1

Introduction to Real-Time Digital Signal Processing

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

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

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

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

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

Real-Time Digital Signal Processing: Implementations and Applications S.M. Kuo, B.H. Lee, and W. Tian

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

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

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

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

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

1.1 Basic Elements of Real-Time DSP Systems

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

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

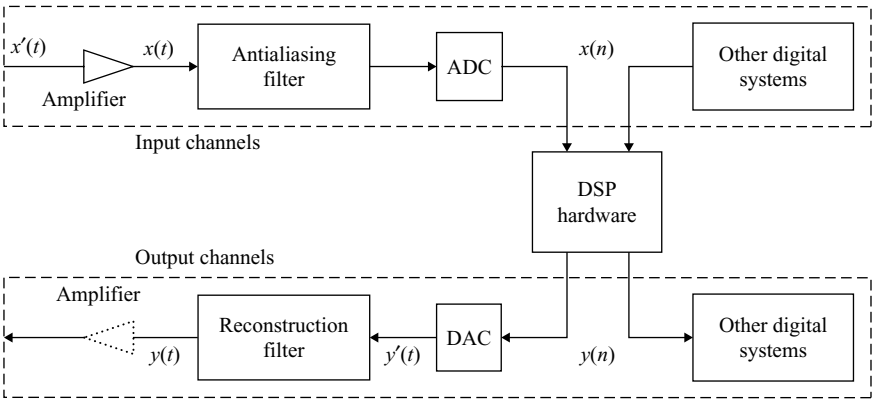


Figure 1.1 Basic functional block diagram of a real-time DSP system

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

1.2 Analog Interface

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

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

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

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

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

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

1.2.1 Sampling

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

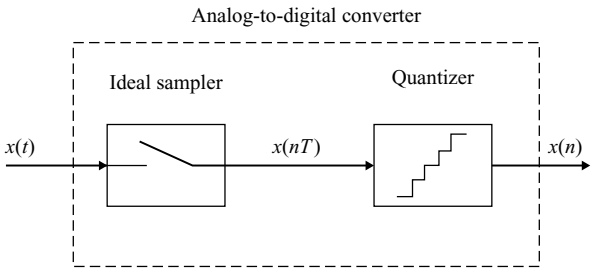


Figure 1.2 Block diagram of an ADC

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

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

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

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

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

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

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

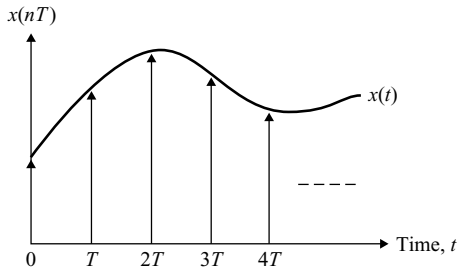


Figure 1.3 Example of analog signal $x(t)$ and discrete-time signal $x(nT)$

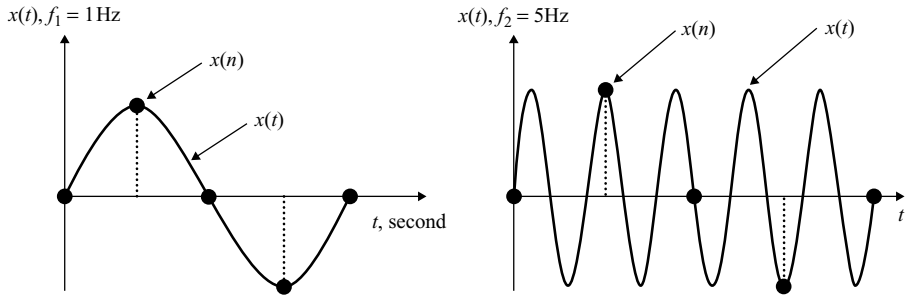
In order to represent an analog signal $x(t)$ by a discrete-time signal $x(nT)$ accurately, the sampling frequency f_s must be at least twice the maximum frequency component (f_M) in the analog signal $x(t)$. That is,

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

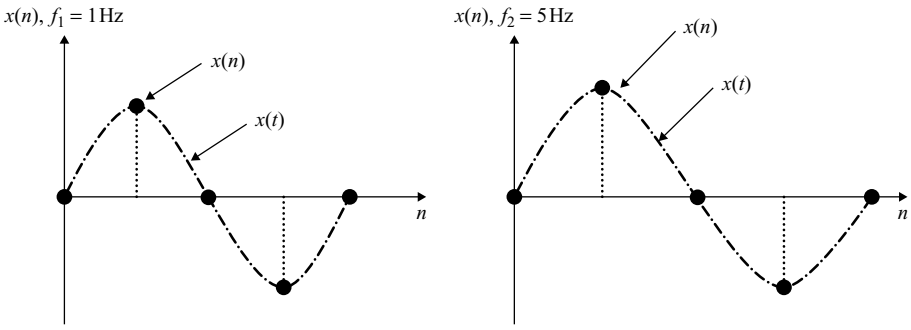
where f_M is also called the bandwidth of the signal $x(t)$. This is Shannon's sampling theorem, which states that when the sampling frequency is greater than twice of the highest frequency component contained in the analog signal, the original signal $x(t)$ can be perfectly reconstructed from the corresponding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

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

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



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



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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.2 Quantization and Encoding

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

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

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

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

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

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

1. An 8-bit ADC with 256 (2^8) levels can only provide 19.5 mV resolution and 48 dB SQNR.

2. A 12-bit ADC has 4096 (2^{12}) levels of 1.22 mV resolution, and provides 72 dB SQNR.

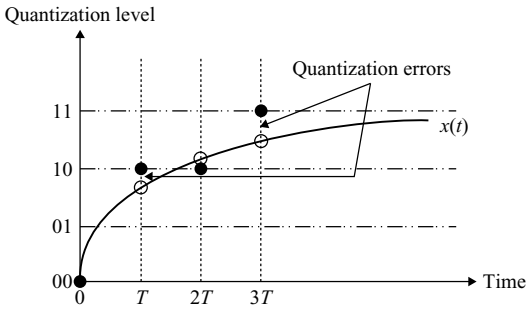


Figure 1.5 Digital samples using a 2-bit quantizer

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

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

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

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

1.2.3 Smoothing Filters

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

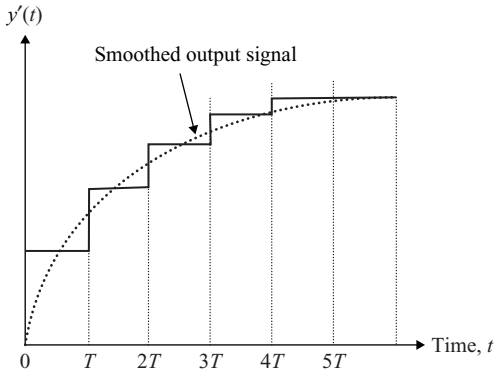


Figure 1.6 Staircase waveform generated by a DAC

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

1.2.4 Data Converters

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

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

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

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

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

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

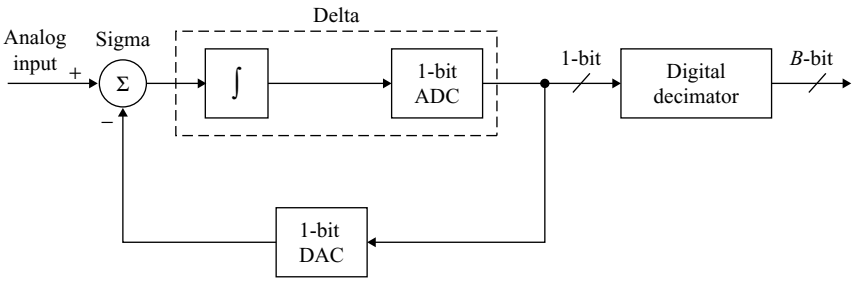


Figure 1.7 A conceptual sigma–delta ADC block diagram

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

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

1.3 DSP Hardware

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

1.3.1 DSP Hardware Options

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

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

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

Table 1.1 Summary of DSP hardware implementations

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

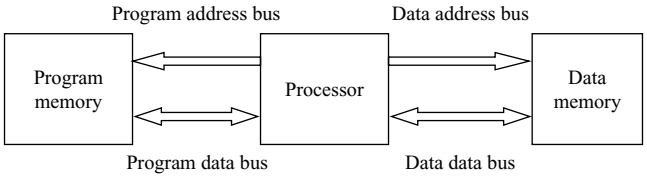
ASIC devices are usually designed for specific tasks that require a lot of computations such as digital subscriber loop (DSL) modems, or high-volume products that use mature algorithms such as fast Fourier transform and Reed–Solomon codes. These devices are able to perform their limited functions much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

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

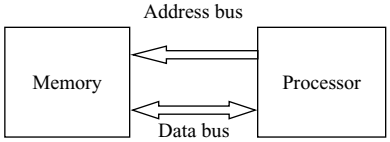
Example 1.5: There are four major FPGA families that are targeted for DSP systems: Cyclone and Stratix from Altera, and Virtex and Spartan from Xilinx. The Xilinx Spartan-3 FPGA family (introduced in 2003) uses 90-nm manufacturing technique to achieve low silicon die costs. To support DSP functions in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

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

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



(a) Harvard architecture



(b) von Newmann architecture

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

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

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

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

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

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

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

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

1.3.2 DSP Processors

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

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

Table 1.2 Current commercially available DSP processors

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

Source: Adapted from [11]

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

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

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

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

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

- Special addressing modes – DSP processors often incorporate dedicated data-address generation units for generating data addresses in parallel with the execution of instruction. These units usually support circular addressing and bit-reversed addressing for some specific algorithms.

- Special program control – Most DSP processors provide zero-overhead looping, which allows the programmer to implement a loop without extra clock cycles for updating and testing loop counters, or branching back to the top of loop.

- Optimize instruction set – DSP processors provide special instructions that support the computational intensive DSP algorithms. For example, the TMS320C5000 processors support compare-select instructions for fast Viterbi decoding, which will be discussed in Chapter 14.

- Effective peripheral interface – DSP processors usually incorporate high-performance serial and parallel input/output (I/O) interfaces to other devices such as ADC and DAC. They provide streamlined I/O handling mechanisms such as buffered serial ports, direct memory access (DMA) controllers, and low-overhead interrupt to transfer data with little or no intervention from the processor's computational units.

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

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

1.3.3 Fixed- and Floating-Point Processors

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

Floating-point arithmetic greatly expands the dynamic range of numbers. A typical 32-bit floating-point DSP processor, such as the TMS320C67x, represents numbers with a 24-bit mantissa and an 8-bit

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

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

	Precision	Dynamic range
Unsigned integer	1	$0 \le x \le 65\,535$
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 DSP 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, signals and coefficients require a large dynamic range and high precisions, or where large memory structures are required, such as in image processing. Floating-point DSP processors also allow for the efficient use of high-level C compilers, thus reducing the cost of development and maintenance. The faster development cycle for a floating-point processor may easily outweigh the extra cost of the DSP processor itself. Therefore, floating-point processors can also be justified for applications where development costs are high and production volumes are low.

1.3.4 Real-Time Constraints

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

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

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

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

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

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

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

1.4 DSP System Design

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

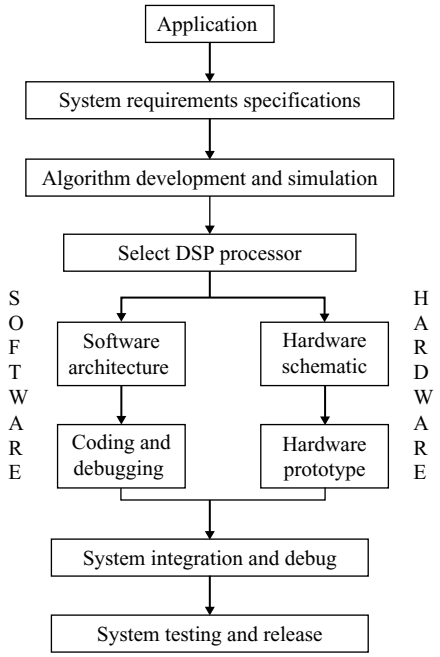


Figure 1.9 Simplified DSP system design flow

1.4.1 Algorithm Development

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

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

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

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

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

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

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

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

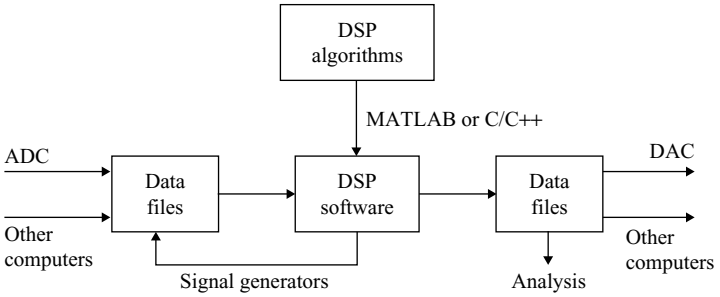


Figure 1.10 DSP software developments using a general-purpose computer

1.4.2 Selection of DSP Processors

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

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

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

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

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

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

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

Table 1.3 Some DSP applications with the relative importance rating

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

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

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

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

1.4.3 Software Development

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

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

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

The software life cycle involves the completion of a software project: the project definition, the detailed specification, coding and modular testing, integration, system testing, and maintenance. Software

maintenance is a significant part of the cost for a DSP system. Maintenance includes enhancing the software functions, fixing errors identified as the software is used, and modifying the software to work with new hardware and software. It is essential to document programs thoroughly with titles and comment statements because this greatly simplifies the task of software maintenance.

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

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

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

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

1.4.4 High-Level Software Development Tools

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

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

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

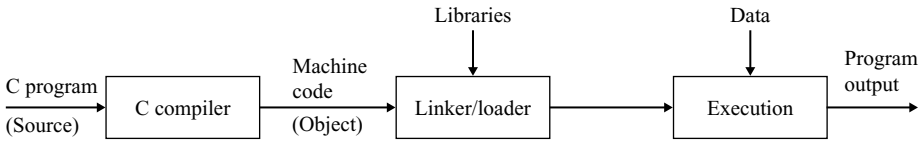


Figure 1.11 Program compilation, linking, and execution flow

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

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

1.5 Introduction to DSP Development Tools

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

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

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

1.5.1 C Compiler

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

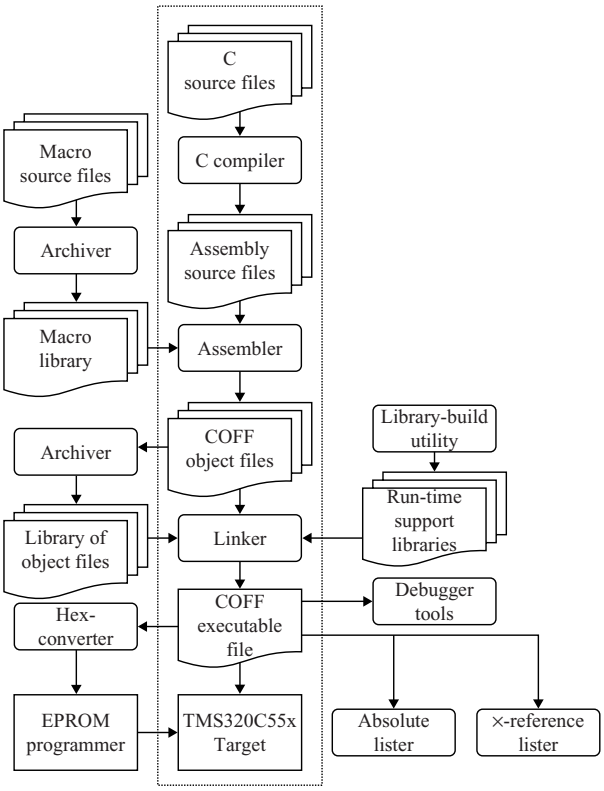


Figure 1.12 TMS320C55x software development flow and tools

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

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

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

1.5.2 Assembler

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

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

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

1.5.3 Linker

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

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

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

Table 1.4 Example of linker command file used by TMS320C55x

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

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

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

`R` – Memory space can be read.

`W` – Memory space can be written.

`X` – Memory space contains executable code.

`I` – Memory space can be initialized.

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

1.5.4 Other Development Tools

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

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

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

1.6 Experiments and Program Examples

Texas Instruments' CCS Integrated Development Environment (IDE) is a DSP development tool that allows users to create, edit, build, debug, and analyze DSP programs. For building applications, the CCS provides a project manager to handle the programming project. For debugging purposes, it provides

breakpoints, variable watch windows, memory/register/stack viewing windows, probe points to stream data to and from the target, graphical analysis, execution profile, and the capability to display mixed disassembled and C instructions. Another important feature of the CCS is its ability to create and manage large projects from a GUI environment. In this section, we will use a simple sinewave example to introduce the basic editing features, key IDE components, and the use of the C55x DSP development tools. We also demonstrate simple approaches to software development and the debug process using the TMS320C55x simulator. Finally, we will use the C5510 DSK to demonstrate an audio loop-back example in real time.

1.6.1 Experiments of Using CCS and DSK

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

Procedures of the experiment are listed as follows:

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

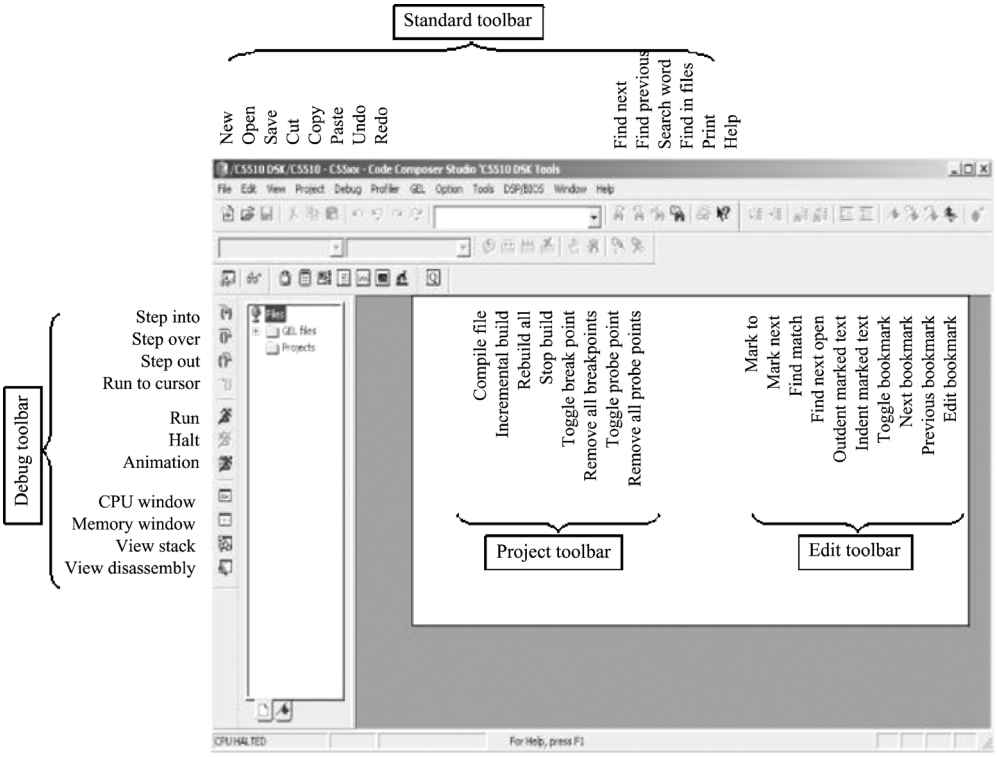


Figure 1.13 CCS IDE

Table 1.5 File listing for experiment `exp1.6.1CCSandDSK`

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

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

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

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

Table 1.6 Program example, `useCCS.c`

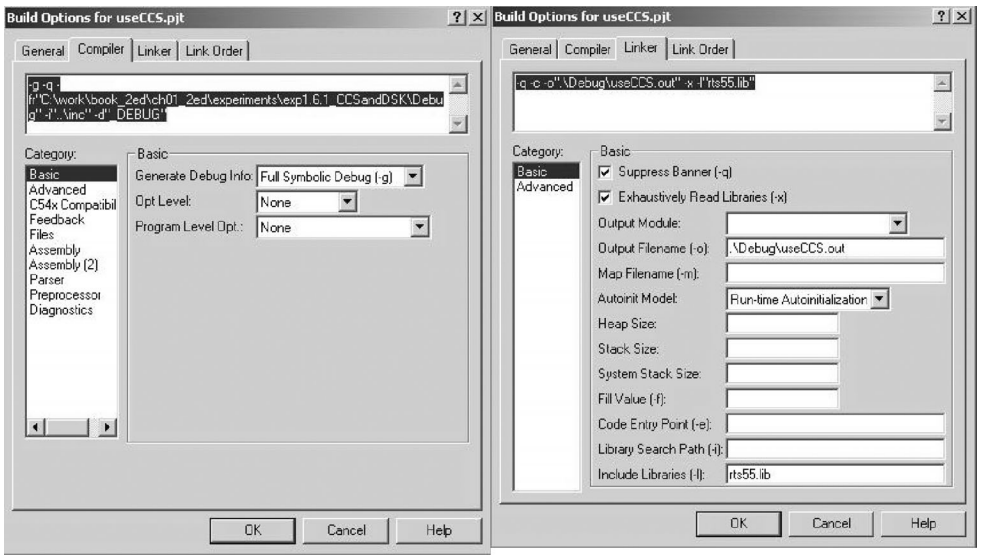
```
#include "useCCS.h"

short outBuffer[BUF_SIZE];

void main()
{
     short i, j;

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

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

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

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

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

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



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

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

1.6.2 Debugging Program Using CCS and DSK

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

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

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

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

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

Procedures of the experiment are listed as follows:

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

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

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

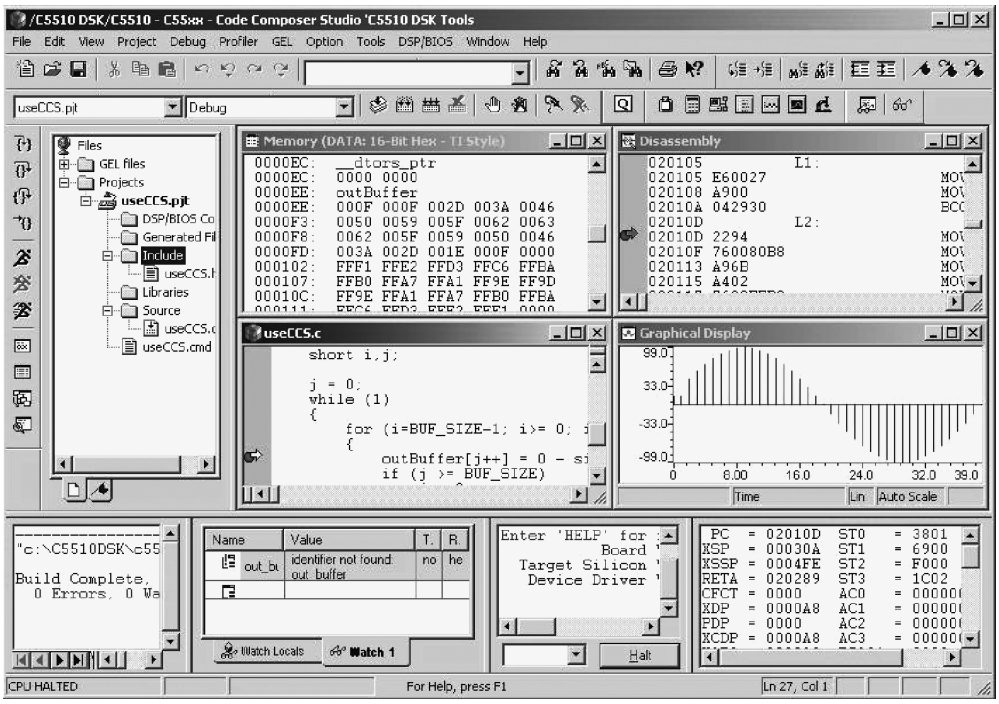


Figure 1.15 CCS screen snapshot of the example using CCS

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

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

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

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

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

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

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

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

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

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

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

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

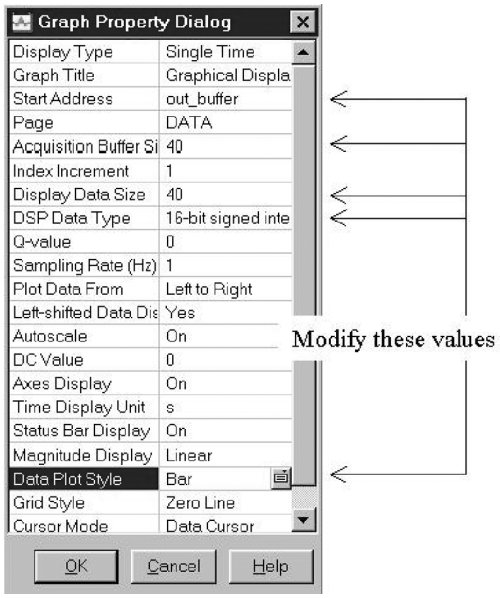


Figure 1.16 Graphics display settings

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

1.6.3 File I/O Using Probe Point

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

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

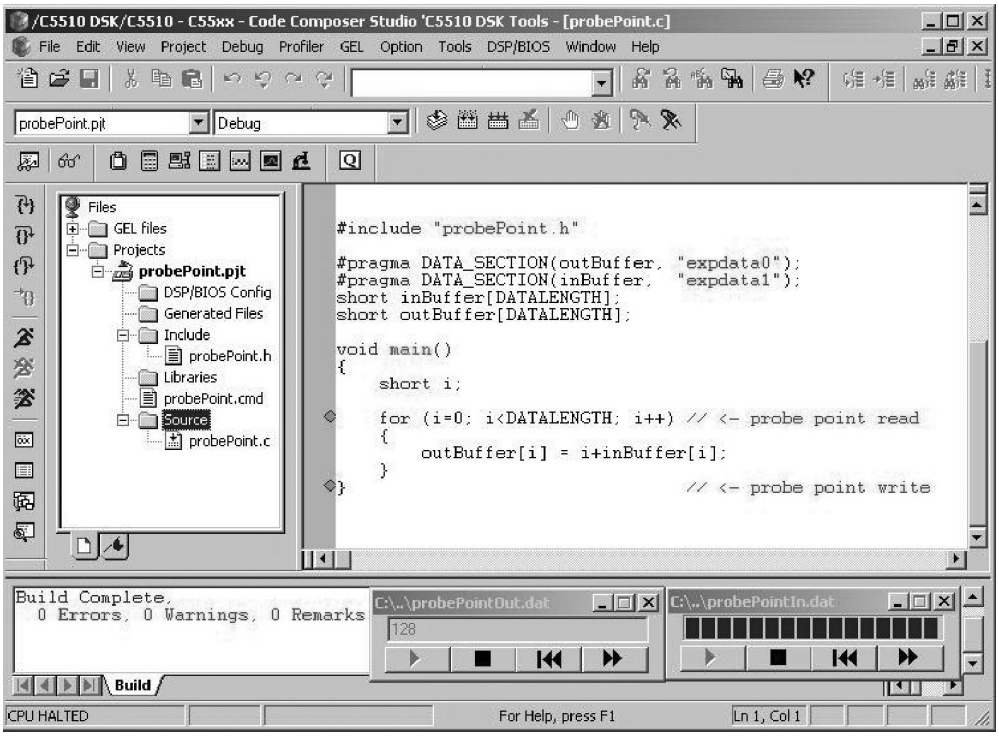


Figure 1.17 CCS screen snapshot of example of using probe point: (a) set up probe point address and length for output data buffer; (b) set up probe point address and length for input data buffer; and (c) connect probe points with files

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

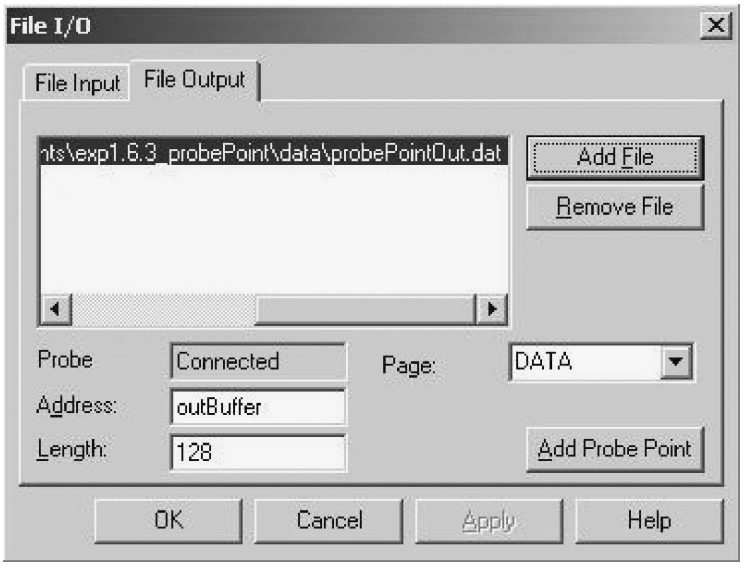
Files	Description
`probePoint.c`	C file for testing probe point experiment
`probePoint.h`	C header file
`probePoint.pjt`	DSP project file
`probePoint.cmd`	DSP linker command file

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

Procedures of the experiment are listed as follows:

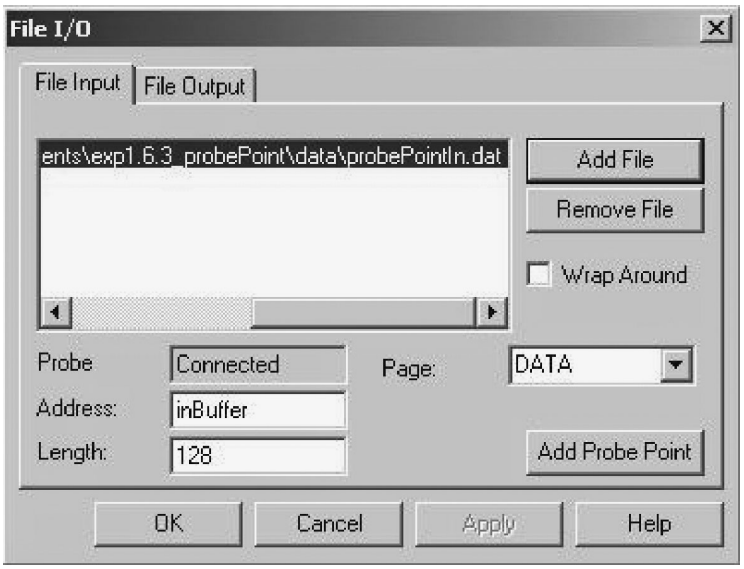
1. *Set probe point position*: To set a probe point, put the cursor on the line where the probe point will be set and click the **Toggle Probe Point** hot button. A blue dot to the left indicates that the probe point is set (see Figure 1.17). The first probe point on the line of the for loop reads data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

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

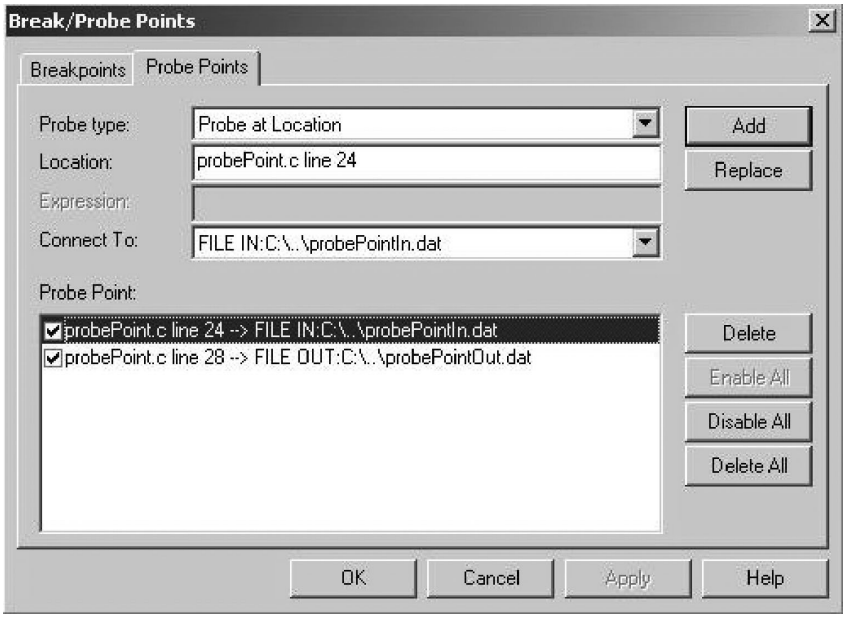


(a) Set up probe point address and length for output data buffer.

Figure 1.18 Connect probe points



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



(c) Connect probe points with files.

Figure 1.18 (*Continued*)

Table 1.9 Data input file used by CCS probe point

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

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

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

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

1.6.4 File I/O Using C File System Functions

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

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

Table 1.10 Data output file saved by CCS probe point

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

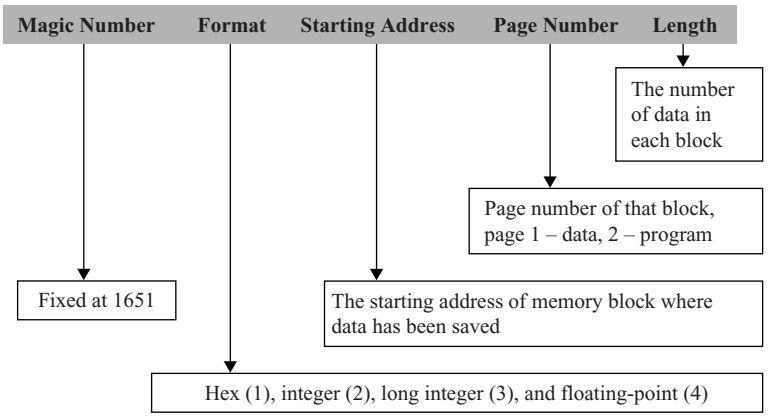


Figure 1.19 CCS file header format

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

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

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

Procedures of the experiment are listed as follows:

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

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

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

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

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

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

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

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

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

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

1.6.5 Code Efficiency Analysis Using Profiler

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

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

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

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

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

Procedures of experiment are listed as follows:

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

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

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

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

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

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

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

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

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

}

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

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

1.6.6 Real-Time Experiments Using DSK

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

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

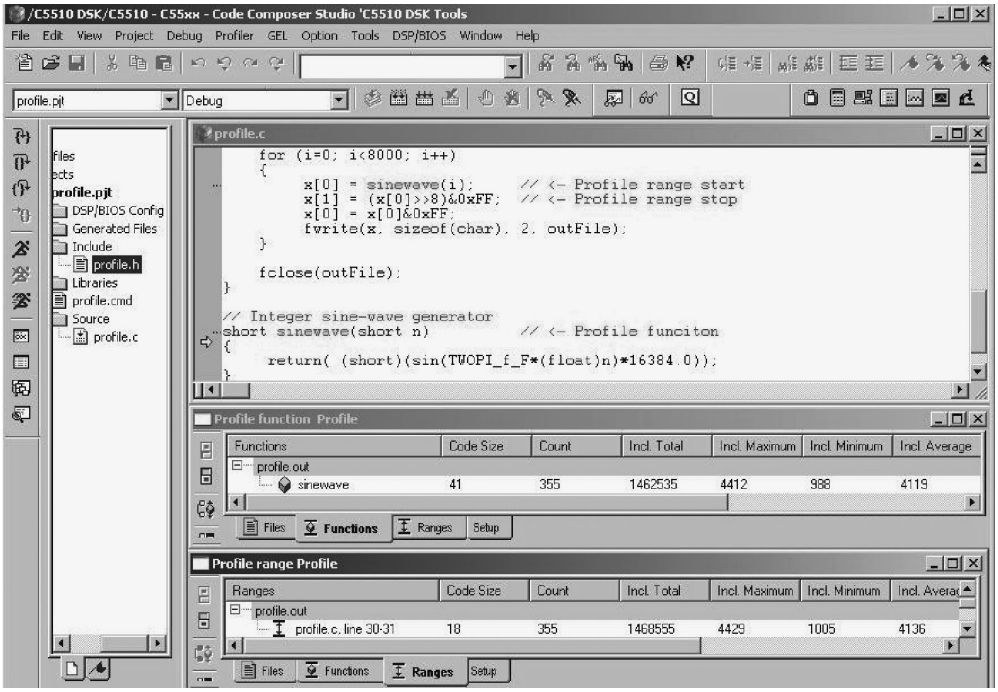


Figure 1.20 Profile window of DSP profile status

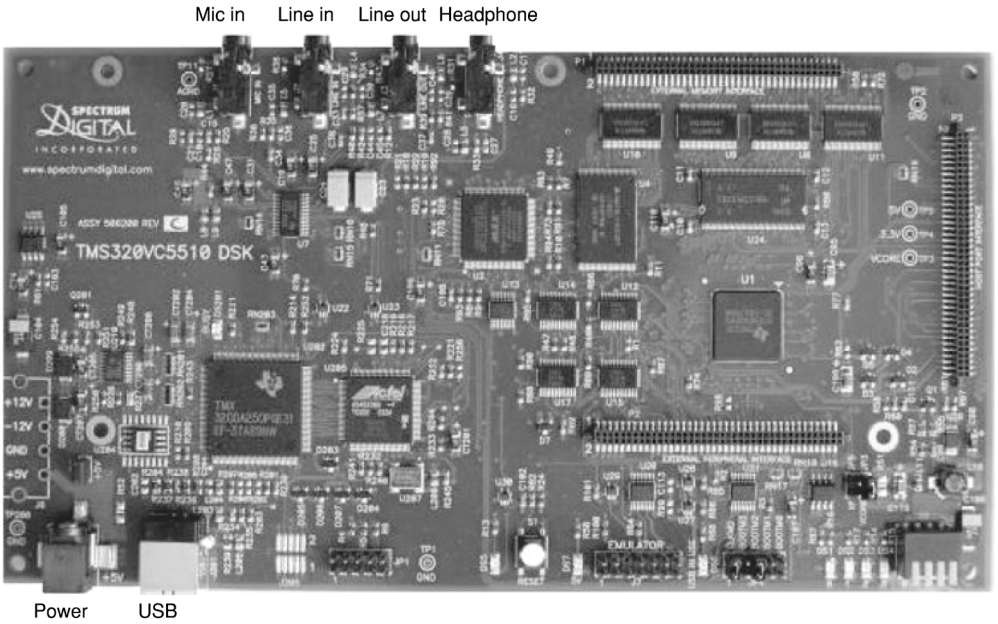


Figure 1.21 TMSVC 5510 DSK

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

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

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

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

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

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

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

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

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

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

Procedures of experiment are listed as follows:

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

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

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

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

1.6.7 Sampling Theory

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

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

Table 1.18 MATLAB code to demonstrate aliasing

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

Procedures of the experiment are listed as follows:

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

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

3. When the sampling frequency is set to 800 Hz, the code will generate a chirp signal sweeping from 0 to 200 Hz. The MATLAB uses the function `sound` to play the continuous chirp signal and plot the entire signal as shown in Figure 1.22(a).

4. Now change the sampling frequency to `fs` = 200 Hz. Because the sampling frequency `fs` does not meet the sampling theorem, the chirp signal generated will have aliasing. The result is audible and can be viewed from a MATLAB plot. The sweeping frequency folded at 100 Hz is shown in Figure 1.22(b).

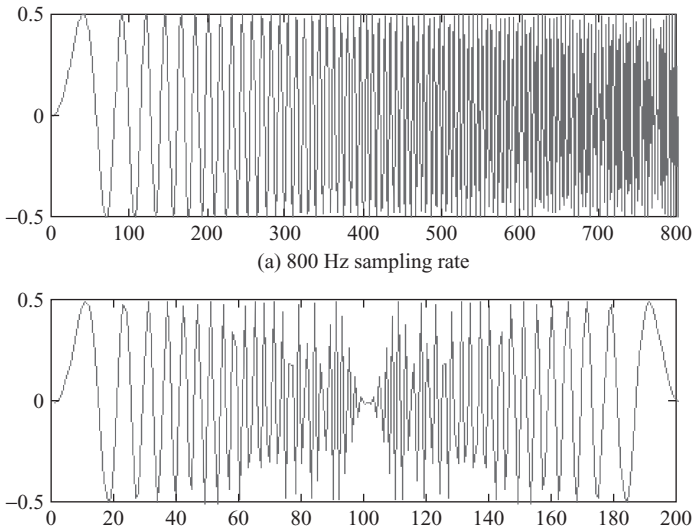


Figure 1.22 Sampling theory experiment using chirp signal: (a) 800 Hz sampling rate; (b) 200 Hz sampling rate

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

1.6.8 Quantization in ADCs

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

Procedures of the experiment are listed as follows:

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

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

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

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

An example of output is listed in Table 1.20.

Table 1.19 MATLAB code for experiment of ADC quantization

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

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

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

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

Table 1.20 Output of ADC quantization

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

References

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

Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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