1

Introduction to Real-Time Digital Signal Processing

Signals can be classified into three categories: continuous-time (analog) signals, discrete-time signals, and digital signals. The signals that we encounter daily are mostly analog signals. These signals are defined continuously in time, have infinite resolution of amplitude values, and can be processed using analog electronics containing both active and passive circuit elements. Discrete-time signals are defined only at a particular set of time instances, thus they can be represented as a sequence of numbers that have a continuous range of values. Digital signals have discrete values in both time and amplitude, thus they can be stored and processed by computers or digital hardware. In this book, we focus on the design, implementation, and applications of digital systems for processing digital signals [1–6]. However, the theoretical analysis usually uses discrete-time signals and systems for mathematical convenience. Therefore, we use the terms “discrete-time” and “digital” interchangeably.

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

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

1. *Flexibility.* Functions of a DSP system can be easily modified and upgraded with software that implements the specific operations. One can design a DSP system to perform a wide variety of tasks by executing different software modules. A digital device can be easily upgraded in the field through the on-board memory (e.g., flash memory) to meet new requirements, add new features, or enhance its performance.

Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Third Edition.
Sen M. Kuo, Bob H. Lee and Wenshun Tian.

2. *Reproducibility.* The functions of a DSP system can be repeated precisely from one unit to another. In addition, by using DSP techniques, digital signals can be stored, transferred, or reproduced many times without degrading the quality. By contrast, analog circuits will not have the same characteristics even if they are built following identical specifications, due to analog component tolerances.
3. *Reliability.* The memory and logic of DSP hardware do not deteriorate with age. Therefore, the performance of DSP systems will not drift with changing environmental conditions or aged electronic components as their analog counterparts do.
4. *Complexity.* DSP allows sophisticated applications such as speech recognition to be implemented using low-power and lightweight portable devices. Furthermore, there are some important signal processing algorithms such as image compression and recognition, data transmission and storage, and audio compression, which can only be performed using DSP systems.

With the rapid evolution in semiconductor technologies, DSP systems have lower overall cost compared to analog systems for most applications. DSP algorithms can be developed, analyzed, and simulated using high-level language software such as C and MATLAB®. The performance of the algorithms can be verified using low-cost, general-purpose computers. Therefore, DSP systems are relatively easy to design, develop, analyze, simulate, test, and maintain.

There are some limitations associated with DSP. For example, the bandwidth of a DSP system is limited by the sampling rate. Also, most of the DSP algorithms are implemented using a fixed number of bits with limited precision and dynamic range, resulting in undesired quantization and arithmetic errors.

1.1 Basic Elements of Real-Time DSP Systems

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

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

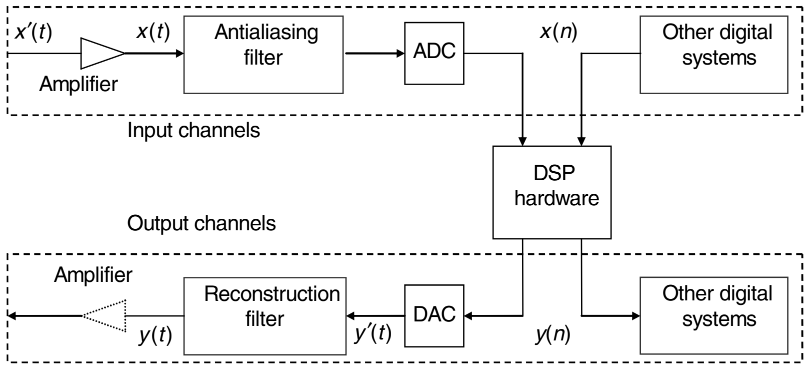


Figure 1.1 Basic functional block diagram of a real-time DSP system

to an analog signal. For some applications, the input signal may be already in digital form and/or the output data may not need to be converted to an analog signal, for example, the processed digital information may be stored in memory for later use. In other applications, DSP systems may be required to generate signals digitally, such as speech synthesis and signal generators.

1.2 Analog Interface

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

The purpose of A/D conversion is to convert the analog signal to digital form for processing by digital hardware. As shown in Figure 1.1, the analog signal $x'(t)$ is picked up by an appropriate electronic sensor that converts pressure, temperature, or sound into electrical signals. For example, a microphone can be used to pick up speech signals. The sensor output signal $x'(t)$ is amplified by an amplifier with a gain of value g to produce the amplified signal

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

The gain value g is determined such that $x(t)$ has a dynamic range that matches the ADC used by the system. If the peak-to-peak voltage range of the ADC is ± 2 volts (V), then g may be set so that the amplitude of signal $x(t)$ to the ADC is within ± 2 V. In practice, it is very difficult to set an appropriate fixed gain because the level of $x'(t)$ may be unknown and changing with time, especially for signals with larger dynamic ranges such as human speech. Therefore, many practical systems use digital automatic gain control algorithms to determine and update the gain value g based on the statistics of the input signal $x'(t)$.

Once the digital signal has been processed by the DSP hardware, the result $y(n)$ is still in digital form. In many DSP applications, we have to convert the digital signal $y(n)$ back to the analog signal $y(t)$ before it can be applied to appropriate analog devices. This process is called the digital-to-analog (D/A) conversion, typically performed by a D/A converter (DAC). One example is a digital audio player, in which the audio music signals are stored in digital format. An audio player reads the encoded digital audio data from the memory and reconstructs the corresponding analog waveform for playback.

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

1.2.1 Sampling

As shown in Figure 1.1, the ADC converts the analog signal $x(t)$ into the digital signal $x(n)$. The A/D conversion, commonly referred to as digitization, consists of the sampling (digitization in time) and quantization (digitization in amplitude) processes as illustrated in Figure 1.2. The basic sampling function can be carried out with an ideal "sample-and-hold"

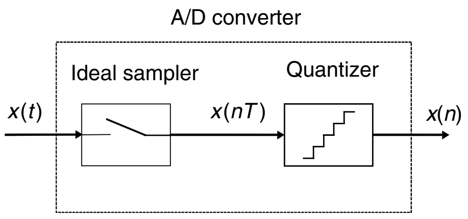


Figure 1.2 Block diagram of an ADC

circuit, which maintains the sampled signal level until the next sample is taken. The quantization process approximates the waveform by assigning a number to represent its value for each sample. Therefore, the A/D conversion performs the following steps:

1. The signal $x(t)$ is sampled at uniformly spaced time instants nT, where n is a positive integer and T is the sampling period in seconds. This sampling process converts an analog signal into a discrete-time signal $x(nT)$ with continuous amplitude value.
2. The amplitude of each discrete-time sample $x(nT)$ is quantized into one of 2^B levels, where B is the number of bits used to represent each sample. The discrete amplitude levels are represented (or encoded) into binary words $x(n)$ with the fixed wordlength B.

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

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

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

where f_s is the sampling frequency in hertz (Hz) or sampling rate in samples per second. The intermediate signal $x(nT)$ is a discrete-time signal with continuous value (a number with infinite precision) at discrete time nT, $n = 0, 1, \ldots, \infty$, as illustrated in Figure 1.3. The

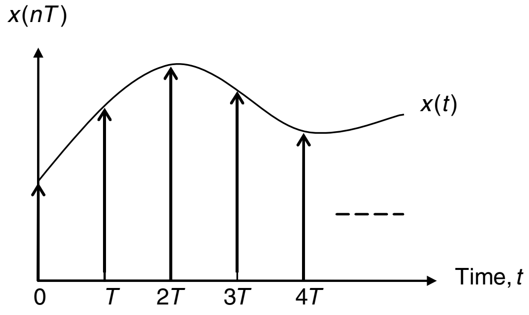


Figure 1.3 Sampling of analog signal $x(t)$ and the corresponding discrete-time signal $x(nT)$

analog signal $x(t)$ is continuous in both time and amplitude. The sampled discrete-time signal $x(nT)$ is continuous in amplitude, but defined only at discrete sampling instants $t = nT$.

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

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

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

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

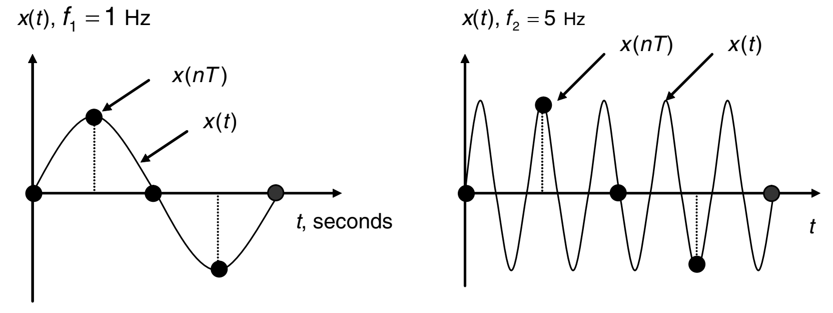
Example 1.1

Consider two sine waves of frequencies $f_1 = 1$ Hz and $f_2 = 5$ Hz that are sampled at $f_s = 4$ Hz, rather than at least 10 Hz according to the sampling theorem. The analog waveforms and the digital samples are illustrated in Figure 1.4(a), while their digital samples and reconstructed waveforms are illustrated in Figure 1.4(b). As shown in the figures, we can reconstruct the original waveform from the digital samples for the sine wave of frequency $f_1 = 1$ Hz. However, for the original sine wave of frequency $f_2 = 5$ Hz, the resulting digital samples are the same as $f_1 = 1$ Hz, thus the reconstructed signal is identical to the sine wave of frequency 1 Hz. Therefore, f_1 and f_2 are said to be aliased to one another, that is, they cannot be distinguished by their discrete-time samples.

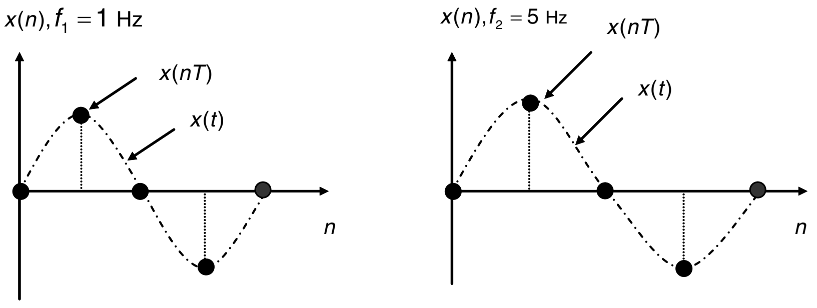
Note that the sampling theorem assumes the signal is bandlimited by f_M. For many practical applications, the analog signal $x(t)$ may have significant frequency components outside the highest frequency of interest, or may contain noise with a wider bandwidth. In some applications, the sampling rate is predetermined by given specifications. For example, most voice communication systems define the sampling rate of 8 kHz (kilohertz). Unfortunately, the frequency components in typical speech can be much higher than 4 kHz. To guarantee that the sampling theorem is satisfied, we must eliminate the frequency components above the Nyquist frequency. This can be done by using an antialiasing filter which is an analog lowpass filter with the cutoff frequency bounded by

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

Ideally, an antialiasing filter should remove all frequency components above the Nyquist frequency. In many practical systems, a bandpass filter is preferred to remove frequency



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



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

Figure 1.4 Example of the aliasing phenomenon

components above the Nyquist frequency, as well as to eliminate undesired DC offset, 60 Hz hum, or other low-frequency noises. For example, a bandpass filter with a passband from 300 to 3400 Hz is widely used in telecommunication systems to attenuate the signals whose frequencies lie outside this passband.

Example 1.2

The frequency range of signals is large, from approximately gigahertz (GHz) in radar down to fractions of hertz in instrumentation. For a specific application with given sampling rate, the sampling period can be determined by (1.2). For example, some real-world applications use the following sampling frequencies and periods:

1. In International Telecommunication Union (ITU) speech coding/decoding standards ITU-T G.729 [7] and G.723.1 [8], the sampling rate is $f_s = 8$ kHz, thus the sampling period $T = 1/8000$seconds $= 125$ μs (microseconds). Note that 1 μs $= 10^{-6}$ seconds.
2. Wideband telecommunication speech coding standards, such as ITU-T G.722 [9] and G.722.2 [10], use the sampling rate of $f_s = 16$ kHz, thus $T = 1/16\,000$ seconds $= 62.5$ μs.
3. High-fidelity audio compression standards, such as MPEG-2 (Moving Picture Experts Group) [11], AAC (Advanced Audio Coding), MP3 (MPEG-1 layer 3) [12] audio, and

Dolby AC-3, support the sampling rate of $f_s = 48$ kHz, thus $T = 1/48\,000$ seconds $=$ 20.833 μs. The sampling rate for MPEG-2 AAC can be as high as 96 kHz.

The speech coding algorithms will be discussed in Chapter 9 and the audio coding techniques will be introduced in Chapter 10.

1.2.2 Quantization and Encoding

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

For example, 2 bits define four equally spaced levels (00, 01, 10, and 11), which can be used to classify the signal into the four subranges illustrated in Figure 1.5. In this figure, the open circles represent the discrete-time signal $x(nT)$, and the solid circles the digital signal $x(n)$. The spacing between two consecutive quantization levels is called the quantization width, step, or resolution. A uniform quantizer has the same spacing between these levels. For uniform quantization, the resolution is determined by dividing the full-scale range by the total number of quantization levels, 2^B.

In Figure 1.5, the difference between the quantized number and the original value is defined as the quantization error, which appears as noise in the output of the converter. Thus, the quantization error is also called the quantization noise, which is assumed to be a random noise. If a B-bit quantizer is used, the signal-to-quantization-noise ratio (SQNR) is approximated by the following equation (to be derived in Chapter 2):

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

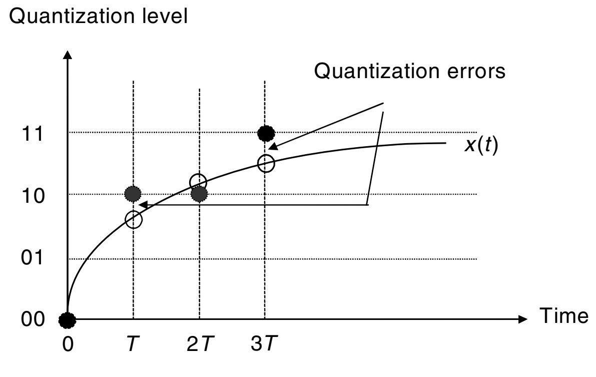


Figure 1.5 Digital samples using 2-bit quantizer

In practice, the achievable SQNR will be less than this theoretical value due to imperfections in the fabrication of converters. Nevertheless, Equation (1.5) provides a simple guideline to determine the required bits for a given application. For each additional bit, a digital signal will have about 6 dB gain in SQNR. The problems of quantization noise and their solutions will be further discussed in Chapter 2.

Example 1.3

If the analog signal varies between 0 and 5 V, we have the resolutions and SQNRs for the following commonly used ADCs:

1. An 8-bit ADC with 256 (2^8) levels can only provide 19.5 mV resolution and 48 dB SQNR.
2. A 12-bit ADC has 4096 (2^{12}) levels of 1.22 mV resolution, and provides 72 dB SQNR.
3. A 16-bit ADC has 65536 (2^{16}) levels, and thus provides 76.294 μV resolution with 96 dB SQNR.

Obviously, using more bits results in more quantization levels (or finer resolution) and higher SQNR.

The dynamic range of speech signals is usually very large. If the uniform quantization scheme is adjusted for loud sounds, most of the softer sounds may be pressed into the same small values. This means that soft sounds may not be distinguishable. To solve this problem, we can use a quantizer with quantization level varying according to the signal amplitude. For example, if the signal has been compressed by a logarithm function, we can use a uniform level quantizer to perform non-uniform quantization by quantizing the logarithm-scaled signal. The compressed signal can be reconstructed by expanding it. The process of compression and expansion is called companding (compressing and expanding). The ITU-T G.711 μ-law (used in North America and parts of Northeast Asia) and A-law (used in Europe and most of the rest of the world) schemes [13] are examples of using companding technology, which will be further discussed in Chapter 9.

As shown in Figure 1.1, the input signal to DSP hardware may be digital signals from other digital systems that use different sampling rates. The signal processing techniques called interpolation or decimation can be used to increase or decrease the sampling rates of the existing digital signals. Sampling rate changes may be required in many multi-rate DSP systems, for example, between the narrowband voice sampled at 8 kHz and wideband voice sampled at 16 kHz. The interpolation and decimation processes will be introduced in Chapter 3.

1.2.3 Smoothing Filters

Most commercial DACs are zero-order-hold devices, meaning they convert the input binary number to the corresponding voltage level and then hold that level for T seconds. Therefore, the DAC produces the staircase-shaped analog waveform $y'(t)$ as shown by the solid line in Figure 1.6, which is a rectangular waveform with amplitude corresponding to the signal value with a duration of T seconds. Obviously, this staircase output signal contains many high-frequency components due to an abrupt change in signal levels. The reconstruction or

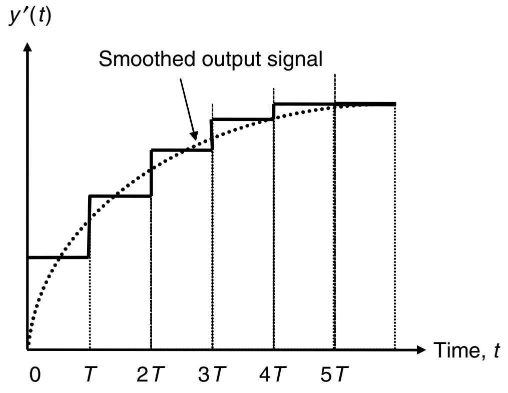


Figure 1.6 Staircase waveform generated by DAC and the smoothed signal

smoothing filter shown in Figure 1.1 smoothes the staircase-like analog signal generated by the DAC. This lowpass filtering has the effect of rounding off the corners (high-frequency components) of the staircase signal and making it smoother, which is shown as the dotted line in Figure 1.6. This analog lowpass filter may have the same specifications as the antialiasing filter with cutoff frequency $f_c \leq f_s/2$. Some high-quality DSP applications, such as professional digital audio, require the use of reconstruction filters with very stringent specifications. To reduce the cost of using high-quality analog filters, the oversampling technique can be adopted to allow the use of low-cost filters with slower roll-off.

1.2.4 Data Converters

There are two methods of connecting an ADC and DAC to a digital signal processor: serial and parallel. A parallel converter receives or transmits all B bits in one pass, while a serial converter receives or transmits B bits in a serial of bit stream, 1 bit at a time. Parallel converters are attached to the digital signal processor's external address and data buses, which are also attached to many different types of devices. Serial converters can be connected directly to the built-in serial ports of digital signal processors. Since serial converters require a few signals (pins) to connect with digital signal processors, many practical DSP systems use serial ADCs and DACs.

Many applications use a single-chip device called an analog interface chip (AIC) or coder/decoder (CODEC or codec), which integrates an antialiasing filter, ADC, DAC, and reconstruction filter on a single chip. In this book, we will use Texas Instruments AIC3204 on the TMS320C5505 eZdsp USB (universal serial bus) stick for real-time experiments. Typical applications using a CODEC include speech systems, audio systems, and industrial controllers. Many standards that specify the nature of the CODEC have evolved for the purposes of switching and transmission. Some CODECs use a logarithmic quantizer, that is, A-law or μ-law, which must be converted into linear format for processing. Digital signal processors implement the required format conversion (compression or expansion) either by hardware or software.

The most popular commercially available ADCs are successive approximation, dual-slope, flash, and sigma–delta. The successive-approximation type of ADC is generally accurate and

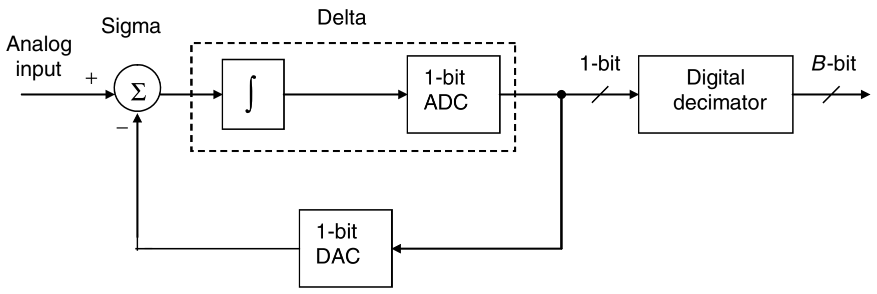


Figure 1.7 A conceptual sigma–delta ADC block diagram

fast at a relatively low cost. However, its ability to follow changes in the input signal is limited by its internal clock rate, so it may be slow to respond to sudden changes in the input signal. The dual-slope ADC is very precise and can produce ADCs with high resolution. However, they are very slow and generally cost more than successive-approximation ADCs. The major advantage of a flash ADC is its speed of conversion; unfortunately, a B-bit ADC requires $(2^B - 1)$ expensive comparators and laser-trimmed resistors. Therefore, commercially available flash ADCs usually have lower bits.

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

In this book, as mentioned above, we use the AIC3204 stereo CODEC on the TMS320C5505 eZdsp for real-time experiments. The ADCs and DACs within the AIC3204 use a sigma–delta technology with integrated digital lowpass filters. It supports a data wordlength of 16, 20, 24, and 32 bits, with sampling rates from 8 to 192 kHz. Integrated analog features consist of stereo-line input amplifiers with programmable analog gains and stereo headphone amplifiers with programmable analog volume control.

1.3 DSP Hardware

Most DSP systems are required to perform intensive arithmetic operations such as repeated multiplications and additions. These operations may be implemented on digital hardware such as microprocessors, microcontrollers, digital signal processors, or custom integrated circuits. The selection of appropriate hardware can be determined by the given application based on the performance, cost, and/or power consumption. In this section, we will introduce several different digital hardware options for DSP applications.

1.3.1 DSP Hardware Options

As shown in Figure 1.1, the processing of digital signal $x(n)$ is performed using the DSP hardware. Although it is possible to implement DSP algorithms on different digital hardware, the given application determines the optimum hardware platform. The following hardware options are widely used for DSP systems:

1. Special-purpose (custom) chips such as application-specific integrated circuit (ASICs).
2. Field-programmable gate arrays (FPGAs).
3. General-purpose microprocessors or microcontrollers (μP/μC).
4. General-purpose digital signal processors.
5. Digital signal processors with application-specific hardware (HW) accelerators.

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

ASIC devices are usually designed for specific tasks that require intensive computation such as digital subscriber loop (DSL) modems, or high-volume products that use mature algorithms such as fast Fourier transforms. These devices perform the required functions much faster because their dedicated architecture is optimized for the required operations, but they lack flexibility to modify the specific algorithms and functions for new applications. They are suitable for implementing well-defined and popular DSP algorithms for high-volume products, or applications demanding extremely high speeds that can only be achieved by ASICs. Recently, the availability of core modules for some common DSP functions can simplify ASIC design tasks, but the cost of prototyping ASIC devices, the longer design cycle, and the lack of standard development tool support and reprogramming flexibility sometimes outweigh their benefits.

FPGAs have been used in DSP systems for years as glue logics, bus bridges, and peripherals for reducing system costs and affording higher levels of system integration. Recently, FPGAs have been gaining considerable attention in high-performance DSP applications, and are emerging as coprocessors [14] for standard digital signal processors that need specific accelerators. In these cases, FPGAs work in conjunction with digital signal processors for integrating pre- and post-processing functions. These devices are hardware reconfigurable, and thus allow system designers to optimize the hardware architecture for implementing algorithms that require higher performance and lower production cost. In addition, designers can implement high-performance complex DSP functions using a fraction of the device, and use the rest of the device to implement system logic or interface functions, resulting in both lower costs and higher system integration.

Table 1.1 Summary of DSP hardware implementations

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

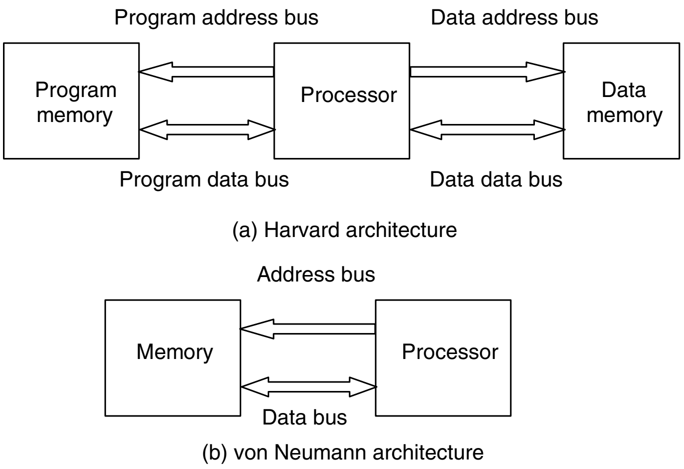


Figure 1.8 Different memory architectures

General-purpose μP/μC become faster and increasingly capable of handling some DSP applications. Many electronics products such as automotive controllers use microcontrollers for engine, brake, and suspension control and are often designed using these processors. If new DSP functions are needed for an existing product based on μP/μC, it is preferable to implement these functions in software than modify existing hardware.

General μP/μC architectures fall into two categories: Harvard architecture and von Neumann architecture. As illustrated in Figure 1.8(a), Harvard architecture has separate memory spaces for the program and the data, thus both memories can be accessed simultaneously. The von Neumann architecture assumes that the program and data are stored in the same memory as illustrated in Figure 1.8(b). Operations such as add, move, and subtract are easy to perform on μP/μC. However, complex instructions such as multiplication and division are slow since they need a series of conditional shift, addition, or subtraction operations. These devices do not have the architecture or on-chip facilities required for efficient DSP operations, and they are not cost effective or power efficient for many DSP applications. It is important to note that some modern microprocessors, specifically for mobile and portable devices, can run at high speed, consume low power, provide single-cycle multiplication and arithmetic operations, have good memory bandwidth, and have many supporting tools and software available for ease of development.

A digital signal processor is basically a microprocessor with architecture and instruction set designed specifically for DSP applications [15–17]. The rapid growth and exploitation of digital signal processor technology is not a surprise, considering the commercial advantages in terms of the fast, flexible, low-power consumption, and potentially low-cost design capabilities offered by these devices. In comparison to ASIC and FPGA solutions, digital signal processors have advantages in ease of development and being reprogrammable in the field to upgrade product features or fix bugs. They are often more cost effective than custom hardware such as ASIC and FPGA, especially for low-volume applications. In comparison to general-purpose μP/μC, digital signal processors have better speed, better energy efficiency or power consumption, and lower cost for many DSP applications.

Today, digital signal processors have become the foundation of many new markets beyond the traditional signal processing areas for technologies and innovations in motor and motion control, automotive systems, home appliances, consumer electronics, medical and healthcare devices, and a vast range of communication and broadcasting equipment and systems. These general-purpose programmable digital signal processors are supported by integrated software development tools including C compilers, assemblers, optimizers, linkers, debuggers, simulators, and emulators. In this book, we use the TMS320C55xx for hands-on experiments.

1.3.2 Digital Signal Processors

In 1979, Intel introduced the 2920, a 25-bit integer processor with a 400 ns instruction cycle and a 25-bit arithmetic logic unit (ALU) for DSP applications. In 1982, Texas Instruments introduced the TMS32010, a 16-bit fixed-point processor with a 16×16 hardware multiplier and a 32-bit ALU and accumulator. This first commercially successful digital signal processor was followed by the development of faster products and floating-point processors. Their performance and price range vary widely.

Conventional digital signal processors include hardware multipliers and shifters, execute one instruction per clock cycle, and use the complex instructions that perform multiple operations such as multiply, accumulate, and update address pointers. They provide good performance with modest power consumption and memory usage, and thus are widely used in automotive applications, appliances, hard disk drives, and consumer electronics. For example, the TMS320C2000 family is optimized for control applications, such as motor and automobile control, by integrating many microcontroller features and peripherals on the chip.

The midrange processors achieve higher performance through the combination of increased clock rates and more advanced architectures. These processors often include deeper pipelines, instruction caches, complex instructions, multiple data buses (to access several data words per clock cycle), additional hardware accelerators, and parallel execution units to allow more operations to be executed in parallel. For example, the TMS320C55xx has two multiply and accumulate (MAC) units. These midrange processors provide better performance with lower power consumption, thus are typically found in portable applications such as medical and healthcare devices like digital hearing aids.

These conventional and enhanced digital signal processors have the following features for common DSP algorithms:

- *Fast MAC units.* The multiply–add or multiply–accumulate operation is required in most DSP functions including filtering, fast Fourier transform, and correlation. To perform the MAC operation efficiently, digital signal processors integrate the multiplier and accumulator into the same data path to complete the MAC operation in a single instruction cycle.
- *Multiple memory accesses.* Most DSP processors adopted modified Harvard architectures that keep the program memory and data memory separate to allow simultaneous fetch of instruction and data. In order to support simultaneous access of multiple data words, digital signal processors provide multiple on-chip buses, independent memory banks, and on-chip dual-access data memory.
- *Special addressing modes.* digital signal processors often incorporate dedicated data address generation units for generating data addresses in parallel with the execution of

instructions. These units usually support circular addressing and bit-reversed addressing needed for some commonly used DSP algorithms.

- *Special program control.* Most digital signal processors provide zero-overhead looping, which allows the implementation of loops and repeat operations without extra clock cycles for updating and testing loop counters, or branching back to the top of the loop.
- *Optimized instruction set.* Digital signal processors provide special instructions that support computationally intensive DSP algorithms.
- *Effective peripheral interface.* Digital signal processors usually incorporate high-performance serial and parallel input/output (I/O) interfaces to other devices such as ADCs and DACs. They provide streamlined I/O handling mechanisms such as buffered serial ports, direct memory access (DMA) controllers, and low-overhead interrupt to transfer data with little or no intervention from the processor's computational units.

These digital signal processors use specialized hardware and complex instructions to allow more operations to be executed in a single instruction cycle. However, they are difficult to program using assembly language and it is difficult to design efficient C compilers in terms of speed and memory usage for supporting these complex instruction architectures.

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

1.3.3 Fixed- and Floating-Point Processors

A basic distinction between digital signal processors is the arithmetic format: fixed-point or floating-point. This is the most important factor for system designers to determine the suitability of the processor for the given application. The fixed-point representation of signals and arithmetic will be discussed in Chapter 2. Fixed-point digital signal processors are either 16-bit or 24-bit devices, while floating-point processors are usually 32-bit devices. A typical 16-bit fixed-point processor, such as the TMS320C55xx, stores numbers as 16-bit integers. Although coefficients and signals are stored only with 16-bit precision, intermediate values (products) may be kept at 32-bit precision within the internal 40-bit accumulators in order to reduce cumulative rounding errors. Fixed-point DSP devices are usually cheaper and faster than their floating-point counterparts because they use less silicon, have lower power consumption, and require fewer external pins. Most high-volume, low-cost embedded applications such as appliances, hard disk drives, audio players and digital cameras use fixed-point processors.

Floating-point arithmetic greatly expands the dynamic range of numbers. A typical 32-bit floating-point digital signal processor, such as the TMS320C67xx, represents numbers with a 24-bit mantissa and 8-bit exponent. The mantissa represents a fraction in the range -1.0 to $+1.0$, while the exponent is an integer that represents the number of binary points that must be shifted left or right in order to obtain the true value. A 32-bit floating-point format covers a large dynamic range, thus the data dynamic range restrictions may be virtually ignored in the design using floating-point processors. This is in contrast to the design of fixed-point systems, where the designer has to apply scaling factors and other techniques to prevent arithmetic overflows, which are very difficult and time-consuming processes. Therefore, floating-point digital signal processors are generally easy to program and use with higher performance, but are usually more expensive and have higher power consumption.

Example 1.4

The precision and dynamic ranges of 16-bit fixed-point processors are summarized in the following table:

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

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

System designers have to determine the dynamic range and precision needed for the applications. Floating-point processors may be needed in applications where coefficients vary in time and the signals and coefficients require large dynamic ranges and high precision. Floating-point processors also support the efficient use of high-level C compilers, thus reducing the cost of development and maintenance. The faster development cycle for floating-point processors may easily outweigh the extra cost of the processor itself for low-quantity products. Therefore, floating-point processors also can be justified for applications where development costs are high and/or production volumes are low.

1.3.4 Real-Time Constraints

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

$$t_p + t_o < T, \tag{1.6}$$

where t_o is the overhead of I/O operations.

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

$$f_M \le \frac{f_s}{2} < \frac{1}{2(t_p + t_o)}. \tag{1.7}$$

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

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

Equation (1.7) also shows that the real-time bandwidth can be increased by reducing the overhead of I/O operations. This can be achieved by using a block-by-block processing approach. With block processing methods, the I/O operations are usually handled by DMA controllers, which place data samples in memory buffers. The DMA controller interrupts the processor when the input buffer is full and the block of signal samples are available for processing. For example, for real-time N-point fast Fourier transforms (to be discussed in Chapter 5), the N input samples have to be buffered by the DMA controller. The block computation must be completed before the next block of N samples arrives. Therefore, the time delay between input and output in block processing is dependent on the block size N, and this may cause a problem for some applications.

1.4 DSP System Design

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

1.4.1 Algorithm Development

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

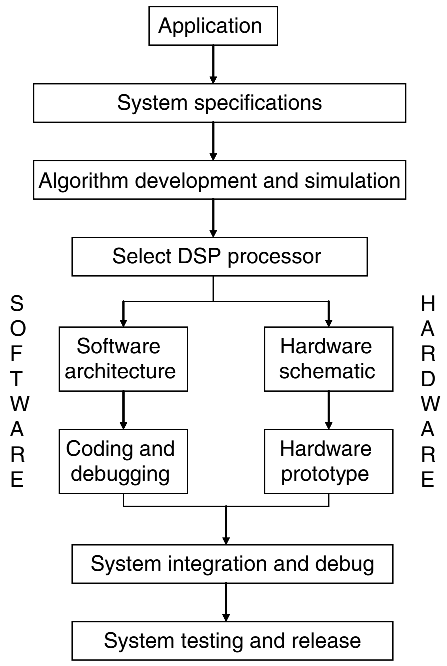


Figure 1.9 Simplified DSP system design flow

At the algorithm development stage, it is easier to use high-level language tools (such as MATLAB® or C/C++) for the algorithmic-level simulations. A DSP algorithm can be simulated using a general-purpose computer to test and analyze its performance. A block diagram of software development using a general-purpose computer is illustrated in Figure 1.10. The testing signals may be internally generated by signal generators, digitized from the experimental setup or real environment based on the given application, or received from other computers via the networks. The simulation program uses the signal samples stored in data file(s) as input(s) to produce output signal(s) that will be saved as data file(s) for further analysis.

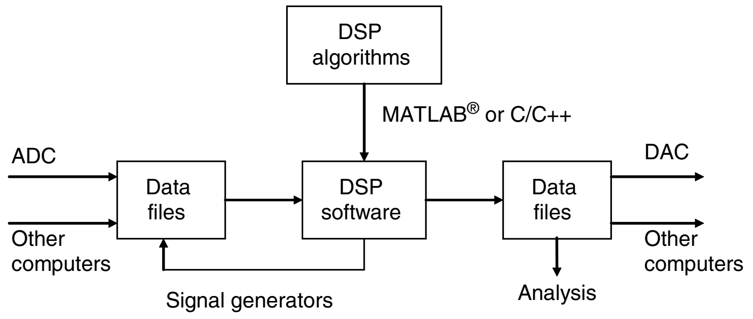


Figure 1.10 DSP software developments using a general-purpose computer

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

1. Using high-level languages such as MATLAB®, C/C++, or other DSP software packages on computers can significantly save algorithm development time. In addition, the prototype C programs used for algorithm evaluation can be ported to different DSP hardware platforms.
2. It is easier to debug and modify high-level language programs on computers using integrated software development tools.
3. I/O operations based on disk files are easy to implement and the behaviors of the system are easy to analyze.
4. Floating-point data formats and arithmetic can be used for computer simulations, thus ease of development.
5. Bit-true simulations of the developed algorithms can be performed using MATLAB® or C/C++ for fixed-point DSP implementation.

1.4.2 Selection of DSP Hardware

As discussed earlier, digital signal processors are used in a wide range of applications from high-performance radar systems to low-cost consumer electronics. DSP system designers require a full understanding of the application requirements in order to select the right DSP hardware for the given application. The objective is to choose the processor that meets the project's requirements with the most cost-effective solution [18]. Some decisions can be made at an early stage based on arithmetic format, performance, price, power consumption, ease of development and integration, and so on. For real-time DSP applications, the efficiency of data flow into and out of the processor is also critical.

Example 1.5

There are a number of ways to measure a processor's execution speed, as follows:

1. MIPS – Millions of instructions per second.
2. MOPS – Millions of operations per second.
3. MFLOPS – Millions of floating-point operations per second.
4. MHz – clock rate in mega hertz.
5. MMACS – Millions of multiply–accumulate operations.

In addition, there are other metrics to be considered such as milliwatts (mW) for measuring power consumption, MIPS per mW, or MIPS per dollar. These numbers provide a simple indication of performance, power, and price for the given application.

As discussed earlier, hardware cost and product manufacture integration are important factors for high-volume applications. For portable, battery-powered products, power consumption is more critical. For low- to medium-volume applications, there will be trade-offs among development time, cost of development tools, and the cost of the hardware itself. The likelihood of having higher performance processors with upwards-compatible software is also an important factor. For high-performance, low-volume applications such as communication infrastructures and wireless base stations, the performance, ease of development, and multiprocessor configurations are paramount.

Example 1.6

A number of DSP applications along with the relative importance for performance, price, and power consumption are listed in Table 1.2. This table shows, for handheld devices, that the primary concern is power efficiency; however, the main criterion for the communication infrastructures is performance.

When processing speed is at a premium, the only valid comparison between processors is on an algorithm implementation basis. Optimum code must be written for all candidates and then the execution time must be compared. Other important factors are memory usage and on-chip peripheral devices, such as on-chip converters and I/O interfaces. In addition, a full set of development tools and support listed as follows are important for digital signal processor selection:

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

1.4.3 Software Development

There are four common measures of good DSP software: reliability, maintainability, extensibility, and efficiency. A reliable program is one that seldom (or never) fails. Since most programs will occasionally fail, a maintainable program is one that is easy to correct. A truly maintainable program is one that can be fixed by someone other than the original programmers. An extensible program is one that can be easily modified when the requirements change. A good DSP program often contains many small functions with only one purpose, which can be easily reused by other programs for different purposes.

Table 1.2 Some DSP applications with the relative importance rating (adapted from [19])

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

Rating: 1 to 3 with 1 being the most important

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

The software life cycle involves the completion of the software project: namely, project definition, detailed specifications, coding and modular testing, system integration and testing, and product software maintenance. Software maintenance is a significant part of the cost for DSP systems. Maintenance includes enhancing the software functions, fixing errors identified as the software is used, and modifying the software to work with new hardware and software. It is important to use meaningful variable names in source code, and to document programs thoroughly with titles and comment statements because this greatly simplifies the task of software maintenance. Programming tricks should be avoided at all costs, as they will not be reliable and will be difficult for someone else to understand even with lots of comments.

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

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

There are two commonly used programming languages in developing DSP software: assembly language and C. Assembly language is similar to the machine code actually used by the processor. Programming in assembly language gives engineers full control of processor functions and resources, thus resulting in the most efficient program for mapping the algorithm by hand. However, this is a very time-consuming and laborious task, especially for today’s highly parallel processor architectures and complicated DSP algorithms. C language, on the other hand, is easier for software development, upgrading, and maintenance. However, the machine code generated by the C compiler is often inefficient in both processing speed and memory usage.

Often the best solution is to use a mixture of C and assembly code. The overall program is written using C, but the runtime critical inner loops and modules are replaced by assembly code. In a mixed programming environment, an assembly routine may be called as a function or intrinsics, or in-line coded into the C program. A library of hand-optimized functions may be built up and brought into the code when required.

1.4.4 Software Development Tools

Most DSP operations can be categorized as being either signal analysis or filtering. Signal analysis deals with the measurement of signal properties. MATLAB® is a powerful tool for signal analysis and visualization, which are critical components in understanding and

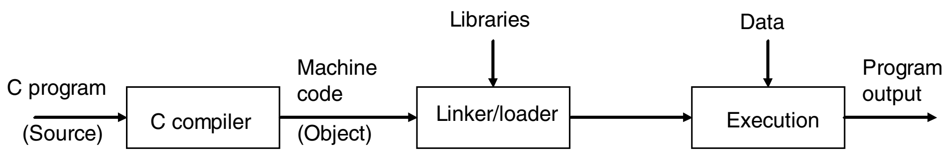


Figure 1.11 Program compilation, linking, and execution flow

developing DSP systems. C is an efficient tool for performing signal processing and is portable over different DSP platforms.

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

The purpose of programming languages is to solve problems involving the manipulation of information. The purpose of DSP programs is to manipulate signals to solve specific signal processing problems. High-level languages such as C/C++ are usually portable, so they can be recompiled and run on many different computer platforms. Although C/C++ is categorized as a high-level language, it can also be used for low-level device drivers. In addition, C compilers are available for most modern digital signal processors. Thus, C programming is the most commonly used high-level language for DSP applications.

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

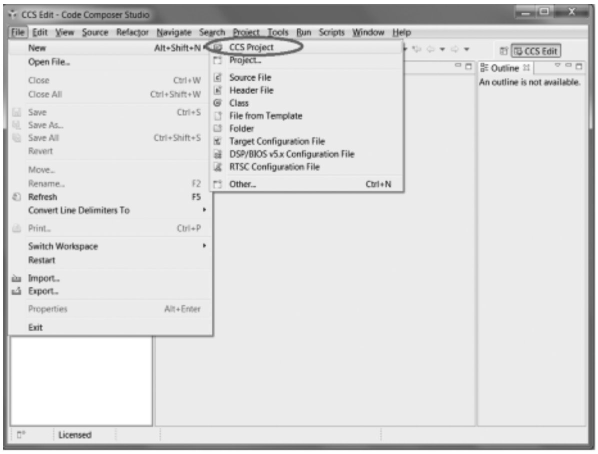
1.5 Experiments and Program Examples

The Code Composer Studio (CCS) is an integrated development environment for DSP applications. CCS has several built-in tools for software development including project build environment, source code editor, C/C++ compiler, debugger, profiler, simulator, and real-time operating system. CCS allows users to create, edit, build, debug, and analyze software programs. It also provides a project manager to handle multiple programming projects for building large applications. For software debugging, CCS supports breakpoints, watch windows for monitoring variables, memory, and registers, graphical display and analysis, program execution profiling, and display assembly and C instructions for single-step instruction traces.

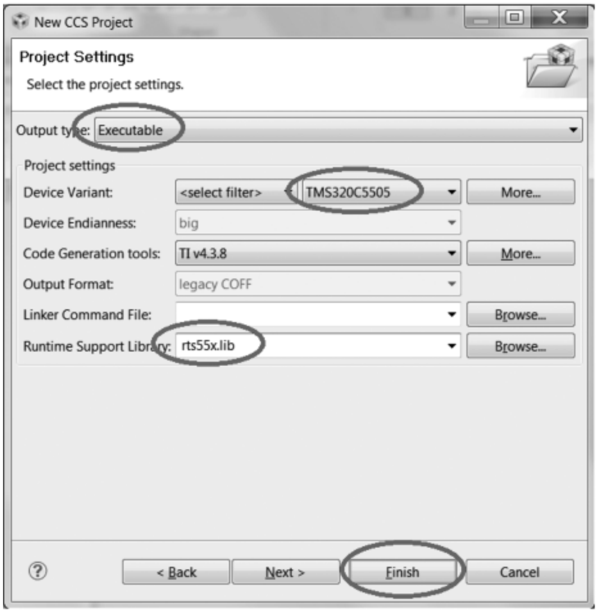
This section uses experiments to introduce several key CCS features including basic editing, memory configuration, and compiler and linker settings for building programs. We will demonstrate DSP software development and debugging processes using CCS with the low-cost TMS320C5505 eZdsp USB stick. Finally, we will present real-time audio experiments using eZdsp, which will be used as prototypes for building real-time experiments throughout the book. To conduct these real-time experiments, we have to connect the eZdsp to a USB port of a computer with CCS installed.

1.5.1 Get Started with CCS and eZdsp

In this book, we use the C5505 eZdsp with CCS version 5.x for all experiments. To learn some basic features of CCS, perform the following steps to complete the experiment Exp1.1.



(a) Create the CCS project, File→New→CCS Project.


New CCS Project

Project Settings

Select the project settings.

Output type: Executable

Project settings

Device Variant: <select filter> TMS320C5505

Device Endianness: big

Code Generation tools: TI v4.3.8

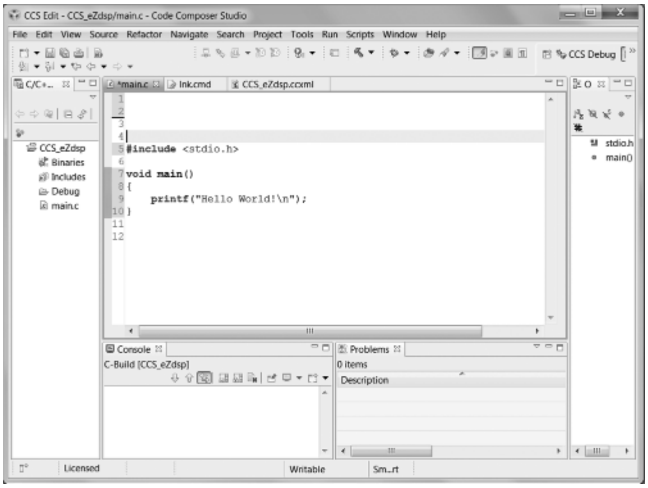
Output Format: legacy COFF

Linker Command File:

Runtime Support Library: rts55x.lib

(b) Select executable and `rts55x.lib` library.

Figure 1.12 Create a CCS project



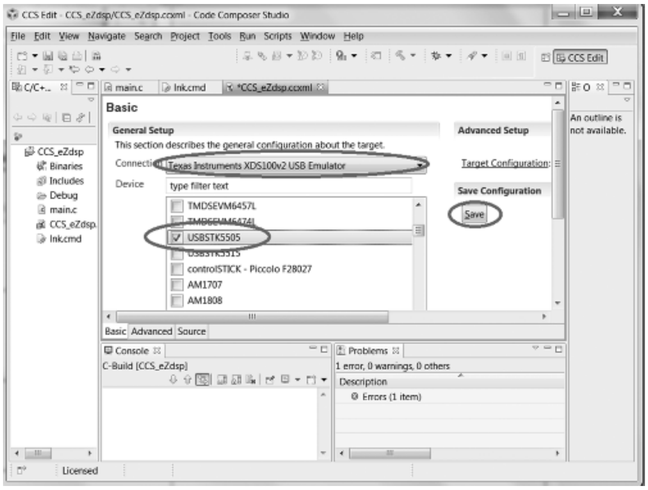
```
#include <stdio.h>

void main()
{
    printf("Hello World!\n");
}
```

Figure 1.13 C program for the experiment Exp1.1

Step 1, start CCS from the host computer and create the C5505 CCS project as shown in Figure 1.12. In this experiment, we use `CCS_eZdsp` as the project name, select executable output type, and use `rts55x` runtime support library.

Step 2, create a C program under the CCS project and name the C file as `main.c` via **File→New→Source File**. Then, use the CCS text editor to write the C program to display "Hello World" as shown in Figure 1.13.

Step 3, create the target configuration file for the C5505 eZdsp. Start from **File→New→Target Configuration File**, select `XDS100v2 USB Emulator` and `USBSTK5505` as the target configuration and save the changes, see Figure 1.14.



Basic

General Setup

This section describes the general configuration about the target.

Connection: Texas Instruments XDS100v2 USB Emulator

Device: type filter text

- [ ] TMDSEVM6457L
- [ ] TMDSEVM6474L
- [x] USBSTK5505
- [ ] USBSTK5515
- [ ] controlSTICK - Piccolo F28027
- [ ] AM1707
- [ ] AM1808

Advanced Setup

Target Configuration:

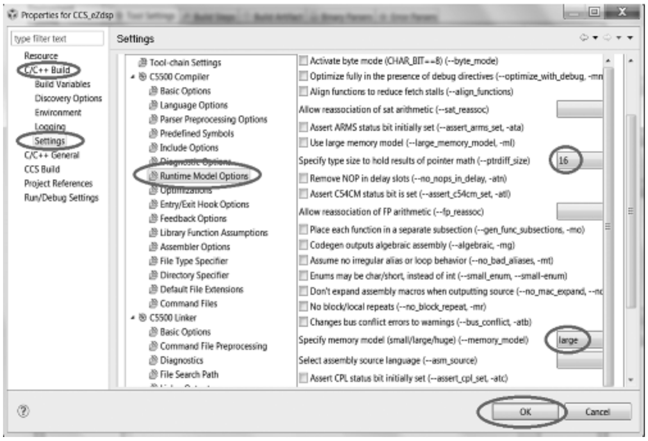
Save Configuration

Save

1 error, 0 warnings, 0 others

Errors (1 item)

Figure 1.14 Create the target configuration



☐ Activate byte mode (CHAR_BIT==8) (--byte_mode)
☐ Optimize fully in the presence of debug directives (--optimize_with_debug, -mn
☐ Align functions to reduce fetch stalls (--align_functions)
Allow reassociation of sat arithmetic (--sat_reassoc)
☐ Assert ARMS status bit initially set (--assert_arms_set, -ata)
☐ Use large memory model (--large_memory_model, -ml)
Specify type size to hold results of pointer math (--ptrdiff_size) [✓ 16]
☐ Remove NOP in delay slots (--no_nops_in_delay, -atn)
☐ Assert C54CM status bit is set (--assert_c54cm_set, -atl)
Allow reassociation of FP arithmetic (--fp_reassoc)
☐ Place each function in a separate subsection (--gen_func_subsections, -mo)
☐ Codegen outputs algebraic assembly (--algebraic, -mg)
☐ Assume no irregular alias or loop behavior (--no_bad_aliases, -mt)
☐ Enums may be char/short, instead of int (--small_enum, --small-enum)
☐ Don't expand assembly macros when outputting source (--no_mac_expand, --nc
☐ No block/local repeats (--no_block_repeat, -mr)
☐ Changes bus conflict errors to warnings (--bus_conflict, -atb)
Specify memory model (small/large/huge) (--memory_model) [✓ large]
Select assembly source language (--asm_source)
☐ Assert CPL status bit initially set (--assert_cpl_set, -atc)

Figure 1.15 Setting the CCS project runtime environment

Step 4, set up the CCS environment. Open the property of the C5505 project we have created by right-clicking on the project, **CCS_eZdsp→Properties**, under the **Resource** window, select and open the **C/C++ Build→Settings→Runtime Model Options**, then set up for large memory model and 16-bit pointer math as shown in Figure 1.15.

Step 5, add the C5505 linker command file. Use the text editor to create the linker command file `c5505.cmd` as listed in Table 1.3, which will be discussed later. This file is available in the companion software package.

Step 6, connect the CCS to the target device. Go to **View→Target Configurations** to open the **Target Configuration** window, select the experiment target, right-click on it to launch the

Table 1.3 Linker command file, `c5505.cmd`

```
-stack    0x2000       /* Primary stack size   */
-sysstack 0x1000       /* Secondary stack size */
-heap     0x2000       /* Heap area size       */
-c                     /* Use C linking conventions: auto-init vars at
                          runtime */
-u _Reset              /* Force load of reset interrupt handler */

MEMORY
{
   MMR     (RW) : origin = 0000000h length = 0000c0h /* MMRs */
   DARAM   (RW) : origin = 00000c0h length = 00ff40h /* On-chip DARAM */
   SARAM   (RW) : origin = 0030000h length = 01e000h /* On-chip SARAM */

   SAROM_0 (RX) : origin = 0fe0000h length = 008000h /* On-chip ROM 0 */
   SAROM_1 (RX) : origin = 0fe8000h length = 008000h /* On-chip ROM 1 */
   SAROM_2 (RX) : origin = 0ff0000h length = 008000h /* On-chip ROM 2 */
   SAROM_3 (RX) : origin = 0ff8000h length = 008000h /* On-chip ROM 3 */
}
```

Table 1.3 (*Continued*)

```
SECTIONS
{
   vectors (NOLOAD)
   .bss          : > DARAM     /* Fill = 0 */
   vector        : > DARAM       ALIGN = 256
   .stack        : > DARAM
   .sysstack     : > DARAM
   .sysmem       : > DARAM
   .text         : > SARAM
    .data        : > DARAM
    .cinit       : > DARAM
    .const       : > DARAM
    .cio         : > DARAM
    .usect       : > DARAM
   .switch       : > DARAM
}
```

target configuration, then right-click on the USB Emulator 0/C55xx and select **Connect Target**, see Figure 1.16.

Step 7, build, load, and run the experiment. From **Project→Build All**, after the build is completed without error, load the executable program from **Run→Load→Load Program**, see Figure 1.17. When CCS prompts for the program to be loaded, navigate to the project folder and load the C5505 executable file (e.g., `CCS_eZdsp.out`) from the Debug folder.

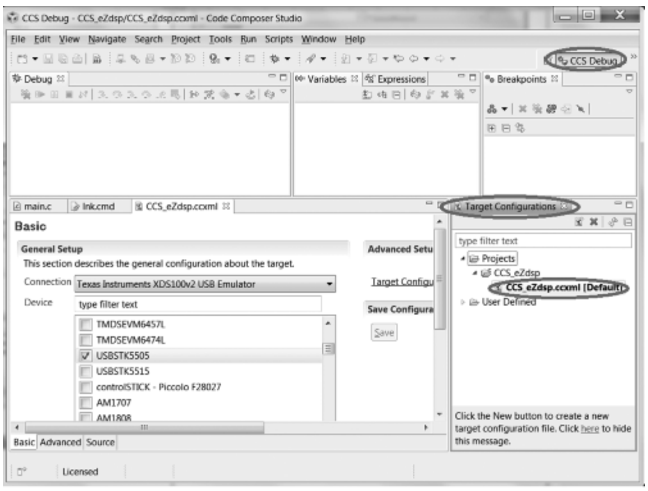
As shown in Table 1.3, the linker command file, `c5505.cmd`, defines the C55xx system memory for the target device and specifies the locations of program memory, data memory, and I/O memory. The linker command file also describes the starting locations of memory blocks and the length of each block. More information on the hardware specific linker command file can be found in the C5505 data sheet [20]. Table 1.4 lists the files used for the experiment Exp1.1.

Procedures of the experiment are listed as follows:

1. Follow the experiment steps presented in this section to create a CCS workspace for Exp1.1.
2. Remove the linker command file `c5505.com` from the project. Rebuild the experiment. There will be warning messages displayed. CCS generates these warnings because it uses default settings to map the program and data to the processor's memory spaces when the linker command file is missing.
3. Load `CCS_eZdsp.out`, and use **Step Over** (F6) through the program. Then, use CCS **Reload Program** to load the program again. Where is the program counter (cursor) location?

Table 1.4 File listing for the experiment Exp1.1

Files	Description
`main.c`	C source file for experiment
`c5505.cmd`	Linker command file

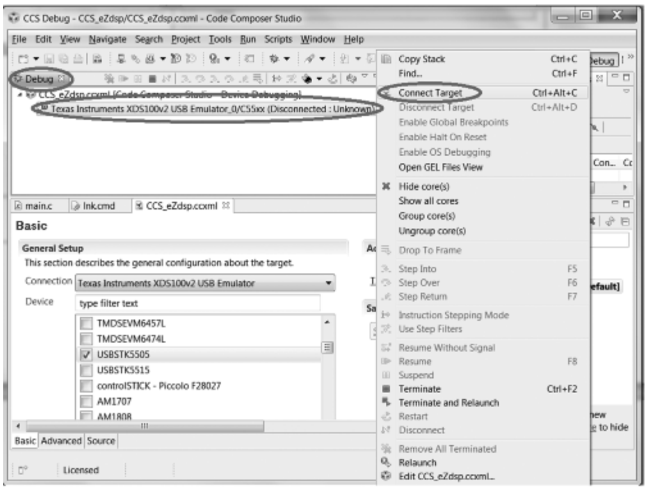

CCS Debug - CCS_eZdsp/CCS_eZdsp.ccxml - Code Composer Studio
File Edit View Navigate Search Project Tools Run Scripts Window Help
CCS Debug
Debug
Variables
Expressions
Breakpoints
main.c
lnk.cmd
CCS_eZdsp.ccxml
Target Configurations
Basic
General Setup
This section describes the general configuration about the target.
Advanced Setu
Target Configu
Save Configura
Save
type filter text
Projects
CCS_eZdsp
CCS_eZdsp.ccxml [Default]
User Defined
Click the New button to create a new target configuration file. Click here to hide this message.
Basic Advanced Source
Licensed

Connection: Texas Instruments XDS100v2 USB Emulator

Device: type filter text

- [ ] TMDSEVM6457L
- [ ] TMDSEVM6474L
- [✓] USBSTK5505
- [ ] USBSTK5515
- [ ] controlSTICK - Piccolo F28027
- [ ] AM1707
- [ ] AM1808

(a) Open target configuration window.



Connection: Texas Instruments XDS100v2 USB Emulator

Device: type filter text

- [ ] TMDSEVM6457L
- [ ] TMDSEVM6474L
- [✓] USBSTK5505
- [ ] USBSTK5515
- [ ] controlSTICK - Piccolo F28027
- [ ] AM1707
- [ ] AM1808

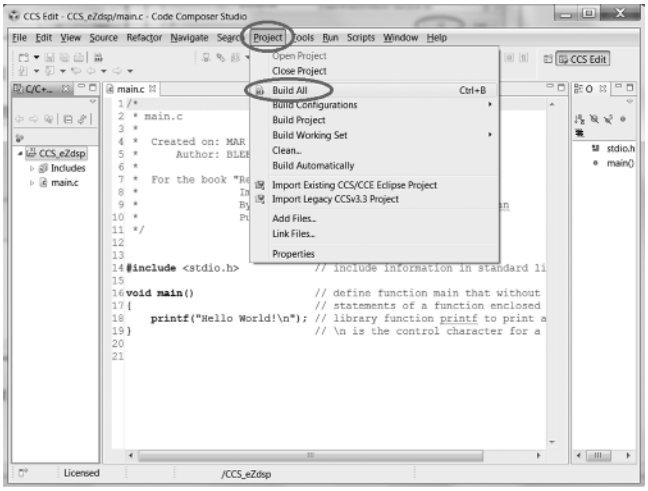
(b) Connect the CCS to target device.

Figure 1.16 Connect CCS with the target device, C5505 eZdsp

4. Use **Resume** (F8) instead of **Step Over** (F6) to run the program again. What will be showing on the console display window? Observe the differences from step 3.
5. After running the program, use **Restart** and **Resume** (F8) to run the program again. What will be showing on the console display window?

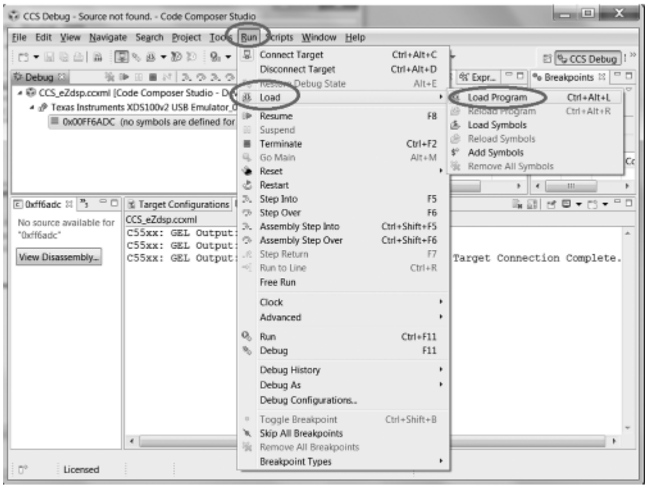
1.5.2 C File I/O Functions

We can use C file I/O functions to access the ASCII formatted or binary formatted data files that contain input signals to simulate DSP applications. The binary data file is more efficient for storage and access, while the ASCII data format is easy for the user to read and check. In



```
/*
 * main.c
 *
 *  Created on: MAR
 *      Author: BLEE
 *
 *  For the book "Re
 *               In
 *               By
 *               Pu
 */


#include <stdio.h>             // include information in standard li

void main()                    // define function main that without
{                              // statements of a function enclosed
    printf("Hello World!\n");  // library function printf to print a
}                              // \n is the control character for a
```

(a) Build the CCS project.


CCS Debug - Source not found. - Code Composer Studio
File Edit View Navigate Search Project Tools Run Scripts Window Help
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
Restore Debug State Alt+E
Load
Resume F8
Suspend
Terminate Ctrl+F2
Go Main Alt+M
Reset
Restart
Step Into F5
Step Over F6
Assembly Step Into Ctrl+Shift+F5
Assembly Step Over Ctrl+Shift+F6
Step Return F7
Run to Line Ctrl+R
Free Run
Clock
Advanced
Run Ctrl+F11
Debug F11
Debug History
Debug As
Debug Configurations...
Toggle Breakpoint Ctrl+Shift+B
Skip All Breakpoints
Remove All Breakpoints
Breakpoint Types
Load Program Ctrl+Alt+L
Reload Program Ctrl+Alt+R
Load Symbols
Reload Symbols
Add Symbols
Remove All Symbols
CCS Debug
Debug
CCS_eZdsp.ccxml [Code Composer Studio - D
Texas Instruments XDS100v2 USB Emulator_0
0x00FF6ADC (no symbols are defined for
0xff6adc
No source available for "0xff6adc"
View Disassembly...
Target Configurations
CCS_eZdsp.ccxml
Expr...
Breakpoints
Licensed

```
C55xx: GEL Output:
C55xx: GEL Output:
C55xx: GEL Output:
Target Connection Complete.
```

(b) Load the executable file to eZdsp.

Figure 1.17 Build, load, and run the experiment using CCS

practical applications, digitized data files are often stored in binary format to reduce memory requirements. In this section, we will introduce the C file I/O functions provided by CCS libraries.

CCS supports standard C library functions such as `fopen`, `fclose`, `fread`, `fwrite` for file I/O operations. These C file I/O functions are portable to other development environments.

The C language supports different data types. To improve program portability, we use the unique type definition header file, `tistdtypes.h`, to specify the data types to avoid any ambiguity.

Table 1.5 lists the C program that uses `fopen`, `fclose`, `fread`, and `fwrite` functions. The input data is a linear PCM (Pulse Code Modulation) audio signal stored in a binary file. Since

Table 1.5 C program using file I/O functions, `fielIO.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "tistdtypes.h"
Uint8 waveHeader[44]={                           /* 44 bytes for WAV file
                                                    header */
      0x52,  0x49,  0x46,  0x46,  0x00,  0x00,  0x00,  0x00,
      0x57,  0x41,  0x56,  0x45,  0x66,  0x6D,  0x74,  0x20,
      0x10,  0x00,  0x00,  0x00,  0x01,  0x00,  0x01,  0x00,
      0x40,  0x1F,  0x00,  0x00,  0x80,  0x3E,  0x00,  0x00,
      0x02,  0x00,  0x10,  0x00,  0x64,  0x61,  0x74,  0x61,
      0x00,  0x00,  0x00,  0x00};
#define SIZE 1024
Uint8 ch[SIZE];                                  /* Declare a char[1024]
                                                    array for experiment */
void main()
{
   FILE *fp1,*fp2;                               /* File pointers */
   Uint32 i;                                     /* Unsigned long integer
                                                    used as a counter */
   printf("Exp. 1.2 --- file IO\n");
   fp1 = fopen("..\\data\\C55DSPUSBStickAudioTest.pcm", "rb");
                                                 /* Open input file */
   fp2 = fopen("..\\data\\C55DSPUSBStickAudioTest.wav", "wb");
                                                 /* Open output file */
   if (fp1 == NULL)                              /* Check if the input file
                                                    exists */
   {
       printf("Failed to open input file 'C55DSPUSBStickAudioTest.
pcm'\n");
       exit(0);
   }
   fseek(fp2, 44, 0);                            /* Advance output file
                                                    point 44 bytes */
   i=0;
   while (fread(ch, sizeof(Uint8), SIZE, fp1) == SIZE)
                                                 /* Read in SIZE of input
                                                    data bytes */
   {
       fwrite(ch, sizeof(Uint8), SIZE, fp2);     /* Write SIZE of data bytes
                                                    to output file */
       i += SIZE;
       printf("%ld bytes processed\n", i);       /* Show the number of data
                                                    is processed */
   }
   waveHeader[40] = (Uint8)(i&0xff);             /* Update the size
                                                    parameter into WAV
                                                    header */
```

Table 1.5 (*Continued*)

```
    waveHeader[41] = (Uint8)(i>>8)&0xff;
    waveHeader[42] = (Uint8)(i>>16)&0xff;
    waveHeader[43] = (Uint8)(i>>24)&0xff;
    waveHeader[4] = waveHeader[40];
    waveHeader[5] = waveHeader[41];
    waveHeader[6] = waveHeader[42];
    waveHeader[7] = waveHeader[43];

    rewind(fp2);                                /* Adjust output file
                                                   point to beginning */
    fwrite(waveHeader, sizeof(Uint8), 44, fp2); /* Write 44 bytes of WAV
                                                   header to output
                                                   file */

    fclose(fp1);                                /* Close input file */
    fclose(fp2);                                /* Close output file */

    printf("\nExp. completed\n");
}
```

the C55xx CCS file I/O libraries support only byte formatted binary data (`char`, 8-bit), the 16-bit PCM data file can be read using `sizeof(char)`, and the output wave (WAV) data file can be written by CCS in byte format [21–23]. For 16-bit short-integer data types, each data read or data write requires two memory accesses. As shown in Table 1.5, the program reads and writes 16-bit binary data in byte units. To run this program on a computer, the data access can be changed to its native data type `sizeof(short)`. The output file of this experiment is a WAV file that can be played by many audio players. Note that the WAV file format supports several different file types and sampling rates. The files used for the experiment are listed in Table 1.6.

Procedures of the experiment are listed as follows:

1. Create CCS workspace for the experiment Exp1.2.
2. Create a C5505 project using `fileIO` as the project name.
3. Copy `fileIOTest.c`, `tistdtypes.h`, and `c5505.cmd` from the companion software package to the experiment folder.
4. Create a data folder under the experiment folder and place the input file `C55DSPUSB-StickAudioTest.pcm` into the data folder.

Table 1.6 File listing for the experiment Exp1.2

Files	Description
`fileIOTest.c`	Program file for testing C file I/O
`tistdtypes.h`	Data type definition header file
`c5505.cmd`	Linker command file
`C55DSPUSBStickAudioTest.pcm`	Audio data file for experiment

5. Set up the CCS project build and debug environment using the 16-bit data format and large runtime support library `rts55x.lib`.
6. Set up the target configuration file `fileIO.ccxml` for using the eZdsp.
7. Build and load the experiment executable file. Run the experiment to generate the output audio file, `C55DSPUSBStickAudioTest.wav`, saved in the data folder. Listen to the audio file using an audio player.
8. Modify the experiment such that it can achieve the following tasks:
   (a) Read the input data file `C55DSPUSBStickAudioTest.pcm` and write an output file in ASCII integer format in `C55DSPUSBStickAudioTest.xls` (or another file format instead of `.xls`). (Hint: replace the `fwrite` function with `fprintf`.)
   (b) Use Microsoft Excel (or other software such as MATLAB®) to open the file `C55DSPUSBStickAudioTest.xls`, select the data column, and plot the waveform of the audio.
9. Modify the experiment to read "`C55DSPUSBStickAudioTest.xls`" created in the previous step as the input file and write it out in a WAV file. Listen to the WAV file to verify it is correct.

1.5.3 User Interface for eZdsp

An interactive user interface is very useful for developing real-time DSP applications. It provides the flexibility to change runtime parameters without the need to stop execution, modify, recompile, and rerun the program. This feature becomes more important for large-scale projects that consist of many C programs and prebuilt libraries. In this experiment, we use the `scanf` function to get interactive input parameters through the CCS console window. We also introduce some commonly used CCS debugging methods including software breakpoints, viewing processor's memory and program variables, and graphical plots.

Table 1.7 lists the C program that uses `fscan` function to read user parameters via the CCS console window. This program reads the parameters and verifies their values. The program will replace any invalid value with the default value. This experiment has three user-defined parameters: gain `g`, sampling frequency `sf`, and playtime duration `p`. The files used for the experiment are listed in Table 1.8.

Table 1.7 C program with interactive user interface, `UITest.c`

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

#define SIZE 48
Int16 dataTable[SIZE];

void main()
{
    /* Pre-generated sine wave data, 16-bit signed samples */
    Int16 table[SIZE] = {
      0x0000, 0x10b4, 0x2120, 0x30fb, 0x3fff, 0x4dea, 0x5a81, 0x658b,
      0x6ed8, 0x763f, 0x7ba1, 0x7ee5, 0x7ffd, 0x7ee5, 0x7ba1, 0x76ef,
      0x6ed8, 0x658b, 0x5a81, 0x4dea, 0x3fff, 0x30fb, 0x2120, 0x10b4,
```

Table 1.7 (*Continued*)

```
    0x0000, 0xef4c, 0xdee0, 0xcf06, 0xc002, 0xb216, 0xa57f, 0x9a75,
    0x9128, 0x89c1, 0x845f, 0x811b, 0x8002, 0x811b, 0x845f, 0x89c1,
    0x9128, 0x9a76, 0xa57f, 0xb216, 0xc002, 0xcf06, 0xdee0, 0xef4c
  };

  Int16 g,p,i,j,k,n,m;
  Uint32 sf;

  printf("Exp. 1.3 --- UI\n");

  printf("Enter an integer number for gain between (-6 and 29)\n");
  scanf ("%d", &g);

  printf("Enter the sampling frequency, select one: 8000, 12000,
  16000, 24000 or 48000\n");
  scanf ("%ld", &sf);

  printf("Enter the playing time duration (5 to 60)\n");
  scanf ("%i", &p);

  if ((g < -6)||(g > 29))
  {
     printf("You have entered an invalid gain\n");
     printf("Use default gain = 0dB\n");
     g = 0;
  }
  else
  {
     printf("Gain is set to %ddB\n", g);
  }
  if ( (sf == 8000)||(sf == 12000)||(sf == 16000)||(sf == 24000)||(sf ==
  48000))
  {
     printf("Sampling frequency is set to %ldHz\n", sf);
  }
  else
  {
     printf("You have entered an invalid sampling frequency\n");
     printf("Use default sampling frequency = 48000 Hz\n");
     sf = 48000;
  }
  if ((p < 5)||(p > 60))
  {
     printf("You have entered an invalid playing time\n");
     printf("Use default duration = 10s\n");
     p = 10;
  }
```

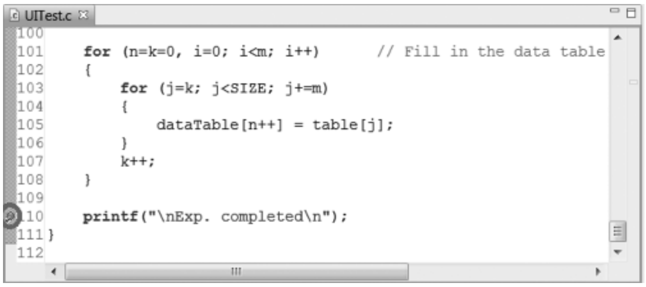
(*continued*)

Table 1.7 (*Continued*)

```
    else
    {
       printf("Playing time is set to %ds\n", p);

     }
    for (i=0; i<SIZE; i++)
       dataTable[i] = 0;

    switch (sf)
    {
       case 8000:
          m = 6;
          break;
       case 12000:
          m = 4;
          break;
       case 16000:
          m = 3;
          break;
       case 24000:
          m = 2;
          break;
       case 48000:
       default:
          m = 1;
          break;
    }

    for (n=k=0, i=0; i<m; i++)      // Fill in the data table
    {
      for (j=k; j<SIZE; j+=m)
      {
         dataTable[n++] = table[j];
      }
      k++;
    }

    printf("\nExp. completed\n");
}
```

Table 1.8 File listing for the experiment Exp1.3

Files	Description
`UITest.c`	Program file for testing user interface
`tistdtypes.h`	Data type definition header file
`c5505.cmd`	Linker command file



```
100
101     for (n=k=0, i=0; i<m; i++)      // Fill in the data table
102     {
103         for (j=k; j<SIZE; j+=m)
104         {
105             dataTable[n++] = table[j];
106         }
107         k++;
108     }
109
110     printf("\nExp. completed\n");
111 }
112
```

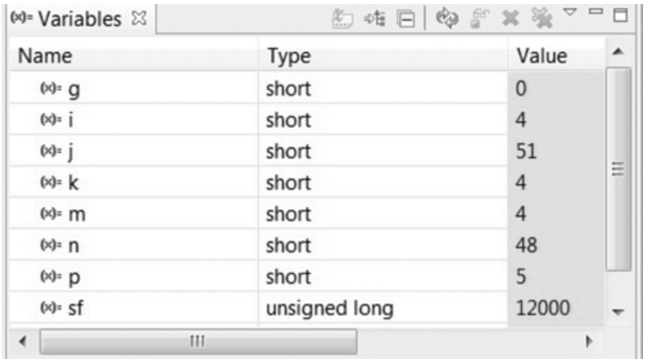
Figure 1.18 Setting a software breakpoint

CCS has many built-in tools including software breakpoints, watch windows, and graphic plots for debugging, testing, and evaluating programs. When the program reaches a software breakpoint, it will halt program execution at that location. CCS will preserve the processor's register and system memory values at that instant for users to validate the results. The software breakpoint can be set by double-clicking on the left sidebar of an instruction. Figure 1.18 shows the breakpoint is set on line number 110. Once the program runs to the breakpoint, it stops, then the user can step over the program statements, or step into the function if this line contains another function. The breakpoint can be removed by double-clicking on the breakpoint itself.

Once the program hits the breakpoint, we can use watch windows to examine registers and data variables. For example, we can use the CCS viewing feature to display data variables such as `g` and `sf` from the CCS menu **View→Variables**, see Figure 1.19.

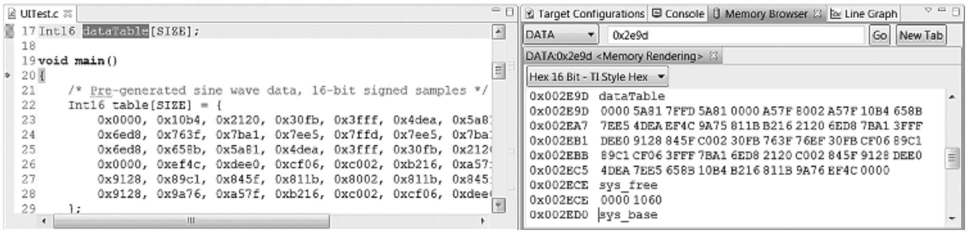
We can also view data values stored in memory. To view memory, we open the memory watch window from **View→Memory Browser**. Figure 1.20 shows the CCS memory watch window that contains the data values stored in `dataTable[SIZE]` at data memory address 0x2E9D.

We can also use the CCS graphical tools to plot the data for visual examination. For this experiment, we activate the plot tools from **Tool→Graph→Single Time**. Open the graph properties setting dialog window, set **Acquisition Buffer** to 48 (table size), select **Data Type** as 16-bit signed integer (based on the data type), set the data **Start Address** to `dataTable` (data memory address), and finally set **Display Data Size** to 48. Figure 1.21 displays the graph of the sinusoidal data stored in the 16-bit integer array `dataTable[SIZE]`.



Name	Type	Value
g	short	0
i	short	4
j	short	51
k	short	4
m	short	4
n	short	48
p	short	5
sf	unsigned long	12000

Figure 1.19 Variable watch window



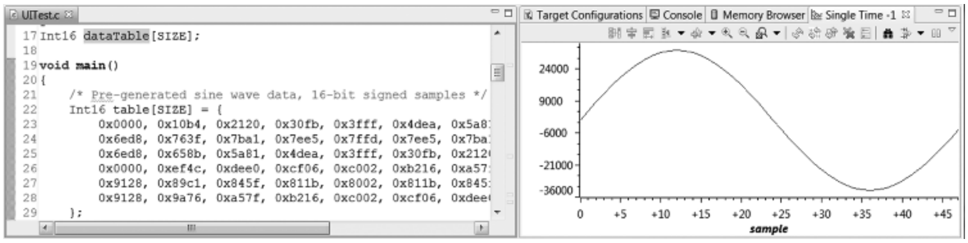
```
Int16 dataTable[SIZE];

void main()
{
    /* Pre-generated sine wave data, 16-bit signed samples */
    Int16 table[SIZE] = {
        0x0000, 0x10b4, 0x2120, 0x30fb, 0x3fff, 0x4dea, 0x5a8
        0x6ed8, 0x763f, 0x7ba1, 0x7ee5, 0x7ffd, 0x7ee5, 0x7ba
        0x6ed8, 0x658b, 0x5a81, 0x4dea, 0x3fff, 0x30fb, 0x212
        0x0000, 0xef4c, 0xdee0, 0xcf06, 0xc002, 0xb216, 0xa57
        0x9128, 0x89c1, 0x845f, 0x811b, 0x8002, 0x811b, 0x845
        0x9128, 0x9a76, 0xa57f, 0xb216, 0xc002, 0xcf06, 0xdee
    };
```

```
0x002E9D  dataTable
0x002E9D  0000 5A81 7FFD 5A81 0000 A57F 8002 A57F 10B4 658B
0x002EA7  7EE5 4DEA EF4C 9A75 811B B216 2120 6ED8 7BA1 3FFF
0x002EB1  DEE0 9128 845F C002 30FB 763F 76EF 30FB CF06 89C1
0x002EBB  89C1 CF06 3FFF 7BA1 6ED8 2120 C002 845F 9128 DEE0
0x002EC5  4DEA 7EE5 658B 10B4 B216 811B 9A76 EF4C 0000
0x002ECE  sys_free
0x002ECE  0000 1060
0x002ED0  sys_base
```

Figure 1.20 Memory watch window

Procedures of the experiment are as follows:

1. Create the CCS project, set up the project build and debug environment, and use `UI` as the project name.
2. Create the C source file `UITest.c` as listed in Table 1.7, which is available in the companion software package. Add linker command file `c5505.cmd` and header file `tistdtypes.h`.
3. Connect the eZdsp to the computer, and set up the proper target configuration file for the eZdsp. Build and load the executable program for experiment.
4. Perform single-step operation to check the program.
5. Set some software breakpoints in `UITest.c`, and step through the program to observe the variable values of `g`, `sf`, and `p`.
6. Rerun the experiment and use the CCS graphical tool to plot the data stored in the array `dataTable[]` for different sampling frequencies at 8000, 12 000, 16 000, 32 000, and 48 000 Hz. Show these plots and compare their differences.
7. Set up the CCS variable watch window and examine what other data types can be supported by the watch window. Change the data type and observe how the watch window displays different data types.
8. How does one find a variable memory's address for setting up the watch windows? Set up both data and variable watch windows, single step through the program, and watch how the variables are updated and displayed on the watch windows.
9. Data values in the memory can be modified via CCS by directly editing the memory locations. Try to change the variable values and rerun the program.
10. CCS can plot different kinds of graphs. Select the graph parameters to plot the array `data` in the same window with the following settings:



```
Int16 dataTable[SIZE];

void main()
{
    /* Pre-generated sine wave data, 16-bit signed samples */
    Int16 table[SIZE] = {
        0x0000, 0x10b4, 0x2120, 0x30fb, 0x3fff, 0x4dea, 0x5a8
        0x6ed8, 0x763f, 0x7ba1, 0x7ee5, 0x7ffd, 0x7ee5, 0x7ba
        0x6ed8, 0x658b, 0x5a81, 0x4dea, 0x3fff, 0x30fb, 0x212
        0x0000, 0xef4c, 0xdee0, 0xcf06, 0xc002, 0xb216, 0xa57
        0x9128, 0x89c1, 0x845f, 0x811b, 0x8002, 0x811b, 0x845
        0x9128, 0x9a76, 0xa57f, 0xb216, 0xc002, 0xcf06, 0xdee
    };
```

Figure 1.21 Use graphical tool to plot data samples

(a) Add *x*–*y* axis labels to the plot.
(b) Add a grid to the graph plot.
(c) Change the display from line to large square.

1.5.4 Audio Playback Using eZdsp

The C programs presented in previous experiments can be executed on the C55xx simulator which comes with the CCS or hardware devices such as the C5505 eZdsp. In this experiment, we use the eZdsp for real-time experiments. The C5505 eZdsp has a standard USB interface to connect with the host computer for program development, debugging, algorithm evaluation and analysis, and real-time demonstration.

CCS comes with many useful supporting functions and programs including device drivers specifically written for C55xx processors and the eZdsp. The latter comes with an installation CD which includes the `C55xx_csl` folder containing files for the C55xx chip support libraries and the `USBSTK_bsl` folder containing files for the eZdsp board support libraries. The experiments given in Appendix C provide detailed descriptions of the chip support libraries and board support libraries. In this experiment, we modify the user interface experiment presented in Exp1.3 to control audio playback using the C55xx board support library and chip support library. This experiment plays audible tones using the eZdsp, and allows the user to control the output volume, sampling frequency, and play time from the CCS console window. The user interface C program, `playToneTest.c`, asks the user to enter three parameters for the experiment: gain (`gain)`, sampling frequency (`sf`), and time duration (`playtime`). After receiving the user parameters, the program generates a tone table, and calls the function `tone` to play the tone in real time using the eZdsp.

After the user parameters are passed to the function `tone` listed in Table 1.9, it calls the function `USBSTK5505_init` to initialize the eZdsp and the function `AIC3204` to initialize the analog interface chip AIC3204. Once enabled, AIC3204 will operate at the user-specified

Table 1.9 C program for real-time tone playback, `tone.c`

```
#define AIC3204_I2C_ADDR 0x18
#include "usbstk5505.h"
#include "usbstk5505_gpio.h"
#include "usbstk5505_i2c.h"
#include <stdio.h>

extern void aic3204_tone_headphone();
extern void tone(Uint32, Int16, Int16, Uint16, Int16*);
extern void Init_AIC3204(Uint32 sf, Int16 gDAC, Uint16 gADC);

void tone(Uint32 sf, Int16 playtime, Int16 gDAC, Uint16 gADC, Int16
*sinetable)
{
    Int16 sec, msec;
    Int16 sample, len;
```

Table 1.9 (*Continued*)

```
/* Initialize BSL */
USBSTK5505_init();

/* Set A20_MODE for GPIO mode */
CSL_FINST(CSL_SYSCTRL_REGS->EBSR, SYS_EBSR_A20_MODE, MODE1);

/* Use GPIO to enable AIC3204 chip */
USBSTK5505_GPIO_init();
USBSTK5505_GPIO_setDirection(GPIO26, GPIO_OUT);
USBSTK5505_GPIO_setOutput( GPIO26, 1);    /*Take AIC3204 chip out
                                             of reset */

/* Initialize I2C */
USBSTK5505_I2C_init();

/* Initialized AIC3204 */
Init_AIC3204(sf, gDAC, gADC);

/* Initialize I2S */
USBSTK5505_I2S_init();

switch (sf)
{
   case 8000:
      len = 8;
      break;
   case 12000:
      len = 12;
      break;
   case 16000:
      len = 16;
      break;
   case 24000:
      len = 24;
      break;
   case 48000:
   default:
      len = 48;
      break;
}
/* Play tone */
for ( sec = 0; sec < playtime; sec++)
{
   for ( msec = 0; msec < 1000; msec++)
   {
      for ( sample = 0; sample < len; sample++)
      {
```

Table 1.9 (*Continued*)

```
               /* Write 16-bit left channel data */
               USBSTK5505_I2S_writeLeft( sinetable[sample]);

             /* Write 16-bit right channel data */
             USBSTK5505_I2S_writeRight(sinetable[sample]);
          }
       }
    }

    USBSTK5505_I2S_close();   // Disable I2S
    AIC3204_rset( 1, 1);      // Reset codec

   USBSTK5505_GPIO_setOutput( GPIO26, 0); // Disable AIC3204
}
```

sampling frequency to convert the digital signal to analog form and send out the stereo tone to the connected headphone or loudspeaker at the user-specified output volume. Table 1.10 lists the files used for the experiment.

The eZdsp uses the TLV320AIC3204 analog interface chip for the A/D and D/A conversions. This experiment uses the function `initAIC3204()` to set up AIC3204 registers for the sampling frequency and gain values entered by the user. The sampling frequency is calculated by

$$f_s = \frac{MCLK \times JD \times R}{P \times NDAC \times MDAC \times DOSR}, \tag{1.8}$$

where the master clock $MCLK = 12\,\text{MHz}$ and $JD = 7.168$ (J and D are two registers of the AIC3204, we set register J with integer 7 and register D with fraction number 168). If we preset the rest of the registers as $R = 1$, $NDAC = 2$, $MDAC = 7$, and $DOSR = 128$, we can change the value of P to set different sampling rates. For example, by changing P from 1, 2, 3, 4, and 6, we can configure AIC3204 to operate at different sampling frequencies at 48, 24, 16, 12, and 8 kHz, respectively. The TLV320AIC3204 data sheet [24] provides some examples for setting these parameters for different sampling rates.

Table 1.10 File listing for the experiment Exp1.4

Files	Description
`playToneTest.c`	Program file for testing eZdsp real-time tone generation
`tone.c`	C source file for tone generation
`initAIC3204.c`	C source file to initialize AIC3204
`tistdtypes.h`	Data type definition header file
`c5505.cmd`	Linker command file
`C55xx_csl.lib`	C55xx chip support library
`USBSTK_bsl.lib`	eZdsp board support library

The experiment procedures are listed as follows:

1. Create the experiment folder, `Exp1.4`, and copy the experiment software to the working directory including all the files in the folder `playTone`, and subfolders `src`, `lib`, `C55x_csl`, and `USBSTK_bsl`.
2. Start CCS and import the existing project workspace for the experiment to CCS.
3. Open the property of the `playTone` project and check **C/C++ Build Settings**. The **Include Options** should include the paths for `C55x_csl ..\C55xx_csl\inc` and `USBSTK_bsl ..\USBSTK_bsl\inc`, and the **Runtime Model Options** should be set for the 16-bit and large-memory model.
4. Connect the eZdsp to the host computer and connect a loudspeaker or headphone to the eZdsp.
5. Use the **Build All** command to rebuild the program, load the program, and run the experiment with user parameters: gain, sampling frequency, and tone playtime duration. Redo the experiment using different values of these three parameters and observe the differences.

1.5.5 Audio Loopback Using eZdsp

The previous audio tone playback experiment is written for sample-by-sample processing, which processes digital signals one sample at a time. On the other hand, data samples can be processed in groups using block-by-block processing. When a processor processes data using the sample-by-sample scheme, the processor may often be in an idle state waiting for the next available sample. That is, after processing one sample, the processor must wait for the next input sample. The idle time depends upon the sampling frequency and the time needed to process each sample. The advantage of sample-by-sample processing is its short processing delay. However, sample-by-sample processing is not very efficient in terms of data I/O overhead due to the waiting time for input samples. In contrast, block-by-block signal processing uses direct memory access (DMA) for data transfer that is performed in parallel with signal processing operations. Such a system can greatly reduce the I/O overhead to achieve the maximum processing efficiency. The trade-off between sample-by-sample processing and block-by-block processing is the minimized processing delay vs. the maximized processing efficiency. Many DSP systems use multithread operating systems so the applications are often programmed using block processing.

This experiment uses the eZdsp for real-time audio playback using block-by-block processing. The signal buffer size is `XMIT_BUFF_SIZE` and this can be adjusted to different sizes for different applications. The audio source can be a microphone or an audio player. The audio source is connected to the eZdsp's audio input STEREO IN jack (J2) using a stereo cable. The processed audio samples are played via a headphone or loudspeaker connected to the eZdsp's audio output HP OUT jack (J3). To use block-by-block processing, we use DMA to transfer input and output audio data samples. The sampling rate is set using the AIC3204. The C5505 DMA manages the data transfer between the C5505 and AIC3204.

Table 1.11 lists the main program for the experiment. It begins by setting the DMA and AIC3204, and then starts looping audio input to the output. The audio path is set for stereo with left and right audio channels. The program uses flags to identify which channel (left or right) of signal is coming from the AIC3204. Each channel uses two DMA data buffers of

Table 1.11 Real-time audio loopback program, `audioLoopTest.c`

```
#include <stdio.h>
#include "tistdtypes.h"
#include "i2s.h"
#include "dma.h"
#include "dmaBuff.h"

#define XMIT_BUFF_SIZE         256
Int16 XmitL1[XMIT_BUFF_SIZE]; /* DMA uses the same buffer names, do not
                                 rename */
Int16 XmitR1[XMIT_BUFF_SIZE];
Int16 XmitL2[XMIT_BUFF_SIZE];
Int16 XmitR2[XMIT_BUFF_SIZE];

Int16 RcvL1[XMIT_BUFF_SIZE];
Int16 RcvR1[XMIT_BUFF_SIZE];
Int16 RcvL2[XMIT_BUFF_SIZE];
Int16 RcvR2[XMIT_BUFF_SIZE];

Int16 dsp_process(Int16 *input, Int16 *output, Int16 size);

extern void AIC3204_init(Uint32, Int16, Int16);

#define IER0 *    (volatile unsigned *)0x0000

#define SF48KHz     48000
#define SF24KHz     24000
#define SF16KHz     16000
#define SF12KHz     12000
#define SF8KHz      8000

#define DAC_GAIN    3    // 3 dB range: -6 dB to 29 dB
#define ADC_GAIN    0    // 0 dB range: 0 dB to 46 dB

void main(void)
{
   Int16 status, i;

   // Clean output buffers before running the experiment
   for (i=0; i<XMIT_BUFF_SIZE; i++)
   {

       XmitL1[i]=XmitL2[i]=XmitR1[i]=XmitR2[i]=0;
       RcvL1[i]=RcvL2[i]=RcvR1[i]=RcvR2[i]=0;
   }

   setDMA_address();    // DMA address setup for each buffer
```

Table 1.11 (*Continued*)

```
    asm(" BCLR ST1_INTM");  // Disable all interrupts
    IER0 = 0x0110;          // Enable DMA interrupt

    set_i2s0_slave();       // Set I2S
    AIC3204_init(SF48KHz, DAC_GAIN, (Uint16)ADC_GAIN); // Set AIC3204
    enable_i2s0();

    enable_dma_int();       // Set up and enable DMA
    set_dma0_ch0_i2s0_Lout(XMIT_BUFF_SIZE);
    set_dma0_ch1_i2s0_Rout(XMIT_BUFF_SIZE);
    set_dma0_ch2_i2s0_Lin(XMIT_BUFF_SIZE);
    set_dma0_ch3_i2s0_Rin(XMIT_BUFF_SIZE);

    status = 1;
    while (status)          // Forever loop for the demo if status is set
    {
      if((leftChannel == 1)||(rightChannel == 1))
      {
        leftChannel = 0;
        rightChannel= 0;
        if ((CurrentRxL_DMAChannel == 2)||(CurrentRxR_DMAChannel == 2))
        {
          status = dsp_process(RcvL1, XmitL1, XMIT_BUFF_SIZE);
          status = dsp_process(RcvR1, XmitR1, XMIT_BUFF_SIZE);
        }
        else
        {
          status = dsp_process(RcvL2, XmitL2, XMIT_BUFF_SIZE);
          status = dsp_process(RcvR2, XmitR2, XMIT_BUFF_SIZE);
        }
      }
    }
}

// Simulated a DSP function
Int16 dsp_process(Int16 *input, Int16 *output, Int16 size)
{
    Int16 i;

    for(i=0; i<size; i++)
    {
      *(output + i) = *(input + i);
    }
    return 1;
}
```

Table 1.12 File listing for the experiment Exp1.5

Files	Description
`audioLoopTest.c`	Program file for testing real-time audio loopback
`vector.asm`	Assembly source file for interrupt vectors
`c5505.cmd`	Linker command file
`tistdtypes.h`	Data type definition header file
`dma.h`	C header file for DMA function and variable definition
`dmaBuff.h`	C header file for DMA buffer definition
`i2s.h`	C header file for I2S function and variable definition
`lpva200.inc`	C55xx assembly include file
`myC55xUtil.lib`	Experiment support library: DMA and I2S functions

equal length. This double-buffer method is often used for block signal processing. While the AIC3204 is filling one of the DMA data buffers, the C5505 process the data available in the other buffer. Once the process is complete, the DMA controller will switch the buffers for the next DMA transfer. The DMA channel identifier is used to manage which DMA buffer will be used. This ping-pong buffering scheme can avoid memory read and write collisions. The ping-pong buffer mechanism will introduce a certain buffering delay. The delay time equals the number of data samples in the buffer multiplied by the sampling period. If the data buffer contains 48 samples, and the sampling frequency is 48 000 Hz, the buffer introduces a time delay of 0.01 seconds.

In this experiment, we include the function `dsp_process` which simply copies the data from the input buffer to the output buffer. In subsequent experiments, we will replace this function by other DSP functions such as digital filters for real-time experiments. The assembly program `vector.asm` handles real-time interrupts for the C5505 system. The TMS320C5505 architecture and assembly language programming are introduced in Appendix C. The files used for the experiment are listed in Table 1.12.

The experiment procedures are listed as follows:

1. Copy the experiment software from the companion software package to the working directory and import the existing project.
2. Connect the eZdsp to the host computer. Connect a loudspeaker or headphone to the eZdsp HP OUT jack. Connect an audio source such as an MP3 player to the eZdsp STEREO IN jack.
3. Use CCS to build the project, load the executable program, and run the experiment.
4. Modify the experiment such that the left audio output channel will output the sum of input signals from both left and right channels, while the right audio output channel will be output the difference of input signals from the left and right channels. (Hint: modify the function `dsp_process`.)
5. Modify the audio loopback experiment such that it runs at 8000 Hz or other sampling frequencies.
6. Write a new function to generate a 1000 Hz tone. Modify the experiment such that it will loop the input audio on the left output channel and output the 1000 Hz tone on the right output channel.

Exercises

1.1. Given an analog audio signal that is bandlimited by 10 kHz:
(a) What is the minimum sampling frequency that allows a perfect reconstruction of the analog signal from its discrete-time samples?
(b) What will happen if a sampling frequency of 8 kHz is used?
(c) What will happen if the sampling frequency is 50 kHz?
(d) When the sampling rate is 50 kHz, and taking only every other sample (this is decimation by 2), what is the sampling frequency of the new signal? Is this causing aliasing?

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

1.3. Assume the dynamic range of the human ear is about 100 dB, and the highest frequency a human can hear is 20 kHz. For a high-end digital audio system designer, what size of converters and sampling rate are needed? When the design uses a 16-bit converter at 44.1 kHz sampling rate, how many bits are needed to store one minute of music?

1.4. A speech file (`timit_1.asc`) was digitized using a 16-bit ADC with one of the following sampling rates: 8, 12, 16, 24, or 32 kHz. Use MATLAB® to play it and find the correct sampling rate. This can be done by running the MATLAB® program `exercise_4.m` under the Exercises directory. This script plays the file at 8, 12, 16, 24, and 32 kHz. Press the Enter key to continue after the program is paused. What is the correct sampling rate?

1.5. Aliasing is caused by using an incorrect sampling rate that violates the sampling theorem. The MATLAB® script below generates a chirp signal, where `fl` and `fh` are the lower and upper frequencies of the chirp signal respectively, and the sampling frequency `fs` is 800 Hz. Edit and run the MATLAB® script, and listen and plot the signal. If we change `fs` to 200 Hz, what will happen and why?

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

References

1. Oppenheim, A.V. and Schafer, R.W. (1989) *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
2. Orfanidis, S.J. (1996) *Introduction to Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.
3. Proakis, J.G. and Manolakis, D.G. (1996) *Digital Signal Processing – Principles, Algorithms, and Applications*, 3rd edn, Prentice Hall, Englewood Cliffs, NJ.
4. Bateman, A. and Yates, W. (1989) *Digital Signal Processing Design*, Computer Science Press, New York.
5. Kuo, S.M. and Morgan, D.R. (1996) *Active Noise Control Systems – Algorithms and DSP Implementations*, John Wiley & Sons, New York.
6. McClellan, J.H., Schafer, R.W., and Yoder, M.A. (1998) *DSP First: A Multimedia Approach*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
7. ITU Recommendation (2012) G.729, Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP), June.
8. ITU Recommendation (2006) G.723.1, Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s, May.
9. ITU Recommendation (1988) G.722, 7kHz Audio-Coding Within 64 kbit/s, November.
10. 3GPP TS (2002) 26.190, AMR Wideband Speech Codec: Transcoding Functions, 3GPP Technical Specification, March.
11. ISO/IEC (2006) 13818-7, MPEG-2 Generic Coding of Moving Pictures and Associated Audio Information, January.
12. ISO/IEC 11172-3 (1993) Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s – Part 3: Audio, August.
13. ITU Recommendation (1988) G.711, Pulse Code Modulation (PCM) of Voice Frequencies, November.
14. Zack, S. and Dhanani, S. (2004) DSP Co-processing in FPGA: Embedding High-performance, Low-cost DSP Functions, Xilinx White Paper, WP212.
15. Kuo, S.M. and Gan, W.S. (2005) *Digital Signal Processors – Architectures, Implementations, and Applications*, Prentice Hall, Upper Saddle River, NJ.
16. Lapsley, P., Bier, J., Shoham, A., and Lee, E.A. (1997) *DSP Processor Fundamentals: Architectures and Features*, IEEE Press, Piscataway, NJ.
17. Berkeley Design Technology, Inc. (2000) The Evolution of DSP Processor, BDTi White Paper.
18. Berkeley Design Technology, Inc. (2000) Choosing a DSP Processor, White Paper.
19. Frantz, G. and Adams, L. (2004) The three Ps of value in selecting DSPs. *Embedded System Programming*, October. Available at: http://staging.embedded.com/design/configurable-systems/4006435/The-three-Ps-of-value-in-selecting-DSPs (accessed May 9, 2013).
20. Texas Instruments, Inc. (2012) TMS320C5505 Fixed-Point Digital Signal Processor Data Sheet, SPRS660E, January.
21. IBM and Microsoft (1991) Multimedia Programming Interface and Data Specification 1.0, August.
22. Microsoft (1994) New Multimedia Data Types and Data Techniques, Rev. 1.3, August.
23. Microsoft (2002) Multiple Channel Audio Data and WAVE Files, November.
24. Texas Instruments, Inc. (2008) TLV320AIC3204 Data Sheet, SLOS602A, October.