# Contents

# Chapter 1
# You and Your TV

Virtually every home in the country has at least one television set, and televisions, like many other domestic appliances, tend to 'be taken for granted. Most people do not have any idea of how they work. Since you are interested in having the television display your ideas and visions in the form of a video game, you should have some idea of how an image is generated on the television screen to best utilize the capabilities of your computer.

## HOW A COMPUTER DISPLAYS A PICTURE

The face of a video display is coated with special phosphors. An electron beam strikes the face of the tube causing the phosphors to glow. On a black and white monitor, this process produces a dot. This glowing dot is called a *pixel*. A pixel is the smallest area that a computer can control on the screen. A color monitor uses three electron beams and three different colored phosphors to create one pixel.

The electron beam scans from left to right across the face of the monitor. By controlling the intensity of the electron beam during its scan, different points on the screen receive different intensities.

When the beam gets to the right edge of the screen, a horizontal sync pulse causes the beam to cut its intensity and return to the left side of the screen on the next line down. This process is repeated 262-1/2 times to form one display screen. At the end of the screen, a vertical sync pulse is initiated. During the vertical sync pulse, the beam returns to the upper left side of the screen and the whole process is ready to start again. On the Commodore 64, the computer's hardware automatically inserts both the horizontal and vertical sync pulses, so the programmer need not worry about generating them.

When a Commodore 64 is driving the monitor, 200 scan lines are used to display text and graphics with the other 62 lines being used for the border. The display process is repeated 60 times per second, providing a flicker free display.

There is a time correlation between the speed of the microprocessor and the position of the beam on the face of the screen. Some special effects can be created by using this fact and changing the display parameters on the fly. In the time it takes for the microprocessor to go through one machine cycle, the beam travels approximately 6 pixels on the display screen.

## WHAT IS ANIMATION

As you were reading the description of the generation of a TV *frame,* you may have noticed that the only thing that a monitor can display is a series of still frames. Thus the question of how you can get animation out of still pictures arises.

There is a characteristic of the human eye called the *flicker fusion frequency,* which allows us to view TV shows and movies without seeing that they are made up of still frames. This frequency is 24 frames a second. Any time a series of pictures is shown at a rate faster than 24 *hertz,* the eye can no longer distinguish the individual pictures. If the computer makes small changes in its display faster than 24 times per second, these changes will give the appearance of being continuous.

It is important to understand that the programs you will be writing create a series of still frames, not continuous motion. Because the computer updates the screen 60 times per second, this is the fastest that any changes can occur on the screen. If an object is in motion at the speed of one pixel per screen change, it takes about 5.3 seconds for the object to get from one side of the screen to the other. If the object needs to go faster, it has to move more than one pixel per screen update. On the other hand, if the object is to go slower, it has to stay still during some screen updates.

Because the video monitor provides a known minimum update time of 1/60 second, this tends to become the time period by which most aspects of the game are measured. This period of time (1/60 second) will be called a *screen.*

From what you have read, you might assume that completely different displays could alternate 30 times per second, and the eye would fuse them. This is true; *multiplexing* is the term used to refer to this technique. Some care must be taken when you attempt to use this technique. The differences between the two screens should not be excessive, and best results are achieved using small fast moving objects.

# Chapter 2
# A Language for Games

The first question that arises when you are starting work on a computer project is, "What language should I program in?" Your first inclination might be to use BASIC. However, if your program is using any type of continuous motion, BASIC would most likely be too slow to be useful. If you try to write anything more than the most simplistic game in BASIC, you will find the motion of the objects slow and erratic. There are limits to how quickly sounds and colors can be changed.

All of these problems arise because the BASIC interpreter controls the computer, not the program. Every line in a BASIC program must be analyzed, decoded, and translated into a series of machine language instructions EVERY time the line is encountered. BASIC is also a very general language in that similar lines may have different functions. By the time a line is analyzed, the resulting section of machine code is not efficient. For all of these reasons, a BASIC program will never run particularly fast.

To achieve smooth animation, the program must be able to update the screen at least 30 times per second. Because BASIC does not have an easy way to talk to the hardware registers, (PEEK and POKE commands must be used), calculating new positions and updating the registers can easily take longer than 1/30 second, causing erratic motion. This same problem occurs when the sound registers must be updated. There may not be enough time to make a consistent sound.

Machine language, on the other hand, is the most efficient form that a program may take. Each instruction in the program is executed exactly as it is entered. There is no interpretation done on the code, so it runs at the highest possible speed. You also know the status of the entire computer at every step—which rarely happens in a BASIC program.

The major drawback to programming in machine language is that it is a collection of hexadecimal codes. This is fine for the computer, because the binary data that these codes represent can be immediately executed, but it is cumbersome for the programmer. Most programmers do not enjoy memorizing all of the hex codes, and even fewer enjoy reading such a program when it is finished.

## ASSEMBLY LANGUAGE

The alternative to these two extremes is to program in *assembly language.* Assembly language is a language that uses an assembler that translates the *mneumonics* (memory aids) for the CPU's instruction set into the binary data that the processor can execute. This translation takes place once before the program is run, so the final program will be machine code. You end up with all of the speed advantages of a machine language program with none of the headaches. A properly written assembly language program is just as efficient as its machine language counterpart, so there is no reason to program in machine language if you can gain access to an assembler.

**With an assembly language program, the programmer normally has more than enough time to do all of the calculations and updates to keep the animation constant and smooth. The** program can **also make use of all of the hardware features of the computer to create effects that are not possible through BASIC. When you are using assembly language, the computer is completely under software control, making it possible to determine what the computer is doing at all times. An assembly language program executes faster than any other type of program, which makes assembly language the language of choice for programming games.**

**Learning to program in assembly language is** not as traumatic an experience as most programmers would have you believe. Unlike BASIC, **the machine will always be doing exactly what you tell it to do. The main difference between BASIC and assembly language is that in assembly language you** must keep track of where the data is in memory and where it must go. BASIC keeps track of variables for you, but it won't tell you where they are. In assembly language, you can define various memory locations so that they have some meaning to **you.**

**For instance, you could define $20 to be the player horizontal position. ($ indicates a hexadecimal number; $20 is equal to 32 in the decimal system. See Chapter 3 for more information.) Whenever you** need to find out what the horizontal position is or need to modify it, you would look into location $20. You can create **your own variables in BASIC using the same technique. You would POKE the value into** $20 and PEEK it back out whenever you needed it. This is a particularly useful technique when you **mix BASIC** and assembly language, because each program **will** know where to find the data from the other program.

## USING AN ASSEMBLER

The rest of this book assumes that you will be programming in assembly language, and most of the examples are written in assembly language. If **you are a BASIC programmer, you may be able to use some of these routines to speed up parts of your programs.**

**Each assembler has** its **own set of** *pseudo opcodes—the* **instructions that tell the assembler to do something other than create code. In order to ensure that you** will be able to use these routines, the **following is a listing of the pseudo-opcodes used by Commodore's Macro Assembler Development System, which is the assembler used in this book. By** modifying these instructions to match those of your assembler, you should be able to **run any of the examples in this book. Normally, you can type in one of these instructions anywhere that it is legal to type** in one of the CPU's opcodes.

| | |
|---|---|
| .BYTE | Reserves one or more bytes of data starting at the current location counter value |
| .WORD | Reserves 16 bit data in a LOW byte-HIGH byte format |
| .DBYTE | Reserves 16 bit data in a HIGH byte-LOW byte format |
| | Program location counter |
| .LIB | Insert another disk file following this command |
| .END | End of file marker |
| | Assigns a value to a symbol |

| | |
|---|---|
| | Specifies the low order 8 bits of a 16 bit value |
| > | Specifies the high order 8 bits of a 16 bit value |
| .MAC | Starts a macro |
| .MND | Ends a macro |

All of the above may be preceded by a label.

| | |
|---|---|
| ? | Precedes a number that specifies which parameter to pass to the macro. It can also be used as a label. |

These are all of the commands that may be different from those in your assembler. With this list, you should be able to read all of the program listings and modify them to work with your assembler. Many assemblers come with a program that will translate files with this syntax into their own syntax. If your assembler has one of these programs, you will not have to make many changes by hand to assemble the listings on the distribution disk.

## WHAT AN ASSEMBLER CAN DO FOR YOU

An assembler relieves you from memorizing the actual machine codes for each of the instructions. It also calculates the distance from one instruction to another for those times when a branch must be taken from the main program. This is not a particularly big deal for a short program (under 50 lines), but as a program grows larger and more complex, the task of repeatedly doing these calculations by hand becomes unreasonable. (If you are the type of person who finds great enjoyment in hand coding machine language programs, let me apologize here for suggesting there is a better way. The rest of us will let the computer do the tedious jobs).

The most useful feature of an assembler is its ability to let you assign names. A name can be given to a memory location, hardware registers, or a location within the program. Once names have been assigned, it is no longer necessary to remember long lists of confusing addresses. You only need to remember the names you have assigned to the addresses. Since you will normally assign names that carry some meaning (at least to you) to the memory locations and program segments, the program becomes infinitely more readable than if the addresses themselves had been used.

But wait, did that last sentence use "readable" when referring to an assembly language program? Yes it did. Assembly language programs become illegible to others because it is rare to see a full listing of the program with all of its definitions; and, often a disassembly of a program is called the original program. (A disassembly has no legitimate names, just addresses.)

All good assemblers also have the ability to use *macro-instructions (macros).* A macro is a shorthand notation that represents a series of assembly language commands. For example, a macro that increments a two byte value by a one byte value could be coded as follows:

```
.MAC DBINC ;REGISTER NAME, DATA
LDA ?1         ;LOAD THE LOWER BYTE
CLC            ;CLEAR THE CARRY BIT
ADC #?2        ;ADD WITH CARRY THE DATA
STA ?1         ;STORE THE LOWER BYTE
LDA ?1+1       ;LOAD THE UPPER BYTE
ADC #$00 ;ADD THE CARRY BIT
STA ?1+1 ;STORE THE UPPER BYTE
.MND           ;END OF MACRO
```

This macro is used to increment any two consecutive bytes, such as a score. If the score needed to be incremented by $20, you would type:

```
DBINC SCORE,$20
```

which would be expanded by the assembler to read:

```
LDA SCORE
CLC
ADC #$20
STA SCORE
LDA SCORE + 1
ADC #$00
STA SCORE +1
```

This is certainly easier than typing the same series of instructions every time that you need to increment a two byte value. A program that uses macros will be easier to read as you work on it than one written using individual instructions. Once you have written and debugged a macro, it becomes a tool that can be quickly used whenever necessary. By building up a library of macros, you will be able to program quite difficult functions in a minimal amount of time.

There are many cases in which it is a better idea to use a subroutine instead of a macro. Each time a macro name is entered into a program, the assembler expands it into its individual instructions. This means that each time the macro is called, it is treated as if you have entered all of its instructions by hand, and uses the same amount of memory as if you had.

For a function that is repeatedly used and takes a large amount of code, it is better to use a subroutine. A subroutine is called using the JSR instruction and is only stored once in the program. If you write a routine to display text on the screen that many different parts of the program are going to be calling on often, it is best treated as a subroutine. You may build up a library of subroutines in the same manner in which you build up a library of macros.

Normally, in the course of working on a game you will run into a situation that requires you to write a specialized routine to perform a certain function. If you think that you will be able to use this function at a later time, you should incorporate it into either your subroutine or macro library. This way, you will quickly have the major routines that are common to most programs at your fingertips.

An assembler must also provide some means of defining data areas and data. Tables of data can be defined, given a name, and stored by the assembler. A good assembler allows you to define data in terms of mathematical expressions. It also allows data to be defined as one or two byte values. There is a further option on two byte values as to whether the high byte or the low byte will be stored first. For many programs, text must be stored for later printing. On some assemblers, you have the option for text to be stored with the high bit on or off. This can be useful for finding the end of a text string.

Finally, an assembler must allow you to enter assembly language commands. After all, that is the point of an assembler.

## HOW TO CHOOSE AN ASSEMBLER

If you are to successfully create your own machine language video games, you must become familiar with your primary design aid, the assembler program. Next to your computer, a good assembler program is the most essential tool for the creation of a machine language program.

There are three parts to a good assembler package:

- A text editor
- The assembler
- A machine language monitor

The text editor is the part of the assembler that you will spend the most time with. It allows you to enter, modify, and update your program. Some assemblers allow you to use a word processor to enter your program. Whatever method you choose, make sure that you are comfortable with the editing commands that are available on your text editor.

It is a good idea to try out an editor before buying it. Some assemblers come with editors that are very limiting in what they allow you to do. Limitations in the editor take from your programming time, so it pays to shop around for a good one.

Once the program has been entered into the editor, it must be assembled before it can be used. Some assembler packages force you to load the assembler at this point, while others already have it loaded. An assembler that has all of the programs you need loaded simultaneously is called a *coresident assembler.*

A coresident assembler can save you quite a bit of time if you like to write a small section of code and immediately try to assemble it to check for errors. If you have a coresident package, remember always to save your source code before attempting to run your program. You can lose all of your latest work if your new program locks up the computer, forcing you to turn it off. In fact, no matter what type of assembler you are using, you should save your source code often as you are writing it.

Another aspect to examine while you are choosing an assembler is the speed with which it can assemble your source code and generate the necessary files on disk. Unfortunately, you can't expect the assembler to work faster than the disk can move the data.

You also should be sure that any printouts generated by the assembler contain all of the information that you would like. The following are some of the items that differ between assemblers:

- Sorted symbol table with absolute addresses
- Macro expansion
- Data expansion
- Absolute addresses for all code
- Absolute addresses for RAM registers

Depending on how you approach debugging your program, these different functions will have different levels of importance to you.

A symbol table is generated by all assemblers at some point. **It is** a list of the names used in the program and the addresses that correspond to the names. A printout of the table can help you verify the assembler is working properly and gives you a quick guide to any location used by the program. For this reason it helps if the machine sorts the table alphabetically before printing. On the other hand, if the assembler only prints out the symbol names and its internal representation of the addresses (usually filled in later by the assembler), the printout is useless for most purposes.

**If** you are going to use macros in your program, it is useful to have an assembler that lets you specify whether or not it should expand the macro before it is printed. When a macro is expanded for print, the macro name is expanded for print, the macro name is printed followed by the code it generates with all of the substitutions shown. On the printout, all of the instructions of the expanded macro are generally preceded by a + symbol. Without a macro expansion on the printout, you must constantly refer back and forth between a listing of your macro library and the section of code where the macro was called. This can be quite time consuming and prone to error as you expand the macro by hand for debugging purposes. But once all of your macros have been debugged and you are familiar with them, you rarely need to see them expanded on your printouts.

Since most assemblers allow you to use expressions in data statements, you should be able to get a printout of the calculated data. With such a listing, you can verify that the assembler generated the expected data. Again, once you are familiar with the operation of your assembler and your data has been debugged you rarely need to see this part of the printout.

Beware of an assembler that won't tell you where it has put your code or data. Your assembler should have in its printout absolute addresses for every instruction, data statement, and hardware or RAM register that has been used. If your assembler does not provide this information, you will find your program extremely difficult to debug.

The output of most assemblers is an intermediate file that contains all of the information needed about your program. This file is usually one form of a *hex file*. Hex files store the information about the program in a hexadecimal format that can be easily transmitted or loaded into the computer. (Binary data is more difficult to transfer from one machine to another.) You will use a program called a *loader* to translate the hex file into a binary file and place the data in the proper place in the computer.

One potentially useful option on most loaders is the ability to relocate the loading address of a program. For example, if you write a program to be placed on a cartridge, it needs to run from a different place in memory than if it is to be stored in RAM. The ability to relocate a program allows you to test it in one location although it is intended to run at another.

The last piece of an assembly language development system is a monitor. This is a program that allows you to examine the computer's memory and change or move the contents. It also must allow you to load and save areas of memory from the disk drive. A good monitor has a small disassembler that allows you to view memory as assembly language commands.

A word of caution: it is not a good idea to get a monitor in a cartridge. A monitor on a cartridge resides in a permanently fixed place in memory. If your program needs to use this area, you can't use the monitor. In fact, if the monitor must reside in only one predefined place in the computer, it can be useless. Either the monitor should be relocatable, or you should be given two or more different versions of the monitor. If you have a version of the monitor that resides in high RAM and another version that stays in low RAM, usually you will be able to use one of them.

# Chapter 3
# Underlying Concepts

At this point, some of the essentials that you need to know in order to understand the rest of this book will be presented. The terms and names that will be used will be defined. The hardware of the Commodore 64, and in particular, a programming model of the 6510 microprocessor will be discussed. Also, the hexadecimal numbering system (base 16), which is used throughout this book, will be described. This numbering system makes the most sense when dealing with computers. Since an understanding of the hexadecimal numbering system and its relationship to bits and bytes will make the hardware descriptions easier to understand, it will be presented first.

## BITS AND BYTES

A *bit* is the smallest meaningful piece of information that can be stored. It can have only two values, 1 or 0. Other terms for these values are shown below:

| | |
|---|---|
| 1 | 0 |
| ON | OFF |
| SET | CLEAR |
| HIGH | LOW |

All of the signals inside of the computer can only be in one of these two possible states. So how does the computer perform so many functions if it only has two states to work with?

By grouping a collection of bits together, the group of bits together can have a value that is equal to 2 raised to the power of the number of bits in the group. If we grouped 4 bits together, the group could have 16 possible values according to the equation 2^N where N is the number of bits in the group:

```
2^N N=4
2^4
2*2*2*2=16
```

A grouping of 4 bits is called a *nibble*. Each of the four bits is given a value depending on its position in the group. Spreading the bits out horizontally, the bit on the right is called the *least significant bit (LSB).* The bit on the left is called the *most significant bit (MSB).* The location of each bit in the byte is given a value of 2^N, where N is the number of bits from the LSB that the location in question is. For instance the LSB is located on itself so its distance (N) would be O. Therefore, the LSB can have a value of 0 or 1 depending on the state of its bit. The next bit on the left would have a value of 2^l.

The group of four bits could have sixteen values, 0 through 15, as you have seen. As discussed earlier, the value of the bit depends on its location in the group and its state. To compute the value of the group of bits, you simply add together the value of each location in the group whose corresponding bit is ON. By summing the total of all of the positions in a group, the maximum value of a group can be determined. In the case of a nibble, the maximum value would be 15.

If 8 bits were grouped together, the equation would be:

```
2^N  N=8
2^8
2*2*2*2*2*2*2*2=256
```

So a group of 8 bits can have 256 different values. This grouping is referred to as a *byte.* Since 0 is the first of the possible values of a byte, the range of values is from 0 to 255. Inside the Commodore 64, all of the data is represented and transferred as bytes. This is what is meant when the C-64 is referred to as an *8 bit machine.* A byte is the standard unit of storage in the Commodore 64.

It was mentioned earlier that hexadecimal would be the standard notation to be used in this book. This is because one hexadecimal *(hex)* digit can represent 16 values or one nibble if it is representing a group of 4 bits. Thus it requires 2 hex digits to represent any 8 bit value or any byte. The correlation between the 4 bits of a nibble, its decimal value, and its hex value is shown in Table 3-1.

**Table 3-1. The Relationship Between the Binary, Hex, and Decimal Number Systems.**

| $2^3$ | $2^2$ | $2^1$ | $2^0$ | DEC | HEX |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 | 2 |
| 0 | 0 | 1 | 1 | 3 | 3 |
| 0 | 1 | 0 | 0 | 4 | 4 |
| 0 | 1 | 0 | 1 | 5 | 5 |
| 0 | 1 | 1 | 0 | 6 | 6 |
| 0 | 1 | 1 | 1 | 7 | 7 |
| 1 | 0 | 0 | 0 | 8 | 8 |
| 1 | 0 | 0 | 1 | 9 | 9 |
| 1 | 0 | 1 | 0 | 10 | A |
| 1 | 0 | 1 | 1 | 11 | B |
| 1 | 1 | 0 | 0 | 12 | C |
| 1 | 1 | 0 | 1 | 13 | D |
| 1 | 1 | 1 | 0 | 14 | E |
| 1 | 1 | 1 | 1 | 15 | F |

You may notice that until the tenth value, (the number 9) the same numbers are used in both hex and decimal numbering systems. In hex however, the next 6 numbers are represented by the first 6 letters in the alphabet.

From this point forward, all hex numbers will be preceded by a dollar sign ($) to differentiate a hex number from a decimal number. This is the standard notation used by virtually all assemblers that assemble code for the 6500 series of microprocessors. If there is no $ preceding a number, it is assumed to be a decimal number, unless it contains any of the letters A-F, in which case you can assume that a mistake has been made.

## THE HARDWARE

Like all computers, the Commodore 64 is made of a number of complex integrated circuit chips. You can conceptualize the internal workings of the computer as being broken down into 5 different sections:

Central processing unit
Memory
Video generation
Sound
Input and Output

In the Commodore 64, the central processing unit *(CPU)* is a 6510 microprocessor chip. It executes the same instruction set as a 6502 microprocessor as used in Apple and ATARI computers. It runs with a clock frequency of 1.0225 MHz. For all practical purposes, this can be considered to be a 1 MHz clock. The 6510 has an addressing range of 65536 bytes (64K).

There are two different types of memory in the Commodore 64. It has 64K of dynamic RAM, which can be banked into the address space of the other chips as necessary. There is also 2OK of ROM in the system. In this ROM are the BASIC programming language and the operating system of the Commodore 64. The operating system is responsible for reading the keyboard, updating the real-time clock, and transferring data in and out of the system, among other things. Since the CPU can only address 64K of memory, all of the RAM cannot be accessed simultaneously with all of the ROM. To overcome this problem, the technique of *bank switching* is used. For instance, if you are not using BASIC, there is no need for the BASIC ROM to be accessible. In this case, it can be replaced with RAM. The CPU cannot tell the difference, so it can be "tricked" into addressing more then 64K of memory.

Video generation is a task that is taken care of by a 6567 *Video Interface chip (VIC II).* All of the various graphic modes of the Commodore 64 are generated by this chip. In the process of generating the video signal, the VIC-II chip refreshes the dynamic ram chips used in the system. The VIC-II chip also generates the system clock from the 8.18 MHz dot clock.

Sound is generated by a 6591 *Sound Interface Device chip (SID).* This chip can generate 3 independent voices each in a frequency range of 0 to 4 kHz. This corresponds to a range of about 9 octaves. Each voice has an independent volume envelope and a choice of waveforms. The SID chip can also provide a number of filtering options for use with its own signals or an externally supplied signal.

Input and output functions are handles primarily by a pair of 6526 *Complex Interface Adapter* chips. Serial communication functions as well as the parallel port are maintained by these chips. They also handle input from the joysticks and the real time clock. These chips each provide a pair of independent 16-bit timers.

If you understand how these four devices work, you can make the computer do anything it is capable of. Your program will be primarily concerned with the VIC-II chip and the SID chip. The CPU is the chip that the program is written for, and it is directed to modify the registers in the other chips at the appropriate time for the intended function. Writing almost any type of program eventually comes down to controlling just a few chips. Once you control the major chips the rest of the program should be easy.

## 6510 ARCHITECTURE

In order to program in assembly language, you must understand the internal functions of the microprocessor. Figure 3-1 is a block diagram of the 6510. The value of the *program counter* is output on the *address* lines of the microprocessor whenever a data access is to be performed on the systems memory. In the Commodore 64, all of the hardware registers appear to be memory locations to the microprocessor, so accesses to hardware registers and memory appear identical.

The *accumulator* is the most important register in the computer. Almost all of the data that passes through the system goes through the accumulator. Every arithmetic function, other than incrementing and decrementing, is performed in the accumulator. Data can be read into the accumulator from memory, modified, and stored back into memory.

The *X* and *Y registers* are very similar. They move data in a manner similar to the accumulator. They can also be used as an index to an array of data. It should be noted that while these two registers are similar, their functions are not identical. Some instructions require the use of the X register while others use the Y register.

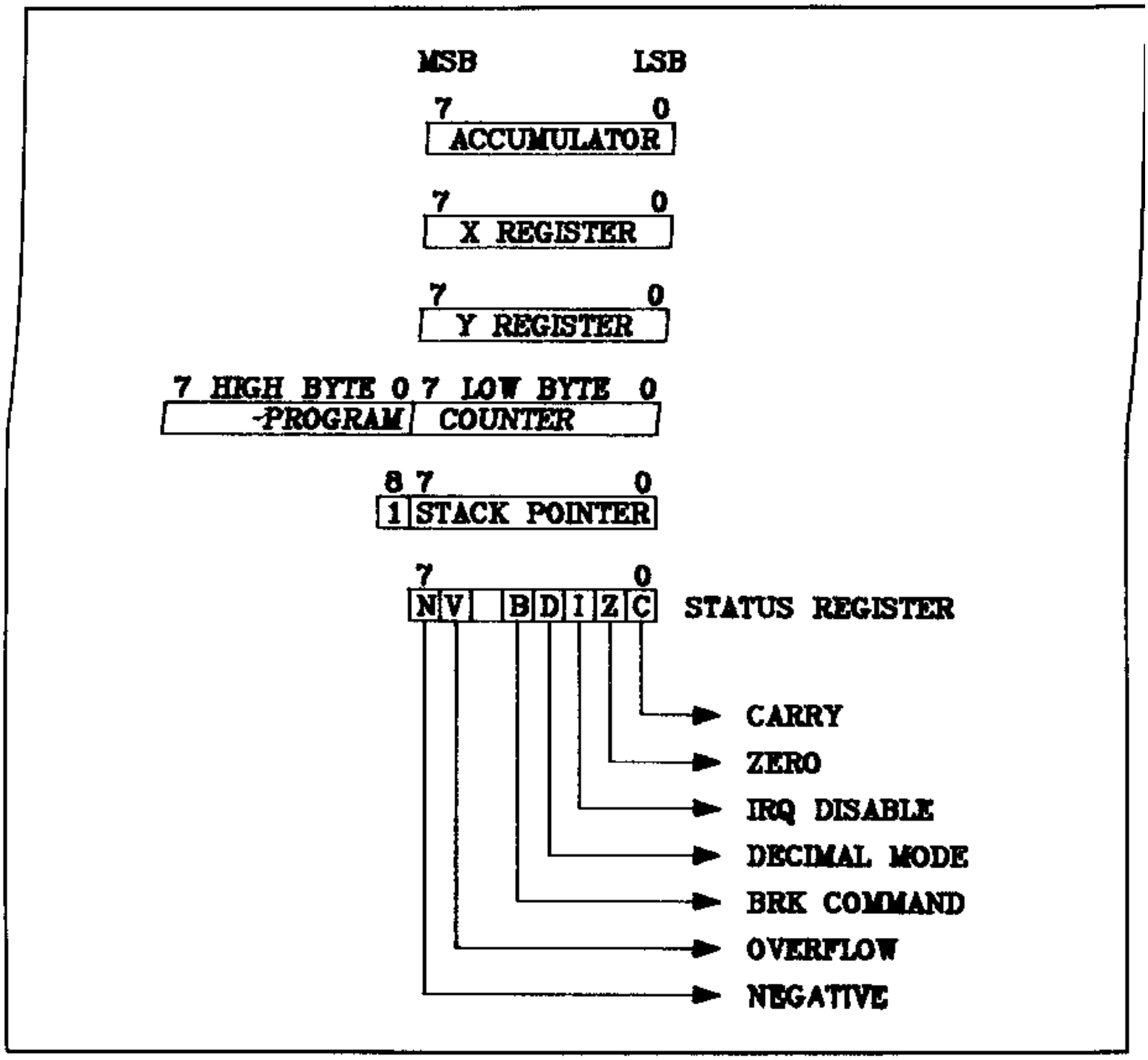


Fig. 3-1. Block diagram of the 6510 microprocessor.

Each of the bits in the *status* register correspond to one of the conditions in the microprocessor. Some of these bits can be changed under software control.

In the Commodore 64, the stack (a temporary data storage area between $0100 to $01FF) is controlled by the *stack pointer*. This register contains the address of the next empty space on the stack. At the start of a program the stack pointer is usually initialized to $FF, which corresponds to the top of the stack. The microprocessor defines the stack to start at $0100. The stack pointer is used as an index from the bottom of the stack.

With these concepts in mind, it is time to look at the language.

# Chapter 4
# The 6510 Assembly Language

In Appendix A are a series of charts that describe all of the instructions available in the 6510 as well as their addressing mode options. You may wish to refer to these charts as you are reading the following sections on instruction types and addressing modes. As you begin to program in assembly language, you will find yourself constantly referring to these charts.

All of the instruction lines used in assembly language take the following format:

```
LABEL OPCODE OPERAND ;comment
```

The *opcode* is the instruction that you want executed. The *operand* is the data, label, or memory address that will be operated on.

*Comments* are particularly useful in documenting your program and should be used often. A properly documented program is much easier to read and understand. A comment can either follow an instruction or be on a line by itself. A comment must be preceded by a semicolon.

*Labels* prevent assembly language from becoming unmanageable. A label must start in column 1 of the program line. A label can be assigned a value or can take on the value of the program counter during assembly. When used as the operand in a branch instruction, the assembler determines the length of a branch, which is much easier than calculating the branch distance by hand. Also, if a change is made in the program, the assembler can compensate for any changes in the branch. If the calculations are done by hand, they have to be redone every time there is a change.

## INSTRUCTION TYPES

There are 4 classes of instructions in the 6510. These are:

- Data movement
- Arithmetic
- Testing
- Flow of control

*Lata movement instructions* are instructions that cruse a value to be loaded from memory, stored into memory, or transferred from one register to another. There are a number of options as to how the address of the byte to be loaded will be determined. In the load accumulator instruction, LDA, there are eight different addressing modes that can be used to determine which byte to load. The different addressing modes are explained in the following section.

*Arithmetic instructions* are used to modify data in some way. This class of instruction includes logical operations, such as the AND and ORA instructions. There are instructions that allow a byte to be rotated as well as addition and subtraction commands. As with the data movement instructions, most of the available addressing modes can be used by the arithmetic instructions.

*fisting instructions* allow a nondestructive test of data in the microprocessor. For instance, when a CMP instruction is used to check a value in the ACCUMULATOR, the data in the ACCUMULATOR will not be changed in any way. The bits in the STATUS register will be changed in the same way as if the data to be compared was subtracted from the ACCUMULATOR. These instructions are generally used to modify the STATUS register prior to executing a branch instruction.

*Flow of control instructions* are the branching and jump instructions. These are used to change the order in which different sections of code are executed. The branch instructions are all *conditional branching* instructions. That is, each instruction checks one of the bits in the status register and, depending on its value, will either branch to the instruction pointed to in the operand or execute the next instruction in line.

Jump and jump to subroutine instructions also fall into the flow of control category. These are known as *absolute commands* because they do not check any conditions before performing a jump.

## ADDRESSING MODES

In the 6510 microprocessor, there are 11 types of addressing modes. They are:

| | |
|---|---|
| Immediate | (Indirect,X) |
| Zero page | (Indirect),Y |
| Zero page,X | Implied |
| Absolute | Relative |
| Absolute,X | Indirect |
| Absolute | |

Many of the addressing modes can be used by the LDA, or load accumulator, instruction. This instruction causes a byte to be loaded into the accumulator.

### Immediate Mode Addressing

In the immediate mode, the data that follows the # character will be loaded into the accumulator. Instead of entering data to be loaded, you could enter a label that has previously been equated to a value. For example, if you had defined the label BLACK to equal O, the following statements would be identical:

```
    LDA #$00
    LDA #BLACK
```

In either case, a 0 is loaded into the accumulator. In all of the other modes, data is loaded from a memory location as indicated by the operand.

### Zero Page Addressing

The *zero page of memory*, the memory locations in the range $00 to $FF, has a special meaning to the 6510. A location in this range can be accessed faster than anywhere else in memory. This results from the fact that the upper byte of the address will always be $00, so it need not be read from the operand during program execution. This also means that a program will be shorter because the upper byte of the address is not part of the code. Exclusive use of zero page memory can cut the program size and execution time by a third.

Zero page addressing modes take a one byte value as an operand to select the memory location. For example, if you want to load the contents of memory location $23 into the accumulator, you could enter the following:

```
    LDA $23
```

## Zero Page Indexed Addressing

Zero page indexed addressing uses the contents of the X register to determine the memory address to be accessed. Using the load accumulator instruction as an example, the memory address to be loaded is generated in the following way:

1. A memory address is specified in the instruction.
2. The value of the X register is added to this address.
3. The data from this generated address is loaded into the accumulator.

Zero page indexed addressing can only be used with the X register. Since the X register can contain an 8 bit value, an offset of up to $FF from the address specified in the instruction can be generated. Indexed addressing is used extensively when you are looking up a value in a table of data. A value corresponding to the distance into the lookup table would be loaded into the X register, then the indexed load instruction would be issued.

## Absolute Addressing

Absolute addressing uses a two byte value in the operand to generate a 16 bit memory address. Due to this, the 6510 can address any byte in the range of $0000 to $FFFF. Normally, the assembler will select the most efficient version of this instruction. If it is possible to use zero page addressing instead of absolute addressing, the assembler will generate this form of the instruction.

## Absolute indexed Addressing

Absolute Indexed addressing works in the same way as zero page indexed addressing except that a two byte address is specified in the operand. Also, the Y register can be used as the index register in absolute indexed addressing. Using a load accumulator instruction, the following steps are taken to load a byte:

1. A memory address is specified in the instruction.
2. The value of the index register is added to this address.
3. The data from this generated address is loaded into the accumulator.

## Indirect Addressing with indexes

So far, all of the addressing modes have assumed that you knew where the data you were interested in was ahead of time. Since this is not always the case, there needs to be a way for the computer to determine an address during program execution.

*Indirect addressing* means that you are not telling the microprocessor where the data that you want to use is, but rather you are telling the microprocessor where it can find the address of the data that you want. The indirect address will always be stored in zero page memory in two consecutive memory locations. The lower order byte of the address is stored in the first memory location, and the high order byte of the address is stored in the next location. This gives a 16 bit address, so that the data can be anywhere in the microprocessor's normal address space.

When you give an indirect addressing command, the operand will be the address in zero page that contains the first byte of the address of the pair of memory locations where the appropriate address is stored. It is important to reserve enough space in zero page RAM to hold all of the indirect addresses that you may be generating.

At this point, indirect addressing should seem pretty easy to use, but there is a catch. The 6510 does not have true indirect addressing abilities for data movement or arithmetic instructions. Instead there are two subclasses of indirect addressing available. These are:

- Indexed Indirect X
- Indirect Indexed Y

We will look at the second one first because it is the most often used. As is implied by the name indirect indexed Y, this command is a combination of indirect addressing and indexed addressing. As was mentioned earlier, the command will have the address of where the address may be found. After the microprocessor generates an address from the two zero page memory locations, the contents of the Y register is added to the address to form the final address. The data at this final address can then be accessed. If the value of the Y register is O, the command will act like a true indirect addressing command. You must be sure that the Y register is set to the desired value before executing this command, or you will never be sure of where the data is coming from (or going to).

In indexed indirect X addressing, the value of the X register is added to the zero page address of where the indirect address can be found. This new zero page address is then used to generate the final address of where the data can be located. If the X register is set to zero, this command will act like a normal indirect addressing command. The normal use for this command is to use the X register as an index into a table of addresses located in zero page RAM.

## Implied Addressing

Instructions that use implied addressing are only one byte long. Instead of having to give an address in the instruction, the microprocessor decides on which one of its internal registers to use based on the instruction. For instance, the TAX instruction will transfer the contents of the accumulator to the X register. This is known to the microprocessor without the need for any other addresses. Because the microprocessor doesn't need to calculate or load an address when using implied addressing, these instructions execute faster than any other type of instruction.

## Relative Addressing

All of the branching instructions in the 6510 use the relative addressing mode. In this mode, instead of specifying an address for the destination of the branch, an offset from the current instruction to the destination of the branch it specifies in the operand. The offset is a one byte value. This gives a branch instruction the ability to branch over a range of +127 bytes to —127 bytes. Normally, you will use a label as the destination of the branch when writing your program, and the assembler will calculate the offset.

## Indirect Addressing

There is only one instruction that uses indirect addressing in the 6510; it is an indirect jump. An indirect jump uses the principles of indirect addressing that were discussed earlier. The main difference between the indirect jump and the other indirect instructions is that a 2 byte value (16 bit) can be given as the address where the indirect data can be found. When using an indirect jump, the X and Y registers play no part in the address generation procedure.

# Chapter 5
# Organizing Your Program

Now that you have some understanding of what an assembler does and of the 6510 assembly language, you can begin to think about how to organize your program. All programs can be broken down as follows:

- Macro library
- System definitions
- RAM definitions
- Data definitions
- Main program
- Subroutines

Although it may not be obvious at this point, this is a very logical outline. The macro library must be assembled first. Quite often there will be macros that define data areas. A macro must be defined prior to its first use. There is no penalty in terms of storage space for a macro that is not used. The source code for a complete macro library is given in the file MACLIB, Listing C-1 in Appendix C. All of the macros in the file are described in detail in Appendix B. As they are given in the file, the macros will work with the Commodore Macro Assembler program. If you are using another assembler, you may have to modify them slightly before they can be used.

Once the macros are in the system, the machine's hardware registers should be defined. This is the starting point when you try to learn how a new computer works, as it forces you to become familiar with the hardware. A full set of system definitions can be found in the file SYSDEF, Listing C-2 in Appendix C. The names that are assigned to the various hardware registers by this file are used throughout the book, so it would be a good idea to refer to the listing at some point. Most of the registers will be described in detail in a later chapter.

The RAM that is to be used as variables for the program needs to be defined next. You will usually find that programming is easier if you define your variables before you start to write your program. These definitions do not have to be completed during the first sitting. As you progress in your program, you will find that you have not defined all of the RAM that you would like to use. Additions to the RAM definitions tend to continue until the program is shipped or scrapped, whichever comes first.

After all the hardware registers and RAM that you are planning to use have been defined, you are ready to start entering data. Where you put the data is a matter of available space and personal preference. If you are going to put data immediately preceding the code, it would be a wise move to put a jump to the first instruction of your program before you define your first byte of data. In this way, you will always know what the starting address of your program is, no matter how the size of the data section may change. The data section will contain all the data that is not code.

This may seem like quite a few preliminaries to the actual program, but all of the steps do need to be taken. The actual source code that you will be creating will be making constant references to all of the names and definitions that have been defined previously.

When you are starting any program, especially one in assembly language, it is important to break the program into a number of smaller routines. Quite often, the small routines can be individually tested and later merged to form a complete program. Also, small segments can be saved for use later in other similar programs so that you won't have to start from scratch every time. Unless you write perfect programs every time, small program segments will be much easier to debug. If you write an entire program and then try to make it run, it can become quite difficult to determine where in the program the problem is. On the other hand, if you had tested all the small program segments before merging them into a complete program, the only problems that you might encounter should be in the interconnections between the program segments. Since you already know that all of the pieces of the program work, you should not have any difficulty finding the bugs.

As you are writing your program, do not be alarmed if you realize that you have not defined something you need to use. Simply write what you have forgotten on paper and use it in the code as if it had been defined. Then, when you feel like taking a break from the creative process, go back and insert your addition into the proper definition file. Until you try to assemble your program, the computer does not know or care what has or has not been defined.

You may have been wondering why the subroutines should be placed at the end of your program. Unlike the macro library, any subroutine in your program will use a certain amount of memory, whether it is used or not. Because of this, any subroutine that is not used should be deleted so as not to waste memory space or the time that it takes the assembler to assemble the subroutine.

By following this general outline, you will have a manageable and modular way with which to approach the design of your program. Most assemblers have a command that allows you to chain together different parts of your program. If your assembler has the ability, you may find it desirable to write all of the different modules of your program as separate files, and then let the assembler link them together as it assembles the program. This has some advantages over creating one massive file. For instance, if you need to add a definition to your RAM definitions, you only need to load the file with your other definitions. If your disk drive is particularly slow (as all Commodore 64 drives are), you will find a partitioning of your program quite a time saver. As your program progresses, the definitions and data areas will rarely need modifying. Keeping all of the parts of the program separate allows you to edit or print the part of the program that you are currently working on without having to deal with those parts of the program that have been tested and debugged.

# Chapter 6
# Working with Interrupts

In the Commodore 64 interrupts serve as a major source of timing and program control. The interrupts are normally used to maintain the real time clocks and the type ahead keyboard buffer. Interrupts can also be used to signal sprite collisions and inform the program when a specific scan line has been reached.

To best understand what an interrupt is, consider a normal program. The microprocessor reads its instructions one at a time and executes them in order. Whatever it is told to do first, is done first. If there is a certain condition that makes it necessary to perform a certain operation immediately, this condition must be repeatedly checked throughout the program to ensure it is taken care of promptly. For example, a collision between a bullet and a player sprite should immediately initiate an explosion sequence. In a normal program, you have to monitor the collision status register constantly and take appropriate action.

The alternative is to let the hardware check for the collision. When a collision occurs, the VIC II chip can send an interrupt request to the microprocessor. If interrupts are enabled, the processor will execute an indirect jump through location $FFFE when it finishes executing its current instruction. Location $FFFE usually points to a point in ROM that has an indirect jump instruction for a point in RAM. In the Commodore 64, the RAM location that ultimately will direct the jump is $0314. If the address of your routine to initiate the explosion sequence is placed in locations $0314 and $0315, this sequence will only be called on when a collision is detected.

Using an interrupt for this purpose relieves the main program of scanning the collision register constantly. Because of this, less of a burden is placed on the microprocessor during the main program.

Interrupts should be used for the part of your program that needs the highest priority in terms of microprocessor time. For instance, if you want to change the background color at a certain point on the screen, you need to use an interrupt. The VIC II chip can generate an interrupt on any scan line that you specify. This type of interrupt is called a *raster* interrupt. By using raster interrupts, your interrupt routine can gain access to the processor at a specific (relatively) point on the screen.

Interrupts are useful when the main program involves lengthy calculations or is going to be busy for quite some time. If you are trying to maintain animation while the main program is running, you need to use some form of interrupt to take control at least once every other screen. Otherwise, the animation will appear jerky.

Before enabling your interrupt routine and thereby disabling the Commodore 64's operating system, you should disable all other sources of interrupts in the machine. Once this is done, you can always find the cause of the interrupt easily. Most of the functions performed by Commodore's operating system either are unnecessary in a game or can be done in a different manner.

The KILL macro essentially shuts down all of the devices in the system capable of generating interrupts. Once the interrupts from these chips have been disabled, you can safely change the interrupt vectors to point at your interrupt routine.

After the KILL macro is called, there are no interrupts generated in the system. This is a good time to change the addresses stored in the interrupt vectors. The nonmaskable interrupt vector (NMINV) should be changed to point at a return from interrupt (RTI) instruction. Nonmaskable interrupts are rarely, if ever, used in a game program. The maskable interrupt vector (CINV) should be changed to point at the first instruction of your interrupt routine.

Listing C-3 in Appendix C is the source code for a program that uses a RASTER interrupt to change the background color of the screen in the middle of the screen. This is also a good time to ensure that you are using your assembler properly. If you can successfully enter and run this short program, you should have no problem getting some of the longer programs to run later. The macro library and the system definitions that were defined earlier (Listings C-1 and C-2) are inserted into the program by the .LIB directive of the assembler. An executable version of this program is shown in Listing C-4.

To run the executable form of this program enter the following commands:

```
LOAD "DEMO.O",8,1
SYS 4096
```

After setting up the system, the main program just loops through itself. The only way that any changes can occur is through interrupts. The first interrupt is generated at the mid point in the screen, as specified by the first RAST macro. After changing the screen color to blue, this interrupt uses the RAST command to set an interrupt to occur at the bottom of the screen. Next, the address of the second interrupt is placed in the interrupt vector. The second interrupt (INT1) works in the same manner as the first, only it points to the first interrupt when it is done. By using two interrupt service routines in this manner, different things can be done on each half of the screen.

You may have noticed that you can see the point on the screen where the colors change, and it seems to be moving. This occurs because the microprocessor must finish the instruction it is currently working on before it can process the interrupt. Since an instruction may take from two to six machine cycles to execute, and the electron beam travels about three pixels per instruction cycle, the color can change in an 18 pixel area. This assumes that the VIC II chip is consistent about when it notifies the processor about the interrupt. Any timing inconsistencies in the VIC II chip enlarges the area where the color changes.

By adding a delay in the interrupt service routine before changing the background color, you can force the color change area to be in the border where it won't show.

After running this program, you can not reset the Commodore 64. By disabling the operating system, you have disabled the keyboard scan routines, effectively making the computer deaf to outside stimuli. When you have finished watching your new program and want to reset the machine, turn it off and back on again.

Warning: Always be sure that you have saved your program and source code before trying to run a new program!

# Chapter 7
# Technical Information

Up until now, this book has dealt with concepts and generalities when referring to the hardware in the Commodore 64. This was important to get you used to the capabilities of the hardware without bogging you down with details. This chapter will go into the details of getting the computer to generate the effects you are after. This chapter will probably be the chapter to which you will refer most often when writing a game program. All of the information that you will need in order to program the VIC II chips and the SID chip will be explained in this chapter. After you have become familiar with the hardware in the Commodore 64, you will not need the full explanation of the hardware registers. When you are up to such a level, you will find it easier to use the listing of the SYSDEF file in Appendix C as a quick reference guide to the registers.

Unless otherwise noted, all references to addresses in this chapter use hexadecimal notation. The names that have been assigned to the registers are the standard names that have been defined in the hardware definition listing, SYSDEF (Listing C-2 in Appendix C). By using this naming convention for the registers, you will begin to gain familiarity with the register names as they are used in the assembler. All of the names are made up of 6 characters or less, so no matter what type of assembler you are using, the same names will be acceptable.

Macros that supply many of the functions described below have been provided. Descriptions of all of the macros can be found in Appendix B, and Listing C-1 in Appendix C provides the source code. The macros may use a few more instructions to perform the function than would be required if you were to provide the data yourself. They do have advantages, however. Your program will be easier to understand if you use the macros, as there will be a recognizable name given to the macro as opposed to a sequence of assembly language instructions.

## COMMODORE 64 ADDRESS SPACE

As you know, inside the Commodore 64 is a 6510 microprocessor, which controls the machine.

It uses a 16 bit address bus allowing it to access 2 46 or 65536 bytes of memory. This is all the memory that can be accessed at one time. Fortunately, as mentioned briefly before, through a technique called *bank switching*, different types of memory can be switched into or out of this address space. Through the use of bank switching, the 8K BASIC ROM can be accessed instead of 8K of RAM. The Commodore 64 switches other sections of RAM with ROM, and also switches an area of RAM with some hardware registers, such as the VIC II chip.

When choosing the appropriate memory map for your program, you must decide which of the functions provided by the Commodore 64 you will be using. For instance, if you do not need BASIC, you can switch the BASIC ROM out of the memory space. Doing so will give you 8K of RAM that you would not otherwise be able to use.

Some of the memory maps available to you will allow you to switch the 4K I/O space at $D000 with 4K of RAM. What this means is that you will no longer have access to the hardware registers at these locations. When it becomes necessary to change any of the values in one of these registers, you will need to switch the I/O space back into the RAM space. This is generally more trouble than it is worth, unless you desperately need the extra memory.

You also have the option of switching out the 8K KERNAL ROM. In most cases, you will not be using any of the KERNAL routines. If this is the case, there is no reason to keep it in memory. 8K of RAM can be switched into the space where the KERNAL ROM was.

**Caution:** All interrupts in the system should be shut down before the KERNAL ROM is switched out of memory. The six bytes of memory from $FFFA to $FFFF in the KERNAL contain the vectors for the interrupts. After the KERNAL has been switched out, new interrupt vectors need to be stored in RAM.

## MEMORY CONTROL AND MAPPING

There are five control lines that control the bank switching of the different memory areas. Three of these lines are controlled by the microprocessor using its internal I/O port.

The three internal control lines are the least significant three bits at address $0001. This is the hardware I/O port of the 6510 processor. Data can be written to this port just as it can be written to any other memory location. Bits 0 and 1 are used to select from the four primary memory maps that are available. Bit 2 selects whether the I/O devices or the character generator ROM will be addressable by the microprocessor in the range of addresses from $D000 to $DFFF. Bit 2 has no effect on the system when the memory map with 64K of RAM accessible is selected. The remaining two control lines are internally pulled high when there is no cartridge plugged into the computer and can be ignored for a disk based program.

Figure 7-1 shows all of the memory mapping possibilities.

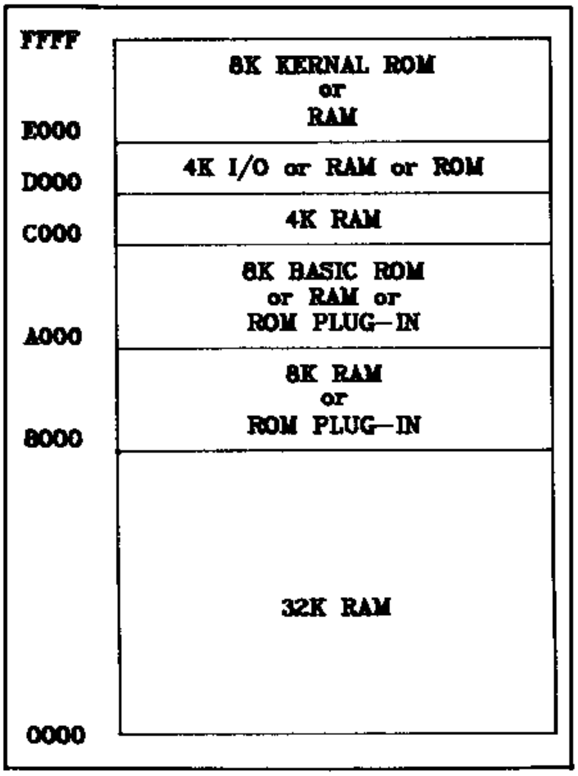


Fig. 7-1. Memory mapping options chart.

The memory maps in Figs 7-2, 7-3, 7-4, and 7-5 can be selected through software.

## GRAPHICS MEMORY LOCATIONS

Although the Commodore 64 has 64K of memory, the VIC chip can only reference 16K of memory at any one time. Fortunately, you can change which of the four 16K blocks of memory in the computer the VIC chip will be able to use. When the Commodore 64 is powered up, bank 0 of memory ($0000-$3FFF) is selected. If you wish to change the bank of memory that the VIC chip will use, you must set the least significant two bits of $DD00 to the value that represents the desired bank. Before doing so, you must set bits 0 and 1 of $DD02 to 1. This will select the control bits that

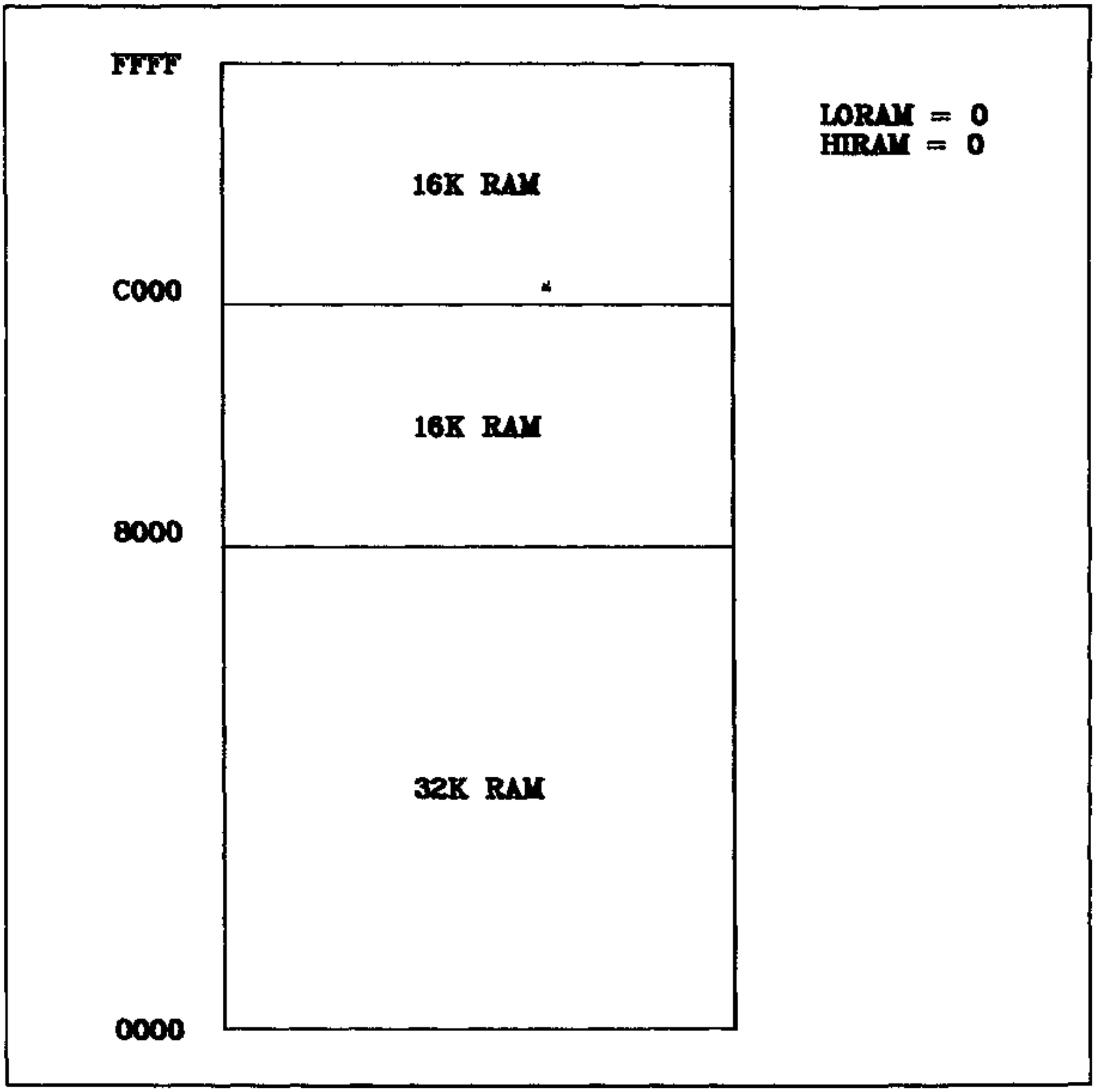


Fig. 7-2. 64K RAM memory map.

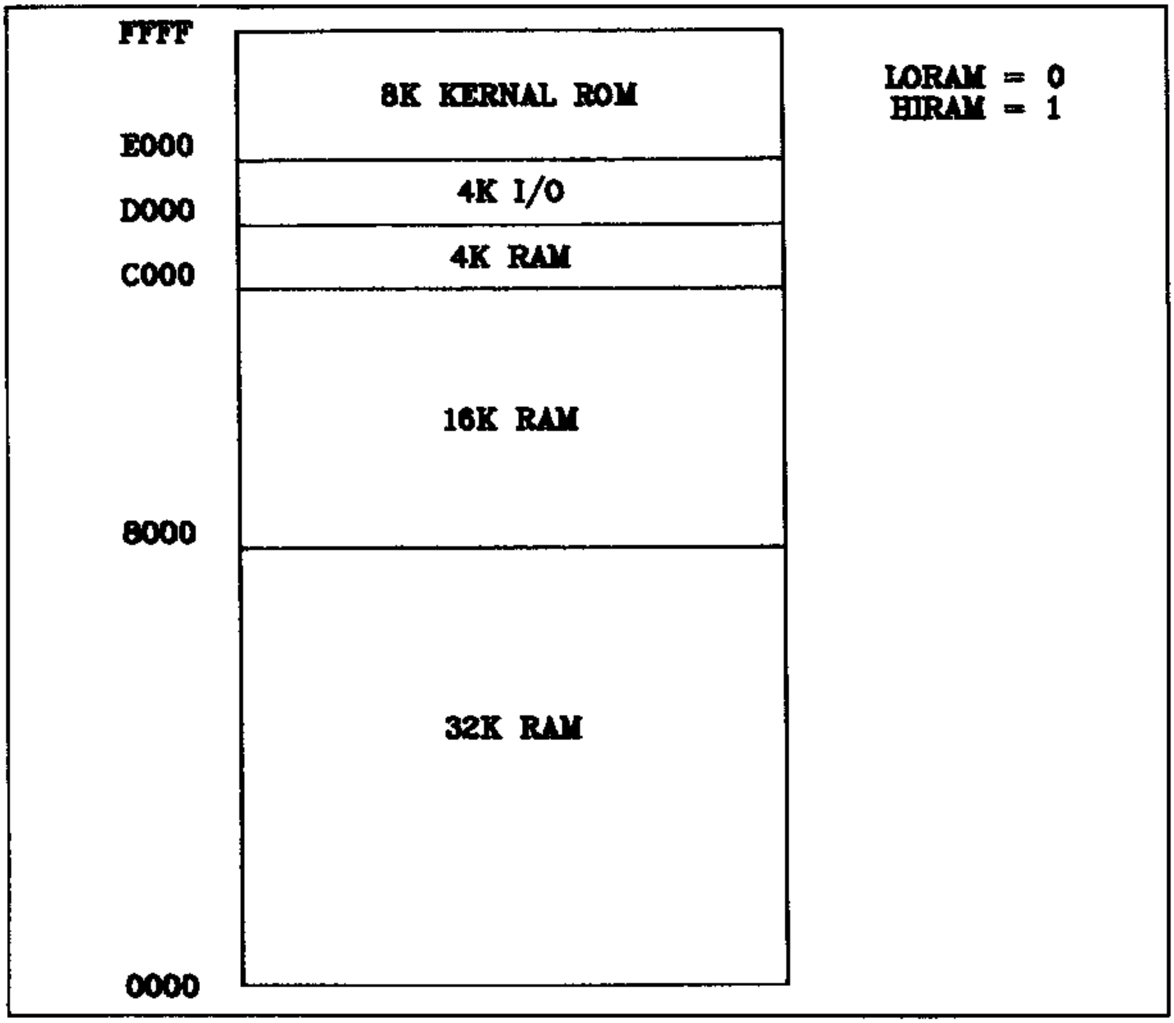


Fig. 7-3. 52K RAM memory map.

control-memory select to outputs. The BANK macro will select the proper bank for you. If you want to change the bank yourself, use the information in Table 7-1 for the appropriate values:

**Table 7-1. The Values to Use When Selecting the Bank of Memory the VIC Chip Will Use.**

| VALUE | BANK # | MEMORY RANGE ADDRESSED |
|---|---|---|
| 3 | 0 | $0000-$3FFF |
| 2 | 1 | $4000-$7FFF |
| 1 | 2 | $8000-$BFFF |
| 0 | 3 | $C000-$FFFF |

## STANDARD TEXT MODE

When the Commodore 64 is first turned on, it powers up in its standard text mode. When in this mode, the screen is arranged as 40 characters by 25 lines. This gives a total of 1000 characters that can be displayed on the screen at any one time. The character to display in each position can be found in the 1000 bytes of RAM starting at $0400. If you want, you can instruct the VIC chip to use a different area of memory to find the text to be displayed. The upper four bits of the VIDBAS register ($D018) control where the VIC chip will find the text. Text memory will always be found on $0400 boundaries.

Each byte in the text memory area is used as an index into a character generator section of memory. Since a byte can have 256 values, each position on the screen can display one of 256 patterns. When the Commodore 64 is powered up, the graphics information that the text memory references is in the character generator ROM. The VIC chip can be instructed to use a different area of memory as the character generator by changing the lower four bits of the VIDBAS register ($D018). This section of memory will be referred to as graphics memory, as the information can be of any type of graphics, not necessarily text.

The Commodore 64 has a built-in character

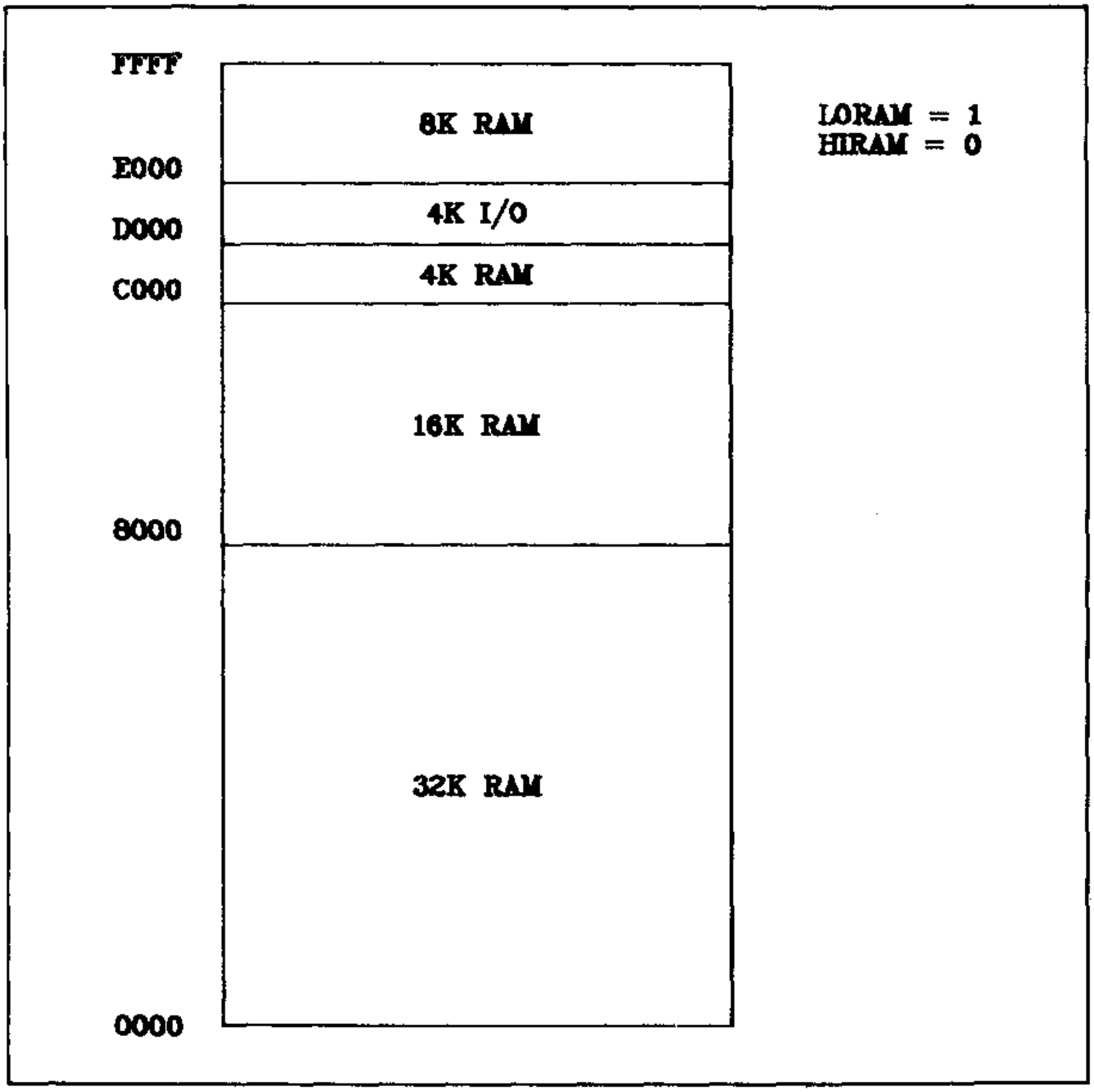


Fig. 7-4. 60K RAM memory map with no KERNAL ROM.

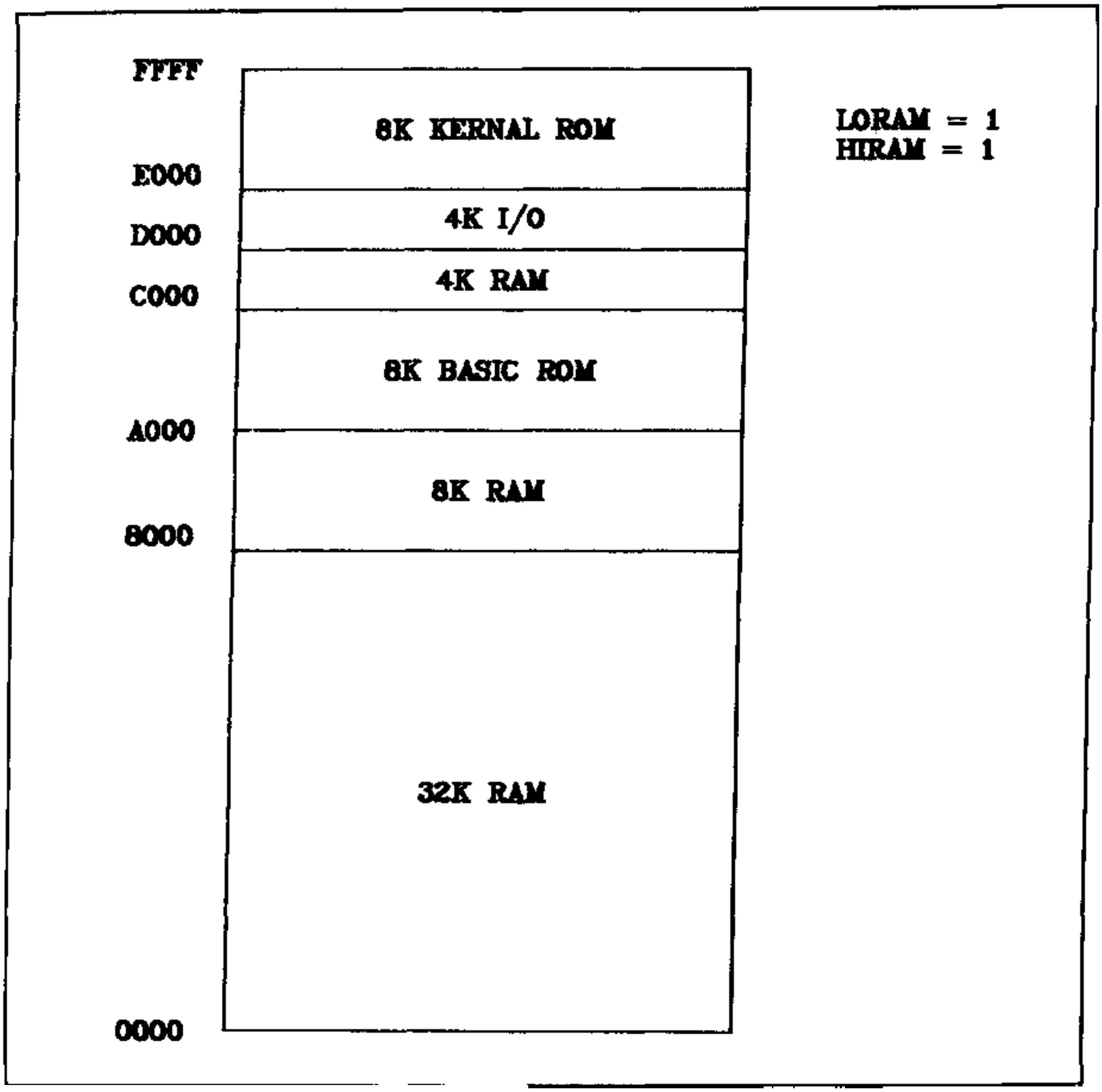


Fig. 7-5. The default memory map with 42K RAM.

generator ROM, which is physically at SD000-$DFFF. You may have noticed that this is the same address range where the hardware registers in the Commodore 64 are. This is possible only because the hardware registers are not available to the microprocessor at the same time as the character generate ROM is. Bit 2 of the I/O port at $01 controls whether the ROM or hardware registers will be available to the microprocessor. When this bit is set to a 1, the hardware registers will be available. When this bit is set to 0, the character generator ROM can be read by the microprocessor. The VIC chip does not see the character generator at this address, however. The VIC chip has been tricked into seeing the character generator from $1000-$1FFF. The VIC chip also sees an image of the character generator ROM from $9000-$AFFF. Notice that the VIC chip can only see an image of the character generator when in banks 0 and 2 of memory. The microprocessor does not see the character generator in the same place that the VIC chip does, so there is no conflict with the microprocessor when using one of the areas in memory to store data that the VIC chip thinks is the character generator. You will normally choose either bank 1 or 3 for your graphics, so that the image of the character generator does not get in your way. If you do need to use text in your program, you can either copy data out of the character generator or create your own character set.

The TXBAS macro can be used to change the base address where the text will be located. Similarly, the GRABAS macro can be used to change the base address of graphics memory. It is important to remember that the text and graphics base addresses must be added to the bank that is currently selected in order to find the absolute address in memory where the data will be found. For example, if you had selected bank 1 of memory ($4000-$7FFF) and the text base address was 80400, the text data would be found at $4400. If you want to change the base address of the registers yourself, the values to use can be found in Table 7-2. When changing a value for the TEXT mode, only the upper bits should be changed in the VIDBAS register. Similarly, when changing a graphics base address, only the lower bits should be changed. A dash indicates a don't care condition. You must remember to add the BASE address register to the addresses to find the real address.

**Table 7-2. The Values To Use To Change the Base Address for Text or Graphics.**

| VIDBAS VALUE Upper Bits | TEXT BASE ADDRESS |
|---|---|
| $0- | $0000 |
| $1- | $0400 |
| $2- | $0800 |
| $3- | $0C00 |
| $4- | $1000 |
| $5- | $1400 |
| $6- | $1800 |
| $7- | $1C00 |
| $8- | $2000 |
| $9- | $2400 |
| $A- | $2800 |
| $B- | $2C00 |
| $C- | $3000 |
| $D- | $3400 |
| $E- | $3800 |
| $F- | $3C00 |

| VIDBAS VALUE Lower Bits | GRAPHICS BASE ADDRESS |
|---|---|
| $-0 | $0000 |
| $-2 | $0800 |
| $-4 | $1000 * |
| $-6 | $1800 * |
| $-8 | $2000 |
| $-A | $2800 |
| $-C | $3000 |
| $-E | $3800 |

* In BANK 0 and 2, ROM images will appear here.

## COLOR MEMORY

As discussed earlier, when in the text mode, the screen is arranged as 25 lines of 40 characters. Each of the characters can be one of 256 different patterns. In addition to being able to select the character to be displayed at each position on the screen, you can also select the color that you would like each character to be. The Commodore 64 provides 16 colors that can be used. There is an area in memory that is reserved to hold the color information. This section of memory is called the *color RAM.* Color RAM starts at $D800 and continues to $DBE7. Unlike the normal RAM used in the Commodore 64, color RAM is made up of nibbles rather than bytes. When you read a value out of color ram, only the least significant four bits are valid. You will read an eight bit value, but the data in the upper four bits is not predictable. Quite often, you must mask the upper four bits before using data from the color RAM.

Color memory does not move, and the VIC chip cannot use a different section of memory for the color information. The MVCOL macro will fill color

RAM with a single color. For instance, if you wanted all of the text tobe white, you would enter "MVCOL WHITE." The text memory and color RAM are treated differently by the different graphics modes. If your picture doesn't look like you expect, check to be sure that the proper graphics mode has been selected.

## CUSTOM CHARACTER SETS

Although the Commodore 64 has a built in character set, you will probably find that set limiting in what it allows you to do. The text mode can be used for quite a few different types of games if you redefine the character set. You can build up figures that are larger than one character size by placing the different pieces of the figure in adjacent locations. This gives you quite a bit of flexibility in using graphics as long as you define your own character set.

Each character is defined in memory as an array of 8 by 8 bits. This is stored as eight sequential bytes. Each byte has eight bits, and each bit represents one pixel on the screen. If a bit is on, the corresponding pixel on the screen will be turned on in the character color. If the bit is off, the background color is used for that pixel. The value in the text RAM is used as a lookup into the graphics RAM to choose which set of eight bytes will represent a character. If you decide to make your own character set, you will have to define any text characters that you may need as well as your graphics figures. Once you instruct the VIC chip to get its graphics data from RAM, it will no longer be able to use the character generator ROM.

Using a machine language monitor, you can enter the data for your character set into RAM. When you are finished, you should save your new set to the disk for later use. The first 8 bytes in your character set will be displayed if you place a 0 into a text memory location (assuming that you had previously instructed the VIC chip to get its graphics information from the place in memory where your new character set was placed). Your character set must begin on a multiple of $0800 in the current bank. The character set may not be placed in one of the locations where the VIC chip can see the character generator ROM.

## MULTICOLOR MODE

To this point, all of the graphics in the text mode have been able to use only two colors for each character position on the screen. Normally a video game will use many more than two colors. To allow more colors to be displayed in a given area on the screen, the Commodore 64 has a multicolor mode that can be selected. In this mode, the character can use the character color and the colors in BCOL0 ($D021—the background color), BCOL1 ($D022), or BCOL2 ($D023). Using this mode, each pixel in a character location can be one of 4 colors. Unfortunately, you have to sacrifice 1/2 of the horizontal resolution to use this mode. This is usually a reasonable sacrifice considering how much more color can be used.

In order to turn on the multicolor mode, you must set bit 4 of the XSCRL° ($DO12) or use the MULTON macro. To turn off the multicolor mode, you must clear bit 4 of the XSCRL register or use the MULTOF macro. In the text mode, the multicolor mode is selected individually for each character position on the screen. If the color in color RAM for a given character position is less than 8, that character position will be in the standard text mode. If the color in color RAM is 8 or greater, that character position will be in the multicolor mode. This allows you to mix standard and multicolor modes on the same screen.

As stated earlier, you sacrifice 1/2 of the horizontal resolution of the normal text mode when you select the multicolor mode. This means that each character will be made up of an array of 4 by 8 pixels. It still takes 8 bytes to define a character, but instead of each bit corresponding to a pixel on the screen, each PAIR of bits represents one of 4 registers to use to find the color for the bit pair. The following chart shows the correlation between bit pairs and registers.

| **Bit Pair** | **Register** |
|---|---|
| 00 | BCOL0 |
| 01 | BCOL1 |

**Bit Pair Register**

| | |
|---|---|
| **10** | **BCOL3** |
| **11** | **Lower** 3 bits of COLOR RAM |

## EXTENDED BACKGROUND COLOR MODE

In certain applications where you do not need a very large character set (less than 65 characters), you can use more colors than the standard text mode will allow by using the extended background color mode. In this mode, you do not sacrifice any resolution to gain extra colors; you only sacrifice the number of characters available from your character set. You can control both the foreground color (using the color RAM) and the background color of each character. The two most significant bits of the character code are used to select a background color from one of four registers, BCOL0 to BCOL3. Because of this, only the first 64 characters from your character set can be used.

The extended background color mode is enabled by setting bit 6 of the YSCRL ($D011) register. This mode can be turned off by clearing bit 6 of the YSCRL register. The following chart shows the relationship between the most significant two bits of the character code and the background color:

| **Bit Pair** | **Register** |
|---|---|
| **00** | **BCOL0** |
| **01** | **BCOL1** |
| **10** | **BCOL2** |
| **11** | **BCOL3** |

## BIT MAPPING

Even though the various text modes available to you in the Commodore 64 provide many different options for displaying graphics, they still restrict you to using predefined shapes. For the majority of video game applications, you will be unable to use a text mode. The Commodore 64 has a high resolution bit mapped graphics mode that allows you to control each pixel on the screen individually. The display has a resolution of 320 pixels horizontally by 200 pixels vertically. This gives a total of 64000 pixels that can be controlled on the screen. Since each pixel is represented by a bit in graphics memory, 8000 bytes of graphics memory are required to represent the display.

There are two different types of bit mapped modes available on the Commodore 64. They are:

- Standard bitmapped mode: 32011 by 20OV, two colors per 8 by 8 group of pixels.
- Multicolor bitmapped mode: 16OH by 20OV, four colors per 8 by 8 group of pixels.

To turn on the standard bitmapped mode, you must set bit 5 of the YSCRL register. The GRAPH macro will perform this function for you. To turn off the bitmapped mode, you must clear bit 5 of the YSCRL register. You can use the TEXT macro to perform this function.

In the standard bitmapped mode, the colors for each 8 by 8 group of pixels are stored in text memory. The high 4 bits control the color of a pixel if its associated memory bit is on, while the lower 4 bits specify the color of a pixel if its bit is off. Color RAM is not used in the standard bitmapped mode. Table 7-3 shows how the bytes of graphics memory are organized with regard to the screen. This sequence is repeated for all 25 rows.

**Table 7-3. The Bytes of Graphics Memory as Organized with Regard to the Screen.**

| | | | | | |
|---|---|---|---|---|---|
| ROW 0 | 0 | $8 | $10 | $18 ......... | $138 |
| | 1 | $9 | $11 | . | $139 |
| | 2 | $A | $12 | . | $13A |
| | 3 | $B | $13 | . | $13B |
| | 4 | $C | $14 | . | $13C |
| | 5 | $D | $15 | . | $13D |
| | 6 | $E | $16 | . | $13E |
| | 7 | $F | $17 | . | $13F |
| ROW 1 | $140 | $148 | $150 | $158 ........ | $278 |
| | $141 | $149 | $151 | . | $279 |
| | $142 | $14A | $152 | . | $27A |
| | $143 | $14B | $153 | . | $27B |
| | $144 | $14C | $154 | . | $27C |
| | $145 | $14D | $155 | . | $27D |
| | $146 | $14E | $156 | . | $27E |
| | $147 | $14F | $157 | . | $27F |

### Multicolor Bitmapped Mode

To turn on the multicolor bit mapped mode, you must first turn on the bit mapped graphics mode as shown above. Then you must set bit 5 of the YSCRL register. You can turn off the multicolor mode by clearing bit 5 of the YSCRL register. The macros MULTON and MULTOF can be used to turn on and off the multicolor modes.

In the multicolor mode, four colors can be displayed in each 4 by 8 group of pixels. Each byte in graphics memory is broken down into 4 bit pairs. Each bit pair specifies where the color information will be found for each pixel. In addition to the two colors that are defined in text memory, the color RAM is used in this mode to hold one more color. The fourth color is the background color that is stored in BCOL0 ($D021). The correspondence between the bit pairs in graphics RAM and where the color is found is shown in the following chart:

| **Bit Pair** | **Color** |
|---|---|
| 00 | BCOL0 |
| 01 | UPPER NIBBLE OF CHARACTER RAM |
| 10 | LOWER NIBBLE OF CHARACTER RAM |
| 11 | COLOR RAM |

## SPRITES

A *sprite* is a small moveable object block that can move independently of the background graphics. The VIC chip can display eight sprites on the screen at any one time. Sprites can be displayed on any one of the display modes, and they will look the same in all of them. You can have up to 256 different sprites defined at any one time, but only eight can be displayed at the same time. The sprite to be displayed can be changed by changing a one byte pointer, so animation can be easily performed by quickly switching through a few different sprite patterns. Sprites can be moved very smoothly by simply giving the VIC chip the X and Y coordinates of the upper left corner of the sprite.

Sprites have different display priorities. That means that the sprite with a higher priority will appear to move in front of a sprite with a lower priority. This can be used to give the illusion of three dimensional movement. The priority of a sprite to the background graphics is individually selected for each sprite. If the background is given priority, the sprite will appear to move behind the background graphics. For instance, if a tree was being displayed in the bit mapped graphics mode, and a sprite in the shape of a dog was to move past the tree, the dog would appear to be moving behind the tree.

Each sprite is a block 24 pixels horizontally by 21 pixels vertically. The pixels that are set to one use 1 of the 16 available colors. The pixels that are set to zero allow the background color to show through (are transparent). Like the other graphics modes, a sprite can be selected to be in the multicolor mode, giving it a resolution of 12 by 21 in three colors plus transparent. Wherever a sprite is transparent, whatever is behind the sprite will show through.

For those times when a larger sprite is necessary, the VIC chip has the option of doubling the horizontal size, the vertical size, or both. You will not increase the detail available in your sprite by using one of the multiply options, only the size. When a sprite is expanded, each of the pixels is twice the size of the pixels in a normal sprite.

### Sprite Pointers

Once a sprite has been defined, the VIC chip needs to be told where to find the pattern. The sprite definition must be in the currently selected bank of memory for it to be displayed. Since each sprite definition takes up 64 bytes, a sprite definition will always start on a $0040 boundary in memory.

A 16K bank of memory can hold 256 sprite definitions, so it will only require one byte to tell the VIC chip which sprite to display. The sprite pointer is a number which, when multiplied by 64, will give the starting address of the sprite definition. Sprite definitions may not be placed in a section of memory where the VIC chip sees an image of the character generator ROM.

The VIC chip will read the eight sprite pointers from the last eight bytes of the 1K of text memory, an offset of $03F8 from the text base address. Since only 1000 out of 1024 bytes of text memory are used to display characters on the screen, the sprite pointers will not interfere with screen graphics. For example, since the default setting of the text memory is at $0400, the first sprite pointer will be $07F8.

## Sprite Controls

For most of the sprite control registers, each bit in the register corresponds to one of the sprites. For example, bit 0 represents sprite O, bit 1 represents sprite 1, and so on. The rest of the sprite controls require a value (such as a vertical location), so there is one register for each sprite.

**Enabling a sprite.** Before a sprite can be seen, it must be enabled. The register SPREN ($D015), has an enable bit for each sprite. If the bit is set, the sprite will be enabled. The sprite will only be seen if the X and Y positions are set to the visible portion of the screen. A sprite can be disabled by clearing the appropriate bit.

**Setting the sprite color.** There are eight registers that are used to hold color information, one for each sprite. Any of the 16 available colors may be selected for each sprite. Each bit that is set in the sprite definition will cause a pixel to be displayed in the sprite color. If the bit is clear, the pixel will be transparent. The sprite color registers are:

| Name | Address |
|---|---|
| SPRCL0 | $D027 |
| SPRCL1 | $D028 |
| SPRCL2 | $D029 |
| SPRCL3 | $D02A |
| SPRCL4 | $D02B |
| SPRCL5 | $D02C |
| SPRCL6 | $D02D |
| SPRCL7 | $D02E |

**Setting the multicolor mode.** The multicolor mode can be individually selected for each ($D01C)/sprite by setting the appropriate bit in the MLTSP ($D01C) register. Setting a bit will enable the multicolor mode, clearing the bit will disable the multicolor mode. When the multicolor mode is enabled, the horizontal resolution drops from 24 pixels across to 12 pixels. Each pair of bits in the sprite definition is treated as a bit pair, whose value determines which of the four colors will be selected for the pixel. Table 7-4 shows the relationship between the bit pairs and the color registers.

**Table 7-4. The Relationship Between the Bit Pairs and the Color Registers in the Multicolor Mode.**

| Bit Pair | Description |
|---|---|
| 00 | TRANSPARENT, SCREEN COLOR |
| 01 | SPRITE MULTICOLOR REGISTER #0 ($D025) |
| 10 | SPRITE COLOR REGISTER |
| 11 | SPRITE MULTICOLOR REGISTER #1 ($D026) |

**Using the sprite multipliers.** Each of the sprites can be expanded in either the X or Y direction. When a sprite is expanded, each pixel is displayed as twice the normal size in the direction of the expansion. The resolution of the sprite does not increase, only the size.

To expand a sprite in the X direction, the appropriate bit must be set in the SPRXSZ ($D01D ) register. To return the sprite to its normal size, clear and bit.

The expansion of a sprite in the Y direction is done in the same way as the X expansion. You must set the appropriate bit in the SPRYSZ ($D017) register to expand the sprite. The sprite can be returned to its normal size by clearing its bit in the SPRYSZ register. The sprite can also be expanded in both the X and Y directions by setting its bit in both registers.

**Positioning sprites.** Each sprite can be positioned independently anywhere on the visible screen and off the visible screen in any direction. Since the screen is 320 pixels wide, it takes more than one byte to specify a horizontal position. Each sprite has its own X position register and Y position register, and a bit in an extra most significant bit register. These registers are shown in Table 7-5.

The location specified by the registers is the position where the upper left corner of the sprite will appear.

**Table 7-5. The Position Registers for the Sprites.**

| Address | Name | Description |
|---|---|---|
| $D000 | SPR0X | SPRITE 0 HORIZONTAL |
| $D001 | SPR0Y | SPRITE 0 VERTICAL |
| $D002 | SPR1X | SPRITE 1 HORIZONTAL |
| $D003 | SPR1Y | SPRITE 1 VERTICAL |
| $D004 | SPR2X | SPRITE 2 HORIZONTAL |
| $D005 | SPR2Y | SPRITE 2 VERTICAL |
| $D006 | SPR3X | SPRITE 3 HORIZONTAL |
| $D007 | SPR3Y | SPRITE 3 VERTICAL |
| $D008 | SPR4X | SPRITE 4 HORIZONTAL |
| $D009 | SPR4Y | SPRITE 4 VERTICAL |
| $D00A | SPR5X | SPRITE 5 HORIZONTAL |
| $D00B | SPR5Y | SPRITE 5 VERTICAL |
| $D00C | SPR6X | SPRITE 6 HORIZONTAL |
| $D00D | SPR6Y | SPRITE 6 VERTICAL |
| $D00E | SPR7X | SPRITE 7 HORIZONTAL |
| $D00F | SPR7Y | SPRITE 7 VERTICAL |
| $D010 | XMSB | MOST SIGNIFICANT BIT REGISTER |

The value placed in the Y position register will specify the vertical position of the sprite on the screen. This value may be up to 255. For an unexpanded sprite to be completely visible, the Y value must be between $32 and $E9. Any other values will place the sprite partially off the screen.

Whatever value is placed in the X position register is the least significant 8 bits of a 9 bit value. Each sprite has a ninth bit in the XMSB ($D010) register. An unexpanded sprite will be completely visible if the 9 bit X value is greater than $18 and less than $140. The HINC and HDEC macros can be used to perform 9 bit increments and decrements of the X position.

Table 7-6 shows the screen coordinates for expanded and unexpanded sprites to be fully visible on the screen. Any sprite positions outside of these limits will be partially or fully off of the screen. This provides an easy way to reveal a sprite gradually.

**Assigning sprite priorities.** As mentioned before, each sprite has a display priority with

**Table 7-6. The Screen Coordinates at which Normal and Expanded Sprites Will Be Fully Visible.**

| POSITION | X | Y | X EXP | Y EXP |
|---|---|---|---|---|
| UPPER LEFT | $18 | $32 | $18 | $32 |
| UPPER RIGHT | $140 | $32 | $128 | $32 |
| LOWER LEFT | $18 | $E5 | $18 | $D0 |
| LOWER RIGHT | $140 | $E5 | $128 | $D0 |

respect to the other sprites and to the background You can create a three dimensional effect by allow ing different sprites to pass in front of each other The priority of one sprite to another is predetermined by the VIC chip. Sprite 0 has the highest priority, meaning that it will appear to be in front of all other sprites. Sprite 7 has the lowest priority of all the sprites.

Each sprite can be individually selected to either have a higher priority than the background or a lower priority. If the sprite's bit in the I ($D01B) register is clear, the sprite will appear tc pass in front of the background. When the bit for the sprite is set in the **BPRIOR** register, the sprite will appear to move behind the background image and in front of the background color. Because sprites can have transparent as one of their colors, any sprite that passes behind a higher priority sprite with transparent in it will show through in the transparent areas.

## COLLISION DETECTION

The VIC chip can detect collisions between sprites and also between a sprite and the background. The VIC chip will defect collisions between the nontransparent portions of sprites.

When a collision between two sprites occurs, their bits are set in the SSCOL ($D01E) register. The data in the SSCOL register will stay valid until the byte is read. After the register is read, the data will be cleared, so it is important to store the data somewhere before analyzing it. The VIC chip will detect a collision even if the sprites are off the screen. One thing that should be noted is that the SSCOL register will only tell you which sprites are

:involved in collisions, not which sprite hit which sprite. If you are multiplexing sprites, the data in the SSCOL register may be useless.

Sprite to background collisions are handled in almost the same way. The SBCOL ($D01F) register will detect a collision between the nontransparent portion of a sprite and the background. In a multicolor screen mode, the bit pair 01 is considered transparent for collision detection. Like the SSCOL register, the data is cleared after reading it.

## BLANKING THE SCREEN

The entire screen can be blanked to the border color by clearing bit 4 of the YSCRL register. The screen can be turned back on by setting bit 4 of the YSCRL register. Blanking the screen does not disrupt any data on the screen. When the screen is blanked, your program will run slightly faster because the VIC chip doesn't need to fetch any data from memory.

## THE RASTER REGISTER

The VIC chip keeps track of which scan line the electron beam is currently on. Since there are more than 255 scan lines in one TV frame, this will be *a* 9 bit value. The least significant 8 bits of the current scan line can be read by reading the RASTER 1$D012) register. The ninth bit can found in bit 7 of the YSCRL register.

You can write a 9 bit value to the RASTER register and bit 7 of the YSCRL register. When the scan line reaches the value that you stored, bit 0 of the VIRQ ($D019) register will be set. If bit 0 of the VIRQM ($D01A) register is set, an interrupt will be sent to the microprocessor. You must remember to store a ninth bit when storing a RASTER number, or the comparison will not take place. The RAST macro will set the 9 bit raster number for you.

## VIDEO INTERRUPTS

Different conditions within the VIC chip can generate interrupts. The interrupt status can be read by reading the video interrupts register, VIRQ ($D019). The bits have the following meanings:

| **Bit** | **Type of Interrupt** |
|---|---|
| 0 | RASTER |
| 1 | SPRITE TO BACKGROUND COLLISION |
| 2 | SPRITE TO SPRITE COLLISION |
| 3 | LIGHT PEN |
| 7 | SET ON ANY ENABLED INTERRUPT |

Once an interrupt bit has been set, a 1 must be written to that bit position in order to clear it. This allows you to process interrupts one at a time, without having to store the data elsewhere.

Interrupts will only be sent to the microprocessor if the corresponding bit in the *video interrupt mask* register, VIRQM ($D01A), is set. You will still be able to read the interrupts from the VIRQ register, but if the appropriate bit in the VIRQM register is not set, no interrupts will be generated. See the section on using interrupts for more information on using interrupts properly.

## SCROLLING

One of the most advanced features of the VIC chip is its ability to smoothly scroll the screen in either the X or Y direction. The VIC chip can scroll the screen using hardware, freeing the microprocessor from the task of finely scrolling the screen. When the screen needs to be scrolled, the VIC chip can be instructed to scroll the screen within a range of 8 pixels in the X direction, the Y direction, or both.

The least significant three bits of the YSCRL ($D011) register control the amount of vertical scrolling. Since this register is also used for a number of control functions, the register should be read before it is changed. The XSCRL ($D016) register works in the same way as the YSCRL register except that the XSCRL register controls the amount of horizontal scrolling. When changing either of these registers, the lower 3 bits should be masked to 0, and the number of pixels to be scrolled should be OR'd to the new value. The result of this procedure can then be stored back into the register.

The following is a routine that can be used to change the value of the YSCRL or XSCRL register. This example shows how to set the YSCRL register to a scroll value of 7 without disturbing the value of the upper bits of the register.

```
LDA YSCRL ;LOAD THE DATA
AND #$F8 ;MASK THE LOWER 3 BITS
ORA #$07 ;SCROLL 7 PIXELS
STA YSCRL ;STORE THE NEW VALUE
```

As the scrolling value goes from 0 to 7 in the YSCRL register, the screen will scroll down. As the value in the XSCRL register goes from 0 to 7, the screen will scroll to the right.

When scrolling the screen, you will usually want to expand the border area of the screen. This will give you an area to place the new graphics to be scrolled onto the screen where they will not be seen. The VIC chip has two controls that will expand the border. The first of these is a 38 column mode. This mode can be selected by clearing bit 3 of the XSCRL register. The VIC chip can be returned to the 40 column mode by setting bit 3 of the XSCRL register. In the 38 column mode, one column on the right side of the screen and one column from the left side of the screen are covered by the border color. This will give you a buffer area where changes to the screen will not be seen.

The other border expansion option is useful for vertical scrolling. By clearing bit 3 of the YSCRL register, when the vertical scroll is set to 3, half of the top row and half of the bottom row will be covered by the border. The VIC chip can be returned to the normal 25 row mode by setting bit 3 of the YSCR L register. When the vertical scroll is set to O, the top line will be entirely covered by the border. When the vertical scroll is set to 7, the bottom line of the screen will be entirely covered by the border.

Once you have reached a maximum scroll value in the X or Y direction, you will have to shift each character on the screen in the direction of the scroll in order to continue scrolling. After moving all of the characters on the screen, you can reset the fine scrolling registers to their minimum value and continue to use the hardware registers to scroll the screen.

There are a number of things that must be taken into account when writing a scrolling program. In order for the screen to appear to be in continuous smooth motion, the routine that will shift each character must be extremely fast. Also, if you are using a number of different colors in color RAM, each character in color RAM must be moved in the direction of the scroll at the same time as the characters in screen memory. If you do not need to scroll the entire screen, your program can be much shorter and will run faster. If your program does not run fast enough, you will see breaks in your graphics where the characters that have been scrolled are adjacent to characters that have not yet been scrolled.

If possible, your routine should be fast enough to reposition the entire screen in one screen update time (1/60 of a second). You can get by with a slower routine if you scroll your screen memory into a different area of memory than the one that is currently being displayed. This must be completed before the fine scrolling register reaches its limit, so when the entire screen needs to be repositioned, it will be ready. Instead of repositioning the entire screen at that point, which would have to be done within 1/60 of a second, all you need to do is to use the TXBAS or GRABAS macro to instruct the VIC chic to get the data from the area that has already been repositioned. The macro that you will use depend: on the graphics mode that the screen is in.

## JOYSTICKS

At some point, you will want to allow the player to have control over his character in the game. Th( most common form of input to a video game is joystick. The Commodore 64 has two input port; that can be used for joysticks. By setting both DDRA ($DC02) and **DDRB** ($DC03) to $00, the two ports will be configured as inputs. Once the port; have been configured, the data from the joystick can be read from JOY1 ($DC00) or JOY2 ($DC01) Bit 4 of a joystick port represents the fire button on that joystick. If that bit is clear, the fire button depressed. The lower five bits of a joystick port represent the direction of the joystick as shown below:

| Bit | Direction |
|---|---|
| 0 | UP |
| 1 | DOWN |
| 2 | LEFT |
| 3 | RIGHT |
| 4 | FIRE BUTTON |

When a contact on the joystick is pressed, its corresponding bit in the joystick register is clear. When two out of the lower four bits are clear, the joystick is on an angle. If you find it more convenient or manageable to have a bit set representing a closed switch, the NOT macro can be used to invert the data. Before using the joystick data as any type of an index into a table of data, you must mask the unused bits in the register. You will normally want to mask the entire upper nibble and treat the fire button separately. The following routine will invert the joystick data from port 0 and mask the unused bits as well as the fire bit. The result will be a four bit value that represents the joystick direction.

```
LDA JOY1        ;READ THE JOYSTICK PORT
NOT             ;COMPLEMENT THE
                ;ACCUMULATOR
AND #$OF        ;MASK THE UPPER BITS TO 0
```

# Chapter 8
# Sound Effects

One of the most advanced features of the Commodore 64 is its ability to generate sounds. Built into the Commodore 64 is a highly advanced sound generator, the sound interface device or SID chip. This chip has the capability of generating three independent tones over a range of more than six octaves. It has controls in each channel to control the attack, decay, and release times, and a sustain level for the volume of each channel. In fact, most of the features found in a musical synthesizer can be found in the SID chip.

Figures 8-1, 8-2, 8-3, and 8-4 show some of the different waveforms that the SID chip can generate. Figure 8-1 shows a triangular waveform, which will produce the cleanest tone, as it is a fairly close approximation of a pure sine wave. Figure 8-2 shows a sawtooth waveform. You will notice that it has sharper edges to it than the triangular waveform. Sharp edges on a waveform tend to generate various harmonics of the tone, so in general, the sharper the edges in a waveform, the more harmonics will be generated. The differences in the harmonic content can be heard as a difference in harshness of the tone. A triangular waveform will produce a very smooth, soft tone, while the square waveform, as shown in Fig. 8-3, produces a very sharp biting tone. When you select the square waveform in the SID chip, you have extra control over the tone of the sound. You can program in a pulsewidth for the wave, which varies the symmetry of the square wave. Depending on the settings that you use, the waveform will be more rectangular than square, as shown in Fig. 8-4.

The SID chip can also generate a noise waveform. This is a random signal that changes at the oscillator frequency. Many games will use this waveform to generate explosions, wind storms, or any type of sound that is not a specific tone. In addition, if channel 3 is set to generate noise, the amplitude of the waveform can be read by the microprocessor at any time. Because this is a constantly changing value, the number read will be a random number.

## FILTERING

After you have created a sound, you can change

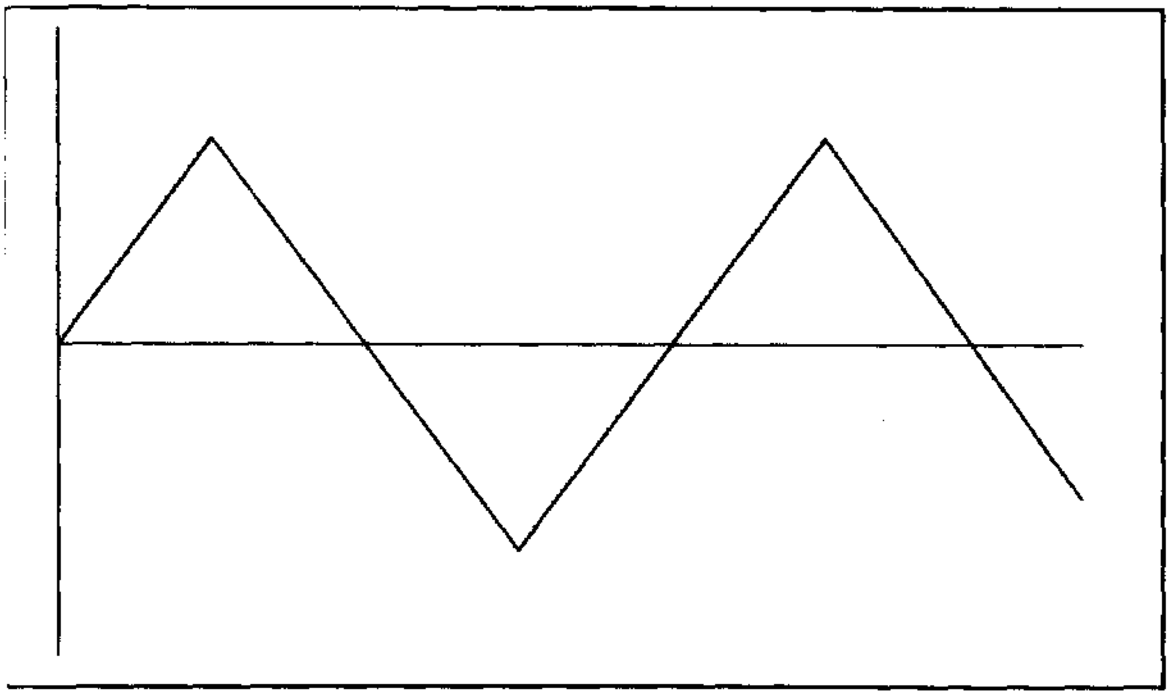


Fig. 8-1. Triangular waveform.

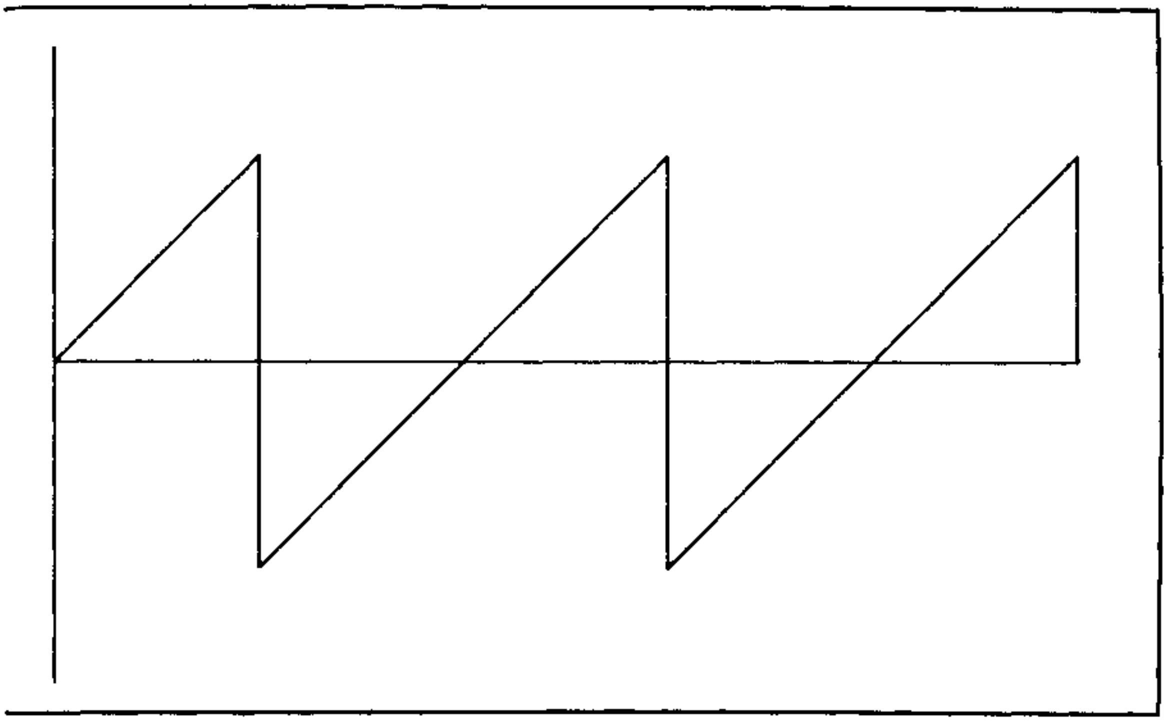


Fig. 8-2. Sawtooth waveform.

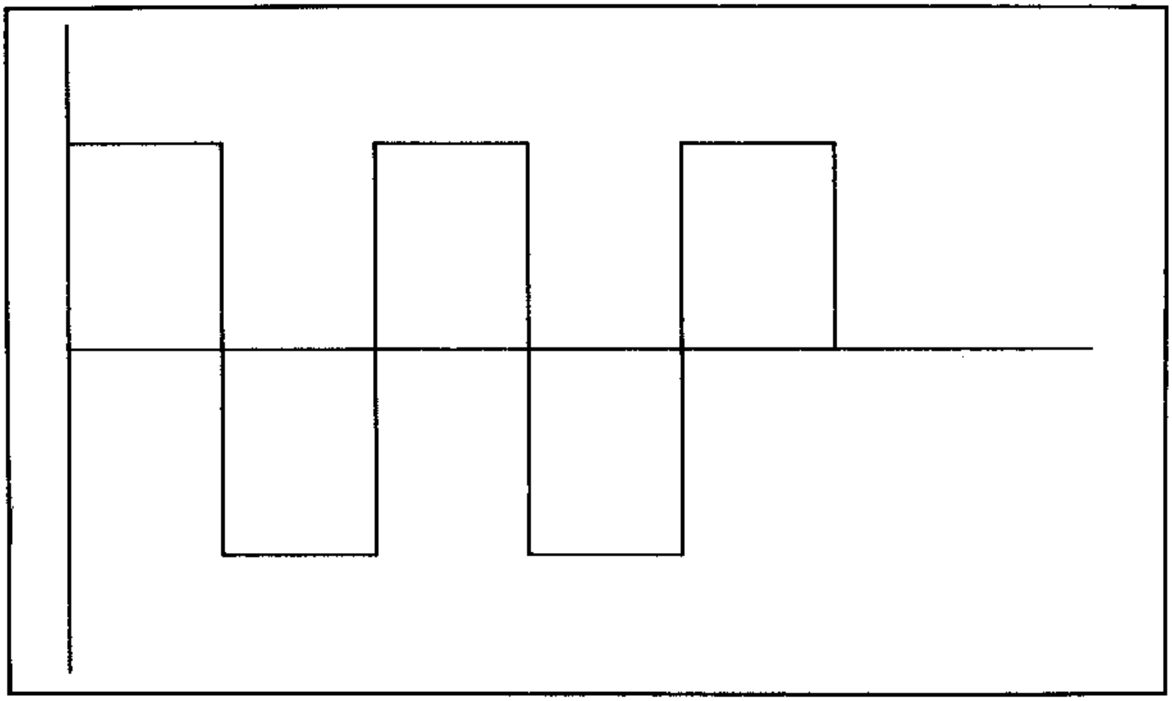


Fig. 8-3. Square waveform.

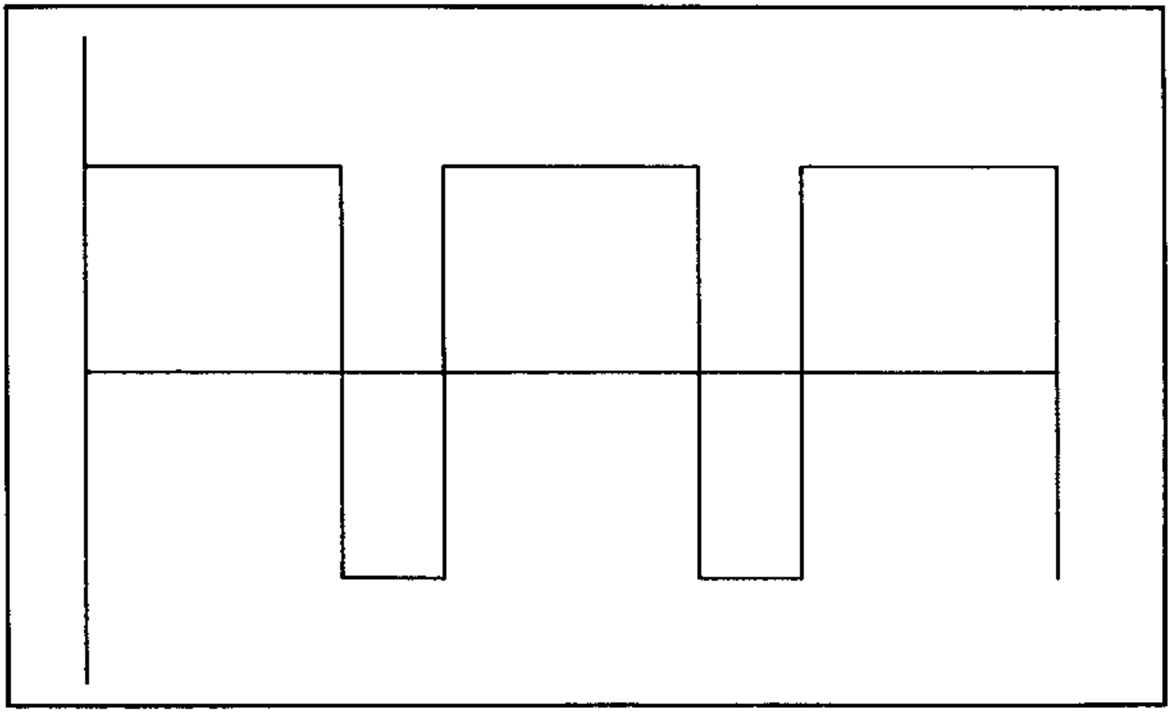


Fig. 8-4. Rectangular waveform.

it drastically by having the sound pass through one or more types of filters. An audio filter changes the sound by cutting down the volume of certain frequencies. Different types of filters will modify a sound in different ways. Each channel can be individually selected as to whether or not it will be passed through the filter.

The ability to route the audio outputs through one or more filters is a very powerful feature of the Commodore 64. Unlike most computers, which allow you to generate only simple tones, the filtering modes of the SID chip allow you to generate complex tones by modifying the harmonic content of the tones. The filters accomplish this task through a technique known as subtractive synthesis. By using an input source that is high in harmonics, the filter can selectively eliminate specific frequencies. Depending on the filtering mode, the same initial tone can be used to create many different sounds. In addition to using the filtering modes in a static fashion (setting them and leaving them), you can control the filter settings in real time. By doing so, you will be able to create sounds such as wind storms and jet engines. These are sounds that cannot normally be generated well by a home computer.

There are three types of filters in the SID chip, a low pass filter, a band pass filter, and a high pass filter. More than one type of filter can be selected at one time. When multiple filters are selected, the effects are additive. A notch filter can be created by selecting both the low pass and high pass filters. The filters will affect the sound in the following ways.

**Low pass filter.** When the low pass filter is selected, all frequencies above the cutoff frequency are attenuated at the rate of 12 dB/Octave. This filtering mode will generate a full sound.

**Band pass filter.** The bandpass filter will attenuate all of the frequencies above and below the cutoff frequency at the rate of 6 dB/Octave. A bandpass filter produces thin sounds.

**High pass filter.** All of the frequencies below the cutoff frequency will be attenuated at the rate of 12 dB/Octave when the highpass filter is selected. Tinny sounds can be generated when using this mode. Figures 8-5 through 8-7 show graphics fre-

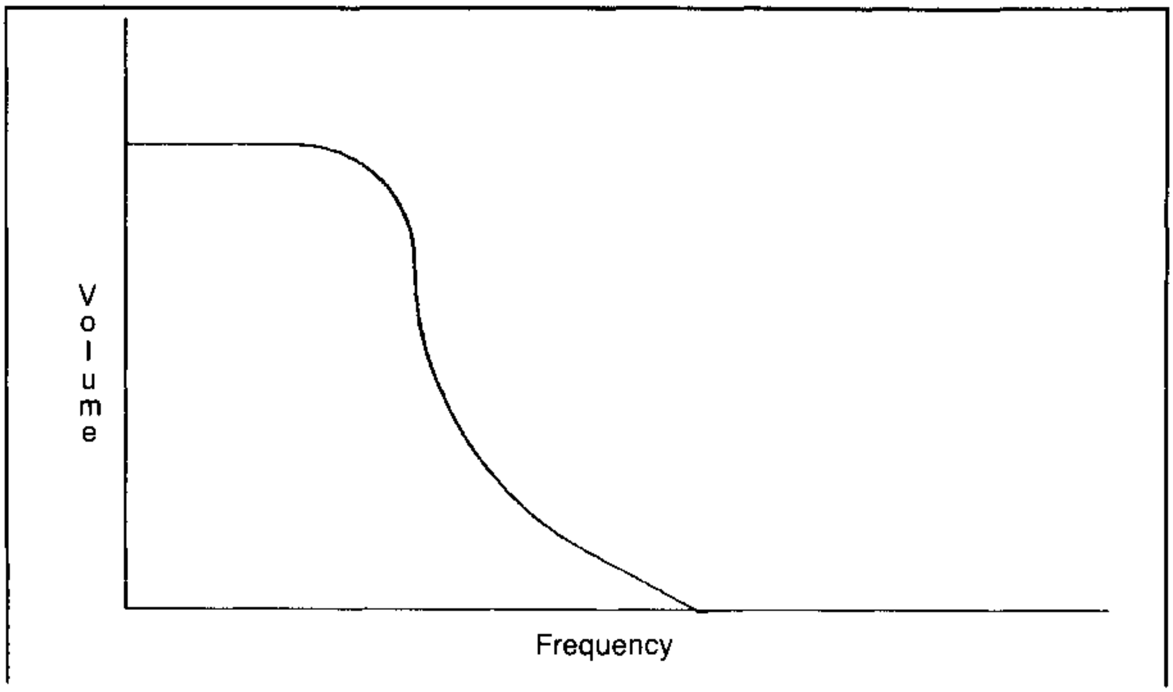
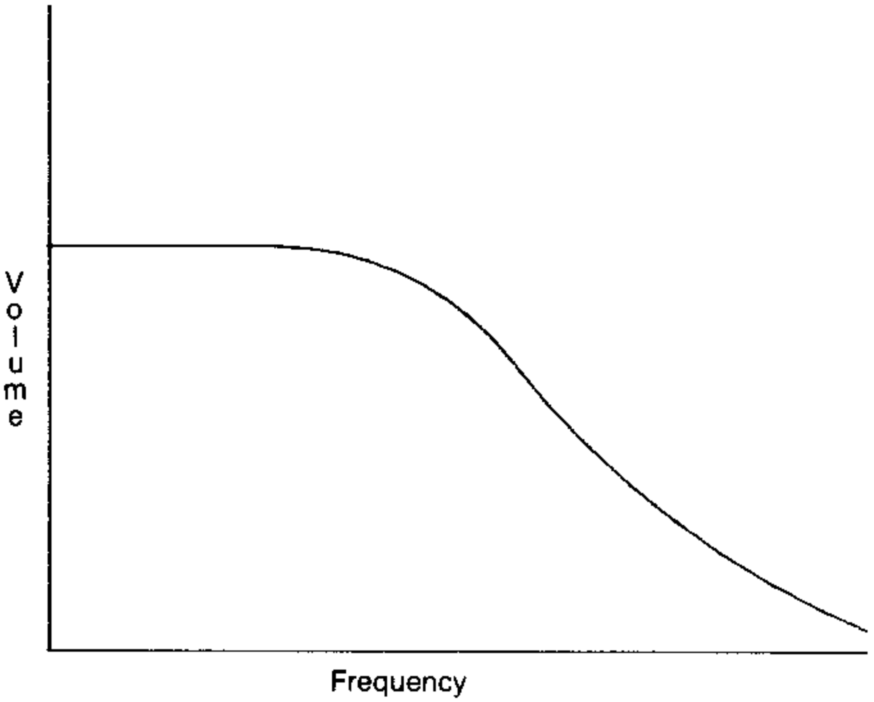
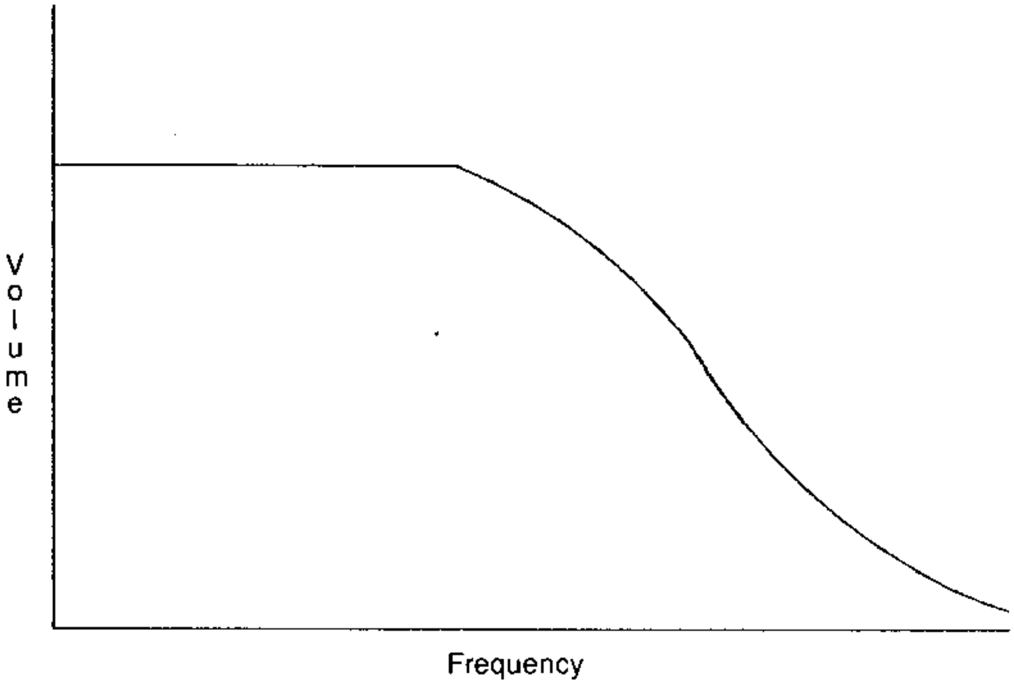


Fig. 8-5. The effects of changing the filter cutoff frequency for a lo-pass filter (continued on p. 40).


Volume
Frequency


Volume
Frequency

uency amplitude as the cutoff frequency of the :iters are changed.

## THE SOUND GENERATOR DEMO

Included in Appendix C is a program that shows many of the abilities of the SID chip. Listing C-5 the source code; listing C-6 is the assembled code. This program loads three machine language files

and DATA in Listing C-9. By listening to this demo, ou will begin to hear some of the possible sounds hat can be created on the Commodore 64. As the :demo is running, it will show you on the screen what

longer attack and decay times will take a while to ±demonstrate, so please be patient. To run the demo, }pe in the following after checking to make sure -he volume on your monitor is turned up:

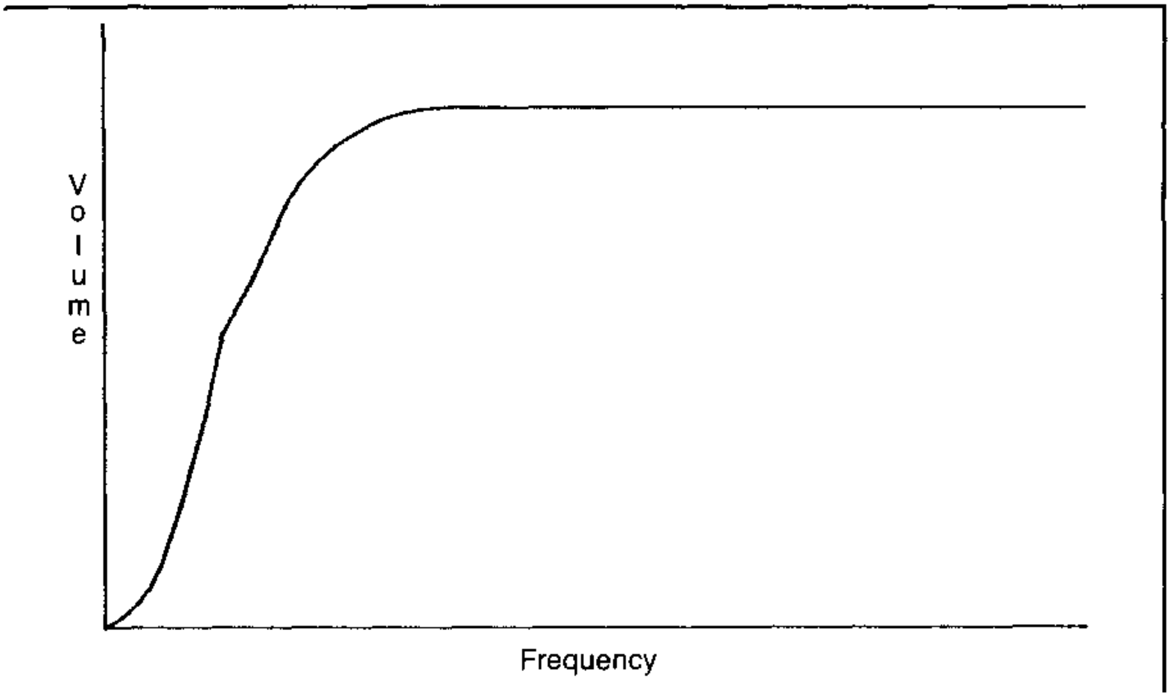
```
LOAD"SOUND DEMO",8,1
SYS 4096
```

Me sounds and effects in the program are by no means all of the sound effects that can be created by the SID chip. Depending on how sophisticated you care to become, you will be able to get many effects that are not directly obvious. For instance, if you write a program that changes the volume of one of the channels in real time, you will have full control over generating different types of tremelo effects.

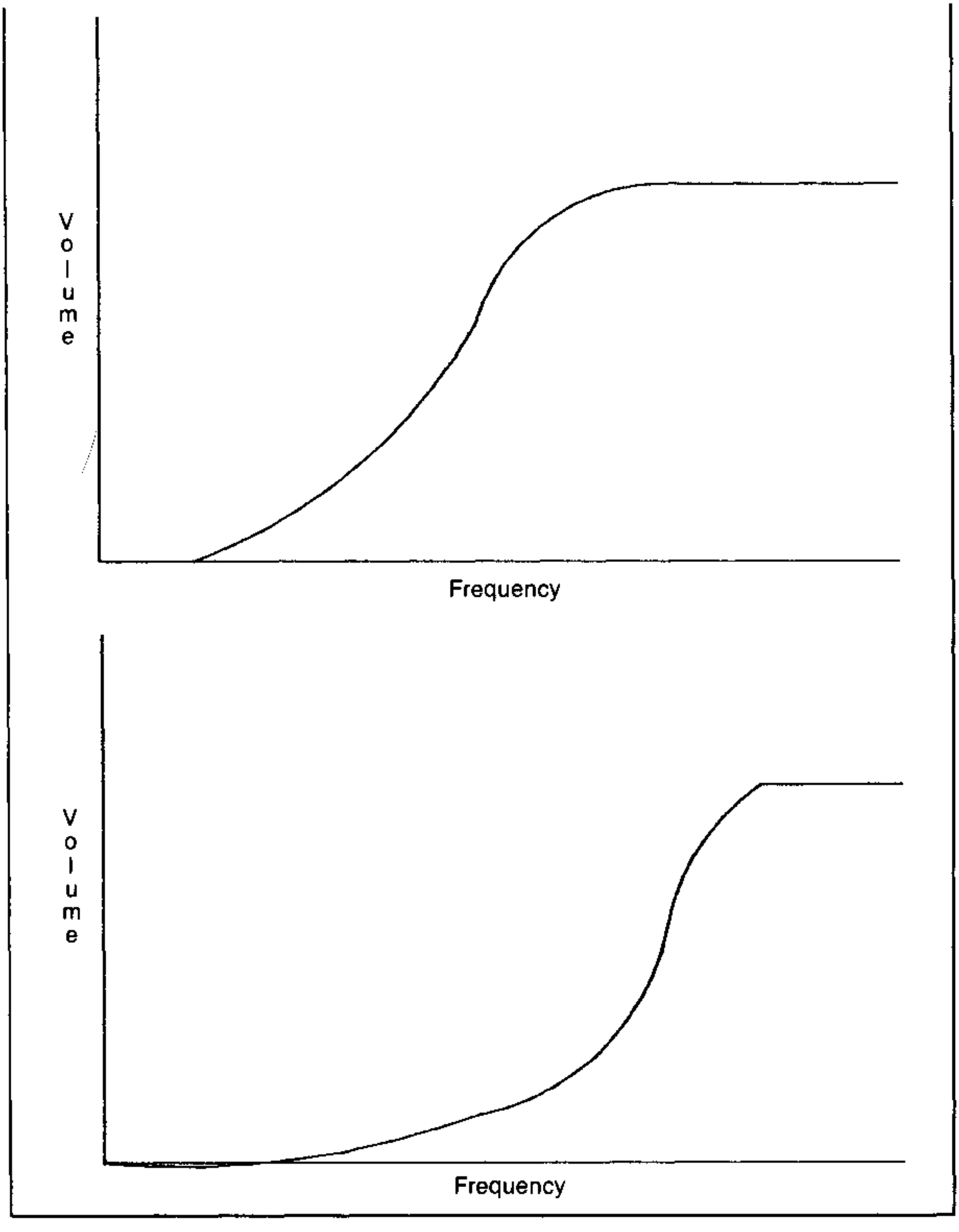
There are program segments in the demo that can be used directly or expanded to help generate almost any type of sound or tune. The routine that generates the short tune that is played to show the effects of different waveforms simply reads a list of notes and note times. If the value of a note is $00, then the routine will quit. By changing the note values in the note table and the time values in the time table, this routine will play any type of tune.

## THE SOUND EDITOR

In order to aid you in choosing values for the different registers, in the SID chip, a sound editing program has been included in Appendix C. Listing



:g 8-6. The effects of changing the filter cutoff frequency for a hi-pass filter (continued on p. 42).


Volume
Frequency
Volume
Frequency

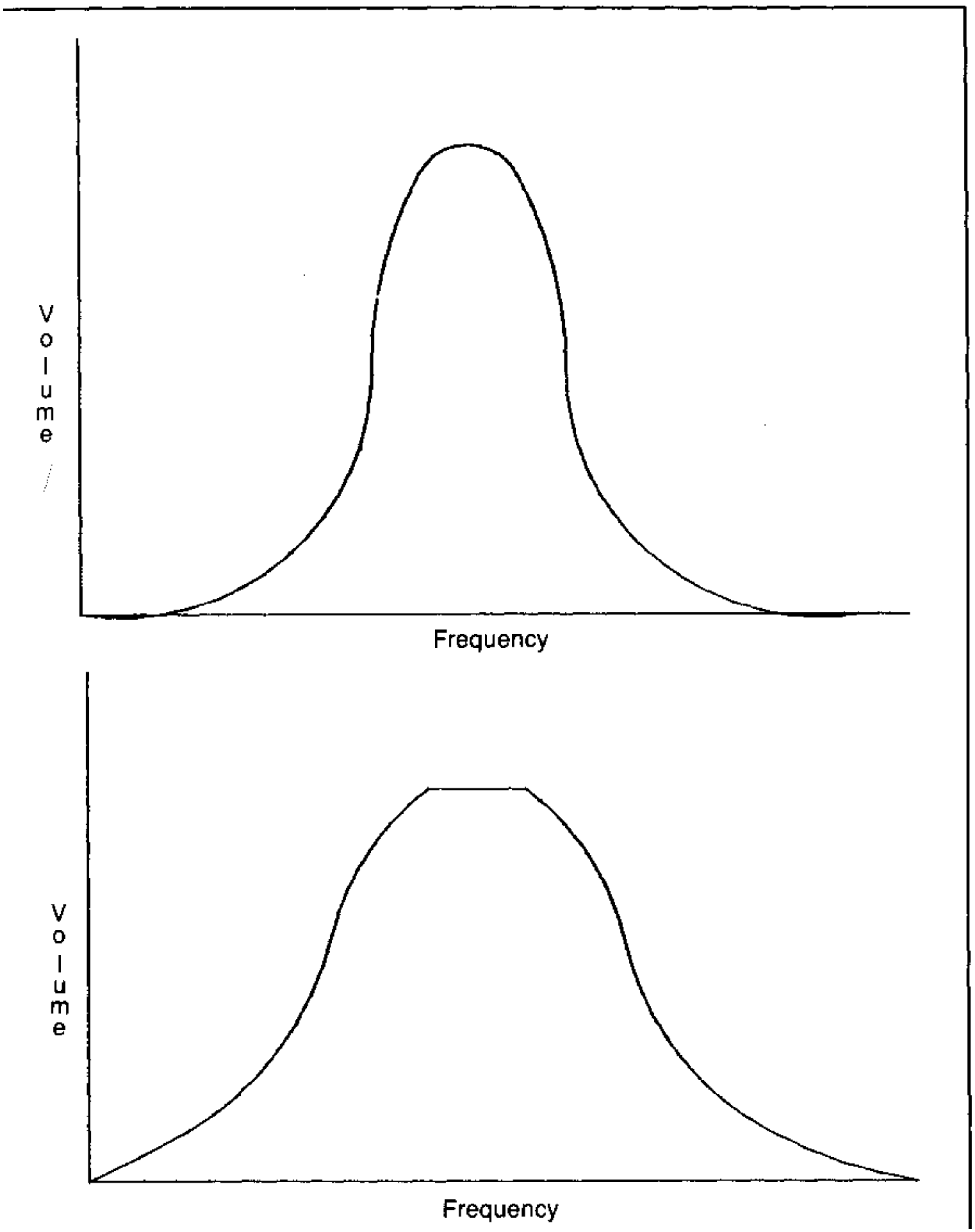
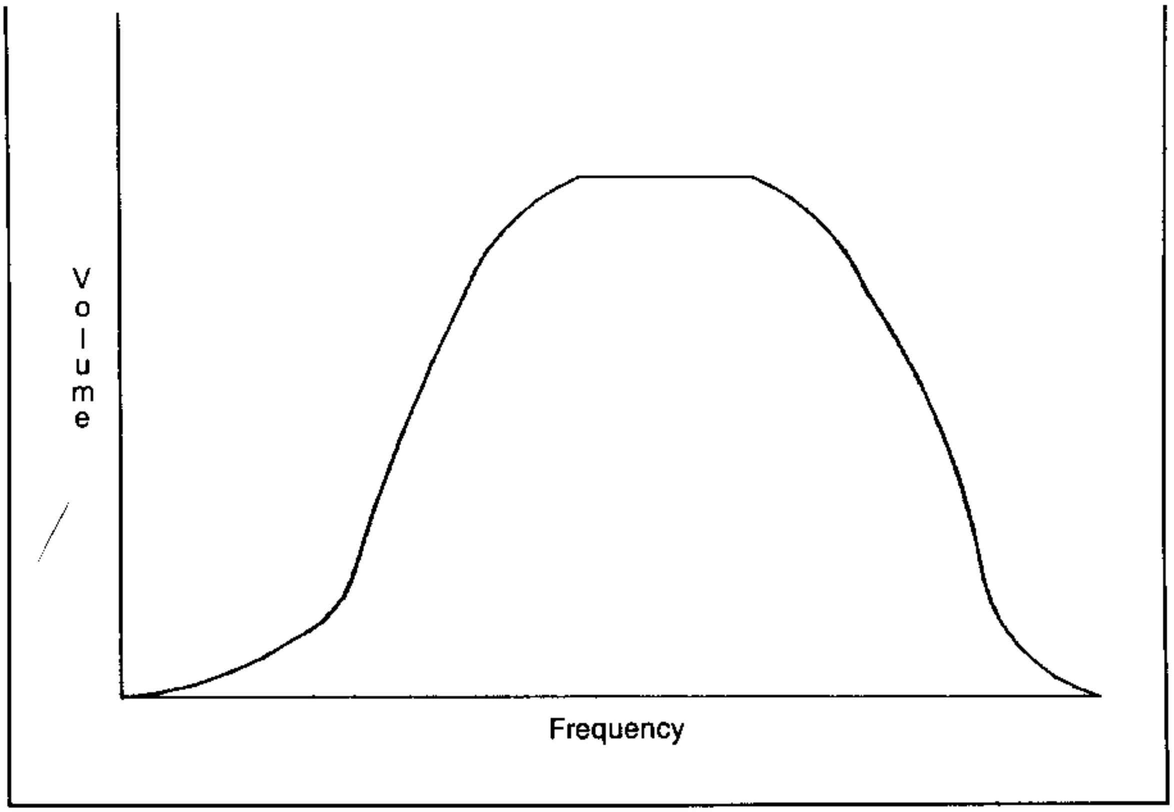


Fig 8-7. The effects of changing the filter cutoff for a band- pass filter (continued on p. 44).



C-10 is the source coed; Listing C-11 is the assembled code. The Sound Editor shows all the SID chips' registers on the Screen using the naming conventions discussed in Chapter 3. A cursor can be moved around the screen, allowing you to change the values that will be loaded into any of the SID registers. All of the numbers are typed onto the screen in hexadecimal notation, so the data that you create can be easily entered into your assembler. After all of the data fields that you care to change have been updated, you can instruct the Sound Editor to transfer the data to the SID chip. If you have enabled one of the channels, this should produce a sound. To run the Sound Editor, type:

```
LOAD"SOUND EDIT", 8, 1
SYS 4096
```

The controls for the Sound Editor are shown in Table 8-1.

**Table 8-1. The Controls for the Sound Editor Program.**

| | |
|---|---|
| CURSOR DOWN | Moves the cursor down one field |
| CURSOR UP | Moves the cursor up one field |
| CURSOR RIGHT | Moves the cursor one field to the right |
| CURSOR LEFT | Moves the cursor one field to the left |
| 0-9 | Allowable numbers for the data fields |
| A-F | Allowable letters for the data fields |
| FI | Transfers the data from the screen to the SID chip and the software timer. |

A brief description of the registers used by the Sound Editor follows. For a more detailed description of each register, refer to the section on the SID chip. A listing of hexadecimal values for 6+ octaves worth of notes is provided in the COMMON file, :.fisting C-7 in Appendix C. In the following description, wherever all three voices have identical registers, voice #1 is used as an example.

| | |
|---|---|
| V1ATDC | The high nibble controls the attack time for this channel; the low nibble controls the decay time. |
| V 1SURL | The high nibble controls the sustain level for the channel; the low nibble controls the release time. |
| V 1 FRLO | This is the low order 8 bits of a 16 bit value that specifies a frequency for the channel. |
| V1FRHI | This is the high order 8 bits of a 16 bit value that specifies a frequency for the channel. |
| V 1 PWLO | This is the low order 8 bits of a 12 bit value that specifies the pulse width of the channel when you are generating a square wave. |
| V 1PWHI | The lower nibble of this register contains the high order 4 bits of a 12 bit value that specifies a pulse width of the channel when generating a square wave. |
| V 1CORG / | This register is the control register for the sound channel. It controls the type of waveform as well as the synchronization mode. The enable bit for the channel is in this register. |

The following registers affect all three voice channels.

| | |
|---|---|
| F LCNLO | The least significant 3 bits of this register are the low order 3 bits of an 11 bit value that controls the filter frequency. |
| FLCNHI | This register contains the high order 8 bits of an 11 bit value that determines the filter frequency. |
| MODVOL | The filtering mode and the maximum volume for all three voice channels is controlled by this register. |
| RESFLT | This register contains the resonance value and the filter enables for all three channels |

The last two registers used by the Sound Editor are not SID registers but RAM locations. These -egisters are treated as a 16 bit value that is used as a time counter. They are decremented every 1/60 of a second. When the 16 bit value has been decremented to $0000, the release sequence is initiated for all three channels.

| | |
|---|---|
| SNDTM1 | This is the low order 8 bits of a 16 bit value that determines the time in 1/60 second intervals before the release sequence is initiated. |
| SNDTM1+1 | This is the high order 8 bits of a 16 bit value that determines the time in 1/60 second intervals before the release sequence is initiated. |

By experimenting with different values in the registers, you will very quickly get a feel for what effect different values have on the sound. Being able to specify the amount of time to play the sound can greatly speed up the time it takes to polish a game, as the values for the sounds can be determined separately from the main program.

# Chapter 9
# Creating Graphics

Up until now,this book has been primarily concerned with background information necessary for creating the program that will ultimately become a game. This is certainly an important part of learning to program a game, but by no means all of it. Because a video game is an audio-visual experience, it will be necessary to create the graphics data that will be operated on by the program.

There are a number of methods that can be used to create and enter graphics information into the computer. The method that you choose is purely a matter of personal preference. This chapter will discuss a few of the options available to you when it is time to create graphics data. Also, instructions for using the graphics utility programs in Appendix C are given in this chapter.

## HAND CODING GRAPHICS

At some point, you will probably be entering graphics data into your machine by hand. As this can be a very time consuming process, this section will show you some techniques that may make your job a little easier.

Before you sit down to enter graphics data, you should have a good idea of what you want the final object to look like. You will need to know the graphics mode that you will be working in. Also, you must decide what colors you are going to use. For instance, if you are using multicolored sprites, each sprite can have one color of its own and can use 2 colors that are common to all the other sprites. To ensure that you will have the proper colors available to you, it would wise to have decided how all of the characters should look before you start.

Once you have chosen a graphics mode to work in, you will be able to start drawing your characters. In Figs. 9-1 through 9-7 you will find some sample graphics layout sheets. These sheets have a fine grid that is proportional to the dimensions of a pixel on the TV screen. The heavy grid is proportional to one character cell on the screen. There are 25 lines of 40 characters each on the Commodore 64.

One thing that you should keep in mind when you are creating graphics for use on the Commodore 64 is that it is often necessary to use more than one

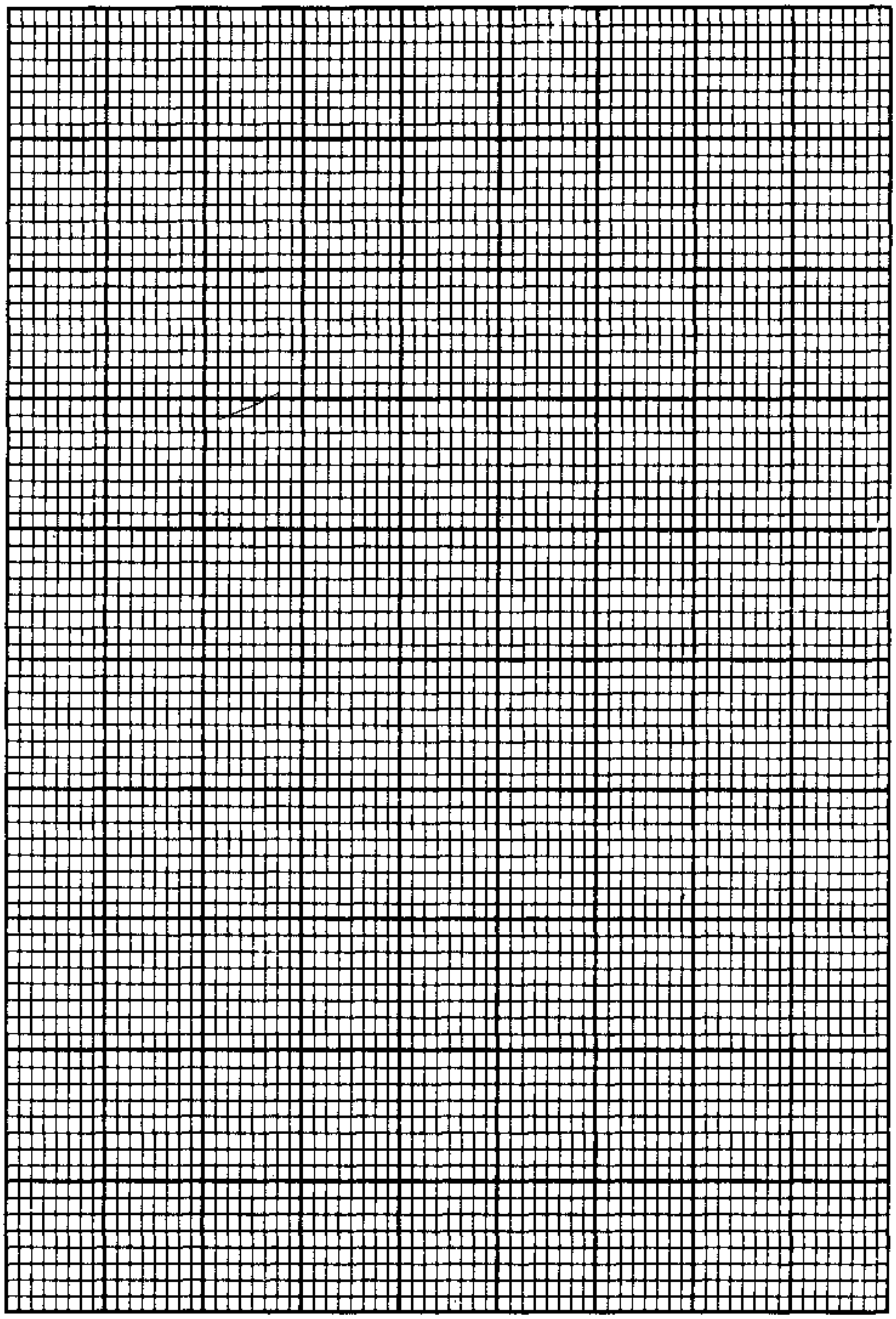


Fig. 9-1. A grid showing character spaces and individual pixels.

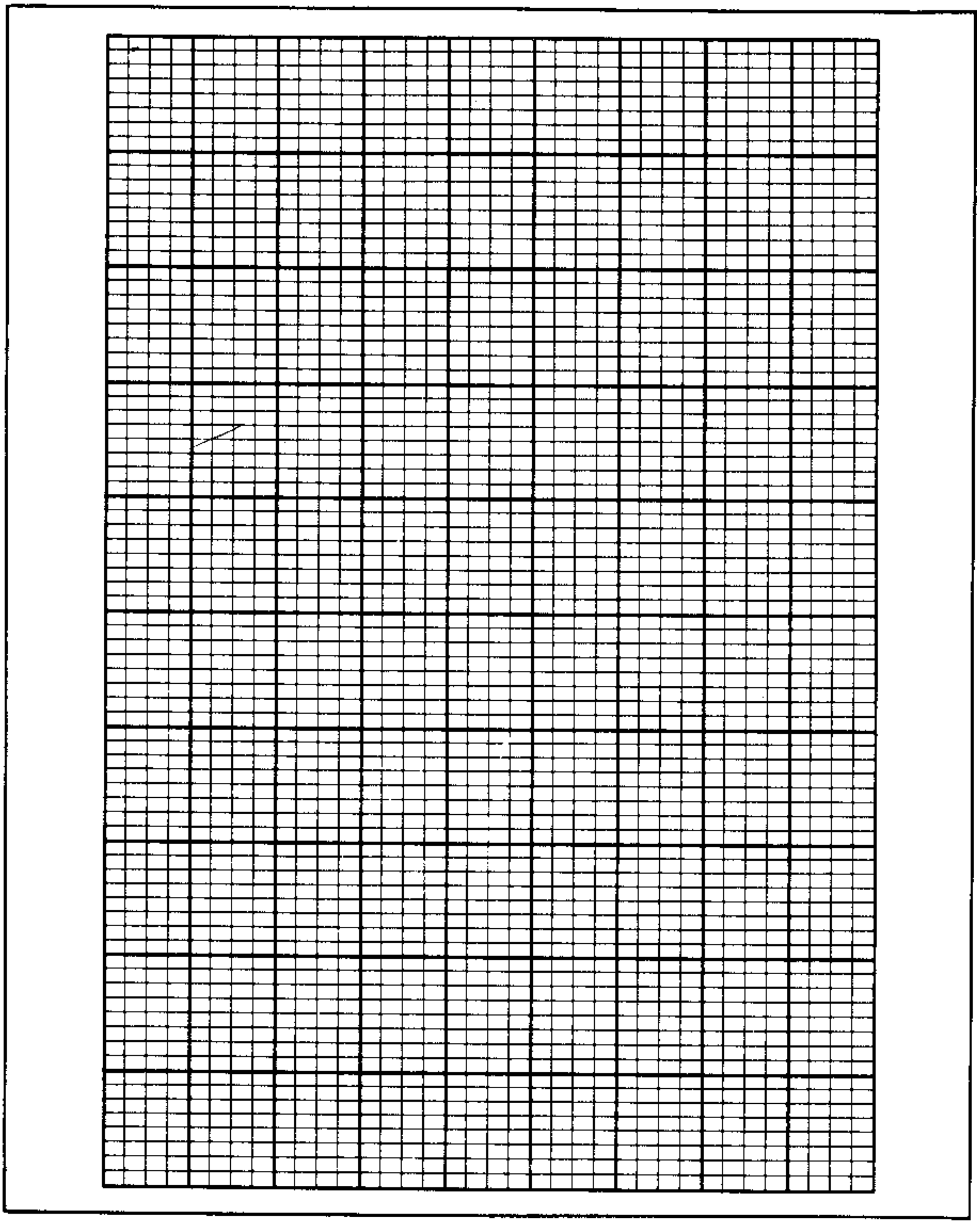


Fig. 9-2. A grid showing character spaces and double-wide dots for multicolor modes.

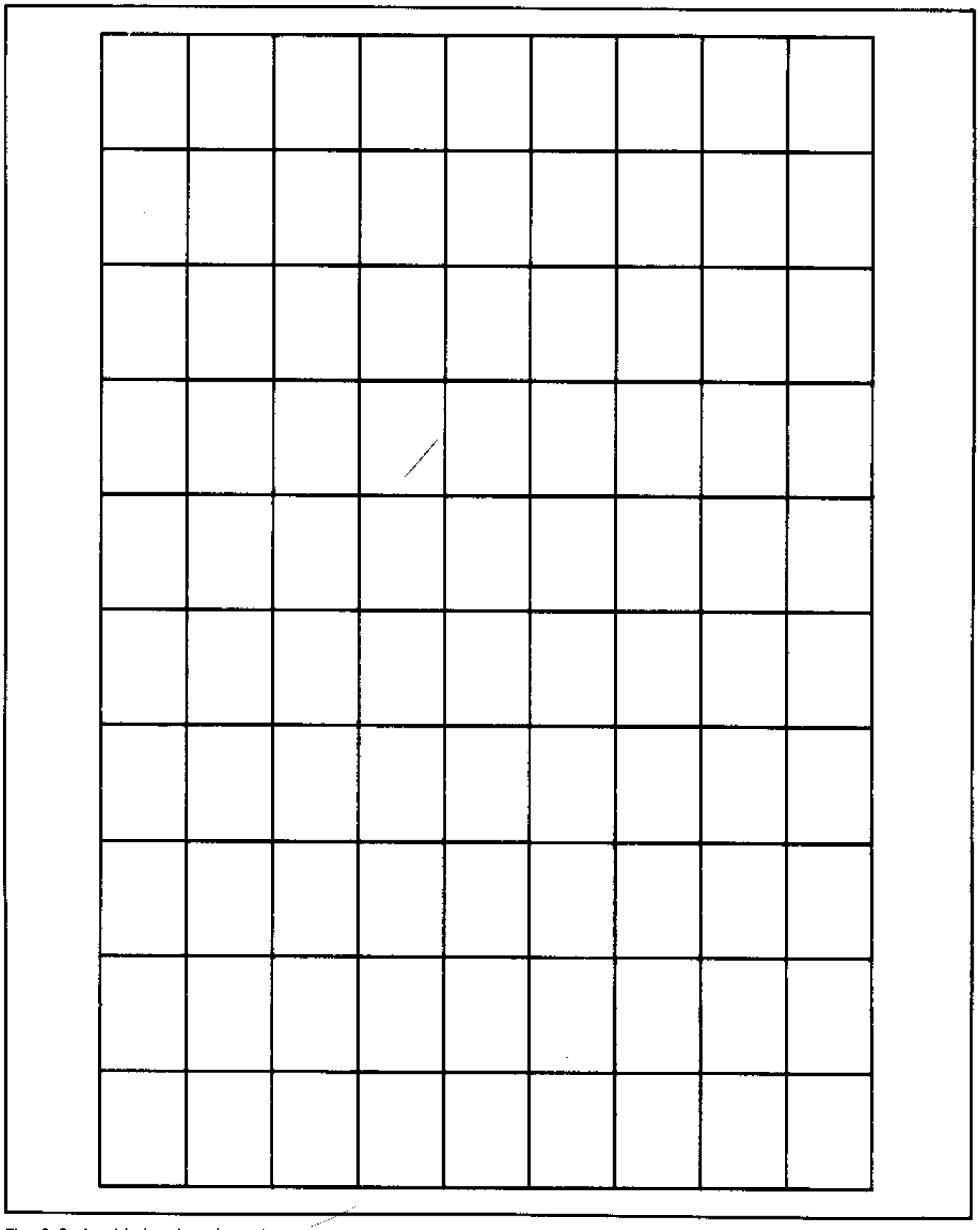


Fig. 9-3. A grid showing character spaces.

pixel of a color on a line to guarantee that the pixel can be seen on the television. Television sets were not designed to be able to display drastic color changes on adjacent pixels. Depending on your color choices, a single pixel in an area of the screen may not be seen. This is due to the time that it takes the TV to turn on and off the appropriate electron guns that brighten a pixel. If the TV does not have enough time to adjust the guns, the pixel will have just started to light when the beams are changed for the next pixel. On the other hand, if you have 2 or more adjacent pixels on a line of the same color, there will be enough time for the beams to be readjusted and the pixels displayed.

The form in Fig. 9-1 is the form to use if you are going to be using the standard bitmapped graphics mode. You will notice t each pixel is rectangular in shape, rather then sq re as might have been expected. Thus, care must taken when attempting to draw geometric patterns on the screen. Since the pixels are rectangular, the normal equations for generating geometric shapes do not hold true. However, if you take into account the 4/3 aspect ratio of the pixels, any shape can be drawn properly.

The form in Fig. 9-2 is to be used if you are going to be using one of the multicolor modes for your graphics. You will notice that each pixel on this form is twice as wide as the pixels on the form 9-1.This is because it takes two bits to represent the four colors available to each block in the multicolor mode, as opposed to the one bit that is needed to determine whether the foreground or background color will be used in the standard color mode. Once again, the pixels are not square, and care must be taken when you are creating shapes.

The form in Fig. 9-3 is generally used to represent the screen in the character graphics mode. It also can be used as an overlay to represent one of the color memory areas in the bitmapped graphics mode.

The forms in Fig. 9-4 through 9-7 can be used as design aids when you are creating sprites. There are four different sizes available. The different sizes correspond to the sprite X and Y multiplier options. If you decide to use one of the e-panded sprites,

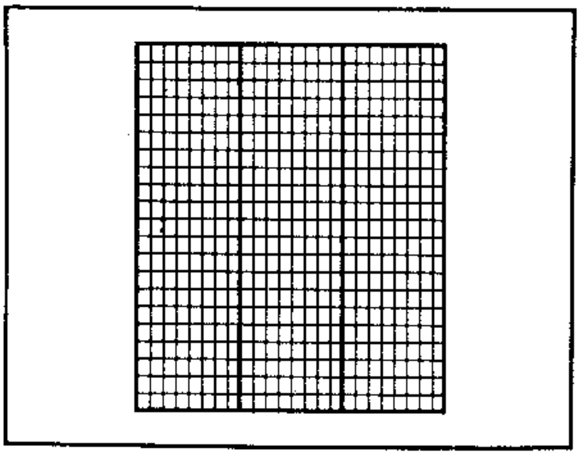


Fig. 9-4. A grid for an unexpanded sprite.

be sure to modify your program to change the SPRXSZ and SPRYSZ registers to the size option that you desire. The size of the grid on the sprite form is identical to the grid on the other graphics forms. This allows the sprites that you create to be placed on top of your background graphics forms to see how the entire screen will look.

Once you have translated your drawing into a series of bytes, you have a couple of options as to what to do with the data. If you have a machine language monitor, you may choose to use the Memory Display option to display the range of memory where you would like your graphics data to go. At this point, your monitor should allow you to change the data on the screen. By using the data from your drawings to modify the data on the screen, you will create a section of memory that represents your graphics. After you have finished entering the data into the monitor, be sure to save the range of memory that you have just modified to the disk. The next step is to enter the address of your graphics into your program so that it can find the graphics later.

Your other option is to enter the data into your assembler using the .BYTE directive. You can then assign names to all of the different areas of your graphics. The assembler can be directed to store the graphics data anywhere in memory. The main disadvantage of entering the data into the assembler is that the data will be reassembled each time a

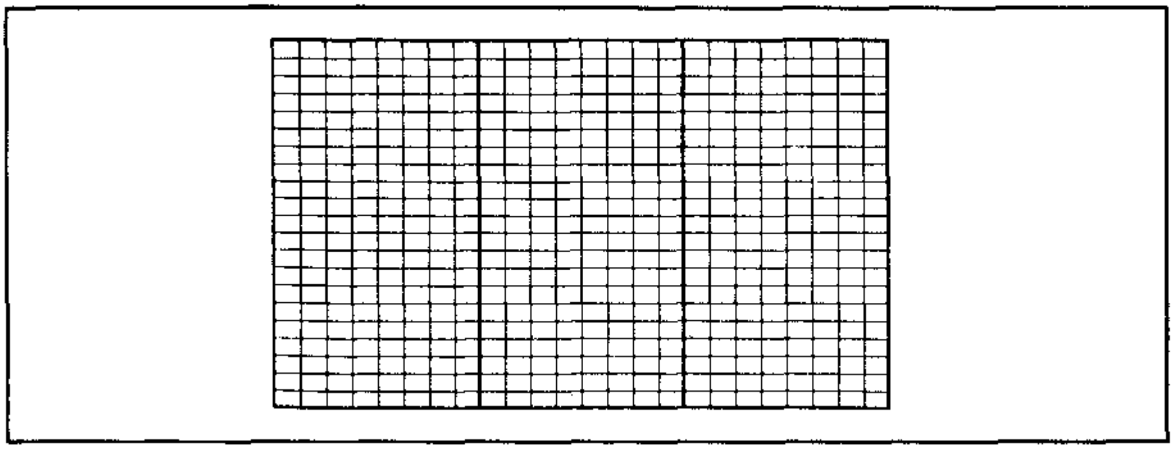


Fig. 9-5. A grid for a horizontally expanded sprite.

change is made in your program. Depending on the amount of graphics data that you have, this can a 1 substantially to the time that it will take to assemble the file. Another option would be to assemble the graphics data separately from the main program. If you do this, you must give the main program the addresses of where the graphics data will be located. In this case, the graphics data need be assembled only once. (Or, at least only as often as is needed to get the data correct.)

## USING A GRAPHICS TABLET

A more popular way in which to enter graphics data into the Commodore 64 for use within the program is to use commercially available graphics packages to generate pictures. One of the most popular and useful of these is the Koala Pad. This package comes with a touch pad and software that allows you to easily generate background screens. Almost any type of graphics package will help speed up the process of generating graphics.

A word of caution: before selecting a graphics package to aid with your drawings, it would be wise to be sure that it will fullfill your needs. There are two major pieces of information that you will need about the package:

- What graphics mode does it use?
- How can the picture be retrieved form the disk?

Different types of games will benefit from the use of one graphics mode over another. If your application requires the use of a certain graphics mode, the graphics package that you choose must use the

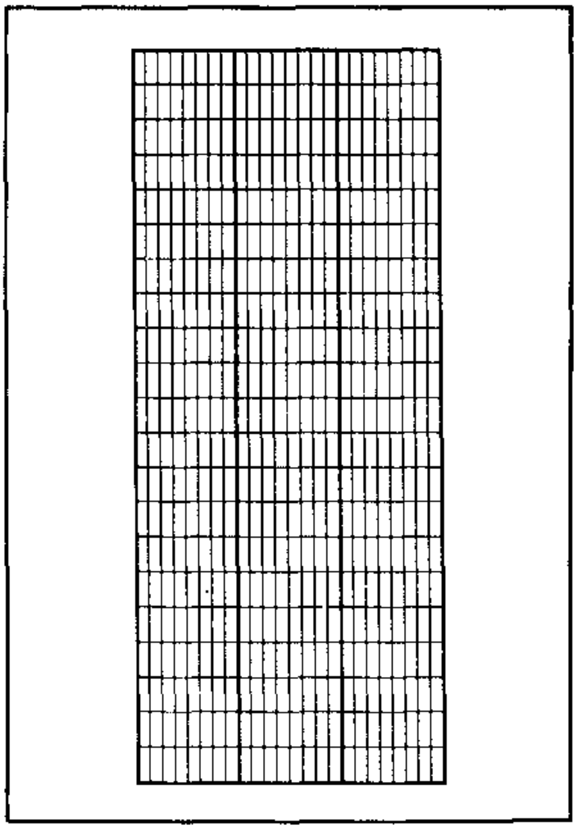


Fig. 9-6. A grid for a vertically expanded sprite.

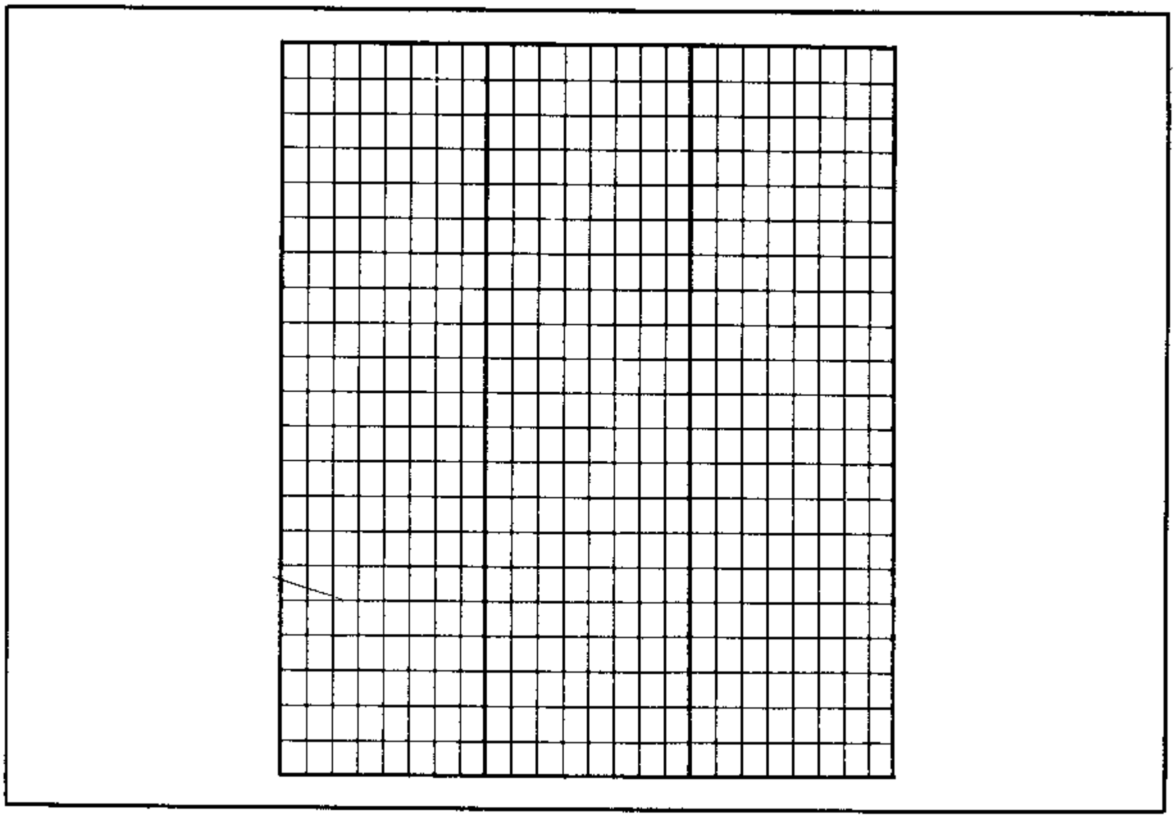


Fig. 9-7. A grid for a sprite that has been expanded both horizontally and vertically.

same mode. Otherwise, the data created will not be useful for your program. This sounds like good advice in theory; however, you will probably find that most of the available packages will use the multicolor bitmapped mode. On the other hand, even if the program can not be used to generate data, it may be useful for testing color choices and to see how things will look.

It is quite possible that you will find a graphics package that will seem to do everything that you would like it to, only to find later on that you can't get your picture back off the disk. It wouldn't seem to be very useful to create a picture if you can't use it. Some graphics packages use what appear to be protected file names to prevent you from retrieving your picture without using the software that created it. You should contact the manufacturer of the graphics package if there is no documentation on how to load the picture wit out their software. If you can get no satisfactory information from the manufacturer, or they claim that you have no right to know, you should not buy their software package.

When shopping for a graphics package, one especially useful feature is a zoom mode. Using a zoom mode, you will normally be able to change individual pixels on the screen with a minimum of effort. This will allow you, among other things, to clean up a drawing that was made free-hand or create a shape pixel-by-pixel, which is too intricate to create in any other way.

Be wary of a piece of software that will not allow you to use all 16 of the colors that the Commodore 64 can display or does not use the full resolution of the screen. After all, there is no reason to sacrifice any of the abilities of your machine because of some other programmer's shortcomings. In fact, most good graphics programs will give you a pallette of more than 16 colors by giving you mixtures of the different colors in different patterns. This can give you a choice of many colors and shades of colors that you may not have been aware were possible.

Another feature that you will learn to appreciate greatly is an OOPS command. This usually allows you to erase the last changes that you have made to your drawing. This is a very useful function when you wish to experiment with color changes and other types of changes or additions that you might not be sure you like. If you don't like your change, you simply give the OOPS command and your drawing is returned to the state it was in before the change.

The following section will give some information about the Koala Pad from Koala Technologies. Some of the information has come from the manufacturer and is not in the documentation that comes with the package. This is not by any means the only software package that will aid in game design; it is only being used as an example. A number of manufacturers have recently released light pens with graphics software, which may be useful. There are also digitizing tablets, joystick controlled graphics software, and keyboard controlled graphics software, which might be suitable. To attempt to evaluate all of the different packages is beyond the scope of this book. The preceding information should aid in your evaluation of a product in the stores, and the following information on the Koala Pad should show you what type of information you will need to properly use the package of your choice.

## Using a Koala Pad

The Koala Pad is a touch sensitive tablet with an active area of 4" by 4" and a resolution of 256 by 256 points. The pad plugs into one of the joystick ports on the Commodore 64 and is treated as a pair of game paddles by the software.

**File format.** Before you try to use the data created on the Koala Pad, you will want to convert the name of the file to something more usable. When the software saves a screen to the disk, it precedes your file name with a single byte whose value is $81. This character prints on the screen as an inverted spade and is inaccessible from the keyboard. Koala uses this character as a flag to identify files that it has saved on the disk. This character is followed by the character string PIC, which is followed by a picture letter and a space.

A pair of utility programs that will convert file names to and from the Koala Pad format are in Appendix C. These are:

| | | |
|---|---|---|
| Listing C-12 | KO-COM | Changes the name from Koala format to Commodore format. |
| Listing C-13 | COM-KO | Changes the name from Commodore format to Koala format. |

These are BASIC programs that will prompt you for the current filename and the name that you would like the file to be called. When using COM-KO, the program will insert the special character at the beginning of the name for you.

After you have changed the name of the file into something that can be loaded, you will be able to use your machine language monitor to examine and reconfigure the data. The data is stored on disk in the following format:

Koala Memory Map

| | |
|---|---|
| $6000 - $7F3F | Graphics image |
| $7F40 - $8327 | Color memory image |
| $8328 - $87OF | Color RAM image |
| $8710 | Background color |

Note: If you are using a cartridge based machine language monitor, you may not be able to read the file as the cartridge replaces the RAM where the data will be loaded.

After you have loaded the file, you can relocate the graphics data and the color data to anywhere that is convenient to your program. You should then save your newly created file back to the disk. You will probably want to move the data since the Koala software is more interested in reducing the size of the disk file than in placing the data in a useful location. It would be a good idea to move the color memory areas to the beginning of a page boundary. This will make it easier and faster to manipulate the data later. The background color may be stored wherever is convenient.

The data was created using the multicolored bit-mapped mode, so be sure to set the multicolor mode bit in the VIC chip before displaying the picture. Also, note that the border color is not stored in the file. You must set the border color to the appropriate value before displaying the picture.

DISPLAY PIC, Listing C-14 in Appendix C, will display a Koala Pad picture on the screen. It will load the machine language routine MVIT in Listing C-15. This program follows the steps above to display the picture. It will also set the border color to black. It will display the picture until the shift key is pressed on the keyboard. This program will not display a picture if you change the location of the data in the file. To run this program enter:

```
LOAD"DISPLAY PIC",8<RETURN>
 RUN
```

Enter the name of the picture to be displayed when prompted. The file name must have been previously changed using the KO-COM utility. A picture in the Koala Pad format is in Listing C-16 in Appendix C under the name PIC A CASTLE. This picture can be viewed using the DISPLAY PIC program. This drawing could make a nice background for a game, if you were so inclined.

## USING THE SPRITE MAKER

Listing C-17 in Appendix C is a sprite making utility in BASIC. It loads two machine language routines, SLIB.O in Listing C-18 and CLSP2 in Listing C-19. Using this program, you will be able to quickly create a sprite in either the one color or multicolor mode. You will be able to change any of the colors in the sprite or the background color. After you have finished designing a sprite, you will be able to save it to the disk for use in a program later. As you are drawing your sprite on the screen, you will be plotting squares on a 24 by 21 array of squares on the left side of the screen. In the upper right section of the screen, the sprite is shown in its true size and color, so you will be able to see exactly what the finished sprite will look like. in the upper left corner of the screen, there will be a white box if you are in the plot mode; otherwise you will be in the unplot mode and the corner will be blank.

To run the Sprite Maker program, you must type the following:

```
LOAD"SPRITE MAKER", 8
RUN
```

Enter the name to be used when saving or loading from disk.

The SPRITE MAKER uses a combination of joystick and keyboard controls. All of the control options are shown below.

| | |
|---|---|
| JOYSTICK | Moves the cursor around the zoomed sprite |
| FIRE BUTTON | Plots or unplots a point |
| F1 | Toggles the plotting mode |
| F3 | Enables the multicolor mode |
| F4 | Disables the multicolor mode |
| F5 | Changes the sprite color |
| F6 | Changes the background color |
| F7 | Changes the multicolor 1 register |
| F8 | Changes the multicolor 0 register |
| S | Saves the sprite to the disk |
| L | Loads the sprite from the disk |

The joystick will move the cursor around the screen. When the fire button is pressed, a square will either be plotted or unplotted, depending on the plot mode at the time. Pressing the F1 key toggles the plotting mode. The mode is set to OFF when the program is first run. The F3 key will enable the multicolor mode. Pressing the F4 key disables the multicolor mode. The multicolor mode is off when the program is first run. Pressing F5 will increment the sprite color register. The F6 key increments the background color. Pressing F7 will increment the sprite multicolor 1 register. This will only show an effect when the multicolor mode has been selected. Pressing F8 will increment the sprite multicolor 0 register. This will only show an effect when the multicolor mode has been selected. Pressing the S button will save the sprite to the disk using the name that you had entered earlier. Pressing the L button will load a sprite pattern from the disk with the name that you entered earlier.

After you have finished editing a sprite, the sprite data will be at $4000. By using a machine language monitor, you will be able to save the binary sprite data from $4000 to $403F. If you will be using multiple sprites, you may wish to move the data to a safe area in memory so that you can merge your new sprites with the old ones.

## USING THE SCREEN MAKER UTILITY

If you are going to use a character graphics mode, you will need some way to specify the placement of the character graphics on the screen. The SCREEN MAKER Listing C-20 in Appendix C will aid you in defining a screen of graphics.This program will load three machine language routines, CLBACK1 in Listing C-21, CLSP1 in Listing C-22, and SLIB.O in Listing C-18. This program will allow you to select any of the characters out of your character set, and place it in any position on the screen. This will allow you to see quickly how your screen will look. When you are finished, you will be able to save the screen to the disk. To run the program, type:

```
LOAD"SCREEN-MAKE",8
RUN
```

Enter the filename to be used when loading or saving a file to the disk when prompted. The following controls are used in this program:

| | |
|---|---|
| JOYSTICK | Moves the cursor around the screen |
| FIRE BUTTON | Plots the selected character on the screen |
| F1 | Increments the current character number |
| F3 | Decrements the current character number |
| F5 | Increments the character color |
| F7 | Increments the background color |
| L | Loads a file from the disk |
| S | Saves a file to the disk |

After you run the program, the screen will clear and the first character in the character set will be displayed in the upper left hand corner of the screen. The next character in the upper left hand corner of the screen shows the current character color. If this color is the same as the background color, the space will appear blank.

Pressing the F1 key will select the next character from the character set. You will see the character in the upper left corner of the screen. By repeatedly pressing the Fl button, you can scan through your entire character set. Button F3 will select the previous character from the character set. By using these two buttons, you will be able to move forward or backward through the character set.

The F5 key will increment the character color. This is the color that will be placed into the color RAM when the character number is placed in the screen RAM. Pressing the F7 key will increment the background color. This is useful when you wish to see what the screen would look like with different background colors.

You may load a screen to be edited using the L command. When you are finished making changes, you should use the S command to save the screen back to the disk. When a screen is saved to the disk, the color information is saved to the disk along with the character placement information.

# Chapter 10
# Some Arcade Games

Before attempting to design your own video game, it would probably be helpful to understand how some other games are designed. In this chapter, you will be shown some of the ways in which some popular arcade video games could be programmed into the Commodore 64. In fact, the description of how to program these games may be quite accurate in terms of how the original was done. Bear in mind, however, that arcade machines tend to have some specialized hardware for graphics creation.

## PAC-MAN

The most popular arcade game in recent times is PAC-MAN. This is a maze type game in which the player controls PAC-MAN in his journey around the maze while being chased by computer controlled ghosts. If PAC-MAN is hit by one of the ghosts, he loses a life. On the other hand, if PAC-MAN manages to eat one of the power pellets on the playfield, for a short amount of time the ghosts will turn blue. During this period of time, PAC-MAN may eat his enemies for a greater score.

Difficulty levels are created by changing the speed of all of the characters and changing the amount of time that PAC-MAN has in which to eat the ghosts. PAC-MAN must eat all of the pellets on the playfield in order to advance to the next difficulty level. 'Nice during each level of play a bonus character appears on the screen for a short period of time. If PAC-MAN can eat the bonus character, he gets bonus points. The bonus character is worth more points on the higher levels.

If you were to program PAC-MAN on the Commodore 64, you would probably use the multicolor character graphics mode for the maze and the pellets. PAC-MAN and the four ghosts would be sprites. The bonus character could also be a sprite. At this point, you would have two sprites to spare because the Commodore 64 allows eight sprites, and only six have been allocated. These remaining two sprites could be used as the four power pallets by repositioning the sprites after the top 2 pellets have been displayed. This technique will be discussed in more detail in a later chapter.

Since the Commodore 64 maintains collision registers to determine collisions between sprites and between sprites and background, determining when PAC-MAN hits a ghost or power pellet should be no problem. The only part of the program that may be difficult is the part where it is determined which of the normal pellets PAC-MAN has eaten. For the most part, putting PAC-MAN on the Commodore 64 would be a very direct translation. On the arcade game, the video monitor is rotated 90 degrees, which makes the maze taller than it is wide. This would be the only major discrepency between the arcade machine and a Commodore 64 translation.

**DONKEY-KONG**

In DONKEY-KONG, the player controls Mario, who is trying to rescue a girl from the clutches of DONKEY-KONG, a large ape. To do this, Mario must climb the various structures that DON j E'S-KONG sits on. To make Mario's life more difficult, DONKEY-KONG keeps throwing barrels and hammers down at Mario. On some screens, there are fireballs that he must dodge. As Mario climbs toward DONKEY-KONG, he can pick up articles of the girl's clothing that she has dropped on her way up. Mario can also pick up a hammer and proceed to beat up the barrels and fireballs for a short amount of time. Mario can jump over the barrels and fireballs, for which he gains points. If Mario can get up to the level where DONKEY-KONG is standing, he either rescues the girl, or depending on the level, DONKEY-KONG carries the girl higher up on the structure. Mario is then presented with the next level of play. The arcade version of the game has four different types of structures that must be climbed.

Translating DONKEY-KONG to the Commodore 64 is very similar to translating PAC-MAN. The background structures, DONKEY-KONG, and the girl can all be made up of character graphics. The articles of clothing that the girl has dropped can also be made up of character graphics. This leaves all of the sprites free for Mario, the barrels and the fireballs.

If you watch an arcade version of DONKEY-KONG very carefully, you may see how its designers avoided the problem of needing a large number of sprites. When a barrel rolls past Mario and heads for the next lower level, it will normally roll off the edge of the screen rather than descending to the next level. Because the player's eyes are normally focused on Mario and the portions of the screen above him, he will not normally notice the disappearing barrels. Because the machine does this, it never needs more than five sprites to display all of the barrels. This same technique will work for a translation of DONKEY-KONG for the Commodore 64. If five sprites were reserved for fireballs and barrels and one for Mario, there would still be two left over for hammers. In fact, you would have even more flexibility in the use of sprites. For the most part there are only 3 or 4 sprites displayed on any given line. Using the technique of repositioning sprites, you could reposition the sprites on different lines.

Like PAC-MAN, in the arcade, DONKEY KONG's screen is rotated 90 degrees from a normal television. For this reason, any translation of DONKEY-KONG to the Commodore 64 will be wider and shorter than the original.

**CENTIPEDE**

In CENTIPEDE, a nasty centipede is running loose in a field of mushrooms. It starts at the top of the screen and winds its way down the screen until it gets to the bottom, where the player's gun is. The players must shoot the centipede without getting hit by it. The centipede has 11 body segments. If the head is hit, the next body segment becomes the new head. If a body segment is hit, the centipede breaks into two parts, each of which has its own head. Whenever a segment of the centipede hits a mushroom, it drops down to the next line and turns around.

In addition to the centipede, the player must also avoid the spiders and fleas. Fleas add to the mushroom field, while spiders destroy mushrooms. Scorpions poison the mushrooms that they touch. A poisoned mushroom causes the centipede to descend straight down the screen.

This game could be a bit of a problem to translate because of the large number of moving objects. The mushrooms can be made using multicolor programmable characters. Since the Commodore 64 has only eight sprites to work with, there would appear to be a shortage of sprites to use in the translation. However, by using the technique of sprite multiplexing, you can trick the computer into working as if it had 16 sprites. This technique is discussed further in a later chapter; if used properly, it would allow the game to be translated for the Commodore 64. Eleven sprites are used for the centipede's segments, 1 sprite for the player, 1 sprite for the shot, 1 for the flea, 1 for the spider, and 1 for the scorpion. This totals 16 sprites, or the number of sprites that multiplexing would provide.

## THE REVENGE OF THE PHOENIX

To help illustrate how some of the different techniques describe in this book translate into a game, an arcade style game, Revenge of the Phoenix, has been included in Listing C-23 in Appendix C. This game uses almost all of the techniques that have been covered earlier. It is included to help illustrate the capabilities of the Commodore 64. To play the game, type the following:

```
LOAD "PHOENIX V1.4N", 8,1
 SYS 32768
```

At this point, the program will go into its introduction mode. If left alone, it will demonstrate how the game plays and eventually return to the introduction. You may interrupt this process at any time by pressing one of the fire buttons. The game can be played by either one or two players. You can choose which mode you want to play by moving the arrow with the joystick on the title page. By pointing at the character that represents the mode that you would like to play and pressing the fire button, you will initiate game play in that mode. Since the two players (high wizard and low wizard) do not have exactly the same capabilities, you may choose which of the two players you would like to control.

### Game Play

In Return of the Phoenix, you will be playing the character of a wizard protecting a castle from the magical phoenix. You are able to stun the phoenix with spells shot from your staff. The goal is to prevent the phoenix from building a bridge in the sky. They will try to get to the bottom of the screen, pick up an energy spell, and bring it to the top of the screen, where a section will be added to the bridge. Game play will continue until the three tier bridge is completed. At this point, both you and the phoenix will have a score. If your score is higher than theirs, you win.

One of the wizards can fly around the screen and go virtually anywhere that a phoenix can. The other wizard must stay near the bottom of the screen. This gives each of the wizards unique playing characteristics. Which one that you care to play is very much a matter of personal preference.

There are nine levels of difficulty in Return of the Phoenix. The skill level that you are currently playing is shown at the top of the screen in the center. As you are playing, the program constantly monitors your playing ability and modifies the skill level accordingly. If the program feels that you are playing very well, it will increase the level of difficulty. On the other hand, if you are playing poorly, the skill level will be decreased. The skill level can only be decreased if you are above level 4. Starting at level 2, the phoenix will start dropping sleep spells on the wizards. If you are hit, you will be unable to move for about three seconds. At the higher levels, the phoenix will shoot more often and move faster and at some levels, the wizards will move faster also.

### Scoring

In this game, you are competing against the phoenix for the high score. The phoenix are controlled by the computer, and they have a different method of scoring then do the wizards. The number of points that the phoenix gets depends on how long you can prevent them from getting energy bricks to the bridge. After every four seconds, the value of the bricks to the phoenix decreases. The bricks can be worth from 5 to 98 points depending on how long you can keep the phoenix from getting a brick to the bridge. All of the rest of the scoring is more standard and shown in Table 10-1.

When the two player mode is selected, both players are working for one score. The players' score is in the upper left corner of the screen. Unlike most video games, in which the two players are competing, both players are working together for a common goal. The phoenix score is in the upper right corner of the screen. This is just as valid a score as the players' score, in that they are working toward their own objectives. If the wizards beat the phoenix score, the bridge will flash and fall down at the end of the game. Similarly, if the phoenix beat the wizards score, they will display their message at the end of the game.

**Table 10-1. The Scoring System for the Revenge of the Phoenix Game.**

| | |
|---|---|
| **Wizard shooting phoenix with energy brick** | **98 points** |
| **Wizard shooting phoenix without brick** | **26 points** |
| **Wizard shooting all 4 phoenix** | **1000 points** |
| **Phoenix putting a wizard to sleep** | **325 points** |

# Chapter 11

# Elements of Game Design

Much of this book has been dedicated to the techniques of programming a video game on the Commodore 64. In this chapter, some of the concepts that need to be used during the design of the game will be discussed.

Be fun
Have an interesting plot
Be visually stimulating
Have sound effects and music
Have varying difficulty levels
Keep score

When you are designing a game, you are writing a program that is intended to be used for the amusement of others. Trying to make the game fun to play should be your primary consideration. There are some problems in trying to design an enjoyable game, however. For instance, when you come up with a game concept, you may think your game will be the best game ever written, only to find, after you have written the program, that it is boring. This can be caused by a number of factors. More often than not, the game idea was good, but the computer lacked the ability to display a game as complex as you wanted.

One thing to beware of when designing a game, is the tendency to have the game play in exactly the same way each time it is played. If there are no random elements in the game, each move by the player will cause a specific move by the computer. This may be challenging at first, but will quickly become boring once it is mastered. The PAC-MAN arcade machine had this problem and it didn't take long before patterns that showed how to beat the machine every time were published. All it takes is an occasional random move for the play to be unpredictable—which will add to the challenge of the game.

After you have spent some time programming the Commodore 64, you will have a good idea of what it is capable of doing. If you take into account the capabilities of the computer during the design phase of your program, you will have a much easier time writing the program. Many of the routines explained in this book will enable you to write more complex games than you may have thought possible. Techniques such as sprite multiplexing make it easy to display more sprites in the same area of the screen than is otherwise possible. This will give you more flexibility in the design of your game, which will make it more fun.

Your game can usually be made more interesting if you discuss some of your ideas with others before you start programming. Because everybody sees things differently, you may be given some ideas that will greatly enhance the play of the game. Many of the large game corporations put a number of game designers in a room and have them toss ideas back and forth. A bull session such as this can be the fastest way to get creative input into the design of a video game.

**VISUAL IMPACT**

The visual impact of the game is the first thing anyone playing your game will notice. If the animation is interesting, it will quickly attract attention. The proper use of color is important. When you are using a normal television as a monitor, certain colors interact with each other better than others. Black characters on a white background will give quite a bit of contrast. The characters will be very clear and sharp. On the other hand, red characters on a blue background will appear fuzzy and indistinct.

Varying the animation sequences that are used to make up a moving figure can add to the attraction of the game, if they are changed at the appropriate time. A game in which a character explodes and fades out after it is shot will be a more interesting game than one in which the character simply disappears. Similarly, figures can be changed to indicate that something is being carried, or that the player is at a different level. At times, simply flashing the colors of a character will add to the visual effect of the game.

**SOUND EFFECTS**

Sound effects and music can add greatly to the appeal of a game. If nothing else, music played while the title page is displayed will hold a player's attention as they are reading the credits, rules, or whatever else you may choose to put on a title page. In some cases, background music may be appropriate to a game. When this is done, it should be played at a relatively low volume with respect to the rest of the sound effects. Loud background music can be quite a distraction and a nuisance when it drowns out the sound effects. You may want to consider creating an option that will allow the game to be played without any background music.

Many games provide a brief break as the player moves from one level of play to another. Quite often, during this pause, a short piece of animation is shown on the screen along with some music. If a game takes a long time to play, these breaks give the player a time to relax and catch his breath.

**DIFFICULTY LEVELS**

Virtually every game has different difficulty levels. The variations can range from simply increasing he speed of some of the characters at different sco es to starting a completely different portion of t e game after a task is completed. For the most part, a game will be the same each time that it is played. Changing the level of difficulty based on the skill of the player will keep the player interested in playing the game even after he has mastered the beginning levels. Presenting him with a new challenge as a reward for gaining skill in the game will help to keep the player interested. You can make the game more fun to play by increasing the speed of the characters at higher levels and increasing the amount of shooting that the player is allowed to do. Some games will introduce a new character at each new difficulty level, so the player will keep playing in order to see all of the different characters.

In the game Return of the Phoenix, Listing C-23 in Appendix C, each new level of difficulty has a new speed for the characters and a different rate of fire. The level of difficulty is decided by the players' skill. If he is playing well, the level of play will increase. When the player starts doing poorly, the level of play will be reduced. This self adjusting difficulty (SAD) system keeps the game interesting by constantly adjusting the level of game play to the player.

## SCORING

A player can tell how well he is doing by looking at the score. Almost all games should have a score of one sort or another. A player should be awarded points for actions that help him toward the goal of the game. The number of points awarded can vary greatly for different types of actions and can even be based on the current level of difficulty. In most cases, you will want to display the score constantly so that the player can tell how he is doing.

Usually, when an action results in points being added to the score, some appropriate sound effect is generated. This sound serves to inform the player that he has done something good without forcing him to look at the score. Under some circumstances, a bonus should be awarded for completing a specific task. Bonuses are often used to tempt a player into a course of action, for some immediate points, which is not necessarily beneficial in the long run.

Deciding on the number of points to award for the various actions is very subjective. You want a normal score to be high enough to make the player feel that he has accomplished something, but not so high as to be incomprehensible. You will probably not decide on your final scoring method on your first try, but will refine it during the testing process.

Note: When you first decide on a scoring method, be sure to reserve enough bytes for the score to accommodate an extremely high scoring player. This will avoid the rollover when the score changes from all 9s to all Os. Never assume that just because you can't get over a certain score that no one can.

As an added feature, you may wish to include a feature that allows the highest scoring players to place their names on a scoreboard. Although not necessary, this feature can add to the competition between a number of players.

The next chapter describes in detail the operation of the BOGHOP game, which is in Appendix C. This game has been designed to demonstrate virtually all the programming techniques described in this book.

If you have never programmed in assembly language, you may wish to try assembling the program as an exercise. Before doing so, you should make a copy of the source code disk and only work with the copy. The program has been structured in such a way that small changes can completely change game play. By reading the comments in the program listing, you will be able to see where you can make changes. This provides you with an easy way to start experimenting with an assembly language program.

This concludes the introduction to arcade game programming on the Commodore 64. Just as in any other field, the best way to learn is by doing. In this book, you have been given a strong foundation on which to build your program. By using the various definition files and libraries from Appendix C, you can spend more time writing your game program and less time coding the groundwork. I hope that you find a great deal of enjoyment in writing games and sharing them with others, as we have.