

COMPUTE!'s
Beginner's Guide to

COMMODORE

64 Sound



John Heilborn

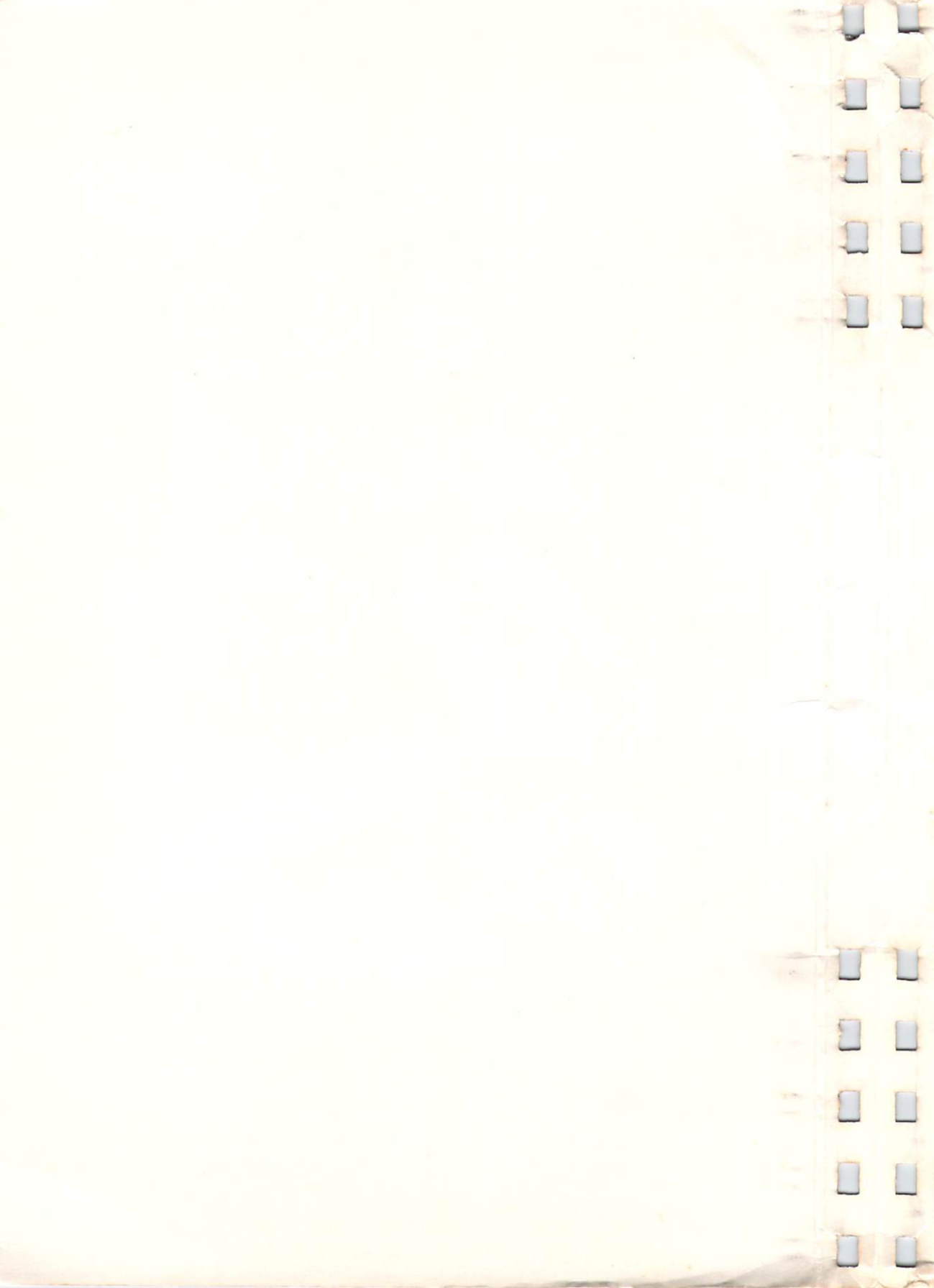
How to create music and sound on your Commodore 64. Includes dozens of type-in-and-run sound effects and two sound editors.

A **COMPUTE!** Books Publication

\$12.95

COMPUTE!'s Beginner's Guide to
Commodore 64 Sound

COMPUTE!
Books



COMPUTE!'s
Beginner's Guide to

COMMODORE

64 SOUND

John Heilborn

COMPUTE! Publications, Inc. 
One of the ABC Publishing Companies

Greensboro, North Carolina

Commodore 64 is a trademark of Commodore Electronics Limited.

Copyright 1984, COMPUTE! Publications, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

ISBN 0-942386-54-X

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Commodore 64 is a trademark of Commodore Electronics Limited.

Contents

Foreword	v
Introduction	vii
Chapter 1: SID: The Sound Interface Device	1
Keeping Registers Under Control	4
Making a Sound	9
Volume Adjustments	21
Changing the ADSR	24
Waveforms	27
Getting Fancy	33
Chapter 2: Music and the Sound Editor	37
Playing Notes	40
Timing and Rhythm	41
Pausing Techniques	43
Tools to Put the Music Together	46
A Simple Sound Editor	48
Using the Editor	59
Harmony and Disharmony	63
A Multivoice Chord Editor	69
Chapter 3: Sound Effects	83
Hard Sounds	85
Hard Sounds Using Pulse Waves	89
Hard Sounds Using Sawtooth Waves	97
Hard Sounds Using Triangle Waves	101
Hard Sounds That Use the Noise Waveform	102
Mixing Waveforms	105
Soft Sounds	106
Soft Pulse Sounds	110
Soft Sawtooth Sounds	112
Soft Triangle Sounds	114
Soft Noise	115
Tapered Sounds	117
Tapered Pulse Waveforms	119
Tapered Triangle Waves	122
Tapered Sawtooth Sounds	123
Tapered Noise	124
More Advanced Sound Techniques	125

Chapter 4: Advanced Functions	127
The Initialization Process	129
The Envelope Generator	134
Using the Sound Envelope	140
Test Bit	146
Additive Synthesis	147
Subtractive Synthesis	152
Using Your Filters	155
Resonance	157
Putting It All Together	158
Chapter 5: Putting It All Together	159
Combinations and DATA	162
Mixing Sound and Graphics	162
Sounds Produced as Needed	162
Background Sounds	173
Using Edited Music Files	181
How to Append DATA	188
Putting the DATA to Work	188
POKEing Music into Memory	191
Sound Game	195
Using What You've Learned	202
Appendices	203
A: A Beginner's Guide to Typing In Programs	205
B: How to Type In Programs	207
C: The Automatic Proofreader	209
D: Commodore 64 Sound Memory Map	213

Foreword

Hidden inside your Commodore 64 are thousands of sounds and musical compositions. Ranging from arcade-quality sound effects to full-scale synthesized orchestrations, they can make your computer “sing” in ways you never thought possible. But unless you know how to unlock those sounds, they’ll stay hidden, deep within the 64’s SID chip.

Until computers became popular, you had to be a musician to create music, or a sound effects expert to produce natural and artificial sounds. But with a Commodore 64, you have a versatile machine that can imitate the delicate sound of a violin, or the crashing roar of an ocean. And it’s often just a matter of a few POKEs and PEEKs.

Finding out what to POKE and what to PEEK can be difficult, especially if you’re just beginning to use sound on the 64. You have to know how to use the 64’s amazing SID chip, how to control its functions, and how to put it all together.

COMPUTE!’s Beginner’s Guide to Commodore 64 Sound shows you how to use this computer’s powerful sound features to create just the sound you want. More importantly, the descriptions and explanations are clear and easy to follow. John Heilborn, noted Commodore author, shows you how to create simple sounds, how to utilize the SID chip’s advanced features, and even how to combine sound with graphics to enhance your programs and games. It doesn’t matter whether you’re an experienced programmer or just starting. This book will show you techniques to create your own sounds and music.

- Use the volume control to create interesting effects.
- Manipulate the *sound envelope* to produce sounds—from whistles to sirens to footsteps in the snow.
- Compose three-part harmony.
- Easily add sound to other programs.
- Experiment with ring modulation, synchronization, and resonance.

To make it all easier, you can type in and run two complete sound editors:

- A One-Voice Sound Editor
- A Chord Editor

You can even create separate music files and, following a simple method, append them to your other programs and games. Without having to retype anything.

With the help of *COMPUTE!'s Beginner's Guide to Commodore 64 Sound*, you'll quickly be creating your own sounds and music.

Introduction

A Definition of Terms

One of the most frustrating things new computer users face is the incredible number of new words that surround computers. While the new terminology provides the experienced computer user with a quick and precise way of communicating complex ideas, it's often intimidating to novices. The unfamiliar words and their definitions can make it difficult for the beginning programmer to understand explanations. Without that understanding, it's hard to make that leap from beginner to experienced user.

It's almost a classic Catch-22 situation: You cannot comprehend the meaning of the terms without first understanding the technology, but you cannot learn the technology without a thorough understanding of the terms. To make matters worse, this book uses terms that are peculiar to two totally different technologies: computers *and* sound.

Most reference books have glossaries that define the terms used throughout the text. Usually, the glossary is at the end of the book and has a short description of each term. For our purposes, that is simply not enough. First, the glossary would be found at the wrong end of the book (you will be needing these terms before you read the book), and second, a glossary would not be sufficiently detailed.

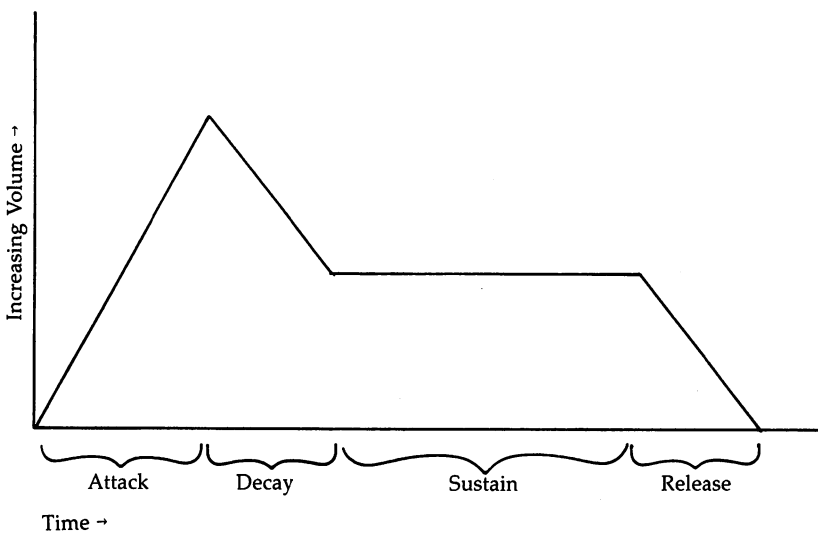
In reality, although this introduction is called "A Definition of Terms," it could more accurately be called "A Definition of Concepts," since every word is much more than just a word: It is the embodiment of an idea. The descriptions of terms presented here will be covered again in the text of the book. It can only be hoped that by describing the terms and concepts *first* as individual ideas and then *again* in the context of the book, they will be more easily understood. After all, an understanding of sound on the Commodore 64 is what you're looking for.

The Terminology of Sound

ADSR

The term *ADSR* is actually a combination of the first letters of the words *Attack*, *Decay*, *Sustain*, and *Release*. They are the four parts of each sound the Commodore 64 makes.

Figure 1. ADSR Envelope

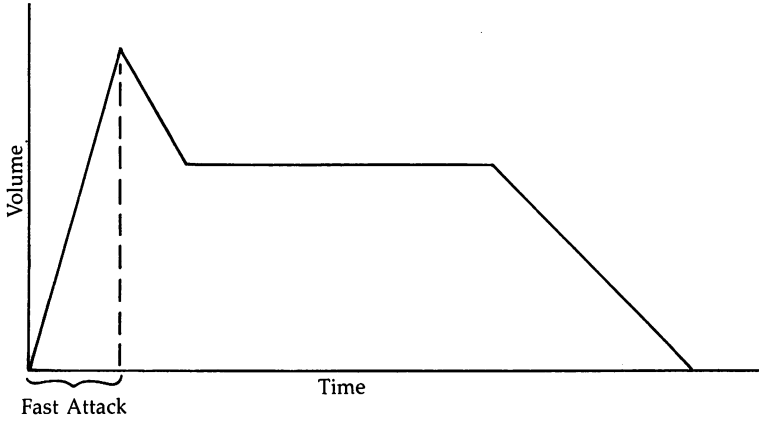


Attack

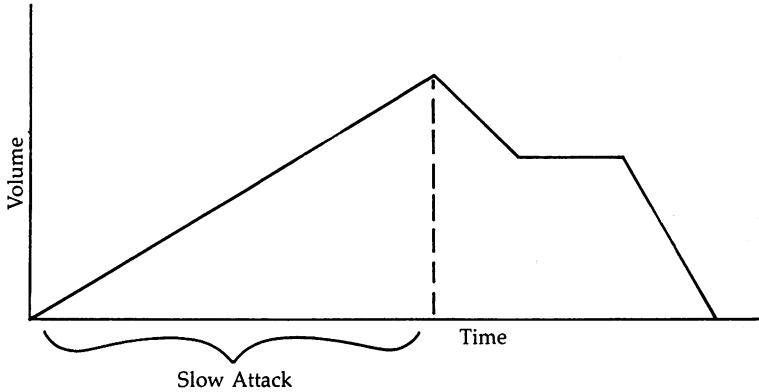
Attack is the first of the four phases of a sound. It is a description of the way that a sound begins. For example, a sound that has a fast attack is one that begins abruptly. Figure 2a shows a graph of a fast attack. A sound with a slow attack will begin quietly and gradually increase in volume. Take a look at Figure 2b to see a diagram of a slow attack.

Figure 2. Attack Rates

a. Fast Attack

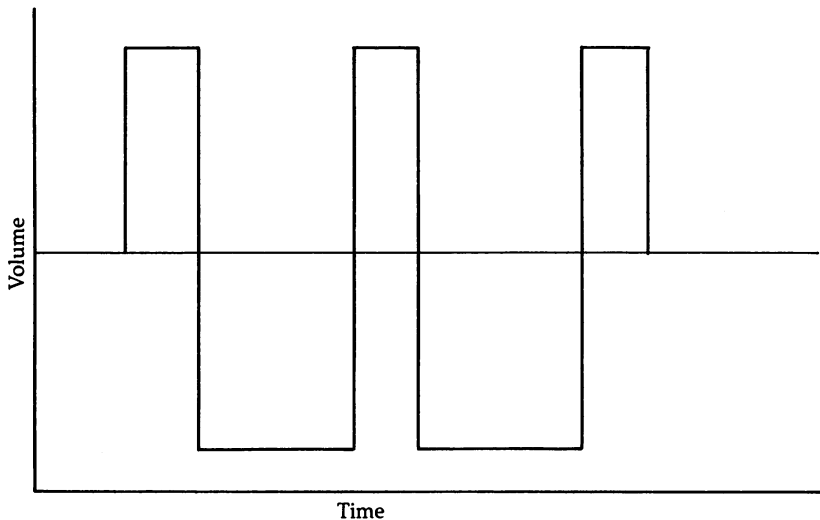


b. Slow Attack



Asymmetrical

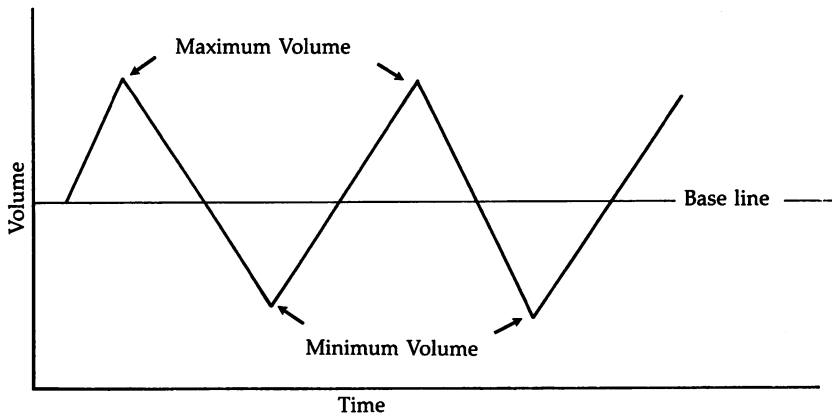
The term *asymmetrical* refers to anything which has been divided into unequal parts. In the case of the 64's sound functions, we'll use the term to describe pulse waves that are unevenly divided. This is done by programming the selected voice with a pulse width that is either more than or less than 50 percent of the total wave.

Figure 3. Asymmetrical Pulse Wave

Base Line

Sound waves are created when something vibrates. As the vibrating object pushes out or up, it compresses the air around it. When it pulls in or down, it creates a very small vacuum. This process is called *compression* and *rarefaction*. You could think of it as similar to the ripples made when you throw a pebble into a pool of water. A number of these ripples, or compressions and rarefactions, make up a sound wave. The sound's frequency (the number of compressions and rarefactions per second) and the amplitude (the size of those compressions and rarefactions) may vary. When we represent a sound wave, the *base line* is the center line for that waveform. It's the position where there is no compression or rarefaction. Look at Figure 4 to see a sample sound wave graphed on its base line.

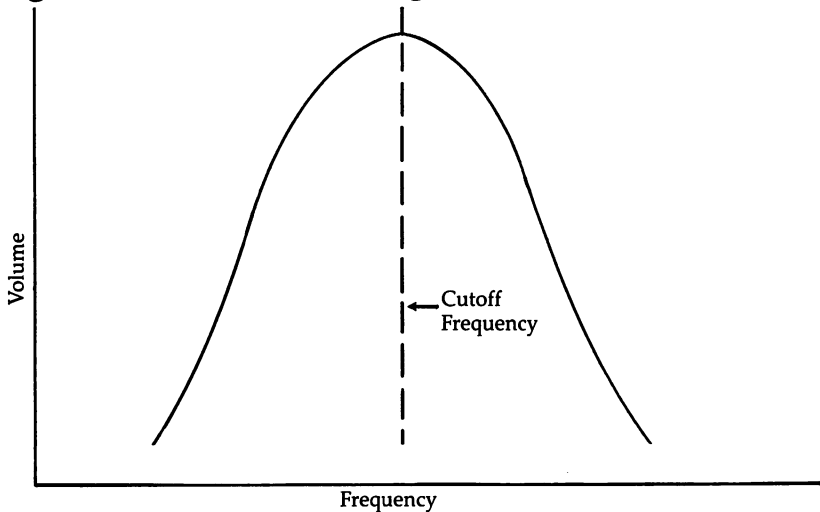
Figure 4. Base Line



Band Pass Filter

A *band pass filter* allows a selected portion of a sound's frequencies to pass through, reducing the volume of any that may be higher or lower than the selected filter cutoff frequency. Since sounds consist of more than one frequency, you'll hear only a portion of the sound's frequencies when a band pass filter is used. Figure 5 illustrates a band pass filter and its effect on a sound's frequencies.

Figure 5. Band Pass Filtering



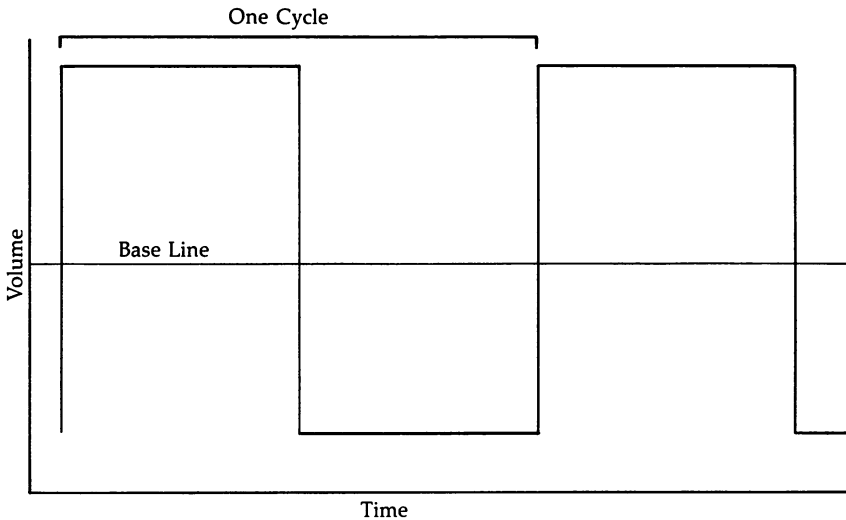
Beats

Two tones that are very close together, yet not exactly the same, will produce a background sound that rises and falls at a rate which varies as the two notes become closer together or farther apart. This background sound is called the *beat*.

Cycle

Sound waves are complete units, each of which has a beginning and an end. One *cycle* of a sound wave is a measurement taken from any point on that sound wave to the exact same position on the next sound wave. Figure 6 shows one cycle of a square wave.

Figure 6. A Sound Cycle



The frequency of a sound can be determined by counting the number of sound waves (cycles) that occur in one second. The frequency is then represented by the expression:

number of cycles counted Hz

Note: Hertz (or Hz) is a measurement of frequency. One hertz is equal to one cycle per second.

Cutoff Point

Cutoff point is simply the frequency which you have selected as the point where the filters begin to reject ranges of frequencies. The low pass, band pass, and high pass filters use this point to determine which frequencies to pass and which to reject.

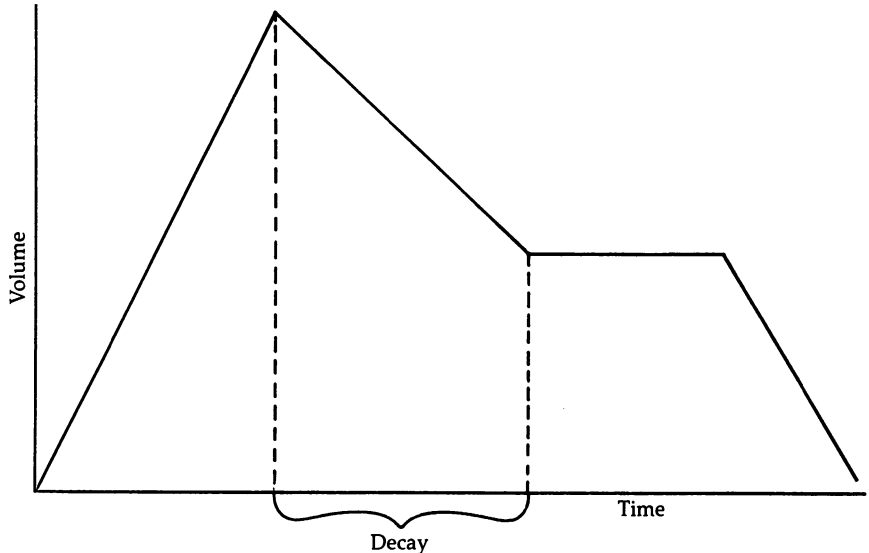
Chords

Any group of three or more notes which produce a harmonic sound can be called a *chord*. While there are mathematical formulas for determining the actual frequency values to use, the word *harmony* is essentially an aesthetic term. Thus, it's probably better to simply listen to the notes together and determine those sounds you wish to make by experimentation.

Decay

Decay is the part of a sound that follows the attack. It describes the drops in volume between the end of the attack phase and the beginning of the sustain phase of a sound. A quick decay causes the sound's volume to drop much faster to the sustain level. A slow decay decreases the volume less rapidly to the sustain setting.

Figure 7. Decay



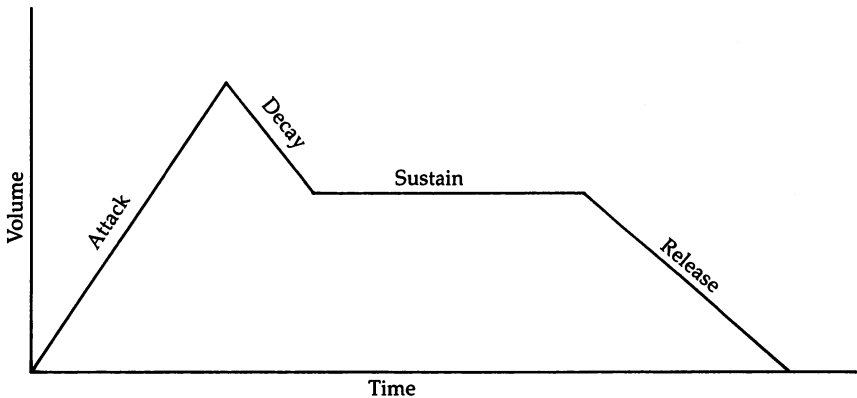
Discord

Any group of three or more notes which, when played, produce a disharmonious sound. As is the case with chords, although there are mathematical formulas for determining the actual frequency values to avoid, discord is essentially an aesthetic term. Because of their applications, disharmonic sounds are better derived by experimentation.

Envelope

In making sound on the 64, a sound *envelope* is a specific combination of the four waveform components (attack, decay, sustain, and release) as defined for any given sound. For example, a sound that begins slowly, drops quickly, maintains a long sustain and then slowly drops off to silence would have a waveform envelope like this:

Figure 8. Sound Envelope



Filtering

All filters, whether they are paper filters like the ones we use to make coffee or electronic ones like this one, have basically the same function. They selectively allow one kind of thing to pass through while blocking others. Coffee filters, for example, allow the water and any dissolved coffee to pass through them while they block the coffee grounds. Electronic filters allow selected frequencies to pass while rejecting or blocking others.

Flat

A major musical scale consists of the music notes C, D, E, F, G, A, and B. However, between these notes there are half steps which are called sharps and *flats*. The flat notes are those which are a half step below the note from which they draw their name.

Frequency

Sound is a series of vibrations conducted through the air. The *frequency* of a sound is determined by counting the number of times that the sound vibrates within a set period of time (typically one second). Frequency is then represented by the expression:

number of cycles counted Hz

Gate Bit

The *gate bit* is the sound control that is normally used to turn the sound on and off. It is found in the same memory location as the waveform control bits (for pulse waves, triangle waves, and so on).

Hard Sounds

An aesthetic term, it usually refers to sounds that begin and end abruptly, with dramatic changes in volume. Beeps, blips, and clicks can be considered *hard sounds*.

Harmony

Harmony is an aesthetic description of the way a group of frequencies sounds. While the actual combination of frequencies that would be considered harmonious varies with taste, they can be calculated using mathematical formulas. However, since the actual sounds produced are always subject to opinion, it is best, at least in most of the applications in this book, to simply experiment with different sounds.

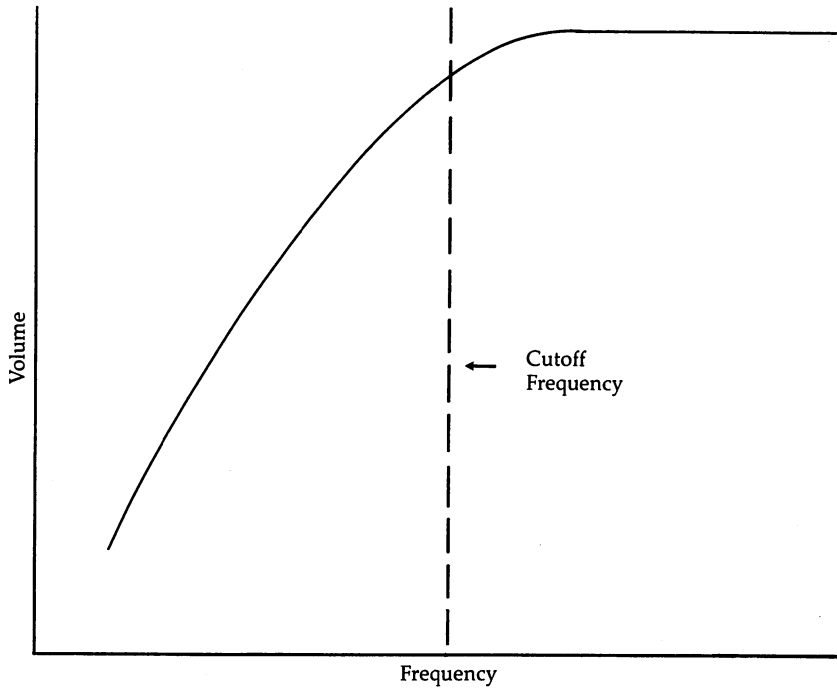
Hertz

Hertz represents cycles per second and refers to the rate at which a vibrating object or function oscillates.

High Pass Filter

The *high pass filter* in the Commodore 64 allows all frequencies above the selected cutoff point to pass while reducing the volume of all those frequencies below the selected cutoff. Figure 9 shows how this works.

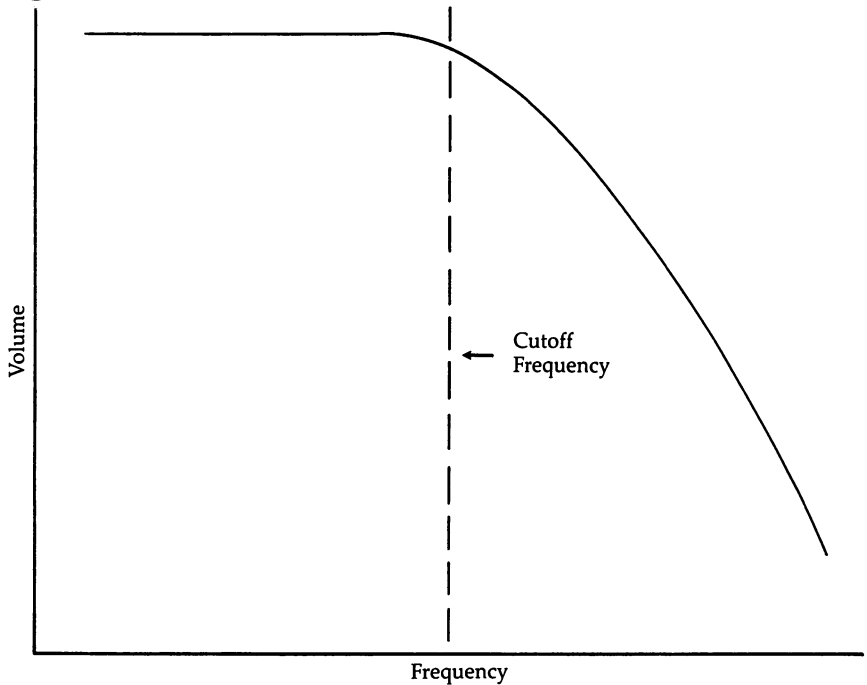
Figure 9. High Pass Filter



Low Pass Filter

The *low pass filter* in the 64 allows all frequencies below the selected cutoff frequency to pass, but reduces the volume of all frequencies above this cutoff. This function is the opposite of the high pass filter.

Figure 10. Low Pass Filter



Lower Pitch Value

The pitch produced by each of the three voices can be any of 65,536 different frequencies. To produce this number of tones, the 64 uses two separate memory locations, an upper pitch register and a lower pitch register. The value in the lower register is called the *lower pitch value*.

Noise

Noise is one of the four different waveforms that can be produced using the Commodore 64 computer. Depending on the selected frequency of the noise, it can sound like anything from a low rumble to a high-pitched hiss.

Octave

An *octave* is the distance between two pitches which have a 2/1 ratio with each other. Our musical scale is built around this relationship. A major scale consists of the pitches from

one octave to the next. Table 1-3 in Chapter 1 shows the octaves and pitches within each octave available on the 64.

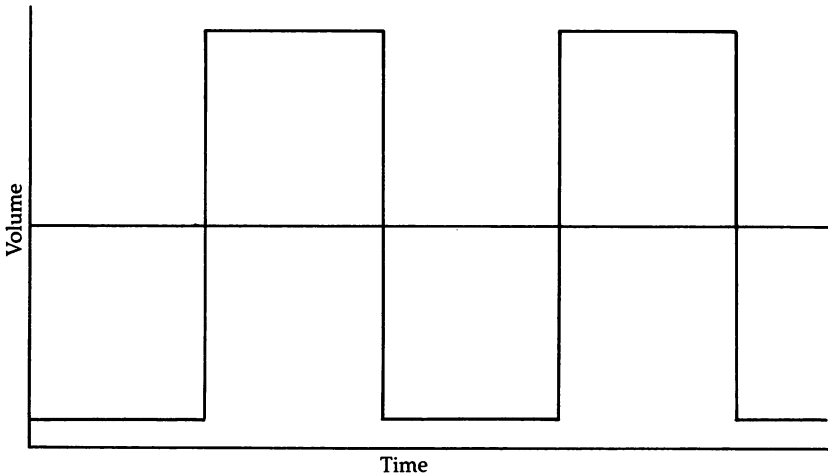
Peak

Peak refers to that point or level which marks the highest part of a waveform.

Pulse Wave

A *pulse wave* is one of the four basic waveforms the computer can produce. Pulse waves rise to a maximum displacement quickly, maintain that level for a set period, and then fall to minimum displacement. This gives pulse waves a very clear, full sound because they are always at one extreme or the other. The following figure shows a pulse wave.

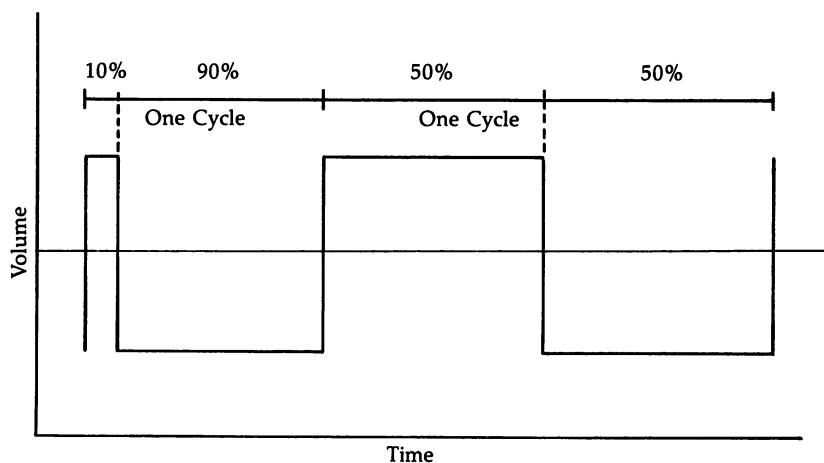
Figure 11. Pulse Wave



Pulse Width

In using the 64, *pulse width* refers to the relative proportion of time that the high and low parts of a pulse wave are at their farthest distance from a zero point. The pulse width refers to the proportion of the wave which is high. By using the pulse width control register, you can program the pulse waves with widths from 0 percent to 100 percent in 4096 steps. The left side of Figure 12 shows a pulse width of 10 percent, while the right side details a pulse width of 50 percent.

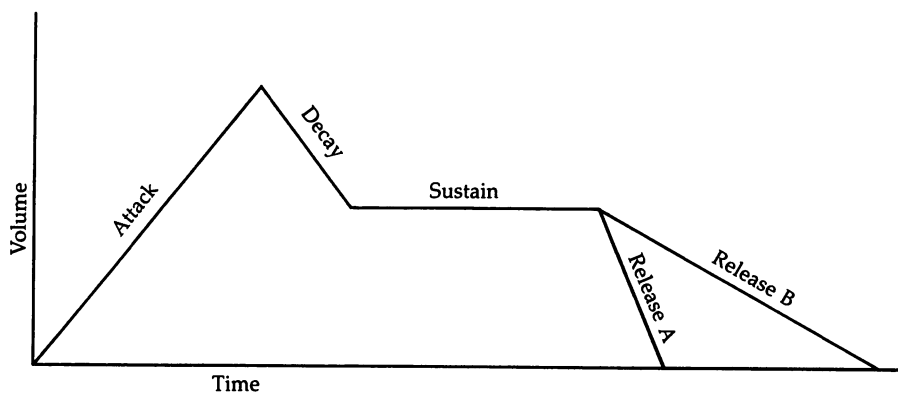
Figure 12. Pulse Width



Release

Release is the fourth and last part of a sound. It describes the way the sound finally shuts off. It behaves very much like decay. A fast release could be illustrated by an almost vertical line immediately after the sustain portion of the sound envelope (illustrated by release A in Figure 13), while a slower release could be shown by a more gradually falling line (shown as release B in Figure 13).

Figure 13. Release



Resonance

Resonance is a special effect that can be used much as a filter, modifying the sounds that are produced as they are played. Resonance emphasizes the cutoff point of a filtered sound. This function, along with synchronization and ring modulation, is used to modify the shape of the basic waves produced by the Commodore 64.

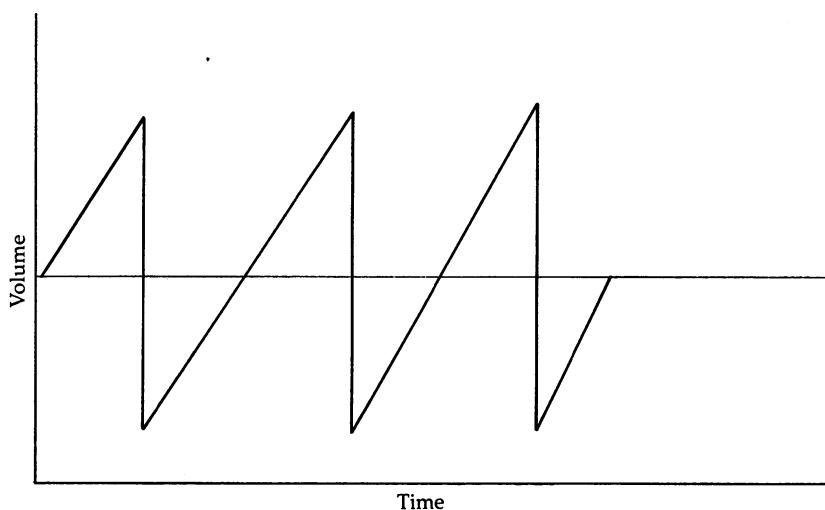
Ring Modulation

Ring modulation is a special effect that can be used much as a filter, modifying the sounds that are produced as they are played. Synchronization, resonance, and this function are all used to modify the shape of the basic waves produced by the 64.

Sawtooth Wave

The *sawtooth wave* is one of the four basic waveforms the 64 can produce. Sawtooth waves rise to maximum volume slowly, and then fall again to silence quickly. This quick drop in volume gives them a very sharp sound. Illustrated, a sawtooth waveform would look something like this:

Figure 14. Sawtooth Wave



Sharp

A major musical scale consists of the musical notes: C—D—E—F—G—A—B. However, between these notes there are half steps, which are called sharps and flats. The *sharps* are those a half step above the note from which they draw their name.

SID

The term *SID* stands for Sound Interface Device. This is the name given to the sound synthesizer chip in the Commodore 64. It is the primary circuit used to produce the three voices (and all their special effects) on the 64.

Soft Sounds

Another aesthetic term that refers to a subjective impression of a sound. Soft sounds are generally thought of as those which both rise and fall slowly in volume.

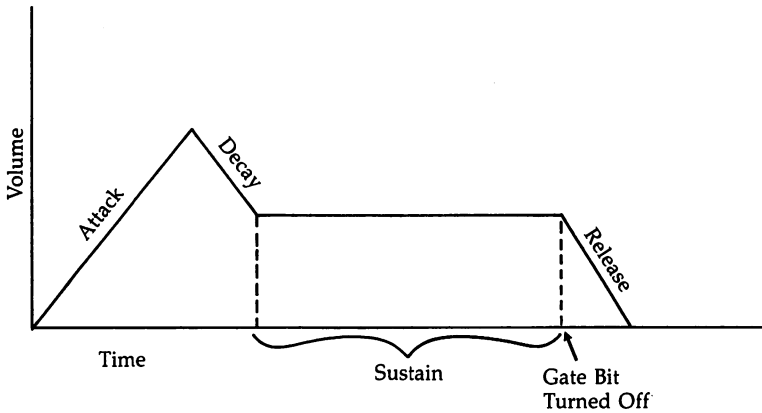
Sound Control Register

The 64 uses 25 memory locations (addresses 54272 through 54296) to control most of the sound functions for its three voices. The individual control functions are each operated through *sound control registers*, which are found in these memory locations. The registers can be manipulated by POKEing appropriate values into them. These registers are sometimes called *write-only registers*, since you can only POKE values into them, not PEEK to see what value is presently there.

Sustain

After a sound has completed its decay, the 64 begins the third cycle, *sustain*. This portion of the sound maintains a specified volume level until the sound is gated off (using the gate bit in that voice's control register). Figure 15 illustrates a long sustain. Sustain is not a function of time, as are the other elements of a sound's ADSR. It is a volume level. The length of the sustain portion of a sound envelope is determined by other methods, usually a delay loop in a sound routine or program. In a graphic representation such as Figure 15, however, shortening the horizontal line between the end of decay and the beginning of the release would illustrate a decrease in the sustain time.

Figure 15. Sustain



Square Wave

The term *square wave* refers to a pulse waveform that has been evenly divided between a high portion and a low portion. This is accomplished by programming the selected voice with a pulse width that is exactly 50 percent of the total wave. This produces the fullest and loudest sound. Sometimes this is called a *symmetrical wave*. Refer to the right-hand side of Figure 12 for an illustration of a square wave.

Synchronization (Sync)

Synchronization is one of the special functions used to modify the shape of one of the basic waveforms (pulse waves, triangle waves, sawtooth waves, and noise). It's produced by the logical ANDing of two voices.

Tapered Sounds

Tapered, a subjective term, is used to describe sounds that rise slowly to maximum volume, or that fall gradually from maximum volume to minimum volume.

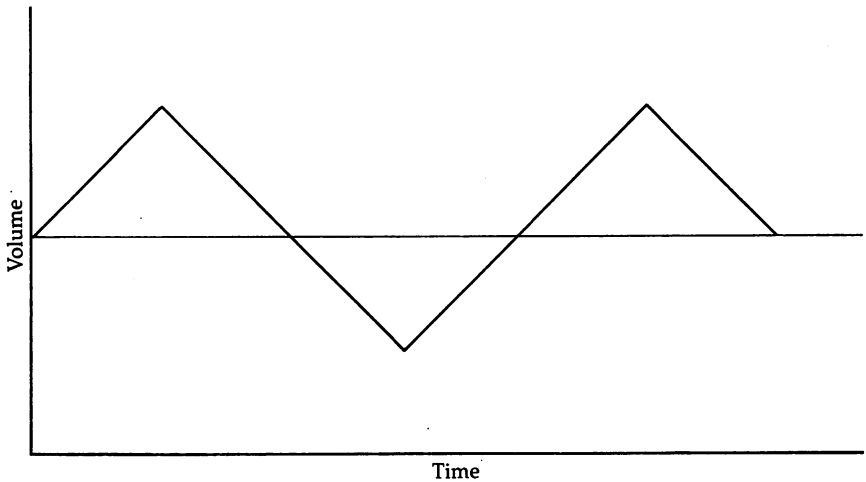
Test Bit

The *test bit* is one of the control functions in the sound registers. It turns the sound off when the bit is turned on and then on again when the bit is off. Although this bit is normally used for special applications such as testing the sound circuits, it can also be used to control the sound without manipulating any of the other sound parameters.

Triangle Wave

A *triangle wave* is one of the four basic waveforms the Commodore 64 can produce. Triangle waves rise to maximum positive displacement and then fall to minimum negative displacement slowly. This gives them a very *soft* sound. The graphic illustration of a triangle waveform is quite simple, as you can see from Figure 16.

Figure 16. Triangle Wave



Upper Pitch Value

The pitches created by each of the computer's three voices can be any of 65,536 different frequencies. To produce this number of pitches, the 64 uses two separate memory locations, an upper pitch register and a lower pitch register. The value in the upper register is called the *upper pitch value*.

Waveform

Waveforms are graphic representations of sound. They show how quickly (or slowly) the sound waves rise and fall, and optionally, their frequency.

You can use them to analyze the different waves so you can make better use of the different qualities of each sound type.

Getting Comfortable

You'll be seeing these terms quite often in the rest of this reference guide. Although the short definitions here will certainly help in understanding the terminology of sound on the 64, oftentimes you can best comprehend sound functions by experimenting yourself. In the next five chapters, you'll see numerous example programs showing how sound can be created and manipulated on the Commodore 64. Changing values and altering parameters in these example programs can be one of the best ways to see how sound works on this computer. It's only a few steps from seeing how a routine creates sound, to modifying it slightly, to beginning to write such programs yourself. Many people learn to program this way.

Of course, this glossary of terms is available to you as you read through the rest of the book. You can turn back to this section and locate the term you've just come across, but don't quite understand.

Becoming comfortable with these terms is important, for the following chapters use them extensively. It's a jargon, certainly, but it allows explanations to be simple, yet complete. Spend the time necessary getting to know these terms, and you'll find the rest of this reference guide much easier to follow, and thus use.

Typing in Programs from This Book

Before you go on, it's a good idea if you first read Appendices A, B, and C, all found in the back of this book. They'll explain how to type in the programs listed in this guide and are especially helpful if you're unfamiliar with COMPUTE!'s listing conventions, or if you're just starting to use your Commodore 64.

Appendix A, "A Beginner's Guide to Typing In Programs," and Appendix B, "How to Type In Programs," serve as an introduction to your 64's editing functions and our listing conventions. Appendix C, "The Automatic Proofreader," includes a program which will make mistake-free entry of our listings simple and easy. It will be easier to enter the programs in this book if you first read the explanation and type in the Automatic Proofreader found in Appendix C.

If you look at the programs in this book, you'll see a REM

statement, preceded by a colon and followed by a number, at the end of each program line. For instance, you might see `:rem 127`. *Do not type in this REM statement or the number as you enter the program.* This is the checksum number generated by the Automatic Proofreader program. You'll compare the number after the `:rem` with the number displayed on your screen. Using Automatic Proofreader, you can insure that the program is entered exactly as shown in the book. It will save you considerable time.



CHAPTER

1
SID:
The Sound Interface
Device



1

SID:

The Sound Interface Device

The Commodore 64, like other computers, operates on numbers. It accepts numbers as input, and it produces numbers as output.

Since the 64 outputs only numbers, you may wonder how it can make sounds, or even how it produces the video display (television picture) you see every time you turn it on. The answer is simple: The pictures and sound that the computer produces really *are* numbers. The computer has memory locations that store the numbers that create the picture and sound. Then it uses some special devices to convert those numbers into sound and pictures.

There are two such devices which the 64 uses to convert numbers into sound and pictures. Pictures are processed with a device called the VIC-II chip, and sounds are created using the SID (Sound Interface Device) chip.

The SID chip is capable of producing three voices at the same time and any of 65,536 different pitches within a full eight-octave range. In addition to that, the SID allows you to control the actual shape of the sounds it produces using filtering and waveform control.

It's almost like having a music synthesizer built into the computer. Much more sophisticated and versatile than other computers' sound-creating devices, the Commodore 64's SID gives you an amazing range of choices when it comes to producing sounds and music. Because of that, it's somewhat more complicated creating sounds on the 64. You have to know how to access and manipulate the various control

registers of the SID chip. It's not difficult, really; it just takes some time to learn. But you'll soon realize it's worth that time, for you can make the Commodore 64 "sing" in ways no other computer can duplicate.

Keeping the Registers Under Control

The way you program the SID to produce sounds is by putting different values (expressed as numbers) into its *sound control registers*. The sound control registers are memory locations, which store the numbers used to produce sound on the 64. Table 1-1 shows the name and function of each of the sound control registers, its location, and the bits that can be set to manipulate that register.

Table 1-1. The SID Sound Control Registers

Sound Control Register Function	Memory Address	Bit(s)
Voice 1 Pitch Value (lower value)	54272	0-7
Voice 1 Pitch Value (upper value)	54273	0-7
Voice 1 Pulse Width (lower value)	54274	0-7
Voice 1 Pulse Width (upper value)	54275	0-3
Voice 1 Waveform Output (gate bit)	54276	0
Voice 1 Sync Bit Enable	54276	1
Voice 1 Ring Modulation Enable	54276	2
Voice 1 Test Bit Enable	54276	3
Voice 1 Triangle Wave Enable	54276	4
Voice 1 Sawtooth Wave Enable	54276	5
Voice 1 Pulse Wave Enable	54276	6
Voice 1 Noise Enable	54276	7
Voice 1 Decay Value	54277	0-3
Voice 1 Attack Value	54277	4-7
Voice 1 Release Value	54278	0-3
Voice 1 Sustain Value	54278	4-7
Voice 2 Pitch Value (lower value)	54279	0-7
Voice 2 Pitch Value (upper value)	54280	0-7
Voice 2 Pulse Width (lower value)	54281	0-7
Voice 2 Pulse Width (upper value)	54282	0-3
Voice 2 Waveform Output (gate bit)	54283	0
Voice 2 Sync Bit Enable	54283	1
Voice 2 Ring Modulation Enable	54283	2
Voice 2 Test Bit Enable	54283	3
Voice 2 Triangle Wave Enable	54283	4
Voice 2 Sawtooth Wave Enable	54283	5

Sound Control Register Function	Memory Address	Bit(s)
Voice 2 Pulse Wave Enable	54283	6
Voice 2 Noise Enable	54283	7
Voice 2 Decay Value	54284	0-3
Voice 2 Attack Value	54284	4-7
Voice 2 Release Value	54285	0-3
Voice 2 Sustain Value	54285	4-7
Voice 3 Pitch Value (lower value)	54286	0-7
Voice 3 Pitch Value (upper value)	54287	0-7
Voice 3 Pulse Width (lower value)	54288	0-7
Voice 3 Pulse Width (upper value)	54289	0-3
Voice 3 Waveform Output (gate bit)	54290	0
Voice 3 Sync Bit Enable	54290	1
Voice 3 Ring Modulation Enable	54290	2
Voice 3 Test Bit Enable	54290	3
Voice 3 Triangle Wave Enable	54290	4
Voice 3 Sawtooth Wave Enable	54290	5
Voice 3 Pulse Wave Enable	54290	6
Voice 3 Noise Enable	54290	7
Voice 3 Decay Value	54291	0-3
Voice 3 Attack Value	54291	4-7
Voice 3 Release Value	54292	0-3
Voice 3 Sustain Value	54292	4-7
Cutoff Filter (lower value)	54293	0-2
Cutoff Filter (upper value)	54294	0-7
Voice 1 Filter Enable	54295	0
Voice 2 Filter Enable	54295	1
Voice 3 Filter Enable	54295	2
External Audio Filter Enable	54295	3
Resonance Filter Value	54295	4-7
Volume Control	54296	0-3
Low Pass Filter Enable	54296	4
Band Pass Filter Enable	54296	5
High Pass Filter Enable	54296	6
Disable Voice 3	54296	7
Voice 3 Numeric Output	54299	0-7
Envelope Generator	54300	0-7

If you were using voice 1, for example, and wanted to use a pulse waveform, you would set bit 6 of location 54276 (by placing a value of 64 in that address), as shown in the above table.

Bits, bytes, and nybbles. The control registers listed in Table 1-1 are used to access all of the sound functions on the 64. To operate all but the last two registers, you use the BASIC command *POKE*. This allows you to change the values in the registers. The format of a *POKE* command is:

```
POKE M,V
```

where M is a number representing a memory location between 0 and 65535, and V is the value to be stored in that location. The number to be stored must be between 0 and 255.

To get a better idea of how this works, let's store a number in memory:

```
POKE 6000,76
```

Now the location 6000 contains the value 76. If you want to read that memory location—in other words, see what value is presently there—you'll need to use the BASIC counterpart of the *POKE* command, *PEEK*.

```
PRINT PEEK (6000)
```

PEEK is used to examine the contents of a memory location. You have to use the *PRINT* command along with the *PEEK* command so the computer will display the value it found in location 6000. Without the *PRINT* command, the computer would not know what to do with the value it found. When you press *RETURN*, the computer should display the value you stored, 76.

By using this method of placing values as well as looking to see what value is currently in a memory location, you can store any number in any memory address that is available, as long as you use values between 0 and 255.

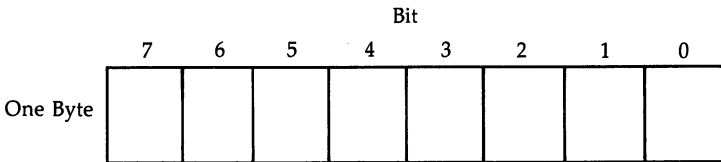
The reason you're restricted to using numbers between 0 and 255 is that the Commodore 64 is an eight-bit computer. In other words, it can accept only numbers that are eight *binary* bits long.

While it is not the purpose of this book to explain how binary numbers and the 64's memory registers work, it is important that you be able to use them. If you need a more detailed explanation of binary numbers than what follows,

you should consult a book on beginning BASIC.

Each memory location in the 64 can store one *byte*, which in turn is made up of eight bits (*binary digits*). The eight bits in each byte are numbered, starting with 0 on the far right, and ending with bit 7 on the far left. A byte, then, would look something like this:

Figure 1-1. Byte's Bits



Each bit in this sample byte has its own individual value. If the bit is *set* (the bit has a 1 stored in it), that value is added to the rest of the *on* (another term for set) bits in the byte. If that bit has a 0 stored in it instead, it's said to be *off*, or *not set*. Then the value of that bit is 0. The bits' values in a byte are given in the following table:

Figure 1-2. Bit Values

Bit	7	6	5	4	3	2	1	0
Bit Value	128	64	32	16	8	4	2	1

Bit 0's value when it's set (a 1 is stored there) is 1. Bit 1's value is 2 when it's set, bit 2's value is 4, and so on. You'll notice that each succeeding bit as you move to the left has a value exactly double that of the previous bit. As long as you can remember that, the values are easy to recall.

Earlier, we said that the numbers placed in the 64's memory locations had to be between 0 and 255. Remember, that is the range of possible values on an eight-bit computer like the Commodore 64. But how are those numbers created? With the

bit values in a byte, by setting (placing a 1 in that bit) some bits, and leaving other bits off (by placing a 0 in that bit). The total value of all set bits becomes the value stored in that memory location's byte.

For example, let's look at the binary number 0:

00000000

This represents a single byte in the 64. All eight bits contain zeros. Not one of the bits has been set or is on. To show the binary number 1, all you'd have to do is set one bit, bit 0. Bit 0, you'll recall, had a value of 1 when it was set. Binary number 1 would look like this:

00000001

Setting both bit 0 and bit 1 would give us a different value. All you have to do is add the bit values (see Figure 1-2) together for all the bits that have been set. 1 (bit 0) + 2 (bit 1) = 3 . Binary number 3 would thus look like:

00000011

Now let's look at the number 255 (as it is stored internally in the computer):

11111111

All the bits have been set by storing 1's in them. Adding together all the bit values gives you a total of 255 ($128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$). If you tried to add anything to this number, the number would overflow, since all the bits are filled. That's why you cannot save a number greater than 255 in any single memory location.

(Of course, you can store larger numbers in the computer, but they are stored as several bytes which are then combined to make larger numbers.)

Clearing the registers. Clearing the sound registers should be the first step in every sound routine you write. You do this because there is no way to determine what values are stored in any of the sound registers. They are sometimes called *write-only* registers, for although you can write to (store numbers in) them, you cannot read them. In other words, PEEKing to these registers will not give you a correct value. All you'll see displayed will be 0's. You can try this yourself by entering the following one-line program and then typing RUN.

```
10 FOR I=54272 TO 54296:PRINT PEEK(I):NEXT
```

A column of 0's will show on the screen. Even if you enter a value to one of the registers by POKEing a value into it (such as POKE 54272,32), the above one-line program will still show that only 0 values are present.

By clearing all of the registers (POKEing them with zeros), you make sure that you will not have any incorrect values in any of the sound registers. It's a good habit to get into when you're programming sound on the 64.

Making a Sound

Although the process of making sound with the SID is somewhat complex, it does not have to be difficult. After you've cleared the sound registers (see the description earlier for that process), there are several steps you should follow when you create any sounds on the 64, even the simplest sound:

Volume. The sound control register at location 54296 controls the volume, which has a range from 0 to 15. Before you set anything else, you should first set the sound volume. If you don't, you won't hear anything when the sound is produced.

Attack, Decay, Sustain, and Release (ADSR). The sixth and seventh control registers for each of the 64's three voices (for example, locations 54277 and 54278 for voice 1) determine the setting of the ADSR. Without placing values in these registers, the sound cannot be created.

Frequency. You also have to select a frequency or pitch for your sound or note. The first two registers for each voice control determine the highness or lowness of the pitch of the sound you produce. One register contains the lower pitch value, while the other contains the upper pitch value. Combined, they allow you to select any of 65,536 frequencies. Table 1-3 shows the actual numbers you need in order to produce a particular note (the lower and upper POKE values) and the frequency of that note.

Waveform and gate bit enabling. There are four possible waveforms you can choose from on the Commodore 64 by POKEing an appropriate value into the fifth register of each voice. Pulse waveforms can create a variety of sounds because other registers (called the pulse-width control registers) determine its shape. The triangle waveform produces a soft, almost subdued sound, while sawtooth waveforms create a sharper,

more twangy sound. The fourth waveform, noise, can vary from a high hiss to a low rumble. After selecting the waveform, however, you also have to turn the gate bit *on*, which is done simply by setting bit 0 in each waveform control register to 1. If you don't turn the gate bit on, you won't hear the sound, no matter which waveform you chose.

Loop to sustain the sound. If you want to hear the note for any length of time, you'll need to insert a FOR-NEXT delay loop in the sound routine. This loop keeps the sound or note playing at the sustain volume for the amount of time you want.

Gate bit disabling. Finally, to turn the sound off, you'll disable the gate bit (found in the same register as the waveform control) by setting bit 0 to 0.

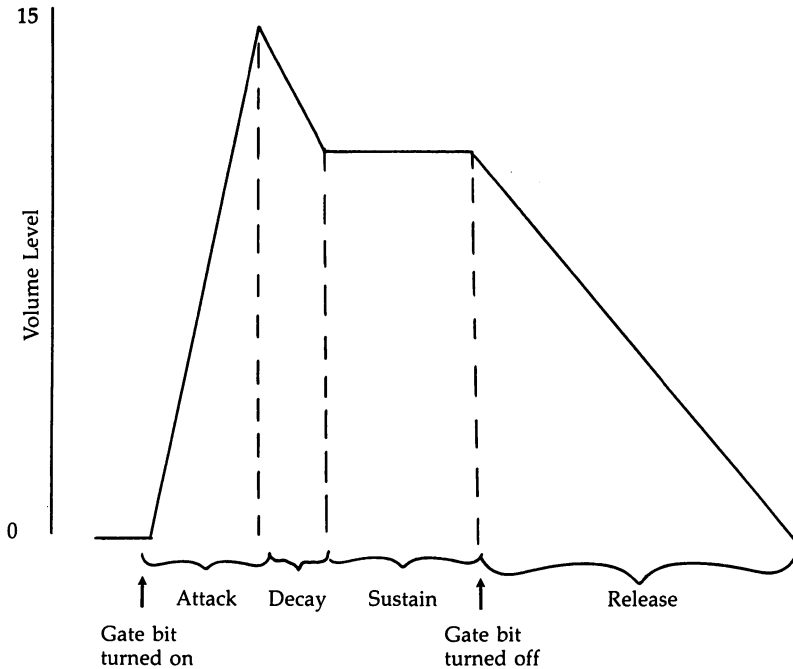
Let's take a look at each of these steps in turn, explaining what you must do with each in order to create a sound on the 64. Then we'll actually create some sounds, even modifying them by changing values in some of the sound control registers.

The volume control on the 64 is handled through the first four bits of address 54296. Values ranging from 0 to 15 can be POKEd into this location to set different levels of sound. A value of 0, however, means that the volume has been turned off. Since there's only one register to control volume, all three voices will play at the same level. One way to get around that is to set the volume note by note. You'll see how to do that in this chapter.

Attack, Decay, Sustain, and Release (ADSR) are the names given to the four sections of the sound envelope. By altering the registers that control these functions, you can significantly change the sound produced by the 64. Look at Figure 1-3 for an illustration of a sound envelope and the pattern of attack, decay, sustain, and release.

To understand the ADSR of a sound, it's probably easiest to think of what a sound, such as an explosion, is like. The sound begins quite loud, and dies down quickly; its volume rapidly increases, stays at that level for a moment or two, and then falls off dramatically. The explosion sound lasts only a few seconds. You can think of all this process of getting louder, then eventually softer, as the ADSR of the note. Each sound, whether natural or artificial, has its own sound envelope made up of an ADSR.

Figure 1-3. Attack, Decay, Sustain, and Release



- **Attack.** A function of time, the *attack* of a note determines how quickly a sound reaches its selected volume. You can set the attack to last from two milliseconds to almost eight seconds. An attack rate of 0 means the sound begins at full volume.

- **Decay.** *Decay* is also a function of time expired. After the initial attack, the volume decreases until it reaches the level specified by the sustain level. Decay indicates how long this process lasts. A fast rate drops the sound quickly from the high point of its attack to the sustain, while a slow rate draws it out.

- **Sustain.** The only portion of the ADSR that is not time-oriented, *sustain* is actually a volume setting. The value you select for this function determines the volume level reached after the decay. For instance, if the sustain is set at 12, the decay will drop the volume from level 15 to level 12. The note stays at this volume until you clear the gate bit. In other

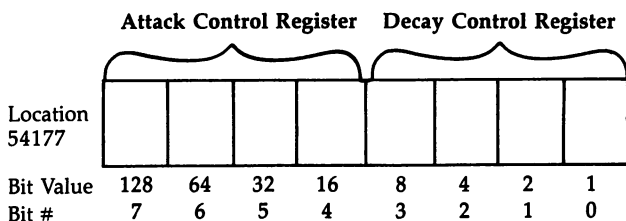
words, you can sustain a note indefinitely simply by not clearing the gate bit. You can use the sustain setting to set different volume levels for each of the 64's three voices. Since the volume control register at location 54296 is a *master* control, setting the volume for *all* voices, using the sustain is the easiest way to control volume when you're creating sounds with multiple voices.

- **Release.** Yet another function of time, *release* is the final descent of the sound's volume. A short release time drops the note off quickly, while a longer release time lengthens the fall of the note.

The SID chip is able to control the ADSR separately for each voice by using two registers. For example, location 54277 controls attack and decay for voice 1, while location 54278 controls the sustain and release. All you have to do is put a value in the register(s) and the SID chip takes care of the rest.

If you were to look at one of the ADSR control registers (there are actually six of them—two for each of the Commodore 64's three voices), you would find the register broken into several individual control bits, each of which has a specific function. Figure 1-4 shows voice 1's control register for attack and decay at location 54277.

Figure 1-4. Attack/Decay Control Register



As you can see, the register is divided into two sections. The upper section (bits 4–7; also called the *high nybble*) controls the length of the attack and the lower (bits 0–3; also called the *low nybble*) controls the length of the decay. Since there are only four bits in each nybble, the values you can use range from 0 to 15. The decay, being the low nybble, will accept those numbers directly as values. Thus to change the length of the decay, all you need to do is enter a value in the

register that is between 0 and 15 (decimal). The larger the number, the longer the decay.

You also have a range of 0 to 15 available when it comes to setting an envelope's attack rate. However, since it's the high nybble, you have to do some multiplication. To obtain the number to enter, you need to multiply the attack value (0-15) by 16, then POKE that total value into the register. To use both an attack and a decay at the same time, add the two values and POKE in the total. For instance, if you wanted an attack value of 8 and a decay of 9, you would figure the total like this:

$$(8 \times 16) + 9 = 137$$

That's the number you'd POKE into the register.

The sustain and release are set in the same way, only using the seventh register of each voice. The release rate is set by the low nybble (bits 0-3), using values from 0 to 15. The sustain level is set in the same way as attack, by multiplying the volume level you want by 16. Adding the release and sustain values together and POKEing the total into the register set both functions.

You can eliminate the multiplication necessary for the attack and sustain rates by referring to Table 1-2. It shows the values for decay and release as well.

Table 1-2. ADSR POKE Values

Value	Attack	Decay	Sustain	Release
0	0	0	0	0
1	16	1	16	1
2	32	2	32	2
3	48	3	48	3
4	64	4	64	4
5	80	5	80	5
6	96	6	96	6
7	112	7	112	7
8	128	8	128	8
9	144	9	144	9
10	160	10	160	10
11	176	11	176	11
12	192	12	192	12
13	208	13	208	13
14	224	14	224	14
15	240	15	240	15

The pitch registers control the frequency of the note or sound and consist of the first two memory locations of each voice. The numbers stored in these locations define a binary number between 0 and 65535. Each time the count in the first location (the lower register) reaches 256, the extra count is carried over to the second location (upper register). The number represented in the two registers is equal to the lower value, plus the upper register multiplied by 256. For instance, if the lower register contains a 3, and the upper register contains a 10, the value represented would be:

$$3 + (10 \times 256) = 2563$$

Most of the time you will be able to use the values in Table 1-3, simply POKEing the first number into the low register and the second into the high register. If you wanted to create an A# from the first octave, for example, and use voice 1, you would use something like:

```
POKE 54272,221:POKE 54273,1
```

The low register (54272) POKES the first pitch value, and the high register (54273) POKES the second pitch value.

Using the low and high registers for each of the Commodore 64's voices, along with Table 1-3, you should be able to create almost any tone or frequency you want. Some applications, like sweeping the scales, are a bit more difficult, but that will be covered later in this chapter.

Table 1-3. Musical Note POKE Values

Musical Note	Actual Frequency	Pitch Value Lower Half	Pitch Value Upper Half
C	16 Hz	12	1
C #	17 Hz	28	1
D	18 Hz	45	1
D #	19 Hz	62	1
E	21 Hz	81	1
F	22 Hz	101	1
F #	23 Hz	123	1
G	24 Hz	145	1
G #	25 Hz	169	1
A	27 Hz	195	1
A #	29 Hz	221	1
B	31 Hz	250	1

1
 SID: The Sound Interface Device

Musical Note	Actual Frequency	Pitch Value Lower Half	Pitch Value Upper Half
C	24 Hz	24	2
C #	34 Hz	56	2
D	37 Hz	90	2
D #	39 Hz	125	2
E	41 Hz	163	2
F	44 Hz	203	2
F #	46 Hz	246	2
G	49 Hz	35	3
G #	52 Hz	83	3
A	55 Hz	134	3
A #	58 Hz	187	3
B	62 Hz	244	3
C	65 Hz	48	4
C #	69 Hz	112	4
D	73 Hz	180	4
D #	76 Hz	251	4
E	82 Hz	71	5
F	87 Hz	151	5
F #	92 Hz	237	5
G	98 Hz	71	6
G #	104 Hz	167	6
A	110 Hz	12	7
A #	117 Hz	119	7
B	123 Hz	233	7
C	131 Hz	97	8
C #	139 Hz	225	8
D	147 Hz	104	9
D #	156 Hz	247	9
E	165 Hz	143	10
F	175 Hz	47	11
F #	185 Hz	218	11
G	196 Hz	142	12
G #	208 Hz	77	13
A	220 Hz	24	14
A #	233 Hz	238	14
B	247 Hz	210	15
C	262 Hz	195	16
C #	277 Hz	194	17
D	294 Hz	208	18
D #	311 Hz	238	19
E	330 Hz	30	21
F	349 Hz	95	22

Musical Note	Actual Frequency	Pitch Value Lower Half	Pitch Value Upper Half
F #	370 Hz	180	23
G	392 Hz	29	25
G #	415 Hz	155	26
A	440 Hz	48	28
A #	466 Hz	221	29
B	494 Hz	164	31
C	523 Hz	134	33
C #	554 Hz	132	35
D	587 Hz	161	37
D #	622 Hz	221	39
E	659 Hz	60	42
F	698 Hz	191	44
F #	740 Hz	104	47
G	784 Hz	58	50
G #	831 Hz	55	53
A	880 Hz	97	56
A #	932 Hz	187	59
B	988 Hz	72	63
C	1046 Hz	12	67
C #	1109 Hz	8	71
D	1175 Hz	66	75
D #	1244 Hz	187	79
E	1319 Hz	121	84
F	1397 Hz	127	89
F #	1480 Hz	209	94
G	1568 Hz	117	100
G #	1661 Hz	110	106
A	1760 Hz	194	112
A #	1865 Hz	118	119
B	1976 Hz	145	126
C	2093 Hz	24	134
C #	2217 Hz	17	142
D	2349 Hz	132	150
D #	2489 Hz	119	159
E	2637 Hz	242	168
F	2794 Hz	254	178
F #	2960 Hz	163	189
G	3136 Hz	234	200
G #	3322 Hz	220	212
A	3520 Hz	132	225
A #	3729 Hz	237	238
B	3951 Hz	34	253

Waveforms are selected by using one of the waveform selection bits in conjunction with the enable, or gate, bit. Each voice on the 64 has its own register to select the waveform, as well as its own gate bit. For example, voice 1 uses the register at location 54276 for both waveform selection and enabling the wave. Refer to Table 1-1 for the other voices' register addresses.

The four waveforms are selected by setting the appropriate bit in the waveform control register. If you were to look at one of these registers (there are three of them—one for each of the Commodore 64's three voices), you would find that it's made up of several control bits, each of which has a specific function. Figure 1-5 shows the waveform control register for voice 1 at location 54276.

Figure 1-5. Waveform Control Register

Location 54276. Wave Form Register for Voice 1								
Bit #	7	6	5	4	3	2	1	0
Bit								
Value	128	64	32	16	8	4	2	1
Noise Wave Enable	Pulse Wave Enable	Sawtooth Wave Enable	Triangle Wave Enable	Test Bit	Ring Modulation	Sync Bit	Enable Bit	

In addition to having specific functions, each bit has a specific bit value. To turn that bit *on*, you simply POKE the appropriate bit value into the control register. Bit 4, which has a value of 16, will turn on the triangle waveform generator. Bit 5, with a value of 32, enables the sawtooth waveform, while bit 6 turns on the pulse waveform generator if its value of 64 is used. The last waveform, noise, is enabled by setting bit 7, which has a value of 128. For example, to enable the pulse waveform bit, you would POKE the register with 64. This will turn on the pulse waveform generator. However, to get any sound out of the SID, you must also turn on the gate bit (bit 0). If you don't, you won't hear your sound. So every time you turn on a waveform bit, you should add 1 (the bit value of bit 0) to the value you select. Thus to turn on the pulse

wave generator and enable it, you would POKE the control register with 65. To turn it off, you would POKE the register with 64. (Remember that the gate bit's value is 1; $65 - 1 = 64$.)

Pulse width must be set whenever you select the pulse waveform for any of the three voices. It's controlled in much the same way as the pitch registers. As with the pitch registers, there are two locations used for pulse width selection in each voice. Voice 1's pulse width registers, for instance, are at addresses 54274 and 54275. However, there are only four bits available in the second register, so the numbers stored in these locations define a binary number between 0 and 4095. Just as in the pitch registers, the number represented by the two registers is equal to the lower value plus the upper value multiplied by 256. For example, if the lower register contains a 0, and the upper register an 8, the value would be:

$$0 + (8 \times 256) = 2048$$

This number signifies the proportion of the wave that will be high. The above value would be exactly 50 percent ($2048 / 4096 = .50$), creating a square waveform. Decreasing or increasing the number represented by the values in these registers will decrease or increase the proportion of the wave that is high and thus change the sound produced. If you used the following statement, for instance, the wave would be high only 25 percent of the time.

```
POKE 54274,0:POKE 54275,4
```

A sustain loop is usually added to a sound routine so that it plays longer. You can force a note to play indefinitely by simply not turning the gate bit off, or by turning the gate bit off manually, perhaps with a function key defined elsewhere in the program. But the easiest way to set the length of the note's sustain level is to use a FOR-NEXT loop as a delay. This loop should be inserted *between* the time you set the waveform (and thus turn on the gate bit) and the time when you turn the gate bit off. If you were using the pulse waveform, and had already set the other sound parameters earlier in the routine, it could look like this:

```
POKE 54276,65      Set pulse waveform and turn on gate bit
FOR T=0 TO 1000:NEXT
                    Delay loop to play the sound for one
                    second
POKE 54276,64      Turn off gate bit
```

Keep in mind that sustain refers only to a volume level, not to a function of time, as the other elements of a sound's ADSR do. That's why you need something like a delay loop in your routines; it's the simplest way to control the *time* of the sustain part of a note.

Turning the gate bit off is the last thing you'll do in most of your sound routines. Once the note has played—having gone through its attack, decay, sustain, and release steps—you need to turn the gate bit off by POKING the waveform control register with a value one less than when you selected the waveform and turned the gate bit on. If you earlier POKED location 54276 with 65 (to turn on the pulse waveform generator and enable the gate bit), now you'd POKE the same location with 64. This will turn off the gate bit, ending the sustain and bringing on the release section of the note's sound envelope. As soon as the release finishes, the sound is completed.

Once you understand the process you need to go through to create sounds with the SID chip, you're ready to produce your first note. Here's a routine that clears the sound control registers; sets the volume, ADSR, frequency, and waveform for the note; enables the gate bit; keeps the note playing; and turns the gate bit off. It's simple, but it's a start.

Program 1-1. Simple Sound

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 25
30 REM - TURN ON VOLUME -                       :rem 95
40 POKE 54296,15                               :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -         :rem 78
60 POKE 54277,0:POKE 54278,240:POKE 54275,8   :rem 3
70 REM - POKE TONE VALUES INTO VOICE 1       :rem 142
80 POKE 54272,48:POKE 54273,28                :rem 55
90 REM - ENABLE TONE REGISTER -                 :rem 233
100 POKE 54276, 65                             :rem 95
110 REM - PLAY TONE FOR 1 SECOND -              :rem 16
120 FOR R=0 TO 1000:NEXT                       :rem 22
130 REM - TURN SOUND REGISTER OFF -            :rem 228
140 POKE 54276,64                             :rem 98
150 REM - TURN VOLUME OFF -                   :rem 208
160 POKE 54296,0                               :rem 44

```

Let's go through this short routine line by line to see how the sound is produced. Line 20 clears the 24 sound control registers, using a FOR-NEXT loop to POKE each register with 0. The volume is turned on in line 40 and is set to its maximum value, 15. Line 60 initializes the ADSR registers, as well as the pulse width register. Attack and decay are both set to 0, indicating the fastest possible rate, by POKEing location 54277 with 0. By POKEing 240 into location 54278, you're signaling the computer to use a high sustain level, and a low release rate. The sound will continue after the decay at a high volume, but will end quickly. Because you're using the pulse waveform in this routine, you also need to set the pulse width register (location 54275) for voice 1. Frequency values are POKEd into the address for voice 1, locations 54272 and 54273 in line 80. By looking at Table 1-3, you can see that these values will create an A note in the fifth octave. Line 100 turns on the gate bit (by setting bit 0 to 1) and selects the pulse waveform (by setting bit 6). The value POKEd into this location is the total of those two bits' values (1+64). See Table 1-1 for the bits to set for these registers. A short delay loop keeps the note playing in line 120, and then the gate bit is disabled in line 140. The volume register is turned off in line 160 by POKEing a 0 value into it.

Don't be discouraged if some of this is confusing to you. Although it may seem complicated, you'll soon see how it all fits together. But it's nice to hear a created sound, even if you're not sure how it all was done. It's relatively simple to modify various registers to change the sound. For instance, by changing the values in the upper and/or lower pitch registers (54272 and 54273), you can change the tone played. Program 1-2 does this.

Program 1-2. Modifying Upper/Lower Tones

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 25
30 REM - TURN ON VOLUME -                       :rem 95
40 POKE 54296,15                               :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -         :rem 78
60 POKE 54275,8:POKE 54277,0:POKE 54278,240   :rem 3
70 REM - GET TONE FROM TONE TABLE -           :rem 141
75 READ L,H: IF L=999 THEN RESTORE: GOTO 75    :rem 119
```



```

77 REM - POKE TONE VALUES INTO VOICE 1      :rem 149
80 POKE 54272,L: POKE 54273,H                :rem 245
90 REM - ENABLE TONE REGISTER -                :rem 233
100 POKE 54276, 65                            :rem 95
110 REM - PLAY TONE FOR 1/8 SECOND -           :rem 119
120 FOR R=0 TO 125: NEXT                      :rem 237
130 REM - TURN SOUND REGISTER OFF -           :rem 228
140 POKE 54276,64                             :rem 98
150 REM - TURN VOLUME OFF -                   :rem 208
160 GOTO 75                                    :rem 60
165 POKE 54296,0                               :rem 49
170 REM - TONE TABLE -                       :rem 116
180 DATA 195,16,194,17,208,18,238,19,30,21,95,22,1
      80,23,29,25,155,26,48,28,221           :rem 15
190 DATA 29,164,31,134,33,164,31,221,29,48,28,155,
      26,29,25,180,23,95,22,30,21           :rem 204
195 DATA 238,19,208,18,194,17,999,999      :rem 83

```

This program uses a number of DATA statements containing the tone values for a musical scale. Each time a tone is to be played, a pair of values is READ from the DATA statements. These are the values used to POKE into the registers at locations 54272 and 54273. Notice that there are an even number of values in the DATA statements beginning in line 180. If you group the numbers in pairs, you will see that the first is the low pitch value and the second is the high pitch value. If you modify this routine by entering data of your own, make sure that there is an even number of values, and that the low pitch value precedes the high pitch value. Line 80 actually places these values in the registers. When all the numbers have been READ, the routine in line 75 RESTORES the DATA and starts again. To stop this routine, press the RUN/STOP and RESTORE keys at the same time.

Volume Adjustments

In addition to changing the frequency of a tone, you can also change the volume (loudness) of a tone. The volume for sounds created by any of the 64's three voices is controlled by the control register at location 54296. Values ranging from 0 to 15 can be POKEd into this register to set the sound volume level. You can even POKE a variable, such as V, into the register and change the sound within the routine. Program 1-3 does this.

Program 1-3. Volume Variables

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 25
25 REM -- INITIALIZE VOLUME CONTROL --         :rem 234
27 V=15: I=1                                   :rem 83
30 REM - TURN ON VOLUME -                       :rem 95
40 POKE 54296,V                                 :rem 31
50 REM - INITIALIZE SPECIAL REGISTERS -        :rem 78
60 POKE 54277,0:POKE 54278,240:POKE 54275,8   :rem 3
70 REM - GET TONE FROM TONE TABLE -          :rem 141
75 READ L,H: IF L=999 THEN RESTORE: GOTO 75    :rem 119

77 REM - POKE TONE VALUES INTO VOICE 1       :rem 149
80 POKE 54272,L: POKE 54273,H                 :rem 245
90 REM - ENABLE TONE REGISTER -               :rem 233
100 POKE 54276, 65                             :rem 95
105 REM - INCREMENT/TEST VOLUME CONTROL-      :rem 225
107 V=V-I:IF V=1 OR V=15 THEN I=I*-1         :rem 164
110 REM - PLAY TONE FOR 1/8 SECOND -          :rem 119
120 FOR R=0 TO 125: NEXT                       :rem 237
130 REM - TURN SOUND REGISTER OFF -          :rem 228
140 POKE 54276,64                              :rem 98
150 REM - TURN VOLUME OFF -                  :rem 208
160 GOTO 75                                     :rem 60
165 POKE 54296,0                                 :rem 49
170 REM - TONE TABLE -                       :rem 116
180 DATA 195,16,194,17,208,18,238,19,30,21,95,22,1
      80,23,29,25,155,26,48,28,221           :rem 15
190 DATA 29,164,31,134,33,164,31,221,29,48,28,155,
      26,29,25,180,23,95,22,30,21          :rem 204
195 DATA 238,19,208,18,194,17,999,999      :rem 83

```

Much the same as Program 1-2, this routine incorporates a volume control variable (V) and an incremental variable (I) to change the volume as the tones are played. Line 27 initializes these values, and they are incremented in line 107. Note that instead of POKEing a set value into 54296, in line 40 you are now POKEing the variable (V) instead.

Using volume to change the nature of a sound can be done on a note-by-note basis. If, for example, you change the volume of a note as it's played, you can create an entirely different kind of sound. That's what Program 1-4 does.

Program 1-4. Note-by-Note Volume

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 REM --- CLEAR SOUND REGISTERS --- :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT :rem 25
30 REM - TURN VOLUME ON :rem 50
40 POKE 54296,15 :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS - :rem 78
60 POKE 54277,0:POKE 54278,240:POKE 54275,8 :rem 3
70 REM - GET TONE FROM TONE TABLE - :rem 141
75 READ L,H: IF L=999 THEN RESTORE: GOTO 75
:rem 119
77 REM - POKE TONE VALUES INTO VOICE 1 :rem 149
80 POKE 54272,L: POKE 54273,H :rem 245
90 REM - ENABLE TONE REGISTER - :rem 233
100 POKE 54276, 65 :rem 95
110 REM - PLAY TONE FOR 1/8 SECOND AND ADJUST VOLU
ME - :rem 237
120 FOR R=15 TO 0 STEP -1:POKE 54296,R:FORT=1TO5:N
EXT:NEXT :rem 122
130 REM - TURN SOUND REGISTER OFF - :rem 228
140 POKE 54276,64 :rem 98
150 REM - TURN VOLUME OFF - :rem 208
160 GOTO 75 :rem 60
165 POKE 54296,0 :rem 49
170 REM - TONE TABLE - :rem 116
180 DATA 195,16,194,17,208,18,238,19,30,21,95,22,1
80,23,29,25,155,26,48,28,221 :rem 15
190 DATA 29,164,31,134,33,164,31,221,29,48,28,155,
26,29,25,180,23,95,22,30,21 :rem 204
195 DATA 238,19,208,18,194,17,999,999 :rem 83
```

The main difference between this routine and Program 1-3 is that here we've used a FOR-NEXT loop in line 120 to adjust the volume each time the routine executes. Set up as variable R, the volume begins at 15, and each time through the loop decreases by 1. The second FOR-NEXT loop in that line is a delay, so that you can hear the changes in volume more easily.

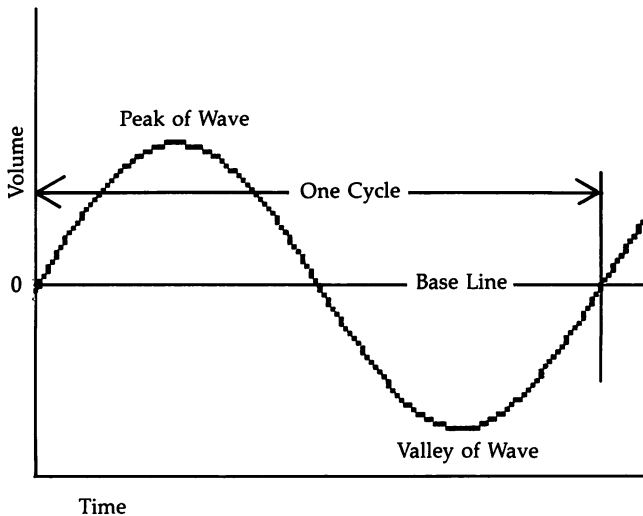
As each note plays, a different volume level is set. You can use this technique in your own routines to make a sound effect begin loudly, fade away, then become loud again. As you can see, modifying the volume control register on the 64 isn't very hard, and can dramatically alter the sound you hear.

Changing the ADSR

We've already looked briefly at the ADSR of a sound envelope, and even explored the registers that control this important part of sound creation on the Commodore 64. The previous example sound routines have included various rates for the attack, decay, and release, as well as volume levels for the note's sustain. But there are a few more things that should be pointed out about sound waves and their ADSRs.

If you graphed the shape of a sound wave, it would look similar to Figure 1-6. A pure sound wave begins at a base level, indicated by zero in the graph, rises to its peak, and falls.

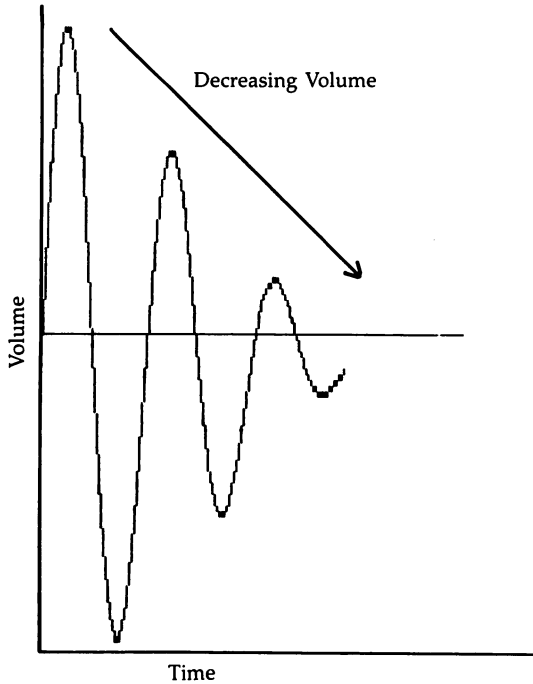
Figure 1-6. Pure Sound Wave



A *cycle* is measured from one point on the sound wave to the same position on the next sound wave. The number of cycles that occur each second is called the *frequency*.

The *volume* of a note is determined by looking at how high or low the wave's peaks and valleys are. The higher the peaks and the lower the valleys, the louder the sound will be. Figure 1-7 shows a sound wave that begins loudly and becomes quiet as time goes on.

Figure 1-7. Decreasing Sound



In this sound wave, you could say that the *attack* is very fast and the *decay* is relatively slow. Remember that *attack* refers to the rate of time it takes the sound to reach maximum volume, and that *decay* is the rate of time it takes the sound to fall to its sustain level. The sound wave shown in Figure 1-7 starts out loud, so its attack is set at a fast rate. On the other hand, it loses volume gradually; that's because its decay is set to a rather slow rate. All of this gives the feeling of a sound that can be called *hard*. To produce a softer sounding tone, you could reverse the attack and decay rates, producing a sound that has a slow attack and a fast decay.

Program 1-5 creates a hard sound, one with a fast attack rate and a slow decay. Using voice 1, the attack/decay register at location 54277 is POKED with a value of 10 in line 40. This sets attack at 0 (fast) and decay at 10 (relatively slow). See Table 1-2 for the values you can POKE into this register to allow for various rates of attack and decay. Note also that we've used a different waveform for this example. Unlike the

previous routines, which used the pulse waveform, Program 1-5 uses the triangle waveform, set in line 60 (and the gate bit enabled) by POKEing a value of 17 into location 54276.

Program 1-5. Fast Attack—Slow Decay

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM CLEAR CHIP :rem 208
20 FOR T= 0 TO 24 :POKE54272+T,0:NEXT :rem 212
25 REM - SET VOLUME LEVEL :rem 180
30 POKE 54296,15 :rem 46
35 REM - ATTACK/DECAY RATE :rem 242
40 POKE 54277,10:POKE54278,0 :rem 248
45 REM - FREQUENCY SET :rem 24
50 POKE 54272,40:POKE 54273,95 :rem 48
55 REM - TRIANGLE WAVE FORM AND GATE BIT ENABLED :rem 246
60 POKE 54276,17 :rem 49
70 FOR T=1 TO 2000:NEXT :rem 238
80 POKE 54276,16 :rem 50
90 GOTO 60 :rem 8

```

Modifying the attack and decay is simple. For instance, if you want a softer sound, you can change line 40 to:

```
40 POKE 54277,208:POKE 54278,0
```

Now the attack is long (indicated by using a value of 208, which is 16×13), and the decay is very short (since the decay is set to 0). Remember that the value you finally POKE into the attack/decay control register is a sum of the attack value (208) and the decay (0). The sound rises to its maximum volume slowly, and falls rapidly to its sustain level (which has been set to volume 0 by POKEing location 54278 with 0). An even softer sound can be created by manipulating the attack and decay rates so that both are long. Changing line 40 again does this:

```
40 POKE 54277,172:POKE 54278,0
```

If you look at Table 1-2 for a moment, you'll see that we've set the attack to 10 (160) and the decay to 12 (12). Adding those values together gives you the total value of 172, which is what is POKEd into the control register.

You can even change some of the earlier examples by POKEing new values in the attack/decay register. For instance, try replacing line 60 in Program 1-2 with this:

```
60 POKE 54275,8:POKE 54277,9:POKE 54278,0
```

This sets the attack to a fast rate (0) and the decay to a longer rate (9). The sustain and release have both been set to 0 so that you can more easily hear the difference attack and decay make. Notice how the sound changes as you run this altered version. Lengthening the rate of the attack should be simple for you now. All you have to do is add 128 (which is $8*16$) to the decay value of 9. The total is 137, which indicates a longer attack, as well as a long decay rate. Change line 60 again so that it reads:

```
60 POKE 54275,8:POKE 54277,137:POKE 54278,0
```

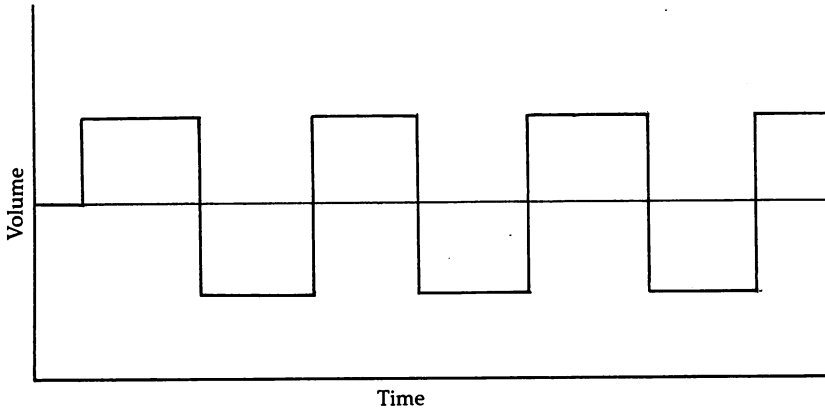
Setting the sustain and release of a note is done in the same way, by POKEing in values from Table 1-2 to the appropriate register. All you have to remember is that the sustain indicates the *volume level* of the note's sustain, while the release signals the rate at which a note falls from that level to volume 0. Experiment with the sound routine examples you've used so far, changing the sustain/release values POKEd into location 54278, the control register for voice 1. You'll be amazed at the different sounds you can create just by changing that one parameter.

Waveforms

In addition to changing the frequency, volume, and ADSR of the notes you produce, the SID also allows you to produce different waveforms. The forms available are pulse waves (which we've been using in most of the examples), triangle waves, sawtooth waves, and noise. The control register for waveform selection in voice 1 is found in location 54276. Changing the value POKEd into that location changes the waveform used to produce the note.

The pulse waves, which we've used in most of the example routines up to now, have all been *symmetrical*. (That's sometimes called a *square waveform*.) That is, they rise, remain at the upper peak for a period of time, fall, and remain at the valley for exactly the same length of time before repeating. Figure 1-8 shows a symmetrical waveform.

Figure 1-8. Square Waveform



Whenever you select a pulse waveform, you must also place values in the *pulse width* registers. Read over the earlier explanation of how to select a pulse width if you've forgotten how to place values in these two registers. By adjusting the pulse width of the pulse waves, we can produce sounds that range from very full to very thin. What you're doing, in effect, is creating an asymmetrical pulse waveform.

Program 1-6. Symmetrical/Asymmetrical Waveforms

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

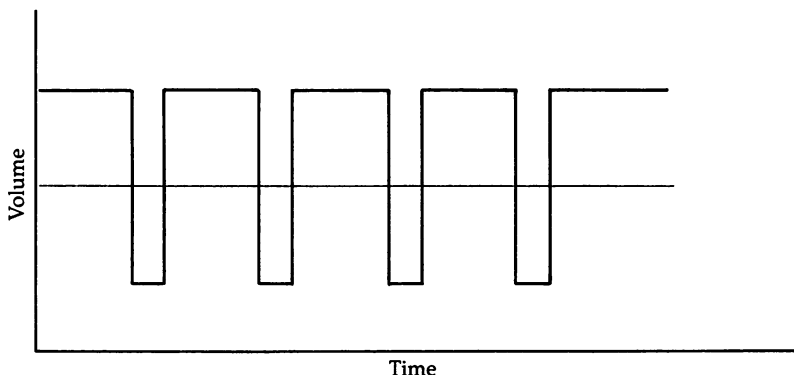
```

10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 25
30 REM - TURN ON VOLUME -                       :rem 95
40 POKE 54296,15                               :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -         :rem 78
60 POKE 54275,8                                 :rem 0
65 POKE 54277,0:POKE 54278,240                 :rem 52
70 REM - POKE TONE VALUES INTO VOICE 1       :rem 142
80 POKE 54272,48:POKE 54273,28                :rem 55
90 REM - ENABLE TONE REGISTER -                 :rem 233
100 POKE 54276, 65                             :rem 95
110 REM - PLAY TONE FOR 1 SECOND. -             :rem 16
120 FOR R=0 TO 1000: NEXT                      :rem 22
125 IF P=1 THEN 130                            :rem 168
126 P=1:POKE 54275,14: GOTO 65                 :rem 54
130 REM - TURN SOUND REGISTER OFF -            :rem 228
140 POKE 54276,64                              :rem 98
150 REM - TURN VOLUME OFF -                   :rem 208
160 POKE 54296,0                               :rem 44

```


The first tone produced by this routine is symmetrical, but the second tone is not. That's because the pulse width is first set to 50 percent in line 60 by POKEing 8 into the upper half of the two-byte pulse width register. Remember that the total value of the registers is the lower half value added to the upper half value multiplied by 256 (Lower value + (Upper value * 256)). In this routine, only the upper register (location 54275) is used. Then, in line 126, the pulse width is changed, so that a value of 14 is POKEd into the upper register. That gives you a total of 3584 (14*256), or a pulse width of 87.5 percent. In other words, that percentage of the wave is high, or above the base line volume. If you graphed that second tone, it would look like this:

Figure 1-9. Asymmetrical Pulse Waveform



Triangle waves, as their name implies, look like the tops and bottoms of triangles, just as symmetrical pulse waves look like the tops and bottoms of squares. They are always symmetrical and produce a very soft, almost subdued sound. Setting bit 4 of the waveform control register turns on the triangle waveform generator. Normally, you'll set both the triangle waveform and the gate bit by POKEing 17 into the control register. Type in and RUN Program 1-7, which demonstrates a triangle waveform.

Program 1-7. Triangle Waveform

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 25
30 REM - TURN VOLUME ON                       :rem 50
40 POKE 54296,15                               :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -       :rem 78
60 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 3
70 REM - GET TONE FROM TONE TABLE -         :rem 141
75 READ L,H: IF L=999 THEN RESTORE: GOTO 75    :rem 119

77 REM - POKE TONE VALUES INTO VOICE 1       :rem 149
80 POKE 54272,L: POKE 54273,H                 :rem 245
90 REM - ENABLE TONE REGISTER -               :rem 233
100 POKE 54276,17                             :rem 92
110 REM - PLAY TONE FOR 1/8 SECOND AND ADJUST VOLU
    ME -                                       :rem 237
120 FOR R=15 TO 0 STEP -1:POKE 54296,R:NEXT :rem 70
130 REM - TURN SOUND REGISTER OFF -          :rem 228
140 POKE 54276,16                             :rem 95
150 GOTO 75                                    :rem 59
160 REM - TURN VOLUME OFF -                  :rem 209
165 POKE 54296,0                               :rem 49
170 REM - TONE TABLE -                      :rem 116
180 DATA 195,16,194,17,208,18,238,19,30,21,95,22,1
    80,23,29,25,155,26,48,28,221             :rem 15
190 DATA 29,164,31,134,33,164,31,221,29,48,28,155,
    26,29,25,180,23,95,22,30,21             :rem 204
195 DATA 238,19,208,18,194,17,999,999      :rem 83

```

Figure 1-10 shows a triangle waveform. Note that it's symmetrical, with equal high and low portions. Unlike the pulse waveform, however, you can't control the percentage of the high part of the wave to the low section. A triangle waveform is *always* symmetrical.

Sawtooth waves are not symmetrical. They rise slowly and fall abruptly. They begin like a triangle wave and end like a pulse wave. Figure 1-11 illustrates a sawtooth waveform.

They produce the sharpest sound of the waveforms available on the Commodore 64. By setting bit 5 of the waveform control register, you can enable the sawtooth waveform. POKEing 33 (32 for the waveform, 1 for the gate bit) will turn

on this waveform's generator. If you've already entered and saved Program 1-7, you can simply change lines 100 and 140 to what you see in Program 1-8. That's all you have to do to switch waveforms.

Figure 1-10. Triangle Wave

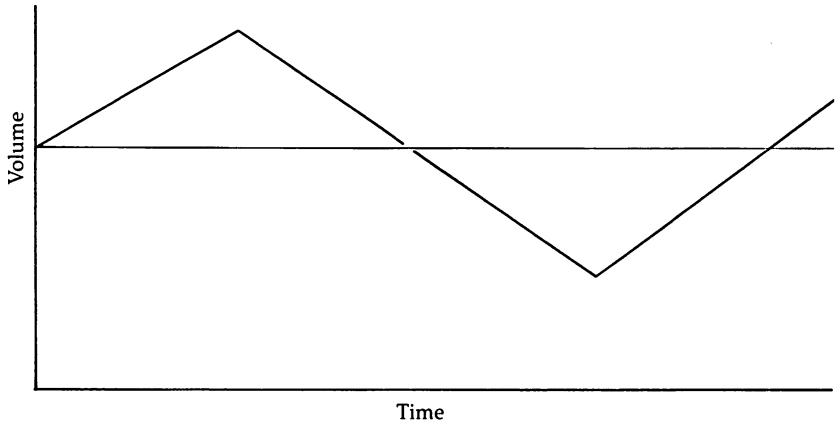
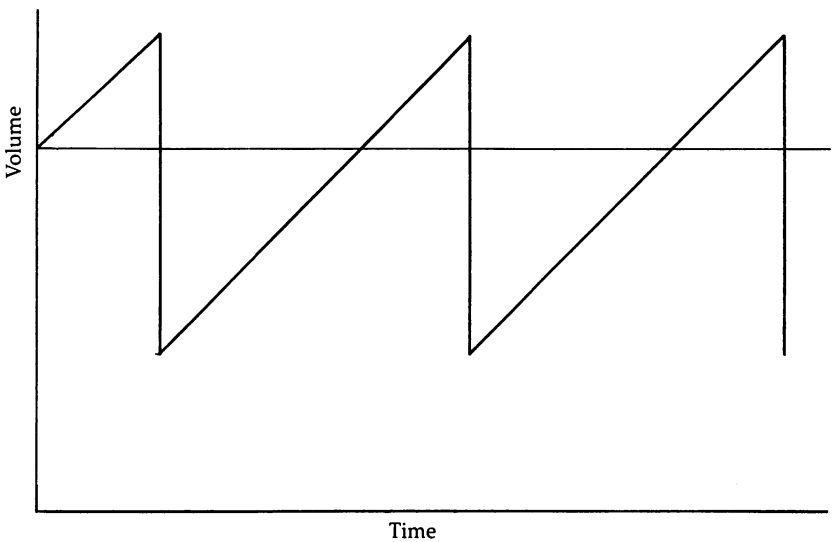


Figure 1-11. Sawtooth Wave



Program 1-8. Sawtooth Waveform

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 25
30 REM - TURN VOLUME ON                       :rem 50
40 POKE 54296,15                               :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -       :rem 78
60 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 3
70 REM - GET TONE FROM TONE TABLE -          :rem 141
75 READ L,H: IF L=999 THEN RESTORE: GOTO 75    :rem 119

77 REM - POKE TONE VALUES INTO VOICE 1       :rem 149
80 POKE 54272,L: POKE 54273,H                 :rem 245
90 REM - ENABLE TONE REGISTER -               :rem 233
100 POKE 54276,33                             :rem 90
110 REM - PLAY TONE FOR 1/8 SECOND AND ADJUST VOLU
    ME -                                       :rem 237
120 FOR R=15 TO 0 STEP -1:POKE 54296,R:NEXT :rem 70
130 REM - TURN SOUND REGISTER OFF -           :rem 228
140 POKE 54276,32                             :rem 93
150 GOTO 75                                    :rem 59
160 REM - TURN VOLUME OFF -                   :rem 209
165 POKE 54296,0                               :rem 49
170 REM - TONE TABLE -                       :rem 116
180 DATA 195,16,194,17,208,18,238,19,30,21,95,22,1
    80,23,29,25,155,26,48,28,221             :rem 15
190 DATA 29,164,31,134,33,164,31,221,29,48,28,155,
    26,29,25,180,23,95,22,30,21             :rem 204
195 DATA 238,19,208,18,194,17,999,999      :rem 83

```

Noise is probably the least understood waveform available on the 64. It generates random waves, creating a raspy, rough sound. It can be used for many different sound effects, and even as a component in music. Depending on the frequencies you select, it can seem like anything from a high hiss to a low rumbling sound. Often called *white noise*, it's especially useful in game sound effects such as explosions, crowd roars, or sputtering engines.

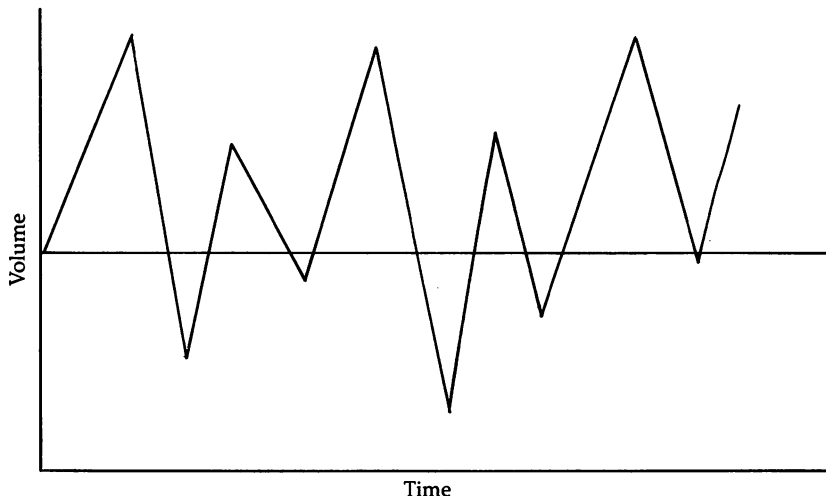
A noise waveform is certainly not symmetrical, and in fact can appear in many forms, one of which is shown in Figure 1-12.

Setting bit 7 of the waveform control register enables the noise generator. You set this bit by POKEing a value of 129 (128 for the waveform, 1 for the gate bit) into the control register of the voice you're using. To hear the noise waveform in

the example we've been using, just change lines 100 and 140 in Program 1-8 to:

```
100 POKE 54276,129  
140 POKE 54276,128
```

Figure 1-12. Noise Waveform



Getting Fancy

Now that you've seen how to create single notes, and even short routines that play a number of notes read from a DATA table, let's look at two short programs which demonstrate a sound effect that sounds impressive, yet is simple to create. It's a good way to get ready for Chapter 2, which will show you how to produce more intricate, and thus more entertaining, sound effects.

Using the frequency registers for voice 1, you can create an effect which sweeps the scales, starting at the bottom of the lowest octave and quickly rising to the highest frequency possible. It may sound difficult, but it's not. Take a look at Program 1-9, which uses this technique.

Program 1-9. Sweeping the Scales

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 FOR R=54272 TO 54296:POKE R,0:NEXT      :rem 24  
20 POKE 54296,15                          :rem 45
```

```

30 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 0
40 FOR H=0 TO 255 :rem 63
50 POKE 54272,0:POKE 54273,H :rem 214
60 POKE 54276,33 :rem 47
70 NEXT :rem 166
80 POKE 54276,32 :rem 48
90 POKE 54296,0 :rem 254

```

Most of this routine should look familiar to you by now. The FOR-NEXT loop in lines 40–70 generates values from 0 to 255 which are POKEd into the high frequency register. Only the high register is actually changed. A total of 256 tones play when you RUN this routine.

You can produce a similar effect by sliding through the pulse width values quickly. Although it sounds completely different from the previous program, Program 1-10 operates much the same way.

Program 1-10. Pulse Width Values

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

20 FOR R=54272 TO 54296:POKE R,0:NEXT :rem 25
30 POKE 54296,15 :rem 46
40 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 1
50 POKE 54272,48:POKE 54273,28 :rem 52
55 POKE 54276,65 :rem 56
60 FOR T=0 TO 4095 STEP 8:H=INT(T/256):L=T-(H*256)
:rem 182
80 POKE 54274,L:POKE 54275,H :rem 249
90 NEXT :rem 168
95 POKE 54276,64 :rem 59
100 POKE 54296,0 :rem 38

```

Line 60 may seem confusing. Actually, it's the heart of the sound routine. It produces the values for the high and low pulse width registers POKEd in line 80. First of all, a FOR-NEXT loop, ranging from 0 to 4095, is established, with a STEP command of 8. The maximum number that the two-byte pulse width register can contain is 4095. Each time through the loop, the variable T increments by 8. H, the variable for the high pulse width register, is the integer of T divided by 256. L, the variable for the low pulse width register, is equal to $T - (H * 256)$. If T equals 256, for instance, H equals 1 and L equals 0. Note that since we're dealing with pulse width, the

pulse waveform is enabled by POKEing the control register with 65.

You're ready to move on to some of the more difficult aspects of creating sound with the SID chip and the Commodore 64. If you still don't understand such things as ADSR, waveform, volume control, and how to select the parameters for sound effects and individual notes, it would be a good idea to go through this chapter again, perhaps even glancing through the glossary in the Introduction. If you don't know how, or why, simple sounds are created on the 64, the later chapters in this book may prove difficult.



CHAPTER

2

Music and the Sound Editor



2

Music and the Sound Editor

Musical instruments produce unique sounds that are based on the design of the instrument. A clarinet sounds like a clarinet because it's made from wood. The air pushed through it by the player moves between the reed and the mouthpiece, making the reed vibrate. As the player presses the keys, the air is forced through various holes, creating sounds. A trumpet sounds like a trumpet because it is made of brass. The player positions his lips and tongue to create sounds, and to stop and start them. The air blown into the mouthpiece moves past valves, various distances through tubes, and eventually out its bell. It's this process that makes the unique sound of the instrument.

Unlike a musical instrument, the Commodore 64 does not produce sounds based on its mechanical design. It creates special sounds by manipulating tones using electronic circuits. Because of this, the 64 is not limited to producing only one type of sound. It can mimic almost any instrument by simply duplicating that instrument's sound *patterns*. In addition, the 64 can even produce sounds that would be impossible for any normal musical instrument.

Since the 64 makes sounds digitally (using numbers), the computer can also store the sounds it makes so you can play them over and over, or mix them with other sounds at will. In a way, it's like having a musical instrument and a recording device in one package.

This chapter will show you how to use your 64 in just this way. You'll see how to program music on the 64, how to use it as a musical instrument, and how to store and play that music back.

Playing Notes

Chapter 1 showed you how to create simple sounds using the SID chip in your 64. Most of those were more like sound effects than actual tunes. Producing more complicated routines isn't that difficult once you know how to turn the sound on, and how to set the various control registers in the SID chip. We'll begin to do that in this chapter.

You'll remember that the pitches on the Commodore 64 are defined by using two memory locations for each of the three available voices. Table 1-1 shows the specific control registers for each voice and their locations in memory. Placing values in those locations selects the frequency of the note. You've already seen how to do this in Chapter 1. Take a look at Program 1-2 in Chapter 1 for a moment. The READ statement in line 75 READs values from the DATA statements in lines 180 through 195. Every time through the loop, the program READs two items from the list, a low value and a high value. These values are then POKEd into the pitch control registers to create the sound. Using this same program, you can play any tune you like by simply changing the values in the DATA statements. All you have to remember is that the values have to be paired: a low pitch value with a high pitch value. For example, if you replaced the DATA statements with a different set of values, you could create a tune like the one that Program 2-1 plays. (If you've previously saved a copy of Program 1-2 from Chapter 1, you can just replace lines 180-195 with those that show in the listing below.)

Program 2-1. Yankee Doodle

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 25
30 REM - TURN ON VOLUME -                       :rem 95
40 POKE 54296,15                               :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -         :rem 78
60 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 3
70 REM - GET TONE FROM TONE TABLE -           :rem 141
75 READ L,H: IF L=999 THEN RESTORE:GOTO 75:rem 119
77 REM - POKE TONE VALUES INTO VOICE 1       :rem 149
80 POKE 54272,L: POKE 54273,H                 :rem 245
90 REM - ENABLE TONE REGISTER -                 :rem 233
100 POKE 54276, 65                            :rem 95
110 REM - PLAY TONE FOR 1/8 SECOND -           :rem 119
```

```

120 FOR R=0 TO 125: NEXT           :rem 237
130 REM - TURN SOUND REGISTER OFF - :rem 228
140 POKE 54276,64                 :rem 98
150 REM - TURN VOLUME OFF -       :rem 208
160 GOTO 75                        :rem 60
165 POKE 54296,0                  :rem 49
170 REM - TONE TABLE -           :rem 116
180 DATA 195,16,195,16,208,18,30,21,195,16,30,21,2
    08,18,142,12,196,16,195,16    :rem 162
185 DATA 208,18,30,21,195,16,0,0,210,15,0,0,195,16
    ,195,16,208,18,30,21,95,22    :rem 131
190 DATA 30,21,208,18,195,16,210,15,142,12,24,14,21
    0,15,195,16,0,0,195,16,0,0    :rem 162
195 DATA 999,999                 :rem 59

```

To hear the differences that changes in the attack and decay rates can introduce, try altering the value in location 54277 in line 60 to 10, 172, or 208. You'll be surprised at the way the sound is modified by such a simple change.

By comparing the low and high register values in the DATA statements to Table 1-3, you should be able to see the actual notes that are being created. The first pair of values, for instance, is made up of a low value of 195 and a high value of 16. Table 1-3 shows that those values create a C note in the fifth octave. The second pair is another C, while the third pair (208 and 18) creates a D in that same octave. It's not too far from knowing the values that create each note to reading sheet music and transferring the notes you see there to values your 64 will understand. If you can read musical notation, it won't be hard.

Timing and Rhythm

Music isn't just a collection of notes played in sequence. It's also very much a matter of timing. *When* you play each note is as important as *which* note you play. Whenever you program a song, you must be aware of both the notes and the timing of the music; otherwise, you won't be able to duplicate the song accurately.

The tune you just played after entering and running Program 2-1 has several pairs of 0's in its DATA statements. The first pair you see is in line 185. The 0's are placed there to provide pauses in the music. Without those pauses, the music would consist of bland notes with no rhythm. You can hear this yourself by eliminating the 0's, and the commas between

them, from lines 185 and 190 in Program 2-1. There are four pairs altogether. After you've done that, RUN the program again and listen for the difference. The first part of the tune is recognizable, but as it reaches the middle, it starts sounding strange. The same *notes* are being played, but the timing and rhythm are off.

By putting the pauses in different places in your music, you can change the whole feel of the song. Remember, one of the main differences in styles of music, such as jazz and rock, is rhythm.

Program 2-2 uses the framework of Program 2-1, but the DATA statements have been changed. Additional pairs of 0's acting as pauses have been placed in the lines. You can simply enter the new DATA statements into your saved version of Program 2-1 and rerun it. Although you'll recognize the tune, it will be distorted by the pauses, slowing it down so that it sounds like a version you'd hear from a jack-in-the-box toy. Try it out.

Program 2-2. "Yankee Doodle"—with Pauses

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 25
30 REM - TURN ON VOLUME -                       :rem 95
40 POKE 54296,15                                :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -         :rem 78
60 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 3
70 REM - GET TONE FROM TONE TABLE -           :rem 141
75 READ L,H: IF L=999 THEN RESTORE: GOTO 75

77 REM - POKE TONE VALUES INTO VOICE 1       :rem 119
80 POKE 54272,L: POKE 54273,H                  :rem 245
90 REM - ENABLE TONE REGISTER -                 :rem 233
100 POKE 54276, 65                             :rem 95
110 REM - PLAY TONE FOR 1/8 SECOND -           :rem 119
120 FOR R=0 TO 125: NEXT                       :rem 237
130 REM - TURN SOUND REGISTER OFF -            :rem 228
140 POKE 54276,64                              :rem 98
150 REM - TURN VOLUME OFF -                    :rem 208
160 GOTO 75                                     :rem 60
165 POKE 54296,0                                :rem 49
170 REM - TONE TABLE -                        :rem 116
180 DATA 195,16,0,0,195,16,208,18,0,0,30,21,195
                                                :rem 247
  
```

```

182 DATA 16,0,0,30,21,208,18,0,0,142,12,196,16,0,0
    ,195,16 :rem 204
185 DATA 208,18,0,0,30,21,195,16,0,0,0,0,210,15
    :rem 220
187 DATA 0,0,0,0,195,16,0,0,195,16,208,18,0,0,30,2
    1,95,22,0,0 :rem 133
190 DATA 30,21,208,18,0,0,195,16,210,15,0,0,142
    :rem 227
192 DATA 12,24,14,0,0,210,15,195,16,0,0,0,0,195,16
    ,0,0,0,0,999 :rem 184
195 DATA 999 :rem 100

```

While you can slow the timing of a song by adding pauses, there's nothing you can add to speed up individual notes. To make a tune play faster, you can do one of two things: begin with faster notes, using shorter rates of attack and decay, for example, decreasing the length of the sustain loop, and only then add pauses to notes which require a longer length; or you can change the timing of the music as it is running by utilizing separate timing loops.

Pausing Techniques

Several short programs can show you how pausing and delays work. In this first example, the speed of the drumbeats is adjusted by maintaining a steady rhythm and adding pauses wherever necessary.

Program 2-3. Snare Drums

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM --- CLEAR SOUND REGISTERS --- :rem 251
20 FOR R=54272 TO 54296:POKE,R,0: NEXT :rem 25
30 REM - TURN ON VOLUME - :rem 95
40 POKE 54296,15 :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS - :rem 78
60 POKE 54277,0:POKE54278,240 :rem 47
70 REM - GET TONE FROM TONE TABLE - :rem 141
75 READ L,H: IF L=999 THEN RESTORE: GOTO 75
    :rem 119
77 REM - POKE TONE VALUES INTO VOICE 1 :rem 149
80 POKE 54272,L:POKE 54273,H :rem 245
90 REM - ENABLE TONE REGISTER - :rem 233
100 POKE 54276,129 :rem 144
110 REM - PLAY TONE FOR 1/64 SECOND - :rem 169
120 FOR R=0 TO 16: NEXT :rem 188
130 REM - TURN GATE BIT OFF :rem 201

```

```

140 POKE 54276,128:GOTO 75                :rem 114
170 REM - NOTE TABLE -                    :rem 116
180 DATA 1,14,1,14,1,14,1,14,0,0,0,0,1,14,0,0,0,0,
      1,14,0,0,0,0,999,999                :rem 25

```

This program uses the noise waveform, accessed by POKEing the waveform control register with 129 (128 for the waveform, 1 to enable the gate bit) to duplicate the sound of a drum. All the notes are played at the same speed, 1/64 second, by using the FOR-NEXT loop in line 120. The only things that change are the pauses between the notes. Also, the notes have the shortest possible attack and decay rates, set by POKEing 0 into location 54277 in line 60. This makes the notes seem quite short. You can hear a different sound by POKEing a value of 10 in this location. That retains a very short attack rate, but sets a long decay. The effect is still one of a snare drum, but now the rhythm is slightly different.

The next example shows another technique for altering rhythm in a tune. It leaves out the pauses, but changes the timing of the individual notes.

Program 2-4. Timing the Drums

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM --- CLEAR SOUND REGISTERS ---      :rem 251
20 FOR R=54272 TO 54296:POKER,0:NEXT     :rem 25
30 REM - TURN ON VOLUME -                 :rem 95
40 POKE 54296,15                          :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -   :rem 78
60 POKE 54277,0:POKE 54278,240           :rem 47
65 REM - POKE TONE VALUES INTO VOICE 1  :rem 146
67 POKE 54272,1: POKE 54273,14          :rem 252
70 REM - GET TIMING FROM RHYTHM TABLE -  :rem 197
75 READ T,TT:IF T=999 THEN RESTORE:GOTO 75:rem 231
90 REM - ENABLE TONE REGISTER -           :rem 233
100 POKE 54276,129                       :rem 144
110 REM - PLAY TONE FOR SPECIFIED TIME-   :rem 222
120 FOR R=0 TO T: NEXT                    :rem 169
130 REM - TURN GATE BIT OFF               :rem 201
140 POKE 54276,128                       :rem 147
165 FOR R=0 TO TT: NEXT:GOTO 75          :rem 229
170 REM - RHYTHM TABLE -                 :rem 26
180 DATA 8,8,8,8,8,8,8,8,128,8,128,8,128,999,999
                                           :rem 14

```


Instead of the pitch values being READ from DATA statements, as in Program 2-3, this routine READs the values for variables T and TT. These are then used in the delay loops found in lines 120 and 165. It's here that the length of the notes is established. T sets the length of the sustain portion of the note to 1/64 second. TT sets the length of the delay *between* notes. As you can see from looking over the DATA statement in line 180, the first three beats play just as long as the delay between the beats. The next three beats, however, play only one-sixteenth as long as the delay between them, for the loop runs from 0 to 128, instead of from just 0 to 8.

This method lets you adjust both the length of time the note is played and the length of the pause between notes. Notice, however, that the flexibility of changing pitches has been given up. Though this poses no major problems while creating the sound of a snare drum, this inflexibility can make a song cumbersome to program.

To be able to manipulate *both* the rhythm and the pitch, you'll need to read four different values from the DATA list for each note (or beat of the drum). In effect, what you're doing is combining aspects of Programs 2-3 and 2-4. Not only will values be READ from DATA for rhythm, but there will also be others READ for the pitch. Program 2-5 combines these methods.

Program 2-5. Pitches and Pauses in "Yankee Doodle"

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM --- CLEAR SOUND REGISTERS ---           :rem 251
20 FOR R=54272 TO 54296:POKE,R:NEXT           :rem 25
30 REM - TURN ON VOLUME -                       :rem 95
40 POKE 54296,15                               :rem 47
50 REM - INITIALIZE SPECIAL REGISTERS -         :rem 78
60 POKE 54277,0:POKE 54278,240                :rem 47
70 REM - GET TIMING FROM RYHTHM TABLE -       :rem 197
75 READ L,H,T,TT:IF L=999 THEN END            :rem 159
77 REM - POKE TONE VALUES INTO VOICE 1       :rem 149
80 POKE 54272,L:POKE 54273,H                 :rem 245
90 REM - ENABLE TONE REGISTER -                :rem 233
100 POKE 54276, 33                            :rem 90
110 REM - PLAY TONE FOR SPECIFIED TIME-       :rem 222
120 FOR R=0 TO T*3: NEXT                      :rem 6
130 REM - TURN GATE BIT OFF                   :rem 201
140 POKE 54276,32                             :rem 93

```

```

145 FOR R=0 TO TT*3: NEXT:GOTO 75           :rem 64
170 REM - RHYTHM TABLE -                 :rem 26
180 DATA 195,16,32,32,195,16,32,32,208,18,32,32,30
    ,21,32,32,195                          :rem 15
182 DATA16,32,32,30,21,32,32,208,18,32,32,142,12,3
    2,32,196,16,32,32,195,16,32          :rem 223
185 DATA 32,208,18,32,32,30,21,32,32,195,16,128,32
    ,210,15                                :rem 225
187 DATA 128,32,195,16,32,32,195,16,32,32,208,18,3
    2,32,30,21,32,32,95,22,32,32        :rem 239
190 DATA 30,21,32,32,208,18,32,32,195,16,32,32,210
    ,15,32,32,142,12,32,32             :rem 172
192 DATA 24,14,32,32,210,15,32,32,195,16,128,32,19
    5,16,128,32,999,999,999,999        :rem 10

```

This program reads several variables in line 75. L and H are the pitch values POKEd into the frequency control registers, while T and TT set the length of the sustain delay loop and the delay between notes, respectively. Instead of pairs of values in the DATA statements then, the values are grouped in fours. The first value becomes L, the second H, the third T, and the fourth TT. That's why there are so many more values in the DATA statements compared to previous examples. Although the list of numbers in the last six lines of the program may seem confusing, especially if you're just starting to use sound on the 64, it is the simplest way to regulate a song's rhythm, and at the same time be able to produce specific notes.

If that mass of numbers makes you think creating realistic music on the Commodore 64 is impossible, don't worry. There's a way to create those numbers relatively painlessly. Using a sound editor, you can easily come up with the values you'll need to hear almost any sound or musical piece you can think of. And just such an editor follows.

Tools to Put the Music Together

The song we produced in the last section is actually very simple. Nevertheless, the data required to produce it is fairly complex, and it's easy to see that a more involved song could make it nearly impossible to keep track of all the note changes and timing values.

One way to simplify this process is to organize the information and put it into a chart that you build as you write your song. Once you have that chart, you could simply translate

this table into data. Table 2-1 shows all the DATA for "Yankee Doodle" set up in the form of a chart like this:

Table 2-1. "Yankee Doodle" DATA

Lower Tone Value	Upper Tone Value	Note Length Value	Pause Length Value
195	16	32	32
195	16	32	32
208	18	32	32
30	21	32	32
195	16	32	32
30	21	32	32
208	18	32	32
142	12	32	32
196	16	32	32
195	16	32	32
208	18	32	32
30	21	32	32
195	16	128	32
210	15	128	32
195	16	32	32
195	16	32	32
208	18	32	32
30	21	32	32
95	22	32	32
30	21	32	32
208	18	32	32
195	16	32	32
210	15	32	32
142	12	32	32
24	14	32	32
210	15	32	32
195	16	128	32
195	16	128	32

If you compare the values in this table with the ones found in the DATA statements in Program 2-5, you'll see that they're identical. By listing the values in groups of four, just as they're READ by the routine, you can more easily see how the music is constructed. Clearly, this is a much better way of composing than just building the data as you go. However, you still need to be familiar with the pitch value table and the multiples of your timing variables if you are going to take full

advantage of a chart like this. To make this table more usable and easier to follow, you can add the names of the notes and the timing count(s) to the actual data needed for the program. Now the table is easier to understand.

Table 2-2. Adding Note Values and Timing Counts

Name of Note	Lower Tone Value	Upper Tone Value	Note Length Value	Pause Length Value	Timing Count Value
C	195	16	32	32	1
C	195	16	32	32	1
D	208	18	32	32	1
E	30	21	32	32	1
C	195	16	32	32	1
E	30	21	32	32	1
D	208	18	32	32	1
G	142	12	32	32	1
C	196	16	32	32	1
C	195	16	32	32	1
D	208	18	32	32	1
E	30	21	32	32	1
C	195	16	128	32	4
B	210	15	128	32	4
C	195	16	32	32	1
C	195	16	32	32	1
D	208	18	32	32	1
E	30	21	32	32	1
F	95	22	32	32	1
E	30	21	32	32	1
D	208	18	32	32	1
C	195	16	32	32	1
B	210	15	32	32	1
G	142	12	32	32	1
A	24	14	32	32	1
B	210	15	32	32	1
C	195	16	128	32	4
C	195	16	128	32	4

A Simple Sound Editor

Once you've worked with a sound table like the examples above for some time, you may decide, although it does afford you a great deal of flexibility in writing music, that it's a bit cumbersome. What you want and need is something more

akin to a *real* instrument. A sound editor can give you that capability.

A good sound editor should make it easy to enter notes. Although making the 64 play similar to something like a piano, for instance, is difficult, you should be able to press keys and hear the note you've selected. The pitch values for the low and high frequency control registers should display as you choose notes, and you should even be able to alter the length of the note's sustain volume, just as you did in earlier routines with FOR-NEXT loops. Entering pauses between notes should be simple. And you should be able to save your creations, as well as load previously composed pieces for listening and possible revision. The simple sound editor that follows, although not as complex as something you might buy from a commercial software house, does meet all those criteria. It provides you with the tools that make the Commodore 64 as easy to use as an instrument, and allows you to store your music, play it back, and even change its tune, tempo (speed), and overall characteristic sounds. Once you've typed it in and saved it to tape or disk, you'll be able to avoid much of the tedium of creating sound on the 64.

Program 2-6. One-Voice Sound Editor

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1 REM ** INITIALIZE SOUND REGISTERS ***      :rem 26
2 REM                                          :rem 22
3 FOR R=54272 TO 54296:POKER,Ø:NEXT:POKE 54275, 8:
   POKE 54278, 24Ø                               :rem 243
4 LE=1ØØ: I=1: BR$=" - ":PA=Ø:IC=Ø           :rem 63
5 DIM I(63), II$(63), K$(63), NO$(99), LO$(99), UP$(99)
   , LL$(99), PA$(99), LB$(99)                 :rem 82
6 K$(16)="C":K$(18)="D":K$(21)="E":K$(22)="F":K$(2
   5)="G":K$(28)="A":K$(31)="B"                :rem 1
7 DIM DD$(999)                                  :rem 185
8 REM ***** DEFINE MUSICAL KEYS *****     :rem 181
9 REM                                          :rem 29
1Ø C$="{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}N[EG]!
   [M]M{DOWN}{6 LEFT}[M] [EG]1[2 M]{DOWN}{6 LEFT}
   [M]N[3 T]M{DOWN}{5 LEFT}{5 T}"           :rem Ø
11 K$(33)="↑C":K$(37)="↑D":K$(42)="↑E":K$(44)="↑F"
   :K$(5Ø)="↑G":K$(56)="↑A"                 :rem 7Ø
12 K$(63)="↑B"                                  :rem 173
13 K$(17)="C#":K$(19)="D#":K$(23)="F#":K$(26)="G#"
   :K$(29)="G#":K$(35)="↑C#"                :rem 76

```

```

14 K$(39)="↑D#":K$(47)="↑F#":K$(53)="↑G#":K$(59)="
A#" :rem 124
20 D$="{2 DOWN}{8 RIGHT}{3 @}{DOWN}{4 LEFT}N[G]'
[M]M{DOWN}{6 LEFT}[M] [G]2[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 16
30 E$="{2 DOWN}{13 RIGHT}{3 @}{DOWN}{4 LEFT}N[G]#
[M]M{DOWN}{6 LEFT}[M] [G]3[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 160
40 F$="{2 DOWN}{18 RIGHT}{3 @}{DOWN}{4 LEFT}N[G]$
[M]M{DOWN}{6 LEFT}[M] [G]4[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 53
50 G$="{2 SPACES}"{2 DOWN}{22 RIGHT}" :rem 241
55 G$=G$+"{RIGHT}{3 @}{DOWN}{4 LEFT}N[G]M[M]
{DOWN}{6 LEFT}[M] [G]5[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 197
60 A$="{2 SPACES}"{2 DOWN}{27 RIGHT}" :rem 125
65 A$=A$+"{RIGHT}{3 @}{DOWN}{4 LEFT}N[G]&[M]M
{DOWN}{6 LEFT}[M] [G]6[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 188
70 B$="{2 SPACES}"{2 DOWN}{32 RIGHT}" :rem 16
75 B$=B$+"{RIGHT}{3 @}{DOWN}{4 LEFT}N[G]'[M]M
{DOWN}{6 LEFT}[M] [G]7[M] [G]{LEFT}{DOWN}
{6 LEFT}{RIGHT}N[3 T]M[G]{LEFT}{DOWN}{5 LEFT}
{5 T}" :rem 158
80 Q$="{8 DOWN}{5 RIGHT}{3 @}{DOWN}{4 LEFT}N[G]
[M]M{DOWN}{6 LEFT}[M] [G]Q[E2 M]{DOWN}{6 LEFT}
[M]N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 180
90 W$="{8 DOWN}{10 RIGHT}{3 @}{DOWN}{4 LEFT}N[G]
[M]M{DOWN}{6 LEFT}[M] [G]W[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 200
100 EE$="{8 DOWN}{14 RIGHT}" :rem 222
105 EE$=EE$+"{RIGHT}{3 @}{DOWN}{4 LEFT}N[G] [M]M
{DOWN}{6 LEFT}[M] [G]E[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 98
110 R$="{8 DOWN}{19 RIGHT}" :rem 56
115 R$=R$+"{RIGHT}{3 @}{DOWN}{4 LEFT}N[G] [M]M
{DOWN}{6 LEFT}[M] [G]R[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 0
120 T$="{8 DOWN}{24 RIGHT}" :rem 204
125 T$=T$+"{RIGHT}{3 @}{DOWN}{4 LEFT}N[G] [M]M
{DOWN}{6 LEFT}[M] [G]T[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 7
130 Y$="{8 DOWN}{29 RIGHT}" :rem 99
135 Y$=Y$+"{RIGHT}{3 @}{DOWN}{4 LEFT}N[G] [M]M
{DOWN}{6 LEFT}[M] [G]Y[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[3 T]M{DOWN}{5 LEFT}{5 T}" :rem 23
140 U$="{8 DOWN}{34 RIGHT}" :rem 241
145 U$=U$+"{RIGHT}{3 @}{DOWN}{4 LEFT}N[G] [M]M
{DOWN}{6 LEFT}[M] [G]U[M] [G]{LEFT}{DOWN}
{6 LEFT}{RIGHT}N[3 T]M[G]{LEFT}{DOWN}{5 LEFT}
{5 T}" :rem 233

```

```

150 SP$="{HOME}{12 DOWN}" :rem 189
160 A$=SP$+A$:B$=SP$+B$:C$=SP$+C$:D$=SP$+D$:E$=SP$
+E$:F$=SP$+F$:G$=SP$+G$ :rem 236
170 Q$=SP$+Q$:W$=SP$+W$:EE$=SP$+EE$:R$=SP$+R$:T$=S
P$+T$:Y$=SP$+Y$:U$=SP$+U$ :rem 65
171 GR$="{HOME}{RVS} NOTE {OFF}_ LOWER _ UPPER _ L
EN - PAUSE - #" :rem 170
172 GR$=GR$+"{HOME}{DOWN}*****+*****+*****+**
***+*****+****" :rem 107
173 LL$="{6 SPACES}_-{7 SPACES}_-{7 SPACES}_-
{5 SPACES}_-{7 SPACES}_-{3 SPACES}" :rem 41
200 REM :rem 118
203 REM ***** DEFINE REVERSE KEYS ***** :rem 32
205 REM :rem 123
210 XC$="{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}{RVS}
£[G]1[M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS} [G]1
[2 M]{DOWN}{6 LEFT}{OFF}[M]{RVS}_N[3 T]_M{OFF}
{DOWN}{5 LEFT}[5 T]" :rem 99
220 XD$="{2 DOWN}{8 RIGHT}{3 @}{DOWN}{4 LEFT}{RVS}
£[G]'[M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS} [G]2
[2 M]{DOWN}{6 LEFT}{RIGHT}_N[3 T]_M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 207
230 XE$="{2 DOWN}{13 RIGHT}{3 @}{DOWN}{4 LEFT}
{RVS}£[G]#[M][*]{DOWN}{6 LEFT}{RIGHT} [G]3
[2 M]{DOWN}{6 LEFT}{RIGHT}_N[3 T]_M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 15
240 XF$="{2 SPACES}"{2 DOWN}{5 RIGHT}" :rem 140
245 XF$=XF$+"{13 RIGHT}{3 @}{DOWN}{4 LEFT}{RVS}£
[G]$_[M][*]{DOWN}{6 LEFT}{RIGHT} [G]4[2 M]
{DOWN}{6 LEFT}{RIGHT}_N[3 T]_M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 227
250 XG$="{2 SPACES}"{2 DOWN}{10 RIGHT}" :rem 31
255 XG$=XG$+"{13 RIGHT}{3 @}{DOWN}{4 LEFT}{RVS}£
[G]$_[M][*]{DOWN}{6 LEFT}{RIGHT} [G]5[2 M]
{DOWN}{6 LEFT}{RIGHT}_N[3 T]_M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 232
260 XA$="{2 SPACES}"{2 DOWN}{15 RIGHT}" :rem 171
265 XA$=XA$+"{13 RIGHT}{3 @}{DOWN}{4 LEFT}{RVS}£
[G]&[M][*]{DOWN}{6 LEFT}{RIGHT} [G]6[2 M]
{DOWN}{6 LEFT}{RIGHT}_N[3 T]_M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 223
270 XB$="{2 SPACES}"{2 DOWN}{20 RIGHT}" :rem 62
275 XB$=XB$+"{13 RIGHT}{3 @}{DOWN}{4 LEFT}{RVS}£
[G]'[M][*]{DOWN}{6 LEFT}{RIGHT} [G]7[2 M]
{DOWN}{6 LEFT}{RIGHT}_N[3 T]_M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 228
290 XC$=SP$+XC$:XD$=SP$+XD$:XE$=SP$+XE$:XF$=SP$+XF
$:XG$=SP$+XG$:XA$=SP$+XA$ :rem 219

```

```

295 XB$=SP$+XB$                               :rem 75
310 XQ$="{6 DOWN}{2 RIGHT}"                    :rem 130
315 XQ$=XQ$+"{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
      {RVS}{L}[G] [M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS}
      [G]Q[2 M]{DOWN}{6 LEFT}{OFF}[M]{RVS}N[3 T]M
      {OFF}{DOWN}{5 LEFT}{5 T}"              :rem 110
320 XW$="{6 DOWN}{7 RIGHT}"                    :rem 26
325 XW$=XW$+"{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
      {RVS}{L}[G] [M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS}
      [G]W[2 M]{DOWN}{6 LEFT}{RIGHT}N[3 T]M{OFF}
      {DOWN}{5 LEFT}{5 T}"                  :rem 83
330 ZE$="{6 DOWN}{12 RIGHT}"                   :rem 156
335 ZE$=ZE$+"{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
      {RVS}{L}[G] [M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS}
      [G]E[2 M]{DOWN}{6 LEFT}{RIGHT}N[3 T]M{OFF}
      {DOWN}{5 LEFT}{5 T}"                  :rem 34
340 XR$="{6 DOWN}{17 RIGHT}"                   :rem 57
345 XR$=XR$+"{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
      {RVS}{L}[G] [M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS}
      [G]R[2 M]{DOWN}{6 LEFT}{RIGHT}N[3 T]M{OFF}
      {DOWN}{5 LEFT}{5 T}"                  :rem 70
350 XT$="{6 DOWN}{22 RIGHT}"                   :rem 205
355 XT$=XT$+"{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
      {RVS}{L}[G] [M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS}
      [G]T[2 M]{DOWN}{6 LEFT}{RIGHT}N[3 T]M{OFF}
      {DOWN}{5 LEFT}{5 T}"                  :rem 77
360 XY$="{6 DOWN}{27 RIGHT}"                   :rem 100
365 XY$=XY$+"{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
      {RVS}{L}[G] [M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS}
      [G]Y[2 M]{DOWN}{6 LEFT}{RIGHT}N[3 T]M{OFF}
      {DOWN}{5 LEFT}{5 T}"                  :rem 93
370 XU$="{6 DOWN}{32 RIGHT}"                   :rem 242
375 XU$=XU$+"{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
      {RVS}{L}[G] [M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS}
      [G]U[2 M]{DOWN}{6 LEFT}{RIGHT}N[3 T]M{OFF}
      {DOWN}{5 LEFT}{5 T}"                  :rem 82
390 XQ$=SP$+XQ$:XW$=SP$+XW$:ZE$=SP$+ZE$:XR$=SP$+XR
      $:XT$=SP$+XT$:XY$=SP$+XY$             :rem 132
395 XU$=SP$+XU$                               :rem 114
400 GOSUB 6400                                  :rem 222
480 REM                                         :rem 128
490 REM ***** GET KEY ROUTINE *****       :rem 20
495 REM                                         :rem 134
500 GET WW$: IFWW$="" THEN 500                 :rem 39
502 IF WW$="" THEN 4710                        :rem 116
503 IF WW$="{F3}" THEN 5010                   :rem 245
504 IF WW$="{F5}" THEN 5510                   :rem 252
505 IF WW$="{RIGHT}" OR WW$="{LEFT}" THEN 4000
                                             :rem 29

```



```

506 IF WW$="{F1}" THEN 4410 :rem 250
507 IF WW$="{UP}" OR WW$="{DOWN}" THEN 4510:rem 13
509 IF ASC(WW$)<40 OR ASC(WW$)>200 THEN 810
:rem 250
510 IF WW$<"1" THEN 520 :rem 172
512 PRINTXC$;:L=195:H=16:GOSUB 1000 :rem 34
515 PRINTC$;:GOTO500 :rem 210
520 IF WW$<"2" THEN 530 :rem 175
522 PRINTXD$;:L=208:H=18:GOSUB 1000 :rem 33
525 PRINTD$;:GOTO500 :rem 212
530 IF WW$<"3" THEN 540 :rem 178
532 PRINTXE$;:L=30:H=21:GOSUB 1000 :rem 230
535 PRINTE$;:GOTO500 :rem 214
540 IF WW$<"4" THEN 550 :rem 181
542 PRINTXF$;:L=95:H=22:GOSUB 1000 :rem 244
545 PRINTF$;:GOTO500 :rem 216
550 IF WW$<"5" THEN 560 :rem 184
552 PRINTXG$;:L=29:H=25:GOSUB 1000 :rem 246
555 PRINTG$;:GOTO500 :rem 218
560 IF WW$<"6" THEN 570 :rem 187
562 PRINTXA$;:L=48:H=28:GOSUB 1000 :rem 245
565 PRINTA$;:GOTO500 :rem 213
570 IF WW$<"7" THEN 580 :rem 190
572 PRINTXB$;:L=164:H=31:GOSUB 1000 :rem 32
575 PRINTB$;:GOTO500 :rem 215
580 IF WW$<"Q" THEN 590 :rem 218
582 PRINTXQ$;:L=134:H=33:GOSUB 1000 :rem 47
585 PRINTQ$;:GOTO500 :rem 231
590 IF WW$<"W" THEN 600 :rem 217
592 PRINTXW$;:L=161:H=37:GOSUB 1000 :rem 58
595 PRINTW$;:GOTO500 :rem 238
600 IF WW$<"E" THEN 610 :rem 192
602 PRINTZE$;:L=60:H=42:GOSUB 1000 :rem 236
605 PRINTEE$;:GOTO500 :rem 25
610 IF WW$<"R" THEN 620 :rem 207
612 PRINTXR$;:L=191:H=44:GOSUB 1000 :rem 47
615 PRINTR$;:GOTO500 :rem 226
620 IF WW$<"T" THEN 630 :rem 211
622 PRINTXT$;:L=58:H=50:GOSUB 1000 :rem 1
625 PRINTT$;:GOTO500 :rem 229
630 IF WW$<"Y" THEN 640 :rem 218
632 PRINTXY$;:L=97:H=56:GOSUB 1000 :rem 16
635 PRINTY$;:GOTO500 :rem 235
640 IF WW$<"U" THEN 500 :rem 210
642 PRINTXU$;:L=72:H=63:GOSUB 1000 :rem 4
645 PRINTU$;:GOTO500 :rem 232
700 GOTO 500 :rem 101
800 REM :rem 124
805 REM **** SHARP VALUES **** :rem 31

```

```

807 REM :rem 131
810 IF WW$<>"!" THEN 820 :rem 162
812 PRINTXC$;:L=194:H=17:GOSUB 1000 :rem 37
815 PRINTC$;:GOTO500 :rem 213
820 IF WW$<>CHR$(34) THEN 830 :rem 248
822 PRINTXD$;:L=238:H=19:GOSUB 1000 :rem 40
825 PRINTD$;:GOTO500 :rem 215
830 IF WW$<>"#" THEN 840 :rem 168
832 PRINTXE$;:L=30:H=21:GOSUB 1000 :rem 233
835 PRINTE$;:GOTO500 :rem 217
840 IF WW$<>"$" THEN 850 :rem 171
842 PRINTXF$;:L=180:H=23:GOSUB 1000 :rem 35
845 PRINTF$;:GOTO500 :rem 219
850 IF WW$<>"%" THEN 860 :rem 174
852 PRINTXG$;:L=155:H=26:GOSUB 1000 :rem 42
855 PRINTG$;:GOTO500 :rem 221
860 IF WW$<>"&" THEN 870 :rem 177
862 PRINTXA$;:L=221:H=29:GOSUB 1000 :rem 34
865 PRINTA$;:GOTO500 :rem 216
870 IF WW$<>"'" THEN 880 :rem 180
872 PRINTXB$;:L=164:H=31:GOSUB 1000 :rem 35
875 PRINTB$;:GOTO500 :rem 218
880 IF WW$<>"Q" THEN 890 :rem 96
882 PRINTXQ$;:L=132:H=35:GOSUB 1000 :rem 50
885 PRINTQ$;:GOTO500 :rem 234
890 IF WW$<>"W" THEN 900 :rem 95
892 PRINTXW$;:L=221:H=39:GOSUB 1000 :rem 60
895 PRINTW$;:GOTO500 :rem 241
900 IF WW$<>"E" THEN 910 :rem 70
902 PRINTZE$;:L=60:H=42:GOSUB 1000 :rem 239
905 PRINTEE$;:GOTO500 :rem 28
910 IF WW$<>"R" THEN 920 :rem 85
912 PRINTXR$;:L=104:H=47:GOSUB 1000 :rem 47
915 PRINTR$;:GOTO500 :rem 229
920 IF WW$<>"T" THEN 930 :rem 89
922 PRINTXT$;:L=55:H=53:GOSUB 1000 :rem 4
925 PRINTT$;:GOTO500 :rem 232
930 IF WW$<>"Y" THEN 940 :rem 96
932 PRINTXY$;:L=187:H=59:GOSUB 1000 :rem 70
935 PRINTY$;:GOTO500 :rem 238
940 IF WW$<>"U" THEN 500 :rem 85
942 PRINTXU$;:L=72:H=63:GOSUB 1000 :rem 7
945 PRINTU$;:GOTO500 :rem 235
950 REM :rem 130
960 REM **** PLAY NOTE **** :rem 105
970 REM :rem 132
1000 POKE 54272,L: POKE 54273, H :rem 78
1010 POKE 54276, 65: POKE 54296, 15 :rem 149
1020 FOR R=0 TO LE:NEXT :rem 22

```

```

1030 POKE 54276,0: POKE 54296, 0           :rem 38
1040 FOR R=0 TO PA:NEXT                     :rem 24
2000 REM                                    :rem 166
2002 REM ** STORE TONE/TIMING VALUES **    :rem 218
2004 REM                                    :rem 170
2005 NOS(IC)=RIGHT$(" {4 SPACES}" +K$(H),5):LO$(IC)=
      RIGHT$(" {4 SPACES}" +STR$(L)+" ",5)  :rem 108
2007 UP$(IC)=RIGHT$(" {3 SPACES}" +STR$(H)+" ",5)
                                             :rem 148
2010 LL$(IC)=RIGHT$(" {2 SPACES}" +STR$(LE),3):PA$(I
      C)=RIGHT$(" {2 SPACES}" +STR$(PA)+" {2 SPACES}",
      5)                                     :rem 147
2020 LB$(IC)=RIGHT$(" " +STR$(INT((PA+LE)/25)),2)
                                             :rem 46
3000 REM                                    :rem 167
3002 REM **** DISPLAY VALUES ****          :rem 223
3004 DD$(SK)=NOS(IC)+BR$+LO$(IC)           :rem 231
3005 DD$(SK)=DD$(SK)+BR$+UP$(IC)+BR$+LL$(IC)+BR$+P
      A$(IC)+BR$+LB$(IC)                   :rem 211
3006 DD$(SK)=" [7]" +RIGHT$(STR$(IC),2)+" {WHT}" +RIGH
      T$(DD$(SK),38)                       :rem 185
3007 UP$(IC)=RIGHT$(" {3 SPACES}" +STR$(H)+" ",5)
                                             :rem 149
3008 PRINT"{HOME}{2 DOWN}";                :rem 12
3010 IF SK<9 THEN FOR R=0 TO SK:PRINTDD$(R);:NEXT:
      GOTO 3020                              :rem 126
3015 FOR R=SK-9 TO SK:PRINTDD$(R);:NEXT    :rem 78
3020 IC=IC+1: SK=SK+1                      :rem 133
3030 RETURN                                 :rem 166
4000 REM                                    :rem 168
4002 REM **** SWITCH MODES ****            :rem 68
4003 REM                                    :rem 171
4005 IF I=-1 THEN 4010                     :rem 48
4007 PRINT"{HOME} NOTE {17 RIGHT}{RVS}";RIGHT$("
      {5 SPACES}" +STR$(LE),5);            :rem 28
4008 PRINT"{OFF}";:I=I*-1:GOTO500         :rem 3
4010 PRINT"{HOME}{RVS} NOTE {OFF}{17 RIGHT} LEN ";
      : I=I*-1: GOTO 500                   :rem 35
4400 REM                                    :rem 172
4402 REM **** PLAY BACK MUSIC ****         :rem 198
4405 REM                                    :rem 177
4410 IF SK=0 THEN GOTO 500                 :rem 96
4415 FOR Q=0 TO IC-1                      :rem 0
4420 H=VAL(UP$(Q)): L=VAL(LO$(Q)): LE=VAL(LL$(Q)):
      PA=VAL(PA$(Q))                       :rem 115
4430 POKE 54272,L: POKE 54273, H          :rem 88
4440 POKE 54276, 65: POKE 54296, 15      :rem 159
4450 FOR R=0 TO LE:NEXT                   :rem 32
4460 POKE 54276,0: POKE 54296, 0        :rem 48
4470 FOR R=0 TO PA:NEXT                   :rem 34
4480 NEXT:GOTO 500                        :rem 23

```

```

4500 REM :rem 173
4502 REM **** CURSOR UP/DOWN **** :rem 233
4505 REM :rem 178
4507 REM --- SCROLL WINDOW --- :rem 105
4508 REM :rem 181
4510 IF I=-1 THEN 4560 :rem 59
4515 PRINT"{HOME}{DOWN}"; :rem 255
4517 IF SK>10 THEN 4530 :rem 152
4520 IFSK>-1 THEN SK=SK-1:IC=IC-1:FOR R=0 TO SK:PRIN
T"{DOWN}";:NEXT:PRINT LL$: :rem 54
4522 IF SK<0 THEN SK=0: IF IC<0 THEN IC=0 :rem 137
4525 GOTO 500 :rem 158
4530 SK=SK-1:IC=IC-1:PRINT"{DOWN}";:FOR R=SK-10 TO
SK-1:PRINT DD$(R);:NEXT:GOTO 500 :rem 52
4540 GOTO 500 :rem 155
4550 REM :rem 178
4555 REM --- CHANGE LENGTH OF TONE --- :rem 248
4557 REM :rem 185
4560 IF WW$="{DOWN}" THEN 4580 :rem 194
4570 IF LE<975 THEN LE = LE+25 :rem 241
4575 PRINT"{HOME} NOTE {17 RIGHT}{RVS}";RIGHT$("
{5 SPACES}"+STR$(LE),5);"{OFF}";:GOTO 500
:rem 63
4580 IF LE>=25 THEN LE = LE-25 :rem 245
4585 PRINT"{HOME} NOTE {17 RIGHT}{RVS}";RIGHT$("
{5 SPACES}"+STR$(LE),5);"{OFF}";:GOTO 500
:rem 64
4700 REM :rem 175
4702 REM **** ENTER A PAUSE **** :rem 62
4705 REM :rem 180
4710 H=0:L=0:PA=LE:LE=0:GOSUB2000 :rem 193
4720 LE=PA:PA=0:GOTO 500 :rem 108
5000 REM :rem 169
5010 REM **** SAVE ROUTINE **** :rem 79
5020 REM :rem 171
5025 PRINT "{CLR}" :rem 48
5030 INPUT "{HOME}{5 DOWN}{2 SPACES}SAVE ON DISK O
R TAPE (D/T)"; ME$ :rem 207
5040 IF ME$="C" THEN GOSUB 6430:GOTO 500 :rem 86
5050 IF ME$<>"D" AND ME$<>"T" THEN 5060 :rem 166
5055 GOTO 5210 :rem 208
5060 PRINT"{DOWN}{6 SPACES}PLEASE ENTER 'D' FOR DI
SK" :rem 137
5070 PRINT"{19 SPACES}'T' FOR TAPE" :rem 80
5080 PRINT"{15 SPACES}OR{2 SPACES}'C' TO CANCEL"
:rem 25
5090 PRINT" SAVE AND RETURN TO THE MAIN PROGRAM
{DOWN}" :rem 83
5100 GOTO 5030 :rem 199
5200 REM :rem 171

```

```

5202 REM **** ENTER NAME OF SONG ****           :rem 104
5205 REM                                         :rem 176
5210 PRINT"{HOME}{13 DOWN}{14 RIGHT}-----"
--"                                           :rem 239
5220 PRINT"{DOWN}(MAX 16 LETTERS)?"           :rem 171
5230 INPUT "{4 UP}NAME OF SONG";NM$          :rem 201
5240 NM$=LEFT$(NM$,16)                         :rem 185
5250 PRINT"{HOME}{12 DOWN}{14 RIGHT}{20 SPACES}"
                                           :rem 18
5260 PRINT"{HOME}{12 DOWN}{14 RIGHT}";NM$;"
{10 SPACES}"                                 :rem 140
5270 INPUT"{HOME}{21 DOWN}IS THIS CORRECT (Y/N)";C
R$                                           :rem 27
5280 IF CR$="Y" THEN 5400                       :rem 233
5285 IF CR$="C" THEN GOSUB 6430: IC=0:SK=0: GOTO 5
00                                           :rem 220
5290 PRINT"{HOME}{12 DOWN}{14 RIGHT}{22 SPACES}"
                                           :rem 22
5300 PRINT"{14 SPACES}"                       :rem 153
5310 GOTO 5210                                 :rem 202
5400 REM                                         :rem 173
5402 REM **** SAVE MUSIC ****                 :rem 175
5405 REM --- DISK ---                          :rem 235
5410 IF ME$="T" THEN 5460                      :rem 226
5420 OPEN 1,8,4,"@:"+NM$+"",W"               :rem 184
5425 FOR Q=0 TO IC-1                          :rem 2
5427 H=VAL(UP$(Q)): L=VAL(LO$(Q)): LE=VAL(LL$(Q)):
PA=VAL(PA$(Q))                               :rem 123
5430 PRINT#1,H:PRINT#1,L:PRINT#1,LE:PRINT#1,PA: NE
XT                                           :rem 221
5440 CLOSE 1                                  :rem 116
5450 GOSUB 6430                               :rem 27
5452 GOTO 500                                  :rem 158
5455 REM --- TAPE ---                          :rem 239
5460 OPEN 1,1,1,NM$: GOTO 5425                :rem 26
5505 REM                                         :rem 179
5510 REM **** LOAD ROUTINE ****               :rem 69
5520 REM                                         :rem 176
5525 PRINT "{CLR}"                             :rem 53
5530 INPUT "{HOME}{5 DOWN}{2 SPACES}LOAD FROM DISK
OR TAPE (D/T)"; ME$                          :rem 92
5540 IF ME$="C" THEN GOSUB 6430: GOTO 500     :rem 91
5550 IF ME$<>"D" AND ME$<>"T" THEN 5560      :rem 176
5555 GOTO 5710                                  :rem 218
5560 PRINT"{DOWN}{6 SPACES}PLEASE ENTER 'D' FOR DI
SK"                                           :rem 142
5570 PRINT"{19 SPACES}'T' FOR TAPE"          :rem 85
5580 PRINT"{15 SPACES}OR{2 SPACES}'C' TO CANCEL"
                                           :rem 30
5590 PRINT" LOAD AND RETURN TO THE MAIN PROGRAM
{DOWN}"                                       :rem 73

```

```

5700 REM                                     :rem 176
5702 REM **** ENTER NAME OF SONG ****      :rem 109
5705 REM                                     :rem 181
5710 PRINT"{HOME}{13 DOWN}{14 RIGHT}-----
--"                                         :rem 244
5720 PRINT"{DOWN}(MAX 16 LETTERS)?"       :rem 176
5730 INPUT "{4 UP}NAME OF SONG";NM$      :rem 206
5740 NM$=LEFT$(NM$,16)                     :rem 190
5750 PRINT"{HOME}{12 DOWN}{14 RIGHT}{20 SPACES}"
                                           :rem 23
5760 PRINT"{HOME}{12 DOWN}{14 RIGHT}";NM$;"
{10 SPACES}"                               :rem 145
5770 INPUT"{HOME}{21 DOWN}IS THIS CORRECT (Y/N)";C
R$                                         :rem 32
5780 IF CR$="Y" THEN 5900                  :rem 243
5785 IF CR$="C" THEN GOSUB 6430: GOTO 500 :rem 105
5790 PRINT"{HOME}{12 DOWN}{14 RIGHT}{22 SPACES}"
                                           :rem 27
5800 PRINT"{14 SPACES}"                   :rem 158
5810 GOTO 5710                             :rem 212
5900 REM                                     :rem 178
5902 REM **** LOAD MUSIC ****             :rem 165
5905 GOSUB 6430                             :rem 32
5910 IF ME$="T" THEN 5960                  :rem 236
5920 OPEN 1,8,4,NM$+"R":SK=0:IC=0         :rem 71
5925 BB=ST: IF BB<>0 THEN 5940             :rem 53
5930 INPUT#1,H:INPUT#1,L:INPUT#1,LE:INPUT#1,PA: GO
SUB 1000                                     :rem 240
5935 GOTO 5925                             :rem 228
5940 CLOSE 1                               :rem 121
5950 GOTO 500                              :rem 161
5955 REM --- TAPE ---                     :rem 244
5960 OPEN 1,1,0,NM$:SK=0:IC=0:GOTO 5925  :rem 155
6400 REM                                     :rem 174
6410 REM **** PRINT DISPLAY SCREEN ***** :rem 140
6420 REM                                     :rem 176
6430 PRINT"{WHT}{CLR}{12 DOWN}[40 U]{HOME}";
                                           :rem 16
6440 PRINT"{HOME}{2 DOWN}";:FOR O=0 TO 9:PRINTLL$;
:NEXT                                       :rem 255
6450 PRINT"{HOME}";GR$;A$;B$;C$;D$;E$;F$;G$;Q$;W$;
EE$;R$;T$;Y$;U$;                         :rem 122
6455 PRINT"{HOME}{13 DOWN}{4 RIGHT}";     :rem 68
6460 PRINT"C{4 SPACES}D{4 SPACES}E{4 SPACES}F
{4 SPACES}G{4 SPACES}A{3 SPACES}";       :rem 118
6465 PRINT"B{HOME}{19 DOWN}{5 RIGHT}↑C{3 SPACES}↑D
{3 SPACES}↑E";                             :rem 240
6470 PRINT"{3 SPACES}↑F{3 SPACES}↑G{3 SPACES}↑A
{3 SPACES}↑B{HOME}{3 DOWN}";             :rem 171
6502 RETURN                                :rem 173

```

Using the Editor

When you first run the Sound Editor, there will be a short pause while the program initializes itself and sets up the arrays for the sound and graphics it uses. When the program has finished initializing, it displays a screen that is divided into two parts. The upper half of the screen looks a great deal like Table 2-2, where you wrote down the POKE values of various notes, the note length values, and the pause length values. The lower half of the screen shows two rows of keys. The top row displays the keys 1-7 (near the top left corner of the keyboard), and the lower row contains the keys Q-U (directly below the previous row).

Notes of the scale. The notes of the scale are labeled above the keys. The top row is one octave lower than the bottom row. To show the difference between the two octaves, the higher keys (bottom row) are preceded by an up arrow (↑). These arrows also appear in the table in the upper half of the screen as you enter those notes so you'll be able to determine what notes are being played. To hear a note play, press the corresponding key.

Using the table. The table in the program is essentially the same table you used when you were writing the different values by hand. The biggest difference is that each of the values in this table is entered automatically each time you press a key on the musical keyboard. To help you use the table more effectively, let's review each of the function headings.

NOTE. When the program is first turned on, the NOTE heading is highlighted. This indicates that the computer is waiting for you to enter a note from the keyboard. This column will display the notes that you enter, and specify which octave it is using and the number of the entry. The note entry numbers begin with 0. You can only enter a maximum of 99 notes.

LOWER/UPPER. These columns show the actual POKE values being entered into the pitch control registers. Press the 1 key, which represents a C note. You'll see the values 195 and 16 listed in the LOWER and UPPER columns. By using this editor, you can avoid having to look up the POKE values for each note you want to play.

LEN. The LEN column displays the length of the note being played. The value displayed in this column is the value

you would use in the sustain delay loop. To change this value, press the Cursor Right/Left key at the lower right corner of the keyboard. This will highlight the LEN column and will display the current timing value.

By pressing the Cursor Up/Down key, you can increase or decrease this value in increments of 25. Pressing SHIFT and Cursor Up/Down increases the value; pressing the unSHIFTed Cursor Up/Down key decreases the length value. The highest possible value is 975, and the lowest is 0.

PAUSE. You can enter a pause as easily as you enter any note. Use the space bar to enter pauses between notes; the length of the pause will be the same as the present note length. To enter longer or shorter pauses, change the LEN value as you did earlier.

When you enter pauses, you'll see that a space is displayed in the note column, and zeros are displayed in the LOWER and UPPER columns.

#. The # column shows the relative length of the notes entered. This is based on a unit of 25 as the basic increment of time. In other words, any note that has a length (LEN) of 25 will have a relative length of 1, a note that has a length (LEN) of 100 has a relative length of 4, and so on. This can help you determine the length to use in your notes if you are writing songs that have complex timing.

Playing music back. To play back the notes you have entered at any point, all you need to do is press the f1 key. This will play all of the notes you have entered so far. You can use this function to check your timing, since the notes are played using the same timing and pauses you've entered.

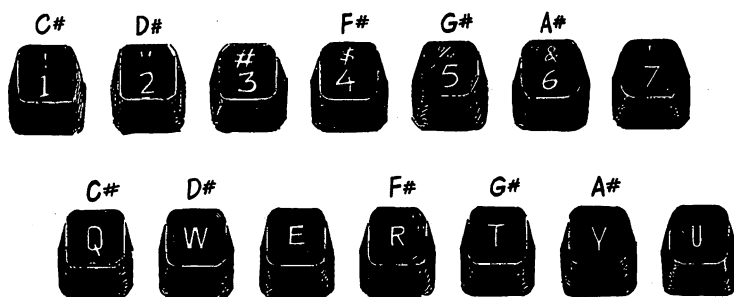
Modifying the music. To change any note that you've entered, you need to delete the old note and enter the new one as you would enter any new note. The display table shows you the last ten notes entered. To change a note, use the Cursor Up/Down key (either SHIFTed or unSHIFTed) when the NOTE column is highlighted. By pressing the key the correct number of times, you can position the note you want to delete at the bottom of the table in the upper half of the screen. *Unfortunately, this will delete all of the notes from that entry number on up.* What you're doing, in effect, is deleting all the notes to that point. Once you press the Cursor Up/Down key as a delete, the note is gone for good. If you keep

pressing the key, you can erase your entire tune. Keep that in mind as you use it.

If you are beyond the tenth note in your song, the table will scroll backwards until it gets to the note you want to change. Remember that as the table scrolls backwards, what is really happening is that the notes are being erased.

Sharps and flats. In addition to the 14 notes obtained by pressing the appropriate keys, there are 10 other notes (called sharps or flats) that can be accessed by pressing the SHIFT key *and* a note key together. Figure 2-1 shows the positions of the sharps on the keyboard.

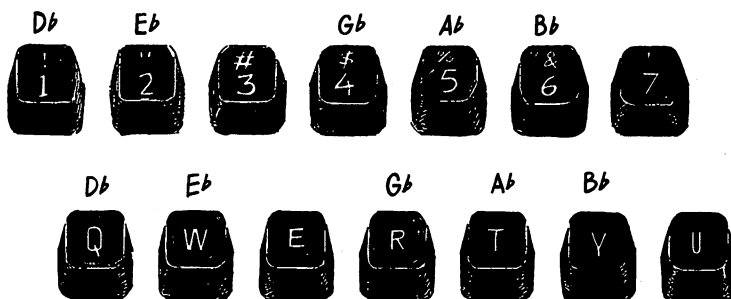
Figure 2-1. Sharp Key Placement



Notice that the sharps are C#, D#, F#, G# and A#. The notes E# and B# are not listed on this musical scale. This is because a sharp note is one that is half a step higher than the note it follows, and a flat is one-half step lower. Since the note F is half a step above E and the note B is half a step below C, there are no corresponding sharps or flats to display.

Figure 2-2 shows the positions of the flats available on the keyboard. Comparing this figure with Figure 2-1, you'll notice that the sharps and flats are accessed with the same keys. Remember that a sharp is a half step up, while a flat is a half step down. Thus Db is the same note as C#. When you select a flat, it will show in the table on the screen as a sharp. To determine the flats, refer to the figure below.

Figure 2-2. Flat Key Positions



SAVEing music. To save a song that you have written on the sound editor, press the f3 key. You will then be asked if you wish to store the song on tape or disk. To save the song on tape, press T and hit RETURN. To store it on disk, press D, then press RETURN. The program will then ask you for the name you wish to store the song as. The name must be less than 16 characters long.

After the song has been stored, the computer will return to the edit screen and will be ready to accept a new song.

LOADing songs. To load a song that you have previously saved to disk or tape, press the f5 key. This will display a screen that is similar to the SAVE screen. By following the directions, you'll be able to load a song from either disk or tape. After the tape or disk drive begins to run, the edit screen will reappear, and the stored notes will be played and displayed on the screen as they are read into memory.

A few hints. If you're entering a large number of notes with the sound editor, there will be times when the last note key you pressed will stay highlighted. If you press any of the note keys, or even one of the function keys, nothing will happen. The computer hasn't locked up. There's nothing wrong. The program is just taking some time to place all the values you've entered into the computer's memory. Wait patiently for the highlighted key to return to its normal display. Then you can continue to add more notes, or play the song, or save it to tape or disk. If you press a key while the last key is high-

lighted, that note will be read into memory once the program hands control back to you. It's a good idea to avoid pressing keys while the program is storing values; otherwise, it will read the keyboard buffer and place as many notes in the song as the number of times you hit that key. You'll have to delete those extra notes if that happens.

If you want to delete a note or notes, but still want a copy of the song as it exists, be sure to save it to tape or disk. Then when the screen returns, you can delete notes and listen to the new tune. If you don't like your alterations, at least you'll have a backup copy of the original version of the song on tape or disk. If you liked the original better, you can just load it and go on from there.

As the Sound Editor is listed, it uses the pulse waveform, with attack and sustain set to 0, the sustain/release control register set with a value of 240. You can change the waveform used by altering lines 1010 and 4440 in the program. Just change the value POKEd into location 54276. Adding an attack/decay value in line 3, where the program initializes several registers, would also change the sound of the notes as you enter them, as well as when you play back the entire song.

Harmony and Disharmony

To produce *harmony* in music, it's necessary to play two or more notes at the same time. Fortunately, the 64 has the ability to play up to three notes simultaneously. Each of these three voices has a similar set of control registers, giving you three groups of seven registers each.

Up to this point you've used only the first voice (control registers 54272-54278). The examples in the next section will use both voice 1 and voice 2 (registers 54279-54285).

Playing two notes at once. When two or more notes are played together, the tones blend and become more than they were individually. If the sound produced by the combined tones is pleasant, they are said to be in harmony. Program 2-7 demonstrates a harmonic sound.

Program 2-7. Harmonious Sounds

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
1 REM *** MUSICAL NOTE DATA ***           :rem 111
2 REM                                         :rem 22
3 DATA 195,16,195,16,400,24,14,24,14,200,47,11,47,
   11,800                                     :rem 175
```

```

4 DATA 24,14,95,22,700,195,16,30,21,700,95,22,208,
  18,1050 :rem 232
5 DATA 999,999,999,999,999 :rem 86
7 REM :rem 27
8 REM ** CLEAR AND INITIALIZE SOUND REGISTERS **
  :rem 91
9 FOR R=54272 TO 54296:POKER,0:POKE 54296,15
  :rem 124
10 REM --- SET UP VOICE #1 --- :rem 174
11 REM :rem 70
12 POKE 54274,8:POKE 54277,0:POKE 54278,240
  :rem 255
13 REM :rem 72
14 REM --- SET UP VOICE #2 --- :rem 179
15 REM :rem 74
16 POKE 54282,8:POKE 54284,0:POKE54285,240:rem 254
17 REM :rem 76
18 REM ** GET NOTE VALUES FROM TABLE ** :rem 119
19 REM :rem 78
21 READ L1,H1,L2,H2,T:IF H1=999THEN END :rem 161
22 REM :rem 72
23 REM -- PUT VALUES INTO REGISTERS -- :rem 184
24 REM :rem 74
25 POKE 54272,L1:POKE 54273,H1 :rem 86
26 POKE 54279,L2:POKE 54280,H2 :rem 94
27 REM :rem 77
28 REM ** TURN ON WAVEFORM ** :rem 67
29 REM :rem 79
30 POKE 54276,65:POKE 54283,65 :rem 55
45 FOR R=0 TO T:NEXT :rem 127
50 REM :rem 73
55 REM ** TURN OFF GATE BIT ** :rem 26
60 REM :rem 74
70 POKE 54276,64:POKE 54283,64 :rem 57
80 GOTO 21 :rem 4

```

Notice that both voice 1 and voice 2 are used in this program; the pitch values are read from the DATA statements in lines 3 through 5 in groups of four. The first value, L1, is the pitch value for the low frequency control register of voice 1, while H1 is the value for that voice's high control register. The third value, L2, is the number POKEd into voice 2's low frequency register, and H2 is used in the high register. Both voices use the pulse waveform and have the same sustain/release levels and rates. As with other example programs, it's not difficult to change the sound by altering the attack/decay rates (found in line 12 for voice 1 and line 16 for voice 2), or by using a different waveform.

A disharmonious sound is easily created by changing the pitch values found in the DATA statements of lines 3, 4, and 5. Using Program 2-7, change those lines to:

```
3 DATA 195,16,195,16,400,24,14,24,14,200,47,11,47,
  11,800
4 DATA 24,14,95,23,700,195,16,30,22,700,95,22,208,
  19,1050
5 DATA 999,999,999,999,999
```

After making these changes, run the program again. The first few notes sound the same as in Program 2-7, but you'll quickly hear the disharmony of the voices.

While you most often expect to find harmonic sounds in music, it's not unusual to find them in nonmusical sounds, such as a car horn. Enter and RUN Program 2-8 for an example of this.

Program 2-8. Car Horn

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
5 L1=195:H1=16:L2=30:H2=21 :rem 145
10 FOR R=54272 TO 54296:POKER,0:NEXT :rem 24
20 POKE 54296,15 :rem 45
30 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 0
35 POKE 54282,8:POKE 54284,0:POKE 54285,240 :rem 255

40 POKE 54272,L1: POKE 54273,H1 :rem 83
50 POKE 54279,L2: POKE 54280,H2 :rem 91
60 POKE 54276,65: POKE 54283,65 :rem 58
70 FOR M=0TO200: NEXT :rem 182
80 POKE 54276,64: POKE 54283,64 :rem 58
90 FOR M=0TO50: NEXT :rem 139
100 POKE 54276,65: POKE 54283,65 :rem 101
110 FOR M=0TO600: NEXT :rem 229
120 POKE 54276,64: POKE 54283,64 :rem 101
130 POKE 54296,0 :rem 41
```

Voices 1 and 2 are again used together in this program. Instead of READING pitch values from a table, as in the previous program, this routine simply initializes the pitch values in line 5. The various control registers are POKED with values, the pulse waveform is enabled in line 60, and a sustain delay loop executes in line 70. Then the gate bit is turned off in line 80, another delay loop executes as a pause between sounds, the gate bit is turned back on, and another sound plays, this

time a bit longer, as set by the sustain loop in line 110. Finally, the gate bit is turned off for good in line 120, which ends the routine.

Although some sound effects depend on harmony, most attention-getting sounds use disharmony instead. Program 2-9 demonstrates one such effect.

Program 2-9. Siren

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 L1=40:H1=41:L2=0:H2=46                                :rem 40
10 FOR R=54272 TO 54296:POKER,0:NEXT                    :rem 24
15 POKE 54296,15                                         :rem 49
20 POKE 54275,8:POKE 54277,0:POKE 54278,240
                                                           :rem 255
30 POKE 54282,8:POKE 54284,0:POKE 54285,240
                                                           :rem 250
35 POKE 54276,65:POKE 54283,65                          :rem 60
40 FOR T=0 TO 3                                          :rem 226
50 FOR M=0 TO 25                                         :rem 16
60 L1=L1+1:H1=H1+1:L2=L2+1:H2=H2+1                    :rem 84
70 POKE 54272,L1:POKE 54273,H1                         :rem 86
80 POKE 54279,L2:POKE 54280,H2                         :rem 94
90 NEXT                                                  :rem 168
100 FOR M=0 TO 25                                        :rem 60
110 L1=L1-1:H1=H1-1:L2=L2-1:H2=H2-1                    :rem 136
120 POKE 54272,L1:POKE 54273,H1                        :rem 130
130 POKE 54279,L2:POKE 54280,H2                        :rem 138
140 NEXT:NEXT                                           :rem 77
150 FOR R=0 TO 30                                        :rem 66
160 POKE 54296,15-(R/2)                                  :rem 147
170 POKE 54273,H1-(R/2):POKE 54380,H2-(R/2)
                                                           :rem 230
180 NEXT:POKE 54276,64:POKE 54283,64                    :rem 228

```

Much of this program you'll recognize from other examples. The main point of interest is the way FOR-NEXT loops have been used to make the sound first rise in pitch (lines 50-90) and then fall (lines 100-140). This pattern repeats itself four times, as noted by the FOR T=0 TO 3 statement in line 40. The siren sound effect is created by this rising and falling of pitch, as well as by the disharmony of the notes played by voices 1 and 2.

Beats, produced when two pitches close in frequency are played at the same time, are another way to use the multiple

voice capabilities of the 64. The speed of the beat (*beat frequency*) is determined by how close the two pitches are. The closer they are to one another, the slower the beat will be. As they drift farther and farther apart, the beat gets faster.

Program 2-10. Beats

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ** INITIALIZE SOUND REGISTERS ***           :rem 30
6 REM                                             :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT              :rem 24
20 POKE 54296,15                                  :rem 45
30 POKE 54272,0:POKE 54273,16                    :rem 243
40 POKE 54277,0:POKE 54278,170                  :rem 47
50 POKE 54284,0:POKE 54285,240                  :rem 42
60 POKE 54276,33:POKE 54283,33                  :rem 48
70 REM                                             :rem 75
80 REM ** SWEEP SCALES LOOP:VOICE #2 **          :rem 114
90 REM                                             :rem 77
100 FOR H2=14 TO 19                               :rem 161
110 FOR L2=0 TO 255                               :rem 163
120 POKE 54279,L2:POKE 54280,H2                 :rem 137
130 NEXT:NEXT                                     :rem 76
140 REM                                             :rem 121
150 REM ** TURN REGISTERS OFF **                 :rem 254
160 REM                                             :rem 123
170 FOR R=0 TO 1500:NEXT                          :rem 32
180 POKE 54276,32:POKE 54283,32                 :rem 97
190 POKE 54296,0                                  :rem 47
  
```

The pitch registers for voice 1 are set in line 30, but those of voice 2 gradually increase as the two FOR-NEXT loops in lines 100-130 execute. Voice 2 starts off at a lower pitch than voice 1, and as it climbs, eventually equals and finally rises above voice 1's frequency.

Chords, unlike harmony or beats, require three or more harmonizing notes played together. It's simplest to just show you an example:

Program 2-11. Organ Chords

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ** INITIALIZE SOUND REGISTERS ***           :rem 30
6 REM                                             :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT              :rem 24
11 POKE 54296,10                                  :rem 40
  
```

```

12 POKE 54277,0:POKE 54278,240           :rem 44
13 POKE 54284,0:POKE 54285,240           :rem 41
14 POKE 54291,0:POKE 54292,240           :rem 38
17 REM                                     :rem 76
18 REM ** TURN ON VOICE #1 **             :rem 165
19 REM                                     :rem 78
20 POKE 54272,210:POKE 54273,15:POKE 54276,33
                                           :rem 87
21 FOR G=0 TO 1500:NEXT                   :rem 224
22 REM                                     :rem 72
23 REM ** TURN ON VOICE #2 **             :rem 162
24 REM                                     :rem 74
25 POKE 54279,77:POKE 54280,13:POKE 54283,33
                                           :rem 56
26 FOR R=0 TO 1500:NEXT                   :rem 240
27 REM ** TURN ON VOICE #3 **             :rem 167
28 REM                                     :rem 78
29 POKE 54286,143:POKE 54287,10:POKE 54290,33
                                           :rem 102
61 REM                                     :rem 75
62 REM ** TURN REGISTERS OFF **           :rem 208
63 REM                                     :rem 77
65 FOR R=10 TO 15 STEP .006               :rem 75
70 POKE 54296,R:NEXT                       :rem 151
80 POKE 54276,32:POKE 54283,32:POKE 54290,32:POKE
54296,0                                     :rem 253

```

Each voice is turned on, one at a time, until all three are producing sound. This three-voice harmony creates the chord. As you can see from the program listing, once each voice is turned on, it continues to play until the gate bit is turned off in line 80. A sound will play indefinitely if the gate bit is never turned off. Take a look, too, at lines 65 and 70. Here the volume changes as the program nears its end. Although the volume control register was earlier set to 15, now it changes to 10, and then gradually increases as the FOR-NEXT loop in line 65 executes. It gives an interesting effect.

Actually, chords don't have to harmonize; if they do not, they produce a *discord*. Exchanging lines 20, 25, and 29 in Program 2-11 for the lines below will give you an idea of what a discord sounds like.

```

20 POKE 54272,175:POKE 54273,13:POKE 54276,33
25 POKE 54279,177:POKE 54280,12:POKE 54283,33
29 POKE 54286,43:POKE 54287,18:POKE 54290,33

```


A Multivoice Chord Editor

To produce multivoice music, you'll need to program two or three voices at the same time. The Sound Editor earlier in this chapter programmed only one of the three voices, and although you could simply use the data as it is displayed and POKE the other registers manually, the process can become tedious.

The "Chord Editor" below maintains the features of the Sound Editor, but allows you to program all three voices at once.

Program 2-12. Chord Editor

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1 REM ** INITIALIZE SOUND REGISTERS ***           :rem 26
2 REM                                             :rem 22
3 FOR R=54272 TO 54296:POKER,0:NEXT              :rem 24
4 LE=10: I=1: BR$=" - "                          :rem 164
5 DIM I(63), II$(63), K$(63), NO$(99), LO$(99), UP$(99)
   ,LL$(99), PA$(99), LB$(99)                    :rem 82
6 V(0)=54276: L(0)=54272: H(0)=54273: POKE 54275,
   {SPACE}8: POKE 54278, 240                      :rem 232
7 V(1)=54283: L(1)=54279: H(1)=54280: POKE 54282,
   {SPACE}8: POKE 54285, 240                      :rem 235
8 V(2)=54290: L(2)=54286: H(2)=54287: POKE 54289,
   {SPACE}8: POKE 54292, 240                      :rem 247
9 NO$(0)="{3 SPACES}":NO$(1)="{3 SPACES}":NO$(2)="{
   {3 SPACES}":LB$="{4 SPACES}":A3=0             :rem 129
10 FOR R=0 TO 2: LO$(R)="{3 SPACES}":UP$(R)="{
   {3 SPACES}": NEXT                             :rem 153
11 K$(16)="C":K$(18)="D":K$(21)="E":K$(22)="F":K$(
   25)="G":K$(28)="A":K$(31)="B"                 :rem 45
12 DIM DD$(999)                                  :rem 229
13 REM ***** DEFINE MUSICAL KEYS *****        :rem 225
14 REM                                             :rem 73
15 C$="{2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}N[G]I
   [M]M[DOWN]{6 LEFT}[M] [G]I[2 M]{DOWN}{6 LEFT}
   [M]N[3 T]M[DOWN]{5 LEFT}{5 T}"               :rem 5
16 K$(33)="↑C":K$(37)="↑D":K$(42)="↑E":K$(44)="↑F"
   :K$(50)="↑G":K$(56)="↑A"                     :rem 75
17 K$(63)="↑B"                                   :rem 178
18 K$(17)="C#":K$(19)="D#":K$(23)="F#":K$(26)="G#"
   :K$(29)="G#":K$(35)="↑C#"                   :rem 81
19 K$(39)="↑D#":K$(47)="↑F#":K$(53)="↑G#":K$(59)="
   A#"                                           :rem 129
20 D$="{2 DOWN}{8 RIGHT}{3 @}{DOWN}{4 LEFT}N[G]'
   [M]M[DOWN]{6 LEFT}[M] [G]2[2 M]{DOWN}{6 LEFT}
   {RIGHT}N[3 T]M[DOWN]{5 LEFT}{5 T}"          :rem 16

```

```

30 E$="{2 DOWN}{13 RIGHT}{E3 @}{DOWN}{4 LEFT}N[E]#
[M]M{DOWN}{6 LEFT}{M} [E]3[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 160
40 F$="{2 DOWN}{18 RIGHT}{E3 @}{DOWN}{4 LEFT}N[E]$
[M]M{DOWN}{6 LEFT}{M} [E]4[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 53
50 G$="{2 SPACES}"{2 DOWN}{22 RIGHT}" :rem 241
55 G$=G$+"{RIGHT}{E3 @}{DOWN}{4 LEFT}N[E]#&[M]M
{DOWN}{6 LEFT}{M} [E]5[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 197
60 A$="{2 SPACES}"{2 DOWN}{27 RIGHT}" :rem 125
65 A$=A$+"{RIGHT}{E3 @}{DOWN}{4 LEFT}N[E]&[M]M
{DOWN}{6 LEFT}{M} [E]6[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 188
70 B$="{2 SPACES}"{2 DOWN}{32 RIGHT}" :rem 16
75 B$=B$+"{RIGHT}{E3 @}{DOWN}{4 LEFT}N[E]G'[M]M
{DOWN}{6 LEFT}{M} [E]7[E] [E]{LEFT}{DOWN}
{6 LEFT}{RIGHT}N[E3 T]M[E]{LEFT}{DOWN}{5 LEFT}
{E5 T}" :rem 158
80 Q$="{8 DOWN}{5 RIGHT}{E3 @}{DOWN}{4 LEFT}N[E]
[M]M{DOWN}{6 LEFT}{M} [E]Q[E2 M]{DOWN}{6 LEFT}
[M]N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 180
90 W$="{8 DOWN}{10 RIGHT}{E3 @}{DOWN}{4 LEFT}N[E]
[M]M{DOWN}{6 LEFT}{M} [E]W[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 200
100 EE$="{8 DOWN}{14 RIGHT}" :rem 222
105 EE$=EE$+"{RIGHT}{E3 @}{DOWN}{4 LEFT}N[E] [M]M
{DOWN}{6 LEFT}{M} [E]E[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 98
110 R$="{8 DOWN}{19 RIGHT}" :rem 56
115 R$=R$+"{RIGHT}{E3 @}{DOWN}{4 LEFT}N[E] [M]M
{DOWN}{6 LEFT}{M} [E]R[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 0
120 T$="{8 DOWN}{24 RIGHT}" :rem 204
125 T$=T$+"{RIGHT}{E3 @}{DOWN}{4 LEFT}N[E] [M]M
{DOWN}{6 LEFT}{M} [E]T[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 7
130 Y$="{8 DOWN}{29 RIGHT}" :rem 99
135 Y$=Y$+"{RIGHT}{E3 @}{DOWN}{4 LEFT}N[E] [M]M
{DOWN}{6 LEFT}{M} [E]Y[E2 M]{DOWN}{6 LEFT}
{RIGHT}N[E3 T]M{DOWN}{5 LEFT}{E5 T}" :rem 23
140 U$="{8 DOWN}{34 RIGHT}" :rem 241
145 U$=U$+"{RIGHT}{E3 @}{DOWN}{4 LEFT}N[E] [M]M
{DOWN}{6 LEFT}{M} [E]U[M] [E]{LEFT}{DOWN}
{6 LEFT}{RIGHT}N[E3 T]M[E]{LEFT}{DOWN}{5 LEFT}
{E5 T}" :rem 233
150 SP$="{HOME}{12 DOWN}" :rem 189
160 A$=SP$+A$:B$=SP$+B$:C$=SP$+C$:D$=SP$+D$:E$=SP$
+E$:F$=SP$+F$:G$=SP$+G$ :rem 236

```

```

170 Q$=SP$+Q$:W$=SP$+W$:EE$=SP$+EE$:R$=SP$+R$:T$=S
P$+T$:Y$=SP$+Y$:U$=SP$+U$ :rem 65
171 GR$="{HOME}{RVS}{4 SPACES}NOTE{3 SPACES}{OFF}-
[7]#1{WHT}LO/UP-[7]#2{WHT}LO/UP-[7]#3{WHT}LO/U
P-LEN" :rem 64
172 GR$=GR$+"{HOME}{DOWN}*****+*****+*****
**+*****+*****" :rem 80
173 LL$="{11 SPACES}-[7 SPACES]-[7 SPACES]-
{7 SPACES}-[4 SPACES]" :rem 76
200 REM :rem 118
203 REM ***** DEFINE REVERSE KEYS ***** :rem 32
205 REM :rem 123
210 XC$="{2 DOWN}{3 RIGHT}[3 @]{DOWN}{4 LEFT}{RVS}
[G][M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS} [G]1
[2 M]{DOWN}{6 LEFT}{OFF}[M]{RVS}[N][3 T]M{OFF}
{DOWN}{5 LEFT}[5 T]" :rem 99
220 XD$="{2 DOWN}{8 RIGHT}[3 @]{DOWN}{4 LEFT}{RVS}
[G]'[M][*]{DOWN}{6 LEFT}{OFF}[M]{RVS} [G]2
[2 M]{DOWN}{6 LEFT}{RIGHT}[N][3 T]M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 207
230 XE$="{2 DOWN}{13 RIGHT}[3 @]{DOWN}{4 LEFT}
{RVS}[G][M][*]{DOWN}{6 LEFT}{RIGHT} [G]3
[2 M]{DOWN}{6 LEFT}{RIGHT}[N][3 T]M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 15
240 XF$="{2 SPACES}"{2 DOWN}{5 RIGHT}" :rem 140
245 XF$=XF$+"{13 RIGHT}[3 @]{DOWN}{4 LEFT}{RVS}[G
[G][M][*]{DOWN}{6 LEFT}{RIGHT} [G]4[2 M]
{DOWN}{6 LEFT}{RIGHT}[N][3 T]M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 227
250 XG$="{2 SPACES}"{2 DOWN}{10 RIGHT}" :rem 31
255 XG$=XG$+"{13 RIGHT}[3 @]{DOWN}{4 LEFT}{RVS}[G
[G][M][*]{DOWN}{6 LEFT}{RIGHT} [G]5[2 M]
{DOWN}{6 LEFT}{RIGHT}[N][3 T]M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 232
260 XA$="{2 SPACES}"{2 DOWN}{15 RIGHT}" :rem 171
265 XA$=XA$+"{13 RIGHT}[3 @]{DOWN}{4 LEFT}{RVS}[G
[G][M][*]{DOWN}{6 LEFT}{RIGHT} [G]6[2 M]
{DOWN}{6 LEFT}{RIGHT}[N][3 T]M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 223
270 XB$="{2 SPACES}"{2 DOWN}{20 RIGHT}" :rem 62
275 XB$=XB$+"{13 RIGHT}[3 @]{DOWN}{4 LEFT}{RVS}[G
[G]'[M][*]{DOWN}{6 LEFT}{RIGHT} [G]7[2 M]
{DOWN}{6 LEFT}{RIGHT}[N][3 T]M{OFF}{DOWN}
{5 LEFT}[5 T]" :rem 228
290 XC$=SP$+XC$:XD$=SP$+XD$:XE$=SP$+XE$:XF$=SP$+XF
$:XG$=SP$+XG$:XA$=SP$+XA$ :rem 219
295 XB$=SP$+XB$ :rem 75
310 XQ$="{6 DOWN}{2 RIGHT}" :rem 130

```

```

315 XQ$=XQ$+" {2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
    {RVS}{L}{G} {M}{*}{DOWN}{6 LEFT}{OFF}{M}{RVS}
    {G}{Q}{2 M}{DOWN}{6 LEFT}{OFF}{M}{RVS}{N}{3 T}{M}
    {OFF}{DOWN}{5 LEFT}{5 T}" :rem 110
320 XW$=" {6 DOWN}{7 RIGHT}" :rem 26
325 XW$=XW$+" {2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
    {RVS}{L}{G} {M}{*}{DOWN}{6 LEFT}{OFF}{M}{RVS}
    {G}{W}{2 M}{DOWN}{6 LEFT}{RIGHT}{N}{3 T}{M}{OFF}
    {DOWN}{5 LEFT}{5 T}" :rem 83
330 ZES=" {6 DOWN}{12 RIGHT}" :rem 156
335 ZES=ZES+" {2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
    {RVS}{L}{G} {M}{*}{DOWN}{6 LEFT}{OFF}{M}{RVS}
    {G}{E}{2 M}{DOWN}{6 LEFT}{RIGHT}{N}{3 T}{M}{OFF}
    {DOWN}{5 LEFT}{5 T}" :rem 34
340 XRS=" {6 DOWN}{17 RIGHT}" :rem 57
345 XRS=XRS+" {2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
    {RVS}{L}{G} {M}{*}{DOWN}{6 LEFT}{OFF}{M}{RVS}
    {G}{R}{2 M}{DOWN}{6 LEFT}{RIGHT}{N}{3 T}{M}{OFF}
    {DOWN}{5 LEFT}{5 T}" :rem 70
350 XTS=" {6 DOWN}{22 RIGHT}" :rem 205
355 XTS=XTS+" {2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
    {RVS}{L}{G} {M}{*}{DOWN}{6 LEFT}{OFF}{M}{RVS}
    {G}{T}{2 M}{DOWN}{6 LEFT}{RIGHT}{N}{3 T}{M}{OFF}
    {DOWN}{5 LEFT}{5 T}" :rem 77
360 XYS=" {6 DOWN}{27 RIGHT}" :rem 100
365 XYS=XYS+" {2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
    {RVS}{L}{G} {M}{*}{DOWN}{6 LEFT}{OFF}{M}{RVS}
    {G}{Y}{2 M}{DOWN}{6 LEFT}{RIGHT}{N}{3 T}{M}{OFF}
    {DOWN}{5 LEFT}{5 T}" :rem 93
370 XUS=" {6 DOWN}{32 RIGHT}" :rem 242
375 XUS=XUS+" {2 DOWN}{3 RIGHT}{3 @}{DOWN}{4 LEFT}
    {RVS}{L}{G} {M}{*}{DOWN}{6 LEFT}{OFF}{M}{RVS}
    {G}{U}{2 M}{DOWN}{6 LEFT}{RIGHT}{N}{3 T}{M}{OFF}
    {DOWN}{5 LEFT}{5 T}" :rem 82
390 XQ$=SP$+XQ$:XW$=SP$+XW$:ZES=SP$+ZES:XRS=SP$+XR
    $:XTS=SP$+XTS:XY$=SP$+XY$ :rem 132
395 XUS=SP$+XUS :rem 114
400 GOSUB 6400 :rem 222
480 REM :rem 128
490 REM ***** GET KEY ROUTINE ***** :rem 20
495 REM :rem 134
500 GET WW$: IFWW$="" THEN 500 :rem 39
502 IF WW$="" THEN 4710 :rem 116
503 IF WW$="{F3}" THEN 5010 :rem 245
504 IF WW$="{F5}" THEN 5510 :rem 252
505 IF WW$="{RIGHT}" OR WW$="{LEFT}" THEN 4000
    :rem 29
506 IF WW$="{F1}" THEN 4410 :rem 250
507 IF WW$="{UP}" OR WW$="{DOWN}" THEN 4510:rem 13

```

```

509 IF ASC(WW$)<40 OR ASC(WW$)>200 THEN 810
      :rem 250
510 IF WW$<>"1" THEN 520
      :rem 172
512 PRINTXC$;:L=195:H=16:GOSUB 1000
      :rem 34
515 PRINTC$;:GOTO500
      :rem 210
520 IF WW$<>"2" THEN 530
      :rem 175
522 PRINTXD$;:L=208:H=18:GOSUB 1000
      :rem 33
525 PRINTD$;:GOTO500
      :rem 212
530 IF WW$<>"3" THEN 540
      :rem 178
532 PRINTXE$;:L=30:H=21:GOSUB 1000
      :rem 230
535 PRINTE$;:GOTO500
      :rem 214
540 IF WW$<>"4" THEN 550
      :rem 181
542 PRINTXF$;:L=95:H=22:GOSUB 1000
      :rem 244
545 PRINTF$;:GOTO500
      :rem 216
550 IF WW$<>"5" THEN 560
      :rem 184
552 PRINTXG$;:L=29:H=25:GOSUB 1000
      :rem 246
555 PRINTG$;:GOTO500
      :rem 218
560 IF WW$<>"6" THEN 570
      :rem 187
562 PRINTXA$;:L=48:H=28:GOSUB 1000
      :rem 245
565 PRINTA$;:GOTO500
      :rem 213
570 IF WW$<>"7" THEN 580
      :rem 190
572 PRINTXB$;:L=164:H=31:GOSUB 1000
      :rem 32
575 PRINTB$;:GOTO500
      :rem 215
580 IF WW$<>"Q" THEN 590
      :rem 218
582 PRINTXQ$;:L=134:H=33:GOSUB 1000
      :rem 47
585 PRINTQ$;:GOTO500
      :rem 231
590 IF WW$<>"W" THEN 600
      :rem 217
592 PRINTXW$;:L=161:H=37:GOSUB 1000
      :rem 58
595 PRINTW$;:GOTO500
      :rem 238
600 IF WW$<>"E" THEN 610
      :rem 192
602 PRINTZE$;:L=60:H=42:GOSUB 1000
      :rem 236
605 PRINTE$;:GOTO500
      :rem 25
610 IF WW$<>"R" THEN 620
      :rem 207
612 PRINTXR$;:L=191:H=44:GOSUB 1000
      :rem 47
615 PRINTR$;:GOTO500
      :rem 226
620 IF WW$<>"T" THEN 630
      :rem 211
622 PRINTXT$;:L=58:H=50:GOSUB 1000
      :rem 1
625 PRINTT$;:GOTO500
      :rem 229
630 IF WW$<>"Y" THEN 640
      :rem 218
632 PRINTXY$;:L=97:H=56:GOSUB 1000
      :rem 16
635 PRINTY$;:GOTO500
      :rem 235
640 IF WW$<>"U" THEN 500
      :rem 210
642 PRINTXU$;:L=72:H=63:GOSUB 1000
      :rem 4
645 PRINTU$;:GOTO500
      :rem 232
700 GOTO 500
      :rem 101
800 REM
      :rem 124
805 REM **** SHARP VALUES ****
      :rem 31
807 REM
      :rem 131
810 IF WW$<>"1" THEN 820
      :rem 162
  
```

```

812 PRINTXC$;:L=194:H=17:GOSUB 1000           :rem 37
815 PRINTC$;:GOTO500                          :rem 213
820 IF WW$<>CHR$(34)THEN 830                  :rem 248
822 PRINTXD$;:L=238:H=19:GOSUB 1000          :rem 40
825 PRINTD$;:GOTO500                          :rem 215
830 IF WW$<>"#" THEN 840                      :rem 168
832 PRINTXE$;:L=30:H=21:GOSUB 1000          :rem 233
835 PRINTE$;:GOTO500                          :rem 217
840 IF WW$<>"$" THEN 850                      :rem 171
842 PRINTXF$;:L=180:H=23:GOSUB 1000          :rem 35
845 PRINTF$;:GOTO500                          :rem 219
850 IF WW$<>"%" THEN 860                      :rem 174
852 PRINTXG$;:L=155:H=26:GOSUB 1000         :rem 42
855 PRINTG$;:GOTO500                          :rem 221
860 IF WW$<>"&" THEN 870                      :rem 177
862 PRINTXA$;:L=221:H=29:GOSUB 1000         :rem 34
865 PRINTA$;:GOTO500                          :rem 216
870 IF WW$<>"'" THEN 880                      :rem 180
872 PRINTXB$;:L=164:H=31:GOSUB 1000         :rem 35
875 PRINTB$;:GOTO500                          :rem 218
880 IF WW$<>"Q" THEN 890                      :rem 96
882 PRINTXQ$;:L=132:H=35:GOSUB 1000         :rem 50
885 PRINTQ$;:GOTO500                          :rem 234
890 IF WW$<>"W" THEN 900                      :rem 95
892 PRINTXW$;:L=221:H=39:GOSUB 1000         :rem 60
895 PRINTW$;:GOTO500                          :rem 241
900 IF WW$<>"E" THEN 910                      :rem 70
902 PRINTZE$;:L=60:H=42:GOSUB 1000          :rem 239
905 PRINTEE$;:GOTO500                          :rem 28
910 IF WW$<>"R" THEN 920                      :rem 85
912 PRINTXR$;:L=104:H=47:GOSUB 1000         :rem 47
915 PRINTR$;:GOTO500                          :rem 229
920 IF WW$<>"T" THEN 930                      :rem 89
922 PRINTXT$;:L=55:H=53:GOSUB 1000          :rem 4
925 PRINTT$;:GOTO500                          :rem 232
930 IF WW$<>"Y" THEN 940                      :rem 96
932 PRINTXY$;:L=187:H=59:GOSUB 1000         :rem 70
935 PRINTY$;:GOTO500                          :rem 238
940 IF WW$<>"U" THEN 500                      :rem 85
942 PRINTXU$;:L=72:H=63:GOSUB 1000          :rem 7
945 PRINTU$;:GOTO500                          :rem 235
950 REM                                         :rem 130
960 REM **** PLAY NOTE *****                :rem 105
970 REM                                         :rem 132
1000 POKE L(A3),L: POKE H(A3), H              :rem 99
1010 POKE V(A3), 65: POKE 54296, 15          :rem 168
1020 FOR R=0 TO LE:NEXT                       :rem 22
1030 VL=54296                                  :rem 173
1040 IFA3=2THENFOR R=0 TO LE STEP.05:NEXT:FORR=0 T
      O 2:POKE V(R),0:NEXT:POKEVL,0          :rem 219

```

```

1100 REM :rem 166
1102 REM ** STORE TONE/TIMING VALUES ** :rem 218
1104 REM :rem 170
1110 POKE MM+49152,L:POKE MM+49153,H :rem 220
1120 MM=MM+2: A3=A3+1 :rem 77
1130 NOS(A3-1)=RIGHT$("{3 SPACES}" + K$(H), 3)
:rem 175
1140 LOS(A3-1)=RIGHT$("{3 SPACES}" + STR$(L), 3)
:rem 96
1150 UPS(A3-1)=RIGHT$("{3 SPACES}" + STR$(H), 3)
:rem 103
1160 IF A3=3 THEN POKE MM+49152,LE: MM=MM+1:LB$=RI
GHT$("{3 SPACES}" + STR$(LE), 4) :rem 17
1170 IF A3=3 THEN FOR QQ=49152+MM TO 49152+MM+6: P
OKE QQ,0: NEXT :rem 169
3000 REM :rem 167
3002 REM **** DISPLAY VALUES **** :rem 223
3004 REM :rem 171
3005 DD$(SK)=NOS(0)+" "+NOS(1)+" "+NOS(2)+"_"+LOS(
0)+"/"+UPS(0)+"_" :rem 179
3006 DD$(SK)=DD$(SK)+LOS(1)+"/"+UPS(1)+"_"+LOS(2)+
"/"+UPS(2)+"_" + LB$ :rem 179
3008 PRINT"{HOME}{2 DOWN}"; :rem 12
3010 IF SK<9 THEN FOR R=0 TO SK:PRINTDD$(R);:NEXT:
GOTO 3020 :rem 126
3015 FOR R=SK-9 TO SK:PRINTDD$(R);:NEXT :rem 78
3020 IF A3=3 THEN NOS(0)="{3 SPACES}":NOS(1)="
{3 SPACES}":NOS(2)="{3 SPACES}":LB$="
{4 SPACES}" :rem 148
3025 IF A3=3 THEN FOR R=0 TO 2: LOS(R)="{3 SPACES}
":UPS(R)="{3 SPACES}": NEXT: A3=0: SK=SK+1
:rem 206
3030 RETURN :rem 166
4000 REM :rem 168
4002 REM **** SWITCH MODES **** :rem 68
4003 REM :rem 171
4005 IF I=-1 THEN 4010 :rem 48
4007 PRINT"{HOME}{4 SPACES}NOTE{3 SPACES}-[7]#1
{WHT}LO/UP-[7]#2{WHT}LO/UP-[7]#3{WHT}LO/UP-
{RVS}"; :rem 207
4008 PRINTRIGHT$("{3 SPACES}" + STR$(LE), 3); "{OFF}";
:I=I*-1: GOTO 500 :rem 254
4010 PRINT"{HOME}{RVS}{4 SPACES}NOTE{3 SPACES}
{OFF}-[7]#1{WHT}LO/UP-[7]#2{WHT}LO/UP-[7]#3
{WHT}LO/UP-LEN ";:I=I*-1: GOTO 500 :rem 211
4400 REM :rem 172
4402 REM **** PLAY BACK MUSIC **** :rem 198
4405 REM :rem 177
4410 IF SK=0 THEN GOTO 500 :rem 96

```

```

4415 FOR Q=49152 TO 49152+(7*(SK-1)) STEP 7
:rem 141
4420 POKE 54272,PEEK(Q):POKE 54273,PEEK(Q+1)
:rem 173
4425 POKE 54279,PEEK(Q+2):POKE 54280,PEEK(Q+3)
:rem 22
4430 POKE 54286,PEEK(Q+4):POKE 54287,PEEK(Q+5)
:rem 27
4435 NX=PEEK(Q+6)
:rem 219
4440 POKE 54276, 65: POKE 54283, 65: POKE 54290, 6
5: POKE 54296, 15
:rem 169
4450 FOR R=0 TO NX STEP.05 :NEXT
:rem 4
4460 POKE 54276, 0: POKE 54283, 0: POKE 54290, 0:
{SPACE}POKE 54296, 0
:rem 196
4470 NEXT:GOTO 500
:rem 22
4500 REM
:rem 173
4502 REM **** CURSOR UP/DOWN ****
:rem 233
4505 REM
:rem 178
4507 REM --- SCROLL WINDOW ---
:rem 105
4508 REM
:rem 181
4509 IF A3<>0 THEN 500
:rem 67
4510 IF I=-1 THEN 4560
:rem 59
4512 IF SK=0 THEN 500
:rem 42
4515 PRINT "{HOME}{DOWN}";
:rem 255
4517 IF SK>10 THEN 4530
:rem 152
4520 IFSK<-1 THEN MM=MM-7:IC=IC-1:FOR R=0TOSK-1:PRIN
T"{DOWN}";:NEXT:SK=SK-1:PRINT LL$
:rem 104
4522 IF SK<0 THEN SK=0: IF IC<0 THEN IC=0
:rem 137
4523 FOR QM=49152+MM TO 49152+MM+7: POKE QM, 0: NE
XT
:rem 5
4525 GOTO 500
:rem 158
4530 SK=SK-1:MM=MM-7:IC=IC-1:PRINT "{DOWN}";:FOR R=
SK-10 TO SK-1:PRINT DD$(R);:NEXT
:rem 59
4540 FOR QM=49152+MM TO 49152+MM+7: POKE QM, 0: NE
XT
:rem 4
4545 GOTO 500
:rem 160
4550 REM
:rem 178
4555 REM --- CHANGE LENGTH OF TONE ---
:rem 248
4557 REM
:rem 185
4560 IF WW$="{DOWN}" THEN 4580
:rem 194
4570 IF LE<255 THEN LE = LE+1
:rem 178
4575 PRINT "{HOME}{4 SPACES}NOTE{3 SPACES}-[7]#1
{WHT}LO/UP-[7]#2{WHT}LO/UP-[7]#3{WHT}LO/UP-
{RVS}";
:rem 217
4576 PRINTRIGHT$("{3 SPACES}"+STR$(LE),3);" {OFF}";
: GOTO 500
:rem 119
4580 IF LE>=1 THEN LE = LE-1
:rem 137
4585 PRINT "{HOME}{4 SPACES}NOTE{3 SPACES}-[7]#1
{WHT}LO/UP-[7]#2{WHT}LO/UP-[7]#3{WHT}LO/UP-
{RVS}";
:rem 218

```



```

4586 PRINTRIGHT$(" {3 SPACES}" + STR$(LE), 3); "{OFF}";
      : GOTO 500                                :rem 120
4700 REM                                       :rem 175
4702 REM **** ENTER A PAUSE ****              :rem 62
4705 REM                                       :rem 180
4710 H=0:L=0: GOSUB 1000                       :rem 239
4720 GOTO 500                                  :rem 155
5000 REM                                       :rem 169
5010 REM **** SAVE ROUTINE ****               :rem 79
5020 REM                                       :rem 171
5025 PRINT "{CLR}"                             :rem 48
5030 INPUT "{HOME}{5 DOWN}{2 SPACES}SAVE ON DISK O
      R TAPE (D/T)"; ME$                        :rem 207
5040 IF ME$="C" THEN GOSUB 6430: GOTO 500      :rem 86
5050 IF ME$<>"D" AND ME$<>"T" THEN 5060       :rem 166
5055 GOTO 5210                                 :rem 208
5060 PRINT"{DOWN}{6 SPACES}PLEASE ENTER 'D' FOR DI
      SK"                                       :rem 137
5070 PRINT"{19 SPACES}'T' FOR TAPE"           :rem 80
5080 PRINT"{15 SPACES}OR{2 SPACES}'C' TO CANCEL"
      :rem 25
5090 PRINT" SAVE AND RETURN TO THE MAIN PROGRAM
      {DOWN}"                                   :rem 83
5100 GOTO 5030                                 :rem 199
5200 REM                                       :rem 171
5202 REM **** ENTER NAME OF SONG ****         :rem 104
5205 REM                                       :rem 176
5210 PRINT"{HOME}{13 DOWN}{14 RIGHT}-----
      --"                                       :rem 239
5220 PRINT"{DOWN}(MAX 16 LETTERS)?"           :rem 171
5230 INPUT "{4 UP}NAME OF SONG";NM$           :rem 201
5240 NM$=LEFT$(NM$,16)                         :rem 185
5250 PRINT"{HOME}{12 DOWN}{14 RIGHT}{20 SPACES}"
      :rem 18
5260 PRINT"{HOME}{12 DOWN}{14 RIGHT}";NM$;"
      {10 SPACES}"                             :rem 140
5270 INPUT"{HOME}{21 DOWN}IS THIS CORRECT (Y/N)";C
      R$                                       :rem 27
5280 IF CR$="Y" THEN 5400                       :rem 233
5285 IF CR$="C" THEN GOSUB 6430: IC=0:SK=0: GOTO 5
      00                                       :rem 220
5290 PRINT"{HOME}{12 DOWN}{14 RIGHT}{22 SPACES}"
      :rem 22
5300 PRINT"{14 SPACES}"                       :rem 153
5310 GOTO 5210                                 :rem 202
5400 REM                                       :rem 173
5402 REM **** SAVE MUSIC ****                 :rem 175
5405 REM --- DISK ---                          :rem 235
5410 IF ME$="T" THEN 5455                       :rem 230
5420 OPEN 1,8,4,"@:"+NM$+"",W"                :rem 184

```

```

5425 FOR Q=49152 TO 49152 + ((SK)*7)           :rem 190
5430 PRINT#1,PEEK(Q): NEXT                     :rem 25
5440 CLOSE 1                                  :rem 116
5450 GOSUB 6430                                :rem 27
5451 PRINT"{HOME}{2 DOWN}";:FOR R=SK-10 TO SK-1:PR
INT DD$(R);:NEXT                             :rem 85
5452 GOTO 500                                  :rem 158
5455 REM --- TAPE ---                          :rem 239
5460 OPEN 1,1,1,NM$: GOTO 5425                :rem 26
5505 REM                                       :rem 179
5510 REM **** LOAD ROUTINE ****               :rem 69
5520 REM                                       :rem 176
5525 PRINT "{CLR}"                             :rem 53
5530 INPUT "{HOME}{5 DOWN}{2 SPACES}LOAD FROM DISK
OR TAPE (D/T)"; ME$                           :rem 92
5540 IF ME$="C" THEN GOSUB 6430: GOTO 500     :rem 91
5550 IF ME$<>"D" AND ME$<>"T" THEN 5560      :rem 176
5555 GOTO 5710                                  :rem 218
5560 PRINT"{DOWN}{6 SPACES}PLEASE ENTER 'D' FOR DI
SK"                                             :rem 142
5570 PRINT"{19 SPACES}'T' FOR TAPE"          :rem 85
5580 PRINT"{15 SPACES}OR{2 SPACES}'C' TO CANCEL"
:rem 30
5590 PRINT" LOAD AND RETURN TO THE MAIN PROGRAM
{DOWN}"                                       :rem 73
5700 REM                                       :rem 176
5702 REM **** ENTER NAME OF SONG ****         :rem 109
5705 REM                                       :rem 181
5710 PRINT"{HOME}{13 DOWN}{14 RIGHT}-----"
---"                                          :rem 244
5720 PRINT"{DOWN}(MAX 16 LETTERS)?"          :rem 176
5730 INPUT "{4 UP}NAME OF SONG";NM$          :rem 206
5740 NM$=LEFT$(NM$,16)                         :rem 190
5750 PRINT"{HOME}{12 DOWN}{14 RIGHT}{20 SPACES}"
:rem 23
5760 PRINT"{HOME}{12 DOWN}{14 RIGHT}";NM$;"
{10 SPACES}"                                  :rem 145
5770 INPUT"{HOME}{21 DOWN}IS THIS CORRECT (Y/N)";C
R$                                             :rem 32
5780 IF CR$="Y" THEN 5900                       :rem 243
5785 IF CR$="C" THEN GOSUB 6430: GOTO 500     :rem 105
5790 PRINT"{HOME}{12 DOWN}{14 RIGHT}{22 SPACES}"
:rem 27
5800 PRINT"{14 SPACES}"                       :rem 158
5810 GOTO 5710                                  :rem 212
5900 REM                                       :rem 178
5902 REM **** LOAD MUSIC ****                 :rem 165
5905 GOSUB 6430                                :rem 32
5910 IF ME$="T" THEN 5960                      :rem 236

```

```

5912 FOR R=49152 TO 49159: POKE R,0: NEXT :rem 139
5915 REM --- DISK --- :rem 241
5920 OPEN 1,8,4,NM$+",R":SK=0:IC=0 :rem 71
5925 GOSUB 6430 :rem 34
5927 INPUT#1,L:BB=ST:INPUT#1,H :rem 103
5928 IF BB<>0 THEN 5940 :rem 150
5929 GOSUB 1000 :rem 26
5930 INPUT#1,L:INPUT#1,H:GOSUB 1000 :rem 58
5931 INPUT#1,L:INPUT#1,H:INPUT#1,LE:GOSUB 1000 :rem 22
5932 BB=ST: IF BB<>0 THEN 5940 :rem 51
5935 GOTO 5927 :rem 230
5940 CLOSE 1: SK=SK+1: IC=IC+1 :rem 115
5950 GOTO 500 :rem 161
5955 REM --- TAPE --- :rem 244
5960 OPEN 1,1,0,NM$:SK=0:IC=0:GOTO 5927 :rem 157
6400 REM :rem 174
6410 REM **** PRINT DISPLAY SCREEN **** :rem 140
6420 REM :rem 176
6430 PRINT"{WHT}{CLR}{12 DOWN}[40 U]{HOME}"; :rem 16
6440 PRINT"{HOME}{2 DOWN}";:FOR O=0 TO 9:PRINTLL$; :rem 255
: NEXT
6445 DL$="{HOME}{13 DOWN}{4 RIGHT}":DN$="{HOME} :rem 196
{19 DOWN}{5 RIGHT}"
6450 PRINT"{HOME}";GR$;A$;B$;C$;D$;E$;F$;G$;Q$;W$; :rem 105
EE$;R$;T$;Y$;U$;DL$;
6460 PRINT"C{4 SPACES}D{4 SPACES}E{4 SPACES}F :rem 152
{4 SPACES}G{4 SPACES}A{4 SPACES}B"DN$"↑C
{3 SPACES}↑D{3 SPACES}↑E";
6470 PRINT"{3 SPACES}↑F{3 SPACES}↑G{3 SPACES}↑A :rem 171
{3 SPACES}↑B{HOME}{3 DOWN}";
6502 RETURN :rem 173

```

Notes of the scale. As in the sound editor, the notes of the scale are labeled above the keys. The top row is one octave lower than the bottom row. To show the difference between the octaves, the keys for the higher notes (bottom row) are preceded by an up arrow (↑). These arrows also appear in the table in the upper half of the screen as you enter notes so you'll be able to tell what notes are being played. To play a note, press the appropriate key.

Using the table. There are a few differences between the tables in the two editors. The first thing you may notice is that the column containing the notes is much wider than before. This is to accommodate the three notes that can be played at once. Additionally, two of the columns have been deleted. These are the PAUSE column and the # column. Pauses are now indicated by a break in the NOTE column (no note designated) and the 0's in the LO/UP value columns.

NOTE. When the program is first turned on, the NOTE heading is highlighted just as before, but now it's three times as wide. This allows room for three notes to be displayed in the same column. When you enter the first note of a chord, that note is played, and the name of the note is displayed in this column. While the computer is waiting for you to enter another note from the keyboard, the first note continues to play. When you enter a second and then a third note, this column shows the notes as you enter them.

After the third note has been played, all three notes will be turned off. You *must* enter at least ten chords when you use this editor.

LO/UP. Each of these three columns shows the actual POKE values being entered into the pitch control registers. This editor creates the values for you that you would need to POKE into the low and high pitch registers; all you need to do is copy them down as the notes are entered. Then you can place these values in the appropriate locations in your own sound routine.

LEN. The value shown in the LEN column is not exactly the same as it was in the Sound Editor. While the number shown is indeed the one that is used in the timing loop, the loop itself has been changed to directly reflect the relative length of the notes in the Chord Editor. To change this value, press the Cursor Right/Left key at the lower right corner of the keyboard. This will highlight the LEN column and will display the current timing value.

Pressing the SHIFTed Cursor Up/Down key increases the timing value, while using the unSHIFTed Cursor Up/Down key decreases the value. However, the Chord Editor uses timing increments of one, allowing you higher timing resolution. The highest timing value available is 255 and the lowest is 0.

#. Since the LEN value now represents the relative length of the tone, the # column is not needed, and has been deleted from this editor.

Modifying the music. Just as with the Sound Editor, you can delete old chords and insert new ones. Using the SHIFTEd or unSHIFTEd Cursor Up/Down key (when the NOTE column is highlighted), you can delete chords from the bottom of the table on up. As in the Sound Editor, what is actually happening as the screen scrolls is that the chords are being erased. Remember that using this function *deletes all of the notes from the bottom of the table on up to the point where you stop pressing the Cursor Up/Down key*. When you use the delete function, you're also erasing entire chords, not just single notes. Otherwise, the music may be distorted.

Sharps and flats. Placement and use of sharps and flats are the same with the Chord Editor as you saw in the Sound Editor earlier. Refer to Figures 2-1 and 2-2 for the keys which select sharps and flats. The explanation of their use and display in the table was covered in the documentation for the Sound Editor. Take a look at that explanation if you've forgotten how to access sharps and flats on the keyboard.

SAVEing/LOADing music. The SAVE and LOAD functions have not been changed. The computer will ask you the same questions when you wish to save or load a song, and the music will be stored in exactly the same way. Since the Chord Editor saves three notes at a time, however, music stored on it will not be readable by the Sound Editor. Nor will music saved by the Sound Editor work with the Chord Editor.

Hints and tips. Keep these things in mind as you're using the Chord Editor. They're almost identical to the hints listed under the Sound Editor explanation. If you're entering a large number of notes, there will be times when the last note key you pressed will stay highlighted. If you press any of the note keys, or even one of the function keys, nothing will happen. Again, there's nothing wrong. The program is just taking some time to place all the values you've entered into the computer's memory. Wait for the highlighted key to return to its normal display. Then you can continue to add more chords, or play the song, or save it to tape or disk. It's a good idea to avoid pressing keys while the program is storing values; otherwise, it will read the keyboard buffer and place as many notes in the song as the number of times you hit that key. You'll have to delete those extra notes if that happens.

If you want to delete a chord or chords, but still want a

copy of the song as it exists, be sure to save it to tape or disk. Then when the screen returns, you can delete chords and listen to the new tune. If you don't like your alterations, at least you'll have a backup copy of the original version of the song on tape or disk. If you liked the original better, you can just load it and go on from there.

As the Chord Editor is listed, it uses the pulse waveform, with attack and sustain set to 0, the sustain/release control register set with a value of 240. You can change the waveform used by altering lines 1010 and 4440 in the program. Just change the value POKEd into the three waveform control registers, locations 54276, 54283, and 54290. You can even add an attack/decay value in lines 6, 7, and 8, where the program initializes control registers for the three voices.

You now have two utilities in hand that will make it much easier to create single or multivoiced sounds on the Commodore 64. You've even seen how to create harmony and disharmony. Now that you have a firm idea of how to create sound on the 64, what can you do with that knowledge? One of the most common uses of sound on this computer is creating sound effects, both duplications of natural sounds and new artificial effects. That's what the next chapter will cover—how to create sound effects. You'll quickly have a small library of sound effects. And that's not far from creating your own.

CHAPTER

3

Sound Effects



3

Sound Effects

Up to now, you've been seeing how the features of the Commodore 64 SID chip operate and how to apply them to create simple sounds. Using the sound control registers, you've learned how to set various memory locations with values, producing sound. The "Sound Editor" and "Chord Editor" in the previous chapter gave you tools to make that process easier.

In this chapter we'll explore methods of using the SID chip to manipulate sounds to make them imitate nature. You'll be using only the functions discussed so far, so if you've read through the first two chapters, as well as the glossary in the Introduction, you're ready to begin.

We'll look at and listen to four different sound types: hard sounds, soft sounds, slowly rising tapered sounds, and slowly falling tapered sounds. We'll examine a number of examples of how these sound types can be used to produce special sound effects, and how they can be combined to make entirely new sounds—sounds that nature cannot create, but your computer can.

Hard Sounds

Definitions sometimes have to be somewhat arbitrary. In order to separate the different kinds of sound effects, we'll do just that. Let's define *hard sounds* as those sounds that both begin and end abruptly. Another way of saying this would be to say that they have rapid attack, decay, and release rates. The sounds climb to maximum volume quickly, fall to the sustain volume level as fast as possible, and drop from that sustain volume rapidly. They may be short, as in Program 3-1, or long, as shown in Program 3-2. Type in and RUN these two short routines to hear examples of the characteristics of a hard sound.

Program 3-1. Short Beep

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 FOR R=54272 TO 54296:POKER,0:NEXT      :rem 24
20 POKE 54296,15                          :rem 45
30 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 0
40 POKE 54272,47:POKE 54273,25           :rem 47
50 POKE 54276,65                          :rem 51
60 FOR R=0 TO 300:NEXT                    :rem 187
70 POKE 54276,64:POKE 54296,0           :rem 3
```

Program 3-2. Longer Beep

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 FOR R=54272 TO 54296:POKER,0:NEXT      :rem 24
20 POKE 54296,15                          :rem 45
30 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 0
40 POKE 54272,47:POKE 54273,25           :rem 47
50 POKE 54276,65                          :rem 51
60 FOR R=0 TO 1000:NEXT                   :rem 233
70 POKE 54276,64:POKE 54296,0           :rem 3
```

You'll notice that the only difference between these two programs is in line 60, which is the sustain delay loop. Program 3-2 plays three times as long only because its delay loop is longer. The attack/decay rate is set to the fastest possible in line 30 of both routines by POKEing the attack/decay control register (location 54277) with 0. The release is also set to the most rapid rate. POKEing the sustain/release control register (location 54278) with 240 sets the sustain volume to maximum ($15 \times 16 = 240$), but the release is 0, indicating that the sound falls off quickly after the sustain.

You can even make hard sounds change volume in the middle of play, as Program 3-3 does.

Program 3-3. Down and Up Sounds

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 FOR R=54272 TO 54296:POKER,0:NEXT      :rem 24
20 POKE 54296,15                          :rem 45
30 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 0
40 POKE 54272,47:POKE 54273,25           :rem 47
50 POKE 54276,65                          :rem 51
55 FOR T=0 TO 300:NEXT                    :rem 193
60 FOR R=15 TO 1 STEP -.3:POKE 54296,R:NEXT:rem 74
65 FOR R=1 TO 15 STEP .3:POKE 54296,R:NEXT :rem 34
70 POKE 54276,64:POKE 54296,0           :rem 3
```

This routine is almost identical to Program 3-1. Lines 60 and 65 are new, however. They are what make the volume level first go down, then back up. Note the use of the STEP command in both lines. In line 60, the volume falls in steps of .3, while it climbs in increments of .3 in line 65.

Often, hard sounds are most interesting when they're repeated. Type in and RUN Program 3-4 to hear this effect.

Program 3-4. Repeater

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24
20 POKE 54296,15                               :rem 45
30 POKE54275,8:POKE 54277,0:POKE54278,240      :rem 0
40 POKE 54272,47:POKE 54273,25                :rem 47
45 FOR R=0 TO 10                               :rem 19
50 POKE 54296,15                               :rem 48
55 POKE 54276,65                               :rem 56
60 POKE 54276,64:POKE 54296,0                 :rem 2
70 NEXT                                         :rem 166
```

Changing a few of the control register values will modify the sound of this significantly. For instance, change line 45 so that the FOR-NEXT loop has a range from 0 to 100, instead of only to 10. This simply lengthens the time the sound plays.

Eliminating line 50 and the POKE 54296,0 from line 60 will also change the sound. Instead of the sound being turned on, then off, each time through the loop, only the gate bit is turned on and off. The sound seems faster and more run-together.

Since the only requirement for a hard sound is that it begin and end abruptly (use rapid attack, decay, and release rates), you can use any of the waveforms in producing them.

The single blip in Figure 3-1 illustrates the way that a hard sound begins at the base line, rises quickly to a high volume, and then falls just as quickly to a low volume. To create this kind of sound, you can use any waveform available on the 64, or even switch between them.

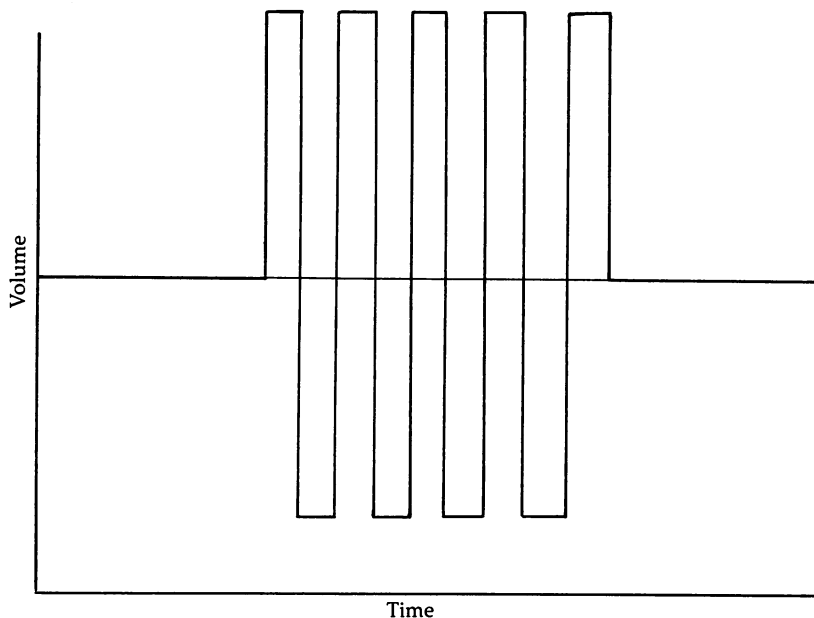
Program 3-5. Switching Waveforms

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24
20 POKE 54296,15                               :rem 45
30 POKE54275,8:POKE 54277,0:POKE54278,240      :rem 0
```

```
40 POKE 54272,47:POKE 54273,25           :rem 47
50 POKE 54276,33                         :rem 46
60 FOR R=0 TO 20:NEXT:POKE 54276,32      :rem 140
62 POKE 54276,17                         :rem 51
65 FOR R=0 TO 20:NEXT:POKE 54276,16     :rem 147
67 POKE 54276,65                         :rem 59
69 FOR R=0 TO 20:NEXT:POKE 54276,64     :rem 154
70 FOR R=0 TO 300:NEXT                   :rem 188
80 GOTO 50                                :rem 6
```

Figure 3-1. Hard Sound



Notice how the waveforms are changed in this program. First, the sawtooth waveform is enabled, then the gate bit is turned off (lines 50 and 60). Next, the triangle waveform is used (lines 62 and 65), and finally the pulse waveform is enabled and the gate bit turned off (lines 67 and 69). Although they almost run together since each waveform plays for such a short time, if you lengthen the FOR-NEXT loops in lines 60, 65, and 69, you'll be able to hear the differences in the sounds the three waveforms make. Try FOR R=0 TO 90

in these loops; the separate waveforms will be easier to tell apart. Eliminate lines 67 and 69 entirely to hear what it sounds like using only *two* waveforms. It's quite different.

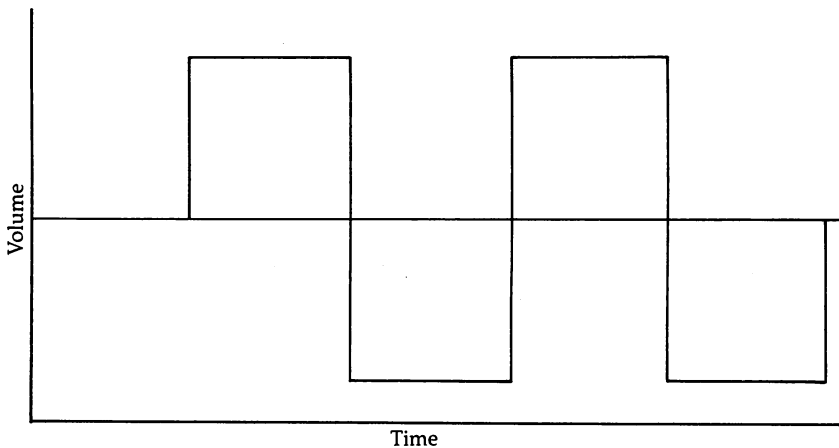
Because the waveforms do sound so different, we'll divide each of the sound types into four sections, separately covering pulse, triangle, sawtooth, and noise waveforms.

Hard Sounds Using Pulse Waves

This combination of sound components produces the hardest sounds the Commodore 64 can make. This is because the sounds start very quickly *and* the waveforms consist of very abrupt changes.

If we take a simple, hard pulse waveform that is symmetrical (also called a *square wave*) and make it very short in duration, it produces a kind of clicking sound. Figure 3-2 shows this square waveform, while Program 3-6 demonstrates the sound effect.

Figure 3-2. Hard Pulse Waveform (Symmetrical)



Program 3-6. Click

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10 FOR R=54272 TO 54296:POKER,0:NEXT          :rem 24
20 POKE 54296,15                               :rem 45
30 POKE 54275,8:POKE 54277,0:POKE 54278,240   :rem 0
40 POKE 54272,47:POKE 54273,65               :rem 51
50 POKE 54276,65                               :rem 51
70 POKE 54276,64:POKE 54296,0                 :rem 3
```

Ticktock. By putting these clicks together in pairs, using two different frequencies, you can produce the regular, ticking sound of a grandfather clock. Enter and RUN Program 3-7 to hear the sound.

Program 3-7. Ticktock

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24
20 POKE 54296,5                                 :rem 252
30 POKE 54275,8:POKE 54277,0:POKE 54278,240   :rem 0
40 POKE 54272,51:POKE 54273,90                :rem 44
50 POKE 54276,65:POKE 54296,5                 :rem 7
60 POKE 54276,64:POKE 54296,0                 :rem 2
70 FOR R=0 TO 400:NEXT                         :rem 189
80 POKE 54272,47:POKE 54273,81                :rem 53
90 POKE 54276,65:POKE 54296,5                 :rem 11
110 POKE 54276,64:POKE 54296,0                :rem 46
120 FOR R=0 TO 400:NEXT                       :rem 233
130 GOTO 40                                     :rem 49

```

The sounds are short because the gate bit is turned on in one line, then turned off in the very next line. There is no sustain delay loop between those two steps to keep the note playing at its sustain level. Like the other hard sounds you've seen, the attack, decay, and release have been set to their most rapid rates. The volume level is set to 5 when the gate bit is turned on (lines 50 and 90), and then set to 0 to turn it off completely (lines 60 and 110). You can eliminate those POKE statements which set the volume control register, but the sound will be different. It won't sound quite so much like a clock, but more like a beeping noise. Turning the sound on, then off, in rapid succession makes that clicking sound you hear in this routine.

You can create some interesting sounds, using this routine, by changing a thing or two. For example, changing the FOR-NEXT loops in lines 70 and 120 (the loops create the pauses between the two clicking sounds) will produce a different sound. Changing other values, such as the pitch values in lines 40 and 80, will also make new sounds.

Adding another section to the routine can give you the effect of an alarm going off after the clock has been ticking for a short time. Add the following lines to Program 3-7 and reRUN it. The clock will ticktock 16 times (indicated by the FOR X=0 TO 15 in line 35) before the alarm goes off.

```

35 FOR X=0 TO 15
130 NEXT
132 REM
134 REM ----- ALARM -----
135 REM
150 POKE 54272,47:POKE 54273,255
160 FOR R=0 TO 150
170 POKE 54276,65
180 POKE 54296,15
190 POKE 54276,64:POKE 54296,0
200 NEXT

```

The alarm sound is constructed in the same way as the ticking noises. The gate bit is turned on and the volume set to 15 in lines 170 and 180. Then the gate bit is turned off and the volume reset to 0 in line 190. Again, turning the gate bit and volume on and off quickly like this gives the sound its unique nature.

Ping-Pong. Table tennis is a game that requires great concentration, quick reflexes, and the ability to stretch just a bit farther. In return it provides the players and observers with a very unique sound.

Program 3-8. Ping-Pong

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ----- INITIALIZE REGISTERS -----           :rem 43
6 REM                                               :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT                :rem 24
11 POKE 54296,15                                     :rem 45
12 POKE 54275,8:POKE 54277,0:POKE 54278,240        :rem 0
20 REM ----- PING SOUND VALUES -----           :rem 53
21 REM                                               :rem 71
30 POKE 54272,51: POKE 54273,75                     :rem 46
40 POKE 54276,65                                     :rem 50
50 POKE 54296,RND(0)*10+5                            :rem 26
60 POKE 54276,64:POKE 54296,0                       :rem 2
62 REM                                               :rem 76
63 REM ----- PAUSE BETWEEN NOTES -----         :rem 198
64 REM                                               :rem 78
65 FOR R=0 TO RND(0)*100+50:NEXT                    :rem 221
66 REM                                               :rem 80
78 REM ----- PONG SOUND VALUES -----           :rem 72
79 REM                                               :rem 84
80 POKE 54272,47:POKE 54273,55                     :rem 54
90 POKE 54276,65                                     :rem 55
100 POKE 54296,RND(0)*10+5                          :rem 70

```

```

110 POKE 54276,64:POKE 54296,0           :rem 46
120 FOR R=0 TO RND(0)*600+300:NEXT       :rem 56
130 GOTO 30                               :rem 48

```

Note the use of the RND function in lines 50, 65, 100, and 120. It varies the timing and loudness of the pings and pongs to add some realism to the illusion of the match.

Creaky door. When a door gets old, its hinges may become rusty. They crack and pop as they creak open, and the faster the door moves, the faster the creaks sound.

Program 3-9. Creaky Door

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

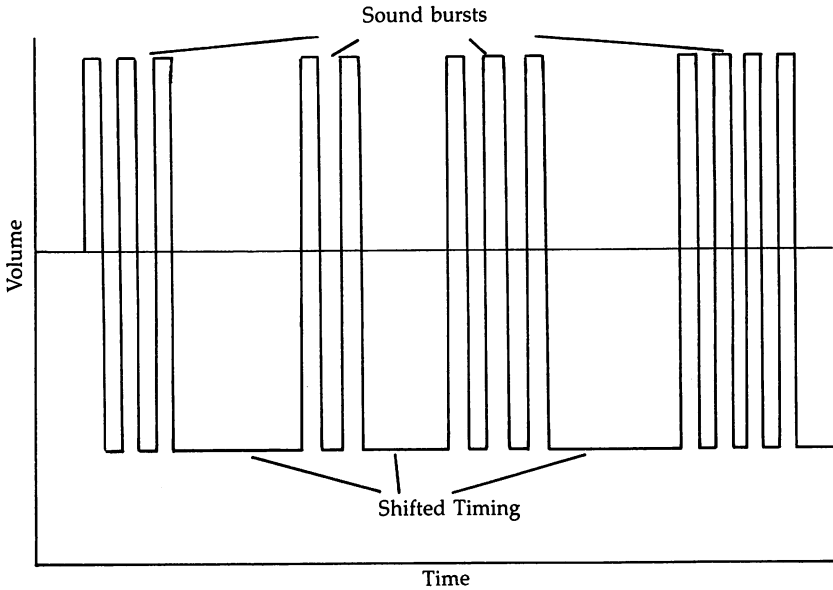
8 REM ---- INITIALIZE REGISTERS ----      :rem 46
9 REM                                     :rem 29
10 FOR R=54272 TO 54296:POKE R,0:NEXT     :rem 24
15 POKE 54296,15                          :rem 49
16 POKE 54275,0:POKE 54277,0:POKE 54278,240 :rem 252
20 REM                                     :rem 70
21 REM ---- CREAK OPEN LOOP ----         :rem 129
23 REM                                     :rem 73
25 FOR R=1 TO 30                          :rem 20
30 POKE 54273,5                            :rem 248
33 POKE 54276,65:POKE 54296,15           :rem 57
34 POKE 54276,64:POKE 54296,0           :rem 3
35 FOR T=300 TO 0 STEP-R:NEXT             :rem 122
36 NEXT                                    :rem 168
40 REM                                     :rem 72
41 REM ---- CREAK CLOSED LOOP ----       :rem 11
42 REM                                     :rem 74
50 FOR R=1 TO 30                          :rem 18
60 POKE 54273,53                          :rem 46
63 POKE 54276,65:POKE 54296,15           :rem 60
64 POKE 54276,64:POKE 54296,0           :rem 6
65 FOR T=300 TO 0 STEP -(31-R):NEXT      :rem 95
70 NEXT                                    :rem 166

```

This program uses an asymmetrical pulse waveform, which is part of the reason it sounds like it does. Line 16 includes a POKE which sets the high pulse width control register to 0, creating an asymmetrical waveform. All the previous examples in this section have used symmetrical, or square, pulse waveforms. Note the use of timing loops to create the effect of the notes starting far apart, then getting closer

together. This gives the sensation of the sound beginning slowly, then picking up speed, just as a creaky door sounds as it opens and then closes. Figure 3-3 illustrates this method.

Figure 3-3. Timing Loops



An alternate method of producing this kind of sound is by using pitch changes to create the effect. Program 3-9 used the same pitch value in the high control register throughout the routine. The clicking sounds of the door opening and closing were created by the short notes made when the gate bit and volume were turned on and off rapidly. Program 3-10 uses a different technique. The pitches change through the course of the routine, while the gate bit and volume remain on. There's usually more than one way to create a sound effect on the 64.

Program 3-10. Creaky Door, Revisited

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
5 REM ---- INITIALIZE REGISTERS ---- :rem 43
7 REM :rem 27
10 FOR R=54272 TO 54296:POKE R,0:NEXT :rem 24
15 POKE 54296,15 :rem 49
16 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 4
```

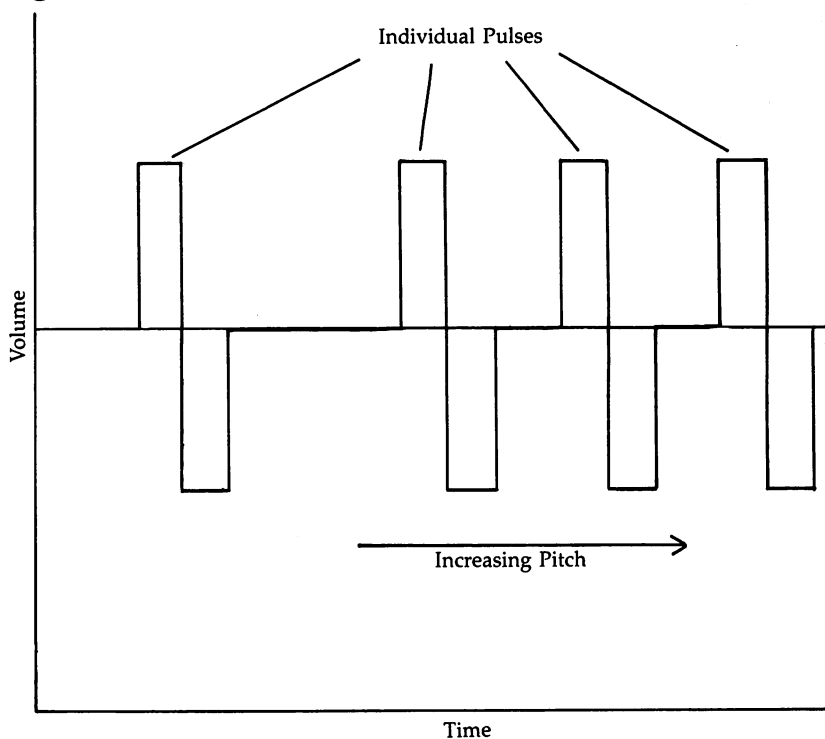
3
Sound Effects

```
17 POKE 54276,65 :rem 54
20 REM :rem 70
21 REM ---- CREAK OPEN LOOP ---- :rem 129
23 REM :rem 73
25 FOR R=10 TO 195 :rem 128
30 POKE 54272,R:POKE 54273,0 :rem 222
35 FOR T=0 TO 10:NEXT :rem 141
36 NEXT :rem 168
40 REM :rem 72
41 REM -/--- CREAK CLOSED LOOP ---- :rem 11
42 REM :rem 74
50 FOR R=195 TO 10 STEP -1 :rem 24
55 POKE 54272,R:POKE 54273,0 :rem 229
65 FOR T=0 TO 10:NEXT :rem 144
70 NEXT :rem 166
71 REM :rem 76
72 REM ---- TURN SOUND OFF ---- :rem 98
73 REM :rem 78
80 POKE 54276,64:POKE 54296,0 :rem 4
```

The 64 can produce pitches (frequencies) so low that the waveform cycles are as much as one second apart (one cycle per second, or 1 Hz). This program takes advantage of this feature and simply increases and then decreases the frequency of the sound to produce the creaks instead of using a timing loop to place each sound the right distance from the last. That's what makes it sound as if the clicks are coming slowly at first, then picking up speed. As the pitches climb in frequency, they sound faster and faster.

Compare Figure 3-3 with the one below, which shows how the incrementing of pitches can cause this type of sound.

Figure 3-4. Incrementing Pitch



Crickets. This sound effect is essentially an exercise in timing. The cricket sounds are made up of bursts of fast clicks spaced in pairs, one making the rising sound, the other the falling sound.

Program 3-11. Crickets

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ---- INITIALIZE REGISTERS ----           :rem 43
7 REM                                           :rem 27
10 FOR R=54272 TO 54296:POKE R,0:NEXT          :rem 24
15 POKE 54296,15                               :rem 49
16 POKE 54275,8:POKE 54277,0:POKE 54278,240   :rem 4
17 POKE 54272,0:POKE 54273,155               :rem 44
18 POKE 54276,65                              :rem 55
21 REM                                         :rem 71
22 REM ---- MAIN CRICK LOOP ----              :rem 123
23 REM                                         :rem 73

```

```

25 FOR K=0 TO 50                                :rem 14
30 FOR I=0 TO 1                                  :rem 212
40 FOR R=0 TO 15 STEP 5                          :rem 132
41 REM                                           :rem 73
42 REM ---- SOUND A CRICK ----                  :rem 232
43 REM                                           :rem 75
50 POKE 54296,R                                  :rem 28
60 POKE 54296,0:NEXT                             :rem 116
70 FOR M=0 TO 30:NEXT                           :rem 135
90 NEXT                                           :rem 168
91 REM                                           :rem 78
92 REM ---- INTERVAL TIMING LOOP ----           :rem 30
93 REM                                           :rem 80
100 FOR D=0 TO RND(0)*1000:NEXT                 :rem 149
120 NEXT:POKE 54276,64                           :rem 217

```

The various FOR-NEXT loops in the program set the number of times the sound pairs are heard (line 25), pair the sounds together (line 30), set the volume to different levels (line 40), and produce the irregular intervals between sound pairs (line 100). This last loop, in line 100, uses the RND function once again to remove the mechanical sound that constant timing might have.

The sounds are actually created in lines 50-70 by turning the sound on and off quickly, with a very short pause in between.

Motor boat. This program uses a counter to modify both the speed of the sound and its volume. The counter keeps track of how many sounds are produced. By doing this, you're able to make a motor boat sound that first starts, revs up to maximum speed, and then putts away down the river. Program 3-12 creates this effect. Type it in and RUN it to hear the sound.

Program 3-12. Motor Boat

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ---- INITIALIZE REGISTERS ----           :rem 43
7 REM                                           :rem 27
10 FOR R=54272 TO 54296:POKE R,0:NEXT          :rem 24
15 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 3
16 POKE 54273,8:POKE 54272,0                   :rem 200
17 POKE 54276,65                                :rem 54
41 REM                                           :rem 73
42 REM ---- BOAT GETTING CLOSER ----           :rem 178
43 REM                                           :rem 75
47 FOR K=0 TO 300                                :rem 64

```

```

50 POKE 54296,(K/25)+3           :rem 90
55 POKE 54296,0                   :rem 255
57 FOR G=0 TO RND(0)*(150-K/2):NEXT :rem 114
60 NEXT                             :rem 165
61 REM                               :rem 75
62 REM --- BOAT GETTING FARTHER AWAY -- :rem 163
63 REM                               :rem 77
70 FOR F=0 TO 450                 :rem 61
72 POKE 54296,15-F/30            :rem 57
75 POKE 54296,0                   :rem 1
77 FOR G=0 TO RND(0)*5:NEXT       :rem 233
80 NEXT                             :rem 167
91 REM                               :rem 78
92 REM ---- TURN SOUND OFF ----   :rem 100
93 REM                               :rem 80
100 POKE 54276,64:POKE 54296,0   :rem 45

```

Line 50 looks at the counter value K (which is increased by the FOR-NEXT loop in line 47) and determines how loud the motor should be by varying the volume based on the value of K. The volume level is never less than 3, and slowly increases as K gets larger. Line 55 turns the sound off; it's this familiar on and off pattern that creates the actual hard sound. In line 57, K is used to modify the RND function. At first, when K is a lower value, the delay loop is longer (maximum of 150), but as K increases, the delay loop becomes shorter, making the motor sound run faster and faster.

As the motor boat gets farther away, the value F in the second loop, found in line 70, becomes larger. As F increases, the volume level in line 72 becomes lower (15-F/30). Also, since the speed of the motor's sounds is already at maximum, a constant value is used with the RND function in line 77.

Hard Sounds Using Sawtooth Waves

If you want to use a sawtooth waveform instead of the pulse waveform, you need to change the value POKEd into location 54276 (for voice 1), 54283 (for voice 2), or 54290 (for voice 3) from 65 to 33. When you listen to a pulse waveform and then compare it to a sawtooth waveform, you'll find the sawtooth waveform has a *sharper* sound. Again, that's an aesthetic term, but it generally refers to a sound that is composed of higher frequencies. It has a higher, piercing sound to it. Let's listen to an example. Type in and RUN Program 3-13.

Program 3-13. Comparing Sounds

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 65
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT             :rem 24
11 POKE 54296,15                                :rem 45
12 POKE 54275,8:POKE 54277,0:POKE 54278,240     :rem 0
17 REM                                           :rem 76
18 REM ---- PULSE WAVE ----                     :rem 113
19 REM                                           :rem 78
20 POKE 54272,47:POKE 54273,25                   :rem 45
30 POKE 54276,65                                  :rem 49
40 FOR R=0 TO 300:NEXT                           :rem 185
50 POKE 54276,64                                  :rem 50
57 REM                                           :rem 80
58 REM ---- PAUSE BETWEEN TONES ----           :rem 202
59 REM                                           :rem 82
60 FOR R=0 TO 500:NEXT                           :rem 189
67 REM                                           :rem 81
68 REM ---- SAWTOOTH WAVE ----                 :rem 102
69 REM                                           :rem 83
70 POKE 54276,33                                  :rem 48
80 FOR R=0 TO 300:NEXT                           :rem 189
90 POKE 54276,64:POKE 54296,0                    :rem 5

```

The first tone you heard was a pulse waveform, and the second was a sawtooth waveform. Both sounds shared the same pitch values, which were POKEd into the appropriate registers in line 20. The only difference was the waveform selected: pulse in line 30, sawtooth in line 70. This one alteration changed the sound quite a bit, didn't it?

Electronic telephone ringer. Because of the sharp sounds obtainable with sawtooth waves, they lend themselves to the kind of piercing tones needed to duplicate the sound of devices like the new electronic telephone ringers.

Program 3-14. Electronic Ringer

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT             :rem 24
11 POKE 54296,15                                :rem 45
12 POKE 54277,0:POKE 54278,240                 :rem 44
20 REM                                           :rem 70
28 REM ---- NUMBER OF TONES ----                :rem 157

```

3
Sound Effects

```
29 REM :rem 79
30 FOR Y=0 TO 5 :rem 232
40 FOR W=0 TO 12 :rem 21
50 REM :rem 73
60 REM ---- HIGH TONE ---- :rem 8
70 REM :rem 75
80 POKE 54272,0:POKE 54273,125 :rem 41
90 POKE 54276,33 :rem 50
100 FOR X=0 TO 30:NEXT :rem 188
110 REM :rem 118
120 REM ---- LOW TONE ---- :rem 7
130 REM :rem 120
140 POKE 54273,100 :rem 134
150 FOR Z=0 TO 30:NEXT :rem 195
160 NEXT :rem 214
170 REM :rem 124
190 REM ---- PAUSE BETWEEN TONES ---- :rem 247
195 REM :rem 131
200 POKE 54276,32 :rem 90
210 FOR R=0 TO 2000:NEXT :rem 23
220 NEXT:POKE 54296,0 :rem 162
```

As you run this program, notice the smooth transition between the two pitches, created by leaving the volume turned on between the changes.

An old jalopy. This program is a series of five routines that tell a story through sound. It's simple, yet it's a good demonstration of a more involved sound effect.

Program 3-15. Old Car

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
5 REM ---- INITIALIZE REGISTERS ---- :rem 43
6 REM :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT :rem 24
11 POKE 54296,15 :rem 45
15 POKE 54277,0:POKE 54278,240 :rem 47
20 POKE 54272,0:POKE 54273,18 :rem 244
30 POKE 54276,33 :rem 44
35 REM :rem 76
36 REM ---- TURN IGNITION KEY ---- :rem 72
37 REM :rem 78
40 FOR R=0 TO 1 :rem 222
50 POKE 54276,33 :rem 46
60 POKE 54276,32 :rem 46
80 NEXT :rem 167
82 FOR R=0 TO 200:NEXT :rem 190
85 REM :rem 81
```

```

86 REM ----- STARTER MOTOR -----           :rem 112
87 REM                                           :rem 83
190 FOR L=0 TO 10                               :rem 62
195 FOR R=0 TO 3                                 :rem 27
200 POKE 54273,5+R                              :rem 164
210 POKE 54276,33:POKE 54296,10                :rem 92
220 NEXT                                         :rem 211
230 FOR R=0 TO 3                                 :rem 17
240 POKE 54273,12-R                             :rem 216
250 POKE 54276,33:POKE 54296,10                :rem 96
260 NEXT                                         :rem 215
270 NEXT                                         :rem 216
280 POKE 54276,32:POKE 54296,0                 :rem 49
285 REM                                           :rem 131
286 REM ----- MOTOR STARTING -----         :rem 233
287 REM                                           :rem 133
290 POKE 54273,5                                :rem 48
295 FOR L=50 TO 0 STEP -.5                       :rem 20
300 POKE 54276,33:POKE 54296,15                 :rem 97
310 POKE 54276,32:POKE 54296,0                 :rem 43
315 FOR P=0 TO RND(0)*L:NEXT                     :rem 52
320 NEXT                                         :rem 212
326 REM ----- MOTOR RUNNING -----         :rem 153
327 REM                                           :rem 128
330 FOR L=0 TO 300                              :rem 108
340 POKE 54276,32:POKE 54296,0                 :rem 46
350 POKE 54276,33:POKE 54296,15-(L/20)         :rem 193
360 FOR P=0 TO RND(0)*3:NEXT                     :rem 27
370 NEXT                                         :rem 217

```

The first thing the program does (as do all the other sound programs) is initialize the sound control registers. Then it produces the click sound of the ignition key. This effect is actually two clicks played close together. After the ignition key clicks, the program pauses for about 1/8 second as the FOR-NEXT loop in line 82 executes. This pause helps emphasize the click and separates it from the start-motor routine that follows.

The start-motor routine creates its peculiar sound by varying the pitch rapidly (lines 200 and 240). The motor-starting routine that follows produces the effect of an unstable motor, first sputtering, then eventually catching and running. The RND function (line 315) makes the engine fire somewhat unevenly. As the motor warms up, however, the randomness is reduced (because the RND function uses the counter value L

as the controlling variable, which decreases slowly in line 295), and the motor stabilizes.

The motor-running routine produces a good, stable motor sound, with a small random value to help maintain the effect of the broken-down car, that slowly fades away as the car drives off. As in previous sound effects in this chapter, this is done by lowering the volume, accomplished in line 350.

As you experiment with this program, you might try changing the values used in the pitch control register (location 54273) to get slightly different-sounding engines. Also, you could try removing the RND function for a more solid engine sound.

Hard Sounds Using Triangle Waves

If you change the value POKEd into the different voices' waveform control register to 17, the Commodore 64 will produce triangle waves. Triangle waves produce an almost muted sound, similar to putting a muffler on the sound. Try switching the waveform to a triangle wave in the last program; the sound will seem like something from a science fiction movie. This flying saucer effect is simple to recreate. Program 3-16 produces this effect.

Flying saucer. This program uses the start-motor portion of Program 3-15 and adds a second voice just slightly out of tune with the first. The program begins with the *start up engines* routine, using both voices. After the engines have been warmed up for a short time, the *running engines* routine, which warbles about twice as fast as the starter, begins, and then the two pitches gradually fade to simulate flying away.

Program 3-16. Flying Saucer

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ---- INITIALIZE REGISTERS ----           :rem 43
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24
15 POKE 54277,0:POKE 54278,240                 :rem 47
17 POKE 54284,0:POKE 54285,240                 :rem 45
20 POKE 54273,18:POKE 54280,19                 :rem 45
21 POKE 54276,17:POKE 54283,17                 :rem 49
35 REM                                           :rem 76
36 REM ---- START UP ENGINES ----             :rem 241
37 REM                                           :rem 78
190 FOR L=1 TO 15                               :rem 68

```

```

195 FOR R=0 TO 3 STEP .7           :rem 188
200 POKE 54273,5+R                 :rem 164
205 POKE 54280,8+R                 :rem 170
215 POKE 54296,L                   :rem 73
220 NEXT                           :rem 211
230 FOR R=0 TO 3 STEP .7           :rem 178
240 POKE 54273,12-R               :rem 216
245 POKE 54280,15-R              :rem 222
255 POKE 54296,L                   :rem 77
260 NEXT                           :rem 215
270 NEXT                           :rem 216
335 REM                             :rem 127
336 REM ----- RUNNING ENGINES ----- :rem 18
337 REM                             :rem 129
490 FOR L=1 TO 45                  :rem 74
495 FOR R=0 TO 3 STEP 3            :rem 141
500 POKE 54273,5+R                 :rem 167
505 POKE 54280,8+R                 :rem 173
515 POKE 54296,15-L/3             :rem 65
520 NEXT                           :rem 214
530 FOR R=0 TO 3 STEP 3            :rem 131
540 POKE 54273,12-R               :rem 219
545 POKE 54280,15-R              :rem 225
555 POKE 54296,15-L/3             :rem 69
560 NEXT                           :rem 218
570 NEXT                           :rem 219
580 POKE 54276,16:POKE 54296,0    :rem 54

```

The sounds get gradually louder by using the variable L as the value for the volume control. The pitches also change as the program executes, for the variable R determines the pitch value POKEd into the control registers. Finally, the volume decreases, again by using a variable and a FOR-NEXT loop to create the effect of the saucer flying away.

Hard Sounds That Use the Noise Waveform

Program 3-16 showed how a hard sound pattern may be softened by using a waveform like the triangle wave. This section will show you the kinds of effects that the fourth waveform, noise, can produce.

The noise waveform is enabled by POKing location 54276 for voice 1, 54283 for voice 2, or 54290 for voice 3, with a value of 129. An example will let you hear the effect of a noise waveform, perhaps the best way of describing it.

Pencil sharpener. To sharpen a pencil, you insert it in the

sharpener and turn the handle around and around. A sound duplicating this can be created by using the start-motor routine from the flying saucer sound effect you heard earlier. Changing the waveform to noise lets you produce the scratchy rotational sound of a mechanical pencil sharpener.

Program 3-17. Pencil Sharpener

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ---- INITIALIZE REGISTERS ----           :rem 43
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT          :rem 24
15 POKE 54277,0:POKE 54278,240                :rem 47
17 POKE 54284,0:POKE 54285,240                :rem 45
20 POKE 54273,18:POKE 54280,19                :rem 45
21 POKE 54276,129:POKE 54283,129              :rem 153
35 REM                                           :rem 76
36 REM ---- SHARPEN PENCILS LOOP ----         :rem 14
37 REM                                           :rem 78
190 FOR L=1 TO 15                              :rem 68
195 FOR R=0 TO 3                               :rem 27
200 POKE 54273,5+R                             :rem 164
205 POKE 54280,8+R                             :rem 170
215 POKE 54296,5                               :rem 50
220 NEXT                                        :rem 211
230 FOR R=0 TO 3                               :rem 17
240 POKE 54273,12-R                             :rem 216
245 POKE 54280,15-R                             :rem 222
255 POKE 54296,5                               :rem 54
260 NEXT                                        :rem 215
270 NEXT                                        :rem 216
277 REM                                           :rem 132
278 REM ---- TURN OFF SOUND ----              :rem 154
279 REM                                           :rem 134
280 POKE 54276,128:POKE 54283,128:POKE 54296,0 :rem 157

```

Running horse. In this routine, you'll hear a horse that starts out at a full gallop. Eventually, the horse slows down, breaking its pace. Finally, the horse slows to a trot and stops when it reaches its stall.

Program 3-18. Horse Galloping

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ---- INITIALIZE REGISTERS ----           :rem 43
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT          :rem 24
11 POKE 54296,15                               :rem 45

```

3
Sound Effects

```
15 POKE 54277,0:POKE 54278,240 :rem 47
20 POKE 54272,0:POKE 54273,218:POKE 54276,129 :rem 95
30 REM :rem 71
40 REM ---- GALLOPING HORSE LOOP ---- :rem 8
50 REM :rem 73
60 FOR L=1 TO 25 :rem 17
70 FOR R=0 TO 3 :rem 227
80 POKE 54296,5 :rem 2
90 POKE 54296,0 :rem 254
110 FOR I=0 TO 30:NEXT :rem 174
120 NEXT :rem 210
125 FOR V=0 TO 150:NEXT:NEXT :rem 109
135 REM :rem 125
140 REM ---- HORSE SLOWING LOOP ---- :rem 191
150 REM :rem 122
160 FOR L=1 TO 5 :rem 16
170 FOR R=0 TO 3 :rem 20
180 POKE 54296,5 :rem 51
190 POKE 54296,0 :rem 47
210 FOR I=0 TO RND(0)*70+80:NEXT :rem 213
220 NEXT :rem 211
225 NEXT :rem 216
235 REM :rem 126
240 REM ---- HORSE TROTting HOME ---- :rem 7
250 REM :rem 123
260 FOR L=1 TO 15 :rem 66
280 POKE 54296,5 :rem 52
300 POKE 54296,0 :rem 40
310 FOR I=0 TO 200:NEXT :rem 223
320 NEXT :rem 212
330 POKE 54276,128:POKE 54296,0 :rem 99
```

It's all a process you've seen before. The noise waveform is selected in line 20, and the sounds are created by turning the volume on and off rapidly within a loop that specifies the number of times the sounds occur (line 60, for instance), the number of beats (line 70), as well as the length of the pauses between each group of sounds (line 110). A similar design is used to make the horse seem to slow, but a random function is used to create the delays between each group of sounds (line 210). To make the horse's hoofbeats finally stop, the last section of the program uses a longer loop (line 260) and a constant pause length between the single beats (line 310).

When you use sounds in your programs, you can easily make them tell a story. Your sounds will not be simply enhancements of the graphics—they can be the story itself.

Mixing Waveforms

In addition to creating sounds that use a single waveform, you'll often find it effective to use several waveforms in one routine to produce special sounds. The telephone-dialing sound effect produced by Program 3-19 is a good example.

Dial telephone. This program uses the noise and pulse waveforms to create the two very different sounds made by standard dial telephones. Type in the following program and RUN it to hear the effect.

Program 3-19. Mixing Waveforms—Dial Phone

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ---- INITIALIZE REGISTERS ----           :rem 43
7 REM                                           :rem 27
10 FOR R=54272 TO 54296:POKE R,0:NEXT          :rem 24
11 POKE 54275,8:POKE 54277,0:POKE 54278,240
                                           :rem 255
21 REM                                           :rem 71
22 REM ---- SEVEN NUMBER LOOP ----           :rem 52
23 REM                                           :rem 73
24 FOR I=0 TO 6                               :rem 220
25 REM                                           :rem 75
26 REM ---- TURN DIAL CLOCKWISE ----         :rem 187
27 REM                                           :rem 77
28 POKE 54296,3                               :rem 2
30 G=RND(0) * 9 + 1                           :rem 11
40 FOR R=0 TO G*1.5                           :rem 178
45 POKE 54276,129                             :rem 104
50 POKE 54273,14                             :rem 42
60 POKE 54273,0                               :rem 246
70 NEXT                                        :rem 166
71 REM                                           :rem 76
72 REM ---- PAUSE AFTER DIAL ----           :rem 191
73 REM                                           :rem 78
80 FOR K=0 TO 200:NEXT                        :rem 181
81 REM                                           :rem 77
82 REM ---- AUTOMATIC DIAL RETURN ----       :rem 87
83 REM                                           :rem 79
90 POKE 54296,5                               :rem 3
95 POKE 54276,65                             :rem 60
100 FOR R=0 TO G                              :rem 33
110 POKE 54273,5                             :rem 39
120 FOR D=0 TO 1:NEXT                        :rem 120
130 POKE 54273,0                             :rem 36
140 FOR O=0 TO 50:NEXT                       :rem 185
150 NEXT                                       :rem 213

```

```
151 REM :rem 123
152 REM-PAUSE BEFORE DIALING NEXT DIGIT :rem 130
153 REM :rem 125
160 FOR J=0 TO 350:NEXT:NEXT :rem 98
161 REM :rem 124
162 REM ---- TURN SOUND OFF ---- :rem 146
163 REM :rem 126
170 POKE 54276,64:POKE 54296,0 :rem 52
```

Although this program may look complicated to you, it's actually only a long version of a process you've seen before. The number of times the telephone is dialed is set in line 24 by the FOR I=0 TO 6 statement. Seven numbers will be dialed, the usual number if you are making a local call. The turning of the rotary dial clockwise is simulated in lines 28-60. The volume is set at 3, the noise waveform is enabled, and the length of the sound is calculated using the RND function in lines 30 and 40. This makes the sound last a random length of time, just as if you were dialing different numbers. The movement for dialing a 1 is less than when you dial a 9, for example. The pitch values are changed by lines 50 and 60 each time through the routine. The sound would lose much of its characteristics if you deleted line 60, for instance. You can try it yourself to hear the change.

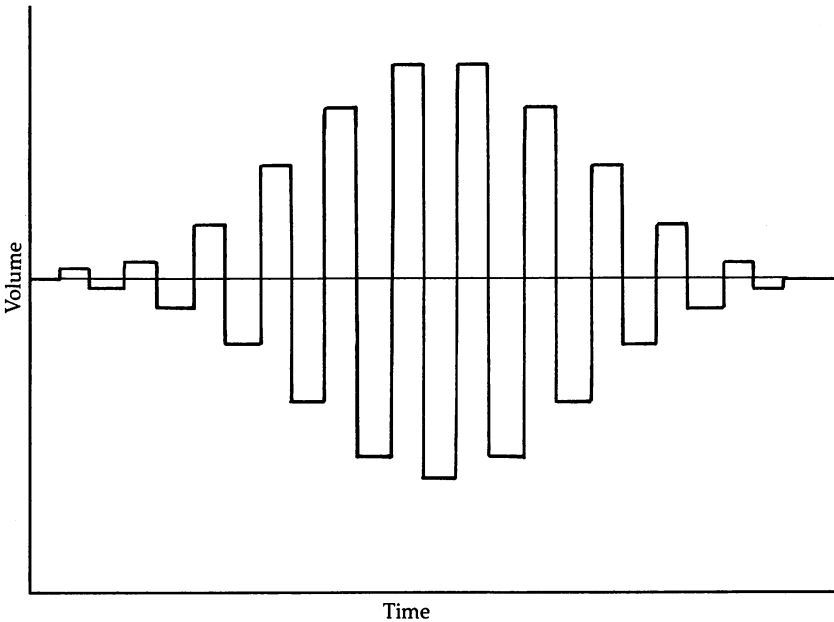
A pause is inserted between the numbers dialed by the delay loop in line 80. Then the dial returns to its initial position, which is duplicated by the section of lines from 90 to 150. The variable G, created in line 30, is used again to set the length of the dial return sound, so that the time for the return matches the time of the initial dialing. Another pause in line 160 executes a delay between the digits dialed.

Experimenting with various waveform mixes can produce some interesting effects. But you're not limited to hard sounds on the 64. You can also create soft sounds.

Soft Sounds

While hard sounds are those that begin and end abruptly, *soft* sounds are those that rise slowly to their maximum volume, fall slowly to the sustain volume level, and then decrease gradually from the sustain level to minimum volume. In other words, the sound's attack, decay, and release are set to high values, and thus slower rates. Illustrated, a soft sound might look like Figure 3-5.

Figure 3-5. Soft Waveform



As with hard sounds, soft sounds can be created with any of the available waveforms: pulse, triangle, sawtooth, or noise. Program 3-20 uses all four waveforms, one after the other, using the same pitch and the same rates of attack, decay, and release.

Program 3-20. Soft Sounds—Attack, Decay, and Release

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM --- INITIALIZE SOUND REGISTERS --      :rem 45
6 REM                                         :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT        :rem 24
11 POKE 54296,15                             :rem 45
12 POKE54275,8:POKE 54277,170:POKE54278,250
                                           :rem 105
13 POKE 54272,0:POKE 54273,45                :rem 246
16 REM                                         :rem 75
17 REM ---- SOFT PULSE WAVE ----            :rem 172
18 REM                                         :rem 77
20 POKE 54276,65                             :rem 48
30 FOR R=0 TO 500:NEXT                       :rem 186
40 POKE 54276,64                             :rem 49

```

```

50 FOR P=0 TO 500:NEXT                :rem 186
56 REM                                :rem 79
57 REM ---- SOFT TRIANGLE WAVE ----  :rem 125
58 REM                                :rem 81
60 POKE 54276,17                       :rem 49
70 FOR R=0 TO 500:NEXT                 :rem 190
80 POKE 54276,16                       :rem 50
90 FOR P=0 TO 500:NEXT                 :rem 190
96 REM                                :rem 83
97 REM ---- SOFT SAWTOOTH WAVE ----  :rem 164
98 REM                                :rem 85
100 POKE 54276,33                      :rem 90
110 FOR R=0 TO 500:NEXT                :rem 233
120 POKE 54276,32                      :rem 91
130 FOR P=0 TO 500:NEXT                :rem 233
136 REM                                :rem 126
137 REM ---- SOFT NOISE WAVE ----    :rem 212
138 REM                                :rem 128
140 POKE 54276,129                    :rem 148
150 FOR R=0 TO 500:NEXT                :rem 237
156 REM                                :rem 128
157 REM ---- TURN SOUND OFF ----     :rem 150
158 REM                                :rem 130
160 POKE 54276,128:POKE 54296,0      :rem 100

```

Each of the waveforms is first enabled, then a short delay loop for the sustain portion of the sound envelope executes, and finally the gate bit is turned off. Another FOR-NEXT loop creates a pause between the different sounds created by each waveform. For instance, the pulse waveform is enabled in line 20, the sustain delay loop is in line 30, and the gate bit is turned off in line 40. The pause between the pulse waveform and the triangle waveform is in line 50.

The attack and decay rates are set in line 12 by POKEing a value of 170 into location 54277. This gives an attack rate of 10 (16×10) and a decay rate of 10. Remember that you add the two values together to arrive at the number to POKE into the location ($160 + 10 = 170$). The sustain level has been set to maximum (16×15) and the release rate to 10 by POKEing 250 into the control register at location 54278.

There's usually more than one way to program a sound on the 64, however, and this is no exception. Since soft sounds are those where the volume increases slowly to maximum and then falls gradually to their minimum, you could also use the volume control register at location 54296 to create

them. It looks like a bit more programming in this example, but sometimes it's the only way to create the exact sound you want. Program 3-21 produces a sound very similar to that made by Program 3-20; but instead of the attack, decay, and release rates being used, this routine uses only the volume control to make a soft style of sound.

Program 3-21. Soft Sounds—Volume Control

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM --- INITIALIZE SOUND REGISTERS --           :rem 45
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT             :rem 24
12 POKE54275,8:POKE 54277,0:POKE54278,240       :rem 0
13 POKE 54272,0:POKE 54273,45                   :rem 246
16 REM                                           :rem 75
17 REM ---- SOFT PULSE WAVE ----                :rem 172
18 REM                                           :rem 77
30 POKE 54276,65                                 :rem 49
50 FOR R=0 TO 30                                 :rem 17
60 POKE 54296,R/2:NEXT                           :rem 247
65 FOR R=30 TO 0 STEP-1                          :rem 177
75 POKE 54296,R/2:NEXT                           :rem 253
80 FOR R=0 TO 100:NEXT                           :rem 187
116 REM                                          :rem 124
117 REM ---- SOFT TRIANGLE WAVE ----            :rem 170
118 REM                                          :rem 126
130 POKE 54276,17                                :rem 95
150 FOR R=0 TO 30                                 :rem 66
160 POKE 54296,R/2:NEXT                          :rem 40
165 FOR R=30 TO 0 STEP-1                         :rem 226
175 POKE 54296,R/2:NEXT                          :rem 46
180 FOR R=0 TO 100:NEXT                          :rem 236
216 REM                                          :rem 125
217 REM ---- SOFT SAWTOOTH WAVE ----            :rem 206
218 REM                                          :rem 127
230 POKE 54276,33                                :rem 94
250 FOR R=0 TO 30                                 :rem 67
260 POKE 54296,R/2:NEXT                          :rem 41
265 FOR R=30 TO 0 STEP-1                         :rem 227
275 POKE 54296,R/2:NEXT                          :rem 47
280 FOR R=0 TO 100:NEXT                          :rem 237
316 REM                                          :rem 126
317 REM ---- SOFT NOISE WAVE ----              :rem 212
318 REM                                          :rem 128
330 POKE 54276,129                               :rem 149
350 FOR R=0 TO 30                                 :rem 68
360 POKE 54296,R/2:NEXT                          :rem 42

```

```

365 FOR R=30 TO 0 STEP-1           :rem 228
375 POKE 54296,R/2:NEXT           :rem 48
380 FOR R=0 TO 100:NEXT          :rem 238
416 REM                           :rem 127
417 REM ----- TURN SOUND OFF ----- :rem 149
418 REM                           :rem 129
420 POKE 54276,128:POKE 54296,0   :rem 99

```

The attack, decay, and release rates have been set as fast as possible by the POKES in locations 54277 and 54278 in line 12. The volume control register is used instead to create the soft sound.

Take a look at lines 30–80, the routine for playing the sound using the pulse waveform. (All four waveforms use the same procedure to create their sounds.) Line 30 enables the pulse waveform. Line 60 slowly increases the volume by POKEing the control register with the variable R, which is established by the FOR-NEXT loop in line 50. The volume gradually decreases in line 75. Line 80 is the length of the sustain portion of the sound. From this point, the program moves to line 130, where the process begins all over again, only this time using the triangle waveform. Each waveform produces a sound that slowly increases, then decreases in volume. It is this slow rising and falling that creates a soft sound.

Soft Pulse Sounds

As you've seen, there are two ways to produce soft sounds. You can use the attack, decay, and release controls, or you can use the volume. Either method can be used with any of the waveforms available to you. Another example of using the attack, decay, and release rates to create a soft sound is shown in Program 3-22. In this program, only the pulse waveform is used.

Birds. If you use the pulse waveform, and select pitches at the top of its frequency range, the notes will be shrill and chirpy.

Program 3-22. Chirps

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --       :rem 0
6 REM                                         :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT        :rem 24
11 POKE 54296,15                             :rem 45
12 POKE 54275,8:POKE 54277,51:POKE 54278,240 :rem 54

```

```

13 POKE 54276,65                :rem 50
17 REM                          :rem 76
18 REM ----- CHIRP LOOPS ----- :rem 184
19 REM                          :rem 78
20 FOR S=0 TO 50                :rem 17
25 POKE 54296,15                :rem 50
37 REM                          :rem 78
38 REM ----- CHIRP UP -----   :rem 210
39 REM                          :rem 80
42 FOR C=0 TO 10 STEP RND(0)*2+1 :rem 90
43 POKE 54273,155+5*C:NEXT      :rem 168
47 REM                          :rem 79
48 REM ----- CHIRP DOWN ----- :rem 102
49 REM                          :rem 81
52 FOR D=10 TO 0 STEP -((RND(0)*2)+1) :rem 43
53 POKE 54273,155+5*D:NEXT      :rem 170
54 POKE 54296,0                 :rem 254
55 FOR T=0 TO RND(0)*75:NEXT    :rem 41
60 NEXT                          :rem 165
70 POKE 54276,64:POKE 54296,0   :rem 3

```

To give the electronic bird more realism, this program uses the counter values (the variables C on the up-chirp and D on the down-chirp) to sweep the top of the frequency scales. This is done by changing the pitch value each time through the program in lines 43 and 53. The RND function is also used to make the chirps slightly different lengths. Note, however, that the STEP value will never get lower than 1 because lines 42 and 52 add 1 to the randomly generated STEP value. This keeps the chirps from becoming unreasonably long.

Line 12 sets the attack/decay value to 51, establishing an attack of 3 ($3*16=48$) and a decay of 3. You could create a soft sound effect by using the volume control exclusively, however. That's what Program 3-23 does.

Squeaky shoes. This program uses an asymmetrical pulse waveform to create the basic sound. The sound is then broken up as the volume is tapered to produce the soft effect. This gives the sound of shoes squeaking and creaking on a floor.

Program 3-23. Squeaky Shoes

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS -- :rem 0
6 REM                                  :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT :rem 24
11 POKE 54275,1:POKE 54277,0:POKE 54278,240 :rem 248
12 POKE 54276,65                      :rem 49

```

```
13 REM :rem 72
113 REM :rem 121
114 REM ---- SQUEAKY SHOES ---- :rem 139
115 REM :rem 123
117 FOR H=0 TO 10 :rem 57
130 FOR R=0 TO 6 :rem 19
140 POKE 54296,R:POKE 54273,20-R*3 :rem 36
145 NEXT :rem 217
147 FOR R=6 TO 0 STEP -1 :rem 181
148 POKE 54296,R:POKE 54273,20-R*3 :rem 44
149 NEXT :rem 221
160 POKE 54296,0 :rem 44
165 FOR U=0 TO 350:NEXT :rem 249
170 NEXT:POKE 54276,64 :rem 222
```

Note the pulse width set in line 11 to a value of 1. This creates the asymmetrical pulse waveform. Instead of using the attack, decay, and release control registers, this routine uses the volume to produce a soft sound. The volume is slowly increased from level 0 to level 6 in line 130 and then decreased by line 147. The STEP -1 command in the latter line brings the volume level gradually down. At the same time, the pitch values are changed, using the variable R to calculate the new pitch.

Try changing the pulse width value in line 11 to make the waveform symmetrical, or square. POKEing 8 into location 54275 will do this. The effect won't sound like squeaky shoes now.

Soft Sawtooth Sounds

From earlier examples in this chapter, you saw how sawtooth waveforms produce a shrill, piercing sound. This quality makes it perfect for reproducing the sounds of many string instruments when you use a soft sort of sound.

Violin. Using the volume control register to create the soft effect, this routine includes a counter variable that increments each time the pitch goes through the timing loops in lines 150 and 165 and also causes the volume to rise and fall as the pitch begins and ends.

Program 3-24. Violins

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
5 REM -- INITIALIZE SOUND REGISTERS -- :rem 0
6 REM :rem 26
```

3
Sound Effects

```

10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24
11 POKE 54277,0:POKE 54278,240                 :rem 43
12 POKE 54276,33                               :rem 44
13 REM                                          :rem 72
14 REM ---- VIOLIN MUSIC DATA ----           :rem 29
15 REM                                          :rem 74
16 DATA 33,134,42,60,50,58,44,191,42,60,44,191,37,
    161,42,60                                   :rem 81
22 REM                                          :rem 72
23 REM ---- MUSIC LOOPS ----                 :rem 191
24 REM                                          :rem 74
25 FOR S=0 TO 2                                 :rem 227
30 FOR G=0 TO 7                                 :rem 216
117 REM                                         :rem 125
118 REM ---- VIOLIN BOW STROKE ----          :rem 119
119 REM                                         :rem 127
120 READ H,L                                   :rem 111
125 POKE 54272,L:POKE 54273,H                 :rem 37
150 FOR R=0 TO 15 STEP .75                    :rem 27
160 POKE 54296,R:NEXT                         :rem 199
165 FOR R=15 TO 0 STEP-.75                   :rem 78
175 POKE 54296,R:NEXT                         :rem 205
197 REM                                         :rem 133
198 REM ---- REPEAT MUSIC LOOP ----          :rem 106
199 REM                                         :rem 135
200 NEXT                                       :rem 209
211 RESTORE                                    :rem 184
216 NEXT                                       :rem 216
217 REM                                         :rem 126
218 REM ---- TURN OFF SOUND ----            :rem 148
219 REM                                         :rem 128
300 POKE 54296,32:POKE 54276,0              :rem 42

```

As in Program 3-23, this routine produces the soft sound by manipulating the volume. The volume is increased and decreased using the variable R, which is obtained from the loops in line 150 and 165, then POKEd into the control register in lines 160 and 175. To play the different notes, the program uses a DATA statement in line 16 that contains all the pitch values. The high and low pitch register values are READ from line 120. You can easily change the tune played by simply using different values in the DATA statements. For example, insert the following new DATA statements in Program 3-24, and the violin will play "Yankee Doodle."

```

16 DATA 33,134,33,134,37,161,42,60,33,134,42,60,37
    ,161,25,29

```

```

17 DATA 33,134,33,134,37,161,42,60,33,134,0,0,31,1
   64,0,0
18 DATA 33,134,33,134,37,161,42,60,44,191,42,60,37
   ,161,33,134,31,164,25,29
19 DATA 28,48,31,164,33,134,0,0,33,134,0,0
30 FOR G=0 TO 31

```

Line 30 has to be changed as well, so that the program knows how many pitch values to READ. By using pitch values of your own creation, you can use the violin sound effect to play almost any tune.

Soft Triangle Sounds

Since triangle waves produce the softest sounds of any of the waveforms, it makes sense that soft triangle sound effects are among the softest you can make with the 64.

Heartbeat. Here's a routine you can use as one of the background sounds to the "Chicken Heart That Ate Cleveland" videogame you always wanted to write. Or you could use this sound in any application that needs a suspenseful effect.

Program 3-25. Heartbeats

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM --- INITIALIZE SOUND REGISTERS --      :rem 45
6 REM                                          :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT          :rem 24
15 POKE 54277,0:POKE 54278,240                :rem 47
20 POKE 54276,17                               :rem 45
116 REM                                       :rem 124
117 REM ---- PACEMAKER LOOP ----             :rem 168
118 REM                                       :rem 126
120 FOR P=0 TO 35                              :rem 66
126 REM                                       :rem 125
127 REM ---- DOWN BEAT ----                  :rem 58
128 REM                                       :rem 127
150 FOR R=0 TO 15 STEP 2                       :rem 179
155 POKE 54273,35-(R+20)                      :rem 192
160 POKE 54296,R/2:NEXT                       :rem 40
161 REM                                       :rem 124
162 REM ---- UP BEAT ----                     :rem 166
163 REM                                       :rem 126
165 FOR R=15 TO 0 STEP-2                       :rem 230
170 POKE 54273,35-(R+20)                      :rem 189
175 POKE 54296,R:NEXT                         :rem 205
200 REM                                       :rem 118
201 REM ---- CYCLE PACEMAKER LOOP ----       :rem 18

```

```

202 REM :rem 120
205 POKE 54276,16 :rem 97
206 FOR G=0 TO 300:NEXT :rem 226
207 POKE 54276,17 :rem 100
208 NEXT :rem 217
216 REM :rem 125
217 REM ---- TURN SOUND OFF ---- :rem 147
218 REM :rem 127
220 POKE 54276,16:POKE 54296,0 :rem 45

```

This program uses a very low pitch and the triangle waveform to produce the muffled sound of a heartbeat. The pitch fluctuates slightly as the sound rises and falls in volume. This is accomplished by POKEing different values of R (the volume variable) into the high tone register in lines 155 and 170.

If you eliminate lines 205 and 207, the heartbeat will have a peculiar echo. Deleting line 206 as well makes the sound faster, as if the heartbeat were suddenly increased.

Soft Noise

Soft noise almost sounds like a contradiction in terms. However, there are many sounds that are soft and yet have the irregular qualities that can best be reproduced using the noise waveform.

Sawing wood. To produce the sawing sound in this program, two different pitch values are used with the noise waveform register. The first pitch is higher than the second and subtly gives the impression of greater saw speed on the up stroke. Type it in and RUN it to hear this illusion.

Program 3-26. Sawing Wood

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ---- CLEAR SOUND REGISTERS ---- :rem 41
6 REM :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT :rem 24
11 POKE 54277,0:POKE 54278,240 :rem 43
12 POKE 54276,129 :rem 98
13 REM :rem 72
14 REM -- NUMBER OF SAW STROKES LOOP -- :rem 171
15 REM :rem 74
16 FOR L=0 TO 15 :rem 16
17 REM :rem 76
18 REM ---- UPWARD SAW STROKE ---- :rem 75
19 REM :rem 78
20 POKE 54273,75 :rem 46

```

```

50 FOR R=0 TO 15 STEP .7           :rem 181
60 POKE 54296,R:NEXT              :rem 150
65 FOR R=15 TO 0 STEP -.7         :rem 232
75 POKE 54296,R:NEXT              :rem 156
116 REM                            :rem 124
117 REM ---- DOWNWARD SAW STROKE ---- :rem 14
118 REM                            :rem 126
120 POKE 54273,65                  :rem 94
150 FOR R=0 TO 15 STEP .6         :rem 229
160 POKE 54296,R:NEXT              :rem 199
165 FOR R=15 TO 0 STEP -.6        :rem 24
175 POKE 54296,R:NEXT              :rem 205
190 NEXT                           :rem 217
216 REM                            :rem 125
217 REM ---- TURN SOUND OFF ----   :rem 147
218 REM                            :rem 127
220 POKE 54276,128:POKE 54296,0   :rem 97

```

This difference in speed would correspond with the way a saw is used, since the saw's cutting edges are used more on the down stroke. Since the cutting stroke would apply more resistance to the saw, it would be slower. That's why the timing loop in lines 150 and 165 is longer. The smaller STEP value gives that part of the sound a longer loop.

The ocean. Similar to Program 3-26, this routine recreates the sound of the ocean by slowing the effect.

Program 3-27. Ocean Waves

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM ---- CLEAR SOUND REGISTERS ---- :rem 41
6 REM                                  :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT :rem 24
11 POKE54275,8:POKE54278,240         :rem 49
12 POKE 54276,129                    :rem 98
13 REM                                :rem 72
14 REM -- NUMBER OF WAVES LOOP --    :rem 27
15 REM                                :rem 74
16 FOR L=0 TO 15                      :rem 16
17 REM                                :rem 76
18 REM ---- WAVE ROUTINE ----        :rem 14
19 REM                                :rem 78
20 POKE 54273,75                      :rem 46
40 IW =RND (0) * .25+.05              :rem 35
45 OW =RND (0) * .02+.01             :rem 37
50 FOR R=1 TO 15 STEP IW              :rem 241
60 POKE 54296,R:NEXT                  :rem 150
65 FOR R=15 TO 1 STEP -OW             :rem 42

```



```
75 POKE 54296,R:NEXT           :rem 156
76 NEXT                        :rem 172
77 REM                          :rem 82
78 REM ---- TURN OFF SOUND ---- :rem 104
79 REM                          :rem 84
80 POKE 54276,128:POKE 54296,0 :rem 53
```

To make the sound more realistic, the length of the incoming and outgoing waves is varied by the RND functions in lines 40 and 45. These values then become a part of the STEP command in the timing loops found in lines 50 and 65. The rest of the program should be familiar to you. The soft sound is created by increasing and decreasing the volume level; the noise waveform is enabled by POKEing location 54276 with 129; and the pitch value is set in line 20. You could simulate faster waves by altering the values in lines 40 and 45, changing them to $IW=RND(0)*.4+.1$ and $OW=RND(0)*.05+.02$, for instance.

Tapered Sounds

Tapered sounds can be defined as sounds that begin at one end of the volume scale and then slowly rise (or fall) to the other end. For example, a rising tapered sound would begin quietly and gradually attain full volume. However, it doesn't fall back to minimum volume, as does a soft sound. It remains at that high volume level.

The simplest way to program a tapered sound is to use the volume control register, as you did when you created soft sounds. Although you can create tapered sounds using the attack, decay, and release registers, it becomes difficult at times, especially when you're dealing with falling tapered sounds. You can produce a tapered sound by setting the attack and decay to a slow rate, then setting the release to a fast rate, but coordinating all three may prove complicated. It's almost impossible to produce a falling tapered sound without some use of the volume control register. And no matter how hard you try, the sounds may not be just right. We'll stick with the volume control register in the examples, to keep things as simple as possible.

Program 3-28. Rising Tapered Sound

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM --- INITIALIZE SOUND REGISTERS --      :rem 45
6 REM                                          :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT        :rem 24
11 POKE 54275,8:POKE 54277,0:POKE 54278,240  :rem 255
12 POKE 54272,0:POKE 54273,45               :rem 245
16 REM                                          :rem 75
17 REM -TAPERED, RISING PULSE WAVE-         :rem 95
18 REM                                          :rem 77
30 POKE 54276,65                             :rem 49
50 FOR R=0 TO 30                             :rem 17
60 POKE 54296,R/2:NEXT                       :rem 247
70 FOR R=0 TO 500:NEXT                       :rem 190
80 POKE 54276,64                             :rem 53
116 REM                                       :rem 124
117 REM -TAPERED, RISING TRIANGLE WAVE-    :rem 93
118 REM                                       :rem 126
130 POKE 54276,17                           :rem 95
150 FOR R=0 TO 30                           :rem 66
160 POKE 54296,R/2:NEXT                     :rem 40
170 FOR R=0 TO 500:NEXT                     :rem 239
180 POKE 54276,16                           :rem 99
216 REM                                       :rem 125
217 REM -TAPERED, RISING SAWTOOTH WAVE-    :rem 129
218 REM                                       :rem 127
230 POKE 54276,33                           :rem 94
250 FOR R=0 TO 30                           :rem 67
260 POKE 54296,R/2:NEXT                     :rem 41
270 FOR R=0 TO 500:NEXT                     :rem 240
280 POKE 54276,32                           :rem 98
316 REM                                       :rem 126
317 REM -TAPERED, RISING NOISE WAVE-       :rem 135
318 REM                                       :rem 128
330 POKE 54276,129                          :rem 149
350 FOR R=0 TO 30                           :rem 68
360 POKE 54296,R/2:NEXT                     :rem 42
370 FOR R=0 TO 500:NEXT                     :rem 241
380 POKE 54276,128                          :rem 153
416 REM                                       :rem 127
417 REM ---- TURN SOUND OFF ----           :rem 149
418 REM                                       :rem 129
420 POKE 54276,128:POKE 54296,0            :rem 99

```

Of course, a falling tapered sound would begin loudly and taper off to silence. Changing four lines in Program 3-28 will let you hear a falling tapered sound. The changes are:

```
50 FOR R=30 TO 0 STEP -1
150 FOR R=30 TO 0 STEP -1
250 FOR R=30 TO 0 STEP -1
350 FOR R=30 TO 0 STEP -1
```

Now the sounds begin at the highest volume level and decrease slowly through the FOR-NEXT loop using the STEP -1 command.

Tapered Pulse Waveforms

As mentioned earlier, there are two different kinds of tapered sounds—those that rise and those that fall. Let's look at rising tapered sounds first.

Accordion. Most musical instruments produce sounds that have a falling tapered sound. That's because they produce sound by making some element vibrate. As the vibrations die down, the sound slowly tapers off.

Accordions use vibrating elements also. The elements are steel reeds. But because the accordion plays the reeds by pumping air across them, the sounds start slowly as the accordionist begins pushing the bellows together, and then gradually rise in volume.

When the accordionist stops pushing the bellows, the air stops. Because the reeds are fairly rigid, they stop vibrating quickly, making the sound stop just as fast. Program 3-29 demonstrates a rising tapered sound, simulating an accordion.

Program 3-29. Accordion

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24
11 POKE 54275,3:POKE 54277,0:POKE54278,240:rem 250
12 POKE 54276,65                               :rem 49
13 REM                                           :rem 72
14 REM ---- ACCORDION MUSIC DATA ----         :rem 222
15 REM                                           :rem 74
16 DATA 33,134,2,33,134,4,44,191,1,44,191,1,42,60,
    1,42,60,4,0,0,2,29,221,2,29                :rem 151
17 DATA 221,4,37,161,1,37,161,2,33,134,4     :rem 156
26 REM                                           :rem 76
27 REM ---- MUSIC LOOPS ----                   :rem 195
28 REM                                           :rem 78
30 FOR G=0 TO 11                               :rem 3
```

```

117 REM                               :rem 125
118 REM ---- PLAY ACCORDION ---      :rem 129
119 REM                               :rem 127
120 READ H,L,T                        :rem 239
125 POKE 54272,L:POKE 54273,H        :rem 37
150 FOR R=0 TO 15 STEP (.5/T)       :rem 184
160 POKE 54296,R:NEXT                :rem 199
175 POKE 54296,0                     :rem 50
197 REM                               :rem 133
198 REM ---- REPEAT MUSIC LOOP ---- :rem 106
199 REM                               :rem 135
200 NEXT                              :rem 209
210 RESTORE:FOR T=0 TO 1000:NEXT:GOTO 30 :rem 76

```

This program is similar to the routine which creates a violin's sound, inasmuch as it reads pitch values from DATA and plays the notes one at a time. Of course, the sounds are quite different. Note that this program uses an asymmetrical pulse waveform, the pulse width set by the value POKEd into memory location 54275. The DATA also includes a timing function for each note played. The DATA is arranged in groups of three numbers, the third being the timing variable. This lets the program play notes of any length.

Piano. Pianos are similar to accordions in that they both produce asymmetrical pulse waveform sounds. However, pianos create waves somewhat more symmetrical, and the sound tapers from loud to soft. Instead of the tapered sound rising, then, Program 3-30 illustrates a falling tapered sound.

Program 3-30. Piano Player

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS -- :rem 0
6 REM                                  :rem 26
10 FOR R=54272 TO 54296:POKE R,0:NEXT :rem 24
11 POKE 54275,7:POKE 54277,0:POKE 54278,240:F=24
                                           :rem 33
12 POKE 54276,65                          :rem 49
13 REM                                  :rem 72
14 REM ---- PIANO MUSIC DATA ----      :rem 195
15 REM                                  :rem 74
16 DATA 33,134,4,33,134,4,33,134,4,33,134,2,33,134
    ,4,25,29,2,28,45,2,21,30,2          :rem 110
17 DATA 25,29,2,28,48,2,0,0,2,16,195,4,18,208,2,21
    ,30,2,16,195,2,18,208,2,21,30      :rem 6
18 DATA 2,16,195,2,18,208,2,21,30,2,16,195,2,18,20
    8,2,0,0,2,16,195,8,33,134,2      :rem 167

```

```

26 REM                               :rem 76
27 REM ---- MUSIC LOOPS ----        :rem 195
28 REM                               :rem 78
29 FOR J=0 TO 1                      :rem 221
30 FOR G=0 TO F                      :rem 231
117 REM                              :rem 125
118 REM ---- PLAY PIANO ----        :rem 147
119 REM                              :rem 127
120 READ H,L,T                      :rem 239
125 POKE 54272,L:POKE 54273,H       :rem 37
150 FOR R=15 TO 0 STEP -(1.5/T)     :rem 22
160 POKE 54296,R:NEXT               :rem 199
175 POKE 54296,0                    :rem 50
197 REM                              :rem 133
198 REM ---- REPEAT MUSIC LOOP ---- :rem 106
199 REM                              :rem 135
200 NEXT                            :rem 209
211 RESTORE:F=F-1                   :rem 25
212 NEXT                             :rem 212
217 REM                              :rem 126
218 REM ---- TURN OFF SOUND ----    :rem 148
219 REM                              :rem 128
300 POKE 54276,64:POKE 54296,0     :rem 47

```

Another interesting feature of this program is the method used to READ the pitch values from the DATA statements in lines 16–18. Initially, the READ loop starting in line 30 READs 25 notes (0–24) because the variable F is given a maximum value of 24 in line 11. After the music has been READ once, however, the program decrements F (subtracts one from it), and the READ routine in line 30 READs only 24 notes. This was done because the final note in the DATA statements is a transitional note and should not be played the second time through the tune.

You can tell that this is demonstrating a falling tapered sound just by looking at line 150, which establishes the variable R. This variable is used in the next line to set the volume control register.

Harpsichord. Harpsichords are similar to pianos in the way they are played, but the sound they produce is made by plucking rather than striking strings. This produces a much sharper tone which can be achieved by making the pulse waveform more asymmetrical.

Since this is so much like Program 3-30, you can simply insert the new lines into that program to hear a harpsichord-like sound. The lines are:

```

11 POKE 54275,15:POKE 54277,0:POKE 54278,240           :rem 45
14 REM ----- HARPSICHORD MUSIC DATA -----         :rem 131
16 DATA 42,60,1,33,134,1,37,161,1,42,60,1,50,58,1,    :rem 72
   44,191,1,44,191,1,56,97,1
17 DATA 50,58,1,50,58,1,67,12,1,63,72,1,67,12,1,50   :rem 10
   ,58,1,42,60,1,33,134,1,37,161
18 DATA 1,42,60,1,44,191,1,50,58,1,56,97,1,50,58,1  :rem 123
   ,44,191,1,42,60,1,37,161,1
19 DATA 42,60,1,33,134,1,31,164,1,33,134,1,37,161,  :rem 156
   1,25,29,1,31,164,1,37,161,1
20 DATA 44,191,1,42,60,1,37,161,1                   :rem 64
30 FOR G=0 TO 35                                       :rem 9
118 REM ----- PLAY HARPSICHORD -----              :rem 83
150 FOR R=15 TO 0 STEP -(.5/T)                        :rem 229

```

Tapered Triangle Waves

Because they produce softer sounds, triangle waves create very smooth, tapered sounds. This first group of examples demonstrates some rising tapered sounds using triangle waveforms.

Clinking a glass with a spoon. This is the simplest kind of tapered sound. The program uses a single loop to control the length and volume of the sound.

Program 3-31. Clinking

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --                 :rem 0
6 REM                                                  :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT                 :rem 24
12 POKE 54275,1:POKE 54277,0:POKE 54278,240         :rem 249
13 POKE 54273,150:POKE 54276,17                     :rem 95
20 REM                                                  :rem 70
21 REM ----- CLINK LOOP -----                   :rem 90
22 REM                                                  :rem 72
23 FOR C= 0 TO 2                                       :rem 209
25 FOR R= 15 TO 0 STEP -.5                             :rem 226
30 POKE 54296,R:NEXT                                  :rem 147
35 NEXT                                                :rem 167
40 REM                                                  :rem 72
41 REM ----- TURN SOUND OFF -----               :rem 94
42 REM                                                  :rem 74
50 POKE 54276,16:POKE 54296,0                       :rem 254

```

You can use the attack/decay control register to produce the tapered sound, instead of a FOR-NEXT loop which decreases the volume level. Look at Program 3-32 to see an example of this.

Program 3-32. Clinking with Attack/Decay

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24
11 POKE 54296,15                               :rem 45
12 POKE 54275,1:POKE 54277,6:POKE54278,6     :rem 159
13 POKE 54273,150:POKE 54276,17              :rem 95
20 REM                                           :rem 70
21 REM ---- CLINK LOOP ----                   :rem 90
22 REM                                           :rem 72
23 FOR C=0 TO 2                                :rem 209
24 POKE 54276,17                               :rem 49
25 FOR R=0 TO 15 STEP .1:NEXT                 :rem 42
30 POKE 54276,16                               :rem 45
35 NEXT                                         :rem 167
40 REM                                           :rem 72
41 REM ---- TURN SOUND OFF ----               :rem 94
42 REM                                           :rem 74
50 POKE 54276,16:POKE 54296,0                 :rem 254

```

Notice that for this example to work properly, the decay and release values must be the same. That's why both location 54277 and location 54278 have the value 6 POKEd into them.

Tapered Sawtooth Sounds

Sawtooth sounds are sharp and lend themselves to alarmlike sounds and similar dramatic effects.

Foghorn. To make this foghorn sound, we'll use the sawtooth waveform and play two different pitches. Notice that the first pitch tapers the sound approximately halfway down the volume level, and the second tone takes the sound the rest of the way to minimum volume as the value of R goes to 0 (line 47).

Program 3-33. Foghorn

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24

```

```

12 POKE 54275,1:POKE 54277,0:POKE 54278,240      :rem 249
13 POKE 54273,5:POKE 54272,200                  :rem 36
14 POKE 54276,33                                  :rem 46
20 REM                                             :rem 70
21 REM ---- FOG HORN UPPER TONE ----            :rem 132
22 REM                                             :rem 72
23 FOR X=0 TO 3                                    :rem 231
25 FOR R=15 TO 9 STEP -.1                         :rem 231
30 POKE 54273,5:POKE 54272,200                  :rem 35
35 POKE 54296,R:NEXT                              :rem 152
40 REM                                             :rem 72
41 REM ---- FOG HORN LOWER TONE ----            :rem 131
42 REM                                             :rem 74
45 FOR R=9 TO 0 STEP -.1                         :rem 179
46 POKE 54273,3                                    :rem 253
47 POKE 54296,R:NEXT                              :rem 155
49 NEXT                                           :rem 172
50 REM                                             :rem 73
51 REM ---- TURN SOUND OFF ----                  :rem 95
52 REM                                             :rem 75
60 POKE 54276,32:POKE 54296,0                   :rem 253

```

Tapered Noise

The noise waveform can produce sounds as varied as your imagination can make them. Since it takes on totally different characteristics when you change its frequency or length, it's one of the most versatile waveforms available to you. Below is an example of how you can use the noise waveform to create a tapered sound.

Walking in the snow. For every step you take in the snow, you crush down thousands of snowflakes. That crunching sound can be recreated by using the noise waveform.

Program 3-34. Crunching Snow

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

6 REM -- INITIALIZE SOUND REGISTERS --           :rem 1
10 FOR R=54272 TO 54296:POKER,0:NEXT            :rem 24
11 POKE 54277,0:POKE 54278,240                  :rem 43
12 POKE 54273,85                                  :rem 48
13 POKE 54276,129                                 :rem 99
18 REM                                             :rem 77
19 REM ---- STEPS THROUGH SNOW ----             :rem 173
20 REM                                             :rem 70
30 FOR H=0 TO 10                                  :rem 3
65 FOR R=0 TO 10 STEP .5                         :rem 180

```



```
75 POKE 54296,R:NEXT           :rem 156
80 POKE 54296,0                :rem 253
82 FOR W=0 TO 750:NEXT         :rem 205
85 NEXT                         :rem 172
216 REM                         :rem 125
217 REM ----- TURN SOUND OFF ----- :rem 147
218 REM                         :rem 127
220 POKE 54276,128:POKE 54296,0 :rem 97
```

Changing the pitch value in line 12, or the maximum volume possible in line 65, or the length of the sound in line 82 can change the sound. Creating a falling tapered sound is simple: Just reverse the FOR-NEXT loop in line 65 so that it reads FOR R=10 TO 0 STEP -.5.

More Advanced Sound Techniques

You've seen how to create sounds using the SID chip on the Commodore 64. Hard sounds, soft sounds, and tapered sounds are just some of the styles of effects you've seen how to produce. And now you have a small library of sound effects ready to use. But there's more to the SID chip. There are such things as filters, ring modulation, and synchronization available to you. The advanced sound controls can add even more to the sounds and music you'd like to produce on your computer. The next chapter will show you how to use these controls as you create impressive sound effects and music.



CHAPTER

4

Advanced
Functions



4

Advanced Functions

The first three chapters introduced you to the basic functions of the Commodore 64 Sound Interface Device (SID). You saw how to turn the various sound control registers on and off, how to produce different tones with those sound registers, and also how to alter the nature of those tones by switching waveforms. The two sound editors gave you tools to create your own music. You also saw how to program effects to produce sounds ranging from imitation musical instruments to duplicating nonmusical devices such as motors and saws.

But there are other tools of the SID chip available. Filters, ring modulation, and synchronization are some of those tools. They can help you create even more impressive sounds and musical pieces on the 64. This chapter reviews some of the SID functions already covered, and includes routines and examples that will help you better utilize them. You'll also see how to use the advanced sound controls and functions. We'll put them to good use in the last chapter.

The Initialization Process

One thing you may have noticed is that at the beginning of every sound program in this book, two or three program lines are used to *initialize* the sound registers. The SID chip's sound control registers are cleared, and then some of them are POKed with values. Chapter 1 gave you an idea of the order in which to set these registers when you want to produce sounds. The sections that follow further explain the operation of each of the basic sound functions, how to use them more efficiently, and what may happen if you do not program them correctly.

Pitch control registers. The pitch control registers consist of two memory locations for each voice. The values in each of these pairs of memory locations are combined by the SID and are treated as a single value. You'll remember that the values of the bits turned on in the upper control register are multiplied by 256, then added to the values of those bits in the lower control register which are on. This total is the number the SID chip uses to select a pitch or frequency. The values for each of the 16 bits in the two registers are detailed by Figure 4-1.

Figure 4-1. Pitch Register Bit Values

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Upper Control Register
(Normal bit values [1-128] multiplied by 256)

Lower Control Register

The memory locations for the pitch control registers are:

- Voice 1: Low Pitch Register **54272**
- Voice 1: High Pitch Register **54273**
- Voice 2: Low Pitch Register **54279**
- Voice 2: High Pitch Register **54280**
- Voice 3: Low Pitch Register **54286**
- Voice 3: High Pitch Register **54287**

Table 1-3, in Chapter 1, shows the pitch values for a standard musical scale, including sharps and flats. However, you may find that you need to produce some pitches or frequencies not listed in the table. Since every frequency you can produce on the 64 has a corresponding value between 0 and 65,535, it's possible to calculate the necessary value from the frequency you want. The formula for determining the total 16-bit value for any given frequency is:

$$\text{Value} = \text{Frequency} / .06097$$

Of course, you can't POKE this number directly into the upper and lower register locations. The largest number that any single

location can hold is 255. To calculate the value for any frequency available through the SID chip (0–3995 hertz) and split it into the two values you *can* use, RUN this short routine. Enter a frequency from 0 to 3995, and the upper (U) and lower (L) pitch values will display on the screen. The note will play, using the sawtooth waveform, until you press any key.

Program 4-1. Frequency Calculation

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

2 FOR T=54272 TO 54272+24:POKET,Ø:NEXT:POKE54296,1
  5:POKE54277,49:POKE54278,248           :rem 197
1Ø PRINT"{CLR}":INPUT "FREQUENCY"; F      :rem 6
15 N=INT(F/.Ø6Ø97)                       :rem 214
2Ø U=INT(N/256)                          :rem 74
3Ø L=N-(U*256)                           :rem 212
4Ø PRINT"U="U,"L="L:POKE54272,L:POKE54273,U:POKE54
  276,33                                  :rem 56
45 PRINT"{2 DOWN}{WHT}PRESS ANY KEY TO RUN AGAIN
  [?]":                                     :rem 81
5Ø GETA$:IF A$="" THEN GOTO 5Ø           :rem 38
55 POKE54276,32                          :rem 5Ø
6Ø GOTO 1Ø                                :rem Ø

```

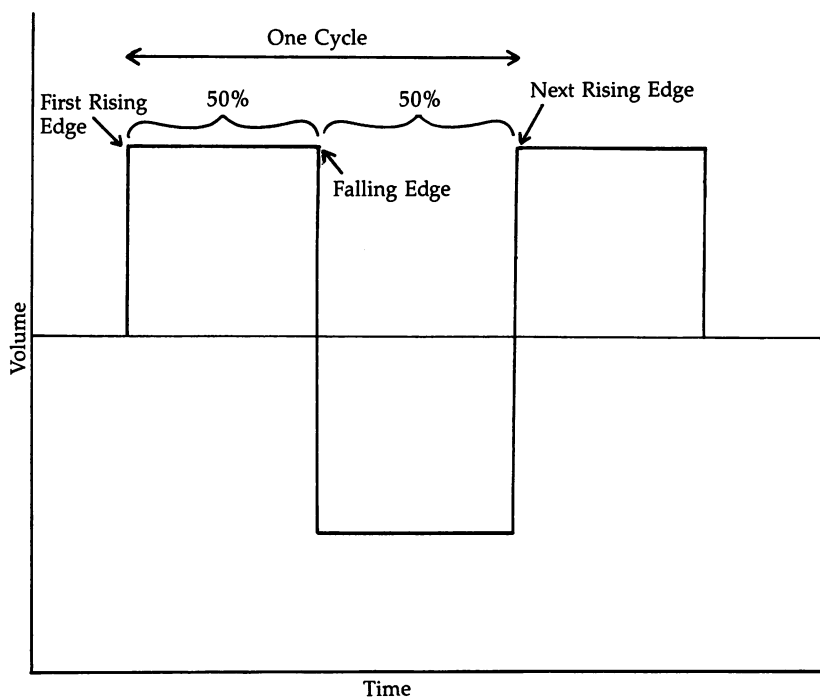
You can use this program to calculate the pitch values for the two registers, at the same time hearing the note you're selecting. If you compare the values displayed with those in Table 1-3, you'll notice that there are often small differences. If you ask for the pitch values of the frequency at 16 hertz, for instance, you'll see a high pitch register value of 1 and a low pitch register value of 6 on the screen. This is slightly different from the high value of 1 and the low value of 12 which show in Table 1-3. Even though the value in the low pitch register is different, the SID chip plays the bottom C note. The difference is just too slight for you to hear anything but that note.

Pulse width control registers. The frequency of a sound wave can be determined by measuring the time the sound wave begins its rising edge on one wave until it begins rising on the next. Figure 4-2 shows the rising and falling edges of one cycle of a pulse wave.

As you can see from Figure 4-2, the wave has a falling edge between the two rising edges. In this figure, the falling edge is exactly in the middle of the wave. We call this a 50 percent pulse wave because the signal falls at the halfway

point of the wave. This proportion of pulse widths produces the loudest and clearest pulse waveform tone. By making the pulse waves more or less than 50 percent, you'll hear a sound that becomes thinner and more tinny. Program 4-2 lets you hear the effect that changing pulse width values can create in a sound. As the program creates sound, it displays the register values for the various pulse widths.

Figure 4-2. Rising and Falling Edges



Program 4-2. Changing Pulse Widths

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 PRINT "{CLR}{2 DOWN}";           :rem 246
6 REM                               :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT :rem 24
11 POKE 54296,15                     :rem 45
12 POKE 54277,0:POKE 54278,240       :rem 44
13 POKE 54272,30:POKE 54273,20      :rem 34
16 POKE 54276,65                     :rem 53

```



```

17 REM                                     :rem 76
18 REM ---- PULSE LOOP ----              :rem 120
19 FOR P=0 TO 100 STEP 5                  :rem 179
20 N=INT(4096/100*P)                      :rem 54
21 U=INT(N/256)                           :rem 75
22 PRINT RIGHT$("{"2 SPACES}"+STR$(P)+"%",4);": ";
                                           :rem 249
23 L=N-(U*256)                             :rem 214
24 PRINT"LOW:";L,"HIGH:";U                :rem 68
28 REM ---- PULSE WIDTH DATA ----       :rem 217
29 REM                                     :rem 79
30 POKE 54274,L: POKE 54275,U             :rem 1
40 FOR R=0 TO 300: NEXT                   :rem 185
50 NEXT                                    :rem 164
57 REM                                     :rem 80
58 REM ---- TURN SOUND OFF ----          :rem 102
59 REM                                     :rem 82
60 POKE 54276,64:POKE 54296,0            :rem 2

```

By changing the value in the two-byte pulse width registers, it's possible to produce waves which pulse anywhere from 0 to 100 percent of the way through the wave cycle. The pulse width control registers are found in the following memory locations:

Voice 1: Pulse Width Low Register	54274
Voice 1: Pulse Width High Register	54275
Voice 2: Pulse Width Low Register	54281
Voice 2: Pulse Width High Register	54282
Voice 3: Pulse Width Low Register	54288
Voice 3: Pulse Width High Register	54289

Here's a short program that allows you to enter a percentage value for any pulse width between 0 and 100. It will then return the necessary POKE values for the pulse width control registers, as well as play a note using those values. Press any key to stop the note and to enter another pulse width percentage.

Program 4-3. Pulse Width Values

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

2 FOR T=54272 TO 54272+24:POKET,0:NEXT:POKE54296,1
  5:POKE54277,49:POKE54278,248              :rem 197
3 POKE 54272,135:POKE 54273,33              :rem 43
10 PRINT"{CLR}":INPUT"PULSE-WIDTH (PERCENTAGE VALU
   E)";P                                     :rem 64
15 N=INT(4096/100*P)                        :rem 58

```

```

20 U=INT(N/256)                                :rem 74
30 L=N-(U*256)                                :rem 212
40 PRINT "U="U, "L="L:POKE54274,L:POKE54275,U:POKE54
   276,65                                       :rem 65
45 PRINT "{2 DOWN}{WHT}PRESS ANY KEY TO RUN AGAIN
   [7]"                                         :rem 81
50 GET A$:IF A$="" THEN GOTO 50                :rem 38
55 POKE54276,64                                :rem 55
60 GOTO 10                                       :rem 0

```

Waveform control registers. The waveform control register has eight separate functions (only five of which we've used so far) that are enabled by turning appropriate bits on and off. The bits and their functions are detailed in Figure 4-3.

Figure 4-3. Waveform Control Register Bits

Bit	7	6	5	4	3	2	1	0
Bit Value	128	64	32	16	8	4	2	1

Noise Waveform	Pulse Waveform	Sawtooth Waveform	Triangle Waveform	Test Bit	Ring Modulation	Sync	Gate Bit
-------------------	-------------------	----------------------	----------------------	----------	--------------------	------	----------

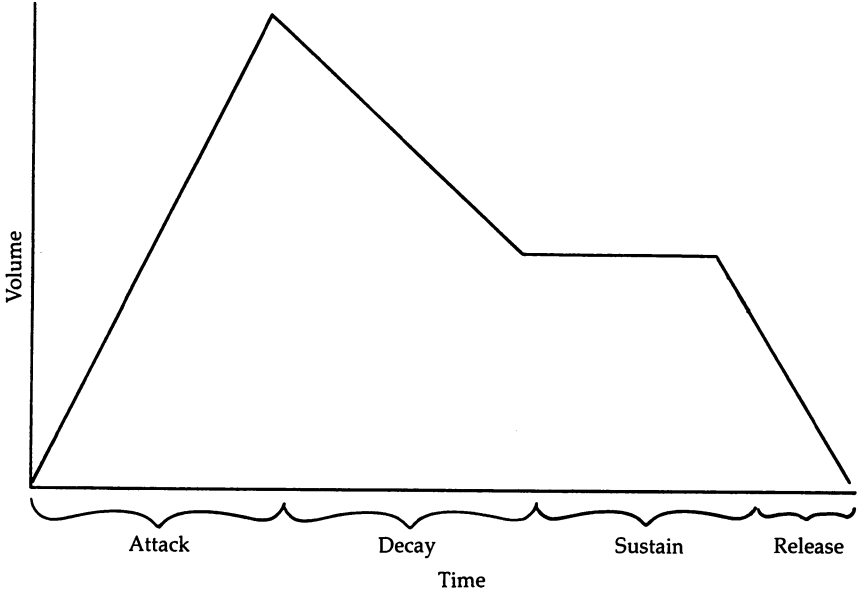
Remember that to turn the sound on, you also need to set the gate bit (bit 0) by adding 1 to the bit value for the waveform you're selecting. When you choose a waveform, then, you always POKE a total value of the waveform enabling bit, plus 1. For instance, enabling the noise waveform and turning the gate bit on requires a total value of 129 POKED into location 54276 for voice 1. Keep this in mind later in the chapter when the other functions of the three waveform control registers are explained.

The Envelope Generator

Throughout this book we've discussed the way that sound can be changed by modifying the sound envelope. Using the ADSR (Attack, Decay, Sustain, and Release), you can significantly change the "shape" of a sound. Highlighting each of these, and showing in more detail how they can be used, may be valuable.

A sound envelope, complete with the four elements, could look something like Figure 4-4.

Figure 4-4. Sound Envelope—ADSR



You'll find the (ADSR) functions for each voice in the following memory locations:

Voice 1: Attack Control Register	54277 bits 4-7
Voice 1: Decay Control Register	54277 bits 0-3
Voice 1: Sustain Control Register	54278 bits 4-7
Voice 1: Release Control Register	54278 bits 0-3
Voice 2: Attack Control Register	54284 bits 4-7
Voice 2: Decay Control Register	54284 bits 0-3
Voice 2: Sustain Control Register	54285 bits 4-7
Voice 2: Release Control Register	54285 bits 0-3
Voice 3: Attack Control Register	54291 bits 4-7
Voice 3: Decay Control Register	54291 bits 0-3
Voice 3: Sustain Control Register	54292 bits 4-7
Voice 3: Release Control Register	54292 bits 0-3

Attack. Attack refers to the length of time it takes the sound to reach maximum volume. If the attack rate is fast, the sound will start abruptly. If it is set to a slower rate, it simply takes longer for the sound to reach that maximum volume level.

Program 4-4. Attack Rate

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT            :rem 24
11 POKE 54296, 15                               :rem 45
12 REM                                           :rem 71
13 REM -- SET ATTACK RATE --                     :rem 204
14 REM                                           :rem 73
15 POKE 54277,0                                  :rem 250
16 REM                                           :rem 75
17 REM                                           :rem 76
18 REM -- SET OTHER REGISTERS --                 :rem 39
19 REM                                           :rem 78
20 POKE 54275,8:POKE54278,240                   :rem 49
21 POKE 54272,0:POKE 54273,21                   :rem 239
22 POKE 54276,65                                 :rem 50
23 REM                                           :rem 73
24 REM -- PLAY NOTE --                           :rem 106
25 REM                                           :rem 75
30 FOR D=0 TO 500:NEXT                           :rem 172
47 REM                                           :rem 79
48 REM -- TURN OFF SOUND --                       :rem 177
49 REM                                           :rem 81
50 POKE 54276,64:POKE 54296,0                   :rem 1

```

This program demonstrates a very rapid attack rate. Look at line 15. Zero is POKEd into the attack/decay control register at location 54277. This sets both attack and decay (which we'll get to in a moment) to the fastest possible rate. The sound rises to maximum volume very quickly.

If the attack is set to a slower rate, however, the sound will emerge gently. To hear this, replace line 15 in Program 4-4 with the following:

```
15 POKE 54277,160
```

You've just set the attack to a rate of 10. (Remember that you multiply that number by 16 to arrive at a value to POKE into the register.) Now the sound reaches maximum volume only after some time has passed. It's become a much *softer* sound. You can use the attack and decay of a sound to create hard or soft sounds, just as you earlier used the volume control.

Some of the programs in other chapters of this book used the volume control register to turn sounds on and off. When you're using the ADSR functions, you need to use the gate bit

to turn sounds on and off. It's the gate bit that initiates the ADSR of a sound. Whenever you manipulate the ADSR (which you will do anytime you create a sound on the 64), you should use the gate bit to turn the sound on and off. If you've been following the pattern of programming sound first established in Chapter 1, you're already doing that. Sometimes, in order to create just the type of sound that you want, you may use the volume control register to turn the sound on and off. Just be aware that the effect may sound different if you do that.

The speed at which a note increases in volume (attack) is determined by the value you POKE into the attack register. The attack rate can be any number between 0 and 15. Table 4-1 shows the POKE values and actual time that elapsed for each attack rate.

Table 4-1. Attack Times

Attack Rate	POKE Value	Time Elapsed
0	0	.002 sec
1	16	.008 sec
2	32	.016 sec
3	48	.024 sec
4	64	.038 sec
5	80	.056 sec
6	96	.068 sec
7	112	.080 sec
8	128	.100 sec
9	144	.250 sec
10	160	.500 sec
11	176	.800 sec
12	192	1.000 sec
13	208	3.000 sec
14	224	5.000 sec
15	240	8.000 sec

For example, if you wanted an attack that lasted 0.1 seconds you would use an attack rate of 8, but you would POKE 128 into the attack/decay register ($8 \times 16 = 128$).

There's one more thing you should keep in mind when you're setting an attack rate. The sound must last at least as long as the time it takes the attack to execute, or the sound will be cut off before it reaches maximum volume. You can

hear this happen if you change line 30 in Program 4-4. Make sure that line 15 still reads POKE 54277,160.

```
30 FOR D=0 TO 150:NEXT
```

Cutting the length of the sustain's FOR-NEXT delay loop cuts the sound before it can reach its peak volume. For the purpose of timing your sounds, a FOR T=0 TO 1000:NEXT loop lasts approximately one second. Add the two lines below to the revised Program 4-4 (the version with POKE 54277,160 as line 15). When you rerun the program, you'll be asked how long, in seconds, you want the sustain delay loop to last.

```
9 INPUT "HOW MANY SECONDS";S  
30 FOR R=0 TO S*1000:NEXT
```

Decay. Decay follows attack in the sound envelope. After a sound has attained maximum volume, it will decay (go down in volume) at a rate specified by the value in the attack/decay control register until it reaches the sustain volume level. The timing of decay is somewhat slower than the attack timing. Table 4-2 outlines the decay rates, the POKE values used, and the time elapsed to execute those rates.

Table 4-2. Decay Times

Decay Rate	POKE Value	Time Elapsed
0	0	.006 sec
1	1	.024 sec
2	2	.048 sec
3	3	.072 sec
4	4	.114 sec
5	5	.168 sec
6	6	.204 sec
7	7	.240 sec
8	8	.300 sec
9	9	.750 sec
10	10	1.500 sec
11	11	2.400 sec
12	12	3.000 sec
13	13	9.000 sec
14	14	15.000 sec
15	15	24.000 sec

Note that the decay rates and POKE values are identical. The decay value is POKEd into the lower half of the attack/decay register, so it's not multiplied by 16, *then* POKEd into the register, as is the attack rate. The decay rate *is* the POKE value. Add these three lines to the original version of Program 4-4. By POKEing location 54277, you're setting voice 1's decay to a rate of 9. (Changing line 20 to decrease the sustain volume level lets you hear the decay a bit easier.)

```
15 POKE 54277,9
20 POKE 54275,8:POKE54278,90
30 FOR D=0 TO 400:NEXT
```

If you want to produce a sound that uses both an attack *and* a decay, you add the POKE value for the attack rate to that of the decay rate, then POKE the sum into the register. In the lines below, which you can add to Program 4-4, you're setting attack to 10 and decay to 9. You also need to set the total delay time so that the entire sound is heard.

```
15 POKE 54277,169
20 POKE 54275,8:POKE54278,90
30 FOR D=0 TO 900:NEXT
```

Sustain. After the attack/decay portions of a sound envelope, the SID chip implements the sustain. This function is similar to the volume control. However, instead of determining the *absolute* volume of the sound, it's a relative level attained during the sound's attack. For example, if the maximum volume reached by the attack was 10 (as set by the volume control register in location 54296) and the sustain was set to 8 (half of the 16 possible levels), the resulting volume during the sustain section of the sound would be 5, or half of 10. Since the sustain level will be only a portion of the maximum volume attained during the attack, the volume of the sound can at best only remain at that level during the sustain. You'd have to set the sustain level to 15 to do that. Volume will *always* drop from the maximum unless the sustain is set to that level.

Replacing lines 20 and 30 of our latest version of Program 4-4 (the one with POKE 54277,169 as line 15) will set the sustain level to 8 (remember that the actual POKE value is the result of multiplying the level by 16). The replacement lines are:

```
20 POKE 54275,8:POKE 54278,128
30 FOR D=0 TO 1500:NEXT
```

You can hear the sound increase in volume through its attack, and then fall to the lower sustain level through its decay. Setting the sustain to an even lower value, such as 48, will make the distinction greater.

Release. Release is the last of the ADSR functions. It's initiated when the gate bit is turned off. Like attack and decay, release is a timing function. It determines how long it will take for the volume to drop from its sustain level to the lowest setting. To set the release rate to 9 and the sustain to 10, all you would need to do is change line 20 in the previous sound routine to:

```
20 POKE 54275,8:POKE 54278,169
```

When you use the release function, you need to ungate the sound without deselecting the waveform. In all of our examples, we did this by POKEing the waveform control register with the waveform bit value *only*. This is 1 less than the value used earlier in the routine, and so turns off the gate bit. Notice also that after the gate bit is turned off, the volume control register is set to 0. If you do not do this, there will be a very low volume residual sound produced until all of the sound registers have been cleared. Make sure that you've allotted enough time in the delay loop so that the release is heard. The elapsed time for various release rates is identical to the times for the decay. Refer to Table 4-2 for those times.

Using the Sound Envelope

By manipulating the ADSR of a sound's envelope, you can create distinctive effects, often by altering just one of the envelope's elements. Changing the shape of the envelope changes the sound. It's that simple. You can even create similar sounds by changing different portions of the ADSR. There's usually more than one way to produce a sound on the 64.

Here, for example, is a simple tone burst using the pulse waveform:

Program 4-5. Envelope Manipulation

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
7 REM                                           :rem 27
10 FOR R=54272 TO 54296:POKE R,0:NEXT           :rem 24
11 POKE 54296,15                               :rem 45
```



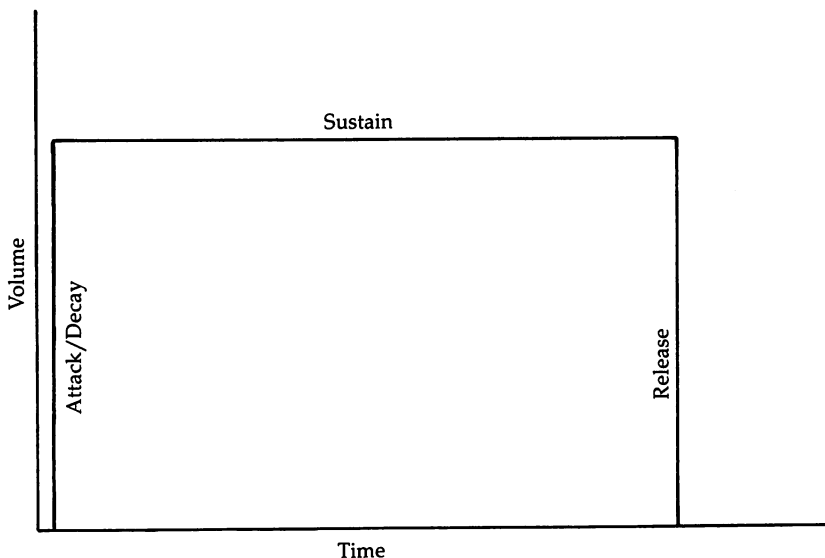
```

12 REM                                     :rem 71
13 REM -- SET A/D/S/R VALUES --         :rem 111
14 REM                                     :rem 73
15 POKE 54277,0:POKE 54278,240          :rem 47
16 REM                                     :rem 75
17 REM -- SET OTHER REGISTERS --         :rem 38
18 REM                                     :rem 77
19 POKE 54275,8                           :rem 4
20 POKE 54273,21                          :rem 37
21 POKE 54276,65                          :rem 49
31 REM                                     :rem 72
32 REM -- TONE TIMING LOOP --             :rem 53
33 REM                                     :rem 74
40 FOR R=0 TO 500:NEXT                   :rem 187
51 REM                                     :rem 74
52 REM -- TURN SOUND OFF --               :rem 172
53 REM                                     :rem 76
70 POKE 54276,64:POKE 54296,0           :rem 3

```

The sound envelope for this ADSR setting would look like Figure 4-5.

Figure 4-5. Tone Burst ADSR



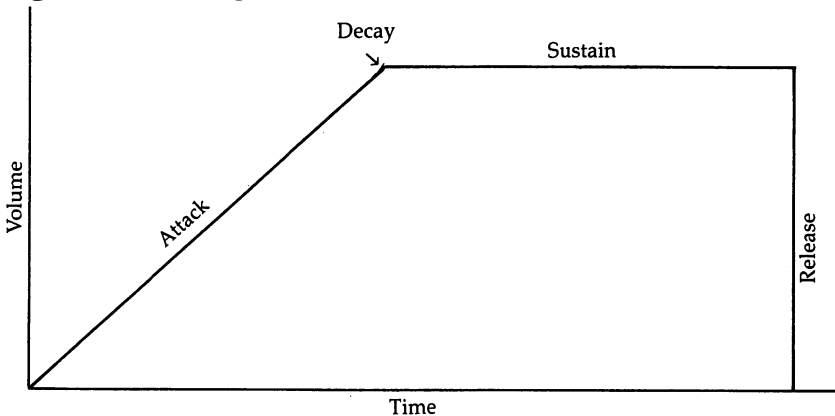
Since the attack, decay, and release are all set to 0, indicating the fastest possible rate, the only part of the envelope that's really present is the sustain level, which is set to its maximum by POKEing 240 into location 54278 in line 15.

If you want the sound to rise slowly, you simply increase the attack value. Add this line 15 in place of the original in Program 4-5 and reRUN the routine.

```
15 POKE 54277,160:POKE 54278,240
```

Now the envelope looks like Figure 4-6.

Figure 4-6. Longer Attack



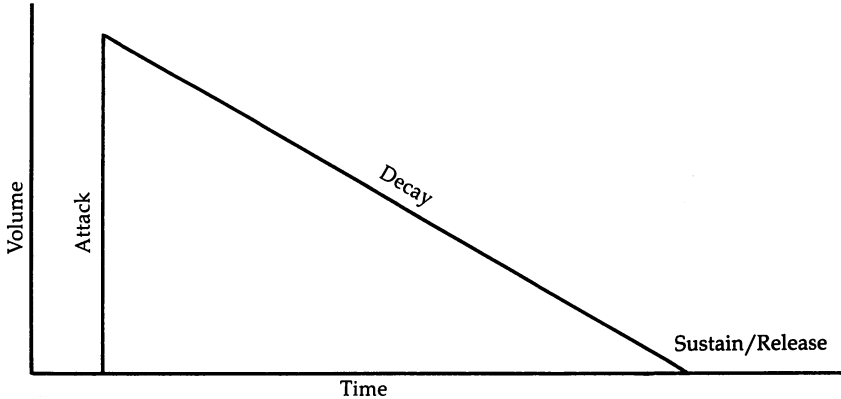
Attack has been set to a rate of 10 (as indicated by POKEing 160 [$10 \times 16 = 160$] into location 54277), but the decay and release have remained set to 0. Notice the difference—both in the sound itself and its illustration—from an attack set to the fastest rate.

Reducing the attack to 0 and increasing the decay gives you an entirely different kind of sound. For this example, the sustain has also been set to 0 level. Change line 15 in Program 4-5 to:

```
15 POKE 54277,9:POKE 54278,0
```

With this ADSR the envelope is like Figure 4-7.

Figure 4-7. Decay Increased



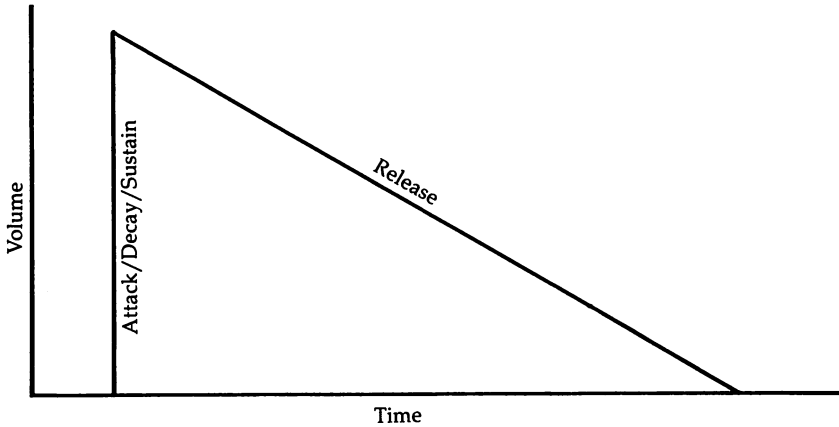
With this kind of sound, you have to be careful to allow enough time for the sound to shut off during the decay. If you are producing programs that have some complex timing, this can cause problems. In those instances, you may find another method easier to use. Add the following changed and new lines to Program 4-5:

```
15 POKE 54277,0:POKE 54278,9
32 REM -- NO TIMING LOOP --
39 POKE 54276,64
70 POKE 54296,0
```

Now the routine is not timed by a loop. It simply sets the sound to full volume and immediately turns the gate bit off, letting the release function "decay" the sound to 0. The final timing loop in line 40 is retained because the sound register maintains a slight residual sound after the release if the registers are not cleared.

Setting the ADSR to these values creates an envelope that looks like Figure 4-8. Notice how similar it is to the previous figure, where decay was used instead.

Figure 4-8. Using Release

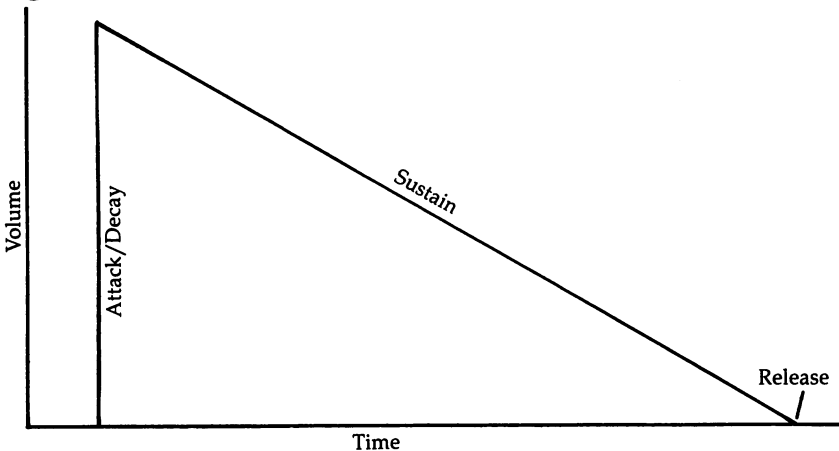


A third way you can produce this same sound is by dynamically manipulating the value in the sustain/release register. Using a FOR-NEXT loop, you can POKE several values into the register, changing the sustain volume setting. Add these two lines to Program 4-5 and reRUN it:

```
40 FOR R=240 TO 0 STEP -3  
41 POKE 54278,R:NEXT
```

We begin with the maximum value in the sustain register and then, using a loop in lines 40 and 41, decrease the value until it's 0. If you looked at the envelope now, it would look like Figure 4-9.

Figure 4-9. Sustain Manipulation



Multivolume sounds. Another feature that makes the sustain function so valuable is that it can be used to produce sounds that are lower in volume than the maximum set by the volume control register. An example of how this might be done is shown in the program below. By changing the value in the sustain register, you can make the sound for any of the registers either louder or softer. By using this feature, you can produce two tones that are set to different sustain levels and therefore different volumes.

Program 4-6. Multivolume

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
7 REM                                           :rem 27
10 FOR R=54272 TO 54296:POKE R,0:NEXT           :rem 24
11 POKE 54296,15                               :rem 45
12 REM                                           :rem 71
13 REM -- SET INITIAL A/D/S/R VALUES --       :rem 121
14 REM                                           :rem 73
15 POKE 54277,0:POKE 54284,0                   :rem 198
16 POKE 54278,240:POKE 54285,240              :rem 149
17 REM                                           :rem 76
18 REM -- SET OTHER VALUES --                 :rem 63
19 REM                                           :rem 78
20 POKE 54273,16:POKE 54272,195               :rem 97
21 POKE 54275,8:POKE 54282,8                   :rem 207
22 POKE 54280,21:POKE 54279,30                :rem 40
30 POKE 54276, 65: POKE 54283, 65              :rem 55
31 REM                                           :rem 72
32 REM -- VOLUME TIMING LOOP --                 :rem 215
33 REM                                           :rem 74
37 FOR R=240 TO 0 STEP -1                       :rem 227
38 POKE 54285,R                                 :rem 32
47 NEXT                                         :rem 170
51 REM                                           :rem 74
52 REM -- TURN SOUND OFF --                     :rem 172
53 REM                                           :rem 76
54 POKE 54276,64:POKE 54283,0                 :rem 1

```

The routine begins with two harmonizing tones and then, by reducing the value in the sustain register, gradually turns one of the tones off while maintaining the other tone at full volume. You may also note that by reducing the value in the sustain/release register (decrementing by 1), you're changing the value in the release register. Since you're not using the release function in this routine, those values aren't used. In

this application, decrementing by 1 is used to produce a longer tone.

As you've seen, there are usually several ways available when you want to produce a specific sound on the Commodore 64. But the ADSR isn't the only control you can use. Other, more advanced functions allow you to do even more with the SID chip.

Test Bit

The test bit, bit 3 of each voice's waveform control register, is a special bit that turns the output from the register *off* when it's turned *on* by POKEing 8 (value of bit 3) to the memory location. Ordinarily it's used to test the Commodore 64's SID chip circuits. However, it can be used for special sound functions because it produces a soft on/off sound. To get a better idea of what this means, you'll need to listen to a sound that is first turned on and off by the volume control. This produces a sound that clicks as it switches on or off. If you use the test bit instead, this clicking will be greatly reduced. To reduce it even further, you can use the gate bit, turning that on and off rapidly to produce the sound. As we noted earlier, there are usually several ways to program a sound effect on the 64.

Program 4-7. Test Bit On/Off

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT             :rem 24
11 POKE 54275,8:POKE 53277,0:POKE 54278,240
                                           :rem 254
12 POKE 54272,47:POKE 54273,25                   :rem 46
13 POKE 54276,65                                 :rem 50
21 REM                                           :rem 71
22 REM -- TURNING SOUND ON/OFF USING --         :rem 217
23 REM -- THE VOLUME CONTROL REGISTER -        :rem 15
24 REM                                           :rem 74
25 FOR M=0 TO 30                                 :rem 14
50 POKE 54296,15                                 :rem 48
60 FOR R=0 TO 50:NEXT                             :rem 141
70 POKE 54296,0:NEXT                             :rem 117
80 FOR N=0 TO 200:NEXT                           :rem 184
81 POKE 54296,15                                 :rem 52
82 REM                                           :rem 78
83 REM -- TURNING SOUND ON/OFF USING --         :rem 224
84 REM -- THE TEST CONTROL REGISTER ---        :rem 216

```

```

85 REM                               :rem 81
95 FOR M=0 TO 30                     :rem 21
96 POKE 54276,65                     :rem 61
98 FOR R=0 TO 50:NEXT                :rem 152
99 POKE 54276,73:NEXT                :rem 184
100 FOR N=0 TO 200:NEXT               :rem 225
101 REM                               :rem 118
102 REM -- TURNING SOUND ON/OFF USING -- :rem 8
103 REM -- THE GATE BIT CONTROLS --   :rem 129
104 REM                               :rem 121
105 FOR M=0 TO 30                     :rem 61
106 POKE 54276,65                     :rem 101
107 FOR R=0 TO 50:NEXT                :rem 191
108 POKE 54276,64:NEXT                :rem 223
109 POKE 54296,0                       :rem 47

```

Lines 95–100 contain the test bit on/off process. The rest of the program is similar to other routines you’ve already seen. The waveform and gate bit are enabled in line 96, then the test bit is turned *on* in line 99 by POKEing 73 to the register. The test bit’s value is 8, so you simply add that to the previous value (65) to turn the bit on. The FOR-NEXT loop then turns the test bit *off* by again POKEing the location with 65. Repeating this process quickly turns the sound on and off in a new way.

You may want to use other methods of turning sound on and off in your own routines, but the test bit technique can come in handy as you’re creating specific sounds. Perhaps you want a bit of a click as the sound is turned on and off—not as much as when you use the volume control register, but more than when you use the gate bit function. In that case, the test bit may work just right.

Additive Synthesis

Creating relatively simple sound effects by using one voice or by switching rapidly between two voices is just one method of creating sounds on the Commodore 64. You have other techniques available that can create some amazing sounds quite different from what you’ve heard so far. One such technique is called *additive synthesis*.

Additive synthesis, although probably a new term to you, is really very simple. It takes two sounds, usually produced by two voices on the 64, and brings them together to form a totally new sound. It *adds* sounds to create a unique *synthesis*

of the two. Both ring modulation and synchronization on the Commodore 64 are examples of additive synthesis.

Ring modulation. Ring modulation is a form of additive synthesis that significantly changes the tone quality of two sounds. Sounds created with ring modulation don't retain their original pitches. Instead, the sums and remainders of the two frequencies are kept. For example, if the first sound is a tone that vibrates at 500 vibrations per second (vps), and the second tone vibrates at 300 vps, then the ring modulated tone is a combination of the sum (800 vps) and the difference (200 vps). It would be a sound whose tone vibrates at 1000 vps.

Most of the time, ring-modulated sounds are very different from the original tones. In fact, ring modulation can create some of the most interesting and unusual sounds possible on your computer. To use ring modulation on the 64, you have to set bit 2 of the waveform control register (refer to Figure 4-3 for an illustration of one of the three registers), at the same time enabling the triangle waveform. To do this, just add 4 (bit 2's value) to the value you'd normally POKE to turn on the gate bit and enable the triangle waveform. In other words, you'd POKE location 54276 with 21 for voice 1 (1 for the gate bit, 4 for ring modulation, and 16 for the triangle waveform). Next, you have to select a frequency for voice 3 by POKEing a number into one of the two pitch control registers. That's all you have to do with voice 3; you don't have to set voice 3's waveform, ADSR, or any other parameters.

The effect of ring modulation is most apparent when the tones are mixed as they sweep the scales. You can do this in one of three ways. The first way, shown in Program 4-8, selects a frequency for the triangle waveform and sets the pitch registers to that frequency. Then voice 3's frequency is rapidly changed by sweeping the possible values in the high pitch control register.

Program 4-8. Ring Modulation—Sweeping Voice 3

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
0 FOR R=54272 TO 54296:POKER,0:NEXT             :rem 24
11 POKE 54296,15                               :rem 45
12 POKE 54277,0:POKE 54278,240                 :rem 44
13 POKE 54273,21                               :rem 39
```



```

14 REM                               :rem 73
15 REM -- TURN ON RING MOD --       :rem 244
16 REM                               :rem 75
17 POKE 54276,21                    :rem 46
18 REM                               :rem 77
19 REM -- SWEEP SCALES W/RING MOD -- :rem 215
20 REM                               :rem 70
30 FOR R=0 TO 255:POKE 54287,R:NEXT  :rem 178
40 POKE 54276,16:POKE 54296,0       :rem 253

```

Line 17 turns the gate bit on, enables the triangle waveform, and selects ring modulation ($1+16+4=21$). The FOR-NEXT loop in line 30 creates values from 0 to 255, which are then POKEd into the high pitch control register of voice 3 (location 54287). The resulting sound is a combination of the sum and difference of the two frequencies produced by voices 1 and 3. It's considerably different than if you simply played the two voices' pitches together.

The second method of using ring modulation sweeps the scales with the triangle waveform of voice 1 and leaves the frequency of voice 3 at a constant value. Program 4-9 shows this method.

Program 4-9. Ring Modulation—Sweeping Voice 1

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS -- :rem 0
6 REM                                  :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT  :rem 24
11 POKE 54296,15                       :rem 45
12 POKE 54277,0:POKE 54278,240        :rem 44
13 POKE 54287,21                       :rem 44
14 REM                                  :rem 73
15 REM -- TURN ON RING MOD --         :rem 244
16 REM                                  :rem 75
17 POKE 54276,21                       :rem 46
18 REM                                  :rem 77
19 REM -- SWEEP SCALES TRIANGLE WAVE -- :rem 202
20 REM                                  :rem 70
30 FOR R=0 TO 255:POKE 54273,R:NEXT    :rem 173
40 POKE 54276,16:POKE 54296,0       :rem 253

```

Instead of sweeping the scale for voice 3's pitch value, now you're sweeping the values for voice 1's frequency. The changes in this routine from Program 4-8 are relatively minor;

lines 13 and 30 are the only ones which are different. You can experiment with the ring-modulated sound by using different pitch values in line 13. Change it to 50, for instance, and the sound is even more unusual.

The third method sweeps the scales using both the frequency of voice 3 *and* voice 1.

Program 4-10. Ring Modulation—Sweeping Both Voices

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKE R,0:NEXT           :rem 24
11 POKE 54296,15                               :rem 45
12 POKE 54277,0:POKE 54278,240                 :rem 44
14 REM                                           :rem 73
15 REM -- TURN ON RING MOD --                   :rem 244
16 REM                                           :rem 75
17 POKE 54276,21                               :rem 46
18 REM                                           :rem 77
19 REM -- SWEEP BOTH VOICES --                 :rem 124
20 REM                                           :rem 70
30 FOR R=0 TO 255:POKE 54273,R:POKE 54287,255-R:NE
   XT                                           :rem 103
40 POKE 54276,16:POKE 54296,0                 :rem 253
```

Synchronization. Synchronization, another form of additive sound synthesis, also adds two tones together to create a new and different effect. It occurs when two waveforms are linked to make the waveform of voice 1 dependent on whether it is in *sync* with the frequency of voice 3. Since the two waveforms are usually *not* in sync, the waveform is distorted. This produces unusual and interesting waveforms. With synchronization in effect, the tone you'll hear depends on the pitch of voice 3, not of voice 1, as you'd normally find true. If you alter voice 3 and keep voice 1's pitch at a constant value, the pitch changes; however, if you change voice 1, and voice 3's pitch remains constant, the *timbre*, or unique characteristics of the waveform, changes. Each manipulation creates its own particular effect.

To use synchronization, you need to set bit 1 of the waveform control register (done by adding 2 to whatever other bit values you're POKEing into the location), then setting voice 3

to a selected pitch, and finally manipulating voice 1's pitch to change the resulting sound. Unlike ring modulation, you're not restricted to just the triangle waveform; you can use the pulse and sawtooth waveforms with synchronization as well. If you're using the triangle waveform, for example, you would POKE 19 into memory location 54276 to turn on the gate bit (1), enable the waveform (16), and select for synchronization (2).

This effect results in a sound that resembles the beat frequencies used earlier in this book. The difference is that the only tone you hear is the resultant beat, which may vary from very slow to very fast.

Take a look at Program 4-11 for an example of synchronization at work.

Program 4-11. Synchronization

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
6 REM                                           :rem 26
10 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 24
11 POKE 54296,15                               :rem 45
12 POKE 54275,8:POKE 54277,0:POKE 54278,240 :rem 0
13 POKE 54287,9                                 :rem 2
14 REM                                           :rem 73
15 REM -- SET SYNC FUNCTION --                 :rem 141
16 REM                                           :rem 75
17 POKE 54276,19                               :rem 53
21 REM                                           :rem 71
23 REM -- SWEEP SCALES W/SYNC --             :rem 255
24 REM                                           :rem 74
30 FOR R=0 TO 255:POKE 54273,R:NEXT           :rem 173
40 POKE 54276,16:POKE 54296,0                 :rem 253

```

In this program, voice 3's pitch is set in line 13 and remains constant as the routine executes. However, in line 30, voice 1's pitch changes as the FOR-NEXT loop increments the values. Note the value of 19 POKEd into location 54276 in line 17. It's a sum of the values for turning the gate bit on (1), enabling the triangle waveform (16), selecting the sync function (2).

You can change the sound by switching the locations POKEd in lines 13 and 30. Make the changes indicated by the following lines and reRUN the program.

```
13 POKE 54273,9  
30 FOR R=0 TO 255:POKE 54287,R:NEXT
```

Quite a difference, wasn't it? Experimenting with synchronization is one of the joys of creating sounds on the 64. You can play with various parameters and end up with a unique sound almost every time. For example, try adding this line to Program 4-11.

```
30 FOR R=255 TO 0 STEP -1:POKE 54273,R:NEXT
```

Now the pitch of voice 1 begins at a high frequency and falls in one-step increments. If you change line 30 again, you can hear an even stranger sound.

```
30 FOR R=255 TO 0 STEP -1:POKE 54273,R:POKE 54287,  
255-R:NEXT
```

Both voices' pitch values are altered by the FOR-NEXT loop now. Play with the sync function on the Commodore 64, and you'll soon find dozens of uses for it in your own programs and games.

Subtractive Synthesis

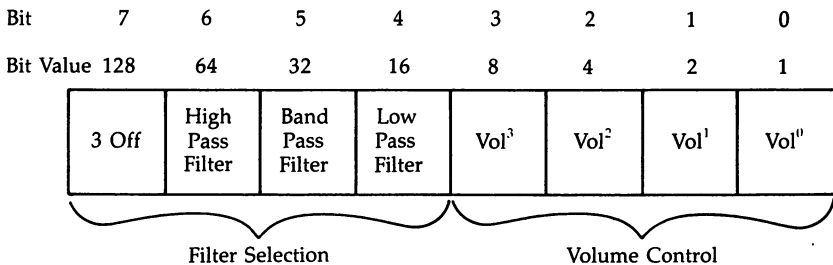
Subtractive synthesis, where sounds are manipulated by *subtracting* parts of a single sound, is another advanced function of the Commodore 64's sound capabilities. Filters are the primary way of altering sounds using subtractive synthesis. There are three filters you can use when you create sounds and music on the 64: the low pass filter, the high pass filter, and the band pass filter.

The filters on the SID chip are similar to filters used for other purposes, whether it's filtering coffee or filtering oil. Some things, such as coffee and oil, are allowed to pass through, while others, such as grounds or dirt, are blocked out. The filters in your computer let parts of the sounds pass through, but selectively block others. And unlike filters in other applications, you can determine what is passed through and what is blocked.

The *low pass* filter is designed to remove the higher frequencies, allowing the lower frequencies to pass through. The *high pass* filter has the opposite effect—it removes the low frequencies while letting the higher frequencies pass. The *band pass* filter allows a band or group of frequencies to pass through, while frequencies above and below the band are blocked.

Filter selection. You select a filter by turning on one of three bits in memory location 54296 (refer to Table 1-1 for the chart of the SID chip's control registers and their locations). Turning on bit 4 enables the low pass filter, setting bit 5 turns on the band pass filter, and setting bit 6 enables the high pass filter. Setting these bits is done as with any other bit: Simply POKE the bit value into the address. POKEing 54296 with 16, for instance, would select the low pass filter. Take a look at Figure 4-10 for the contents of location 54296, the bit functions, and their values.

Figure 4-10. Location 54296



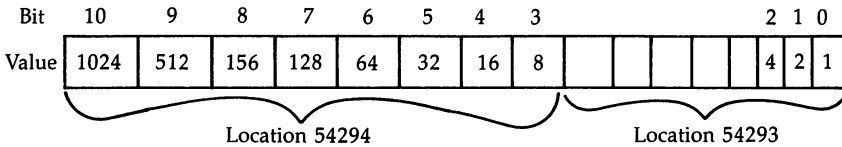
You've previously used this location only for a volume control. Note, however, that this function uses only the lower four bits. When you're selecting a filter, then, you'll have to add its bit value to the volume level value. For instance, if you want to use the low pass filter *and* set the volume at maximum, you'd POKE this location with 31 (16 for the filter, 15 for maximum volume). Keep this in mind as you program sound using filters.

You can add filters together by POKEing both filter values into location 54296. For example, adding the low and high pass filters together will create something called a *band reject* filter, where only the higher and lower frequencies are allowed to pass. The middle frequencies are blocked. To select something like this, you'd POKE 80, plus the desired volume value, into the address.

Cutoff point. The amount of sound that's removed by a filter is determined by the *cutoff point*. The filter cuts off the sound beginning at this point. You control the cutoff point by

setting different bits in an 11-bit register. The lower three bits of location 54293 and all eight bits in location 54294 make up this register. The possible values range from 0 to 2047. Figure 4-11 details the two-byte control for the filter cutoff point.

Figure 4-11. Cutoff Controls



Notice that the possible values in location 54293 range from 0 to 7, while those in location 54294 range from 0 to 255. Those are the number ranges you would actually POKE into the two addresses. However, since this is considered an 11-bit address by the SID chip, the bits can be thought of as having the values listed in Figure 4-11.

The higher the number in the cutoff register, the higher the cutoff frequency. Ranging from approximately 30 Hz to 12,000 Hz, the cutoff frequency determines which of the sound's frequencies pass and which are blocked. The number in the 11-bit address is relative to the sound's frequency. In other words, if you set the seventh bit of location 54294, meaning the number in the address is 1024 (see Figure 4-11), the cutoff frequency is half of the sound's frequencies. If you're using the high pass filter, for instance, and POKE 128 into location 54294 (to set the seventh bit), the bottom half of the sound's frequencies is blocked, while the top half is allowed to pass.

Through experimentation, I've found that by increasing the number in the 11-bit address by 160 each time the sound's frequency is doubled, the cutoff point remains in the same relative position. If you increase the sound's frequency from 64 to 128 Hz, for example, increasing the number in the cutoff point's 11-bit address by 160 will keep the cutoff frequency in the same place, relative to the sound's frequency.

This may sound confusing at first, but if you experiment with the filters and various cutoff points, you'll quickly get an

idea of how it all works. Setting various cutoff points and using a variety of filters is probably the best way to learn, and thus hear, how they operate.

Using Your Filters

To use filters on the Commodore 64, you need to do several things:

1. First of all, you need to choose which voice you're going to filter. The voice selection control consists of the lower three bits in location 54295. To filter voice 1, you'd POKE 54295 with 1; to filter voice 2, POKE 54295,2; and to filter voice 3, POKE 54295,4. To filter *more* than one voice, all you have to do is add the bit values and POKE the total into the location. Selecting voice 1 and voice 2, for instance, is done by POKEing 54295 with 3. Refer to Table 1-1, which shows the SID chip's registers and memory locations.
2. Choose your filter by POKEing the appropriate number into location 54296.
3. Select the cutoff point by POKEing location 54293 and/or 54294 with the desired value. See the explanation above for that process.

Now you're ready to filter sounds on the 64. As a reference, these are the locations used on the 64 to enable and set parameters for filtering:

Cutoff Point (lower value)	54293 bits 0-2
Cutoff Point (higher value)	54294 bits 0-7
Voice 1 Filter Enable	54295 bit 0
Voice 2 Filter Enable	54295 bit 1
Voice 3 Filter Enable	54295 bit 2
Low Pass Filter	54296 bit 4
Band Pass Filter	54296 bit 5
High Pass Filter	54296 bit 6

Here's our first example of filter use on the 64. Type in and RUN Program 4-12 to hear the low pass filter.

Program 4-12. Low Pass Filtering

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

5 REM -- INITIALIZE SOUND REGISTERS --           :rem 0
7 REM                                           :rem 27
10 FOR R=54272 TO 54296:POKE R,0:NEXT           :rem 24
11 POKE 54296,31:POKE 54295,1                   :rem 250
12 POKE 54275,8:POKE 54277,0:POKE 54278,240    :rem 0

```

```
13 POKE 54273,10:POKE 54272,5           :rem 243
14 POKE 54276,65                         :rem 51
21 REM                                    :rem 71
22 REM -- VARYING FILTER LOOP --         :rem 28
23 REM                                    :rem 73
24 FOR R=0 TO 255                         :rem 75
25 POKE 54294,R:NEXT                     :rem 149
51 REM                                    :rem 74
52 REM -- TURN SOUND OFF --              :rem 172
53 REM                                    :rem 76
54 POKE 54276,64:POKE 54283,0           :rem 1
```

Line 11 selects the low pass filter and sets the volume to 15 by POKEing 54296 with 31 (16 for low pass filter plus 15 for volume), then enables the filter for voice 1 by POKEing 1 into location 54295. Most of the rest of the program is similar to other routines you've seen, with the exception of lines 24 and 25. These two lines establish a FOR-NEXT loop to create a number range from 0 to 255, then POKE those values into the high byte of the two-byte cutoff point control register. The cutoff point quickly rises from a low frequency to a high frequency.

Changing the filter changes the sound you hear. Alter line 11 so that the first statement POKes 54296 with 47 (for a band pass filter and volume at 15) or 79 (for a high pass filter and volume at 15). The frequencies passed and blocked are now different, which creates a new sound.

You've heard how the different cutoff points affect the sound. If you use a different waveform you'll get a different filtered sound, since the filters modify the sound shapes. For instance, change lines 14 and 54 in Program 4-12 to:

```
14 POKE 54276,17
54 POKE 54276,16:POKE 54283,0
```

As you can hear, the triangle wave produces a much softer sound than the pulse wave. On the other hand, the sawtooth waveform makes a much harsher sound. Add these lines to Program 4-12:

```
14 POKE 54276,33
24 FOR R=0 TO 255 STEP .5
54 POKE 54276,32:POKE 54283,0
```

By changing the filters in each of the last two modifications to Program 4-12, you can hear different effects. Experimenting with filters is perhaps the best way to learn exactly what they

can do for your sounds. Change the filters, the voices that are used, the cutoff points, and the sound frequencies in Program 4-12 and its modifications. You'll be surprised at the effects you'll create.

Resonance

The process of filtering is a subtractive one. In other words, you begin with a complete sound wave and then, by taking out various parts of it, produce a new kind of sound.

Resonance is a process that adds to the sound wave, emphasizing those parts that are near to the cutoff frequency. When you use filters, try adding resonance. The change will be dramatic.

To use resonance on the 64, you'll need to set additional bits in location 54295. That location is also used to select the voices to be filtered, so if you want to use resonance, you need to add bit values together. Location 54295 looks like Figure 4-12.

Figure 4-12. Resonance and Voice Filter Controls

Bit	7	6	5	4	3	2	1	0
Bit Value	128	64	32	16	8	4	2	1

Resonance ³	Resonance ²	Resonance ¹	Resonance ⁰	External Audio Filtering	Voice 3 Filtered	Voice 2 Filtered	Voice 1 Filtered
------------------------	------------------------	------------------------	------------------------	--------------------------------	---------------------	---------------------	---------------------

Location 54295

If you want maximum resonance and are filtering voice 1, you'll POKE location 54295 with 241 ($128 + 64 + 32 + 16 = 240$ for maximum resonance and 1 for filtering voice 1). Adding this line to Program 4-12 will set resonance to the maximum value, and emphasize the frequencies near the cutoff point:

11 POKE 54296, 31:POKE 54295, 241

Notice how the volume jumps when the frequency sweeps past the selected cutoff point. These kinds of effects can be used very effectively in generating sounds that appear to be moving past, toward, or away from the listener. As in the other examples of filtering, changing the type of filter will

alter the sound. Try using the band pass or high pass filter with various levels of resonance for even more unusual sound effects.

Putting It All Together

Now that you've seen how to program even the more advanced functions of the Commodore 64's SID chip, what can you do with them? How can you use the sound capabilities in your *own* programs, especially in the games you'd like to write? After all, that's one of the pleasures of creating sound and music on the 64: using it to add a professional touch to your own programs and games. For some people, creating sound is its own reward, but for most of us, it's only one portion of our programming techniques.

How can you add sound to other programs? It's not that difficult once you know how to create the sound routines and effects. It's only a matter of making the pieces fit. Chapter 5 shows you how to put it all together.

CHAPTER

5

Putting It All
Together



5

Putting It All Together

By now you know how to create sounds and music on the Commodore 64. You've seen how simple sounds are made, how to create the exact effect you want, and even how to use the more advanced functions of the SID chip such as filtering, ring modulation, and synchronization. Along with the sound editors in Chapter 2, these techniques allow you to produce almost any sound or musical piece you can imagine.

Unfortunately, there's not much you can do with sound effects alone. Sure, they're fun to create and listen to, but that's probably not the reason why you wanted to learn how to create sounds on your 64. Most programmers use sound, not as an end in itself, but as an integral part of larger programs. If you've used any commercial software, or typed in programs and games from magazines, you know how important sound is. Many times it's not only an enhancement, it's a vital part of the program. Sound provides the program user with information, feedback, and entertainment. Without it, the program would seem dull. That's especially true of game programs. Try playing your favorite arcade game with the sound turned off and you'll see what I mean.

You need a way (or preferably several ways) to add your own sound effects and music to other, longer programs. It's not just a matter of sticking them in the program wherever they seem to fit. To successfully insert sound and music routines in other programs, there are a number of things you need to know and keep in mind. That's what this chapter will try to show you.

Combinations and DATA

This chapter explains several ways of combining sound with other programs on your Commodore 64. Most of the time you'll want to put sound together with graphics, the pictures you create on the screen. We'll concentrate on mixing sound and graphics for that reason.

We'll cover several methods of combining sound with graphics. That will work best because combining the two elements is often governed by the particular effect you're trying to produce. Having a method available for several different situations will give you a wider variety to choose from when you begin to mix your own sounds with your own pictures.

In addition to combining sound with graphics, you'll also see how to enhance the sound editors in Chapter 2. By using a special DATA-making program, you'll be able to convert the music produced by the editors into DATA statements that you can include in your BASIC programs.

Finally, we'll use all this to create a simple videogame that uses many of these techniques to show you a practical application of what you've read. By the time you're finished with this chapter, you'll be able to mix sounds and graphics with ease.

Mixing Sound and Graphics

There are two major kinds of sounds that you'll want to combine with pictures on the Commodore 64. These are:

- Sounds produced as needed
- Sounds produced all the time

The reason to make this distinction is that the nature of the actual sound is generally not as important as *when* you want the sound played. For example, sounds that are produced all the time must be interwoven into the structure of the program; however, sounds that are needed intermittently can simply be played at the proper moment by inserting the routine (often called a *subroutine*) to play the sound and then returning to the program.

Sounds Produced as Needed

Since sounds that interrupt the program as they are needed are generally easier to program, we'll cover them first. These sounds are not heard all the time, but are produced only in

response to some action that occurs from within your program.

The bouncing ball. To show this kind of sound and graphics combination, let's take a look at a simple program which calls for sound only at certain times. A crude game based on *Pong*, one of the first arcade games, this program creates two barriers and a single bouncing ball. While the ball is traveling between the two barriers, no sounds are needed. But each time the ball strikes a barrier, a bounce-style sound effect should be heard.

This program produces a sound and reverses the ball's direction after each count of 25. The whole process is cyclic. Since the ball will never do anything unexpected, you don't need to concern yourself with actually interrupting the program to produce the sounds; they're simply planned ahead of time.

Program 5-1. Bouncing Ball

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
1 REM ** INITIALIZE SOUND REGISTERS **           :rem 240
2 REM                                           :rem 22
3 FOR R=54272 TO 54296:POKER,0:NEXT             :rem 234
4 POKE 54296,15:POKE 54275,8:POKE 54277,0:POKE 542
  78,240:POKE 54273,50                          :rem 213
5 PRINT"{CLR}{9 DOWN}{8 RIGHT}";              :rem 85
7 REM                                           :rem 27
8 REM ** MOVE BALL TO THE RIGHT **             :rem 24
9 REM                                           :rem 29
10 FOR R=0 TO 25                                :rem 17
20 PRINT" {RVS} {OFF}{LEFT}";                  :rem 175
30 FOR G=0 TO 35:NEXT                           :rem 130
40 NEXT                                          :rem 163
41 REM                                           :rem 73
42 REM ** PLAY SOUND **                         :rem 177
43 REM                                           :rem 75
45 GOSUB 110                                     :rem 123
46 REM                                           :rem 78
47 REM ** MOVE BALL TO THE LEFT **             :rem 248
48 REM                                           :rem 80
50 FOR R=25 TO 0 STEP -1                        :rem 175
60 PRINT"{3 LEFT}{RVS} {OFF} ";                :rem 237
70 FOR G=0 TO 35:NEXT                           :rem 134
80 NEXT                                          :rem 167
81 REM                                           :rem 77
82 REM ** PLAY SOUND **                         :rem 181
83 REM                                           :rem 79
```

```
85 GOSUB 110 :rem 127
90 GOTO 10 :rem 3
100 REM :rem 117
101 REM ** SOUND SUBROUTINE ** :rem 183
102 REM :rem 119
110 POKE 54276,65:POKE 54276,64 :rem 103
130 RETURN :rem 116
```

The sound registers are initialized in lines 3 and 4, before the program sets up the pictures of the ball and barriers. Lines 45 and 85 call the subroutine which actually plays the sound. Note that the sound is called automatically by the program; it doesn't react to a particular situation (as in most programs which use sounds produced as needed), but plays the sound as soon as the FOR-NEXT loops in lines 10 and 50 have counted up to or down from 25. The sound routine consists of lines 110 and 130. Since all the other sound parameters were set at the beginning of the program, all you have to do is turn the gate bit on, select a waveform (both done by POKEing location 54276 with 65), and then turn off the gate bit (POKE 54276,64). The RETURN in line 130 simply sends the program back to the GOSUB command in line 45 or line 85.

However, adding sounds to pictures is usually more complicated.

Planning for the unplanned: another bouncing ball. In the program above, we used the PRINT command to place the ball and barriers on the screen. Since the program was carefully planned, it didn't need to check the screen to find the barriers, but knew where they were and planned for them ahead of time.

Unfortunately, it's not always so easy to plan for sounds. Sometimes you'll want the program to be able to look at the screen and determine whether it needs to play a sound, based on what's there. This program POKES the ball and barriers onto the screen and PEEKs the locations that the ball will be moving to. If the barrier is in the next location, the ball bounces and the sound plays.

Program 5-2. POKEing and PEEKing for Sound

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
1 REM ** INITIALIZE SOUND REGISTERS ** :rem 240
2 REM :rem 22
3 FOR R=54272 TO 54296:POKER,0:NEXT :rem 234
```


5
Putting It All Together

```
4 POKE 54296,15:POKE 54275,8:POKE 54278,240:POKE 5
  4273,50 :rem 7
5 LB=1390:LC=55662:RB=1418:RC=55690:BL=1391:BC=556
  63:DR=1 :rem 223
6 PRINT"{CLR}"; :rem 213
7 REM :rem 27
8 REM ** SET UP BARRIERS ** :rem 175
9 REM :rem 29
10 POKE LB,160:POKE LC,1 :rem 54
20 POKE RB,160:POKE RC,1 :rem 67
30 POKE BL,160:POKE BC,1 :rem 46
41 REM :rem 73
42 REM ** MOVE BALL ** :rem 68
43 REM :rem 75
45 POKE BL,32:POKE BC,6 :rem 7
50 IF PEEK(BL+DR)=160 THEN GOTO 110 :rem 135
60 BL=BL+DR:BC=BC+DR :rem 194
70 POKE BL,160:POKE BC,1 :rem 50
75 FOR G=0 TO 10:NEXT :rem 132
80 GOTO 45 :rem 10
90 REM :rem 77
92 REM ----- BOUNCE SUBROUTINE ----- :rem 55
100 REM :rem 117
101 REM ** SOUND SUBROUTINE ** :rem 183
102 REM :rem 119
110 POKE 54276,65:POKE 54276,64 :rem 103
121 REM :rem 120
122 REM ** REVERSE DIRECTION ** :rem 222
123 REM :rem 122
130 DR=DR* -1 :rem 133
140 GOTO 45 :rem 55
```

If you compare this with Program 5-1, it doesn't appear to be much more versatile. The major changes are those that POKE and PEEK the various screen memory locations to place the ball and barriers on the screen, and then to erase the ball as it moves. The sound is identical to that used in Program 5-1, and it's called by much the same process. The difference is that in Program 5-1, the sound was called automatically after a certain time, while in this routine it's accessed *only* when a condition is met.

That condition is established in line 50. If the number found by PEEKing location BL (the ball's location in screen memory) plus DR (the direction the ball moves in; a value of either 1 or -1) is 160, the sound is played by calling the one-line subroutine in line 110. What's happening here is that line 50 is looking at the space *ahead* of the path of the ball. That's

because it's PEEKing at the location of the ball (BL) plus or minus (from DR) 1. If that location contains a character with screen code 160 (the character which makes up the ball and barriers), then a bounce sound is played. Any other time, the sound isn't called.

Adding a routine that allows you to move the barriers isn't too difficult and will let you hear the bouncing sound no matter where the barriers are. Take a look at Program 5-3, which is an expanded version of the previous program.

Program 5-3. Moving Barriers

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1 REM ** INITIALIZE SOUND REGISTERS **      :rem 240
2 REM                                       :rem 22
3 FOR R=54272 TO 54296:POKER,0:NEXT        :rem 234
4 POKE 54296,15:POKE 54275,8:POKE 54278,240:POKE 5
  4273,50                                     :rem 7
5 LB=1390:LC=55662:RB=1418:RC=55690:BL=1391:BC=556
  63:DR=1                                     :rem 223
6 PRINT"{CLR}";                             :rem 213
7 REM                                       :rem 27
8 REM ** SET UP BARRIERS **                 :rem 175
9 REM                                       :rem 29
10 POKE LB,160:POKE LC,1                    :rem 54
10 POKE RB,160:POKE RC,1                    :rem 67
10 POKE BL,81:POKE BC,1                     :rem 0
42 REM ** MOVE BALL **                       :rem 68
43 REM                                       :rem 75
45 OL=BL:OC=BC                              :rem 93
60 BL=BL+DR:BC=BC+DR                        :rem 194
61 IF PEEK(BL)=160ORBL<LB+1ORBL>RB-1THEN BL=OL:BC=
  OC:GOTO91                                   :rem 130
63 POKEBL,81:POKEBC,1:POKEOL,32:POKEOC,6:POKELB,16
  0:POKELC,1                                 :rem 7
64 POKERB,160:POKERC,1                     :rem 75
75 FOR G=0 TO 10:NEXT                       :rem 132
78 REM                                       :rem 83
79 REM **** MOVE BARRIER ROUTINE ****      :rem 8
81 OB=LB:OC=LC                              :rem 93
82 GETA$:IF A$=""THEN42                     :rem 243
83 IF A$="{F1}"THEN 150                     :rem 42
84 IF A$="{F3}"THEN 160                     :rem 45
85 IF A$="{F5}"THEN 170                     :rem 48
86 IF A$="{F7}"THEN 180                     :rem 51
90 GOTO42                                    :rem 8
91 POKELB,160:POKELC,1:POKERB,160:POKERC,1 :rem 90
  
```

5
Putting It All Together

```
92 REM ----- BOUNCE SUBROUTINE ----- :rem 55
100 REM :rem 117
101 REM ** SOUND SUBROUTINE ** :rem 183
102 REM :rem 119
110 POKE 54276,65:POKE 54276,64 :rem 103
130 DR=DR* -1 :rem 133
140 GOTO 42 :rem 52
150 REM :rem 122
151 REM ** MOVE LEFT BARRIER TO LEFT ** :rem 90
152 REM :rem 124
153 OB=LB:OC=LC :rem 141
154 LB=LB+1:LC=LC+1 :rem 64
155 POKE LB,160:POKE LC,1:POKEOB,32:POKEOC,6
:rem 88
159 GOTO 42 :rem 62
160 REM :rem 123
161 REM * MOVE LEFT BARRIER TO RIGHT ** :rem 132
162 REM :rem 125
163 OB=LB:OC=LC :rem 142
164 LB=LB-1:LC=LC-1 :rem 69
165 POKE LB,160:POKE LC,1:POKEOB,32:POKEOC,6
:rem 89
169 GOTO 42 :rem 63
170 REM :rem 124
171 REM * MOVE RIGHT BARRIER TO LEFT ** :rem 133
172 REM :rem 126
173 BR=RB:CR=RC :rem 161
175 RB=RB+1:RC=RC+1 :rem 91
176 POKERB,160:POKERC,1:POKEBR,32:POKECR,6:rem 109
179 GOTO 42 :rem 64
180 REM :rem 125
181 REM * MOVE RIGHT BARRIER TO RIGHT * :rem 175
182 REM :rem 127
183 BR=RB:CR=RC :rem 162
185 RB=RB-1:RC=RC-1 :rem 96
186 POKERB,160:POKERC,1:POKEBR,32:POKECR,6:rem 110
189 GOTO 42 :rem 65
```

Now you can move the barriers left and right by pressing the function keys (the keys on the far right of the keyboard). The f1 key moves the left barrier to the right, while the f3 key moves it to the left. Pressing the f5 key moves the right barrier to the right, and the f7 key moves it to the left. If you press the f3 or f5 key enough times, the barrier will move to the line above or below, respectively. If you do this, the ball has to travel even further before it bounces off.

Although much of this program is similar to Program 5-2,

there are some obvious differences. First of all, the program can read the function keys when you press them. Lines 82-86 do this. If one of the four function keys is pressed, then the program goes to the appropriate subroutine in line 150, 160, 170, or 180 to move the barrier in the right direction. In those subroutines, the old locations of the barriers (both in screen and color memory) are changed to the new locations, the new locations have 1 or -1 added to them (signifying movement to the right or left), and then the new positions are POKEd into memory. The old location is erased with a space (screen POKE code 32), and then the routine returns to line 42.

The sound routine is identical to that in Program 5-2. It's called in much the same way, by PEEKing into the next location and seeing if there's anything there. If there is, that means a collision is about to happen, and the sound is played. The statements in line 61 check for collisions, no matter where they are on the screen, and access the sound effect if it's needed. That line is rather involved, so let's take a closer look at its logic. It's important to our discussion of mixing sound and graphics, for this is the line that actually decides if sound should be played.

If the location of the ball character (BL) contains 160, meaning the barrier character, or if the ball's location is less than the left barrier's location plus one, then the sound will play. That's expressed by the part of the line which reads $IF\ PEEK(BL)=160\ OR\ BL < LB+1$. In other words, if the ball is moving to the left, it will "see" the barrier character as it tries to occupy the same location. If the ball character is in the location just to the right of the left barrier ($LB+1$) and is trying to move even further to the left (trying to make BL less than $LB+1$), it will also "see" the barrier. Either of those conditions, if true, will access the sound because the THEN portion of line 61 will execute. The part of the first statement which reads $OR\ BL > RB-1$ does the same thing, only when the ball is moving to the right and trying to go even further in that direction when it is next to the right barrier. If that condition exists, the program detects the right barrier and again calls for the bouncing sound to play.

As you can see, the process of PEEKing and POKEing can be involved. Its versatility usually outweighs the complicated programming you have to go through, however.

Music box. Another application for this kind of program is to use the ball character as a cursor on the screen and use the barriers to call DATA. For example, we could place musical notes on the screen and use the cursor to play the notes as it passes over them, much like the way a music box plays notes as the tines of the musical drum pick them.

Program 5-4. Music Box

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1 REM ** INITIALIZE **                :rem 175
2 REM                                  :rem 22
3 FOR R=54272 TO 54296:POKER,Ø:NEXT:POKE 54275,8:P
  OKE 54278,24Ø                        :rem 243
4 POKE 54296,15:POKE 54275,8:POKE 54278,24Ø :rem 8
9 CR=1Ø24                               :rem 21Ø
2Ø REM                                  :rem 7Ø
25 REM *** MUSICAL NOTE DATA ***      :rem 165
27 REM                                  :rem 77
29 DIM U(12),L(12)                      :rem 122
3Ø U(3)= 33{2 SPACES}: L(3)= 134        :rem 19Ø
32 U(4)= 37{2 SPACES}: L(4)= 161        :rem 198
34 U(5)= 42{2 SPACES}: L(5)= 6Ø        :rem 148
35 U(6)= 44{2 SPACES}: L(6)= 191        :rem 2Ø6
37 U(7)= 25{2 SPACES}: L(7)= 29         :rem 161
39 U(1)= 28{2 SPACES}: L(1)= 48         :rem 155
41 U(2)= 31{2 SPACES}: L(2)= 164        :rem 191
5Ø REM                                  :rem 73
55 REM **** CREATE MUSIC DISPLAY ****   :rem 233
57 REM                                  :rem 8Ø
58 PRINT"{CLR}";:FOR M=Ø TO 5            :rem 191
6Ø PRINT"{4 SPACES}C{3 SPACES}C{3 SPACES}D
  {3 SPACES}E{3 SPACES}C{3 SPACES}E{3 SPACES}D"
                                          :rem 18
61 PRINT "C{3 SPACES}C{3 SPACES}D{3 SPACES}E
  {3 SPACES}C{1Ø SPACES}B"                :rem 2Ø4
62 PRINT"C{3 SPACES}C{3 SPACES}D{3 SPACES}E
  {3 SPACES}F{3 SPACES}E{3 SPACES}D{3 SPACES}C
  {3 SPACES}B{3 SPACES}G{3 SPACES}A{3 SPACES}B
  {3 SPACES}C{11 SPACES}C"                :rem 236
63 NEXT                                  :rem 168
112 REM                                  :rem 12Ø
115 REM *** MOVE CURSOR/PLAY MUSIC ***  :rem 114
117 REM                                  :rem 125
12Ø N=PEEK(CR)                           :rem 41
125 POKE CR,N+128                         :rem 156
13Ø IF N>7THEN 15Ø                       :rem 171
145 GOTO 2Ø5                              :rem 1Ø6

```

```

150 POKE CR,N                               :rem 212
160 CR=CR+1:IF CR=2023 THEN CR=1024       :rem 132
165 GOTO 120                                 :rem 104
200 REM                                     :rem 118
201 REM ** SOUND SUBROUTINE **             :rem 184
202 REM                                     :rem 120
205 POKE 54273,U(N):POKE 54272,L(N)       :rem 111
210 POKE 54276,65                           :rem 97
220 FOR R=0 TO 40:NEXT                     :rem 186
230 POKE 54276,64                           :rem 98
240 GOTO 150                                :rem 101

```

This program uses some rather elaborate ways to create the song. Notice the upper and lower pitch values that are initialized in lines 30–41. These are the pitch values that will later be POKEd into the appropriate control registers to actually produce the individual notes as the cursor moves across them. The notes, as initialized, are: C, D, E, F, G, A, and B. Line 30, for instance, establishes the pitch values for the C note.

Lines 58–63 PRINT the notes on the screen. Using this program and the pitch values set up earlier, you could alter these lines to PRINT any combination of notes on the screen, letting the cursor “play” the tune. The next major section of the program moves the cursor and plays the musical notes when the cursor passes over them. This is done through a process of PEEKing and POKeing to screen memory, seeing where the cursor is (line 120), and checking to see that the location is a space and not a letter from A to G (line 130). If the screen POKE code in the location *isn't* greater than 7, it means there's a note character there, and the program goes to line 205 to play the note. Line 160 moves the cursor one location to the right, and if it's at the bottom right-hand corner, it wraps around to the upper left-hand corner and starts all over again.

The sound routine isn't much longer than in the previous routines in this chapter. The pitch values initialized earlier are now POKEd into the correct upper and lower control registers, the pulse waveform is enabled and the gate bit turned on, a short sustain delay loop plays the note, and then the gate bit is turned off.

To change the musical notes or the timing, you can change the notes that are PRINTed on the screen in lines 60 through 62, adding as many additional lines as you need to put your whole song on the screen. While this program is not

the most practical way of producing music, it does demonstrate how the screen and the music can interact.

Bouncing sprites. Sprites, like characters, can be put anywhere on the screen. However, since sprites do not use screen memory locations for placement, you can't PEEK to find them. Sprites *can* be detected in another way, which is actually easier to use than PEEKing and POKEing. By looking at two memory locations called sprite collision registers, which signal collisions between two sprites, or between a sprite and a character, you can tell if a sprite has bumped into something. You don't really need to know how this works, since it's a matter of sprite control, not sound. The important thing is that you can use the sprite collision registers to combine sound and graphics.

Program 5-5. Sprite Collisions and Sound

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1 REM ** INITIALIZE SOUND REGISTERS **      :rem 240
2 REM                                       :rem 22
3 FOR R=54272 TO 54296:POKER,0:NEXT:POKE54275,8:PO
  KE54278,240                               :rem 243
4 POKE 54296,15:POKE 54275,8:POKE 54278,240:POKE 5
  4273,50                                     :rem 7
5 REM                                       :rem 25
6 REM ** PUT OBSTACLES ON SCREEN **        :rem 184
7 REM                                       :rem 27
10 PRINT"{CLR}{WHT}{9 DOWN}{6 RIGHT}{RVS} {OFF}
   {25 RIGHT}{RVS} {OFF}":F=-7             :rem 79
11 REM                                       :rem 70
12 REM ** PROGRAM SPRITE DATA **          :rem 248
13 REM                                       :rem 72
15 FOR R=3200 TO 3262                       :rem 17
17 POKE R,0:NEXT                           :rem 190
20 POKE 53269,1:POKE 2040,50               :rem 183
30 FOR R=3218 TO 3239 STEP 3               :rem 138
40 POKE R,255                               :rem 173
50 NEXT                                     :rem 164
70 POKE 53249,116:R=70:POKE 53248,R       :rem 126
71 REM                                       :rem 76
72 REM ** MOVE SPRITE ROUTINE **           :rem 41
73 REM                                       :rem 78
80 R=R+F                                     :rem 186
100 POKE 53264,ABS(256<R):POKE53248,RAND255
                                           :rem 158
101 REM                                       :rem 118

```

```

102 REM ** CHECK FOR COLLISIONS **           :rem 99
103 REM                                       :rem 120
105 A=PEEK(53279):IF(AAND1=1)THEN 210       :rem 35
110 GOTO 80                                  :rem 51
115 FOR R=0 TO 500:NEXT                      :rem 238
120 FOR R=350 TO 0 STEP -1                  :rem 14
130 POKE 53264,ABS(256<R):POKE53248,RAND255 :rem 161
                                           :rem 212
140 NEXT                                     :rem 212
150 FOR R=0 TO 500:NEXT                      :rem 237
160 GOTO 80                                  :rem 56
170 REM                                       :rem 124
180 REM ** BOUNCE SUBROUTINE **             :rem 241
190 REM                                       :rem 126
210 POKE 54276,65:POKE 54276,64            :rem 104
225 F=F*-1:R=R+(F*4):POKE53264,ABS(256<R):POKE5324 :rem 174
      8,RAND255:K=PEEK(53279)
230 GOTO 80                                  :rem 54

```

In many ways, this is similar to the bouncing ball programs you've already seen. The sound registers are initialized and then two obstacles are placed on the screen. Lines 12-70 set up the sprite that moves across the screen and bounces off the barriers. Sprite 0 is used in this routine; it's enabled in line 20 by the POKE 53269,1 statement and then created by the next three lines. Its X and Y positions are set to 70 and 116 in line 70.

The sprite is moved by lines 80 and 100. Collision checking, the vital part of this sound and graphics mixture, takes place in lines 105-160. The important line is line 105. Location 53279 checks for collisions between sprites and other characters on the screen (location 53278 checks for collisions between sprites). When a collision occurs, the sprite involved has its bit in location 54279 set to 1. Since we're using sprite 0, bit 0 is the one that will be set on collision. First the collision register is PEEKed. If the value found there is 1, meaning a collision has happened, then the statement A AND1=1 is true and the program calls the sound subroutine to produce the bouncing sound. If the value in that bit is 0, no collision has occurred, and no sound is played.

The sound subroutine you've seen before. The waveform and gate bit are turned on, then off. The rest of the program resets several variables and moves the sprite. The last statement in line 225, K=PEEK(53279), is necessary to clear the

collision register so that the next time it's looked at it won't give a wrong value.

For more detailed information on how to create sprites and use them in your programs, refer to such books as *Commodore 64 Programmer's Reference Guide* or *COMPUTE!'s Reference Guide to Commodore 64 Graphics*. The thing to keep in mind concerning sound is that you can easily use the sprite-to-sprite or sprite-to-character collision registers to call sound at specific points in your program or game. In many ways, it's easier than checking for character locations using POKEs and PEEKs. As with most programming techniques, experimentation is important.

Background Sounds

Though each of the examples so far has been different in some way, they all have had one thing in common: The sounds were produced as a result of some timing function or an event occurring on the screen. In this section, we'll concentrate on sounds that are produced *while* something else is going on.

Such sounds come in many forms and are used for different reasons and effects. Perhaps you want a continuous sound playing in the background as the game runs. Or even background music that you've composed. You could use background sound for other effects, too, such as a sound effect that's connected to an action on the screen. It plays all the time, however, instead of just once in a while, as when you call a sound subroutine. Engine noises, footsteps, or wind noise might be some of the effects you'll want to use to emphasize onscreen actions. These kinds of continuous sounds are sometimes simple, other times more complicated. You can create effects that start with one sound and then change later. They're more complex, but still not too difficult to program.

Sounds connected to an action. The first background sound we'll look at begins and continues while the action takes place onscreen. In this first example, we'll take advantage of the fact that the Commodore 64's SID chip continues making a sound as long as it's not turned off.

Program 5-6. Lawn Mower

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
1 REM ** INITIALIZE SOUNDS & SCREEN **      :rem 113
2 REM                                          :rem 22
```

5
Putting It All Together

```

3 FOR R=54272 TO 54296:POKER,0:NEXT           :rem 234
4 POKE 54296,15:POKE 54275,8:POKE 54278,240 :rem 8
5 POKE 54273,1:POKE 54272,1:POKE 54276,65  :rem 152
6 PRINT"{CLR}"                               :rem 154
10 REM                                       :rem 69
20 REM **** LAWN/GRASS SHAPE TABLE ****    :rem 80
30 REM                                       :rem 71
40 LM$="[5][A]*{RED}{DOWN}{2 LEFT}N{DOWN}
   {2 LEFT}N{DOWN}{3 LEFT}{BLK}WW{WHT}[*]{RVS}
   {2 SPACES}{OFF}[7]"                       :rem 64
42 NM$="[5]{2 SPACES}{RED}{DOWN}{2 LEFT} {DOWN}
   {2 LEFT} {DOWN}{3 LEFT}{BLK}{2 SPACES}{WHT}
   {3 SPACES}[7]"                             :rem 7
45 CG$="{GRN}{UP}{2 LEFT}:.:.{UP}{3 LEFT}:{2 DOWN}
   .."                                         :rem 159
47 NG$="{GRN}{UP}{2 LEFT}{4 SPACES}{UP}{3 LEFT}
   {2 DOWN}:.:."                             :rem 186
50 GR$="{GRN}{5 SPACES}-GTY-TYGH-TYG-GHT-GHT--GHHG
   "                                           :rem 212
52 GC$="{GRN}{UP}{2 LEFT}:."                 :rem 231
55 PO$="{HOME}{23 DOWN}"                     :rem 72
60 REM                                       :rem 74
70 REM **** DISPLAY LAWN & MOWER ****       :rem 147
80 REM                                       :rem 76
90 PRINT PO$;GR$;"{3 UP}{2 RIGHT}";LM$;"[7]
   {HOME}"                                     :rem 65
100 REM                                       :rem 117
110 REM **** REV UP LAWN MOWER ****         :rem 20
120 REM                                       :rem 119
130 POKE 54276,65                             :rem 98
140 FOR R=1 TO 3 STEP.3                       :rem 175
150 POKE 54273,R                             :rem 72
160 FOR T=0 TO R*30:NEXT                     :rem 58
170 POKE 54276,64                             :rem 101
180 FOR T=0 TO R*30:NEXT                     :rem 60
190 POKE 54276,65:NEXT                       :rem 225
200 REM                                       :rem 118
210 REM **** MOW LAWN ****                  :rem 236
220 REM                                       :rem 120
230 FOR R=0 TO 30                             :rem 65
240 PRINT PO$;GR$;"{3 UP}{2 RIGHT}";LM$;CG$;
                                           :rem 102
245 FOR T=0 TO 20: NEXT                       :rem 193
250 PRINT PO$;GR$;"{3 UP}{2 RIGHT}";NM$;NG$;
                                           :rem 116
260 GR$=LEFT$(GR$,LEN(GR$)-1)                :rem 102
270 NEXT                                       :rem 216
290 POKE 54276,64:POKE 54296,0              :rem 55

```

```

300 PRINT PO$;GR$;"{3 UP}{2 RIGHT}";LMS$;GC$;"[7]
      {HOME}" :rem 84
310 PRINTTAB(14)"LAWN MOWED!" :rem 193
320 GOTO 320 :rem 99

```

The lawn mower motor sound in this program is created by selecting a very low pitch. The pitch values are first set to 1 in line 5 for both the upper and lower control registers. Later, in line 150, the value of the upper register changes slightly. Turning the gate bit on, then off, then on again in lines 130-190, produces the sound effect.

The lawn mower moves across the lawn, and when it stops, the sound is turned off. A simple message displays at the top of the screen, and by continually going to line 320, the program prevents the READY message from spoiling the picture. As you can see, the program initializes the sound control registers and starts the sound by enabling the pulse waveform and turning on the gate bit. The sound continues as the rest of the program executes. When the program has run, the sound is turned off in line 290. Simple, isn't it?

Changing sounds on the fly. As you can see, some sounds need only be turned on and they'll run without further instructions. In some applications, however, you may want to use sounds that change while the program is in the middle of animating a figure. Since these sounds are changed during the program, there must be a way to update the picture and sound as the animation occurs. It's best to alternate the two operations, updating one, then the other, each time the program goes through a loop. Program 5-7 uses this method of mixing sound and graphics.

Program 5-7. Helicopter Sound and Animation

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1 REM ** INITIALIZE SOUND REGISTERS ** :rem 240
2 REM :rem 22
3 FOR R=54272 TO 54296:POKE R,0:NEXT :rem 234
4 POKE 54296,15:POKE 54275,8:POKE 54278,240 :rem 8
5 POKE 54273,1:POKE 54272,1:POKE 54276,65 :rem 152
6 REM :rem 26
7 REM *** HELICOPTER DISPLAY SHAPES *** :rem 224
8 REM :rem 28
10 REM :rem 69
11 REM --- HELICOPTER (BLADE LEFT) --- :rem 23
12 REM :rem 71

```

5
Putting It All Together

```

15 A$="{8 SPACES}[5 O][P][F]{25 SPACES}"
                                                    :rem 83
20 A$=A$+"{13 SPACES}[@][J]{3 SPACES}[3 @]
[A]I{17 SPACES}"
                                                    :rem 150
30 A$=A$+"{12 SPACES}{RVS}[£]{6 SPACES}{OFF}[£] J
[X]{17 SPACES}"
                                                    :rem 86
40 A$=A$+"{12 SPACES}{RVS}{4 SPACES}{OFF}[£][2 T]
{21 SPACES}"
                                                    :rem 109
50 A$=A$+"{12 SPACES}[4 T]
                                                    :rem 69
60 REM
                                                    :rem 74
61 REM --- HELICOPTER (BLADE RIGHT) ---
                                                    :rem 111
62 REM
                                                    :rem 76
80 B$="{14 SPACES}[D][P][5 O]{19 SPACES}"
                                                    :rem 71
90 B$=B$+"{13 SPACES}[@][J]{3 SPACES}[3 @][U]
[S]{17 SPACES}"
                                                    :rem 169
100 B$=B$+"{12 SPACES}{RVS}[£]{6 SPACES}{OFF}[£]
[Z]K{17 SPACES}"
                                                    :rem 119
110 B$=B$+"{12 SPACES}{RVS}{4 SPACES}{OFF}[£]
[2 T]{21 SPACES}"
                                                    :rem 157
111 B$=B$+"{12 SPACES}[4 T]
                                                    :rem 117
112 REM
                                                    :rem 120
113 REM --- HELICOPTER (BOTH -FAST-) --
                                                    :rem 79
114 REM
                                                    :rem 122
122 C$="{8 SPACES}[5 O][P][I][P][5 O]
{19 SPACES}"
                                                    :rem 183
123 C$=C$+"{13 SPACES}[@][J]{3 SPACES}[3 @][U]
[S]{17 SPACES}"
                                                    :rem 216
124 C$=C$+"{12 SPACES}{RVS}[£]{6 SPACES}{OFF}[£]
[Z]K{17 SPACES}"
                                                    :rem 127
125 C$=C$+"{12 SPACES}{RVS}{4 SPACES}{OFF}[£]
[2 T]{21 SPACES}"
                                                    :rem 165
126 C$=C$+"{12 SPACES}[4 T]
                                                    :rem 125
127 REM
                                                    :rem 126
128 REM **** DISPLAY HELICOPTER ****
                                                    :rem 212
129 REM
                                                    :rem 128
130 PRINT "{WHT}{CLR}":SP$="{HOME}{18 DOWN}
{8 RIGHT}"
                                                    :rem 172
131 REM
                                                    :rem 121
132 REM **** REV UP HELICOPTER ****
                                                    :rem 75
133 REM
                                                    :rem 123
140 FOR G=0 TO 255 STEP 5
                                                    :rem 224
150 POKE 54272,G
                                                    :rem 60
160 PRINTSP$;A$:PRINTSP$;B$:NEXT
                                                    :rem 51
171 REM
                                                    :rem 125
172 REM **** HELICOPTER TAKE OFF ****
                                                    :rem 189
173 REM
                                                    :rem 127
240 PRINTSP$;C$
                                                    :rem 140
250 FOR R=30 TO 0 STEP -.75
                                                    :rem 70

```

```

255 POKE 54296,R/2                :rem 180
260 PRINT:FOR F=0TO10:NEXT:NEXT   :rem 239
270 PRINT TAB(15)"TAKE OFF!"     :rem 25
280 GOTO 280                      :rem 109

```

Most of this program listing is used to create the picture of the helicopter. The parts that update the sound and graphics each time through the loop are relatively short. Look at the Rev Up Helicopter routine in lines 140–160 and the Helicopter Takeoff routine in lines 240–260. This is where the updating takes place. For instance, as the helicopter's motor speeds up (lines 140 and 150), the picture is changed accordingly in line 160. Each time through the FOR-NEXT loop, the sound and then the graphics are altered. The order of execution is reversed when the helicopter takes off, for the picture is first changed in line 240, and the sound in line 255. Even though the two operations are done in separate lines, it happens so fast that it seems they're taking place at the same time.

Background music. Music is one of the most noticeable aspects of sound in many programs, especially in games. The type of music and its pace have an important effect on what the player thinks of the game. What would *Pac-Man* be without its background musical score? When you're creating graphics that move about the screen, music can add a professional touch to the display. It's not necessary, but it will certainly enhance the program.

Music played as background to animation can be synchronized to that animation or it can play independently. Of course, it's easier to create a musical piece that's synchronized to the animation, because it requires no additional timing. All you need to do is change the notes each time you update the animation, much like we did in the helicopter demonstration.

Synchronizing music and animation can be very effective because both components become tightly interwoven. One easy way to do this is by simply marking time with the display as the music is played. Program 5-8 is a good example.

Program 5-8. Title Screen

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

90 REM **** MUSIC DATA ****      :rem 56
95 REM                             :rem 82
100 FOR L=0 TO 25:PRINT:NEXT      :rem 123
200 DATA 33,134,100,0,33,134,100,0,37,161,100,0,42
    ,60,100,0,33,134,100,0,42,60   :rem 161

```

```

210 DATA 100,0,37,161,100,0,25,29,100,0,33,134,100
,0,33,134,100,0,37,161,100,0 :rem 161
220 DATA 42,60,100,0,33,134,100,0,0,0,100,31,164
,100,0,0,0,100,33,134,100,0 :rem 175
230 DATA 33,134,100,0,37,161,100,0,42,60,100,0,44,
191,100,0,42,60,100,0,37,161 :rem 173
240 DATA 100,0,33,134,100,0,31,164,100,0,25,29,100
,0,28,48,100,0,31,164,100,0 :rem 118
250 DATA 33,134,100,0,0,0,0,100,33,134,100,0,999,9
99,999,999 :rem 144
251 REM :rem 124
252 REM ** INITIALIZE SOUND REGISTERS * :rem 46
253 REM :rem 126
260 FOR R=54272 TO 54296:POKER,0:NEXT :rem 79
261 POKE 54296,15:POKE 54275,8:POKE 54278,240:POKE
54276,65 :rem 117
270 DIM A$(31) :rem 141
271 REM :rem 126
272 REM ** SCREEN TITLE DATA TABLE ** :rem 235
273 REM :rem 128
280 FOR R=0 TO 31:A$(R)=" ":NEXT :rem 173
290 A$(9)="Y":A$(10)="A":A$(11)="N":A$(12)="K":A$(
13)="E":A$(14)="E":A$(15)=" " :rem 195
300 A$(16)="D":A$(17)="O":A$(18)="O":A$(19)="D":A$(
(20)="L":A$(21)="E" :rem 29
310 A$(8)=" ":A$(22)=" " :rem 216
320 PRINT"{CLR}{7 DOWN}{4 RIGHT}"; :rem 31
321 REM :rem 122
322 REM **** GET MUSIC DATA **** :rem 70
323 REM :rem 124
360 READ H,L,LE,P:IF H=999 THEN 420 :rem 108
361 REM :rem 126
362 REM *** DISPLAY NEXT CHARACTER *** :rem 93
363 REM :rem 128
370 PRINTA$(I);:I=I+1 :rem 198
371 REM :rem 127
372 REM *** PLAY NEXT NOTE *** :rem 39
373 REM :rem 129
380 POKE 54272,L:POKE 54273,H :rem 40
390 FOR R=0 TO 3*(P+LE):NEXT :rem 24
400 POKE 54276,64:POKE 54276,65 :rem 105
410 GOTO 360 :rem 103
411 REM :rem 122
412 REM ** TURN OFF SOUND/CLR SCREEN ** :rem 160
413 REM :rem 124
420 POKE 54276,64:POKE 54296,0 :rem 50
430 GOTO 430 :rem 103

```

Once the music DATA, the sound registers, and the title screen DATA are established, the program begins putting characters on the screen and playing notes. Storing the characters in an array allows you to use a single line to print the various letters and symbols on the screen. Line 370 does this. As each character is placed on the screen, the appropriate note plays. Lines 380–410 play the notes, using the values obtained from the DATA statements near the top of the program. Line 360 actually READs the DATA, while line 380 POKEs the upper and lower pitch values into the register.

This program's loops are similar to those in Program 5-7 in that the sound and picture are alternately updated to give the impression of happening simultaneously.

Nonsynchronized music produces a more casual effect than synchronized music. The idea is to produce a picture that runs at a rate which appears to be unrelated to the music. Since music is generally rhythmic, it will require a continuous clock while the program is running if it is to play all the time. While the music is running, it's possible to use the same clock to trigger events on the screen. By doing this, it might seem that the sound and picture could be synchronized. However, by running the clock at a very fast rate and triggering at different times, the picture and sound can appear to be running at unrelated rates. Type in and RUN Program 5-9 to hear and see this nonsynchronization.

Program 5-9. Nonsynchronized Sound and Graphics

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

90 REM **** MUSIC DATA ****                :rem 56
95 REM                                        :rem 82
99 FOR L=0 TO 25:PRINT:NEXT                  :rem 92
100 DATA 33,134,100,0,33,134,100,0,37,161,100,0,42
    ,60,100,0,33,134,100,0,42,60             :rem 160
105 DATA 100,0,37,161,100,0,25,29,100,0,33,134,100
    ,0,33,134,100,0,37,161,100,0             :rem 164
110 DATA 42,60,100,0,33,134,100,0,0,0,0,100,31,164
    ,100,0,0,0,0,100,33,134,100,0           :rem 173
115 DATA 33,134,100,0,37,161,100,0,42,60,100,0,44,
    191,100,0,42,60,100,0,37,161           :rem 175
120 DATA 100,0,33,134,100,0,31,164,100,0,25,29,100
    ,0,28,48,100,0,31,164,100,0           :rem 115
125 DATA 33,134,100,0,0,0,0,100,33,134,100,0,0,0,0
    ,100,999,999,999,999                   :rem 98
252 REM                                        :rem 125

```

5
Putting It All Together

```
253 REM ** INITIALIZE SOUND REGISTERS *      :rem 47
254 REM                                       :rem 127
260 FOR R=54272 TO 54296:POKER,0:NEXT        :rem 79
261 POKE 54296,15:POKE 54275,8:POKE 54278,240:POKE
    54276,65                                :rem 117
270 DIM A$(200)                              :rem 187
271 REM                                       :rem 126
272 REM ** SCREEN TITLE DATA TABLE **      :rem 235
273 REM                                       :rem 128
280 FOR R=0 TO 31:A$(R)=" ":NEXT             :rem 173
285 FOR R=32 TO 200:A$(R)=" ":NEXT           :rem 235
290 A$(9)="Y":A$(10)="A":A$(11)="N":A$(12)="K":A$(
    13)="E":A$(14)="E":A$(15)=" "          :rem 195
300 A$(16)="D":A$(17)="O":A$(18)="O":A$(19)="D":A$(
    20)="L":A$(21)="E"                      :rem 29
310 A$(8)=" ":A$(22)=" "                    :rem 216
320 PRINT"{CLR}{7 DOWN}{4 RIGHT}";          :rem 31
321 REM                                       :rem 122
322 REM **** GET MUSIC DATA ****           :rem 70
323 REM                                       :rem 124
355 GOTO 480                                  :rem 114
360 READ H,L,LE,P:IF H=999 THEN 420          :rem 108
361 GOTO 380                                  :rem 110
362 REM                                       :rem 127
363 REM *** DISPLAY NEXT CHARACTER ***       :rem 94
364 REM                                       :rem 129
370 PRINTA$(I);:I=I+1                        :rem 198
371 RETURN                                    :rem 123
372 REM                                       :rem 128
373 REM *** PLAY NEXT NOTE ***              :rem 40
374 REM                                       :rem 130
380 POKE 54272,L:POKE 54273,H               :rem 40
390 FOR R=0 TO 3*(P+LE):NEXT                 :rem 24
400 POKE 54272,0:POKE 54273,0              :rem 237
410 RETURN                                    :rem 117
411 REM                                       :rem 122
412 REM ** PLAY TUNE AGAIN **               :rem 245
413 REM                                       :rem 124
420 RESTORE                                  :rem 186
440 GOTO 360                                  :rem 106
450 REM **** SYSTEM TIMER ****              :rem 51
460 REM                                       :rem 126
480 IF X/2=INT(X/2) THEN GOSUB 360          :rem 94
485 IF X/2=INT(X/2) THEN X=X+1:GOTO 480     :rem 162
490 IF X/3=INT(X/3) THEN GOSUB 370         :rem 98
495 IF X/3=INT(X/3) THEN X=X+1:GOTO 480     :rem 165
500 X=X+1:GOTO 480                           :rem 237
```


In Program 5-9, the music and the characters appear at different times because the system timer (lines 450-500) causes the program to jump to the Display Next Character routine at lines 362-371 and the Play Next Note routine in lines 372-410 at different counts of the timer. Of course, by using a faster clock, one that incorporated fewer or faster sub-routines, you could introduce much more complex timing, creating the illusion of totally nonsynchronized timing.

In order for the song to last as long as it takes the display to appear, it has to be played more than once. The RESTORE command in line 420 lets the program use the same DATA statements over again to play the notes. Without this command, you would have to enter the DATA statements at least three times to make the sound and graphics end at the same time.

Using Edited Music Files

You've got a better idea now of how to merge sound with graphics on the screen. Using sound that's called either for specific events, or for continuous background sound or music, you can get impressive results. If you've already created sound effects or music, however, you have to retype it all when you add the sound to an existing program. If you're using DATA statements to play notes, probably the most efficient method of programming sound, that can be a chore.

You've probably already used the two sound editors in Chapter 2 to create music. But all you could do with that music, and the DATA numbers which would produce those notes and pauses, is store it on tape or disk. You could later load it to modify the notes, but that was all. There was no way to take those DATA numbers and add them directly to another program. Not without retyping them all.

The next program allows you to take those music files and append them with any other programs simply and easily.

Program 5-10. Disk File Merger

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```
10000 REM ---- GET/OPEN DATA FILE ----      :rem 184
10001 REM                                       :rem 214
10005 PRINT "{ 3 RIGHT } {WHT} {CLR}":INPUT"FILE NAME
      [7]";F$                               :rem 68
10010 OPEN 1,8,4,F$+".MUS,R"                :rem 191
```

```

10011 REM                               :rem 215
10012 REM - FIRST LINE/LINE INCREMENTS- :rem 49
10013 REM                               :rem 217
10015 INPUT "{WHT}STARTING LINE NUMBER[7]"; X
                                           :rem 90
10022 INPUT "{WHT}LINE INCREMENTS[7]"; LI :rem 88
10023 X$=STR$(X)                          :rem 117
10030 L$=RIGHT$(X$,LEN(X$)-1)+" DATA "   :rem 207
10032 REM                                 :rem 218
10033 REM ---- GET MUSIC DATA ----      :rem 190
10034 REM                                 :rem 220
10040 INPUT#1,A$:BB=ST:CT=CT+1            :rem 13
10050 IF BB<>0 THEN 10090                 :rem 220
10060 L$=L$+LEFT$(A$,LEN(A$)-1)+CHR$(44) :rem 72
10061 REM                                 :rem 220
10062 REM ---- DISPLAY DATA LINE ----    :rem 157
10063 REM                                 :rem 222
10070 IF LEN(L$)>76 THEN L$=LEFT$(L$,LEN(L$)-1):PR
      INTL$:X=X+LI:GOTO10023              :rem 111
10080 GOTO 10040                          :rem 39
10090 L$=LEFT$(L$,LEN(L$)-1):PRINTL$     :rem 24
10100 CLOSE 1                             :rem 153

```

If you're using a Datassette instead of a disk drive, replace line 10010 in Program 5-10 with the following line:

```
10010 OPEN 1,1,0,F$+".MUS"
```

Note: You *must* use the extension .MUS on all of the music files you create with either of the sound editors if you want to read them using this program.

LOADing and line numbering. To allow you some flexibility in appending the DATA statements with your own programs, this routine doesn't insert the DATA statements in your BASIC programs. Instead, it simply displays the DATA statements on the screen. This allows you the option of changing them before you put them into the other program.

After you've typed in and run Program 5-10, you'll first be asked to enter the name of the music file. Make sure the disk or tape which includes your music files is inserted in the drive or Datassette. Do not include the extension .MUS in the name when you enter it at this point. The program automatically adds the extension to the filename. This .MUS extension is included so that you won't accidentally try to use a file which does not contain music data created by the sound editors.

After you've entered the filename, the program asks you to supply a starting line number. This should be the beginning line number you want the DATA statements to have once they're appended to your other program. Make sure that you select line ranges that are vacant in the other program, or the DATA statements will overwrite sections when you put the two pieces together. The line numbers for the DATA statements also need to be higher than the last line of the program you'll append them to. For example, if your program runs from line 100 to line 5000, make sure the first DATA statement is higher than 5000. You'll see why you have to do this in a bit. Remember that if you have a program which READS more than one set of DATA, the DATA needs to be in the same order as the READ statements. In other words, if the DATA statements which include the music values are at the end of the program (as they should be; see above), the line which READs those values should be the last such statement in the program.

Finally, Program 5-10 asks you to enter the line increment value. This will determine the numbering sequence. If you enter 5010 for your starting line number and an increment value of 10, the DATA statements will be numbered 5010, 5020, 5030, and so on.

Appending DATA. RUN Program 5-10, specify the music filename, the beginning line number, and the increment value. The DATA statements will begin to appear on the screen. Once nine DATA lines are displayed, which is the safe maximum to have on the screen at one time, press the RUN/STOP key. This will stop the program and give you the opportunity to create a new program file consisting only of DATA statements. To do this, first type NEW. This erases everything in the computer's memory. Don't worry; you haven't lost the DATA lines. As long as they're on the screen, you can still retrieve them. To do this, move the cursor key to the beginning of the first DATA line and press RETURN. Repeat this for all nine displayed lines. Next, SAVE these nine lines to tape or disk, making sure to use a new filename. A good idea would be to use the same name as the file created by the sound editor, but dropping the .MUS extension. The program is now a program file, not a sequential file, and so can be appended to yet another program.

If you have more than nine lines of DATA statements,

you'll have to repeat the process. LOAD Program 5-10 again, call the same music file (the one that had the .MUS extension) from tape or disk, and specify the same beginning and increment values as you did earlier. When the program begins to display the DATA lines, let the first nine scroll off the screen so that you're looking at the next group of nine lines. Type NEW again, LOAD the program file you've created which has the first nine DATA lines in it, move the cursor up to the first line you see on the screen (which is actually the tenth DATA line), and hit RETURN to add the next nine lines to the first nine. SAVE this longer version of your DATA program file by using SAVE"@0:filename",8 if you have a disk drive. If you're using a Datassette, create another file called *filename.1*, or something like that. Now the program file has 18 lines of DATA. You can LIST it to make sure.

Repeat this process as often as necessary, adding nine lines to the DATA program file in each step. When you finish, you should have a program file on tape or disk that is a number of DATA lines.

Let's go through an example. It may make it easier to understand.

You've used the "One-Voice Sound Editor" program from Chapter 2 and created a music file consisting of seventy-odd notes. Using the SAVE option in the program, you saved the note, length, and pause values to disk. SCALE.MUS is the music file's name. (Remember that when you save music files and want to later append them to other programs using Program 5-10, you must use the .MUS extension.)

After LOADING and RUNNING Program 5-10, you enter SCALE as the filename, line 20000 as the starting line, and 10 as the increment value. You picked 20000 as the starting line because you know that the program you want to append the DATA statements to runs from lines 100 to 19000. The DATA lines will begin to display on the screen. As soon as line 20080 (the ninth line) shows on the screen, hit the RUN/STOP key. The screen should look something like Figure 5-1. All you see on the screen now is a series of DATA statements. You first type NEW, which erases the program in the 64's memory. The screen scrolls up, but all nine DATA lines are still on the screen. Move the cursor key up to line 20000 and hit RETURN. The cursor moves down to line 20010. Repeat this until all nine lines have been entered. Type LIST, and the

Figure 5-1. Appending First Nine Lines

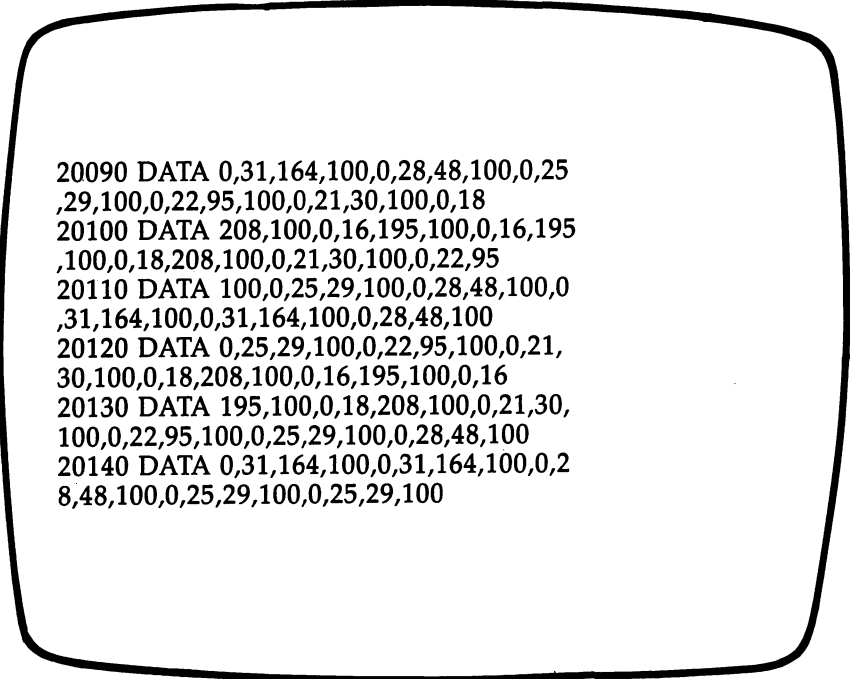
```
FILE NAME ? SCALE
STARTING LINE NUMBER ? 20000
LINE INCREMENTS ? 10
20000 DATA 16,195,100,0,18,208,100,0,21,
30,100,0,22,95,100,0,25,29,100,0,28,48
20010 DATA 100,0,31,164,100,0,31,164,100
,0,28,48,1,0,25,29,100,0,22,95,100
20020 DATA 0,21,30,100,0,18,208,100,0,16
,195,100,0,16,195,100,0,18,208,100,0
20030 DATA 21,30,100,0,22,95,100,0,25,29
,100,0,28,48,100,0,31,164,100,0,31,164
20040 DATA 100,0,28,48,100,0,25,29,100,0
,22,95,100,0,21,30,100,0,18,208,100,0
20050 DATA 16,195,100,0,16,195,100,0,18,
208,100,0,21,30,100,0,22,95,100,0,25
20060 DATA 29,100,0,28,48,100,0,31,164,1
00,0,31,164,100,0,28,48,100,0,25,29,100
20070 DATA 0,22,95,100,0,21,30,100,0,18,
208,100,0,16,195,100,0,16,195,100,0,18
20080 DATA 208,100,0,21,30,100,0,22,95,1
00,0,25,29,100,0,28,48,100,0,31,164,100
BREAK IN 10070
READY.
```

DATA lines 20000–20080 appear on the screen. Now you can save this program file to tape or disk, using the name SCALE (without the .MUS extension).

But there was more DATA than the nine lines could contain. So you have to repeat the process. LOAD Program 5-10 again, enter SCALE.MUS as the filename, 20000 as the line number, and 10 as the increment value, just as you did before. This time, however, let the DATA lines scroll up until line 20090 is near the top of the screen. It should look like Figure 5-2.

Type NEW to erase Program 5-10 from memory. LOAD SCALE from tape or disk, and cursor up to line 20090. Hitting the RETURN key six times completes the process. LIST the program and you should see all the DATA lines, from 20000 to 20140.

Figure 5-2. Next Nine Lines



```
20090 DATA 0,31,164,100,0,28,48,100,0,25
,29,100,0,22,95,100,0,21,30,100,0,18
20100 DATA 208,100,0,16,195,100,0,16,195
,100,0,18,208,100,0,21,30,100,0,22,95
20110 DATA 100,0,25,29,100,0,28,48,100,0
,31,164,100,0,31,164,100,0,28,48,100
20120 DATA 0,25,29,100,0,22,95,100,0,21,
30,100,0,18,208,100,0,16,195,100,0,16
20130 DATA 195,100,0,18,208,100,0,21,30,
100,0,22,95,100,0,25,29,100,0,28,48,100
20140 DATA 0,31,164,100,0,31,164,100,0,2
8,48,100,0,25,29,100,0,25,29,100
```

To complete your creation of the program file SCALE, type SAVE"@0:SCALE",8 for disk or SAVE"SCALE.1" if you're using a Datassette. You've now got a program file called SCALE (or SCALE.1) which is a 15-line set of DATA statements.

But you can't use this program file on its own to produce music. You have to somehow append it to a longer program. This last step is easy, even though it involves a number of steps.

First of all, LOAD the program you want to append the DATA statements to. Remember that this program's highest line number should be *lower* than the lowest line number of the DATA statement program file. Then, in direct mode (without line numbers), enter the following commands, hitting RETURN after each line:

```
POKE 43,PEEK(45)-2  
POKE 44,PEEK(46)  
LOAD"filename",8 (or just "filename" if using tape)  
POKE 44,8:POKE 43,1
```

Substitute the name of your DATA program file for *filename*. Make sure the disk or tape that includes the DATA program file is in the drive or Datassette before you hit RETURN for the third line of commands. The drive or Datassette will run for a while. Then enter the last two POKE commands.

As long as the second program's line numbers are all higher than the highest line number of the first program, this method of appending programs will work. That's why we used line numbers starting at 20000 for the DATA program file example discussed earlier. After you've entered the above commands, SAVE the appended program to tape or disk. Your music DATA values are now part of your longer program or game.

For instance, continuing with our example of SCALE, you would load the longer program (say "GAME", with line numbers from 1000 to 19000), then enter the first two lines of commands. The third line would read LOAD"SCALE",8 for disk, LOAD"SCALE.1" for tape. Once you've entered the last two POKES, you could save the appended program as "GAME.2", or something like that.

How to Append DATA

As a handy reference, here are the steps you need to go through to turn the music files created by your sound editors into sections of another program.

1. LOAD and RUN Program 5-10. Specify filename (without .MUS extension), starting line, and line increment.

2. Let Program 5-10 display nine lines of DATA, then hit RUN/STOP.

3. Type NEW. Cursor up to first line, then hit RETURN. Repeat for all nine lines onscreen.

4. SAVE nine lines as the program file.

5. If you have more than nine lines of DATA, LOAD Program 5-10 again, using the same filename, starting line, and increment. Let the screen scroll until the next nine lines are displayed.

6. Type NEW.

7. LOAD DATA program file created in the fourth step, cursor up to the first line displayed on screen, then hit RETURN. Repeat for all other lines onscreen.

8. SAVE DATA program file again, using "@0:filename",8 or "filename.1".

9. When all the DATA statements are in the program file, append it to your other program by LOADING the first program and entering the four lines of direct mode commands.

10. SAVE the appended version using SAVE and REPLACE or a new filename.

Putting the DATA to Work

If the music DATA that you're retrieving and appending to other programs was made using the One-Voice Sound Editor, the DATA is in the following order:

- TONE (High Pitch Value)
- TONE (Low Pitch Value)
- LENGTH (Of Note)
- LENGTH (Of Pause—if any)

You have to have a way to play the music, however. You can either refer to the many examples earlier in this book that READ DATA to create sounds and music, or you can use a routine such as this one to READ the DATA statements and play them back.

Program 5-11. READING Routine

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1000 REM :rem 165
1010 REM * INITIALIZE SOUND REGISTERS * :rem 45
1020 REM :rem 167
1030 FOR R=54272 TO 54296:POKER,0:NEXT :rem 123
1040 POKE 54296,15:POKE 54275,8:POKE 54278,240 :rem 153
:rem 149
1060 POKE 54276,65 :rem 149
1070 REM :rem 172
1080 REM **** READ MUSIC DATA **** :rem 180
1090 REM :rem 174
1100 READ H,L,LE,P:IF H=999 THEN 1210 :rem 195
1110 REM :rem 167
1120 REM **** PLAY MUSIC **** :rem 175
1130 REM :rem 169
1140 POKE 54272,L:POKE 54273,H :rem 83
1150 FOR R=0 TO 1*(P+LE):NEXT :rem 65
1160 POKE 54272,0:POKE 54273,0 :rem 33
1170 GOTO 1100 :rem 196
1180 REM :rem 174
1190 REM **** TURN OFF SOUND **** :rem 172
1200 REM :rem 167
1210 POKE 54276,64:POKE 54296,0 :rem 96
1300 REM **** SAMPLE DATA **** :rem 212
1306 REM :rem 174
1310 DATA 33,134,100,0,33,134,100,0,37,161,100,0,4 :rem 212
      2,60,100,0,33,134,100,0,42,60
1320 DATA 100,0,37,161,100,0,25,29,100,0,33,134,10 :rem 212
      0,0,33,134,100,0,37,161,100,0
1330 DATA 42,60,100,0,33,134,100,0,0,0,0,100,31,16 :rem 134
      4,100,0,0,0,0,100,33,134,100
1340 DATA 0,33,134,100,0,37,161,100,0,42,60,100,0, :rem 120
      44,191,100,0,42,60,100,0,37
1350 DATA 161,100,0,33,134,100,0,31,164,100,0,25,2 :rem 84
      9,100,0,28,48,100,0,31,164
1360 DATA 100,0,33,134,100,0,0,0,0,100,33,134,100, :rem 128
      0
1370 REM :rem 175
1380 REM *** DUMMY DATA TO END SONG *** :rem 3
1390 REM :rem 177
1393 DATA 999,999,999,999 :rem 26

```

The DATA statements at the end of this routine (lines 1310–1360) are there to demonstrate the way the program works. Ordinarily, *your* music DATA numbers would be here. The part of the program that reads and plays the music is in lines 1100–1210. Notice that line 1393 contains four extra

DATA numbers (999,999,999,999). These are used by the program to indicate the end of the song. When the program finds the 999's, it jumps to the routine in line 1210.

To use the "Chord Editor," and append DATA program files, follow the directions given earlier. Use Program 5-10, create the DATA program file, and append it to the other program. You'll use the same set of direct mode commands to append the two.

Playing back music created by the Chord Editor from Chapter 2 is similar to the process you used above. However, now the DATA is in a seven-number grouping:

- Voice 1 (Low Pitch Value)
- Voice 1 (High Pitch Value)
- Voice 2 (Low Pitch Value)
- Voice 2 (High Pitch Value)
- Voice 3 (Low Pitch Value)
- Voice 3 (High Pitch Value)
- Length (Of Note)

The following program has seven dummy values (999's). Remember that this program will not play music that was created using the One-Voice Sound Editor because the two DATA formats are incompatible.

Program 5-12. Chord Editor READER

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

10 REM ** INITIALIZE SOUND REGISTERS **           :rem 32
20 REM                                           :rem 70
30 FOR R=54272 TO 54296:POKER,0:NEXT             :rem 26
39 POKE 54296,15                                 :rem 55
40 POKE 54275,8:POKE 54278,240                   :rem 51
41 POKE 54282,8:POKE 54285,240                   :rem 48
42 POKE 54289,8:POKE 54292,240                   :rem 54
44 REM                                           :rem 76
45 REM *** GET DATA ***                         :rem 67
46 REM                                           :rem 78
50 READ L1,H1,L2,H2,L3,H3,LE                     :rem 2
60 IF L1=999 THEN 140                             :rem 30
64 REM                                           :rem 78
65 REM *** PLAY NOTE ***                         :rem 183
66 REM                                           :rem 80
70 POKE 54272,L1:POKE 54273,H1                   :rem 86
80 POKE 54279,L2:POKE 54280,H2                   :rem 94
90 POKE 54286,L3:POKE 54287,H3                   :rem 102

```

```
101 POKE 54290,65 :rem 92
102 POKE 54283,65 :rem 95
103 POKE 54276,65 :rem 98
104 REM :rem 121
105 REM *** PLAY FOR LENGTH OF NOTE ** :rem 246
106 REM :rem 123
110 FOR R=0 TO LE*15:NEXT :rem 117
114 REM :rem 122
115 REM *** TURN OFF NOTE *** :rem 209
116 REM :rem 124
120 POKE 54290,64 :rem 92
121 POKE 54283,64 :rem 95
122 POKE 54276,64 :rem 98
130 GOTO 50 :rem 50
134 REM :rem 124
135 REM *** END SONG *** :rem 135
136 REM :rem 126
140 POKE 54290,64 :rem 94
141 POKE 54283,64 :rem 97
142 POKE 54276,64 :rem 100
1305 REM **** DUMMY TONE VALUES **** :rem 143
1306 REM :rem 174
1307 DATA 999,999,999,999,999,999,999 :rem 154
```

To hear this routine play music, you have to create a music file with the Chord Editor, change it to a program file, and finally append it to this. Of course, you'd replace the seven dummy values with your own set of DATA statements. Since the values will be in the proper format, all you have to do is append the DATA statements to the end of this program, and you'll hear the chord combinations.

POKEing Music into Memory

There's another method of using the music data to create sound, however. Instead of changing the music file produced by one of the two sound editors into a DATA program file, then appending the DATA statements to another program, you can POKE the values into the 64's memory, and then have the program PEEK those locations for you. Once you've PEEKed those addresses, you can POKE the values back into the correct sound control registers. This method may not be appropriate for every program you write that uses sound, but it is an interesting technique. It sounds harder than it is.

You have to be careful where you place the music DATA, however. If you POKE it into an area of memory that is used

by your BASIC program, for example, it can be overwritten. A safe place for this DATA begins with location 49152. BASIC will never harm this area. Be aware, however, that many machine language programs and routines use 49152 as a starting point for placing values. If your program includes some machine language routines, make sure you're not trying to use these locations for two purposes.

Program 5-13 takes the music files you created with a sound editor and POKES the DATA into memory, starting at location 49152. You don't have to change the music files to DATA program files. Nor do you have to actually append them to your program. The process is much simpler.

First of all, type in and RUN Program 5-13. When it asks you for the filename, supply it, making sure not to include the .MUS extension. (As with the first method of using music files, this technique requires that they be originally created with the extension when you SAVE them using the sound editors.) The program will load the music file and POKe the values into memory, beginning at location 49152. It will also display the last location it used to store the values. You'll need this number later. Without turning off the computer, LOAD the other program, the one that will actually play the music. We'll see an example of this kind of program in a moment. First, here's Program 5-13:

Program 5-13. POKeIng Music

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1000 PRINT "{WHT}{CLR}":INPUT"FILE NAME[7]";F$
                                     :rem 184
1010 OPEN 1,8,4,F$+".MUS,R"         :rem 143
1020 CT=49152                         :rem 156
1040 INPUT#1,A:BB=ST                  :rem 184
1050 POKe CT,A                        :rem 249
1060 CT=CT+1                          :rem 142
1070 IF BB<>0 THEN 1100               :rem 118
1080 GOTO 1040                         :rem 199
1100 CLOSE 1:PRINT CT                 :rem 199

```

If you're using a Datassette instead of a disk drive, change line 1010 in this program to:

```
1010 OPEN 1,1,0,F$+".MUS"
```

Program 5-13 POKeS the music values into the available

memory space starting at location 49152 and then shows the last address used. If you wanted, you could include this routine as part of a longer program and have it load and POKE the values directly. All you would have to do is set the final address as another variable, and then later use that variable to stop the music. As long as both the longer program and the music file are on the same disk (this technique would be too difficult to use with tape), the program would automatically load and POKE the values, then use them to play the tune.

Program 5-14 is an example of a routine you could use that would actually play the music. This program will play only music you created through the One-Voice Sound Editor. Note that you'll have to change the XXXXX characters in line 1105 to your own final address (as displayed for you by Program 5-13), in order to stop the music. If you don't put a value here (and eliminate line 1105 entirely), you will first hear your tune, then a series of random notes. What's happening here is that the program is reading *all* of the computer's memory from location 49152 on, PEEKing the values that are normally there, and POKEing them into the higher and lower pitch control registers. The computer is playing its own memory.

Program 5-14. PEEKing Values—One-Voice Sound Editor

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1000 REM                                     :rem 165
1010 REM * INITIALIZE SOUND REGISTERS *     :rem 45
1020 REM                                     :rem 167
1030 FOR R=54272 TO 54296:POKER,0:NEXT      :rem 123
1040 POKE 54296,15:POKE54275,8:POKE54278,240:CT=49
      152                                     :rem 172
1060 POKE 54276,65                           :rem 149
1070 REM                                     :rem 172
1080 REM **** READ MUSIC DATA ****         :rem 180
1090 REM                                     :rem 174
1100 H=PEEK(CT):L=PEEK(CT+1):LE=PEEK(CT+2):P=PEEK(
      CT+3):CT=CT+4                           :rem 40
1105 IF CT>XXXXXX THEN 1210                 :rem 214
1110 REM                                     :rem 167
1120 REM **** PLAY MUSIC ****              :rem 175
1130 REM                                     :rem 169
1140 POKE 54272,L:POKE 54273,H             :rem 83
1150 FOR R=0 TO 1*(P+LE):NEXT              :rem 65

```

```

1160 POKE 54272,0:POKE 54273,0           :rem 33
1170 GOTO 1100                           :rem 196
1180 REM                                 :rem 174
1190 REM **** TURN OFF SOUND ****       :rem 172
1200 REM                                 :rem 167
1210 POKE 54276,64:POKE 54296,0        :rem 96

```

Program 5-15 is a similar program for the Chord Editor. It demonstrates how to play the music using the values earlier POKEd into memory. The main difference between this program and the previous routine is that seven values are PEEKed instead of four to play the notes and a pause. Again, make sure you change line 1105 to the highest address displayed by Program 5-13 so that the music stops.

Program 5-15. PEEKing Values—Chord Editor

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1000 REM                                 :rem 165
1010 REM * INITIALIZE SOUND REGISTERS *   :rem 45
1020 REM                                 :rem 167
1030 FOR R=54272 TO 54296:POKER,0:NEXT    :rem 123
1040 POKE 54296,15                       :rem 144
1045 POKE 54275,8:POKE 54278,240        :rem 153
1050 POKE 54282,8:POKE 54285,240        :rem 145
1055 POKE 54289,8:POKE 54292,240        :rem 155
1065 CT=49152                             :rem 165
1070 REM                                 :rem 172
1080 REM **** READ MUSIC DATA ****      :rem 180
1090 REM                                 :rem 174
1100 L1=PEEK(CT):H1=PEEK(CT+1):L2=PEEK(CT+2):H2=PEEK(CT+3) :rem 157
1102 L3=PEEK(CT+4):H3=PEEK(CT+5):LE=PEEK(CT+6):CT=CT+7 :rem 200
1105 IF CT>XXXXXX THEN 1210             :rem 214
1110 REM                                 :rem 167
1120 REM **** PLAY MUSIC ****           :rem 175
1130 REM                                 :rem 169
1140 POKE 54272,L1:POKE 54273,H1        :rem 181
1141 POKE 54279,L2:POKE 54280,H2        :rem 189
1142 POKE 54286,L3:POKE 54287,H3        :rem 197
1150 POKE 54276,65                       :rem 149
1151 POKE 54283,65                       :rem 148
1152 POKE 54290,65                       :rem 147
1158 REM                                 :rem 179
1159 REM ** PLAY FOR LENGTH OF NOTE **   :rem 6
1160 REM                                 :rem 172

```

```

1165 FOR R=0 TO 100*LE:NEXT           :rem 219
1167 REM                               :rem 179
1169 REM ** TURN OFF NOTE **         :rem 183
1170 REM                               :rem 173
1171 POKE 54276,64:POKE 54283,64:POKE 54290,64
                                         :rem 159
1172 GOTO 1100                         :rem 198
1180 REM                               :rem 174
1190 REM **** TURN OFF SOUND ****    :rem 172
1200 REM                               :rem 167
1210 POKE 54276,64:POKE 54283,64:POKE 54290,64:POK
    E 54296,0                           :rem 104

```

With two methods of using the sound editors' values available to you, it shouldn't be difficult to append music to your own programs. Whatever technique you decide to use depends on the circumstances of your own programming situation. Either one might work, or one might be easier than the other.

Adding sound, even complex musical pieces, isn't hard. And mixing sound with graphics is almost as easy. The last program in this chapter shows an example of how it all works together.

Sound Game

To illustrate the way that some of the components of sound and animation can work together, here's a simple game that uses many of the features we've covered. After you've typed it in and run it, take a closer look at the program listing. You're certain to find ways to use sound, or ways to modify the existing sounds, that I haven't thought of. That's half the fun of programming sound (or anything) on the Commodore 64. You can learn almost as much from experimenting with other people's programs as you can by writing your own.

Program 5-16. A Sound Game

For mistake-proof program entry, be sure to read "Automatic Proofreader," Appendix C.

```

1 REM ** INITIALIZE SOUND REGISTERS **   :rem 240
2 REM                                     :rem 22
3 FOR R=54272 TO 54296:POKE R,0:NEXT:POKE 54275,8:
    POKE 54278,240:POKE 54282,8           :rem 197
4 POKE 54285,240:POKE 54289,8:POKE 54292,240
                                         :rem 55

```

5
Putting It All Together

```
5 REM :rem 25
6 REM **** SET UP DISPLAY SCREEN **** :rem 209
7 REM :rem 27
10 PRINT "{CLR}":F=5: SH=53262: TN=1 :rem 21
11 POKE 53280,0:POKE 53281,0 :rem 183
12 PRINT "{HOME}{RVS}{12 DOWN}{7 RIGHT}S P A C E
{4 SPACES}S H O O T E R{OFF}" :rem 18
13 REM :rem 72
14 REM ** PROGRAM SPRITE DATA ** :rem 250
15 REM :rem 74
16 FOR R=12800 TO 13311:READA:POKER,A:NEXT:rem 172
17 REM :rem 76
18 REM -- PUT SPRITES ON SCREEN -- :rem 129
19 REM :rem 78
20 POKE 53269,255 :rem 98
21 FOR P=0TO7:POKE P+2040, P+200: NEXT :rem 61
25 FOR G=53287 TO 53295: POKE G,14:NEXT :rem 64
26 REM :rem 76
27 REM ---- SPRITE POSITIONS ---- :rem 84
28 REM :rem 78
30 POKE 53248, 160: POKE 53249,229 :rem 148
31 POKE 53248, 160: POKE 53249,229 :rem 149
32 POKE 53250, 160: POKE 53251,229 :rem 136
33 POKE 53252, 30 : POKE 53253,116 :rem 84
34 POKE 53254, 240: POKE 53255,110 :rem 134
35 POKE 53256, 240: POKE 53257,110 :rem 139
36 POKE 53258, 30 : POKE 53259,116 :rem 99
37 POKE 53260, 255: POKE 53261,116 :rem 143
38 POKE 53262, 30 : POKE 53263,116 :rem 91
39 R=70 :rem 98
40 REM :rem 72
41 REM ---- SPRITE COLORS ---- :rem 90
42 REM :rem 74
43 POKE 53287, 1 :rem 252
44 POKE 53288, 14 :rem 50
45 POKE 53289, 5 :rem 4
46 POKE 53290, 2 :rem 250
47 POKE 53291, 7 :rem 1
48 POKE 53292, 1 :rem 253
49 POKE 53293, 12 :rem 49
50 POKE 53294, 12 :rem 42
55 REM :rem 78
56 REM ---- EXPAND RINGED PLANET ---- :rem 244
57 REM :rem 80
58 POKE 53271, 24: POKE 53277, 24 :rem 51
60 GOTO 1800 :rem 104
71 REM :rem 76
72 REM **** MOVE SHIP ROUTINE **** :rem 46
73 REM :rem 78
```


5
Putting It All Together

```
74 POKE SH,PEEK(SH)+F :rem 227
75 IF NT=3 THEN TN=TN*-1:NT=0 :rem 142
76 MU=150+(50*TN) :rem 143
77 POKE 54286,MU: NT=NT+1 :rem 139
78 IF ((SH=53260) AND (PEEK(SH)<30)) THEN:SH=53262
:F=5:POKE 53260,255:GOTO 89 :rem 150
79 IF ((SH=53262) AND (PEEK(SH)>250)) THEN:SH=5326
0:POKE 53262,30:F=-5 :rem 223
81 REM :rem 77
82 REM -- CHECK FOR FIRED LAUNCHER -- :rem 3
83 REM :rem 79
89 IFFR=1THENPOKE53251,PEEK(53251)-10:POKE54276,12
9:POKE54296,15:POKE 54276,128 :rem 10
91 REM :rem 78
92 REM - CHECK IF ROCKET IS OFF SCREEN- :rem 149
93 REM :rem 80
95 IF PEEK(53251)<20 THEN FR=0: POKE 53251,225: PO
KE 54276,0 :rem 64
101 REM :rem 118
102 REM -- MOVE ROCKET LAUNCHER -- :rem 124
103 PP=53250 :rem 112
104 GET A$ :rem 218
106 IF A$="{RIGHT}"ANDPEEK(PP)<225THENPOKE53248,PE
EK(53248)+5:POKEPP,PEEK(PP)+5 :rem 254
107 IFA$="{LEFT}"ANDPEEK(53248)>50THENPOKE53248,PE
EK(53248)-5:POKEPP,PEEK(PP)-5 :rem 183
108 IF A$="{F1}"THEN FR=1 :rem 200
170 REM :rem 124
180 REM * CHECK FOR ROCKET/COLLISION ** :rem 227
190 REM :rem 126
195 A=PEEK(53278):IF A=254 THEN 220 :rem 65
210 GOTO 74 :rem 55
220 POKE 54296,15: POKE 54280,10:POKE54283,129
:rem 143
222 FOR R=0 TO 25:POKE 53281,R: POKE 54280, R: NEX
T:POKE 54283,128 :rem 201
225 POKE 54280, 2: POKE 53281, 0 :rem 241
230 FR=0: POKE 53251,225: POKE 54276,0 :rem 149
370 REM :rem 126
380 REM ** SCORE SUBROUTINE ** :rem 179
390 REM :rem 128
430 SR=SR+1:PRINT"{HOME}{2 DOWN}{2 RIGHT}SCORE=";S
R :rem 141
435 IF SR=5 THEN 2200 :rem 53
440 GOTO 74 :rem 60
1000 REM **** SPRITE #1 **** :rem 32
1010 REM :rem 166
1020 DATA 0,60,0,0,36,0,0,36,0,0,36,0,0,36,0,0,36,
0,0,36,0,0,66,0,0,255,0,0,66 :rem 141
```

Putting It All Together

```

1030 DATA 0,0,255,0,0,66,0,0,255,0,0,66,0,0,255,0,
      0,66,0,0,255,0,255,255,255 :rem 86
1040 DATA 248,0,31,255,0,255,255,255,255,0 :rem 9
1050 REM :rem 170
1060 REM **** SPRITE #2 **** :rem 39
1070 REM :rem 172
1080 DATA 0,24,0,0,24,0,0,24,0,0,24,0,0,24,0,0,24
      :rem 23
1090 DATA 0,0,24,0,0,52,0,0,110,0,0,110,0,0,110,0,
      0,110,0,0,110,0,0,110,0,0,110 :rem 135
1100 DATA 0,0,110,0,0,110,0,0,110,0,0,239,0,1,239,
      128,1,153,128,0 :rem 35
1110 REM :rem 167
1120 REM **** SPRITE #3 **** :rem 37
1130 REM :rem 169
1140 DATA 0,255,0,3,255 :rem 91
1150 DATA 192,13,252,112,30,115,248,63,143,252,127
      ,255,250,127,127,118,190,119 :rem 252
1160 DATA 207,239,233,63,255,206,255,255,223,191,2
      47,159,31,127,191,158,119,207 :rem 75
1170 DATA 206,63,243,252,31,253,248,15,62,240,3,25
      5,64,0,255,128,0,0,0,0,0,0 :rem 114
1171 DATA 255 :rem 128
1180 REM :rem 174
1190 REM **** SPRITE #4 **** :rem 45
1200 REM :rem 167
1210 DATA 0,0,0,0,0,8,0,0,20,0,0,40,0,70,80,1,143,
      160,3,30,64,2,60,96,4,120,224 :rem 201
1220 DATA 0,241,192,1,227,144,3,199,48,1,142,96,1,
      124,192,2,185,128,5,0,0,10,0 :rem 221
1230 DATA 0,20,0,0,40,0,0,16,0,0,0,0,0,0,0,0 :rem 89
1240 REM :rem 171
1250 REM **** SPRITE #5 **** :rem 43
1260 REM :rem 173
1270 DATA 0,0,0,0,0,12,0,0,28,0,0,56,0,126,112
      :rem 145
1280 DATA 1,255,224,3,255,192,3,255,224,7,255,240,
      7,255,240,7,255,240,3,255,240 :rem 44
1290 DATA 1,255,224,1,255,192,3,191,0,7,0,0,14,0,0
      ,28,0,0,56,0,0,48,0,0,0,0,0,0 :rem 211
1300 REM :rem 168
1310 REM **** SPRITE #6 **** :rem 41
1320 REM :rem 170
1340 DATA 0,255,0,3,255,192,15,255,240,31,255,248,
      63,255,252,127,255,254,127 :rem 165
1340 DATA 255,254,255,255,255,255,255,255,255,255,
      255,255,255,255,255,255 :rem 253
1350 DATA 127,255,254,127,255,254,63,255,252,31,25
      5,248,15,255,240,3,255,192,0 :rem 12
1360 DATA 255,0,0,0,0,0,0,0,255 :rem 204
1370 REM :rem 175
198

```



```
2210 IF L1=999 THEN POKE 54276,0:POKE 54283,0:POKE
      54287,2:POKE54273,99:GOTO4000      :rem 90
2215 POKE 54272,L1: POKE 54273, U1      :rem 198
2220 POKE 54279,L2: POKE 54280, U2      :rem 201
2230 POKE 54286,L3: POKE 54287, U3      :rem 209
2240 FOR M=0 TO L*5: NEXT                :rem 48
2250 GOTO 2205                            :rem 203
3000 REM                                  :rem 167
3002 REM ----- FANFARE DATA -----   :rem 120
3003 REM                                  :rem 170
3005 DATA 195,16,134,33,195,16,10,208,18,208,18,16
      1,37,10,95,22,95,22,191,44        :rem 161
3010 DATA 10,30,21,30,21,60,42,10,208,18,208,18,16
      1,37,10,29,25,29,25,58,50,10     :rem 212
3020 DATA 29,25,29,25,58,50,2,58,50,58,50,29,25,2,
      58,50,58,50,29,25,20,0,0,0      :rem 139
3030 DATA 0,0,0,20,58,50,29,25,58,50,10,97,56,48,2
      8,48,28,10,195,16,134,33,134    :rem 245
3040 DATA 33,10,72,63,164,31,164,31,10,48,28,97,56
      ,48,28,10,161,37,161,37,208    :rem 207
3050 DATA 18,10,161,37,161,37,208,18,2,161,37,208,
      18,161,37,2                      :rem 178
3060 DATA 999,999,999,999,999,999,999 :rem 152
4000 PRINT"{HOME}{2 DOWN}{2 RIGHT}YOU WIN!!!": POK
      E 54296,0                          :rem 33
4010 GOTO 4010                            :rem 195
```

Use the Left/Right Cursor key and the SHIFT key to move the missile base at the bottom of the screen. Pressing the SHIFT key at the same time as the cursor key moves the base to the left; an unSHIFTed cursor key moves the base right. To fire a missile at the gray target ship, hit the f1 key.

Much of the program is taken up in the creation of the sprites which move across the screen. Since we're mainly concerned with sound, we'll describe the sound sections in more detail than the graphics.

Once the sound control registers are initialized in lines 3 and 4, the program creates the screen display and the sprites that will move across that display. As soon as the sprites are set up, the program shifts to line 1800, which starts the opening music routine.

Using all three voices, a series of DATA statements (lines 2000-2040) are READ (line 1805), and the values POKEd into the appropriate pitch control registers (lines 1815-1830). The length of the pause between chords is set in line 1840 by the FOR-NEXT loop. The pulse waveform is enabled for all three voices and the volume is set to maximum in line 1800. Once

the tune has played, the program returns to line 74 to move the target ship between the planets.

The background sound you hear as the ship moves back and forth is created in lines 76 and 77. Both the sound and the sprite movement take place in the same loop, so it seems as if they're happening simultaneously. The variable MU is set to a value of 100 or 200 by line 76. Every third time through the loop, MU switches from 100 to 200, or from 200 to 100. Line 77 takes this value and POKEs it into the high pitch control register for voice 3. What all this does is make the background sound alternate quickly between a high and low frequency producing the droning sound you hear.

When you press the f1 key, the missile is launched. As it moves, it makes yet another sound. This is created in line 89 by simply turning on and off the noise waveform in voice 1. You don't have to set ADSR or frequency values for this sound, because values are still stored in those control registers. Line 1810, for instance, includes a pitch value for voice 1, and since the sound registers aren't cleared in the middle of the program, the ADSR values initialized in lines 3 and 4 are still available. Leaving some registers with values can be convenient at times, although it may give you problems if you think they've been cleared when you're trying to debug a program. Use the technique carefully.

When the missile hits the ship, a short explosion sound, created by lines 220 and 222, plays. Using voice 2 and the noise waveform, the sound plays for only a brief moment. As soon as the FOR-NEXT loop in line 222 completes, the sound falls off.

When you've destroyed five ships, the game ends. Another piece of music plays, this time accessed from lines 2200-2250. All the Commodore 64's voices are used. The DATA in lines 3005-3060 is READ in line 2205 and then POKEd into the pitch control registers. A delay loop in line 2240 sets the length of each group of notes.

As you can hear, the sounds fit well into the theme of the game. Explosions, movement sounds, and opening and closing music add much to the game's "feel." It's not a complicated, arcade-style game, but it's not meant to be. The important thing to remember is that adding sound to movement and graphics doesn't have to be difficult. Even the simplest sound and music make a program seem more professional, and more fun to use.

Using What You've Learned

Your Commodore 64 is important to you. If it wasn't, you probably wouldn't have bought it. You enjoy playing games on it and using it for home applications. It may even be an educational tool. But if you're like many 64 users, you enjoy programming on it most of all.

Now you've got yet another tool to use—sound and music. You've seen how simple sounds can be created, how sound editors can help you produce more elaborate sound effects, and even how to use some of the advanced functions of the SID chip.

But you're probably just beginning to use sound and music on your 64. Getting comfortable with this new tool takes time, and especially practice. Just like any other skill, you'll get better as you spend time programming sound and music on your computer. But there are some things you can do that may help.

Look at other programs published in magazines or books and see if you can find how the programmer used sound and music. Oftentimes, someone else will think of a way to create or use sound that you haven't thought of. On the other hand, you may look at a published program and wonder why the programmer used such an inefficient technique. Change the sound and music you see in other programs and see what effects you can create. You may end up with a better program or game than was originally published.

Don't be afraid to experiment. As long as you have a copy of your program on tape or disk, you can't lose anything in trying. You may find an interesting sound by accident.

Above all, listen to other programs and games, especially commercial software. You probably can't change anything, but it may give you ideas of ways to use sound and music that you or I haven't considered.

Sound is an important aspect of our lives that we often take for granted. It's somewhat like that in computer programming, too. Sound and music are sometimes secondary to other programming skills. It doesn't have to be like that. With this book, the knowledge you now have, and your own imagination, you can make sound an important and impressive part of any program.

Appendices



A Beginner's Guide to Typing In Programs

What Is a Program?

A computer cannot perform any task by itself. Like a car without gas, a computer has *potential*, but without a program, it isn't going anywhere. Most of the programs published in this book are written in a computer language called BASIC. BASIC is easy to learn and is built into all Commodore 64s.

BASIC Programs

Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, or number is significant. A common mistake is substituting a letter such as O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, you must enter all punctuation such as colons and commas just as they appear in the book. Spacing can be important. To be safe, type in the listings *exactly* as they appear.

Braces and Special Characters

The exception to this typing rule is when you see the braces, such as {DOWN}. Anything within a set of braces is a special character or characters that cannot easily be listed on a printer. When you come across such a special statement, refer to Appendix B, "How to Type In Programs."

About DATA Statements

Some programs contain a section or sections of DATA statements. These lines provide information needed by the program. Some DATA statements contain actual programs (called machine language); others contain graphics codes. These lines are especially sensitive to errors.

If a single number in any one DATA statement is mistyped, your machine could lock up, or crash. The keyboard and STOP key may seem dead, and the screen may go blank. Don't panic—no damage is done. To regain control, you have

to turn off your computer, then turn it back on. This will erase whatever program was in memory, *so always SAVE a copy of your program before you RUN it*. If your computer crashes, you can load the program and look for your mistake.

Sometimes a mistyped DATA statement will cause an error message when the program is RUN. The error message may refer to the program line that READs the data. *The error is still in the DATA statements, though.*

Get to Know Your Machine

You should familiarize yourself with your computer before attempting to type in a program. Learn the statements you use to store and retrieve programs from tape or disk. You'll want to save a copy of your program, so that you won't have to type it in every time you want to use it. Learn to use your machine's editing functions. How do you change a line if you made a mistake? You can always retype the line, but you at least need to know how to backspace. Do you know how to enter reverse video, lowercase, and control characters? It's all explained in your computer's manuals.

A Quick Review

1. Type in the program a line at a time, in order. Press RETURN at the end of each line. Use the DEL key to correct mistakes.
2. Check the line you've typed against the line in the book. You can check the entire program again if you get an error when you run the program.

How to Type In Programs

To make it easy to know exactly what to type when entering one of these programs into your computer, we have established the following listing conventions.

Generally, Commodore 64 program listings will contain words within braces which spell out any special characters: {DOWN} would mean to press the cursor down key. {5 SPACES} would mean to press the space bar five times.

To indicate that a key should be *shifted* (hold down the SHIFT key while pressing the other key), the key would be underlined in our listings. For example, S would mean to type the S key while holding the SHIFT key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (e.g., {10 N}), you should type the key as many times as indicated (in our example, you would enter ten shifted N's).

If a key is enclosed in special brackets, [\leftarrow], you should hold down the *Commodore key* while pressing the key inside the special brackets. (The Commodore key is the key in the lower left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as necessary.

Rarely, you'll see a solitary letter of the alphabet enclosed in braces. These characters can be entered by holding down the CTRL key while typing the letter in the braces. For example, {A} would indicate that you should press CTRL-A.

About the *quote mode*: You know that you can move the cursor around the screen with the CRSR keys. Sometimes a programmer will want to move the cursor under program control. That's why you see all the {LEFT}'s, {HOME}'s, and {BLU}'s in our programs. The only way the computer can tell the difference between direct and programmed cursor control is the quote mode.





































Once you press the quote (the double quote, SHIFT-2), you are in the quote mode. If you type something and then try to change it by moving the cursor left, you'll only get a bunch of reverse-video lines. These are the symbols for cursor left. The only editing key that isn't programmable is the DEL key; you can still use DEL to back up and edit the line. Once you

B
Appendix

type another quote, you are out of quote mode.

You also go into quote mode when you INSerT spaces into a line. In any case, the easiest way to get out of quote mode is to just press RETURN. You'll then be out of quote mode, and you can cursor up to the mistyped line and fix it.

Use the following table when entering cursor and color control keys:

When You Read:	Press:	See:	When You Read:	Press:	See:
{CLR}	SHIFT CLR/HOME		{E1}	COMMODORE 1	
{HOME}	CLR/HOME		{E2}	COMMODORE 2	
{UP}	SHIFT 		{E3}	COMMODORE 3	
{DOWN}			{E4}	COMMODORE 4	
{LEFT}	SHIFT 		{E5}	COMMODORE 5	
{RIGHT}			{E6}	COMMODORE 6	
{RVS}	CTRL 9		{E7}	COMMODORE 7	
{OFF}	CTRL 0		{E8}	COMMODORE 8	
{BLK}	CTRL 1		{F1}	f1	
{WHT}	CTRL 2		{F2}	SHIFT f1	
{RED}	CTRL 3		{F3}	f3	
{CYN}	CTRL 4		{F4}	SHIFT f3	
{PUR}	CTRL 5		{F5}	f5	
{GRN}	CTRL 6		{F6}	SHIFT f5	
{BLU}	CTRL 7		{F7}	f7	
{YEL}	CTRL 8		{F8}	SHIFT f7	

The Automatic Proofreader

Charles Brannon

“The Automatic Proofreader” will help you type in program listings without typing mistakes. It is a short error-checking program that hides itself in memory. When activated, it lets you know immediately after typing a line from a program listing if you have made a mistake. Please read these instructions carefully before typing any programs in this book.

Preparing the Proofreader

1. Using the listing below, type in the Proofreader. Be very careful when entering the DATA statements—don't type an I instead of a 1, an O instead of a 0, extra commas, etc.
2. SAVE the Proofreader on tape or disk at least twice *before running it for the first time*. This is very important because the Proofreader erases part of itself when you first type RUN.
3. After the Proofreader is saved, type RUN. It will check itself for typing errors in the DATA statements and warn you if there's a mistake. Correct any errors and SAVE the corrected version. Keep a copy in a safe place—you'll need it again and again, every time you enter a program from this book, *COMPUTE!'s Gazette*, or *COMPUTE!* magazine.
4. When a correct version of the Proofreader is run, it activates itself. You are now ready to enter a program listing. If you press RUN/STOP-RESTORE, the Proofreader is disabled. To reactivate it, just type the command SYS 886 and press RETURN.

Using the Proofreader

All listings in this book have a *checksum number* appended to the end of each line—for example, :rem 123. *Don't enter this statement when typing in a program*. It is just for your information. The rem makes the number harmless if someone does type it in. It will, however, use up memory if you enter it, and it will confuse the Proofreader, even if you entered the rest of the line correctly.

When you type in a line from a program listing and press

RETURN, the Proofreader displays a number at the top of your screen. *This checksum number must match the checksum number in the printed listing.* If it doesn't, it means you typed the line differently than the way it is listed. Immediately recheck your typing. Remember, don't type the rem statement with the checksum number; it is published only so you can check it against the number which appears on your screen.

The Proofreader is not picky with spaces. It will not notice extra spaces or missing ones. This is for your convenience, since spacing is generally not important. But occasionally proper spacing *is* important, so be extra careful with spaces, since the Proofreader will catch practically everything else that can go wrong.

There's another thing to watch out for: If you enter the line by using abbreviations for commands, the checksum will not match up. But there is a way to make the Proofreader check it. After entering the line, LIST it. This eliminates the abbreviations. Then move the cursor up to the line and press RETURN. It should now match the checksum. You can check whole groups of lines this way.

Special Tape SAVE Instructions

When you're done typing a listing, you must disable the Proofreader before SAVEing the program on tape. Disable the Proofreader by pressing RUN/STOP-RESTORE (hold down the RUN/STOP key and sharply hit the RESTORE key). This procedure is not necessary for disk SAVES, *but you must disable the Proofreader this way before a tape SAVE.*

SAVE to tape erases the Proofreader from memory, so you'll have to load and run it again if you want to type another listing. SAVE to disk does not erase the Proofreader.

Hidden Perils

The Proofreader's home in the 64 is not a very safe haven. Since the cassette buffer is wiped out during tape operations, you need to disable the Proofreader with RUN/STOP-RESTORE before you save your program. This applies only to tape use. Disk users have nothing to worry about.

Not so for 64 owners with tape drives. What if you type in a program in several sittings? The next day, you come to your computer, load and run the Proofreader, then try to load the partially completed program so you can add to it. But

since the Proofreader is trying to hide in the cassette buffer, it is wiped out!

What you need is a way to load the Proofreader after you've loaded the partial program. The problem is, a tape LOAD to the buffer destroys what it's supposed to load.

After you've typed in and run the Proofreader, enter the following lines in direct mode (without a line number) exactly as shown:

```
A$="PROOFREADER.T": B$="{10 SPACES}": FOR X=1  
TO 4: A$=A$+B$: NEXTX  
FOR X=886 TO 1018: A$=A$+CHR$(PEEK(X)): NEXTX  
OPEN 1,1,1,A$:CLOSE1
```

After you enter the last line, you will be asked to PRESS RECORD & PLAY on your cassette recorder. Put this program at the beginning of a new tape. This gives you a new way to load the Proofreader. Anytime you want to bring the Proofreader into memory without disturbing anything else, put the cassette in the tape drive, rewind, and enter:

```
OPEN1:CLOSE1
```

You can now start the Proofreader by typing SYS 886. To test this, PRINT PEEK (886) should return the number 173. If it does not, repeat the steps above, making sure that A\$ ("PROOFREADER.T") contains 13 characters and that B\$ contains 10 spaces.

You can now reload the Proofreader into memory whenever LOAD or SAVE destroys it, restoring your personal typing helper.

Incidentally, you can protect the cassette buffer on the Commodore 64 with POKE 178,165. With this POKE, the 64 will not wipe out the cassette buffer during tape LOADs and SAVEs.

Replace Original Proofreader

If you typed in the original version of the Proofreader from the October 1983 issue of *COMPUTE!'s Gazette*, you should replace it with the improved version below.

Automatic Proofreader

```
100 PRINT "{CLR}PLEASE WAIT...":FORI=886TO1018:READ  
A:CK=CK+A:POKEI,A:NEXT
```

C
Appendix

```
110 IF CK<>17539 THEN PRINT"{DOWN}YOU MADE AN ERRO  
R":PRINT"IN DATA STATEMENTS.":END  
120 SYS886:PRINT"{CLR}{2 DOWN}PROOFREADER ACTIVATE  
D. ":NEW  
886 DATA 173,036,003,201,150,208  
892 DATA 001,096,141,151,003,173  
898 DATA 037,003,141,152,003,169  
904 DATA 150,141,036,003,169,003  
910 DATA 141,037,003,169,000,133  
916 DATA 254,096,032,087,241,133  
922 DATA 251,134,252,132,253,008  
928 DATA 201,013,240,017,201,032  
934 DATA 240,005,024,101,254,133  
940 DATA 254,165,251,166,252,164  
946 DATA 253,040,096,169,013,032  
952 DATA 210,255,165,214,141,251  
958 DATA 003,206,251,003,169,000  
964 DATA 133,216,169,019,032,210  
970 DATA 255,169,018,032,210,255  
976 DATA 169,058,032,210,255,166  
982 DATA 254,169,000,133,254,172  
988 DATA 151,003,192,087,208,006  
994 DATA 032,205,189,076,235,003  
1000 DATA 032,205,221,169,032,032  
1006 DATA 210,255,032,210,255,173  
1012 DATA 251,003,133,214,076,173  
1018 DATA 003
```


Commodore 64 Sound Memory Map

This appendix is a memory map of the Commodore 64's SID (Sound Interface Device) chip. Like any other map, it shows you where things are. This memory map includes descriptions of all the sound functions explained in this book and shows you their locations.

This is intended as a quick reference guide. If you need additional information on any of the functions, refer to the description of that feature in the text of the book.

Memory Location	Function Description	Bit(s)
54272	Voice 1 Pitch Lower Value	0-7
54273	Voice 1 Pitch Upper Value	0-7
54274	Voice 1 Pulse Width Lower Value	0-7
54275	Voice 1 Pulse Width Upper Value	0-3
54276	Voice 1 Waveform Control Register	0-7
54276	Voice 1: Gate Bit Initiates start of all voice 1 sounds	0
54276	Voice 1: Sync Function control bit	1
54276	Voice 1: Ring Modulation control bit	2
54276	Voice 1: Test Bit	3
54276	Voice 1: Triangle Wave control bit	4
54276	Voice 1: Sawtooth Wave control bit	5
54276	Voice 1: Pulse Wave control bit	6
54276	Voice 1: Noise control bit	7
54277	Voice 1 Decay Control Register	0-3

D
Appendix

Memory Location	Function Description	Bit(s)
54277	Voice 1 Attack Control Register	4-7
54278	Voice 1 Release Control Register	0-3
54278	Voice 1 Sustain Control Register	4-7
54279	Voice 2 Pitch Lower Value	0-7
54280	Voice 2 Pitch Upper Value	0-7
54281	Voice 2 Pulse Width Lower Value	0-7
54282	Voice 2 Pulse Width Upper Value	0-3
54283	Voice 2 Waveform Control Register	0-7
54283	Voice 2: Gate Bit Initiates start of all voice 2 sounds	0
54283	Voice 2: Sync Function control bit	1
54283	Voice 2: Ring Modulation control bit	2
54283	Voice 2: Test Bit	3
54283	Voice 2: Triangle Wave control bit	4
54283	Voice 2: Sawtooth Wave control bit	5
54283	Voice 2: Pulse Wave control bit	6
54283	Voice 2: Noise control bit	7
54284	Voice 2 Decay Control Register	0-3
54284	Voice 2 Attack Control Register	4-7
54285	Voice 2 Release Control Register	0-3
54285	Voice 2 Sustain Control Register	4-7
54286	Voice 3 Pitch Lower Value	0-7
54287	Voice 3 Pitch Upper Value	0-7

D
Appendix

Memory Location	Function Description	Bit(s)
54288	Voice 3 Pulse Width Lower Value	0-7
54289	Voice 3 Pulse Width Upper Value	0-3
54290	Voice 3 Waveform Control Register	0-7
54290	Voice 3: Gate Bit Initiates start of all voice 3 sounds	0
54290	Voice 3: Sync Function control bit	1
54290	Voice 3: Ring Modulation control bit	2
54290	Voice 3: Test Bit	3
54290	Voice 3: Triangle Wave control bit	4
54290	Voice 3: Sawtooth Wave control bit	5
54290	Voice 3: Pulse Wave control bit	6
54290	Voice 3: Noise control bit	7
54291	Voice 3 Decay Control Register	0-3
54291	Voice 3 Attack Control Register	4-7
54292	Voice 3 Release Control Register	0-3
54292	Voice 3 Sustain Control Register	4-7
54293	Cutoff Point: Lower Value	0-2
54294	Cutoff Point: Upper Value	0-7
54295	Filter Control Register	0-3
54295	Voice 1: Filter control bit	0
54295	Voice 2: Filter control bit	1
54295	Voice 3: Filter control bit	2
54295	External Filter control bit	3

D
Appendix

Memory Location	Function Description	Bit(s)
54295	Resonance Control Register	4-7
54296	Volume Control Register	0-3
54296	Low Pass Filter control bit	4
54296	Band Pass Filter control bit	5
54296	High Pass Filter control bit	6
54296	Voice 3 Output disable bit	7

Index

- "Accordion" program 119-20
- additive synthesis 147-48
- ADSR envelope i, vii, 10-12, 134-46
 - changing 24-27
 - POKE values 13
 - registers 9, 135
- asymmetrical ii
- attack i, 11, 25, 135-37
 - times 137
- attack/decay control register 12-13
- "Attack Rate" program 136
- "Automatic Proofreader" program xvii-xviii, 209-12
- background music 179-81
- background sounds 173-75
- band pass filter iv, 152
- base line iii-iv
- beats v
- "Beats" program 66-67
- bit 6-8
- "Bouncing Ball" program 163-64
- byte 7-8
- "Car Horn" program 65-66
- "Changing Pulse Widths" program 132-33
- "Chirps" program 110-11
- "Chord Editor" program 69-79
 - instructions 79-82
- "Chord Editor READER" program 88-89
- "Click" program 89
- "Clinking with Attack/Decay" program 123
- "Clinking" program 122-23
- Commodore 64 Programmer's Reference Guide* 173
- "Comparing Sounds" program 98
- compression iii
- COMPUTE!'s Reference Guide to Commodore 64 Graphics* 173
- "Creaky Door" programs 92-94
 - pitch changes and 93
- "Crickets" program 95-96
- "Crunching Snow" program 124-25
- cutoff point (filters) vi, 153-55
- cycle v, 24
- DATA statement 21, 206
- decay i, vi, 11, 25, 138-39
 - times 138
- "Dial Phone" program 105-6
- discord vii
- disharmony 63-65
- "Disk File Merger" program 181-88
 - instructions 182-88
- "Down and Up Sounds" program 86-87
- eight-bit computer 6
- "Electronic Ringer" program 98-99
- envelope (*see* ADSR envelope)
- "Envelope Manipulation" program 140-41
- experimentation, value of 202
- "Fast Attack - Slow Decay" program 26
 - modifying 26-27
- filters vii, 152-58, 161
 - cutoff point 153-55
 - selection 153
 - using 155
- flat viii
- "Flying Saucer" program 101-2
- "Foghorn" program 123-24
- FOR-NEXT loop 23
- "Frequency Calculation" program 131
- frequency vii, 24
 - sound registers for 9, 13
- function keys 167-68
- gate bit viii, 18-19, 68, 88, 137, 149
 - disabling 10, 19
 - enabling 9-10
- hard sounds viii, 25, 85-106
 - defined 85
 - noise waveform and 102-5
 - pulse waves and 89-97
 - sawtooth waves and 97-101
 - triangle waves and 101-2
- "Harmonious Sounds" program 63-65
- harmony viii, 63-65
- harpsichord 121-22
- "Heartbeats" program 114-15
- "Helicopter Sound and Animation" program 175-77
- hertz v, viii-ix
- high pass filter ix, 152
- "Horse Galloping" program 103-4
- Hz (*see* hertz)
- "Lawn Mower" program 173-75
- "Longer Beep" program 86
- lower pitch value x
- low pass filter ix-x, 152
- "Low Pass Filtering" program 155-56
- memory map, sound 213-16
- "Modifying Upper/Lower Tones" program 20-21
- "Motor Boat" program 96-97
- "Moving Barriers" program 166-68
- "Multivolume Sounds" program 145
- "Music Box" program 169-71
- music files 181-91

- musical instruments 39
- noise x, 32–33, 102–5
- nonsynchronized music 179
- “Nonsynchronized Sound and Graphics” program 179–81
- “Note-by-Note Volume” program 23
- nybble 12
- octave x–xi
- “Ocean Waves” program 116–17
- “Old Car” program 99–101
- “One-Voice Sound Editor” program 48–59
- “Organ Chords” program 67–68
- pausing 42–46
- peak xi
- PEEK command 6
 - music data and 191–95
- “PEEKing Values—One-Voice Sound Editor” program 193–94
- “PEEKing Values—Chord Editor” program 194–95
- “Pencil sharpener” program 103
- “Piano Player” program 120–21
- “Ping-Pong” program 91–92
- pitch control registers 14, 40, 130–31
 - table 14–16
- “Pitches and Pauses in ‘Yankee Doodle’” program 45–46
- POKE command 6
 - music data and 191–95
- “POKEing and PEEKing for Sound” program 164–65
- “POKEing Music” program 192–93
- pulse wave viii, xi, 88, 97
- pulse width xi–xii, 18
- pulse width control registers 131–32
- “Pulse Width Values” program 135–36
- READ and DATA statement 21
 - rhythm and pitch and 45–46
- “READING Routine” program 189–90
- release i, xii, 12, 140, 146
- REM statement xvii–xviii
- “Repeater” program 87
- resonance xiii, 157–58
- rhythm 41–42, 45–46
- ring modulation xiii, 128, 148–50, 161
- “Ring Modulation—Sweeping Both Voices” program 150
- “Ring Modulation—Sweeping Voice 1” program 149
- “Ring Modulation—Sweeping Voice 3” program 148–49
- “Rising Tapered Sound” program 118–19
- “Sawing Wood” program 115–16
- sawtooth wave xiii, 30–32, 88, 97
- “Sawtooth Waveform” program 32
- sharp xiv
- “Short Beep” program 86
- SID chip xiv, 3–35, 40, 85, 129–31, 156, 161, 173
- “Simple Sound” program 19–20
- “Siren” program 66
- “Snare Drums” program 43–44
- soft sounds xiv, 85, 106–25
 - noise waveform and 115–16
 - pulse waveform and 110–12
 - sawtooth waveform and 112–14
 - triangle waveform and 114–18
- “Soft Sounds—Attack, Decay, Sustain and Release” program 107–9
- “Soft Sounds—Volume Control” program 109–11
- sound control registers xiv, 4–5
 - clearing 8–9
 - initializing 129–30
 - table 4–5
- sound effects 85–125
 - software and 161
- sound envelope (*see* ADSR envelope)
- “Sound Game” program 195–201
- Sound Interface Device (*see* SID)
- sounds
 - altering 175–77
 - continuous 173
 - variably in programs 162–73
- sound wave, pure 24
- sprite collision registers 171, 172
- “Sprite Collisions and Sound” program 171–73
- sprites 171–73
- square wave xv, 27–28
- “Squeaky Shoes” program 111–12
- subtractive synthesis 152–58
- sustain i, xiv–xv, 11–12, 139–40, 145
- sustain loop 18–19
- “Sweeping the Scales” program 33–34
- “Switching Waveforms” program 87–88
- symmetrical wave (*see* square wave)
- “Symmetrical-Asymmetrical Waveforms” program 28–29
- synchronization xv, 127, 150–52, 161
- “Synchronization” program 151
- system clock 179, 181
- tapered sounds xv, 116
 - noise waveform and 124–25
 - pulse waveform and 119–22
 - sawtooth waveform and 123–24
 - slowly falling 85
 - slowly rising 85
 - triangle waveform and 122–23
- test bit xv, 146
- “Test Bit On/Off” program 144

Index

- "Ticktock" program 90-91
- "Timing the Drums" program 44-45
- "Title Screen" program 177-78
- triangle wave viii, xvi, 29-30, 31, 88, 101
- "Triangle Waveform" program 30
- typing in programs xvii-xviii, 205-8
- upper pitch value xvi
- VIC-II chip 3
- "Violins" program 112-14
- volume 10, 24
 - adjusting 21-23
 - sound control register of 9
- "Volume Variables" program 22
- waveform xvi, 27-34
 - mixing waveforms 105-6
 - selecting 17-18
- waveform control bit viii
 - enabling 9-10
- waveform control registers 134, 146
- white noise (*see* noise)
- "Yankee Doodle"
 - data chart 47
 - note values and timing counts chart 48
- "Yankee Doodle" program 40-41
- "Yankee Doodle" program with pauses 42-43



Notes

Notes



Notes

Notes



If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!** Magazine. Use this form to order your subscription to **COMPUTE!**.

For Fastest Service,
Call Our **Toll-Free** US Order Line
800-334-0868
In NC call **919-275-9809**

COMPUTE!

P.O. Box 5406
Greensboro, NC 27403

My Computer Is:

- Commodore 64 TI-99/4A Timex/Sinclair VIC-20 PET
 Radio Shack Color Computer Apple Atari Other _____
 Don't yet have one...

- \$24 One Year US Subscription
 \$45 Two Year US Subscription
 \$65 Three Year US Subscription

Subscription rates outside the US:

- \$30 Canada
 \$42 Europe, Australia, New Zealand/Air Delivery
 \$52 Middle East, North Africa, Central America/Air Mail
 \$72 Elsewhere/Air Mail
 \$30 International Surface Mail (lengthy, unreliable delivery)

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

Payment Enclosed

VISA

MasterCard

American Express

Acc't. No. _____

Expires _____ / _____



If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!'s Gazette** for Commodore.

For Fastest Service
Call Our **Toll-Free** US Order Line
800-334-0868
In NC call **919-275-9809**

COMPUTE!'s GAZETTE

P.O. Box 5406
Greensboro, NC 27403

My computer is:

Commodore 64 VIC-20 Other _____
01 02 03

- \$20 One Year US Subscription
- \$36 Two Year US Subscription
- \$54 Three Year US Subscription

Subscription rates outside the US:

- \$25 Canada
- \$45 Air Mail Delivery
- \$25 International Surface Mail

Name _____
Address _____
City _____ State _____ Zip _____
Country _____

Payment must be in US Funds drawn on a US Bank, International Money Order, or charge card. Your subscription will begin with the next available issue. Please allow 4-6 weeks for delivery of first issue. Subscription prices subject to change at any time.

- Payment Enclosed VISA
- MasterCard American Express

Acct. No. _____ Expires _____ / _____

The *COMPUTE!'s Gazette* subscriber list is made available to carefully screened organizations with a product or service which may be of interest to our readers. If you prefer not to receive such mailings, please check this box .



COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**

800-334-0868
In NC call 919-275-9809

Quantity	Title	Price	Total
_____	Machine Language for Beginners	\$14.95*	_____
_____	Home Energy Applications	\$14.95*	_____
_____	COMPUTE!'s First Book of VIC	\$12.95*	_____
_____	COMPUTE!'s Second Book of VIC	\$12.95*	_____
_____	COMPUTE!'s First Book of VIC Games	\$12.95*	_____
_____	COMPUTE!'s First Book of 64	\$12.95*	_____
_____	COMPUTE!'s First Book of Atari	\$12.95*	_____
_____	COMPUTE!'s Second Book of Atari	\$12.95*	_____
_____	COMPUTE!'s First Book of Atari Graphics	\$12.95*	_____
_____	COMPUTE!'s First Book of Atari Games	\$12.95*	_____
_____	Mapping The Atari	\$14.95*	_____
_____	Inside Atari DOS	\$19.95*	_____
_____	The Atari BASIC Sourcebook	\$12.95*	_____
_____	Programmer's Reference Guide for TI-99/4A	\$14.95*	_____
_____	COMPUTE!'s First Book of TI Games	\$12.95*	_____
_____	Every Kid's First Book of Robots and Computers	\$ 4.95†	_____
_____	The Beginner's Guide to Buying A Personal Computer	\$ 3.95†	_____

* Add \$2 shipping and handling. Outside US add \$5 air mail; \$2 surface mail.

† Add \$1 shipping and handling. Outside US add \$5 air mail; \$2 surface mail.

Please add shipping and handling for each book ordered. _____

Total enclosed or to be charged. _____

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

Payment enclosed Please charge my: VISA MasterCard
 American Express Acc't. No. _____ Expires / _____

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Allow 4-5 weeks for delivery.





A Thousand Sounds

One of the most powerful features of the Commodore 64 computer is its ability to create sound. Its SID chip, perhaps the most advanced sound creation device in any personal computer, can create natural and artificial sound effects, as well as "sing" in three-part harmony. But creating and controlling sound on the 64 can be confusing, especially if you're just starting out.

COMPUTE!'s Beginner's Guide to Commodore 64 Sound shows you how to use the impressive sound and music features of your computer. Whether you're an experienced programmer or a beginner, you'll find that the clear and detailed explanations make it simple to produce sound and music on the 64. To help you compose your own music, we've even included complete single-voice and chord editor programs that will turn your computer into a keyboard.

- Learn how to change the sound envelope to produce just the sound or effect you want.
- Dozens of example sound effects that you can type in and run, or use in your own programs and games.
- Change waveforms and create sound, from sirens to sawing wood.
- Use ring modulation, synchronization, and resonance to produce unusual artificial sounds.
- All terms are explained in an easy-to-follow glossary.
- Figures, charts, and tables are included for quick reference.

John Heilborn, author of the best-selling *COMPUTE!'s Reference Guide to Commodore 64 Graphics*, returns with clear writing and complete explanations to help you understand and master sound programming. If you want to program sound for musical compositions, arcade-style games, or just for fun, everything you need to get you started is right here.