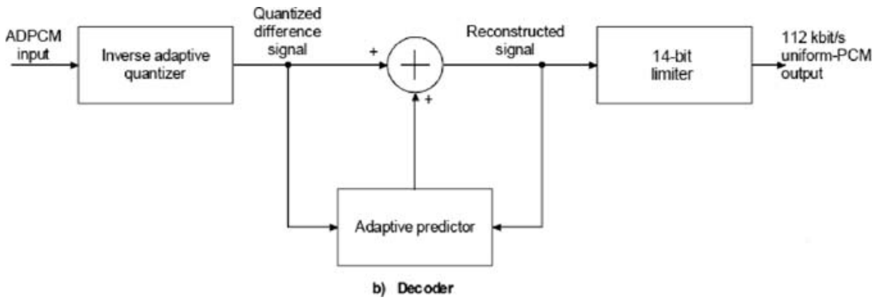

ADPCM input
Inverse adaptive quantizer
Quantized difference signal
+
+
Reconstructed signal
14-bit limiter
112 kbit/s uniform-PCM output
Adaptive predictor
b) Decoder

Fig. 7.4 Block diagram of G.726A speech decoder (taken from ITU-T G.726A recommendations)

Since 5, 4, 3, or 2 bits are used to represent the signal, and since this standard algorithm is based on an 8-kHz input sampling rate, the resultant output data rates are 40, 32, 24, and 16 Kbps. The most common data rates are 32 and 16 Kbps, resulting in Compression Ratios of 4:1 and 8:1, respectively.

Decoder

The decoder can mimic the prediction-filter update, without requiring the encoder to transmit prediction-filter coefficients. In fact, the only data that the encoder transmits to the decoder is the quantized-error signal. This keeps the communication bandwidth usage at a minimal level. Figure 7.4 shows a simplified block diagram of the decoder.

G.726/G.726A provides greater levels of compression compared with G.711 and is therefore more suitable where some amount of computational headroom is available to the processor and where memory space to store the encoded data is at a premium. Another subtlety to keep in mind is that this speech coder operates on 14-bit input samples, so the digital data obtained from an ADC may need to be aligned suitably. Finally, the quality of the decoded speech decreases with decreasing output data rates, with the 32-Kbps mode providing an approximate MOS rating of 4.1. At 40 Kbps, the perceived quality of the decoded speech is comparable to G.711, which speaks to the merits of this technique.

ITU-T G.722 Speech Coder [6]

The G.711 and G.726A speech coding techniques operate on speech that has been sampled at a rate of 8 kHz. While this sampling rate is perfectly acceptable for telephone quality speech, there are limits to the fidelity of the 8-kHz sampling since the maximum bandwidth of the input speech must be less than 4 kHz (also called Narrowband speech) as per the Nyquist–Shannon Sampling Theorem. Voiced speech mainly concentrated within this frequency band, but unvoiced speech has significant energy content up to 7 kHz. Therefore, to accurately sample all the frequency components of speech, an input sampling rate of 16 kHz is more effective. Moreover, nonspeech signals such as musical tones and chimes would need higher sampling rates such as 16 kHz for good tonal quality.

The G.722 Speech Coder is one of the few standardized speech encoding/decoding algorithms that require a 16-kHz input sampling rate and can therefore be described as a Wideband speech coder.

Encoder

As shown in the simplified encoder block diagram in Fig. 7.5, the input signal samples are split into two subbands: a lower subband with a bandwidth up to 4 kHz and a higher subband between 4 and 8 kHz. This splitting is accomplished using a special pair of filters called Quadrature Mirror Filters (QMF). The lower and higher subbands are encoded using 6 and 2 bits, respectively, per sample, resulting in encoded data rates of 48 Kbps and 16 Kbps. These two bit-streams are then interleaved to produce an overall output data rate of 64 Kbps, i.e., a Compression Ratio of 4:1.

Figures 7.6 and 7.7 show a more detailed view of the lower subband and higher subband encoder, respectively. The principle of operation is very similar to the ADPCM algorithm used in G.726A, but with different bit rates being used and the fact that the output data rate is constant. An interesting point to note is that

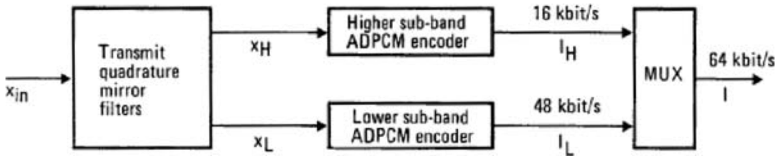

x_{in}
Transmit quadrature mirror filters
x_H
x_L
Higher sub-band ADPCM encoder
Lower sub-band ADPCM encoder
16 kbit/s
I_H
48 kbit/s
I_L
MUX
64 kbit/s
I

Fig. 7.5 Simplified block diagram of G.722 encoder (taken from ITU-T G.722 recommendations)

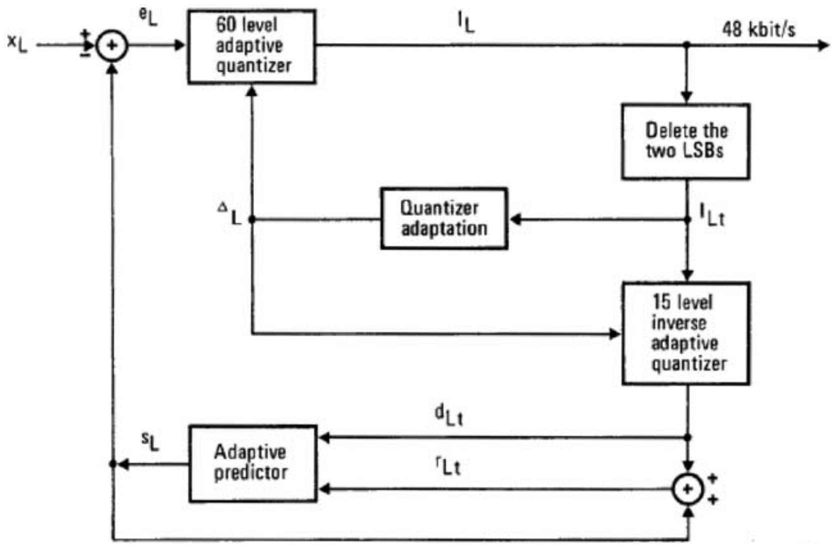


Fig. 7.6 Block diagram of the lower subband ADPCM encoder (taken from ITU-T G.722 recommendations)

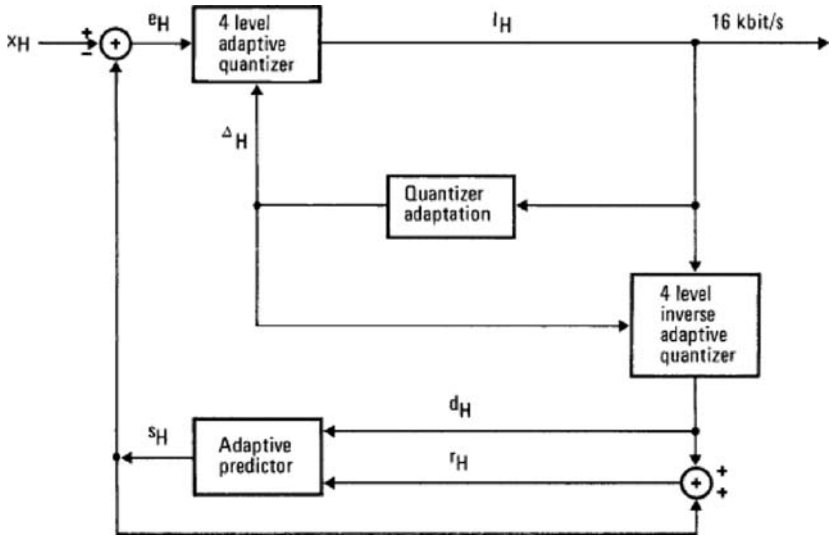


Fig. 7.7 Block diagram of the higher subband ADPCM encoder (taken from ITU-T G.722 recommendations)

the Inverse Adaptive Quantizer in the lower subband feedback loop uses a less precise quantization (15-level or 4-bit) than the 60-level (6-bit) Adaptive Quantizer. The higher subband encoder uses 4-level (2-bit) encoding, reflecting the more precise representation needed for voiced speech compared with unvoiced speech.

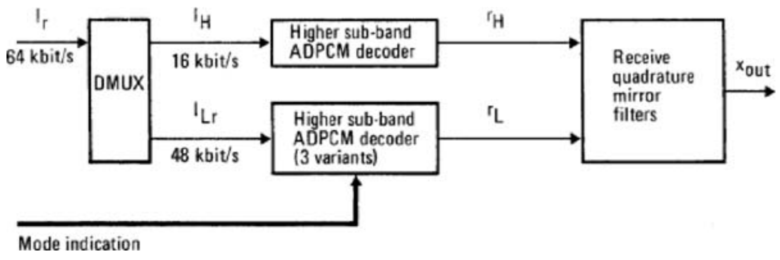

I_r
64 kbit/s
DMUX
I_H
16 kbit/s
I_{Lr}
48 kbit/s
Higher sub-band ADPCM decoder
Higher sub-band ADPCM decoder (3 variants)
r_H
r_L
Receive quadrature mirror filters
x_{out}
Mode indication

Fig. 7.8 Simplified block diagram of G.722 decoder (taken from ITU-T G.722 recommendations)

Decoder

On the decoder side, the bit-stream is separated into the 16-Kbps and 48-Kbps subband bit-streams corresponding to the encoded higher and lower subbands, respectively, and then decoded separately before combining using Quadrature Mirror Filters. An optional feature of the G.722 algorithm is that 8 Kbps or 16 Kbps out of the lower subband encoded bit-stream of 48 Kbps can be used to carry an auxiliary data channel instead of speech, and a Mode Indication signal is needed at the lower subband decoder to specify which, if any, of these optional modes is being used.

Figure 7.8 shows a simplified block diagram of the G.722 Speech Decoder, while Figs. 7.9 and 7.10 illustrate the detailed block diagrams of the lower and higher subband decoders, respectively.

In terms of computational requirements, G.722 requires slightly higher MIPS and memory than typical G.726A implementations (but still less than 20 MIPS on most processors), mainly owing to the fact that two encoders and two decoders need to be executed. Just as with G.711 and G.726A, frame processing is not mandatory, but the application might choose to process the data in frames to reduce data transfer overheads and to be able to exploit the presence of Direct Memory Access (DMA) in many DSP/DSC architectures. Also like in the other two standards we have seen, lookup tables, predetermined decision procedures, and standardized rudimentary computations are used extensively, except for the Quadrature Mirror Filters which require MAC operations. The MOS rating is typically observed to be around 4.1, comparable with the performance of the G.726A coder operating in its 32-Kbps mode.

Summary

In this chapter, we have studied several approaches to represent the waveform of a speech signal with a reduced number of bits per encoded sample. These algorithms mainly produce high to medium output data rates, such as 64 Kbps, 32 Kbps, and

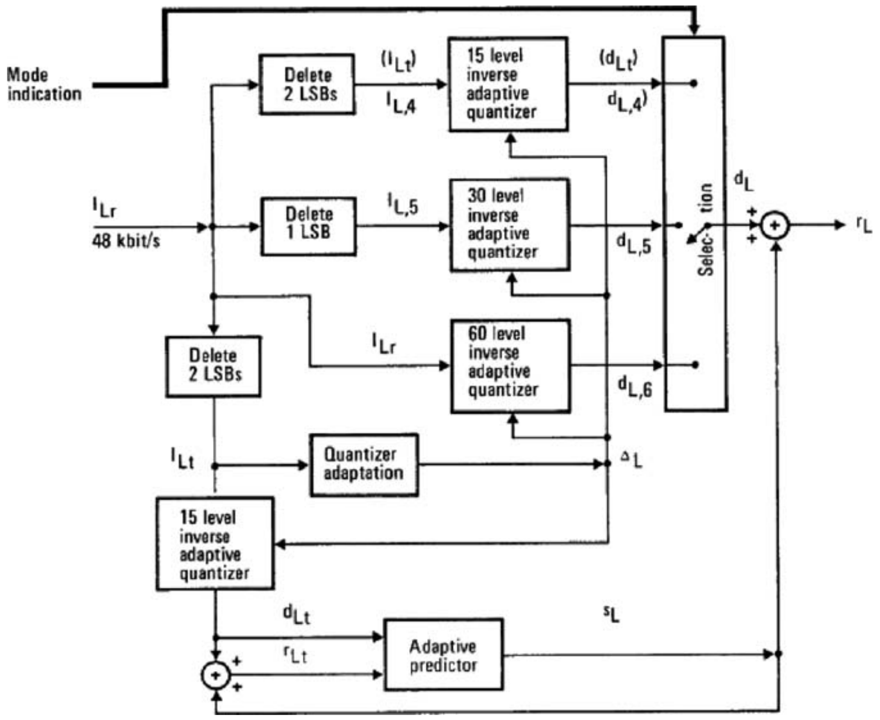


Fig. 7.9 Block diagram of the lower subband ADPCM decoder (taken from ITU-T G.722 recommendations)

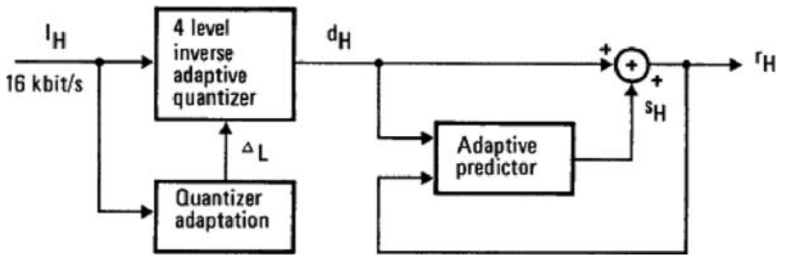


Fig. 7.10 Block diagram of the higher subband ADPCM decoder (taken from ITU-T G.722 recommendations)

16 Kbps, with generally decreasing speech quality as the bit rate is decreased. The computational resource requirements range from absolutely minimal in G.711 to moderate requirements in G.722 and G.726A. Due to the high speech quality and relatively low computational needs, all these algorithms are tremendously popular in embedded speech processing applications. Moreover, Waveform Coder algorithms are generally royalty free, which will no doubt result in their increased adoption in cost-sensitive embedded systems in the near future.

References

1. Rabiner LR, Schafer RW Digital processing of speech signals, Prentice Hall, 1998
2. Chau WC Speech coding algorithms, Wiley-Interscience, 2003
3. Spanias AS (1994) Speech coding: a tutorial review. Proc IEEE 82(10):1541–1582
4. Sinha P (2007) Speech compression for embedded systems. In: Embedded systems conference, Boston, MA
5. ITU-T Recommendation G.711: Pulse code modulation (PCM) of voice frequencies
6. ITU-T Recommendation G.722: 7 kHz audio-coding within 64 kbit/s
7. ITU-T Recommendation G.726: 40, 32, 24, 15 kbit/s Adaptive differential pulse code modulation (ADPCM)
8. ITU-T Recommendation G.726 Annex A: extensions of recommendation G.726 for use with uniform-quantized input and output

Chapter 8
Voice Coders

Abstract In Chap. 7, we saw various techniques for representing the waveform of speech signals in such a way that the number of bits used to represent each sample was minimized. Generally, such algorithms exploit the inherent redundancy and spectral characteristics of the speech signal, but nevertheless the original waveform can be reproduced at the decoder to a large extent. However, these Waveform Coder algorithms did not provide a very high compression ratio; hence they are not very effective when low output data rates are required in an application, either due to constraints in the memory available for storing the encoded speech or due to limited communication bandwidth. In this chapter, we will shift our focus to speech encoding algorithms that attempt to parameterize each segment of speech by encoding the characteristics of a human speech production model rather than the waveform itself. This class of speech coders, known as Voice Coders or simply Vocoders, provides applications with a greater degree of speech compression, albeit at the cost of not being able to reproduce the speech waveform itself. There are a large variety of Vocoder standards providing various capabilities, including several standards for mobile communications such as TIA IS54 VSELP and ETSI GSM Enhanced Full Rate ACELP. Only a few representative coding techniques suitable for embedded applications are described in this chapter, including some specific standardized vocoders (G.728, G.729, and G.723.1) and an open-source speech coding algorithm (Speex).

Linear Predictive Coding [1–4]

Unlike Waveform Coders that attempt to represent the original waveform with a reduced data rate by exploiting the statistical redundancies present in the waveform, Voice Coders (also referred to as Vocoders) do not attempt to encode the speech waveform itself or its spectral characteristics. Instead, Vocoders attempt to model the behavior of the human speech production system during each speech segment and encode a finite set of such characteristics as parameters. This results in a substantial reduction in the amount of data required to represent the speech frame, since now it is not the original waveform that is being encoded but simply a set of model parameters. The characteristics that are most representative of the human speech production system are the formant frequencies of the vocal tract and the properties of the excitation signal generated by the vocal cords (which, as we have seen in Chap. 3, may be a fundamental Pitch Frequency for voiced speech and simply a noise source model for unvoiced speech). Speech parameters vary from one frame to another, so the method to estimate these model parameters must be adaptive enough to dynamically track such variations as optimally as possible.

The most popular and effective way to encode these speech model parameters is through a technique known as Linear Prediction. We have already seen a variant of Linear Prediction in the ADPCM-based coders discussed in Chap. 7. However, in those coders the Adaptive Predictors were designed to predict the current speech sample based on a linear combination of past input speech samples and encoding the difference between this predicted sample value and the true value (thereby reducing the size of encoded data). In Vocoders the same concept is used in a broader way to predict the parameters of the human speech production system using a linear combination of past input speech samples. Indeed, Linear Predictive Coding (LPC) forms the foundation of most Vocoder algorithms used today.

Thus, the predicted nth signal sample $y[n]$ is given by the following difference equation, where a_i are a set of coefficients known as LPC coefficients:

$$y[n] = \sum_{i=1}^{N} a_i x[n-i]. \tag{8.1}$$

The difference $e[n]$ between this predicted value and the actual nth sample is known as the Prediction Error, and is given by:

$$e[n] = x[n] - y[n] = x[n] - \sum_{i=1}^{N} a_i x[n-i]. \tag{8.2}$$

In order to model the human speech parameters as closely as possible, LPC algorithms attempt to minimize the above Prediction Error and encode the set of LPC coefficients that produce this minimum error. Thus, to determine the optimal coefficient values, the Prediction Error equation needs to be differentiated with respect to each coefficient and equated to zero. This results in an array of N linear equations in the Nth dimension (where N is the prediction filter order or number of coefficients), as given below

$$Ra = r, \tag{8.3}$$

where

$$R = \begin{bmatrix} R(0) & R(1) & \dots & R(N-1) \\ R(1) & R(0) & \dots & R(N-2) \\ \vdots & \vdots & \vdots & \vdots \\ R(N-1) & R(N-2) & \dots & R(0) \end{bmatrix}, \tag{8.4}$$

$$r = \begin{bmatrix} R(1) \\ R(2) \\ \vdots \\ R(N) \end{bmatrix}. \tag{8.5}$$

Levinson–Durbin Recursive Solution

In theory, solving the above system of linear equations requires a matrix inversion operation over very large dimensions. This would be computationally prohibitive in typical embedded processing architectures as it is an $O(N^3)$ operation. Hence, alternate algorithms to compute the LPC coefficients without matrix inversion. Fortunately, the matrix $\boldsymbol{R}$ in this case is always a Hermitian Toeplitz matrix, and several optimized recursive algorithms exist that can be used to solve the above equations and obtain the optimal LPC coefficients. These techniques are typically $O(N^2)$, thereby significantly reducing the amount of computations needed. Two such algorithms are Leroux–Gueguen and Levinson–Durbin. An advantage of the Levinson–Durbin method is that both the coefficients and the corresponding Reflection Coefficients are obtained, and the Reflection Coefficients can be used to generate a more numerically stable representation of the LPC coefficients; hence, the computational steps in Levinson–Durbin algorithm are briefly outlined below.

$$J_0 = R[0], \tag{8.6}$$

$$k_l = \frac{1}{J_{l-1}} \left(R[l] - \sum_{i=1}^{l-1} a_i^{(l-1)} R[l-i] \right). \tag{8.7}$$

For values of l up to N, compute (8.8) and (8.9)

$$a_l^{(l)} = k_l. \tag{8.8}$$

For $i = 1, 2, \ldots, (l-1)$, compute (8.9)

$$a_i^{(l)} = a_i^{(l-1)} - k_l a_{l-i}^{(l-1)}. \tag{8.9}$$

The mean-squared Prediction Error for the lth order equation is

$$J_l = J_{l-1} \left(1 - k_l^2\right). \tag{8.10}$$

At the end of this recursive procedure, the final LPC coefficients are given by the set: $a_1^{(N)}, a_2^{(N)}, \ldots, a_N^{(N)}$.

Short-Term and Long-Term Prediction

In Vocoders, the LPC technique is generally used in multiple instances to obtain different parameters of the human speech production model. These can be categorized into two: Short-Term Prediction and Long-Term Prediction.

When LPC is being used to determine the optimal set of coefficients representing the formant frequency response of the vocal tract at any given time, a relatively large number of coefficients are used. However, these only involve sample values in the immediate vicinity of the current sample, as the formant frequencies are relatively high frequencies (compared with the fundamental pitch frequency, for example). Thus, the process of determining the formant LPC coefficients is known as Short-Term Prediction and is similar to the difference equation described by (8.1).

The LPC methodology can also be used to determine the pitch frequency of the excitation signal model. Of course, this model only applies to periods of voiced speech; for unvoiced speech a pseudorandom noise generator is used to model the excitation. Note that the fundamental pitch frequency is relatively low; however, the pitch varies much more frequently compared with the formant characteristics of the vocal tract. Hence, a higher-order difference equation needs to be used for (8.1), and this needs to be computed more frequently than the Short-Term Prediction, e.g., four times within each speech frame rather than once for every frame. This might seem to be computationally more demanding. However, in general it has been observed that most of the LPC coefficients may be assumed to be zero; in fact, most Vocoders use a single coefficient, representing the fundamental pitch frequency, as follows:

$$e_s[n] = \beta e_s[n-T] \quad \text{where } \beta = \text{Pitch Gain},\ T = \text{Pitch Period}. \tag{8.11}$$

Other Practical Considerations for LPC

Speech signals have a typical characteristic that exhibits a reduction in amplitude at high frequencies. This could potentially result in LPC coefficients with marginal numerical stability. To alleviate such problems, the speech samples are often pre-emphasized by running them through a first-order high-pass FIR filter. This process emphasizes the high frequency components of speech to some extent. However, care should be taken that the characteristics of the speech are not altered significantly. At the decoder, the decoded samples are similarly de-emphasized to compensate for the effect of the pre-emphasis.

The LPC coefficients can be such that when the original speech is synthesized, the poles of the synthesis filter are located too close to the unit circle, potentially leading to instability. This problem can be avoided by scaling each LPC coefficient by a factor less than 1, thereby pushing the poles of the synthesis filter away from the unit circle. This scaling flattens the frequency spectrum of the speech signal and expands its bandwidth. Bandwidth expansion also helps make the algorithm more resilient to errors in the signal channel between the encoder and decoder, as the impulse response duration is now shorter and thus the effect of channel errors lasts for a shorter duration.

In some cases, especially when the linear equations to be solved are ill conditioned, it is beneficial to apply a windowing function to the autocorrelation values in the matrix. This effectively smoothes out sharp peaks in the spectral characteristics of the signal.

For the Long-Term Prediction to be an effective model for the excitation signal, speech frames or subframes need to be classified as either voiced or unvoiced so that Long-Term Prediction can be performed only on voiced sounds. This voiced/unvoiced selection can be performed using a combination of calculations such as Short-Time Energy, Zero Crossing Rate, and Prediction Gain (ratio of signal energy to prediction error energy).

In many speech coding algorithms, the LPC coefficients are not directly quantized, but rather are transformed into some kind of alternate representation before quantization. This is mainly to improve the computational stability during synthesis. For example, the Reflection Coefficient values obtained during the Levinson–Durbin Recursion may be used instead, since Reflection Coefficients are directly related to the structure of the human vocal tract system.

A very popular alternate representation of the LPC coefficients is Line Spectral Frequency (LSF). Let us split the prediction filter polynomial $A(z)$ into two complex conjugate polynomials as given below:

$$P(z) = A(z)\left(1 + z^{-(M+1)}\frac{A(z^{-1})}{A(z)}\right), \tag{8.12}$$

$$Q(z) = A(z)\left(1 - z^{-(M+1)}\frac{A(z^{-1})}{A(z)}\right). \tag{8.13}$$

Then, the LSFs are the frequencies resulting in zeros of $P(z)$ or $Q(z)$. Since they occur in complex conjugate pairs, they are also known as a Linear Spectral Pair (LSP). These polynomials have zeros at $z = \pm 1$; since these zeros are known the order can effectively be reduced by eliminating them, thereby reducing the amount of encoded data. There are various root-finding techniques that can be used to transform the LPC values into the corresponding LSPs; the readers are pointed to the references for a detailed description of some of these methods.

Finally, a common feature of most standard Vocoders is that there is some requirement for Noise Weighting (also called Perceptual Enhancement). This refers to a noise-shaping filtering operation that causes the noise spectrum to be modified in such a way that more of the noise is present in those frequency bands where the speech is louder. In other words, the noise is concentrated in those frequency regions where the ear cannot perceive it clearly; this is related to the Masking phenomenon that was described in Chap. 3.

Vector Quantization [1–4]

Unlike Scalar Quantization, which is a single-sample mapping to a specific single value, Vector Quantization maps a block of data points in a multidimensional space. This mapping is performed based on optimizing some kind of distance measure such as Euclidean Distance or other some performance metric. While sample values themselves may be grouped into frames and quantized using Vector Quantization, a more common application is in quantizing the speech model parameters generated by an encoder. Thus, in the case of typical Vocoders, the LPC coefficients or LSF values may be mapped into a suitable multidimensional space. Once a set of values ("vectors") has been mapped into the vector space, the corresponding vector indices can be transmitted instead of the values themselves; this results in a reduction in the overall amount of data needed to represent a speech frame.

Most practical Vocoder algorithms utilize a predefined set of reference vectors known as a Vector Codebook. This is especially true for coders based on the Code Excited Linear Prediction (CELP) methodology such as the open-source Speex Vocoder algorithm. In CELP-based coders, both the excitement and LPC parameters are quantized based on Vector Codebook entries. For a given set of input vectors, the Codebook is searched and only the index of the nearest vector needs to be stored in the encoded data, thus further reducing the data size and increasing the effective Compression Ratio.

In general, this Codebook search process is very demanding from a computational perspective. Since the matching of the vector to the codebook entries often involve the computation of a Euclidean Distance (described further in Chap. 10), it follows that CPU architectures that have some form of built-in instruction support for computing Euclidean Distance would be able to perform a codebook search more efficiently. Moreover, codebooks consume large amounts of memory. Since codebooks are often constant tables, it is preferable to store them in program memory (thereby saving scarce RAM resources on many processors), assuming the CPU has an efficient and seamless means of accessing data from program memory.

In many Vocoder algorithms, the Vector Codebook is not simply multidimensional but rather structured in ways that optimize the memory usage and search time. A common scheme is to split the Codebook into multiple stages with only a subset of vector dimensions represented in each stage. The bits used to represent the Codebook index is also split across the different stages. This mechanism is known as Multistage Vector Quantization, or simply MSVQ, and results in large memory savings in storing the Codebook.

The usage of MSVQ structures allows for several options to optimize the Codebook search procedure, and this is indeed a significant area of ongoing research. The most basic search procedure is a Full Search, which requires the most amount of computations since the number of Euclidean Distance computations and comparisons is $(N_1 N_2 \cdots N_K)$, where N_1, N_2, etc. are the number of vectors in each stage. This method is the most optimal because all possible vector combinations are being considered. The next method is a Sequential Search, which sequentially finds the best-fit for each dimension. This method requires $(N_1 + N_2 + \cdots + N_K)$ Euclidean

Distance computations and comparisons. However, this strategy is suboptimal. An intermediate technique is Tree Search, in which more than one index is passed between successive stages. The computations needed for Tree Search is not as high as Full Search but not as low as Sequential Search and is one of the preferred search methods used in many Vocoders.

Several other improvements are possible, both in the Codebook structure as well as in the search algorithm. The Vector Quantizer (VQ) can be placed in a closed-loop configuration similar to the one used for Differential PCM, resulting in a methodology known as Predictive VQ. Also, the vector can be split into smaller vectors with different Codebooks, resulting in simpler computation and less memory usage. A popular technique to make the Vector Quantizer more resilient to channel noise is to use two independent Codebooks such that the output code-vector is the sum of the two individual code-vectors. This scheme is known as Conjugate VQ.

Speex Speech Coder [8]

Several ITU, mobile communication, and other Vocoder standards exist for low bit rates, such as the ITU-T G.729, G.728, and G.723.1 standards. However, a common feature of most of these algorithms is that they involve significant royalty costs. One exception to this trend is the open-source Speex algorithm, which is based on the CELP algorithm. Please note that in many applications that require some kind of interoperability with other units (e.g., a Voice-over-IP phone), it may be necessary to use a standardized speech coding technique; however, for proprietary or closed systems (e.g., a pair of walkie-talkies or a digital voice recorder), an open-source algorithm is generally perfectly acceptable and provides significant cost benefits. In general, the Speex Vocoder is suitable for a wide variety of communication and voice record/playback applications.

The Speex algorithm supports a wide variety of output data rates, including a variable bit-rate mode, wherein the output data rate varies to maintain a prespecified level of speech quality. Additionally, this algorithm supports three different input-sampling rates – 8 kHz (narrowband mode), 16 kHz (wideband mode), and 32 kHz (ultra-wideband mode). Very high Compression Ratios are supported. For instance, for a 16-kHz input sampling rate the output data rate can be 9.8 Kbps, a Compression Ratio of 26:1. The input-speech samples are processed in blocks of 160 samples in narrowband mode and 320 samples in wideband mode.

For Short-Term Prediction, a 10-coefficient Linear Predictor is computed using the Levinson–Durbin recursive technique, and the resulting coefficients are transformed into Linear Spectral Pairs for encoding using Vector Quantization. For Long-Term Prediction, a three-coefficient Linear Predictor is used (for delays of $T - 1$, T, and $T + 1$ samples relative to the current sample). Thus, Speex uses an integer pitch period and a fixed Codebook. This pitch prediction is performed for each subframe (with the number of subframes depending on the mode selection), while the Short-Term Prediction is performed once for an entire frame. The final excitation signal is the sum of the pitch prediction and a parameter searched from a fixed codebook known as the Innovation Codebook. This codebook entry represents the residual information that the linear and pitch prediction failed to capture. Perceptual Enhancement is also performed.

If Wideband Mode is used, then the signal is split into two equal bands using a Quadrature Mirror Filter, similar to the G.722 subband structure described in Chap. 7. The upper band is allocated less bits in the encoded data structure, as there is no pitch prediction performed for the upper band. This does not degrade speech quality, as voiced speech is predominantly concentrated in the lower subband. The lower subband is encoded exactly as it would be if the Speex Vocoder were used in Narrowband Mode.

The Speex algorithm also provides an optional Voice Activity Detection (VAD) feature, which can reduce the number of bits used to encode intervals that do not contain any speech. For example, nonspeech frames can be encoded using 5 bytes instead of 20 bytes – just enough to reproduce the background noise. This feature should not be used when deterministic bit rates are required, for example in communications between two walkie-talkies.

The computational and memory requirements depend greatly on the specific mode of operation selected, but are generally higher than any of the Waveform Coders, which we studied in Chap. 7. As an example, the Wideband Speex Speech Coder consumes around 20 MIPS for encoding and 5 MIPS for decoding on a dsPIC33F DSC device. Note that since the encoder includes a closed-loop Analysis-by-Synthesis structure to minimize the prediction errors, the encoder already includes all computations that the decoder would normally perform; as a result, a Speex encoder requires substantially higher MIPS than a Speex decoder.

ITU-T G.728 Speech Coder [5]

The G.728 Vocoder is based on a technique called Low Delay CELP (LD-CELP) and produces an output data rate of 16 Kbps for a standardized input sampling rate of 8 kHz, thereby providing an 8:1 compression ratio. A key characteristic of the G.728 algorithm is the very small algorithmic delay (unlike many other vocoders such as CELP) due to the very short processing frame length of 20 samples. This, of course, comes at the cost of increased computational complexity (since various computations are executed more frequently) and relatively low compression ratio. To avoid the high overhead of computing autocorrelation when the frame sizes are small, recursive methods such as the Chen window method are used.

Since the frames are only 20 samples long, the LPC estimation can be performed directly on the samples from the previous frame; therefore, as soon as a subframe of 5 samples is available the LPC encoding can proceed rather than waiting for the entire frame to be buffered. Moreover, since the LPC coefficients are computed from the previous frame, the coefficients need not be transmitted to the decoder, as the decoder already has the data from the previous frame and can compute the LPC coefficients by itself. Thus, G.728 uses a Backward Adaptive Linear Prediction scheme.

To minimize the output data rate, the G.728 avoids the Long-Term Predictor altogether. Instead, it employs a relative high-order short-term synthesis filter with 50 coefficients. This is not only sufficient for estimating pitch information for both male and female speech, but also makes the algorithm less dependent on the specific properties of the vocal tract and therefore provides better performance than other coders if nonspeech signals such as musical tones or chimes are involved. Moreover, since no LPC coefficients are encoded, using a higher-order prediction filter does not increase the bit rate. The coder also includes a tenth-order prediction filter for the Excitation Gain in logarithmic form. Lastly, the Vector Codebook consists of a 7-bit Shape Codebook and a 3-bit Gain Codebook; by separating the gain into a different Codebook, computational complexity is reduced.

Due to the more frequent processing of frames, the G.728 Vocoder requires more MIPS than the other coders discussed in this chapter. For example, execution on typical fixed-point DSP devices might require up to 30 MIPS.

The block diagrams of the G.728 encoder and decoder are illustrated in Figs. 8.1 and 8.2, respectively.

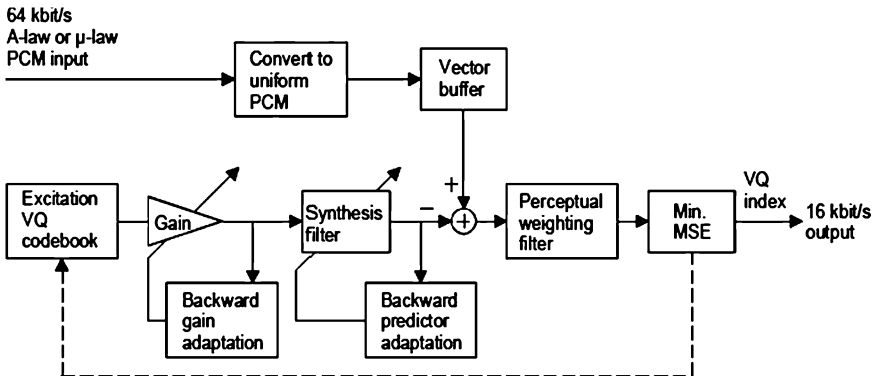


Fig. 8.1 Block diagram of G.728 LD-CELP speech encoder (taken from ITU-T G.728 recommendations)

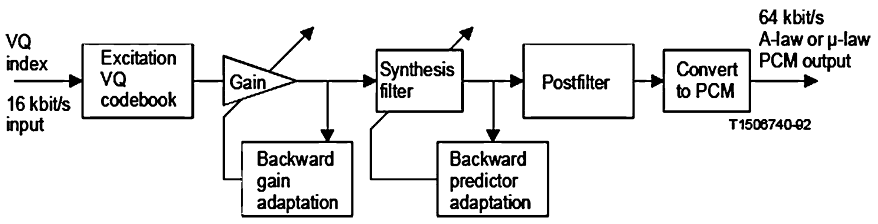


Fig. 8.2 Block diagram of G.728 LD-CELP speech decoder (taken from ITU-T G.728 recommendations)

ITU-T G.729 Speech Coder [6]

The G.729 Vocoder is based on a technique called Conjugate Structure Algebraic CELP (CS-ACELP) and generates an 8-Kbps output data rate for an input sampling rate of 8 kHz. Thus, this coder produces a 16:1 compression ratio. The primary motivation of this algorithm was to reduce the computational burden associated with the code search procedures incurred by other CELP variants. Instead of storing the full Vector Codebook in memory, G.729 utilizes simple algebraic computations such as additions and data-shifts to implement the excitation code-vectors at run-time. This approach results in considerable computational savings compared to conventional CELP algorithms.

The algebraic codebook structure used by the G.729 algorithm is based on a principle called Interleaved Single-pulse Permutation. The excitation code-vector consists of four pulses interleaved in time and having a value of $+1$ or -1 at any given codebook position. A code-vector has 40 such positions, corresponding to the length of a subframe. The code-vector is computed in run-time by adding the four pulse sequences, as follows:

$$v[n] = \sum_{i=0}^{3} s_i \delta[n - m_i] \quad \text{where } n = 0, 1, \ldots, 39. \tag{8.14}$$

Seventeen bits are needed in all for indexing into the codebook, including 4 sign bits and 13 position bits. This codebook methodology is also known as Adaptive Codebook. For the pitch period, 8 bits are used to encode the pitch during the first subframe, whereas for subsequent subframes only pitch differences are encoded with 5 bits, resulting in a reduced data rate. A special feature of the G.729 algorithm (unlike, say, Speex) is that fractional pitch period values are also supported.

Simplified block diagrams of the G.729 encoder and decoder are shown in Figs. 8.3 and 8.4, respectively.

G.729 Annex A specifies some techniques to reduce the computational complexity of the G.729 coder. These include using decimation during the open-loop pitch analysis and using a tree codebook search, among other changes. Note that the G.729A coder is completely backward-compatible with the base G.729 standard.

The computational requirements and decoded speech quality (MOS) of the G.729 Vocoder are similar to those of the Speex Vocoder operating in Narrowband Mode. Thus, G.729 and Speex can be considered alternative to each other in applications that require a high compression ratio such as 16:1, with the ultimate choice dictated by interoperability and cost requirements of the application.

ITU-T G.723.1 Speech Coder [7]

The G.723.1 Dual-Rate Speech Coder follows opposite requirements from the G.728 Vocoder. Whereas the G.728 coder traded off output data rate for the sake of very low algorithmic delays, the G.723.1 has the highest delay among all the coders

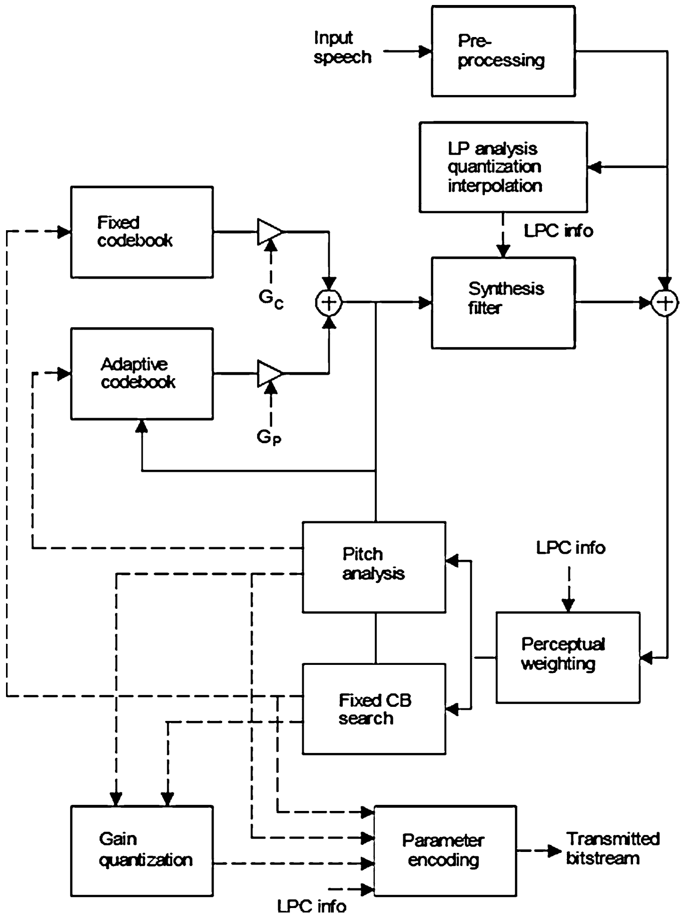


Fig. 8.3 Block diagram of the G.729 CS-ACELP speech encoder (taken from ITU-T G.729 recommendations)

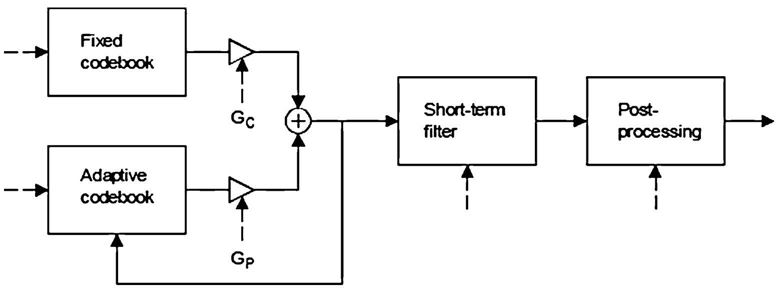


Fig. 8.4 Block diagram of G.729 CS-ACELP speech decoder (taken from ITU-T G.729 recommendations)

discussed in this chapter: 67.5 ms. However, the output data rate is the lowest: it supports two bit rates, 5.3 Kbps and 6.3 Kbps. These characteristics make the G.723.1 coder a good candidate for applications such as walkie-talkie and videophones in which minimizing the bit rate is a more critical requirement than having low delays.

The operating principle of this code is similar to the others discussed in this chapter: it uses an analysis-by-synthesis LPC methodology. A 5-tap pitch synthesis filter is used and the coefficients are encoded using Vector Quantization, while the excitation gain is encoded using Scalar Quantization. Even subframes are encoded with 7 bits per subframe, while for odd subframes only the difference is encoded using only 2 bits per subframe. Like G.729, the codebook structure used in G.723.1 is ACELP. Out of the 17 bits used to index the codebook in the 5.3-Kbps mode, there are 4 sign bits, 12 position bits, and one special bit called Grid that distinguishes between the even and the odd subframes. For the 6.3-Kbps mode, the number of pulses and bits used to index the codebook are higher than for the 5.3-Kbps mode.

Due to the higher compression ratio, the overall computational requirements of the G.723.1 Vocoder are slightly higher than both G.729 and Speex. However, it is not as demanding as the low-delay G.728 Vocoder. For speech encoding applications that require an output data rate of less than 8 Kbps, particularly for communication applications that suffer from degraded channel quality at higher bit rates, the G.723.1 speech coder is perhaps the only viable option.

Summary

In this chapter, we have studied various commonly used techniques to encode speech signals based on modeling key parameters of the human speech production system. Such Voice Coder algorithms provide greater compression ratios than any Waveform Coders, albeit at the cost of greater computation and slightly reduced decoded speech quality. In spite of the computational requirements, these algorithms provide tremendous benefit for many embedded applications, especially those that are memory constrained or use limited-bandwidth speech communication channels. Advances and optimizations in CPU architectures and greater adoption of DSP and DSC platforms will greatly enhance the practical usability of these algorithms even in low-cost embedded control applications that need to compress human speech.

References

1. Rabiner LR, Schafer RW Digital processing of speech signals, Prentice Hall, 1998.
2. Chau WC Speech coding algorithms, Wiley-Interscience, 2003.
3. Spanias AS (1994) Speech coding: a tutorial review. Proc IEEE 82(10):1541–1582.
4. Sinha P (2007) Speech compression for embedded systems. In: Embedded systems conference, Boston, MA.

5. ITU-T Recommendation G.728: Pulse code modulation (PCM) of voice frequencies.
6. ITU-T Recommendation G.729: 7 kHz audio-coding within 64 kbit/s.
7. ITU-T Recommendation G.723.1: 40, 32, 24, 15 kbit/s adaptive differential pulse code modulation (ADPCM).
8. Valin JM (2007) The Speex Codec Manual – Version 1.2 Beta 3.

Chapter 9
Noise and Echo Cancellation

Abstract One of the key requirements in most real-life applications involving speech signals is that the speech should be devoid of unwanted noise and other interferences. Most embedded applications operate in nonideal acoustic environments. Ambient noise can often severely impair the quality of the speech signal, to the point that a person hearing it may not even be able to understand the content of the speech. If the speech is further processed by processing algorithms such as speech compression, these algorithms could potentially provide suboptimal performance when the input speech is noisy. Tasks such as speech recognition have even more stringent requirements for the quality of the speech input, and are particularly sensitive to background noise interference. To some extent such noise can be filtered using regular digital filtering techniques, but in many cases the noise does not follow a deterministic frequency profile and might even change dynamically. Such situations call for more advanced algorithms that can adapt to the noise and remove it. Besides noise, echo generated due to various electrical and acoustic reasons can also be a significant cause of signal corruption and must be eliminated. This chapter explores various techniques to reduce or eliminate noise and echo from speech signals, and looks at real-life applications where such algorithms are critical.

Benefits and Applications of Noise Suppression [4–8]

The term Noise Suppression refers to a class of filter algorithms that can be used to substantially reduce the effect of unwanted additive noise that may be corrupting a desired speech signal. For example, a real-life application might need to capture speech uttered into a microphone by a person. Now, if the microphone is located such that there is a large amount of background noise, then this noise would also be captured by the microphone. This would be especially significant if the microphone is relatively omni-directional, as the noise sources from all directions could possibly drown out the speech spoken from one direction. Once the noise is sampled by the system (e.g., using an ADC) along with the speech, subsequent processing stages would not operate reliably, and listeners of the speech would have a very negative listening experience and probably not understand the content of the speech.

In some scenarios, the noise might be from a single source and have a very specific frequency. For instance, a single tone (i.e., "narrowband noise") coming from some kind of machinery located nearby would have a very deterministic frequency. Such types of noise are easy to eliminate using a highly selective band-stop filter known as a Notch Filter. However, in most situations, the source of the noise may not be known, or even if it is, its frequency content may be wideband or even dynamically varying over time. Also, most systems need to be designed to be able to deal with a variety of conditions, as it is generally not practical to customize a product's software or hardware to suit each possible set of conditions it has to operate it. Therefore, it is necessary to employ sophisticated algorithms that can dynamically track the profile of noise being encountered by the system and suppress it; hence the need for a variety of such Noise Cancellation algorithms.

Noise Cancellation algorithms are useful for a variety of end-applications. The most common applications are those that require some sort of live speech input, either through a microphone or through a noisy communication channel. Some application examples are listed below:

- Mobile Hands-Free Kits – Mobile Hands-Free Kits are a mobile phone accessory that allows users to carry on a conversion using a mobile phone without having to physically hold the cellphone or talk into the microphone located on the handset itself. Instead, a Hands-Free Kit device containing a separate microphone and speaker (connected to the handset) is used for talking and listening. In many nations and states, it is mandatory for car drivers to use a Hands-Free Kit (HFK). Many Hands-Free Kits are designed to use the Audio System of the car instead of separate speakers. Also, many Hands-Free kits do not require the mobile phone to be directly connected to it; rather, communication of speech signals with the cellphone is conducted through an industry-standard wireless communication interface known as Bluetooth®. In all these systems, the acoustic environment inside a car is very noisy, mainly as a result of road noise and the noise generated by other cars on the road. Hence it is difficult to carry on a conversation, as the speech captured by the HFK microphone is usually heavily mixed with background noise. It is, therefore, absolutely necessary for the HFK processing software to include some kind of Noise Cancellation to remove (or at least greatly suppress) the effect of this noise.
- Speakerphones – Speakerphones are also susceptible to noise being captured by the microphone (though to a lesser degree than an HFK system), especially since the near-end talker may not be physically very close to the phone unit. High-end speakerphones can definitely differentiate themselves from competitive solutions if they incorporate effective Noise Cancellation algorithms.
- Intercom Systems – Intercom systems, such as the ones in hallways of office buildings, factories, hospitals, and airports, as well as those intended for door security and access control in large apartment buildings and offices, have to operate under a variety of ambient noise environments and therefore benefit from Noise Cancellation. In the case of intercom systems for access control to a building or complex, the noise may be a combination of people talking on the street

and noise from street vehicular traffic; if the intercom is located in a hallway, the noise is exclusively that of a large number of people talking simultaneously (also popularly known as "babble noise").

- Medical Emergency Communication Units – Portable (and generally wearable) communication devices used for transmission of help requests in the case of medical emergencies can also benefit from reduction of ambient noise, as this is a life-critical application in which it is necessary to be able to quickly communicate the required medical and location information to the emergency response personnel.
- Hearing Aids – Aids for hearing-impaired persons provide a substantial amount of amplification, and when there is a lot of background noise in a particular location this may result in an unpleasant auditory experience for the user of the hearing aid. Therefore, such systems benefit greatly from a reduction in noise, while keeping desired speech and other audio signals unaffected and distinct.
- Walkie-Talkies – Walkie-talkie or 2-way Digital Radio systems are often used in noisy environments, for example, workers in an industrial facility, technicians performing repairs or maintenance on machinery, policemen at a crime scene, etc. Noise Cancellation algorithms provide a great tool to enhance the speech communication through such devices.
- Amateur Radios – Amateur radios and other such communication devices receive speech signals from remote sources, often through relatively noisy communication channels. In such applications, it is very useful to remove the effect of noise that may have got mixed with the desired signal during communication.
- Bluetooth® Helmets for Motorcycle Drivers – Drivers of motorcycles, snowmobiles, and other open-air vehicles might require an alternative to HFK systems. In these applications, the helmet might contain a microphone and speaker and directly communicate with a cellphone placed anywhere in the motorcycle. Such vehicles tend to be noisy by themselves, not to mention the road noise that is very distinct as there is no vehicle enclosure; therefore, Noise Cancellation algorithms are practically a necessity for such systems.
- In general, any application that involves capturing speech through a microphone can be enhanced by incorporating Noise Cancellation algorithms.
- Sometimes, the noise might not be directly mixed with the speech signal itself but may, nevertheless, be audible to the listener. For example, in a Noise Cancelling Headphone, the main audio channel may be directly derived from a radio or CD player and the noise may simply be ambient noise entering the listener's ear directly. This class of applications (as well as those in which the objective is simply to remove ambient noise from a particular area) calls for a type of closely related algorithms known as Active Noise Cancellation.

The general concept of a Noise Suppression system using a dedicated software library is shown in Fig. 9.1.

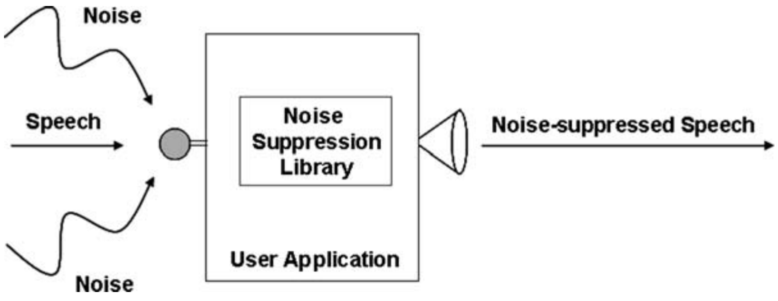


Fig. 9.1 General concept of a noise suppression system

Noise Cancellation Algorithms for 2-Microphone Systems

The most intuitive approach to cancelling the effect of noise from speech signals may be to separately capture the noise from its source and simply subtract this noise from the noise-corrupted speech signal. This is, of course, possible only if we have access to the noise by itself.

Spectral Subtraction Using FFT [1, 7]

In scenarios where the source of the noise is very close to the source of the desired speech signal (i.e., the Primary microphone), the acoustic modification of the noise in the path from its source to the Primary microphone may be negligible. In this case, a simplistic Frequency Domain methodology may be employed. Periodically, an FFT can be computed for the corrupted signal as well as for the "pure" noise reference signal (captured by a second microphone known as the Reference microphone). Then, the FFT of the noise reference can be subtracted from that of the "speech + noise" signal, thereby yielding the FFT spectrum of a relatively purified speech signal. The Inverse FFT of this difference would then produce a relatively noise-free speech signal; essentially the noise is being "subtracted" from the noisy speech signal.

Adaptive Noise Cancellation [1, 7]

Although the above methodology may be elegant and easily-understood, it is generally not viable in practical applications. This is because there is an acoustic path between the noise source and the Primary microphone, which modified the characteristics of the noise much as a linear filter would do. Therefore, this acoustic path needs to be effectively modeled if the reference noise signal can be truly "subtracted" from the noisy speech signal. Also, this acoustic path might vary over time, resulting in the effect of the noise on the Primary microphone varying dynamically. Therefore, whatever model we choose to use for the acoustic path between the Reference microphone and the Primary microphone should not be a fixed model (e.g., an FIR or IIR filter with constant coefficients). Essentially, an Adaptive Filter will need to be employed.

At this point, a brief overview of Adaptive Filters would be appropriate, as such filters provide the basis for not only Adaptive Noise Cancellation algorithm such as we are discussing here but also the Echo Cancellation algorithms that will be discussed later in this chapter. An Adaptive Filter, as the name suggests, is a digital filter whose coefficients are not constant. Rather, the coefficient values are periodically (and frequently) updated based on changes in environmental parameters and the characteristics of the input data. The adaptation algorithm mainly refers to the process and criteria used in updating the coefficients, whereas the filter itself may utilize any of the FIR or IIR filter topologies studied in previous chapters. Similarly, the filtering itself can be performed either in the time domain or in the frequency domain (i.e., operate on the FFT of the inputs rather than the inputs samples themselves).

Several criteria exist for determining the new values of the filter coefficients (given their current values as well as the input data sets), such as Least Mean Squares (LMS) and Recursive Least Squares (RLS). The RLS technique, while actually being one of the most effective and robust adaptation methods, is computationally prohibitive for embedded systems. Hence, I will limit my description of Adaptive Filters here to the LMS technique and its variants.

The basic structure of an LMS Adaptive Filter is shown in Fig. 9.2. An FIR structure has been chosen for its simplicity, numerical stability, and popularity. According to the LMS adaptation criterion, for every new input sample $x[n]$, the filter coefficients are adapted to minimize the Mean Squared Error (MSE) between a desired or reference signal $d[n]$ and filter output $y[n]$.

In the context of Adaptive Noise Cancellation, the inputs $x[n]$ are the source noise samples captured by the Reference microphone input, the inputs $d[n]$ are the noise-corrupted speech signal samples captured by the Primary microphone, and

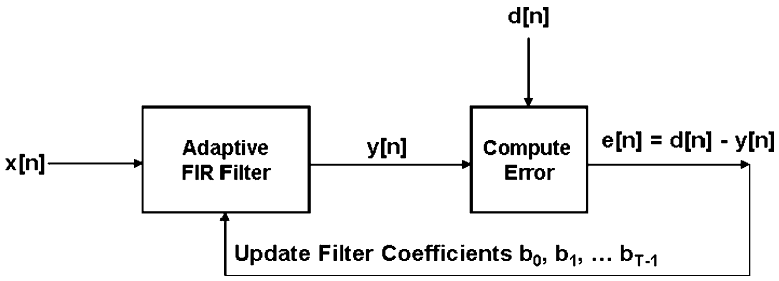


Fig. 9.2 Basic LMS adaptive FIR filter algorithm

the "error" outputs $e[n]$ are the resultant noise-cancelled speech signals that will be provided to any further processing stages in the application. Here, the Adaptive FIR Filter serves as a model for the effective acoustic path the noise traverses between its source (as captured by the Reference microphone) and its destination (the Primary microphone). For this reason, the Adaptive Filter may also be considered a system identification problem.

Thus, if the acoustic path has been modeled perfectly, the filter output corresponds to the noise as it would appear at the point where it gets mixed with the desired speech signal. The process of adapting the filter coefficients, therefore, tries to minimize the difference (mathematically speaking, the mean-squared difference) between the noisy speech signal and the filtered noise, and when the filter coefficients have finally converged to their optimal values after several iterations of the algorithm, the error signal effectively becomes the desired speech signal with the effect of the noise removed from it.

A simplified list of steps for the LMS Filter algorithm is given below:

- Perform FIR filtering.

$$y[n] = b_0 \times x[n] + b_1 \times x[n-1] + \ldots + b_{\mathrm{T}-1} \times x[n-(\mathrm{T}-1)] \tag{9.1}$$

- Compute error between reference signal and filter output.

$$e[n] = d[n] - y[n] \tag{9.2}$$

- Update filter coefficients for next iteration.

$$b_k = b_k + \mu\left(e[n] \times x[n-k]\right), \tag{9.3}$$

where T is the number of filter coefficients, $k = 0, 1, 2, \ldots, T-1$, μ is the Adaptation Coefficient (Step Size).

From the above equations, it becomes apparent that the Adaptation Coefficient is a critical determining factor in how quickly the filter coefficients (and therefore the model of the acoustic path of the noise) converge to their final values. On the one hand, if μ is selected to be too small, the filter will take a long time to converge, and the algorithm will not be able to respond to rapid changes in the characteristics of the data or the acoustic path. In such cases, the Mean Squared Error might never reach its optimal value, thereby causing some amount of residual noise to remain in the signal for a long time. On the other hand, if μ is selected to be too large, the filter adaptation will occur very fast, but the Mean Squared Error will oscillate around its optimal value without settling in at one final value. Thus, the noise will get cancelled quickly but will keep reappearing. If μ is increased even further beyond limits (LMS and other adaptation schemes do provide some limits to the step size based on the signal energy), the filter operation will be unstable.

Unlike the scenarios described here, there are many applications, for example, most telephony or mobile HFK applications, wherein it is simply not possible to isolate the noise in such a way that a reference noise source can be obtained. For such applications, a 2-microphone solution is completely ineffective, and a 1-microphone solution that does not require a reference noise source must be utilized.

Noise Suppression Algorithms for 1-Microphone Systems [4]

For applications in which it is not practical to obtain an isolated noise reference signal, the noise profile must be obtained from the sampled (noisy) speech frames sampled by the Primary microphone, and the Reference microphone is eliminated from the system. Every sampled frame would need to be analyzed to determine whether it contained any valid speech signals (a common operation in Speech Processing, known as Voice Activity Detection, or VAD). If it did not contain speech, the current frame is treated as a "noise-only" frame, and is utilized to update a running estimate of the characteristics of the ambient noise. In other words, a Noise Profile internal to the algorithm is updated whenever a frame containing only noise (or silence, if there is no significant noise present) is detected.

A typical example of such a Noise Cancellation algorithm, the dsPIC® DSC Noise Suppression Library from Microchip Technology, is illustrated in Fig. 9.3. I will henceforth refer to this class of algorithms as Noise Suppression algorithms, since there is no explicit cancellation taking place (unlike in the 2-signal reference-based systems we have seen so far).

The speech signal, typically sampled by a microphone (or in some cases, received through a communication channel) is filtered to remove any high-frequency noises as well as any DC components or power-supply (50 or 60 Hz) noise. Then, an FFT

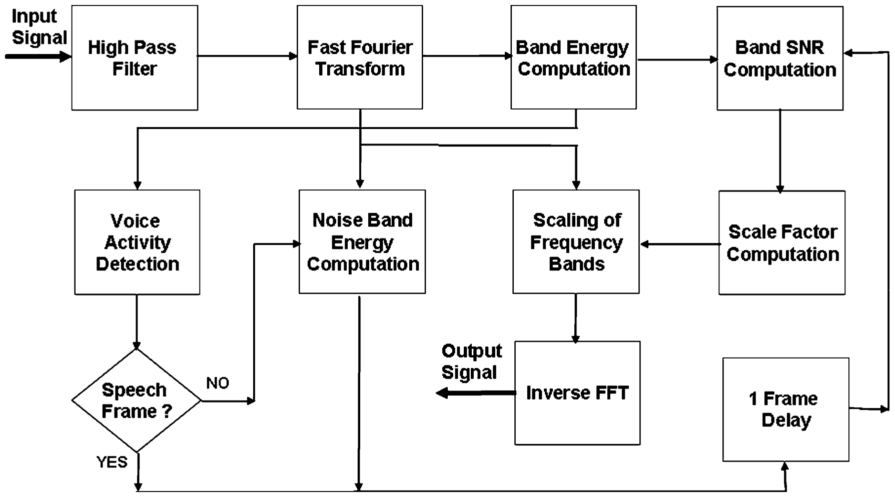


Fig. 9.3 A typical 1-microphone noise suppression algorithm example

is computed; the FFT size and window are chosen such that there is some overlap across successive frames. The signal energy present in a finite number of frequency bands is then computed; these bands (16 bands in this example) are spaced to approximately correspond to the critical bands of the human auditory system. Thus, the frequency bands are nonlinear, with lower-frequency bands being finer (containing a smaller number of FFT frequency bins) and higher-frequency bands being progressively wider (containing a larger number of FFT frequency bins).

The energy in each frequency band is calculated as follows:

$$E_{\text{ch}}(m,i) = \text{Max}\left[E_{\min}, \left\{\alpha \times E_{\text{ch}}(m-1,i) + (1-\alpha) \times \left(\sum G_k^2/N_i\right)\right\}\right], \tag{9.4}$$

where $\alpha = 0.55$ is an empirically-fixed weighting factor, $E_{\min} = 0.0625$, m denotes the index of the current frame, i denotes the index of a frequency band, N_i is the number of frequency bins in frequency band i, G_k is the FFT magnitude of each frequency bin k.

For each of the 16 frequency bands, the Signal-to-Noise Ratio for that band is calculated, based on the following:

- Signal Band Energy, computed as per (9.4)
- Noise Band Energy estimate, updated during periods of speech inactivity

Then based on the Signal-to-Noise Ratio for each band, a Scale Factor is determined on a per-band basis. Thus, the Scale Factor is essentially inversely proportional to the relative amount of noise present in each frequency band. This scale factor is then applied to the FFT outputs that have already been calculated for the current frame (all frequency bins within the same frequency band are scaled by the same amount).

Finally, the Inverse FFT is computed, resulting in a time domain output signal in which each frequency band has been scaled in proportion to how much that particular frequency band had been affected by the ambient noise. Frequency bands with more noise content have been scaled to a greater extent compared with frequency bands with less noise content.

The most critical element of this algorithm is perhaps the VAD, which analyzed every single frame and classifies it as either a Speech Frame or a Noise Frame. There are many alternative methods of performing VAD; however, this example uses a simple but effective policy: it compares the signal energy of all the frequency bands in the current frame with that of the corresponding frequency bands in the Noise Profile. If at least a certain prespecified number of bands (e.g., 5 out of 16) have demonstrated a sudden increase in energy by a certain prespecified factor (e.g., 1.5 times), the current frame is assumed to be containing speech. If not, the Noise Profile energy estimates are suitably updated with the information from the current frame, using (9.4).

More complex algorithms, and more intricate implementations for each of the above steps, are certainly possible. However, the choice of algorithms must be weighed with the computational bandwidth and memory usage in an embedded application.

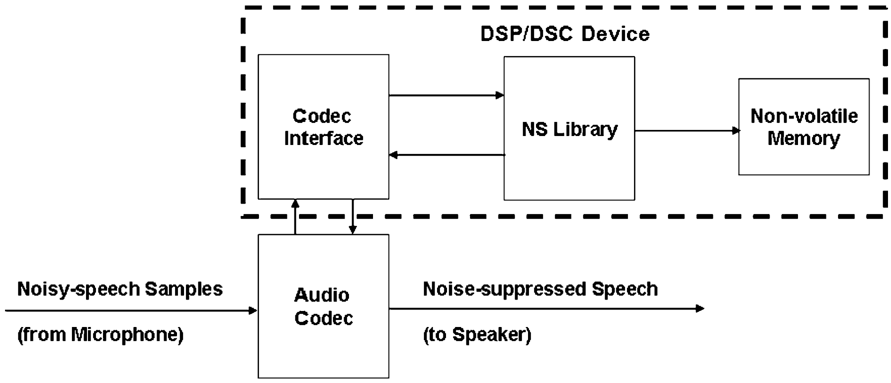


Fig. 9.4 Noise suppression application example – digital voice recorder

The simplified block diagram (Fig. 9.4) shows the usage of a Noise Suppression (NS) Software Library in a Digital Voice Recorder application. The voice of the talking person is captured by a microphone and converted into digital samples by an external Audio Codec. A DSP or DSC device interfaces with the codec through a dedicated codec interface, receiving the potentially noisy speech samples and also playing out the recorded speech as needed. Input samples are passed through the NS Library, and the resulting noise-suppressed speech samples are stored in internal program memory or to some external nonvolatile memory device.

Active Noise Cancellation Systems [1,7]

Active Noise Cancellation systems are those in which the objective is simply to cancel or reduce the ambient noise in a certain area. The noise in such application is typically not mixed with speech in an electrical sense; rather, it might simply reach the ear of a person who may be talking over a phone or listening to music on a headphone, that is, the noise can be consider as “mixed” with speech only in an acoustic sense.

The algorithms used for such applications are somewhat similar to those used in 2-Microphone Noise Cancellation systems. Adaptive Filters such as LMS are the most prevalent software technique. However, in this case, since the effect of the noise is not mixed with the speech samples, the “subtraction” is often effected in the acoustic domain: the Adaptive Filter output is used to generate an “Anti-Noise” sound that is played out through a speaker to acoustically nullify the sound coming from the noise source. Another key difference is that there is no Primary microphone in these applications; rather, an Error microphone is placed downstream to capture the residual noise (which of course should ideally be zero), so that the filter coefficients can be updated as per the LMS criterion. An example of such a system is shown in Fig. 9.5 for a Fan Noise Cancellation application.

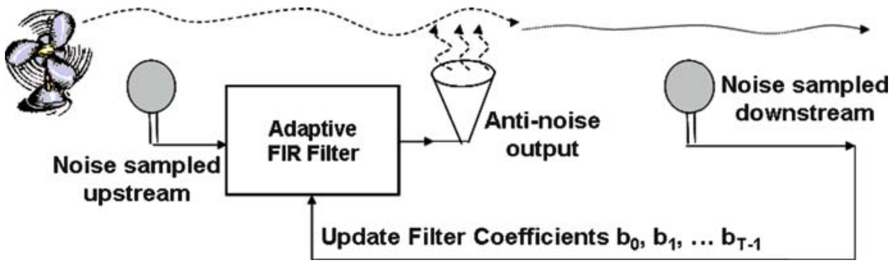


Fig. 9.5 Example of an active noise cancellation application – fan noise cancellation

Note that in certain applications such as a Noise Cancelling Headphone, the Antinoise speaker can be eliminated by incorporating the antinoise signal into the audio signal fed into the headphone.

Benefits and Applications of Echo Cancellation [1, 5, 6]

Many Speech Processing applications involve both a microphone and a speaker, that is, speech played out as well as speech captured as input. Many of these applications also happen to be communication applications such as Mobile Phones, Speakerphones, and Intercom Systems. Now, if the speaker is located in proximity to the microphone, it is common for part of the sound waves coming out of the speaker to impinge on the microphone, thereby reentering the signal chain as a feedback signal.

If this signal is then transmitted to some other remote device (e.g., the other end of a telephone conversation), the listener at the other end of the communication link would perceive this feedback as a distinct echo. In general, the effect of this echo would always be experienced at the other end of the communication link. To make matters worse, there could be multiple reflection paths between the speaker and the microphone, making it even more challenging to remove this echo. Echo cancellation algorithms strive to remove this echo at the transmitting node itself.

The general concept of Acoustic Echo Cancellation (AEC) is illustrated in Fig. 9.6. A portion of the incoming far-end speech R_{IN} enters the microphone along with the valid near-end speech, resulting in a sampled signal S_{IN} that is a combination of the near-end speech and the acoustic echo. The AEC algorithm, typically in the form of a software library, operates on this S_{IN} signal, so that the transmitted signal S_{OUT} only contains the near-end speech with the echo portion removed to a large extent.

A closely related problem is that of Line Echo Cancellation (LEC). In the case of Line Echo, the source of the echo is not acoustic feedback occurring between a speaker and a microphone located close to each other. The echo in this case is caused by electrical reflection at telecommunication circuits such as 2-to-4-wire

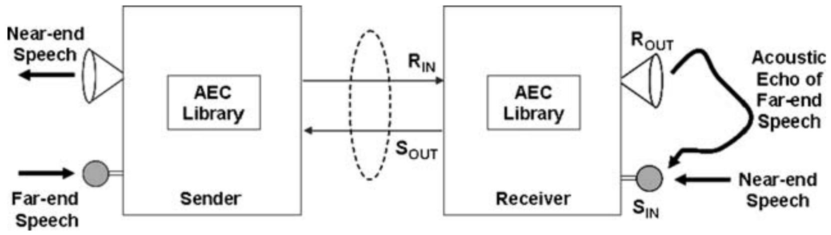


Fig. 9.6 General concept of an acoustic echo cancellation system

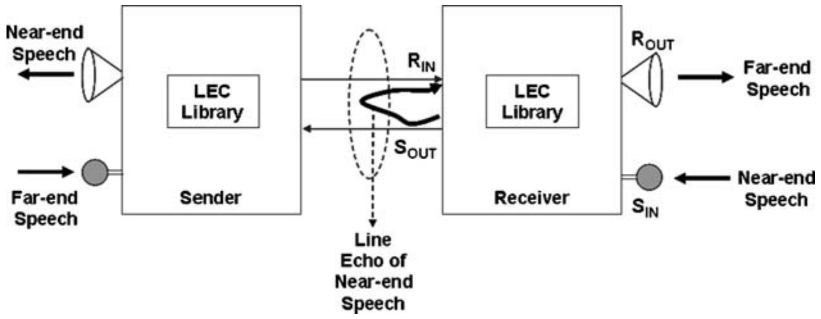


Fig. 9.7 General concept of a line echo cancellation system

hybrid as well as devices such as routers and switches in a Voice-over-IP or wireless communication network. Thus, the echo is generated at the communication-interface side of the system, rather than the audio-interface as was the case with Acoustic Echo. The algorithms used for LEC are very similar to those used for AEC, except that the echo cancellation is performed on the received far-end speech signal rather than the transmitted near-end speech signal (Fig. 9.7).

As noted already, the echo may be generated by multiple reflection paths. The Acoustic or LEC algorithm needs to be able echo delays corresponding to the maximum possible echo (acoustic or electrical) path present in the application. This maximum delay is known as the Echo Tail Length, and is a key parameter that defines the capabilities of the algorithm.

Both AEC and LEC are associated with specific International Telecommunications Union (ITU) recommendations: these standards are G.167 and G.168, respectively. The standards define the performance and testing requirements of Echo Canceller systems.

Many of the applications that benefit from AEC and/or LEC are the same ones that benefit from Noise Suppression. However, not all applications that use NS might include a speaker, and therefore do not require AEC. Also, not all applications involve any form of communications; hence, only some of those applications really require AEC. Some important ones are listed below:

- Voice-over-IP Phones

- Analog Telephone Adapters
- Speakerphones
- Intercom Systems
- Public Address Systems
- Medical Emergency Communication Devices
- Security Panels
- Mobile Hands-Free Kits

Acoustic Echo Cancellation Algorithms [2,5]

The block diagram of a typical AEC solution is shown in Fig. 9.8.

The core function used for most AEC algorithms is the Adaptive LMS FIR Filter, similar to what has already been discussed for 2-Microphone Noise Cancellation systems. In this class of applications, the reference or desired signal is the near-end (echo-corrupted) speech signal S_{IN}, usually sampled from the microphone. The input to the adaptive filter is the received far-end speech signal R_{IN}, usually received from a communications interface. Thus, the filter models the acoustic path between the speaker and the microphone. The signal flow of the Adaptive Filter algorithm is shown in Fig. 9.9.

In some AEC algorithms, the Adaptation Constant μ may be readjusted for every frame (based on the energy of the most recently-transmitted near-end speech frame) before performing the filter coefficient updates, as in the following:

$$\mu = \frac{1}{0.7324 + E_{S_{\mathrm{OUT}}}[n-1]}. \tag{9.5}$$

Moreover, to avoid problems of numerical stability when updating filter coefficients, most practical AEC implementations use a Normalized LMS Filter. In Normalized

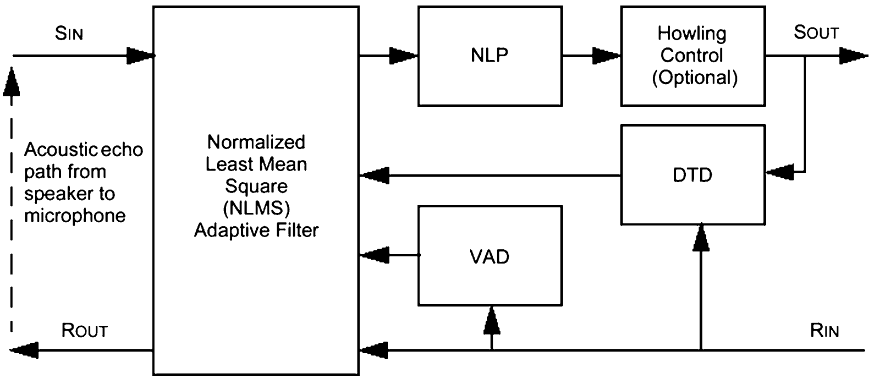

SIN
Acoustic echo path from speaker to microphone
ROUT
Normalized Least Mean Square (NLMS) Adaptive Filter
NLP
Howling Control (Optional)
SOUT
DTD
VAD
RIN

Fig. 9.8 Block diagram of a typical acoustic echo cancellation algorithm

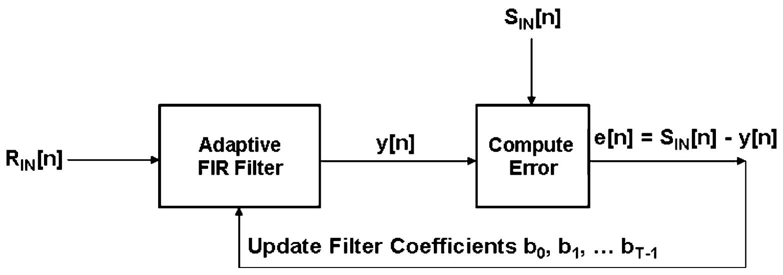


Fig. 9.9 Adaptive LMS FIR filter for acoustic echo cancellation

LMS, the Adaptation Constant is normalized by the average received far-end energy for the current frame. Since this division can cause divide-by-zero errors, a small constant is added to the denominator, as shown below:

$$\mu_{\text{normalized}} = \frac{1}{\delta + E_{\text{R}_{\text{IN}}}[n]}. \tag{9.6}$$

The order of the LMS FIR Filter is directly proportional to the Echo Tail Length. Hence, the choice of a filter size is often a trade-off between the available memory and computational speed of the processor and the maximum echo delays the algorithm can eliminate. The Echo Tail Length requirement for any particular application is determined by its acoustic environment as well as the loudness of the speaker output.

Before Adaptive Filtering, all signals are passed through a band-pass filter to remove power-line and high-frequency noise, just as explained for NS algorithms. The VAD function is identical to the one used in the NS algorithm, except here its only role is to inhibit the adaptation of the LMS Filter when there is no speech present in the received far-end speech signal (since no echo would be generated when there is no received speech anyway).

A simple Double Talk Detector (DTD) function compares the far-end output (signal going to the speaker) and near-end input (signal sampled from the microphone) to detect double-talk scenarios and prevent LMS Filter adaptation in frames during which double-talk is present. This simplistic approach is based on the assumption that the echo of the far-end signal would always have lower signal energy than the corresponding original far-end signal. DTD helps to prevent unwanted divergence of the filter coefficients during frames when the microphone input contains not only echo but also live near-end speech.

The linear Adaptive Filter used in this algorithm is generally not capable of modeling any nonlinearity in the acoustic path between the speaker and the microphone. For this reason, a Nonlinear Processor (typically a simple attenuator) is inserted in the signal transmission path to suppress any residual echo that may be present.

Finally, some AEC algorithms support an additional capability called Howling Control. Howling, a common phenomenon in Public Address systems and many other high-power applications, occurs when the feedback between the speaker and

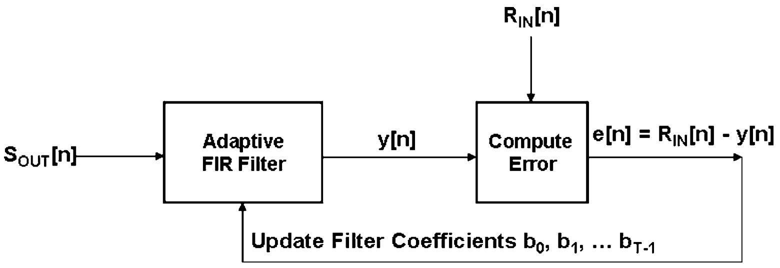

$R_{IN}[n]$
$S_{OUT}[n]$
Adaptive FIR Filter
y[n]
Compute Error
$e[n] = R_{IN}[n] - y[n]$
Update Filter Coefficients $b_0, b_1, \ldots b_{T-1}$

Fig. 9.10 Adaptive FIR LMS filter for line echo cancellation

the microphone keeps going back and forth and causes positive feedback to build up, resulting in a howl-like sound. A simple Howling Control mechanism may be a frequency shifter that shifts the frequency of received signals slightly so that after a few feedback loops the signal is removed from the audible frequency range.

Line Echo Cancellation Algorithms [3, 6]

As mentioned earlier, the algorithms used for LEC are very similar to the one I have just described, except that the reference signal is R_{IN} and the input to the Adaptive Filter is S_{OUT} (Fig. 9.10).

Moreover, for some applications Line Echo Cancellers are required to have the ability to detect specific tones with phase reversals, and inhibit the echo canceller in response to it. For example, this is a useful feature in software implementations of modem standards.

Unlike AEC, LEC algorithms do not have any need to implement audio-specific additional features such as Howling Control.

Computational Resource Requirements

The computational resource requirements of NS, AEC, and LEC algorithms ultimately depend on the specific algorithm and configuration being used. We have also seen that several algorithmic blocks are common between the three types of tasks, so there is certainly a scope of sharing common functions in applications that use two or more of these algorithms.

Noise Suppression [4]

Typical 1-Microphone Noise Suppression algorithms using FFT-based techniques, such as the one we have seen, tend to be very optimized and lightweight. For example, the Noise Suppression Library for a dsPIC® DSC requires around 8 KB of program memory, 1.5 KB of data memory, and only 3.5 MIPS to execute. This makes such algorithms very amenable to be used along with other more complex algorithms, such as a Speakerphone application, which might often execute Noise Suppression along with both Acoustic and LEC.

Most 2-Microphone Noise Cancellation algorithms based on the LMS Filter methodology can be expected to have computational requirements similar to those of Acoustic or LEC algorithms.

Acoustic Echo Cancellation [5]

Unlike noise reduction algorithms, there is not too much variation in the types of algorithms used for Echo Cancellation, as the vast majority of algorithms are based on LMS Filters. Moreover, the bulk of the data memory and MIPS requirements of AEC algorithms are consumed by the LMS algorithm, and therefore directly dependent on the filter length. The filter length is in turn dependent on the Echo Tail Length, which in turn depends on the complexity of the acoustic environment in which the application is intended to be operated.

Let us consider a typical AEC implementation. For a 64 ms Echo Tail Length (a very common configuration), the dsPIC® DSC AEC Library requires around 9 KB of program memory, 5.5 KB of data memory, and 16 MIPS.

Line Echo Cancellation [6]

LEC algorithms, being very similar in nature to AEC algorithms and facing the same application considerations stated earlier, have similar computational resource needs. For example, the dsPIC® DSC LEC Library requires around 6 KB of program memory, 10 KB of data memory, and 15 MIPS, metrics that are fairly common across 16-bit DSC/DSP architectures.

Summary

In this chapter, we have explored some exceedingly popular Speech Processing tasks: Noise Suppression/Cancellation, AEC, and LEC. A large base of embedded applications, especially in the telecommunications segment, utilizes various combinations and configurations of these algorithms. Therefore, an efficient implementation of these algorithms is crucial to many systems. Some emerging applications, such as Voice-over-IP Phones and Analog Telephone Adapters, not only execute two or more of these algorithms but also combine them with full-duplex Speech Encoding/Decoding, encryption and communication stack processing, thereby forcing a drive toward greater computational performance and data access efficiency in future processor architecture platforms.

References

1. PM Clarkson Optimal and Adaptive Signal Processing, CRC Press, 1993.
2. ITU-T G.167 Recommendations for Acoustic Echo Cancellers.
3. ITU-T G.168 Recommendations for Digital Network Echo Cancellers.
4. Microchip Technology Inc. dsPIC® DSC Noise Suppression Library User's Guide,
5. Microchip Technology Inc. dsPIC® DSC Acoustic Echo Cancellation Library User's Guide.
6. Microchip Technology Inc. dsPIC® DSC Line Echo Cancellation Library User's Guide.
7. http://www.ti.com – website of Texas Instruments.
8. http://www.analog.com – website of Analog Devices.

Chapter 10
Speech Recognition

Abstract Speech Recognition, or the process of automatically recognizing speech uttered by human users just a human listener would, is one of the most intriguing areas of speech processing. Indeed, it can be the most natural means for a human to interact with any electronics-based system as it is similar to how people receive and process a large proportion of information in their daily lives. The related tasks of Speaker Recognition and Speaker Identification are also vital elements of many embedded systems, especially those in which some kind of secure access is required. This chapter explains the benefits and various applications of speech and speaker recognition, and provides a broad overview of algorithms and techniques that are employed for these purposes. While the subtle details of such algorithms are an extensive area of research and tend to be mathematical, the core concepts will be described here with a view to efficient implementation on real-time embedded applications.

Benefits and Applications of Speech Recognition [2,3]

Speech Recognition, or the automated interpretation of human speech by an electronic system, is one of the most powerful speech processing tools available to embedded control applications. Essentially, it can be viewed as a Pattern Recognition problem: detecting a pattern of uttered speech to be a close match to prespecified reference patterns, or "templates," of corresponding utterances. In the context of embedded systems, the result can often be the detection of specific commands spoken by the end-user of the system. The number of such commands is dictated by the needs of the application as well as the capabilities and memory size of the processing hardware platform being used to implement the system. The simplified block diagram below demonstrates a typical example of a speech recognition system operating within an embedded control system framework. In this example, a dsPIC30F DSC device and the corresponding Speech Recognition Library is being used to control a generic device (Fig. 10.1).

Broadly, the problem of Speech Recognition can be classified into four broad categories, listed below. Conceivably there is also a need for recognition of continuous

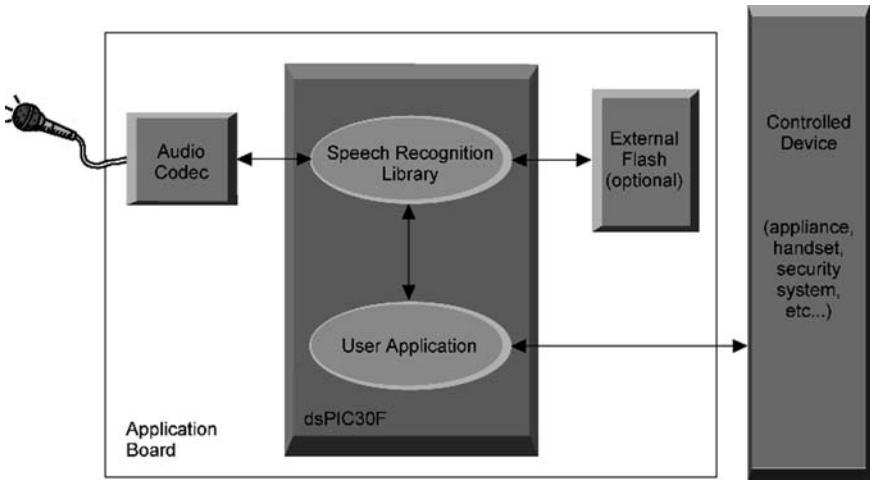

Audio Codec
Speech Recognition Library
External Flash (optional)
Controlled Device
(appliance, handset, security system, etc...)
User Application
dsPIC30F
Application Board

Fig. 10.1 Example block diagram of an embedded speech processing system [3]

speech rather than individual words or commands, but this is usually less common in embedded control applications and more computationally demanding.

- Speaker-Independent Speech Recognition: These are applications in which any user can potentially use the product of interest without any security implications, and the system should be able to interpret the user's uttered commands accurately. Some examples are the following:
  - A voice-controlled baking oven or microwave oven – The manufacturers of the appliance do not know in advance who will use the end-product, and it is very likely that several people would need to cook food items using the same unit. Typically, command words such as "cook," "set," "bake," "broil," "off," and "on" need to be recognized and acted upon, besides interpreting the 10 digits (e.g., to specify the cooking time or to set the clock).
  - An Audio Player or Video Player – With the help of speech recognition, the user can invoke commands such as "play," "stop," "skip," "shuffle," "rewind," "bass," etc. without having to get up and walk to the unit or even have a confusing set of remote controls for multiple gadgets. Again, in many cases several family members might operate the same device, thereby requiring some sort of speaker-independent recognition algorithm.
  - Aids for Hearing-Impaired Persons – Devices could be designed that "listen" to spoken words and translates them into a visual text string, for the benefit of hearing-impaired persons.
  - Industrial Control Systems – Many spots and machines in industrial sites are difficult for operators to physically reach, often making it near-impossible to adjust, say, the temperature of a chemical reactor or the speed of a pump.

  Incorporating speaker-independent speech recognition algorithms into the control implementation of such systems can make day-to-day operation much easier.
  - Smart Toys – Many toys, especially those that involve some kind of motion or action, can utilize voice-based control, where a small child using it can simply tell the toy to operate in certain ways without having to press any switches or turn any knobs. Another significant application is in educative toys, especially those that are intended to help children develop spoken language learning skills.
  - Intercom Systems – Many industrial and corporate locations might contain intercom systems for employees to communicate with each other and exchange information, and also for accessing help during security and medical emergencies. To save time and improve accessibility, it may be useful to incorporate voice-controlled dialing features to such phone systems, thereby requiring speaker-independent recognition of the 10 digits as well as commands such as "hang-up," "dial," and some prespecified calling destinations like "security."

- Speaker-Dependent Speech Recognition: This refers to applications where the end-product has either been designed or customized for a specific person, or it includes a self-training mechanism or tool whereby the end-user can generate templates specific to his/her own voice for the keywords and commands of interest. As we will see, this is generally a less intricate problem compared with its speaker-independent variant, and has some different speech preprocessing requirements. Some examples of such applications are given below:
  - Voice dialing in a Mobile Hands-Free Kit – The need to manually press keys to dial numbers on a mobile phone sometimes defeats the advantages of using a hands-free kit, for example, one used in a car or truck cabin. For the system to be truly "hands-free," the dialing (as well as related actions such as answering an incoming call or hanging up) should be processed using word recognition. Now, since a mobile phone is typically used by one person (or at least a very small set of persons), this can utilize a speaker-dependent algorithm and possibly simplify the software and computational needs as a result.
  - Transcription machines – Often used by doctors and journalists, such devices can be used to take notes and directly convert them to text files for storage and patient records. For example, a doctor might dictate notes while performing a surgery or performing some physical examinations, so there is obviously no way for them to manually enter or type in the required information, and hence speech recognition is necessary. Moreover, speech recognition keywords can also be used to access and enter data (verbally) into different types of "forms" corresponding to different medical procedures and ailments. The same applies to journalists, for example, someone reporting from a scene of battle where the terrain and conditions may not be suitable for typing, and directly recognizing spoken words and communicating them to the news desk may be more useful. Since these are both personal devices, they can be speaker-dependent.

  - Voice Control in Cars – It may be beneficial for automotive manufacturers to incorporate voice-based control of some noncritical car functions. For example, adjusting the car audio system, positioning of the side mirrors, turning on wipers and air-conditioner control can be quite distracting for a driver from a safety standpoint; having such functions be voice-controllable frees up the driver's hands to concentrate on core driving tasks. In fact, one can take it a step further by enabling voice controls of key tasks such as changing gears and enabling/disabling cruise control. Again, since one or two people typically use each car, it may be desirable to make these controls speaker-dependent for ease of software implementation.
  - Robots – With rapid advances in the field of multifunction robotic technology, and the popularity of building robots as a hobby of choice among budding electronics enthusiasts, it is only a matter of time before robots become ubiquitous in consumer households as well, performing a number of simple household tasks such as fetching a magazine, cleaning a floor, or simply providing entertainment to the family. Voice-based controls, specifically trained for members of the same household or other such small sets of users, would be one of the most significant enabling factors for wider acceptance and development of robots for consumer use.

- Speaker Verification: This refers to a class of applications which utilize certain prespecified or randomly-generated keywords uttered by a person to verify his/her identity. Essentially, this is a kind of biometric pattern recognition: using the characteristics of a person's voice (the way the person speaks certain words) to either grant or deny access to a system resource or device. Unlike the previous two categories of speech processing tasks, here the focus is on security rather than control. Here are a couple of examples:

  - Security Access in Buildings – Security panels for gaining access to corporate and military facilities can utilize speaker verification algorithms to grant entry or other access to individuals.
  - Keyless Vehicle Entry – Speaker Verification can be used by the driver of a car to gain entry into the car by uttering a prespecified password.
  - Personal Electronic Devices – Personal electronic products such as laptop PCs and mobile phones can employ speaker verification to log on or turn on the devices. In this case, the person's voice can be used as a password.

- Speaker Identification: This is a task that is closely related to Speaker Verification, except that here the speaker's utterance needs to be compared with those of several different speakers and the speaker identified out of a pool of possible users.

  - At a simplistic level, the objective may be to identify the user from multiple legitimate users and possibly customize the operation of the product by loading a set of user parameters, or "user profile." A user accessing a videogame machine may be one such example, wherein the machine might store the settings and results from your previous usage and bring them up the next time you use the system.

– A more complex application may be in the area of security/surveillance or forensics, where some of the possible speakers may not be legitimate users, and indeed the objective may be to identify certain known malicious users who may be trying to gain access to certain resources. Banks and safety vaults may belong to this class.
– An even more complex scenario is one where the users of the system may not even be aware that their uttered words are being recorded or analyzed, for example, in surveillance systems at public places or airports. The objective here is typically to detect the presence of known miscreants. The challenge for such applications is that isolated words may be difficult to achieve, and the words that will be uttered cannot be predicted. Hence, this scenario may not be suitable for embedded systems but more suitable for offline processing of the speech in a PC-based system.

Speech Recognition Using Template Matching [1,3]

One of the early and well-understood methods for recognizing words is by breaking up the uttered speech into its constituent phonetic units or Phonemes. The phonemes, in turn, are divided into short time windows or frames, similar to what we have seen for other speech processing algorithms. In this methodology, it is assumed that when the same person utters the same word on different occasions, the speech waveform does not vary considerably, except for the whole word or parts of it being spoken faster or slower. Therefore, these algorithms utilize a set of prestored reference waveforms, or Templates, for specific words spoken by each speaker. Subsequently, during actual operation of the system, when the user speaks into the device, the sampled waveforms are decomposed in a similar manner and compared with all the reference templates. Obviously, whether all such templates would be compared or only the ones corresponding to a given word or a given speaker depends on the specifics of the application, as we have seen in the previous section.

Now, for each frame (which is typically 10 to 20 ms), instead of directly using the waveform samples, the algorithm may be represented as a set of parameters or Feature Vectors. These can be represented in a multidimensional vector space, and the vector distance between each Feature Vector of the incoming speech frame and every Feature Vector in the templates is calculated. Generally, greater weight is given to those features that represent the phonetics of the uttered speech rather than the specific subtleties of the person's speech (though there are exceptions such as Speaker Identification or Speaker Verification in which the speaker's characteristics are given more weight). The most common distance metric used in speech recognition, and indeed in most Pattern Recognition algorithms, is the Euclidean Distance. The Euclidean Distance between two N-dimensional vectors $\mathbf{x}$ and $\mathbf{y}$ is given by

$$\mathrm{ED} = \sqrt{\sum_{n=1}^{N} (x_n - y_n)^2}. \tag{10.1}$$

In embedded applications, it is more common to simply use the square of the above expression to avoid the computation-intensive square-root computation. In fact, some DSP/DSC processor architectures even provide a specialized single-cycle MAC-class instruction to efficiently compute the Euclidean Distance. For example, consider the following code snippet using the dsPIC30F/33F instruction set:

```
clr      A, [w10] + = 2,   w5
repeat   #15
edac     w5*w5, A, [w9] + = 2, [w10] + = 2, w5
```

In this example, only 3 instructions and 18 instruction-cycles are needed to calculate the Euclidean Distance between two 16-dimensional Feature Vectors. The EDAC instruction shown here subtracts two numbers fetched from memory, computes the square of the difference, and adds the result to the current contents of the accumulator. By placing this special instruction inside a hardware-managed loop instruction (after the accumulator has been cleared initially), the final contents of the accumulator would be the required Euclidean Distance metric, obtained very efficiently by judiciously utilizing DSP-oriented architectural features available on processors.

The location of these constant Feature Vector templates is also an important design decision. For low-cost systems, it may be useful to avoid the need for external memory devices by storing these templates in on-chip Flash memory or even in RAM (depending on how much RAM or Flash the rest of the application is using). If the number of such templates (which of course depends on the number of words that need to be recognized) exceeds the available on-chip memory, and also if customizability of the words is an application requirement, then external memory devices such as Serial EEPROM or even Smart Cards should be considered.

A simple method of designing Feature Vectors is to pass the speech signal (both the ones used for training the system initially and the live input speech from the user) through a bank of band-pass filters, with each band representing critical bands of the human speech production system. This would not only generate a relatively small number of vector elements (good for fast comparisons), but would also remove the effect of the fundamental pitch frequency. This is particularly useful in languages such as English where there is less relationships between the pitch at which a given phoneme is uttered and its meaning.

Often, utterances might have large variations of level for the same phonemes or from one part of a word to the next. This can result in recognition problems, and can be avoided by using the logarithm of the level or by normalizing the levels across the filters. However, this is a trade-off as some level variations (especially in fricatives and transitions) do have phonetic significance.

Another problem in detecting isolated words is determining when exactly a word ended and a pause started, that is, end-point detection. Of course, a simple technique would be to assume that when the signal level in a frame rises above a certain threshold it contains speech, that is, a word has started. However, some sounds such as /f/ might start with a low-level; similarly, stop-consonants might end with a low-level. In both of these cases, the portion with the low level does have phonetic information; hence a simplistic energy-based end-point detection may not always suffice.

When a person utters a word, the speed of the utterance may not be consistent every time; in fact, even the relative speed at which different portions of the word are uttered are not constant. This is especially true for vowels, and presents a significant challenge for reliably comparing the Feature Vectors of a live word utterance with those of the corresponding templates. Of course, one could increase or decrease the entire time-scale before comparison, but even this would not account for speed variations within the word itself, and more sophisticated methods are needed. The most popular method used, especially in Speaker-Dependent Speech Recognition or Speaker Verification/Identification, is Dynamic Time Warping.

Dynamic Time Warping (DTW) is a Dynamic Programming algorithm in which the constrained warping function is optimally chosen at each step. Between a known start and end points (frames), the distance metric (e.g., Euclidean Distance) is computed for each frame transition. This process is repeated for all possible paths between the start and end points, and the true distance metric is taken to be the lowest of these values. However, considering all possible paths is computationally excessive; therefore, the DTW algorithm picks out the minimum-cost path at each transition, as given in (10.2) for two vectors x and y with indices i and j, respectively. Here, $d(i, j)$ and $D(i, j)$ being the point and cumulative distance metrics, respectively.

$$D(i, j) = \min\left[D(i-1, j), D(i-1, j-1), D(i, j-1)\right] + d(i, j). \qquad (10.2)$$

In some cases, it may be useful to assign a timescale distortion penalty, especially to distinguish between how different speakers might pronounce a particular word and thereby identify the speaker. These penalties are simply applied as correction factors in the above equation. Also, sometimes coarticulation of consecutive words might make it difficult to distinguish between pairs of different words, because of which combinations of words might need to be considered. In some cases, language syntax rules may be utilized to optimize the number of vectors considered.

Finally, silence frames can be represented by Silence Templates, which are designed to be at a level slightly higher than that of the background noise. Other extraneous sounds emitted by humans (such as coughs and clearing of the throat) can be represented as Wildcard Templates.

In general, fixed speech template matching, even with the usage of Dynamic Time Warping, does not always yield the most reliable Speech Recognition performance in the face of variability in the time-scale and other factors of speech utterance. This has resulted in development of sophisticated statistical techniques for comparison with templates, based on the probabilities of transitioning from one state to another across speech frames. These techniques are especially popular in Speaker-Independent Speech Recognition algorithms, and are outlined in the following section.

Speech Recognition Using Hidden Markov Models [1,3]

Instead of a fixed template match that provides the minimum distance metric along an optimal path from the first frame of a word to the last, an alternative methodology would be to develop a statistical representation, or Model, that would produce sets of Feature Vectors such that the statistical properties over all these sets would be similar to that of the uttered speech. The ultimate objective in this case is to determine which word model would have been most likely to have produced the speech feature vectors that have been observed; this is a case of minimizing an a posteriori probability (since the speech frame has already been sampled).

In this methodology, the successive template frames of each word is thought of as a sequence of states; in fact, a single such state can contain multiple frames. For simplicity, this entire sequence of states culminating in the final output (last frame) is considered as a Stochastic Process, that is, a nondeterministic process in which each possible state transition is defined by statistical probabilities. Given that the model is currently at a particular state, say S_n, there is a probability that the next state transition would be to a particular state. This is known as a Transition Probability. Note that the transition does not have to be to a different state: it could even be to the same state where it currently is. Another key assumption of this technique is that the next state only depends on the current state and its Transition Probabilities: it does not on any of the previous states of the model (i.e., the model effectively "hides" the influence of its past states). For this reason, this underlying model behavior is commonly referred to as a Hidden Markov Model (HMM) and the sequence of states is called a First-Order Hidden Markov Chain.

Similarly, each state in the Hidden Markov Chain can produce any of several Feature Vectors. The probability of generating any one such Feature Vector from this set is called Emission Probability, and is expressed as a Probability Density Function (PDF).

Without going deep into the mathematical derivations of the model probabilities, let me state here that the probability of emitting the complete set of Feature Vectors that have actually been observed (based on the sampled frames associated with the word) is given by (10.3), where I = Initial State, F = Final State, $S_1 - S_{T-1}$ are the intermediate states, and a and b are Transition Probabilities and Emission Probabilities, respectively (with their index indicating the corresponding emitted Feature Vector and State Transition):

$$P(y_1,\ldots,y_T) = \sum_{\text{all-possible-sequences}} a_{IS_1}\left(\prod_{t=1}^{T-1} b_{S_t}(y_t)\, a_{S_t S_{t+1}}\right) b_{S_T}(y_t)\, a_{S_T F}\,. \tag{10.3}$$

While the above equation seems rather daunting, in practice there are recursive methods of computing the final state from the knowledge of the Transition and Emission Probabilities, as given in (10.4)–(10.6) below. Let $\alpha_j(t)$ be the probability that the model could have generated the first t feature vectors that have been observed and being in state j during the $t-$th frame. Then, we have the following recursive steps for finding the final state.

$$\alpha_j(1) = a_{Ij} b_j(y_1), \tag{10.4}$$

$$\alpha_j(t) = \left(\sum_{i=1}^{N} \alpha_i(t-1) a_{ij} \right) b_j(y_t). \tag{10.5}$$

Thus, the probability of the model producing the final set of observed features is given by:

$$P(y_1, \ldots, y_T) = \alpha_F(T) = \sum_{i=1}^{N} \alpha_i(T) a_{iF}. \tag{10.6}$$

Along with the above probability analyses, it is also useful to include the probability of utterance of the word itself. This simply results in a scaling of the above computations based on Bayes' Rule.

The generation of HMMs is often done off-line (especially in Speaker-Independent Speech Recognition algorithms) during a complex training process utilizing techniques such as the Baum-Welch parameter reestimation. The reader is pointed to the references for more detailed descriptions of the training process. When training is done off-line, the resulting HMM templates are stored as constant data in nonvolatile memory.

Viterbi Algorithm [1,3]

The final probability values (computed as explained in the previous section) for many HMM state sequence are observed to be very small compared with some of the state sequences, so much so that they can effectively be neglected without any degradation in the system performance. To take it a step further, one can consider only the most probable state sequence while discarding all others, providing significant computational savings. This results in a modified version of (10.3) that only considers the most probably sequence over T frames, as shown in (10.7).

$$\hat{P}(y_1, \ldots, y_T) = \max_{\text{all-possible-sequences}} P(y_1, \ldots, y_T, s_1, \ldots, s_T) \tag{10.7}$$

This probability can be computed very efficiently using a popular dynamic programming algorithm called the Viterbi Algorithm, as given in (10.8)–(10.10) below.

$$\overset{\wedge}{\alpha}_j(1) = a_{Ij} b_j(y_1), \tag{10.8}$$

$$\overset{\wedge}{\alpha}_j(t) = \max_i \left(\overset{\wedge}{\alpha}_i(t-1) a_{ij} \right) b_j(y_t), \tag{10.9}$$

$$\hat{P}(y_1, \ldots, y_T) = \overset{\wedge}{\alpha}_F(T) = \max(\alpha_i(T) a_{iF}). \tag{10.10}$$

The Viterbi algorithm can also be used to simply the training procedure relative to Baum-Welch Estimation. When on-line training is needed, for instance during the initial training process in a Speaker-Dependent Speech Recognition application, using the Viterbi Algorithm can therefore lead to substantial computational savings.

Front-End Analysis [1, 3]

So far in this chapter, we have mainly focused on the comparison of input feature vectors with prestored template feature vectors (or statistical models generating the same), in order to determine whether the sampled speech or word has been recognized or not. A key element of this equation is, of course, the choice of Feature Vectors to be used. The general steps that go toward converting raw speech samples (perhaps obtained from a microphone, though an ADC or codec) to a discrete set of Feature Vectors is collectively known as Front-End Analysis.

First, the speech signal samples need to be preemphasized. Normally the frequency spectrum of voiced speech tapers off at a rate of around 6 dB per octave at higher frequencies. This is compensated for by preemphasizing the higher frequencies so that the frequency response is approximately flat. This can be done over all speech samples without classifying each frame into voiced or unvoiced, as it has generally been observed that unvoiced speech frames are not adversely affected by this preemphasis.

As with other speech processing algorithms, the speech frames are smoothed using a window such as Hamming or Hanning. This removes discontinuities at frame boundaries and thus prevents spurious frequencies from being injected due to frame processing. Processing frames of 10 ms are fairly common in Speech Recognition algorithms.

The signal is then passed through a bank of band-pass IIR filters, which usually have overlapping triangular frequency responses. Fourth-order filters implemented as a cascade of 2 biquad sections is a fairly common topology. The center frequencies of the filters are spaced in such a way as to approximate the response of the human auditory system; this kind of an arrangement is known as the Bark scale or mel scale, and is extremely popular in Speech Recognition applications.

Instead of filter banks, and sometimes in addition to filter banks, Linear Predictive Coding (LPC) coefficients can also be obtained from the samples. A major advantage of LPC is that it models the response of the vocal tract and effectively eliminates the effect of pitch or excitation from the resultant parameters; this is very beneficial in Speaker-Independent Speech Recognition, as pitch variations between speakers could potentially cause incorrect word identification. The LPC coefficients are then subjected to a special kind of transformation called Cepstral Analysis.

Cepstrum (the opposite of the word "spectrum") is a technique that is used to separate the excitation portion of a speech spectrum from the portion representing the vocal tract characteristics. This is very important because it is the vocal tract filter properties that primarily define the phonetic characteristics of the words uttered by people. Besides Speech Recognition, Cepstral Analysis is an important element of many other speech processing algorithms.

The Cepstrum of the signal is obtained by first calculating the log magnitude spectrum of the signal and then computing an FFT on it. In practice, because the spectrum is symmetric for real signals, a DCT is computed instead of an FFT. If there are N spectral components $\mathrm{A_1}$–$\mathrm{A_N}$, the j-th Cepstral coefficient ($j = 0$ to $N - 1$) is given by (10.11).

$$c_j = \sqrt{\frac{2}{N}\sum_{i=1}^{N} A_i \cos\left(\frac{\pi j\,(i-0.5)}{N}\right)}. \tag{10.11}$$

By truncating the Cepstral response (since the excitation portion of the signal spectrum is mostly concentrated at the higher coefficients) and reconstructing the original signal, the resultant "smoothed" signal is devoid of information about the excitation, and can therefore be used as Feature Vectors.

Other Practical Considerations [1,3]

There are some other practical considerations to keep in mind specifically for Speaker-Independent Speech Recognition. Different speakers have different vocal tract sizes and shapes, resulting in differences in the formant frequencies. One way to account for such differences is through some kind of Speaker Normalization or Vocal Tract Normalization, which is a method of warping the formant frequencies so as to provide a fair basis for comparison between words spoken by speakers whose vocal tract characteristics may be different from those whose utterances had been used to train the recognition system. Another, more practical but initially more time-consuming, is to utilize the recordings of a very large number of speakers during the off-line HMM generation process, and averaging their responses. There is also substantial ongoing research in the area of Model Adaptation, which refers to techniques for adjusting the word models as speech inputs from newer speakers become available.

To improve the robustness of the algorithm to environmental and channel noise effects, several noise reduction techniques may be employed. One such technique is a special kind of spectral band-pass filtering called Relative Spectral (RASTA), which removes signal components that are either varying too fast or too slowly, thereby keeping only those portions of the speech that are most closely related to the phonetic aspects of speech.

Finally, it may be noted that many library and DSP/DSC vendors not only provide ready-to-use software libraries for various Speech Recognition algorithms, but some also provide GUI-based tools to generate HMM data for a user-specified set of words to be recognized. These tools also allow the application developer to specify the model of noise to be used by the training process that generates the HMMs, helping to tune these models appropriately to the noise environments expected in the usage of the end-product.

Performance Assessment of Speech Recognizers [2,3]

For Word Recognition systems, the primary measure of accuracy and reliability of the system is the Recognition Accuracy, which is simply the percentage of valid utterances (i.e., utterances from the bank of words that are supported by the system) that are correctly interpreted by the algorithm. Except for mission-critical applications, 90% recognition accuracy may be considered reasonably good, especially for those applications in which the resultant control action can be stopped manually (if needed).

For Speaker Verification and Speaker Identification systems, the key performance parameters are the False Acceptance Rate and the False Rejection Rate. False Acceptance Rate is the percentage of utterances for which an imposter or another user is wrongly accepted as a particular legitimate user; obviously for high-security applications, it is critical that this rate be as low as possible. False Rejection Rate is the percentage of times a legitimate user is rejected as unidentified (or identified as another user). For a personal communication device, it would be annoying if this rate is too high, whereas it may be acceptable to have a relatively high False Acceptance Rate since the device would mostly be in the possession of its owner. The key in any Speaker Verification/Identification system is to decide on a decision threshold (e.g., distance metric at which a speaker is determined to be the claimed person) suitable for the application's needs. Often, this is selected to be the Equal Error Rate, which is the threshold at which the False Acceptance and False Rejection Rates are equal.

Finally, two other popular performance metrics are the following:

- Detection Cost: A suitably weighted average of the False Acceptance Rate and False Rejection Rate.
- Receiver Operating Characteristics: These are plots of the probability of correct acceptance vs. the probability of incorrect acceptance.

Computational Resource Requirements [3]

The feature vector analysis routines (and any associated filtering or noise reduction techniques) are the most demanding portion of any Speech Recognition algorithm, not only because the underlying computations are complex but also because these tasks must be performed in real-time on every speech frame. Although the actual template matching and decision (whether using the HMM methodology or simple waveform distance measures) is very computationally complex, they are typically only executed when a word end-point or period of silence has been detected; in other words, the real-time constraints are less intense in this case.

The effective execution time of Speech Recognition algorithms depends on whether the HMM data and Vector Codebooks are stored in on-chip Program Memory (which is much faster) or in some kind of off-chip memory device or Smart Card. For example, the dsPIC30F Speech Recognition Library from Microchip Technology requires approximately 8 MIPS when the HMM and Vector Codebook data are stored entirely on-chip, but may consume substantially more time to execute if these are stored and accessed from an off-chip source. Accessing data through a peripheral that supports Direct Memory Access (DMA) transfers would alleviate some of this memory access cost.

Last but not the least, Speech Recognition algorithms tend to be memory-intensive, and having sufficient amounts of on-chip program and data memory with fast access to them in instructions is the key to having a low-cost and efficient whole product solution. Some memory consumption metrics for a typical Speaker-Independent Speech Recognition algorithm (the dsPIC30F Speech Recognition Library) are presented below:

- Algorithm Code – 3.5 KB
- General Data – 3 KB
- Vector Codebook – 8 KB
- HMM – 1.5 KB per word

Summary

In this chapter, we have briefly explored the fascinating and futuristic subject of Speech Recognition, which includes the related tasks of Speaker Verification and Identification. There is no doubt that these application areas will become increasingly important in a wide variety of embedded control applications, many of which currently do not involve any kind of speech inputs whatsoever. Developments in CPU and peripheral architectures as well as research and development in the area of computation-optimized algorithms for Speech Recognition will only widen its usage in embedded systems.

References

1. J Holmes, W Holmes Speech Synthesis and Recognition.
2. LR Rabiner, RW Schafer Digital Processing of Speech Signals.
3. Microchip Technology Inc. dsPIC30F Speech Recognition Library User's Guide.

Chapter 11
Speech Synthesis

Abstract In the previous chapter, we have seen mechanisms and algorithms by which a processor-based electronic system can receive and recognize words uttered by a human speaker. The converse of Speech Recognition, in which a processor-based electronic system can actually produce speech that can be heard and understood by a human listener, is called Speech Synthesis. Like Speech Recognition, such algorithms also have a wide range of uses in daily life, some well-established and others yet to emerge to their fullest potential. Indeed, Speech Synthesis is the most natural user interface for a user of any product for receiving usage instructions, monitor system status, or simply carrying out a true man–machine communication. Speech Synthesis is also closely related to the subjects of Linguistics and Dialog Management. Although quite a few Speech Synthesis techniques are mature and well-understood in the research community, and some of these are available as software solutions in Personal Computers, there is tremendous potential for Speech Synthesis algorithms to be optimized and refined so that they gain wide acceptability in the world of embedded control.

Benefits and Applications of Concatenative Speech Synthesis [2]

The generation of audible speech signals that can be understood by users of an electronic device or other systems is of significant benefit for several real-life applications. These applications are spread out over all major application segments, including telecom, consumer, medical, industrial, automotive, and medical. Different Speech Synthesis algorithms have differing sets of capabilities, and the choice of the suitable algorithm needs to be such that it satisfies the needs of the particular application that is being designed.

The term Speech Synthesis may mean something as simple as the device producing appropriate selections of simple words based on a prerecorded set stored in memory. In this simplistic scenario, Speech Synthesis is essentially a Speech Playback task similar to what we have seen in the context of Speech Decoding algorithms. Yet, this is all the functionality that some applications might need. For example, a temperature sensor might continuously monitor the temperature of a machine or burner and verbalize the temperature at periodic intervals, for example, every minute. Alternatively, it might simply generate an alarm message if the temperature exceeds a predefined safe limit. A Public Address System in an office building might be set up to give a simple evacuation order if it detects a fire, in which case implementing a complex Speech Synthesis for generating only a few simple isolated words or sentences would be an overkill to say the least.

There are yet other applications in which the speech messages generated by the device or system are more substantive, and can consist of numerous complete sentences or groups of sentences. However, the sentences might all contain a common manageable subset of words. In such cases, individual words or phrases may be encoded and stored in memory, and when needed these words and phrases are concatenated together in an intelligent way to generate complete sentences. Some examples of such applications are the following:

- Talking Clock and Calendar – In this application, the number of possible sentences is almost infinite, certainly as numerous as the number of possible dates and times. However, the number of digits and number of months are constant, so using concatenation of prerecorded phrases and words, any date and time can be generated. Such systems would be particularly useful for visually impaired people, who may not be able to view the time and date on a watch or clock.
- Vending Machines and Airline Check-In Kiosks – Vending machines, as well as related devices such as automated check-in kiosks at airports, can be greatly benefited by adding simple spoken sentences that indicate the next step or action for the user to perform. Although the required actions may be obvious based on the on-screen display, many users are simply more comfortable when instructed verbally. Moreover, some machines may be space-constrained (e.g., small ticket machines in buses or roadside parking spots), and therefore having a visual display is out of the question and synthetic speech instructions and messages are the only viable option for a user interface.
- Dolls and Toys – It is certainly not uncommon for dolls and other toys to incorporate spoken sentences. However, the number of such verbal messages is usually extremely limited, especially for low-cost toys. Generating a wider variety of sentences by concatenating a relatively large set of common words would add a lot of value to the user experience of the child using the toy or doll. Moreover, these messages could be constructed in a creative manner based on various ambient conditions such as position, posture, and temperature, providing an entertaining (and potentially educationally-oriented) concoction of smartly-selected messages.
- Gaming Devices – Video games as well as casino gaming devices can be greatly benefited by the addition of some simple spoken instructions and feedback provided to the user, for example, on the results of a round of the video game, or the amount of money won in a Jackpot machine. Many video games feature human-like animated characters in a realistic setting, so having some of these characters speak some combinations of simple phrases could be of immense entertainment value for the user.

- Robots – Robotics is another very interesting application area and one that is yet to fully evolve in terms of either intelligent speech production or large-scale consumer spread. Robots of the future will need to "talk" to their masters (and possibly other robots too!) by exploiting a variety of Speech Synthesis techniques and a very large database of phrases and sentences to be concatenated together. The future frontier would then be to make the speech sound more human and natural. It is more likely that robotic speech would rely more on generation of speech based on linguistic and phonetic rules rather than simply concatenating shorter speech segments.

It must be kept in mind, however, that most of the above scenarios only pertain to enunciation of simple sentences. If larger, more complex sentences or more descriptive messages are needed, then a regular Speech Playback based on decoding of preencoded speech segments (as described in previous chapters in the context of Speech Encoding/Decoding applications) would be more suitable.

Benefits and Applications of Text-to-Speech Systems [2]

The most futuristic class of Speech Synthesis applications (at least, futuristic as far as embedded control applications are concerned) are those that do not involve any prerecorded speech segments. Instead, these applications directly "read" sentences from an ASCII text data file (or any other form of data that represents a text string) and enunciate these sentences through a speaker. Such applications and systems are popularly known as Text-to-Speech (TTS) systems, and are closely related to the science of Linguistics. These text files might reside in nonvolatile memory or in an external memory or Smart Card device, or they could be dynamically communicated through some kind of communications interface such as UART, Ethernet, or ZigBee®. In fact, a text file can even be thought of as a highly compressed form of encoded speech since only a single byte of information is often sufficient to represent an entire phoneme, and sentences to be spoken out by a system might have been stored as text to save on storage space to begin with. Therefore, as TTS technology evolves and is optimized further, it is conceivable that the applications we have already discussed gradually shift their methodology to use text files. Here are some ideas about applications that can greatly benefit from incorporating TTS:

- E-Book "Readers," with a difference – We have already seen the emergence and growing popularity of devices that can be used by people to read books in an electronic format. The advantages are, of course, increased storage and portability as well as anytime accessibility of books for purchase. Any visit to a large bookstore should make it obvious that Audio Books are also quite popular among a segment of the population, for example, one can listen to an Audio Book while on a long drive in one's car, but reading a book in such a scenario is definitely not an option. In future, these two alternative technologies for absorbing the information or story from a book will gradually merge in a variety of consumer and educational products, in small portable devices that can download hundreds of books on demand, be carried in a shirt-pocket, and can read out the contents of the book to the user through a speaker or headset. In an automotive environment, it is even possible that such devices plug into the car's audio system.

- Audio Document Scanners – There could be a large variety of devices developed that would allow visually impaired people to "listen" to the contents of any documents or newspapers they need to read but are unable to. These would typically be portable camera-like devices that can be held close to the document or page of interest, and the device would then scan the document, store it internally as a text file, convert the text to speech, and read it aloud for the user through a speaker or headphone.
- GPS Systems – The voice messages that suggest directions and locations in GPS units also utilize TTS systems.

These are but a small glimpse of some types of products and systems that would be made possible by the increasing optimization and adoption of Speech Synthesis algorithms in real-life cost-conscious embedded applications in a variety of market segments. Now that we have seen the potential of such techniques, let us understand the operation of these algorithms in a little more detail.

Speech Synthesis by Concatenation of Words and Subwords [1]

As noted earlier, a simple form of generating speech messages to be played out is to concatenate individual word messages. The individual words are produced by decompressing preencoded recordings of these words; therefore, this method of Speech Synthesis is essentially a variant of Speech Decoding. The Speech Compression techniques that are commonly used for this purpose are those based on Linear Predictive Coding (LPC), as these techniques result in very low bit rate encoding of speech data. Although techniques based on concatenation of words are popular, it does have its disadvantage in terms of customizability. Even adding a single word requires that the same persons who had recorded the original set of words record the additional word as well; this is not always feasible.

A more sophisticated approach is to concatenate smaller pieces of individual words. These smaller units can be Diphones, which are units of phonetic speech that represent the transition from one phoneme to the next; or they can even be Multiphones, which are units consisting of multiple transitions along with the associated phonemes.

A common problem of concatenative methods is that of discontinuities between the spectral properties of the underlying concatenated units. Such discontinuities can be reduced by effective use of interpolation at the discontinuities as well as by using smoothing windows.

Ultimately, the quality of the generated speech in vocoder-based methods is limited by the quality of the vocoder technique; therefore, the choice of vocoder must be made very carefully based on the speech quality needs of the specific application of interest. If the speech quality produced is not deemed sufficient, alternate methods such as concatenation of waveform segments are adopted.

Speech Synthesis by Concatenating Waveform Segments [1]

In the case of vowels, concatenation of consecutive waveform segments may result in discontinuities, except when the joining is done at the approximate instant when a pitch pulse has died down and before the next pulse starts getting generated, that is, at the lowest-amplitude voiced region. Such instants are noted at the appropriate points of the waveform through Pitch Markers for purposes of designing the concatenation. This synchronized manner in which the consecutive segments are joined is known as a "pitch-synchronous" concatenation. Moreover, a smoothing window is applied to the waveform segment, causing it to taper off at the ends. Also, consecutive waveform segments are slightly overlapped with each other, resulting in an "overlap-add" method of joining these segments. This overall process is called Pitch-Synchronous Overlap-Add (PSOLA). PSOLA is a popular technique in speech synthesis by concatenating waveform segments, and besides smoothing discontinuities this method also allows pitch and timing modifications directly in the waveform domain.

The pitch frequency of the synthesized signal can be increased or decreased by simply moving successive pitch markers to make them closer together or wider apart relative to each other. The window length used for the concatenation described above should be such that moving the pitch markers does not adversely affect the representation of the vocal tract parameters. An analysis window around twice the pitch period is particularly popular in this regard.

Similarly, timing modifications, that is, making the speech playback slower or faster, can be accomplished by replicating the pitch markers or deleting alternate pitch markers. The impact on the audible speech quality must be understood before performing any timing modifications.

There are several variants of the basic Time Domain PSOLA algorithm described above. These are briefly outlined below:

- Frequency Domain PSOLA (FD-PSOLA): In this method, the short-term spectral envelope is computed, and prosodic modifications (e.g., pitch and timing modifications) as well as waveform concatenation are performed in the frequency domain itself.
- Linear Predictive PSOLA (LP-PSOLA): In this method, the TD-PSOLA techniques already described are applied to the prediction error signal rather than the original waveform. Besides the separation of excitation from vocal tract parameters and the resultant ease of prosodic modifications, this technique also enables usage of codebooks and thus reduces memory usage.
- Multiband Resynthesis PSOLA (MBR-PSOLA): In this approach, the voiced speech is represented as the sum of sinusoidal harmonics. Subsequently, the pitch

is made constant, thereby eliminating the need pitch markers and avoiding any pitch mismatch between the concatenated waveform segments.

- Multiband Resynthesis Overlap-Add (MBROLA): This is a technique in which a differential PCM technique is applied to samples in adjacent pitch periods, resulting in the large memory savings characteristic of Differential PCM schemes.

Speech Synthesis by Conversion from Text (TTS) [1]

TTS systems attempt to mimic the process by which a human reader reads and interprets text and produces speech. The speech thus generated should be of reasonable quality and naturalness so that listeners accept and are able to understand it. The conversion from text to speech is essentially a two-stage procedure. First, the input text must be analyzed to extract the underlying phonemes used as well as the points of emphasis and the syntax of the sentences. Second, the above linguistic description is used as a basis to generate the required prosodic information and finally the actual speech waveform.

Some of the key tasks and steps involved in the first phase, that is, the Text Analysis, are listed in the subsections below.

Preprocessing

The input text is typically in the form of ASCII character strings and is stored digitally in memory. The overall character string needs to be parsed and split into sentences, and sentences into words. As part of this process, extraneous elements such as punctuation marks are removed once they have served their purpose in the parsing flow. At this stage, the TTS system also records specific subtleties of how various words need to be pronounced, based on a large number of phonetic rules that are known in advance as well as the specific context of each word.

Morphological Analysis

This step involves parsing the words into their smallest meaningful pieces (also known as Morphs), and categorized into Roots (the core of a composite word) and Affixes (which includes prefixes and suffixes in various forms). Again, predefined sets of rules can be utilized to perform this additional level of parsing. Once this is done, it greatly simplifies the process of determining the pronunciation of words, as only Root Morphs need to be stored in the pronunciation dictionary (since Affixes do not typically have any variability in their pronunciations). Also, this implies that compound words do not have to be explicitly stored in pronunciation dictionaries. Analysis of Morphs also enables extraction of syntactic information such as gender, case, and number.

Phonetic Transcription

Each word (or its underlying Morphs) is then searched in the pronunciation dictionaries to determine the appropriate pronunciation of the word. Basically, the word is now represented in terms of its Phonemes. If the word's pronunciation could not be determined using the dictionary, the individual letters are analyzed to generate a close estimate of how the word should be uttered. Another important part of this phase is to apply postlexical rules, which broadly refers to any changes in phonetic representation that may be needed based on a knowledge of which words precede and succeed the word being analyzed.

Syntactic Analysis and Prosodic Phrasing

The knowledge of syntax that may have been gained in the Morphological Analysis phase is utilized to further resolve unclear pronunciation decisions, based on the grammatical structure of the sentence. For example, how the sentence is divided into phrases is dependent on the case of the sentence and where the nouns and verbs lie. This results in more refined enunciations of the sentence. Statistical analyses are extensively used in this stage, especially to determine the phrase structure of the sentence.

Assignment of Stresses

The sentence is then analyzed more finely to determine on which words, and where within each word, the emphases should lie. This is, of course, based on a large number of language rules, for example, which words carry the main content of a phrase and which words are simply function words such as adjectives and articles. By this stage, the Linguistic Analysis of the text is complete.

The two main aspects of Prosody Generation, from which the waveform generation follow naturally, are listed below.

Timing Pattern

This stage involves the determination of the speech durations for each segment (frame). This is determined by a number of heuristic rules; for example, the locations and number of occurrences of a vowel within a word defines whether it would be of long duration or not. This stage is an area of continuous improvement, especially the challenge of making these heuristic decisions in an automated fashion and imbibing natural features of human speech behavior such as pauses and deliberate speed-changes.

Fundamental Frequency

The Fundamental Frequency for each segment is usually generated as a series of filtered Phrase Control and Stress Control commands generated by the previous stages of the algorithm. It is particularly challenging to mimic the natural pitch and stress variations of human speech. Fortunately, some languages such as English are less sensitive to differences in pitch, compared to other languages that assign specific lexical meanings to pitch variations.

Computational Resource Requirements [1]

In terms of computational bandwidth requirements, Speech Synthesis algorithms are not particularly intensive. For example, a concatenative algorithm based on the basic Time Domain PSOLA methodology, or even a parametric TTS algorithm, would probably require less than 20 MIPS on many DSP/DSC architectures. Similarly, the MIPS requirements for techniques based on LPC and other speech encoding techniques are primarily dictated by the specific speech decoding algorithms being used.

However, the memory requirements for Speech Synthesis algorithms are by far the most demanding of all the major categories of speech processing tasks. For example, a pronunciation dictionary for each word may be around 30 bytes, which means even a 1,000-word dictionary would require around 30 KB. Therefore, in embedded applications a trade-off might need to be made between the memory usage (and therefore the system cost) and the number of words supported.

Summary

Text Synthesis algorithms and techniques form one of the most promising areas of Speech Processing for future research and increased adoption in embedded environments. While a consistent focus in Speech Synthesis algorithmic improvements will be how to improve the quality and naturalness of the speech, what will make it even more critical for integration into embedded systems is a continuous effort at reducing the memory usage and other computational needs of such algorithms, especially TTS algorithms. As Speech Synthesis algorithms gradually become more optimized yet increasingly sophisticated, the scope for incorporating this key functionality into consumer devices is almost endless. Moreover, Speech Synthesis algorithms will work in conjunction with Speech Recognition and Artificial Intelligence principles to provide a seamless Dialog Management system in a variety of applications.

References

1. J Holmes, W Holmes Speech Synthesis and Recognition, CRC Press, 2001.
2. LR Rabiner, RW Schafer Digital Processing of Speech Signals, Prentice Hall, 1998.

Chapter 12
Conclusion

The road ahead for the usage of speech processing techniques in embedded applications is indeed extremely promising. As product designers continually look for ways to differentiate their products from those of their competitors, enhancements to their user interface become a critical area of differentiation. Production and interpretation of human speech will be one of the most important component of the user interface of many such systems. As we have seen in various chapters of this book, in many cases the speech processing aspect may be integral to the core functionality of the product. In yet other cases, it may simply be part of add-on features to enhance the overall user experience.

It is a very likely scenario that speech-based data input and output will be as commonplace in embedded applications as, say, entering data through a keyboard or through switches and displaying data on an LCD display or bank of LEDs. Being a natural means of communication between human beings, speech-based user interfaces will provide a greater degree of comfort for human operators, thereby enabling greater acceptance of digital implementations of products and instruments that were traditionally analog or even nonelectronic. Whether in the industrial, consumer, telecom, automotive, medical, military, or other market segments, the incorporation of speech-based controls and status reporting will add tremendous value to the application's functionality and ease of use.

Of course, applications that are inherently speech based, such as telecommunication applications, will continue to benefit from a wider adoption of speech processing in embedded systems. For example, advances in speech compression techniques and the optimization of such algorithms for efficient execution on low-cost embedded processors will enable more optimal usage of communication bandwidth. This would also often reduce the cost of the product of interest, in turn increasing its adoption in the general population.

Continuing research and advancement in the various areas of speech processing will expand the scope of speech processing algorithms. Speech recognition in embedded systems will gradually no longer be limited to isolated word recognition but encompass the recognition and interpretation of continuous speech, e.g., in portable devices that can be used by healthcare professionals for automated transcription of patient records. Research efforts will also enable more effective recognition of multiple languages and dialects, as well as multiple accents of each language. This would, in turn, widen the employment of speech recognition in real-life products of daily use. Similar advances can be expected in Speech Synthesis, wherein enunciation of individual isolated words connected in a simplistic fashion would gradually be replaced by a more natural-sounding generation of complete sentences with the proper semantic emphases and expression of emotions as needed. These advances would find their way, for example, in a wider proportion of toys and dolls for children. Noise and Echo Cancellation algorithms would also develop further to produce greater Echo Return Loss Enhancement and finer control over the amount of noise reduction. Robustness of these algorithms, over a wide variety of acoustic environments and different types of noise, will also be an area of increasing research focus. Speech Compression research will continue to improve the quality of decoded speech while simultaneously reducing the data rate of encoded speech, i.e., increases in compression ratio. As speech processing becomes more and more popular in real-time embedded systems, an increasingly important research area would be in optimizing the computational requirements of all the above classes of speech processing algorithms.

Optimizing and improving the performance and robustness of algorithms is one approach to enable efficient implementation in real systems. An equally important endeavor should be to enhance the processing architecture that is used to implement speech processing algorithms and applications that utilize these algorithms. Architectural enhancements to allow higher processing speeds while minimizing the power consumed by the processing device at the same time will be crucial in efficiently implementing such applications. Increasing processor speeds typically causes increases in power consumption, hence many architectural enhancements would have to rely on more subtle means of speeding up speech processing algorithms, e.g., by more extensive and carefully designed DSP features and a wide range of DSP-oriented instructions. More efficient forms of memory accesses, both for program memory as well as constants and data arrays, will be another vital component of ongoing research in CPU architectures. Beyond a certain point, utilization of architectural concepts that utilize the inherent parallelism present in on-chip computational resources will provide the key for greater performance; superscalar and Very Long Instruction Word (VLIW) architectures are key examples of such architectural enhancements. For some algorithms and applications requiring even faster processing speeds, it may be necessary to employ Parallel Processing techniques, either using multiple processor devices or using devices that contain multiple processors within the same chip.

As we have already seen, for a whole product solution it is necessary but not sufficient to utilize an effective CPU architecture: the peripheral modules used for the application must also provide the right set of capabilities for the task at hand. Therefore, a lot of future research and development will be in the area of enhanced on-chip peripherals. Ever-increasing ADC speeds and resolutions, integration of powerful DAC modules on-chip, as well as faster and improved communication peripherals will all contribute to making speech processing applications greatly more feasible in real-time embedded applications.

In summary, Embedded Speech Processing will remain a significant area of technical advancement for many years to come, completely revolutionizing the way embedded control and other embedded applications operate and are used in a wide variety of end-applications.

References

1. Proakis JG, Manolakis DG Digital signal processing – principles, algorithms and applications, Prentice Hall, 1995.
2. Rabiner LR, Schafer RW Digital processing of speech signals, Prentice Hall, 1998.
3. Chau WC Speech coding algorithms, Wiley-Interscience, 2003.
4. Spanias AS (1994) Speech coding: a tutorial review. Proc IEEE 82(10):1541–1582
5. Hennessy JL, Patterson DA, Computer architecture – a quantitative approach, Morgan Kaufmann, 2007.
6. Holmes J, Holmes W Speech synthesis and recognition, CRC Press, 2001.
7. Sinha P (2005) DSC is an SoC innovation. Electron Eng Times, July 2005, pages 51–52.
8. Sinha P (2007) Speech compression for embedded systems. In: Embedded systems conference, Boston, October 2007

Index